ME964
High Performance Computing for Engineering Applications

Spring 2011

Dan Negrut
Assistant Professor
Department of Mechanical Engineering
University of Wisconsin, Madison
January 18, 2011

“I think there is a world market for maybe five computers.”
T. J. Watson, chairman of IBM, 1943.
Today’s Game Plan

- Course logistics
- Brief overview of syllabus
- Motivation and central themes of this class
- Start quick overview of C programming language
Instructor: Dan Negrut

- Polytechnic Institute of Bucharest, Romania
  - B.S. – Aerospace Engineering (1992)

- The University of Iowa
  - Ph.D. – Mechanical Engineering (1998)

- MSC.Software
  - Product Development Engineer 1998-2005

- The University of Michigan
  - Adjunct Assistant Professor, Dept. of Mathematics (2004)

- Division of Mathematics and Computer Science, Argonne National Laboratory

- The University of Wisconsin-Madison, Joined in Nov. 2005
  - Research Focus: Computational Dynamics (Dynamics of Multi-body Systems)
  - Established the Simulation-Based Engineering Lab (http://sbel.wisc.edu)
Good to know…

- **Time**: 9:30 Tu & Th
- **Location**: 1163ME
- **Office**: 2035ME
- **Phone**: 608 890-0914
- **E-Mail**: negrut@engr.wisc.edu
- **Course Webpage**: [http://sbel.wisc.edu/Courses/ME964/2011/index.htm](http://sbel.wisc.edu/Courses/ME964/2011/index.htm)
- **Grades reported at**: [learnuw.wisc.edu](http://learnuw.wisc.edu)
ME 964 Spring 2011

- **Office Hours:**
  - Monday 2 – 4 PM
  - Wednesday 2 – 4 PM

- Call or email to arrange for meetings outside office hours

- Walk-ins are fine as long as they are in the afternoon

- **TAs:**
  - Arman Pazouki
  - Toby Heyn
No textbook is required, but there are some recommended ones:

- **Highly recommended, useful in this class**
- NVIDIA CUDA C Programming Guide V3.2, 2010:
- Jason Sanders and Edward Kandrot: CUDA by Example: An Introduction to General-Purpose GPU Programming, Addison-Wesley Professional, 2010 (on reserve, Wendt Lib.)
- Peter Pacheco: Parallel Programming with MPI, Morgan Kaufmann, 1996 (on reserve, Wendt Lib.)
- B. Kernighan and D. Ritchie, The C Programming Language
- B. Stroustrup, The C++ Programming Language, Third Edition

- **Further reading**
- Michael J. Quinn: Parallel Programming in C with MPI and OpenMP, McGraw Hill, 2003
Course Related Information

- Handouts will be printed out and provided before each lecture

- Lecture slides (PPT and PDF) will be made available online at class website

- Video streaming of class anticipated to be available on the same day at
  - [http://mediasite.engr.wisc.edu/Mediasite/Catalog/pages/catalog.aspx?catalogId=31c0b7c4-3a0f-410b-bacf-0c238380112f&folderId=96ee9eab-32a4-4321-8b45-6eae85c267ef&rootDynamicFolderId=e5b4a945-c68f-45b2-9eb7-b2512f5122cd](http://mediasite.engr.wisc.edu/Mediasite/Catalog/pages/catalog.aspx?catalogId=31c0b7c4-3a0f-410b-bacf-0c238380112f&folderId=96ee9eab-32a4-4321-8b45-6eae85c267ef&rootDynamicFolderId=e5b4a945-c68f-45b2-9eb7-b2512f5122cd)

- Grades will be maintained online at Learn@UW

- Syllabus will be updated as we go
  - It will contain info about
    - Topics we cover
    - Homework assignments
  - Available at the course website
Grading

- Homework 40%
- Midterm Exam 10%
- Midterm Project 20%
- Final Project 25%
- Course Participation 5%

- Total 100%

NOTE:
- Score related questions (homeworks/exams/labs) must be raised prior to next class after the homeworks/exams/lab is returned.
Homework Policies

- About eight or nine HWs assigned
  - No late HW accepted
    - HW due at 11:59 PM on the day indicated as due date

- Homework with lowest score will be dropped when computing final score

- Homework and projects should be emailed to me964uw2011@gmail.com
  - To get credit for your work the email time-stamp should be prior to the assignment due time/date
Midterm Exam

- One midterm exam
- Scheduled during regular class hours
- Tentatively scheduled on April 21
- Doesn’t require use of a computer (it’s a pen and paper exam)
- It’s a “closed books” exam
  - You can bring annotated copies of the papers that you are asked to read
Midterm Project

- Each one of you will have to select one of four topics by March 1
  - Topic 1: Simplified N-Body problem on the GPU
  - Topic 2: Collision detection on the GPU
  - Topic 3: Finite Element Analysis on the GPU
  - Topic 4: GPU-based parallel solution of sparse large positive definite linear system using Cholesky decomposition

- Topics listed according to their level of difficulty
  - Topics 2, 3 conference paper worth if implemented right
  - Topic 4 journal paper worth if implemented right

- Due on April 13 at 11:59 PM
- Accounts for 20% of final grade
- Project is individual
Final Exam Project

- Scheduled for Tuesday, May 10, 12:25 PM

- The Final Project is due on May 9, at 11:59 PM

- Two hour time slot used to have Final Project presentations

- Additional presentation time slots will very likely be needed during finals’ week

- I will come up with a way for you to select your time slot based on your availability during the finals’ week
Final Exam Project

- Final Project (accounts for 25% of final grade):
  - It is an individual project
  - You choose a problem that suites your research or interests
  - You are encouraged to tackle a meaningful problem
    - Attempt to solve a useful problem rather than a problem that you are confident that you can solve
    - Projects that are not successful are ok, provided you aim high enough and demonstrate good work
    - Continuing the Midterm Project is ok for Topics 2, 3, and 4
  - Tentatively,
    - Work on Final Project will start on April 15
    - Presentation of topic tentatively scheduled for April 7 and 12
Class Participation

- Accounts for 5% of final grade. To earn the 5%, you must:
  - Contribute at least five meaningful posts on the class Forum
    - Forum is live at: http://sbel.wisc.edu/Forum/index.php?board=3.0
    - Forum meant to serve as a quick way to answer some of your questions by instructor and other ME964 colleagues
    - Your ME964 Forum account is already set up, you should have got an email with login info
# Scores and Grades

<table>
<thead>
<tr>
<th>Score</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>92-100</td>
<td>A</td>
</tr>
<tr>
<td>86-91</td>
<td>AB</td>
</tr>
<tr>
<td>78-85</td>
<td>B</td>
</tr>
<tr>
<td>70-77</td>
<td>BC</td>
</tr>
<tr>
<td>60-69</td>
<td>C</td>
</tr>
<tr>
<td>50-59</td>
<td>D</td>
</tr>
</tbody>
</table>

- Grading will **not** be done on a curve
- Final score will be rounded to the nearest integer prior to having a letter assigned
  - Example:
    - 85.59 becomes AB
    - 85.27 becomes B
Rules of Engagement

- You are encouraged to discuss assignments with other class students
  - Post and read posts on Forum

- Getting **verbal** advice and suggestions from anybody is fine

- Any copying of non-trivial code is not acceptable
  - Non-trivial = more than a line or so
  - Includes reading someone else’s code and then going off to write your own

- Use of third party libraries that directly implement the solution of a HW/Project is not acceptable
Rules of Engagement

- Breaking the rules:
  - Zero points on HW/Exam/Project at first occurrence
  - Automatic F final grade upon second occurrence

- These rules are vague and not meant to police you

- I count on your honesty more than anything else
A Word on Hardware…

- The course is designed to leverage Newton, a cluster with 48 CPU cores and 24 GPU cards
  - CPUs: Intel Xeon 5520, a quadcore chip
  - GPUs: NVIDIA TESLA C1060
    - 240 Scalar Processors each
    - 4 GB global memory on the device

- Each student receives an individual account on Newton to be used for
  - GPU computing
  - MPI-enabled parallel computing
  - OpenMP multi-core computing

- A second 64 CPU core and 32 GPU card cluster available by end of February (we’ll call this machine Euler)

- Advice: if possible, do all the programming on a local machine with Developer Studio and CUDA. Move to the cluster for “production” runs
A Word on Software…

- Newton managed with Windows HPC Server 2008 R2

- We will be using Microsoft Developer Studio 2008
  - Already available on Newton
  - You can download for free and install Developer Studio thanks to UW-Madison agreement with Microsoft

- We will be using for GPU computing NVIDIA’s CUDA 3.2

- If Euler becomes available, it will run Linux but Linux will not be supported due to limited TA/instructor bandwidth
Staying in Touch…

- Please do not email me unless you have a personal problem
  - Examples:
    - Good: Schedule a meeting outside office hours
    - Bad: Asking me clarifications on Problem 2 of the current assignment (this needs to be on the Forum)
    - Bad: telling me that you can’t compile your code (this should also go to the Forum)

- Any course-related question should be posted on the Forum
  - I continuously monitor the Forum
  - If you can answer a Forum post, please do so (counts towards your 5% class participation and helps me as well)
  - Keeps all of us on the same page
Course Objectives

- Introduce student to existing High-Performance Computing (HPC) software and hardware
  - Usually “high-performance” refers to parallel architectures or vector machines; i.e., architectures that have the potential to run much faster compared to your desktop computer

- Help you recognize and appreciate the fact that there are numerous applications/problems that can be solved in a parallel fashion

- Help you gain basic skills that will help you map this applications onto a parallel computing hardware/software stack

- Present basic software design patterns for parallel computing
Course Objectives

- What I’ll try to accomplish
  - Provide enough information for you to start writing software that can leverage parallel computing to hopefully reduce the amount of time required by your simulations to complete
  - Emphasis is on GPU computing

- What I will not attempt to do
  - Investigate how to design new parallel computing languages or language features, compilers, how new hardware should be designed, etc.

- To summarize,
  - I’m a Mechanical Engineer, a consumer of parallel computing
  - I’m not a producer of parallel computing
There are multiple choices when it comes to implementing parallelism
- Pthreads, Intel’s TBB, OpenMP, MPI, Ct, Cilk, CUDA, Etc.

Emphasis will be on HPC on the Graphics Processing Unit (GPU)
- GPU computing typically associated with fine grain parallelism
- Three lectures will be dedicated to the Message Passing Interface (MPI) HPC model, which is aimed at coarse grain parallelism
- One lecture dedicated to OpenMP

Why emphasize GPU Computing?
- There are more than 60 million computers in use today that have a CUDA enabled GPU card
- GPU computing proved to deliver significant speedups at very affordable prices
GPU Proved Fast in Several Applications

- **Medical Imaging**
  - U of Utah
  - 146X

- **Molecular Dynamics**
  - U of Illinois, Urbana
  - 36X

- **Video Transcoding**
  - Elemental Tech
  - 18X

- **Matlab Computing**
  - AccelerEyes
  - 50X

- **Astrophysics**
  - RIKEN
  - 100X

- **Financial Simulation**
  - Oxford
  - 149X

- **Linear Algebra**
  - Universidad Jaime
  - 47X

- **3D Ultrasound**
  - Techniscan
  - 20X

- **Quantum Chemistry**
  - U of Illinois, Urbana
  - 130X

- **Gene Sequencing**
  - U of Maryland
  - 30X
Who Will Be the ME964 Student?

- Hard to pinpoint the typical student
- 45 students enrolled coming from 14 UW departments
  - Astronomy (2), Biomedical Engineering (1), Chemical Engineering (2), Chemistry (1), Civil and Environmental Engineering (2), Computer Science (3), Electrical Engineering (8), Engineering Mechanics (1), Business Management and Human Resources (1), Materials Science (1), Mechanical Engineering (13), Medical Physics (4), Nuclear Engineering and Engineering Physics (2), Physics (4)

- Title says “High Performance Computing for Engineering Applications”
- Typical student is from the College of Engineering
- I did not advertise the class with the CS department since the material would probably be boring
High Performance Computing for Engineering Applications

Why This Title?

- Computer Science: ISA, Limits to Instruction Level Parallelism and Multithreading, Pipelining, Memory Hierarchy, Memory Transactions, Cache Coherence, etc.
  - Long story short: how should a processor be built?

- Electrical Engineering: how will we build the processor that the CS colleagues have in mind?

- This class: how to use the system built by electrical engineers who implemented the architecture devised by the CS colleagues
  - At the end of the day, in our research we’ll be dealing with one of the seven dwarfs…
Phillip Colella’s “Seven Dwarfs”

High-end simulation in the physical sciences = 7 numerical methods:

1. Structured Grids (including locally structured grids, e.g. Adaptive Mesh Refinement)
2. Unstructured Grids
3. Fast Fourier Transform
4. Dense Linear Algebra
5. Sparse Linear Algebra
6. Particles
7. Monte Carlo

- If add 4 more for embedded, covers all 41 EEMBC benchmarks
  8. Search/Sort
  9. Filter
  10. Combinational logic
  11. Finite State Machine

- Note: Data sizes (8 bit to 32 bit) and types (integer, character) differ, but algorithms the same

---

Overview of Material Covered

- Quick C Intro
- General considerations vis-à-vis trends in the chip industry
- Overview of parallel computation paradigms and supporting hardware/software
- GPU computing and the CUDA programming model
- Brief intro to MPI and OpenMP programming
- Midterm/Final Project related discussions
- Two or three tentative guest lectures
Overview of the GPU (CUDA) component...

- GPU Computing and CUDA Intro
- CUDA Memory Model
- CUDA Hardware
- GPU Compute Core
- Bank Conflicts
- Control Flow in CUDA
- Parallel Programming - Application Performance
- Parallel Programming - Algorithm Styles
Prerequisites

- This is a high-level graduate class in a very fluid topic
- Familiarity with C is expected

- Good programming skills are necessary
  - Understanding pointers
  - Being able to wrestle with a compile error on your own
  - Having used a debugger
  - Having used a profiler
At the beginning of the road…

- Teaching the class for the second time
  - There will be rough edges
  - There might be questions that I don’t have an answer for
    - I promise I’ll follow up on these and get back with you (on the Forum)

- Please ask questions (be curious)
My Advice to You

- If you can, do something remarkable, innovate, amaze everybody...
Acknowledgements

- Students helping with this class
  - Andrew Seidl
  - Arman Pazouki
  - Toby Heyn
  - Naresh Khude

- College of Engineering - financial support for video recording

- Department of ME – support of one TA

- Microsoft – financial support to develop the course material

- NVIDIA – financial support to build Newton and Euler
End: Discussion of Syllabus

Beginning: Quick Review of C
ME964
High Performance Computing for Engineering Applications

Quick Overview of C Programming
January 20, 2011

“There is no reason for any individual to have a computer in their home.”
Before We Get Started…

- Last time
  - Course logistics & syllabus overview
  - Discussed Midterm Projects
    - Discrete Element Method on the GPU
    - Collision Detection on the GPU
    - Basic Finite Element Analysis on the GPU
    - Sparse Linear Solver on the GPU

- Today
  - Quick overview of C Programming
    - Essential read: Chapter 5 of “The C Programming Language” (Kernighan and Ritchie)
    - Acknowledgement: Slides on this C Intro include material due to Donghui Zhang and Lewis Girod

- Correction:
  - Email your homework a this address: me964uw@gmail.com
Auditing the Course

● Why auditing?
  ● Large participation justifies another offering of this course
  ● Augments your experience with this class
    ● You can get an account on the GPU cluster
    ● You will be added to the email list
    ● Can post questions on the forum

● How to register for auditing:
  ● In order to audit a course, a student must first enroll in the course as usual. Then the student must request to audit the course online. (There is a tutorial available through the Office of the Registrar.) Finally, the student must save & print the form. Once they have obtained the necessary signatures, the form should be turned in to the Academic Dean in the Grad School at 217 Bascom. The Grad School offers more information on Auditing Courses in their Academic Policies and Procedures.

Tutorial website: [http://www.registrar.wisc.edu/isis_helpdocs/enrollment_demos/V90CourseChangeRequest/V90CourseChangeRequest.htm](http://www.registrar.wisc.edu/isis_helpdocs/enrollment_demos/V90CourseChangeRequest/V90CourseChangeRequest.htm)

Auditing Courses: [http://www.grad.wisc.edu/education/acadpolicy/guidelines.html#13](http://www.grad.wisc.edu/education/acadpolicy/guidelines.html#13)
C Syntax and Hello World

#include <stdio.h>  
/* The simplest C Program */  
int main(int argc, char **argv)  
{  
    printf("Hello World\n");  
    return 0;  
}  

#include inserts another file. “.h” files are called “header” files. They contain declarations/definitions needed to interface to libraries and code in other “.c” files.

A comment, ignored by the compiler

The main() function is always where your program starts running.

Blocks of code (“lexical scopes”) are marked by { … }

What do the <> mean?

Return ‘0’ from this function
Lexical Scoping

Every Variable is Defined within some scope. A Variable cannot be referenced by name (a.k.a. Symbol) from outside of that scope.

Lexical scopes are defined with curly braces { }.

The scope of Function Arguments is the complete body of that function.

The scope of Variables defined inside a function starts at the definition and ends at the closing brace of the containing block.

The scope of Variables defined outside a function starts at the definition and ends at the end of the file. Called “Global” Vars.
### Comparison and Mathematical Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>less than or equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>greater than or equal</td>
</tr>
<tr>
<td>!=</td>
<td>not equal</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>logical and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>logical not</td>
</tr>
<tr>
<td>+</td>
<td>plus</td>
</tr>
<tr>
<td>-</td>
<td>minus</td>
</tr>
<tr>
<td>*</td>
<td>mult</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
</tr>
<tr>
<td>%</td>
<td>modulo</td>
</tr>
<tr>
<td>&amp;</td>
<td>bitwise and</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>^</td>
<td>bitwise xor</td>
</tr>
<tr>
<td>~</td>
<td>bitwise not</td>
</tr>
<tr>
<td>&lt;&lt;</td>
<td>shift left</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>shift right</td>
</tr>
</tbody>
</table>

#### Beware division:
- $5 / 10 \rightarrow 0$ whereas $5 / 10.0 \rightarrow 0.5$
- Division by 0 will cause a FPE

#### Don’t confuse & and &&:
- $1 \& 2 \rightarrow 0$ whereas $1 \&\& 2 \rightarrow <true>$

#### The rules of precedence are clearly defined but often difficult to remember or non-intuitive. When in doubt, add parentheses to make it explicit.
Assignment Operators

\[
\begin{array}{l}
x = y \quad \text{assign } y \text{ to } x \\
x++ \quad \text{post-increment } x \\
++x \quad \text{pre-increment } x \\
x-- \quad \text{post-decrement } x \\
--x \quad \text{pre-decrement } x \\
\end{array}
\]

\[
\begin{array}{l}
x += y \quad \text{assign } (x+y) \text{ to } x \\
x -= y \quad \text{assign } (x-y) \text{ to } x \\
x *= y \quad \text{assign } (x*y) \text{ to } x \\
x /= y \quad \text{assign } (x/y) \text{ to } x \\
x %= y \quad \text{assign } (x\%y) \text{ to } x \\
\end{array}
\]

Note the difference between ++x and x++ (high vs low priority (precedence)):

\[
\begin{array}{l}
\text{int } x=5; \\
\text{int } y; \\
y = ++x; \\
/* x == 6, y == 6 */
\end{array}
\]

\[
\begin{array}{l}
\text{int } x=5; \\
\text{int } y; \\
y = x++; \\
/* x == 6, y == 5 */
\end{array}
\]

Don’t confuse “=“ and “==“!

\[
\begin{array}{l}
\text{int } x=5; \\
\text{if } (x==6) \quad /* \text{false } */ \\
\quad \{ \\
\quad \quad /* \ldots */ \\
\quad \} \\
/* x \text{ is still 5 } */
\end{array}
\]

\[
\begin{array}{l}
\text{int } x=5; \\
\text{if } (x=6) \quad /* \text{always true } */ \\
\quad \{ \\
\quad \quad /* x \text{ is now 6 } */ \\
\quad \} \\
/* \ldots */
\end{array}
\]
A Quick Digression About the Compiler

Compilation occurs in two steps: “Preprocessing” and “Compiling”

In Preprocessing, source code is “expanded” into a larger form that is simpler for the compiler to understand. Any line that starts with ‘#’ is a line that is interpreted by the Preprocessor.

- Include files are “pasted in” (#include)
- Macros are “expanded” (#define)
- Comments are stripped out ( /* */ , // )
- Continued lines are joined ( \ )

The compiler then converts the resulting text (called translation unit) into binary code the CPU can execute.
C Memory Pointers

- To discuss memory pointers, we need to talk a bit about the concept of memory

- We’ll conclude by touching on a couple of other C elements:
  - Arrays, typedef, and structs
The “memory”

Memory: similar to a big table of numbered slots where bytes of data are stored.

The number of a slot is its **Address**. One byte **Value** can be stored in each slot.

Some data values span more than one slot, like the character string “Hello\n”

A **Type** provides a logical meaning to a span of memory. Some simple types are:

<table>
<thead>
<tr>
<th>char</th>
<th>char [10]</th>
<th>int</th>
<th>float</th>
<th>int64_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>a single character (1 slot)</td>
<td>an array of 10 characters</td>
<td>signed 4 byte integer</td>
<td>4 byte floating point</td>
<td>signed 8 byte integer</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>‘H’ (72)</td>
</tr>
<tr>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>6</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>7</td>
<td>‘l’ (108)</td>
</tr>
<tr>
<td>8</td>
<td>‘o’ (111)</td>
</tr>
<tr>
<td>9</td>
<td>‘\n’ (10)</td>
</tr>
<tr>
<td>10</td>
<td>‘\0’ (0)</td>
</tr>
<tr>
<td>11</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>
What is a Variable?

A Variable names a place in memory where you store a Value of a certain Type.

You first Declare a variable by giving it a name and specifying its type and optionally an initial value.

```
char x;
char y = 'e';
```

Variable x declared but undefined.

The compiler puts x and y somewhere in memory.

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4</td>
<td>Some garbage</td>
</tr>
<tr>
<td></td>
<td>5</td>
<td>'e' (101)</td>
</tr>
<tr>
<td></td>
<td>6</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10</td>
<td></td>
</tr>
<tr>
<td></td>
<td>11</td>
<td></td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

Initial value

Name

What names are legal?

Type is single character (char)

extern? static? const?
Multi-byte Variables

Different types require different amounts of memory. Most architectures store data on “word boundaries”, or even multiples of the size of a primitive data type (int, char)

```
char x;
char y='e';
int z = 0x01020304;
```

0x means the constant is written in hex

An int requires 4 bytes

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Addr</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>x</td>
<td>4</td>
<td>Some garbage</td>
</tr>
<tr>
<td>y</td>
<td>5</td>
<td>‘e’ (101)</td>
</tr>
<tr>
<td>z</td>
<td>8</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>11</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>12</td>
<td></td>
</tr>
</tbody>
</table>

padding
Memory, a more detailed view...

- A sequential list of words, starting from 0.

- On 32bit architectures (e.g. Win32): each word is 4 bytes.

- Local variables are stored in the stack

- Dynamically allocated memory is set aside on the heap (more on this later…)

- For multiple-byte variables, the address is that of the smallest byte (little endian).
Example...

<table>
<thead>
<tr>
<th></th>
<th>+0</th>
<th>+1</th>
<th>+2</th>
<th>+3</th>
</tr>
</thead>
<tbody>
<tr>
<td>V1</td>
<td>944</td>
<td>940</td>
<td>936</td>
<td>932</td>
</tr>
<tr>
<td>V2</td>
<td>924</td>
<td>928</td>
<td>932</td>
<td>936</td>
</tr>
<tr>
<td>V3</td>
<td>916</td>
<td>912</td>
<td>908</td>
<td>904</td>
</tr>
<tr>
<td>V4</td>
<td>900</td>
<td>908</td>
<td>904</td>
<td>900</td>
</tr>
</tbody>
</table>
Another Example

#include <iostream>

int main() {
    char c[10];
    int d[10];
    int* darr;

    darr = (int *)malloc(10*sizeof(int));
    size_t sizeC = sizeof(c);
    size_t sizeD = sizeof(d);
    size_t sizeDarr = sizeof(darr);

    free(darr);
    return 0;
}

What is the value of:
- sizeC
- sizeD
- sizeDarr

NOTE: C is a compile-time operator that returns the size, in multiples of the size of char, of the variable or parenthesized type-specifier that it precedes.
Can a C function modify its arguments?

What if we wanted to implement a function `pow_assign()` that modified its argument, so that these are equivalent:

```c
float p = 2.0;
/* p is 2.0 here */
p = pow(p, 5);
/* p is 32.0 here */
```

```c
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}
```

Would this work?

Native function, to use you need `#include <math.h>`
In C you can’t change the value of any variable passed as an argument in a function call…

```c
void pow_assign(float x, uint exp)
{
    float result=1.0;
    int i;
    for (i=0; (i < exp); i++) {
        result = result * x;
    }
    x = result;
}
```

// a code snippet that uses above // function
{
    float p=2.0;
    pow_assign(p, 5);
    // the value of p is 2 here...
}

In C, all arguments are passed by value

Keep in mind: pass by value requires the variable to be copied. That copy is then passed to the function. Sometime generating a copy can be expensive…

But, what if the argument is the *address* of a variable?
C Pointers

- What is a pointer?
  - A variable that contains the memory address of another variable or of a function

- In general, it is safe to assume that on 32 bit architectures pointers occupy one word
  - Pointers to int, char, float, void, etc. ("int*", "char*", "*float", "void*"), they all occupy 4 bytes (one word).

- Pointers: *very* many bugs in C programs are traced back to mishandling of pointers…
Pointers (cont.)

- The need for pointers
  - Needed when you want to modify a variable (its value) inside a function
    - The pointer is passed to that function as an argument
  - Passing large objects to functions without the overhead of copying them first
  - Accessing memory allocated on the heap
  - Referring to functions
A **Valid** pointer is one that points to memory that your program controls. Using invalid pointers will cause non-deterministic behavior

- Very often the code will crash with a SEGV, that is, Segment Violation, or Segmentation Fault.

There are two general causes for these errors:

- Coding errors that end up setting the pointer to a strange number
- Use of a pointer that was at one time valid, but later became invalid

Good practice:

- Initialize pointers to 0 (or NULL). NULL is never a valid pointer value, but it is known to be invalid and means “no pointer set”.

```c
char * get_pointer()
{
    char x=0;
    return &x;
}

char * ptr = get_pointer();
*ptr = 12; /* valid? */
```

Will `ptr` be valid or invalid?
A pointer to a variable allocated on the stack becomes invalid when that variable goes out of scope and the stack frame is “popped”. The pointer will point to an area of the memory that may later get reused and rewritten.

```c
char * get_pointer()
{
    char x=0;
    return &x;
}

int main()
{
    char * ptr = get_pointer();
    *ptr = 12; /* valid? */
    other_function();
    return 0;
}
```

But now, `ptr` points to a location that’s no longer in use, and will be reused the next time a function is called!

Here is what I get in DevStudio when compiling:
main.cpp(6) : warning C4172: returning address of local variable or temporary
Example: What gets printed out?

```c
int main() {
    int d;
    char c;
    short s;
    int* p;
    int arr[2];
    printf( "%p, %p, %p, %p, %p
", &d, &c, &s, &p, arr );
    return 0;
}
```

• NOTE: Here &d = 920 (in practice a 4-byte hex number such as 0x22FC3A08)

Q: What does get printed out by the `printf` call in the code snippet above?
Example:
Usage of Pointers & Pointer Arithmetic

```c
int main() {
    int d;
    char c;
    short s;
    int* p;
    int arr[2];
    p = &d;
    *p = 10;
    c = (char)1;
    p = arr;
    *(p+1) = 5;
    p[0] = d;
    *( (char*)p + 1 ) = c;
    return 0;
}
```

Q: What are the values stored in `arr`? [assume little endian architecture]
Example [Cntd.]

```c
p = &d;
*p = 10;
c = (char)1;
p = arr;
*(p+1) = 5; // int* p;
p[0] = d;
*( (char*)p + 1 ) = c;
```

**Question:** `arr[0] = ?`
Use of pointers, another example...

- Pass pointer parameters into function

```c
void swap(int *px, int *py)
{
    int temp;
    temp = *px;
    *px = *py;
    *py = temp;
}
int a = 5;
int b = 6;
swap(&a, &b);
```

- What will happen here?

```c
int * a;
int * b;
swap(a, b);
```
Dynamic Memory Allocation (Allocation on the Heap)

- Allows the program to determine how much memory it needs at run time and to allocate exactly the right amount of storage.
  - It is your responsibility to clean up after you (free the dynamic memory you allocated)

- The region of memory where dynamic allocation and deallocation of memory can take place is called the heap.
ME964
High Performance Computing for Engineering Applications

Wrapping up Overview of C Programming
Starting Overview of Parallel Computing
January 25, 2011

"I have traveled the length and breadth of this country and talked with the best people, and I can assure you that data processing is a fad that won't last out the year."
The editor in charge of business books for Prentice Hall, 1957.
Before We Get Started…

- Last time
  - Quick overview of C Programming
    - Essential reading: Chapter 5 of “The C Programming Language” (Kernighan and Ritchie)
  - Mistakes (two) in the slides addressed in Forum posting

- Today
  - Wrap up overview of C programming
  - Start overview of parallel computing
    - Why and why now

- Assignment, due on Feb 1, 11:59 PM:
  - Posted on the class website
  - Related to C programming
  - Reading: chapter 5 of “The C Programming Language” (Kernighan and Ritchie)
  - Consult the on-line syllabus for all the details
Recall that variables are allocated **statically** by having declared with a given size. This allocates them in the stack.

Allocating memory at run-time requires **dynamic** allocation. This allocates them on the heap.

```c
int * alloc_ints(size_t requested_count)
{
    int * big_array;
    big_array = (int *)calloc(requested_count, sizeof(int));
    if (big_array == NULL) {
        printf("can't allocate %d ints: %m\n", requested_count);
        return NULL;
    }

    /* big_array[0] through big_array[requested_count-1] are valid and zeroed. */
    return big_array;
}
```

**calloc()** allocates memory for N elements of size k

**sizeof()** reports the size of a type in bytes

**Returns NULL if can’t alloc**

It’s OK to return this pointer. It will remain valid until it is freed with **free()**. However, it’s a bad practice to return it (if you need is somewhere else, declare and define it there...).
Caveats with Dynamic Memory

Dynamic memory is useful. But it has several caveats:

Whereas the stack is automatically reclaimed, dynamic allocations must be tracked and free()’d when they are no longer needed. With every allocation, be sure to plan how that memory will get freed. Losing track of memory causes “memory leak”.

Whereas the compiler enforces that reclaimed stack space can no longer be reached, it is easy to accidentally keep a pointer to dynamic memory that was freed. Whenever you free memory you must be certain that you will not try to use it again.

Because dynamic memory always uses pointers, there is generally no way for the compiler to statically verify usage of dynamic memory. This means that errors that are detectable with static allocation are not with dynamic
Moving on to other topics… What comes next:

- Creating logical layouts of different types (structs)
- Creating new types using typedef
- Using arrays
- Parsing C type names
Data Structures

- A data structure is a collection of one or more variables, possibly of different types.

- An example of student record

```c
struct StudRecord {
    char name[50];
    int id;
    int age;
    int major;
};
```
A data structure is also a data type

```c
struct StudRecord my_record;
struct StudRecord * pointer;
pointer = & my_record;
```

Accessing a field inside a data structure

```c
my_record.id = 10;
// or
pointer->id = 10;
```
Allocating a data structure instance

This is a new type now

```
struct StudRecord* pStudentRecord;
pStudentRecord = (StudRecord*)malloc(sizeof(struct StudRecord));
pStudentRecord ->id = 10;
```

**IMPORTANT:** Never calculate the size of a data structure yourself. Rely on the sizeof() function

- Example: Because of memory padding, the size of “struct StudRecord” is 64 (instead of 62 as one might estimate)
The “typedef” Construct

```c
struct StudRecord {
    char name[50];
    int id;
    int age;
    int major;
};

typedef struct StudRecord RECORD;

int main() {
    RECORD   my_record;
    strcpy_s(my_record.name, "Joe Doe");
    my_record.age = 20;
    my_record.id = 6114;

    RECORD* p = &my_record;
    p->major = 643;
    return 0;
}
```

Using typedef to improve readability…
Arrays

Arrays in C are composed of a particular type, laid out in memory in a repeating pattern. Array elements are accessed by stepping forward in memory from the base of the array by a multiple of the element size.

```c
/* define an array of 10 chars */
char x[5] = {'t','e','s','t','\0'};

/* access element 0, change its value */
x[0] = 'T';

/* pointer arithmetic to get elt 3 */
char elt3 = *(x+3); /* x[3] */

/* x[0] evaluates to the first element; * x evaluates to the address of the * first element, or &(x[0]) */

/* 0-indexed for loop idiom */
#define COUNT 10
char y[COUNT];
int i;
for (i=0; i<COUNT; i++) {
    /* process y[i] */
    printf("%c\n", y[i]);
}
```

Brackets specify the count of elements. Initial values optionally set in braces.

Arrays in C are 0-indexed (here, 0…4)

Q: What’s the difference between “char x[5]” and a declaration like “char *x”?
At this point we have seen a few basic types, arrays, pointer types, and structures. So far we’ve glossed over how types are named.

C type names are parsed by starting at the type name and working outwards according to the rules of precedence:

REMEMBER THIS: (), which stands for function, and [], which stands for array, have higher precedence than *, which stands for pointer.
Another less obvious construct is the “pointer to function” type. For example, qsort: (a sort function in the standard library)

```c
void qsort(void *base, size_t nmemb, size_t size, int (*compar)(const void *, const void *));

/* function matching this type: */
int cmp_function(const void *x, const void *y);

/* typedef defining this type: */
typedef int (*cmp_type)(const void *, const void *);

/* rewrite qsort prototype using our typedef */
void qsort(void *base, size_t nmemb, size_t size, cmp_type compar);
```

- `void *` is a pointer to memory of unknown type.
- `size_t` is an unsigned int.
- `const void *` means the function is not allowed to modify memory via this pointer.
- The last argument is a comparison function.
sizeof() can take a variable reference in place of a type name. This guarantees the right allocation, but don’t accidentally allocate the sizeof() the pointer instead of the object!

```c
/* allocating a struct with malloc() */
struct my_struct *s = NULL;
s = (struct my_struct *)malloc(sizeof(*s)); /* NOT sizeof(s)!! */
if (s == NULL) {
    printf(stderr, "no memory!");
    exit(1);
}
memset(s, 0, sizeof(*s));

/* another way to initialize an alloc'd structure: */
struct my_struct init = {
    counter: 1,
    average: 2.5,
    in_use: 1
};

/* memmove(dst, src, size) (note, arg order like assignment) */
memmove(s, &init, sizeof(init));

/* when you are done with it, free it! */
free(s);
s = NULL;
```

malloc() allocates n bytes

Always check for NULL. Even if you just exit(1).

malloc() does not zero the memory, so you should memset() it to 0.

memmove is preferred because it is safe for shifting buffers

Use pointers as implied in-use flags!
High Level Question: Why is Software Hard?

- **Complexity**: Every conditional (“if”) doubles the number of paths through your code, every bit of state doubles possible states
  - Recommendation: reuse code with functions, avoid duplicate state variables

- **Mutability**: Software is easy to change.. Great for rapid fixes… And rapid breakage… Always one character away from a bug
  - Recommendation: tidy, readable code, easy to understand by inspection, provide *plenty* of meaningful comments.

- **Flexibility**: Problems can be solved in many different ways. Few hard constraints, easy to let your horses run wild
  - Recommendation: discipline and use of design patterns
Software Design Patterns

- A really good book if you are serious about programming
End: Quick Review of C

Beginning: Discussion of Hardware Trends
Sequential computing is arguably losing steam...

The next decade seems to belong to parallel computing

- Objectives of course segment:
  - Discuss some barriers facing the traditional sequential computation model
  - Discuss some solutions suggested by recent trends in hardware and software industries
  - Overview of hardware and software solutions in relation to parallel computing
Acknowledgements

- Presentation on this topic includes material due to
  - Hennessy and Patterson (Computer Architecture, 4th edition)
  - John Owens, UC-Davis
  - Darío Suárez, Universidad de Zaragoza
  - John Cavazos, University of Delaware
  - Others, as indicated on various slides
CPU Speed Evolution

[log scale]
...we can expect very little improvement in serial performance of general purpose CPUs. So if we are to continue to enjoy improvements in software capability at the rate we have become accustomed to, we must use parallel computing. This will have a profound effect on commercial software development including the languages, compilers, operating systems, and software development tools, which will in turn have an equally profound effect on computer and computational scientists.

John L. Manferdelli, Microsoft Corporation
Distinguished Engineer, leads the eXtreme Computing Group (XCG) System, Security and Quantum Computing Research Group
Three Walls to Serial Performance

- Memory Wall
- Instruction Level Parallelism (ILP) Wall
- Power Wall


http://www.ctwatch.org/quarterly/articles/2007/02/the-many-core-inflection-point-for-mass-market-computer-systems/
Memory Wall

- Memory Wall: What is it?
  - The growing disparity of speed between CPU and memory outside the CPU chip.

- The growing memory latency is a barrier to computer performance improvements
  - Current architectures have ever growing caches to improve the “average memory reference” time to fetch or write instructions or data

- All due to latency and limited communication bandwidth beyond chip boundaries.
  - From 1986 to 2000, CPU speed improved at an annual rate of 55% while memory access speed only improved at 10%.
Memory Bandwidths
[typical embedded, desktop and server computers]

© 2007 Elsevier, Inc. All rights reserved.
Memory Speed: Widening of the Processor-DRAM Performance Gap

- The processor: victim of its own success
  - So fast it left the memory behind
  - The CPU-Memory team can’t move as fast as you’d like (based on CPU top speeds) with a sluggish memory

- Plot on next slide shows on a *log* scale the increasing gap between CPU and memory

- The memory baseline: 64 KB DRAM in 1980

- Memory speed increasing at a rate of approx 1.07/year

- Processors improved
  - 1.25/year (1980-1986)
  - 1.52/year (1986-2004)
  - 1.20/year (2004-2010)
Memory Speed: Widening of the Processor-DRAM Performance Gap

![Graph showing the widening gap between processor and memory speeds over time.](Image)

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Memory Latency vs. Memory Bandwidth

- Latency: the amount of time it takes for an operation to complete
  - Measured in seconds
  - The utility “ping” in Linux measures the latency of a network
  - For memory transactions: send 32 bits to destination and back, measure how much time it takes → gives you latency

- Bandwidth: how much data can be transferred per second
  - You can talk about bandwidth for memory but also for a network (Ethernet, Infiniband, modem, DSL, etc.)

- Improving Latency and Bandwidth
  - The job of the friends in Electrical Engineering
  - Once in a while, our friends in Materials Science deliver a breakthrough
  - Promising technology: optic networks and layered memory on top of chip
Memory Latency vs. Memory Bandwidth

- Memory Access Latency is significantly more challenging to improve as opposed to improving Memory Bandwidth

- Improving Bandwidth: add more “pipes”. Relatively easy, not cheap though
  - Requires more pins that come out of the chip for DRAM, for instance. Tricky

- Improving Latency: not obvious what the solution is

- Analogy:
  - If you carry commuters with a train, add more cars to a train to increase bandwidth
  - Improving latency requires the construction of high speed trains
    - Very expensive
    - Requires qualitatively new technology
Latency vs. Bandwidth Improvements Over the Last 25 years

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Memory Wall, Conclusions

[IMPORTANT ME964 SLIDE]

- Memory trashing is what kills execution speed

- Many times you will see that when you run your application:
  - You are far away from reaching top speed of the chip
    AND
  - You are at top speed for your memory
    - If this is the case, you are trashing the memory
    - Means that basically you are doing one or both of the following
      - Move large amounts of data around
      - Move data often

<table>
<thead>
<tr>
<th>Memory Access Patterns</th>
</tr>
</thead>
<tbody>
<tr>
<td>To/From Registers</td>
</tr>
<tr>
<td>To/From Cache</td>
</tr>
<tr>
<td>To/From RAM</td>
</tr>
<tr>
<td>To/From Disk</td>
</tr>
</tbody>
</table>
Computer architecture – its three facets are as follows:

- Instruction set architecture (ISA) – the set of instructions that the processor can do
  - Examples: RISC, X86, ARM, etc.
  - The job of the friends in the Computer Science department

- Microarchitecture (organization) – cache levels, amount of cache at each level, etc.
  - The detailed low level organization of the chip that ensures that the ISA is implemented and performs according to specifications
  - Mostly CS but Electrical Engineering is relevant

- System design – how to connect things on a chip, buses, memory controllers, etc.
  - Mostly a job for our friends in the Electrical Engineering
Instruction Level Parallelism (ILP)

- ILP: a relevant factor in reducing execution times after 1985
- Idea: overlap the execution of independent instructions and improve performance
- Two approaches to discovering ILP
  - Dynamic: relies on hardware to help discover and exploit the parallelism dynamically at run time
    - It is the dominant one in the market
  - Static: relies on compiler to identify parallelism in the code and leverage it
- Examples where ILP expected to improve efficiency

```c
for( int=0; i<1000; i++)
  x[i] = x[i] + y[i];

1. e = a + b
2. f = c + d
3. g = e * f
```
The ILP Wall

- For ILP to make a dent, you need large blocks of instructions that can be [attempted to be] run in parallel

- Best examples: if-loops

- Duplicate hardware speculatively executes future instructions before the results of current instructions are known, while providing hardware safeguards to prevent the errors that might be caused by out of order execution

- Branches must be “guessed” to decide what instructions to execute simultaneously
  - If you guessed wrong, you throw away that part of the result

- Data dependencies may prevent successive instructions from executing in parallel, even if there are no branches.
The ILP Wall

- **ILP, the good:**
  - Existing programs enjoy performance benefits without any modification.
  - Recompiling them is beneficial but entirely up to you as long as you stick with the same ISA (for instance, if you go from Pentium 2 to Pentium 4 you don’t have to recompile your executable).

- **ILP, the bad:**
  - Improvements are difficult to forecast since the “speculation” success is difficult to predict.
  - Moreover, ILP causes a super-linear increase in execution unit complexity (and associated power consumption) without linear speedup.

- **ILP, the ugly:** serial performance acceleration using ILP has stalled because of these effects.
The Power Wall

- Power, and not manufacturing, limits traditional general purpose microarchitecture improvements (F. Pollack, Intel Fellow)

- Leakage power dissipation gets worse as gates get smaller, because gate dielectric thicknesses must proportionately decrease.

Adapted from F. Pollack (MICRO’99)
The Power Wall

- Power dissipation in clocked digital devices is proportional to the clock frequency and feature length imposing a natural limit on clock rates.

- Significant increase in clock speed without heroic (and expensive) cooling is not possible. Chips would simply melt.

- Clock speed increased by a factor of 4,000 in less than two decades
  - The ability of manufacturers to dissipate heat is limited though…
  - Look back at the last five years, the clock rates are pretty much flat.

- Problem might one day be addressed by a new Materials Science breakthrough
Trivia

- AMD Phenom II X4 955 (4 core load)
  - 236 Watts

- Intel Core i7 920 (8 thread load)
  - 213 Watts

- Human Brain
  - 20 W
  - Represents 2% of our mass
  - Burns 20% of all energy in the body at rest
Conventional Wisdom (CW) in Computer Architecture

- Old CW: Power is free, Transistors expensive
- New CW: “Power wall” Power expensive, Transistors free (Can put more on chip than can afford to turn on)

- Old: Multiplies are slow, Memory access is fast
- New: “Memory wall” Memory slow, multiplies fast (200-600 clocks to DRAM memory, 4 clocks for FP multiply)

- Old : Increasing Instruction Level Parallelism via compilers, innovation (Out-of-order, speculation, VLIW, …)
- New CW: “ILP wall” diminishing returns on more ILP

- New: Power Wall + Memory Wall + ILP Wall = Brick Wall
  - Old CW: Uniprocessor performance 2X / 1.5 yrs
  - New CW: Uniprocessor performance only 2X / 5 yrs?
Intel’s “Platform 2015” documentation, see http://download.intel.com/technology/computing/archinnov/platform2015/download/RMS.pdf

First of all, as chip geometries shrink and clock frequencies rise, the transistor leakage current increases, leading to excess power consumption and heat.

Secondly, the advantages of higher clock speeds are in part negated by memory latency, since memory access times have not been able to keep pace with increasing clock frequencies.

Third, for certain applications, traditional serial architectures are becoming less efficient as processors get faster further undercutting any gains that frequency increases might otherwise buy.
What can be done?
Moore’s Law

- 1965 paper: Doubling of the number of transistors on integrated circuits every two years
  - Moore himself wrote only about the density of components (or transistors) at minimum cost

- Increase in transistor count is also a rough measure of computer processing performance
  - Moore quote: “Moore's law has been the name given to everything that changes exponentially. I say, if Gore invented the Internet, I invented the exponential”

Moore’s Law (1965)

- “The complexity for minimum component costs has increased at a rate of roughly a factor of two per year (see graph on next page). Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least 10 years. That means by 1975, the number of components per integrated circuit for minimum cost will be 65,000. I believe that such a large circuit can be built on a single wafer.”

“Cramming more components onto integrated circuits” by Gordon E. Moore, Electronics, Volume 38, Number 8, April 19, 1965
The Ox vs. Chickens Analogy

Seymour Cray: "If you were plowing a field, which would you rather use: Two strong oxen or 1024 chickens?"

- Chicken is gaining momentum nowadays:
  - For certain classes of applications, you can run many cores at lower frequency and come ahead at the speed game

- Example (John Cavazos):
  - Scenario One: one-core processor w/ power budget W
    - Increase frequency by 20%
      - Substantially increases power, by more than 50%
      - But, only increase performance by 13%
  - Scenario Two: Decrease frequency by 20% with a simpler core
    - Decreases power by 50%
    - Can now add another dumb core (one more chicken…)

43
Intel’s Vision:
Evolutionary Configurable Architecture

Large, Scalar cores for high single-thread performance

Multi-core array
- CMP with ~10 cores

Scalar plus many core for highly threaded workloads

Many-core array
- CMP with 10s-100s low power cores
- Scalar cores
- Capable of TFLOPS+
- Full System-on-Chip
- Servers, workstations, embedded...

Dual core
- Symmetric multithreading

CMP = “chip multi-processor”
“Parallelism for Everyone”
Parallelism changes the game
- A large percentage of people who provide applications are going to have to care about parallelism in order to match the capabilities of their competitors.

Presentation Paul Petersen, Sr. Principal Engineer, Intel
Paul Otellini, President and CEO, Intel

"We are dedicating all of our future product development to multicore designs" 
"We believe this is a key inflection point for the industry."

Larrabee a thing of the past now. Knights Ferry and Intel’s MIC (Many Integrated Core) architecture with 32 cores for now. Public announcement: May 31, 2010
Putting things in perspective...

<table>
<thead>
<tr>
<th>The way business has been run in the past</th>
<th>It will probably change to this…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increasing clock frequency is primary method of performance improvement</td>
<td>Processors parallelism is primary method of performance improvement</td>
</tr>
<tr>
<td>Don’t bother parallelizing an application, just wait and run on much faster sequential computer</td>
<td>Nobody is building one processor per chip. This marks the end of the La-Z-Boy programming era</td>
</tr>
<tr>
<td>Less than linear scaling for a multiprocessor is failure</td>
<td>Given the switch to parallel hardware, even sub-linear speedups are beneficial as long as you beat the sequential</td>
</tr>
</tbody>
</table>
End: Discussion of Computational Models and Trends

Beginning: Overview of HW&SW for Parallel Computing
Before We Get Started…

- Last time
  - Wrap up overview of C programming
  - Start overview of parallel computing
    - Focused primarily on the limitations with the sequential computing model
    - These limitations and Moore’s law usher in the age of parallel computing

- Today
  - Discuss parallel computing models, hardware and software
  - Start discussion about GPU programming and CUDA

- Thank you, to those of you who took the time to register for auditing
The memory baseline is 64 KB DRAM in 1980 with a 1.07/year improvement in latency. CPU speed improved at 1.25/year till 1986, 1.52/year until 2004, and 1.2/year thereafter.
Vision of the Future

- “Parallelism for Everyone”
- Parallelism changes the game
  - A large percentage of people who provide applications are going to have to care about parallelism in order to match the capabilities of their competitors.

**competitive pressures = demand for parallel applications**

Presentation Paul Petersen, Sr. Principal Engineer, Intel
Intel Larrabee and Knights Ferris

- Paul Otellini, President and CEO, Intel
  - "We are dedicating all of our future product development to multicore designs"
  - "We believe this is a key inflection point for the industry."

Larrabee a thing of the past now. Knights Ferry and Intel’s MIC (Many Integrated Core) architecture with 32 cores for now. Public announcement: May 31, 2010
# Putting things in perspective…

<table>
<thead>
<tr>
<th>The way business has been run in the past</th>
<th>It will probably change to this…</th>
</tr>
</thead>
<tbody>
<tr>
<td>Increasing clock frequency is primary method of performance improvement</td>
<td>Processors parallelism is primary method of performance improvement</td>
</tr>
<tr>
<td>Don’t bother parallelizing an application, just wait and run on much faster sequential computer</td>
<td>Nobody is building one processor per chip. This marks the end of the La-Z-Boy programming era</td>
</tr>
<tr>
<td>Less than linear scaling for a multiprocessor is failure</td>
<td>Given the switch to parallel hardware, even sub-linear speedups are beneficial as long as you beat the sequential</td>
</tr>
</tbody>
</table>
End: Discussion of Computational Models and Trends

Beginning: Overview of HW&SW for Parallel Computing
Amdhal’s Law


“A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude”

- Let $r_s$ capture the amount of time that a program spends in components that can only be run sequentially
- Let $r_p$ capture the amount of time spent in those parts of the code that can be parallelized.
- Assume that $r_s$ and $r_p$ are normalized, so that $r_s + r_p = 1$
- Let $n$ be the number of threads used to parallelize the part of the program that can be executed in parallel
- The “best case scenario” speedup is

$$S = \frac{T_{old}}{T_{new}} = \frac{1}{r_s + \frac{r_p}{n}}$$
Amdahl’s Law

Sometimes called the law of diminishing returns

In the context of parallel computing used to illustrate how going parallel with a part of your code is going to lead to overall speedups

The art is to find for the same problem an algorithm that has a large $r_p$
- Sometimes requires a completely different angle of approach for a solution

Nomenclature: algorithms for which $r_p=1$ are called “embarrassingly parallel”
Example: Amdhal’s Law

- Suppose that a program spends 60% of its time in I/O operations, pre and post-processing.
- The rest of 40% is spent on computation, most of which can be parallelized.
- Assume that you buy a multicore chip and can throw 6 parallel threads at this problem. What is the maximum amount of speedup that you can expect given this investment?
- Asymptotically, what is the maximum speedup that you can ever hope for?
A Word on “Scaling”

- **Algorithmic Scaling** of a solution algorithm
  - You only have a mathematical solution algorithm at this point
  - Refers to how the effort required by the solution algorithm scales with the size of the problem
  - Examples:
    - Naïve implementation of the N-body problem scales like $O(N^2)$, where $N$ is the number of bodies
    - Sophisticated algorithms scale like $O(N \cdot \log N)$
    - Gauss elimination scales like the cube of the number of unknowns in your linear system

- **Scaling on an implementation on a certain architecture**
  - **Intrinsic Scaling**: how the wall-clock run time increase with the size of the problem
  - **Strong Scaling**: how the wall-clock run time of an implementation changes when you increase the processing resources
  - **Weak Scaling**: how the wall-clock run time changes when you increase the problem size but also the processing resources in a way that basically keeps the ration of work/processor constant
  - Order of relevance: strong, intrinsic, weak

- A thing you should worry about: is the Intrinsic Scaling similar to the Algorithmic Scaling?
  - If Intrinsic Scaling significantly worse than Algorithmic Scaling:
    - You might have an algorithm that thrashes the memory badly, or
    - You might have a sloppy implementation of the algorithm
Overview of Large Multiprocessor Hardware Configurations
Newton: 24 GPU Cluster
~ Hardware Configurations ~
Some Nomenclature

- Shared addressed space: when you invoke address “0x0043fc6f” on one machine and then invoke “0x0043fc6f” on a different machine they actually point to the same global memory space
  - Issues: memory coherence
    - Fix: software-based or hardware-based

- Distributed addressed space: the opposite of the above

- Symmetric Multiprocessor (SMP): you have one machine that shares amongst all its processing units a certain amount of memory (same address space)
  - Mechanisms should be in place to prevent data hazards (RAW, WAR, WAW). Goes back to memory coherence

- Distributed shared memory (DSM):
  - Also referred to as distributed global address space (DGAS)
  - Although physically memory is distributed, it shows as one uniform memory
  - Memory latency is highly unpredictable
Example, SMP

- Shared-Memory Multiprocessor (SMP) Architecture

Usually SRAM

Usually DRAM

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Comments, SMP Architecture

- Multiple processor-cache subsystems share the same physical off-chip memory

- Typically connected to this off-chip memory by one or more buses or a switch

- Key architectural property: uniform memory access (UMA) time to all of memory from all the processors
  - This is why it’s called symmetric
SRAM vs. DRAM

- **SRAM – Static Random Access Memory**
  - Six transistors
  - Need only be set once, no need to recharge as long as power is not cut off
  - Bulky and expensive
  - Very fast
  - Usually used for cache memory

- **DRAM – Dynamics Random Access Memory**
  - One transistor and two capacitors
  - The “Dynamic” attribute: Capacitors need to be constantly recharged
    - Therefore, longer access times, more power thirsty
    - Compact
    - Used for off-chip memory
Example

- Distributed-memory multiprocessor architecture (Newton, for instance)
Comments, distributed-memory multiprocessor architecture

- Basic architecture consists of nodes containing a processor, some memory, typically some I/O, and an interface to an interconnection network that connects all the nodes.

- Individual nodes may contain a small number of processors, which may be interconnected by a small bus or a different interconnection technology, which is less scalable than the global interconnection network.

- Popular interconnection network: Mellanox and Qlogic InfiniBand
  - Bandwidth: 40 GB/sec
  - Latency: in the microsecond range
  - Requires special network cards: HCA – “Host Channel Adaptor”

- InfiniBand offers point-to-point bidirectional serial links intended for the connection of processors with high-speed peripherals such as disks.
  - Basically, a protocol and implementation for communicating data very fast
  - It supports several signaling rates and, as with PCI Express, links can be bonded together for additional throughput
  - Similar technologies: Fibre Channel, PCI Express, Serial ATA, etc.
Examples...

- **Shared-Memory**
  - Nehalem micro-architecture, released in October 2008
  - AMD “Barcelona” (quad-core)
  - Sun Niagara

- **Distributed-Memory**
  - IBM BlueGene/L
  - Cell (see http://users.ece.utexas.edu/~adnan/vlsi-07/hofstee-cell.ppt)

- **Mini-cores**
  - GPGPUs – General Purpose GPUs
Flynn’s Taxonomy of Architectures

- SISD - Single Instruction/Single Data
- SIMD - Single Instruction/Multiple Data
- MISD - Multiple Instruction/Single Data
- MIMD - Multiple Instruction/Multiple Data
Single Instruction/Single Data Architectures

Your desktop, before the spread of dual core CPUs

Flavors of SISD

Instructions:

Pipelining

Instruction-Level Parallelism (ILP)
More on pipelining…
Related to the Idea of Pipelining...

- Most processors have multiple pipelines for different tasks, and can start a number of different operations each cycle
- Example: each core in an Intel Core 2 Duo chip
  - 14-stage pipeline
  - 3 integer units (ALU)
  - 1 floating-point addition unit (FPU)
  - 1 floating-point multiplication unit (FPU)
  - 2 load/store units
  - In principle, capable of producing 3 integer and 2 FP results per cycle
  - FP division is very slow

Credits: Mike Giles, Oxford University
Single Instruction/Multiple Data Architectures

Processors that execute same instruction on multiple pieces of data: NVIDIA GPUs

Single Instruction/Multiple Data
[Cndtd.]

- Each core runs the same set of instructions on different data
- Examples:
  - Graphics Processing Unit (GPU): processes pixels of an image in parallel
  - CRAY’s vector processor, see image below

Slide Source: Klimovitski & Macri, Intel
SISD versus SIMD

Writing a compiler for SIMD architectures is VERY difficult (inter-thread communication complicates the picture...)

Slide Source: ars technica, Peakstream article
Multiple Instruction/Single Data

Not useful, not aware of any commercial implementation...

Multiple Instruction/Multiple Data

As of 2006, all the top 10 and most of the TOP500 supercomputers were based on a MIMD architecture
Multiple Instruction/Multiple Data

- The sky is the limit: each PU is free to do as it pleases
- Can be of either shared memory or distributed memory categories
HPC: Where Are We Today?
[Info lifted from Top500 website: http://www.top500.org/]

<table>
<thead>
<tr>
<th>NAME/MANUFACTURER/PURPOSE</th>
<th>LOCATION</th>
<th>COUNTRY</th>
<th>CORES</th>
<th>Rmax PTFlops/s</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tianhe-1A</td>
<td>NUDT/NSCC/Tianjin</td>
<td>China</td>
<td>186,368</td>
<td>2.57</td>
</tr>
<tr>
<td>Jaguar</td>
<td>DOE/SC/ORNL</td>
<td>USA</td>
<td>224,162</td>
<td>1.76</td>
</tr>
<tr>
<td>Nebulae</td>
<td>NSCC</td>
<td>China</td>
<td>120,640</td>
<td>1.27</td>
</tr>
<tr>
<td>Tsubame 2.0</td>
<td>TiTech</td>
<td>Japan</td>
<td>73,278</td>
<td>1.19</td>
</tr>
<tr>
<td>Hopper</td>
<td>DOE/SC/LBNL</td>
<td>USA</td>
<td>153,408</td>
<td>1.05</td>
</tr>
</tbody>
</table>
Where Are We Today?

[Cntd.]

- **Abbreviations/Nomenclature**
  - MPP – Massively Parallel Processing
  - Constellation – subclass of cluster architecture envisioned to capitalize on data locality
  - MIPS – “Microprocessor without Interlocked Pipeline Stages”, a chip design of the MIPS Computer Systems of Sunnyvale, California
  - SPARC – “Scalable Processor Architecture” is a RISC instruction set architecture developed by Sun Microsystems (now Oracle) and introduced in mid-1987
  - Alpha - a 64-bit reduced instruction set computer (RISC) instruction set architecture developed by DEC (Digital Equipment Corporation was sold to Compaq, which was sold to HP)
How is the speed measured to put together the Top500?
  - Basically reports how fast you can solve a dense linear system
Some Trends...

- **Consequence of Moore’s law**
  - Transition from a speed-based compute paradigm to a concurrency-based compute paradigm

- **Amount of power for supercomputers is a showstopper**
  - Example:
    - Exascale Flops/s rate: reach it by 2018
    - Budget constraints: must be less than $200 million
    - Power constraints: must require less than 20 MW hour
  - Putting things in perspective:
    - World’s (China’s) fastest supercomputer: 4.04 Mwatts for 2.57 Petaflop/s
    - Oak Ridge Jaguar’s – US fastest supercomputer: 7.0 Mwats for 1.76 Petaflop/s
    - Faster machine for less power: the advantage of GPU computing
Parallel Programming Support (non-GPU)

- Message Passing Interface (MPI)
  - Originally aimed at distributed memory architectures, now very effective on shared memory

- OpenMP

-Threads
  - Pthreads (“P” comes from Posix)
  - Cell threads

- Parallel Libraries
  - Intel’s Thread Building Blocks (TBB) - mature
  - Microsoft’s Task Parallel Library - mature
  - SWARM (GTech) – small scope
  - STAPL (Standard Template Adaptive Parallel Library, B. Stroustrup Texas A&M) – undergoing effort
GPU Parallel Programming Support

- CUDA (NVIDIA)
  - C/C++ extensions

- Brook (Stanford)
  - Relies on language extensions
    - Draws on OpenGL v1.3+, DirectX v9+ or AMD's Close to Metal for the computational backend
    - Runs on Windows and Linux

- Brook+ (AMD/ATI)
  - AMD-enhanced implementation of Brook

- SH (Waterloo)
  - Became RapidMind, commercial venture, acquired in 2009 by Intel
  - Library and language extensions
  - Works on multicores as well

- PeakStream
  - Now defunct, acquired by Google, June 2007
Why Dedicate So Much Time to GPU?

- It’s fast for a variety of jobs
  - Really good for data parallelism (requires SIMD)
  - Bad for task parallelism (requires MIMD)

- It’s cheap to get one ($120 to $480)

- It’s everywhere
  - There is incentive to produce software since there are many potential users of it…
GPU Proved Fast in Several Applications

- 146X Medical Imaging U of Utah
- 36X Molecular Dynamics U of Illinois, Urbana
- 18X Video Transcoding Elemental Tech
- 50X Matlab Computing AccelerEyes
- 100X Astrophysics RIKEN

- 149X Financial simulation Oxford
- 47X Linear Algebra Universidad Jaime
- 20X 3D Ultrasound Techniscan
- 130X Quantum Chemistry U of Illinois, Urbana
- 30X Gene Sequencing U of Maryland
CPU vs. GPU – Flop Rate (GFlop/Sec)

- Single Precision
- Double Precision

GFlop/Sec

2003 2004 2005 2006 2007 2008 2009 2010

Tesla 8-series
Tesla 10-series
Tesla 20-series
Westmere 3 GHz
Nehalem 3 GHz
GPU vs. CPU – Memory Bandwidth

[GB/sec]

- Tesla 8-series
- Tesla 10-series
- Tesla 20-series
- Nehalem 3 GHz
- Westmere 3 GHz

GB/Sec

2003 2004 2005 2006 2007 2008 2009 2010
# Key Parameters

## GPU, CPU

<table>
<thead>
<tr>
<th></th>
<th>GPU – NVIDIA Tesla C2050</th>
<th>CPU – Intel core i7 975 Extreme</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processing Cores</td>
<td>448</td>
<td>4</td>
</tr>
<tr>
<td>Memory</td>
<td>3 GB</td>
<td>- 32 KB L1 cache / core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 256 KB L2 (I&amp;D)cache / core</td>
</tr>
<tr>
<td></td>
<td></td>
<td>- 8 MB L3 (I&amp;D) shared by all cores</td>
</tr>
<tr>
<td>Clock speed</td>
<td>1.15 GHz</td>
<td>3.20 GHz</td>
</tr>
<tr>
<td>Memory bandwidth</td>
<td>140 GB/s</td>
<td>32.0 GB/s</td>
</tr>
<tr>
<td>Floating point</td>
<td>515 x 10^9</td>
<td>70 x 10^9</td>
</tr>
<tr>
<td>operations/s</td>
<td>Double Precision</td>
<td>Double Precision</td>
</tr>
</tbody>
</table>
IBM BlueGene/L

- Entry model: 1024 dual core nodes
- 5.7 Tflop/s
- Linux OS
- Dedicated power management solution
- Dedicated IT support
- Only decent options for productivity tools (debugging, profiling, etc.)
  - TotalView
- Price (2007): $1.4 million
"I hear there's rumors on the Internets that we're going to have a draft."

G. W. Bush
Before We Get Started…

- Last time
  - Wrap up overview of parallel computing
    - Amdhal’s Law of diminishing return
    - Flynn’s taxonomy of computer architectures
    - Types of parallel computing architectures

- Today
  - HPC vs. HTC
  - Start discussion about GPU programming and CUDA
  - Building CUDA apps in Visual Studio 2008. Running apps through the HPC scheduler on Newton

- Assignment 2 posted
  - Problem 1: making sure you can run a CUDA job
  - Problem 2: simple problem -> run basic job on the GPU (one block with four parallel threads)
  - Use the forum to post questions/answers/comments
  - Due: Feb 8, 11:59 pm, email to me964uw@gmail.com
High Performance Computing (HPC) vs. High Throughput Computing (HTC)

- **High Performance Computing**
  - Topic of interest in this class
  - The idea: run one executable as fast as you can
    - Might spend one month running one DFT job or a week on a CFD job…

- **High Throughput Computing**
  - The idea: run as many applications as you can, possibly at the same time on different machines
  - Example: bone analysis in ABAQUS
    - You have uncertainty in the length of the bone (20 possible lengths) in the material of the bone (10 values for Young’s modulus) in the loading of the bone (50 force values with different magnitude/direction). Grand total: 10,000 ABAQUS runs
    - We have 1400 workstations hooked up together on-campus -> use Condor to schedule the 10,000 independent ABAQUS jobs and have them run on scattered machines overnight
  - Example: folding@home – volunteer your machine to run a MD simulation when it’s idle
High Performance Computing (HPC) vs. High Throughput Computing (HTC)

- High Performance Computing
  - Usually one cluster (e.g. Newton) or one massively parallel architecture (e.g. IBM Blue Gene or Cray) that is dedicated to running one large application that requires a lot of memory, a lot of compute power, and a lot of communication
    - Example: each particle in a MD simulation requires (due to long range electrostatic interaction) to keep track of a large number of particles that it interacts with. Needs to query and figure out where these other particles are at any time step of the numerical integration
  
  - What is crucial is the interconnect between the processing units
    - Typically some fast dedicated interconnect (e.g. InfiniBand), which operates at 40 GB/s
      - Euclid@UW-Madison: 1 GB/s Ethernet, Bluewaters@UIU/C: 100 GB/s, Tianhe-I claims double the speed of Infiniband

- Typically uniform hardware components: e.g. 10,000 Intel Xeon 5520, or 64 Tesla C2050 cards, etc.

- Comes at a premium $$$
High Performance Computing (HPC) vs. High Throughput Computing (HTC)

- **High Throughput Computing**
  
  - Usually a collection of heterogeneous compute resources linked through a slow connection, most likely Ethernet
    - Example: 120 Windows workstations in the CAE labs (all sorts of machines, some new, some old)
  
  - When CAE machine 58 runs an ABAQUS bone simulation there is no communication needed with CAE machine 83 that runs a different ABAQUS scenario
  
  - Don’t need to spend any money, you can piggyback on resources that are willing to make themselves available
  
  - Very effective to run Monte Carlo type analyses
High Performance Computing (HPC) vs. High Throughput Computing (HTC)

- You can do HPC on a configuration that has slow interconnect
  - It will run very very slow…

- You can do HTC on an IBM Blue Gene
  - You need to have the right licensing system in place to “check out” 10,000 ABAQUS licenses
  - You will use the processors but will waste the fast interconnect that made the machine expensive in the first place

- University of Wisconsin-Madison well known due to the pioneering work in the area of HTC done by Professor Miron Livny in CS
  - UW-Madison solution for HTC: Condor, used by a broad spectrum of organizations from academia and industry
  - Other commercial solutions now available for HTC: PBSWorks, form Altair
  - Google and Amazon are heavily invested in the HTC idea

- The line between HPC and HTC is blurred when it comes to cloud computing
  - Cloud computing: you rely on hardware resources made available by a third party. The solution of choice today for HTC. If the machines in the cloud linked by fast interconnect one day might consider running HPC jobs there as well…
End: Overview of H&S for parallel computing

Beginning: GPU Computing, CUDA Programming Model
Acknowledgements

- Many slides herein include material developed at the University of Illinois Urbana-Champaign by Professor W. Hwu and Adjunct Professor David Kirk (the latter was Chief Scientist at NVIDIA back in the day)

- The slides are used with the permission of the authors, which is gratefully acknowledged
  - Slides that include material produced by professors Hwu and Kirk contain a HK-UIUC logo in the lower left corner of the slide

- Several other slides are lifted from other sources as indicated along the way
Why GPU computing in ME964?

- Class devoted to High Performance Computing in Engineering Applications

- GPU computing is not quite High Performance Computing (HPC)
  - However, it shares with HPC the important aspect that they both draw on parallel programming
  - A bunch of GPUs can together lead to a HPC cluster, see example of Tianhe-I, the fastest supercomputer in the world at this time

- GPUs are called sometimes accelerators or co-processors
  - Complement the capability of the CPU core[s]

- GPU proved very useful in computing collision detection, image processing, ray tracing, N-body problems, CFD, FFT, etc.

- More than 60 million NVIDIA GPU cards in use today
CPU (the “host”)

GPU w/ local DRAM (the “device”)

Wikipedia
Bandwidth in a CPU-GPU System

- **CPU**
  - processing elements
  - cache: 40 GB/s

- **Co-processor**

- **System Memory**
  - 6-30 GB/s

- **Device Memory**
  - 1-160 GB/s

Infiniband to next node: 1-2 GB/s

Device memory to next node: 1-8 GB/s

Robert Strzodka, Max Plank Institute, Germany
PCI-Express Latency

Latency, DRAM Memory Access

<table>
<thead>
<tr>
<th>Year of introduction</th>
<th>Chip size</th>
<th>Slowest DRAM (ns)</th>
<th>Fastest DRAM (ns)</th>
<th>Column access strobe (CAS)/data transfer time (ns)</th>
<th>Cycle time (ns)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1980</td>
<td>64K bit</td>
<td>180</td>
<td>150</td>
<td>75</td>
<td>250</td>
</tr>
<tr>
<td>1983</td>
<td>256K bit</td>
<td>150</td>
<td>120</td>
<td>50</td>
<td>220</td>
</tr>
<tr>
<td>1986</td>
<td>1M bit</td>
<td>120</td>
<td>100</td>
<td>25</td>
<td>190</td>
</tr>
<tr>
<td>1989</td>
<td>4M bit</td>
<td>100</td>
<td>80</td>
<td>20</td>
<td>165</td>
</tr>
<tr>
<td>1992</td>
<td>16M bit</td>
<td>80</td>
<td>60</td>
<td>15</td>
<td>120</td>
</tr>
<tr>
<td>1996</td>
<td>64M bit</td>
<td>70</td>
<td>50</td>
<td>12</td>
<td>110</td>
</tr>
<tr>
<td>1998</td>
<td>128M bit</td>
<td>70</td>
<td>50</td>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>2000</td>
<td>256M bit</td>
<td>65</td>
<td>45</td>
<td>7</td>
<td>90</td>
</tr>
<tr>
<td>2002</td>
<td>512M bit</td>
<td>60</td>
<td>40</td>
<td>5</td>
<td>80</td>
</tr>
<tr>
<td>2004</td>
<td>1G bit</td>
<td>55</td>
<td>35</td>
<td>5</td>
<td>70</td>
</tr>
<tr>
<td>2006</td>
<td>2G bit</td>
<td>50</td>
<td>30</td>
<td>2.5</td>
<td>60</td>
</tr>
</tbody>
</table>

Figure 5.13 Times of fast and slow DRAMs with each generation. (Cycle time is defined on page 310.) Performance improvement of row access time is about 5% per year. The improvement by a factor of 2 in column access in 1986 accompanied the switch from NMOS DRAMs to CMOS DRAMs.

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Parallel Computing on a GPU

- NVIDIA GPU Computing Architecture
  - Via a separate HW interface
  - In laptops, desktops, workstations, servers

- Tesla C2050 deliver 0.515 Tflop in double precision

- Programming model scales transparently
  - Large applications have good potential for strong scaling

- Multithreaded SIMT model uses application data parallelism and thread parallelism

- Programmable in C with CUDA tools
  - “Extended C”

Tesla C2050

Tesla C1060
CPU vs. GPU – Flop Rate (GFlops)

- Tesla 8-series
- Tesla 10-series
- Tesla 20-series
- Westmere 3 GHz
- Nehalem 3 GHz

- Single Precision
- Double Precision

GFlop/Sec vs. Year (2003-2010)
What is Driving this Evolution?

- The GPU is specialized for compute-intensive, highly data parallel computation (owing to its graphics rendering origin)
  - More transistors can be devoted to data processing rather than data caching and control flow
  - Where are GPUs good: high arithmetic intensity (the ratio between arithmetic operations and memory operations)

- The fast-growing video game industry exerts strong economic pressure that forces constant innovation
ALU – Arithmetic Logic Unit
[one-slide detour]

- Digital circuit that performs arithmetic and logical operations
- Fundamental building block of a processing unit (CPU and GPU)

A and B operands (the data, coming from input registers)
F is an operator (“+”, “-”, etc.) – specified by the control unit
R is the result, stored in output register
D is an output flag passed back to the control unit
Today the GPU can be used for more than just generating graphics
  - The computational resources are there, they are most of the time underutilized

GPU, going beyond graphics:
  - The GPU is connected to the CPU by a reasonable fast bus (8 GB/s is typical today)
  - The idea is to use the GPU as a co-processor
    - Farm out big parallel jobs to the GPU
    - CPU stays busy with the control of the execution and “corner” tasks
    - You have to copy data down into the GPU, and then fetch results back
      - Ok if this data transfer is overshadowed by the number crunching done using that data (remember Amdahl’s law...)

18
What is GPGPU?

- General Purpose computation using GPU in applications other than 3D graphics
  - GPU accelerates critical path of application

- Data parallel algorithms leverage GPU attributes
  - Large data arrays, streaming throughput
  - Fine-grain **SIMD** parallelism
  - Low-latency floating point (FP) computation

- Applications – see [http://GPGPU.org](http://GPGPU.org)
  - Game effects, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting
Shaders

A shader: set of software instructions mostly used to calculate rendering effects on graphics hardware with a good degree of flexibility.

Shaders are used to program the graphics processing unit (GPU) programmable rendering pipeline:
- Represent a set of instructions executed by a GPU thread.

Shader-programming replaced the fixed-function pipeline that allowed only common geometry transformation and pixel-shading functions.

Shaders enable customized effects.
GPGPU Constraints

- Dealing with graphics API
  - Working with the corner cases of the graphics API

- Addressing modes
  - Limited texture size/dimension

- Shader capabilities
  - Limited outputs

- Instruction sets
  - Lack of Integer & bit ops

- Communication limited
  - Between pixels
  - Only gather (can read data from other pixels), but no scatter (can only write to one pixel)

Summing Up: Mapping computation problems to graphics rendering pipeline was tedious…
CUDA

- “Compute Unified Device Architecture” put together by NVIDIA
  - Came out in 2006, very basic support back then

- What is it? A general purpose programming model/infrastructure
  - Parallel programming model
  - Instruction set architecture (ISA)

- Targeted software stack
  - Compute oriented drivers, language, and tools

- Driver for loading computation programs into GPU
  - Standalone Driver - Optimized for computation
  - Interface designed for compute - graphics free API
  - Explicit GPU memory management
CUDA Architecture
The Software Stack

- What we are using in this class is CUDA C. Can go beyond that though, CUDA represents an architecture that facilitates interaction with the GPU (hardware accelerator)
  - In other words, we are looking at CUDA through a pair of C binoculars…
The 30,000 Feet Perspective

- The CUDA model draws on three key abstractions
  - A hierarchy of thread groups
  - A hierarchy of memory spaces
  - Synchronization mechanisms
Running Code on Parallel Computers

[one slide detour]

- You come to rely on compiler to figure out the parallelism in a piece of code and then map it to an underlying hardware
  - VERY hard, the holy grail in parallel computing

- You rely on parallel libraries built for a specific underlying hardware
  - Very convenient, the way to go when such libraries are available

- You rely on language extensions and with human interaction you facilitate the process of generating a parallel executable
  - This is the solution embraced with CUDA
The CUDA Way: Extended C

- Declaration specifications:
  - global, device, shared, local, constant

- Keywords
  - threadIdx, blockIdx

- Intrinsics
  - __syncthreads

- Runtime API
  - For memory, symbol, execution management

- Kernel launch

```c
__device__ float filter[N];
__global__ void convolve (float *image) {
  __shared__ float region[M];
  ...
  region[threadIdx.x] = image[i];
  __syncthreads()
  ...
  image[j] = result;
}

// Allocate GPU memory
void *myimage = cudaMalloc(bytes)

// 100 blocks, 10 threads per block
convolve<<<<100, 10>>> (myimage);
```
CUDA Programming Model: A Highly Multithreaded Coprocessor

- The GPU is viewed as a compute device that:
  - Is a co-processor to the CPU or host
  - Has its own DRAM (device memory, or global memory in CUDA parlance)
  - Runs many threads in parallel

- Data-parallel portions of an application are executed on the device as kernels which run in parallel on many threads

- Differences between GPU and CPU threads
  - GPU threads are extremely lightweight
    - Very little creation overhead
  - GPU needs 1000s of threads for full efficiency
    - Multi-core CPU needs only a few heavy ones
GPU: Underlying Hardware

- NVIDIA nomenclature used below, reminiscent of GPU’s mission

- The hardware organized as follows:
  - One Stream Processor Array (SPA)…
    - … has a collection of Texture Processor Clusters (TPC, ten of them on C1060) …
      - …and each TPC has three Stream Multiprocessors (SM) …
        - …and each SM is made up of eight Stream or Scalar Processor (SP)
NVIDIA TESLA C1060
[Newton’s GPU cards]

- 240 Scalar Processors
- 4 GB device memory
- Memory Bandwidth: 102 GB/s
- Clock Rate: 1.3GHz
- Approx. $1,250
GPU Processor Terminology

- Keep in mind that the GPU is a SIMD device, so it works on “streams” of data
  - Each “GPU thread” executes one general instruction on the stream of data that it is assigned to handle
  - The NVIDIA calls this model SIMT (single instruction multiple thread)

- The number crunching power comes from a vertical hierarchy:
  - One Stream Processor Array (SPA)…
    - …which has a collection of Texture Processor Clusters (TPC, ten of them on Tesla C1060) …
      - …and each TPC has three Stream Multiprocessors (SM) …
        - …and each SM is made up of eight Stream or Scalar Processor (SP)

- The quantum of scalability is the SM
  - The more $ you pay the more SMs you get on your CPU
Compute Capability [of a Device] vs. CUDA Version

- “Compute Capability of a Device” (usually referred as “compute capability”) tells you how advanced your hardware is
  - Defined by a major revision number and a minor revision number

- Example:
  - Newton’s Tesla C1060 is compute capability 1.3
  - Tesla C2050 is compute capability 2.0
  - The major revision number is up to 2 (Fermi architecture)
  - The minor revision number indicates incremental changes within an architecture class

- A higher compute capability indicates an abler piece of hardware

- The CUDA Version indicates what version of the software you are using to operate on the hardware
  - Right now, the most recent version of CUDA, released in November 2010, is 3.2

- In a perfect world, you have the most recent CUDA (version 3.2), running on the most recent architecture (compute capability 2.1)
Compatibility Issues

- The basic rule: the CUDA Driver API is backward, but not forward compatible
  - Makes sense, the functionality in later versions increased, was not there in previous versions
- In this class we’ll use version 3.2 of CUDA (the latest)
Setting up a Visual Studio 2008 CUDA Project
Before We Get Started…

● Assumptions
  ● Visual Studio 2008 installed
  ● CUDA Toolkit and GPU Computing SDK 3.2 installed
  ● (Optional) Developer Drivers installed

● Overview
  ● Set system environment variables for missing DLLs
  ● Run CUDA VS Wizard
  ● Set up project properties
  ● Compile and run example CUDA program
Environment Setup

- Why? Some DLLs missing from the %PATH%
  - Add ;%NVSDKCOMPUTE_ROOT%\C\common\lib to system PATH environment variable
  - Under: My Computer -> System Properties -> Advanced -> Environment Variables
  - Alternatively: copy any missing DLLs to output dir
Configuring the Project

- Install CUDA VS Wizard* ([http://goo.gl/Fh55o](http://goo.gl/Fh55o))
- Start Visual Studio, File -> New Project
- CUDA{32|64} -> CUDAWinApp, give name
- Next -> select Empty Project -> Finish

- Right click project -> Properties (if using shrUtils from Nvidia)
  - Linker -> General
    - Add ‘$(NVSDKCOMPUTE_ROOT)\shared\lib’ to Add’l Lib Dirs
  - Linker -> Input
    - Add shrUtils{32|64}{|D}.lib to Add’l Deps
    - (Choose 32 or 64, nothing or D if release or debug)
  - CUDA Build Rule -> General
    - Add ‘$(NVSDKCOMPUTE_ROOT)\shared\inc’ to Add’l Include Dirs

* (Preferable) Use CUDA build rules directly, but the Wizard sets various other options for you
Writing Code

- F7 to compile, F5 to debug (if you have a CUDA-capable GPU)
- Once it works, copy to Newton (next)
Helpful Hints

(Compiling)
- Missing header (*.h) files:
  - Add path to the ‘include’ dir containing the header
- Symbol not found
  - Add *.lib file under dependencies, may need to add library dir

(When running program)
- *.dll not found
  - Either add the DLL dir to system path or copy DLL to program dir
- Black (cmd) window opens, quickly disappears
  - Run from command prompt or
  - (dangerous if on cluster) add a system(“pause”) at the end
Helpful Hints (cont’d)

- For code completion (‘Intellisense’) and pretty colors:
  - Run *.reg file in %CUDA_PATH%\extras\visual_studio_integration
  - In Visual Studio, Tools -> Options -> Text Editor -> File Extension
    - Extension: cu
    - Editor: Microsoft Visual C++
Running a CUDA app on the GPU cluster (Newton)
Some Quick Notes…

- Must be inside the College of Engineering firewall to access Newton
  - From a CAE lab
  - On UWNNet-Engineering wireless
  - Via Engineering VPN (*not* WiscVPN)
  - Remote Desktop via Kaviza - [http://remote.cae.wisc.edu/dt/](http://remote.cae.wisc.edu/dt/)

- User accounts managed by CAE
  - All students registered for the class are eligible
  - Auditors from outside CoE can request a temporary account
  - Will see ‘ME964’ under your groups in My.CAE once you have access to the cluster

- This presentation will cover how to submit jobs via Remote Desktop
  - Future presentation will show how to install HPC Pack and submit jobs from your local machine
Getting Started

- Copy all files for your program to Newton
  - `\newton.msvc.wisc.edu\Data\ME964\%username%`
    - (replace `%username%` with your CAE username)
Remote Desktop to Newton

- Computer: newton.msvc.wisc.edu
- User name: ENGR\%username%
Start HPC Job Manager

- Start -> All Programs -> Microsoft HPC Pack 2008 R2
Creating a Job

- New Single-Task Job
  - Command line: program to run
  - Working directory: `\newton\Data\ME964\%username%`
  - Give a filename for STDERR & STDOUT, though STDIN is optional
  - Select number of cores to use
    - Your program must be written to take advantage of them (e.g., via threads)
Creating a Job - Notes

- Single-Task Job simplest to setup
- New Job gives you much more control
- Parametric Sweep lets you run the same program with different parameters (Monte Carlo)

- GPUs are not currently reserved – be careful
  - Working on this, HPC Pack does not natively let you do it

- All files for your program must reside in the working directory – unlike Condor, HPC Pack does not take care of this for you
Finishing Up

- After submitting, you can monitor the job’s progress in the Job Manager.
- Once it finishes (or fails), double click for more info:
  - Which tasks finished/failed
  - Task outputs
  - Where each task ran
Why Did It Fail?

- Most common: libraries not installed
- 2nd most common: compiled using wrong version of CUDA Toolkit
  - When in doubt, reinstall from \newton.msvc.wisc.edu\Data\Downloads\Drivers

- Does it run locally?
- Check log files for STDOUT and STDERR

- Still not working? Contact Andrew
Other Notes

- Programs must be able to finish/die without any user interaction, otherwise will hang

- OpenGL/DirectX are not available
  - TCC Mode enabled, cards are only good for number crunching

ME964
High Performance Computing for Engineering Applications

Building CUDA apps under Visual Studio
Accessing Newton
CUDA Programming Model
CUDA API
February 03, 2011

We live in a society exquisitely dependent on science and technology, in which hardly anyone knows anything about science and technology.

Carl Sagan
Before We Get Started...

- **Last time**
  - HPC vs. HTC
  - Quick overview of the CPU/GPU hardware and latency/bandwidths of relevance
  - GPGPU, GPU programming and CUDA
  - Started discussion on building CUDA apps in Visual Studio 2008

- **Today**
  - Andrew: wrap up building CUDA apps in Visual Studio 2008
  - Andrew: running apps through the HPC scheduler on Newton
  - End quick overview of CUDA programming model
  - Start discussion of CUDA API

- **HW**
  - Due next Tu: two problems
  - Includes reading of two papers: Amdhal and Manferdelli (links on the class website)
Setting up a Visual Studio 2008 CUDA Project
Before We Get Started…

- Assumptions
  - Visual Studio 2008 installed
  - CUDA Toolkit and GPU Computing SDK 3.2 installed
  - (Optional) Developer Drivers installed

- Overview
  - Set system environment variables for missing DLLs
  - Run CUDA VS Wizard
  - Set up project properties
  - Compile and run example CUDA program
Environment Setup

- Why? Some DLLs missing from the %PATH%
  - Add ;%NVSDKCOMPUTE_ROOT%\C\common\lib to system PATH environment variable
  - Under: My Computer -> System Properties -> Advanced -> Environment Variables
  - Alternatively: copy any missing DLLs to output dir
Configuring the Project

1. Install CUDA VS Wizard* ([http://goo.gl/Fh55o](http://goo.gl/Fh55o))
2. Start Visual Studio, File -> New Project
3. CUDA{32|64} -> CUDAWinApp, give name
4. Next -> select Empty Project -> Finish

5. Right click project -> Properties (if using shrUtils from Nvidia)
   - Linker -> General
     - Add ‘$(NVSDKCOMPUTE_ROOT)\shared\lib’ to Add’l Lib Dirs
   - Linker -> Input
     - Add shrUtils{32|64}{{|D}.lib to Add’l Deps
     - (Choose 32 or 64, nothing or D if release or debug)
   - CUDA Build Rule -> General
     - Add ‘$(NVSDKCOMPUTE_ROOT)\shared\inc’ to Add’l Include Dirs

* (Preferable) Use CUDA build rules directly, but the Wizard sets various other options for you
Writing Code

- F7 to compile, F5 to debug (if you have a CUDA-capable GPU)
- Once it works, copy to Newton (next)
Helpful Hints

(Compiling)
- Missing header (*.h) files:
  - Add path to the ‘include’ dir containing the header
- Symbol not found
  - Add *.lib file under dependencies, may need to add library dir

(When running program)
- *.dll not found
  - Either add the DLL dir to system path or copy DLL to program dir
- Black (cmd) window opens, quickly disappears
  - Run from command prompt or
  - (dangerous if on cluster) add a system(“pause”) at the end
Helpful Hints (cont’d)

- For code completion (‘Intellisense’) and pretty colors:
  - Run *.reg file in %CUDA_PATH%/extras/visual_studio_integration
  - In Visual Studio, Tools -> Options -> Text Editor -> File Extension
    - Extension: cu
    - Editor: Microsoft Visual C++
Running a CUDA app on the GPU cluster (Newton)
Some Quick Notes…

- Must be inside the College of Engineering firewall to access Newton
  - From a CAE lab
  - On UWNet-Engineering wireless
  - Via Engineering VPN (not WiscVPN)
  - Remote Desktop via Kaviza - [http://remote.cae.wisc.edu/dt/](http://remote.cae.wisc.edu/dt/)

- User accounts managed by CAE
  - All students registered for the class are eligible
  - Auditors from outside CoE can request a temporary account
  - Will see ‘ME964’ under your groups in My.CAE once you have access to the cluster

- This presentation will cover how to submit jobs via Remote Desktop
  - Future presentation will show how to install HPC Pack and submit jobs from your local machine
Getting Started

- Copy all files for your program to Newton
  - `\newton.msvc.wisc.edu\Data\ME964\%username%`
  - (replace %username% with your CAE username)
Remote Desktop to Newton

- Computer: newton.msvc.wisc.edu
- User name: ENGR\%username%
Start HPC Job Manager

- Start -> All Programs -> Microsoft HPC Pack 2008 R2
Creating a Job

- **New Single-Task Job**
  - Command line: program to run
  - Working directory: `\newton\Data\ME964\%username%`
  - Give a filename for STDERR & STDOUT, though STDIN is optional
  - Select number of cores to use
    - Your program must be written to take advantage of them (eg via threads)
Creating a Job - Notes

- Single-Task Job simplest to setup
- New Job gives you much more control
- Parametric Sweep lets you run the same program with different parameters (Monte Carlo)

- GPUs are not currently reserved – be careful
  - Working on this, HPC Pack does not natively let you do it

- All files for your program must reside in the working directory – unlike Condor, HPC Pack does not take care of this for you
Finishing Up

- After submitting, you can monitor the job’s progress in the Job Manager.
- Once it finishes (or fails), double click for more info:
  - Which tasks finished/failed
  - Task outputs
  - Where each task ran
Why Did It Fail?

- Most common: libraries not installed
- 2\textsuperscript{nd} most common: compiled using wrong version of CUDA Toolkit
  - When in doubt, reinstall from \url{\newton.msvc.wisc.edu\Data\Downloads\Drivers}

- Does it run locally?
- Check log files for STDOUT and STDERR

- Still not working? Contact Andrew
Other Notes

- Programs must be able to finish/die without any user interaction, otherwise will hang

- OpenGL/DirectX are not available
  - TCC Mode enabled, cards are only good for number crunching

Back to the Overview of the CUDA Programming Model
A Simple C-CUDA Program

- You want to add two vectors A and B and store the result in a vector C
- Assume that the size of the vectors is N=512
- Here’s how things get done (some details omitted such as `#define N 512`)

```c
// Kernel definition
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main()
{
    ...
    // Kernel invocation with N threads
    VecAdd<<<1, N>>>(A, B, C);
}
```
Execution Configuration: Grids and Blocks

- A kernel is executed as a **grid of blocks of threads**
  - All threads in a kernel can access several device data memory spaces

- A **block [of threads]** is a batch of threads that can cooperate with each other by:
  - Synchronizing their execution
  - Efficiently sharing data through a low latency **shared memory**

- **Threads from two different blocks cannot cooperate!!!**
  - This has important software design implications
Block and Thread Index (Idx)

- Threads and blocks have Indices
  - Used by each thread to decide what data to work on
  - Block Index: a pair of uint
  - Thread Index: a triplet of three uint

- Why this 2D and 3D layout?
  - Simplifies memory addressing when processing multidimensional data
    - Image processing
    - Solving PDEs on subdomains
    - …
More on Thread Organization and ID
[at the block level]

- Each block organizes its threads in a 3D structure defined by its three dimensions: \( D_x \), \( D_y \), and \( D_z \) that you specify.

- A block on Tesla C1060 cannot have more than 512 threads \( \Rightarrow \) \( D_x \times D_y \times D_z \leq 512 \).
  
  - Note: On Fermi architecture this is 1024.

- Each thread in a block can be identified by a unique index \((x, y, z)\), and

  \[
  0 \leq x \leq D_x \quad 0 \leq y \leq D_y \quad 0 \leq z \leq D_z
  \]

- A triplet \((x, y, z)\) is a virtual thing that you settle upon when dealing with your algorithm. In hardware, a thread doesn’t have an index, but a unique thread id, which is computed as \( t_{id} = x + y \times D_x + z \times D_x \times D_y \).

- In general, operating for vectors typically results in you choosing \( D_y = D_z = 0 \). Handling matrices typically goes well with \( D_z = 0 \). For handling for instance PDEs in 3D you might want to have all three block dimensions nonzero.
A Couple of Built-In Variables

- It’s essential for each thread to be able to find out the grid and block dimensions and the block and thread indices.

- Each thread when executing a *device* function has access to the following built-in variables:
  - `threadIdx(uint3)`: contains the thread index within a block.
  - `blockDim(dim3)`: contains the dimension of the block.
  - `blockIdx(uint3)`: contains the block index within the grid.
  - `gridDim(of dim3)`: contains the dimension of the grid.
  - `[warpSize(uint)`: provides warp size, we’ll talk about this later…]
Example: Adding Two Matrices

- You have two matrices A and B of dimension $N \times N$ ($N=16$)
- You want to compute $C=A+B$ in parallel
- Code provided below (some details omitted, such as `#define N 16`)

```c
// Kernel definition
__global__ void MatAdd(float A[N][N], float B[N][N],
                         float C[N][N])
{
    int i = threadIdx.x;
    int j = threadIdx.y;
    C[i][j] = A[i][j] + B[i][j];
}

int main()
{
    ...
    // Kernel invocation with one block of $N \times N \times 1$ threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```
Given that the device operates with groups of threads of consecutive ID, and given the scheme a few slides ago to compute a thread ID based on the thread & block index, is the array indexing scheme on the previous slide good or bad?

The “good or bad” refers to how data is accessed in the device’s global memory.

In other words should we have

\[ C[i][j] = A[i][j] + B[i][j] \]

or…

\[ C[j][i] = A[j][i] + B[j][i] \]
CUDA Device Memory Space Overview

- Each thread can:
  - R/W per-thread registers
  - R/W per-thread local memory
  - R/W per-block shared memory
  - R/W per-grid global memory
  - Read only per-grid constant memory
  - Read only per-grid texture memory

- The host can R/W global, constant, and texture memory

**IMPORTANT NOTE:** Global, constant, and texture memory spaces are **persistent** across kernels called by the same host application.
Global, Constant, and Texture Memories (Long Latency Accesses by Host)

- **Global memory**
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads

- **Texture and Constant Memories**
  - Constants initialized by host
  - Contents visible to all threads

**NOTE:** We will not emphasize texture memory in this class.
The CUDA API
What is an API?

- **Application Programming Interface (API)**
  - A set of *functions*, *procedures* or *classes* that an operating system, library, or service provides to support requests made by computer programs (from Wikipedia)
  - Example: OpenGL, a graphics library, has its own API that allows one to draw a line, rotate it, resize it, etc.

- **Cooked up analogy (for the mechanical engineer)**
  - Think of a car, you can say it has a certain Device Operating Interface (DOI):
    - A series of pedals, gauges, steering wheel, etc. This would be its DOI

- In this context, CUDA provides an API that enables you to tap into the computational resources of the NVIDIA’s GPUs
  - This is what replaced the old GPGPU way of programming the hardware
On the CUDA API

- Reading the CUDA Programming Guide you’ll run numerous references to the CUDA Runtime API and CUDA Driver API
  - Many time they talk about “CUDA runtime” and “CUDA driver”. What they mean is CUDA Runtime API and CUDA Driver API

- CUDA Runtime API – is the friendly face that you can choose to see when interacting with the GPU. This is what gets identified with “C CUDA”
  - Needs nvcc compiler to generate an executable

- CUDA Driver API – this is more like how it was back in the day: low level way of interacting with the GPU
  - You have significantly more control over the host-device interaction
  - Significantly clunkier way to dialogue with the GPU, typically only needs a C compiler

- I don’t anticipate any reason to use the CUDA Driver API
Talking about the API: The C CUDA Software Stack

- Image at right indicates where the API fits in the picture

An API layer is indicated by a thick red line:

- NOTE: any CUDA runtime function has a name that starts with “cuda”
  - Examples: cudaMalloc, cudaFree, cudaMemcpy, etc.
- Examples of CUDA Libraries: CUFFT, CUBLAS, CUSP, thrust, etc.
Going back to the G80 HW…

split personality

Two distinct personalities in the same entity, each of which prevails at a particular time.

Some aspects of the personality of the GPU don’t fit with the computational hat that we placed on the GPU’s head.
Putting Things in Perspective...

- CUDA programming model – basic concepts and data types
  - Just finished this…

- CUDA application programming interface - basic
  - Working on it right now

- Simple example to illustrate basic concepts and functionality
  - Coming up shortly

- Performance features will be covered later
CUDA Device Memory Allocation

- `cudaMalloc()`
  - Allocates object in the device Global Memory
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object

- `cudaFree()`
  - Frees object from device Global Memory
  - Pointer to freed object
A Small Detour: A Matrix Data Type

- NOT part of CUDA
- It will be frequently used in many code examples
  - 2 D matrix
  - Single precision float elements
  - Width * height elements
  - Matrix entries attached to the pointer-to-float member called “elements”
  - Matrix is stored row-wise

```c
typedef struct {
  int width;
  int height;
  float* elements;
} Matrix;
```
CUDA Device Memory Allocation (cont.)

- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md.elements
  - “d” is often used to indicate a device data structure

```c
BLOCK_SIZE = 64;
Matrix Md;
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

cudaMalloc((void**)&Md.elements, size);
cudaFree(Md.elements);
```

All the details are spelled out in the CUDA Programming Guide 1.1 (see the resources section of the class website)

VERY USEFUL, PLEASE READ…
CUDA Host-Device Data Transfer

- **cudaMemcpy()**
  - Memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
CUDA Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToDevice are symbolic constants

```c
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToDevice);
```
Assignment 2 Pseudocode

Problem 2 can be implemented as follows (four steps):

**Step 1**: Allocate memory on the device (see cudaMalloc)

**Step 2**: Invoke kernel with one block, the block has four threads (see vector add example for passing the device pointer to the kernel)
   NOTE: each thread populates the allocated device memory with the result it computes

**Step 3**: Copy back to host the data in the device array (see cudaMemcpy)

**Step 4**: Free the memory allocated on the device (see cudaFree)
Most of the time I don't have much fun. The rest of the time I don't have any fun at all. – Woody Allen
Before We Get Started…

- **Last time**
  - Andrew: wrap up building CUDA apps in Visual Studio 2008
  - Andrew: running apps through the HPC scheduler on Newton
  - Very high-level overview of the CUDA programming model
  - Discussed Index issues in the context of the “execution configuration” and how the index of a thread translates into an ID of a thread
  - Brief discussion of the memory spaces in relation to GPU computing

- **Today**
  - Discussion of the CUDA API
  - One-on-one with Andrew if you have compile/build issues in CUDA
    - 3-5 PM in room 2042ME

- **HW**
  - HW2: due date was 02/08. Now 02/10
  - HW3 has been posted. Due date: 02/15
    - Small matrix-vector multiplication
    - Matrix addition – requires use of multiple blocks
Putting Things in Perspective...

- CUDA programming model and execution configuration
  - Basic concepts and data types - just finished this...

- CUDA application programming interface
  - Working on it next

- Simple example to illustrate basic concepts and functionality
  - Coming up shortly

- Performance features will be covered later
The CUDA API
What is an API?

- **Application Programming Interface (API)**
  - A set of functions, procedures or classes that an operating system, library, or service provides to support requests made by computer programs (from Wikipedia)
  - Example: OpenGL, a graphics library, has its own API that allows one to draw a line, rotate it, resize it, etc.

- **Cooked up analogy (for the mechanical engineer)**
  - Think of a car, you can say it has a certain Device Operating Interface (DOI):
    - A series of pedals, gauges, steering wheel, etc. This would be its DOI

- In this context, CUDA provides an API that enables you to tap into the computational resources of the NVIDIA’s GPUs
  - This is what replaced the old GPGPU way of programming the hardware
On the CUDA API

- Reading the CUDA Programming Guide you’ll run numerous references to the CUDA Runtime API and CUDA Driver API
  - Many time they talk about “CUDA runtime” and “CUDA driver”. What they mean is CUDA Runtime API and CUDA Driver API

- CUDA Runtime API – is the friendly face that you can choose to see when interacting with the GPU. This is what gets identified with “C CUDA”
  - Needs nvcc compiler to generate an executable

- CUDA Driver API – this is more like how it was back in the day: low level way of interacting with the GPU
  - You have significantly more control over the host-device interaction
  - Significantly clunkier way to dialogue with the GPU, typically only needs a C compiler

- I don’t anticipate any reason to use the CUDA Driver API
Talking about the API: The C CUDA Software Stack

- Image at right indicates where the API fits in the picture

  An API layer is indicated by a thick red line:

- NOTE: any CUDA runtime function has a name that starts with “cuda”
  - Examples: cudaMalloc, cudaFree, cudaMemcpy, etc.
- Examples of CUDA Libraries: CUFFT, CUBLAS, CUSP, thrust, etc.
## CUDA Function Declarations

<table>
<thead>
<tr>
<th>Function Declaration</th>
<th>Executed on the:</th>
<th>Only callable from the:</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>__device__ float DeviceFunc()</code></td>
<td>device</td>
<td>device</td>
</tr>
<tr>
<td><code>__global__ void KernelFunc()</code></td>
<td>device</td>
<td>host</td>
</tr>
<tr>
<td><code>__host__ float HostFunc()</code></td>
<td>host</td>
<td>host</td>
</tr>
</tbody>
</table>

- `__global__` defines a kernel function
  - Must return `void`
- `__device__` and `__host__` can be used together
CUDA Function Declarations (cont.)

- __device__ functions can’t have their address taken

- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments
    - Something like `printf` would not work…
Compiling CUDA

- Any source file containing CUDA language extensions must be compiled with **nvcc**
  - You spot such a file by its .cu suffix

- **nvcc** is a **compile driver**
  - Works by invoking all the necessary tools and compilers like cudacc, g++, cl, ...

- **nvcc** can output:
  - C code
    - Must then be compiled with the rest of the application using another tool
  - ptx code (CUDA’s ISA)
  - Or directly object code (cubin)
Compiling CUDA

- **nvcc**
  - Compile driver
  - Invokes cudacc, gcc, cl, etc.

- **PTX**
  - Parallel Thread eXecution
  - Like assembly language
  - NVIDIA’s ISA

```
ld.global.v4.f32  {{f1,f3,f5,f7}, [$r9+0];
mad.f32          $f1, $f5, $f3, $f1;
```
Compiling CUDA extended C

## The nvcc Compiler – Suffix Info

<table>
<thead>
<tr>
<th>File suffix</th>
<th>How the nvcc compiler interprets the file</th>
</tr>
</thead>
<tbody>
<tr>
<td>.cu</td>
<td>CUDA source file, containing host and device code</td>
</tr>
<tr>
<td>.cup</td>
<td>Preprocessed CUDA source file, containing host code and device functions</td>
</tr>
<tr>
<td>.c</td>
<td>'C' source file</td>
</tr>
<tr>
<td>.cc, .cxx, .cpp</td>
<td>C++ source file</td>
</tr>
<tr>
<td>.gpu</td>
<td>GPU intermediate file (device code only)</td>
</tr>
<tr>
<td>.ptx</td>
<td>PTX intermediate assembly file (device code only)</td>
</tr>
<tr>
<td>.cubin</td>
<td>CUDA device only binary file</td>
</tr>
</tbody>
</table>
CUDA API: Device Memory Allocation

- **cudaMalloc()**
  - Allocates object in the device **Global Memory**
  - Requires two parameters
    - **Address of a pointer** to the allocated object
    - **Size of** allocated object

- **cudaFree()**
  - Frees object from device **Global Memory**
    - Pointer to freed object
Example Use: A Matrix Data Type

- NOT part of CUDA API
- It will be frequently used in many code examples
  - 2 D matrix
  - Single precision float elements
  - Width * height elements
  - Matrix entries attached to the pointer-to-float member called “elements”
  - Matrix is stored row-wise

```c
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;
```
CUDA Device Memory Allocation (cont.)

- Code example:
  - Allocate a 64 * 64 single precision float array
  - Attach the allocated storage to Md.elements
  - “d” is often used to indicate a device data structure

```c
BLOCK_SIZE = 64;
Matrix Md;
int size = BLOCK_SIZE * BLOCK_SIZE * sizeof(float);

cudaMalloc((void**)&Md.elements, size);
...
//use it for what you need, then free the device memory
cudaFree(Md.elements);
```

All the details are spelled out in the CUDA Programming Guide 3.2 (see the resources section of the class website)
CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to source
    - Pointer to destination
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
CUDA Host-Device Data Transfer (cont.)

- Code example:
  - Transfer a 64 * 64 single precision float array
  - M is in host memory and Md is in device memory
  - cudaMemcpyHostToDevice and cudaMemcpyDeviceToDevice are symbolic constants

```c
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToDevice);
```
Assignment 2 Pseudocode
[short detour, helpful with assignment]

Problem 2 can be implemented as follows (four steps):

**Step 1:** Allocate memory on the device (see cudaMalloc)

**Step 2:** Invoke kernel with one block, the block has four threads (see vector add example for passing the device pointer to the kernel)

  NOTE: each of the four threads populates the allocated device memory with the result it computes

**Step 3:** Copy back to host the data in the device array (see cudaMemcpy)

**Step 4:** Free the memory allocated on the device (see cudaFree)
Simple Example: Matrix Multiplication

- A straightforward matrix multiplication example that illustrates the basic features of memory and thread management in CUDA programs
  - Leave shared memory usage until later
  - For now, concentrate on
    - Local variable and register usage
    - Thread ID usage
    - Memory data transfer API between host and device
Square Matrix Multiplication Example

- Compute $P = M \times N$
  - The matrices $P$, $M$, $N$ are of size $\text{WIDTH} \times \text{WIDTH}$

- Software Design Decisions:
  - One thread handles one element of $P$
  - Each thread will access all the entries in one row of $M$ and one column of $N$
    - $2 \times \text{WIDTH}$ read accesses to global memory
    - One write access to global memory
Multiply Using One Thread Block

- One Block of threads computes matrix P
  - Each thread computes one element of P

- Each thread
  - Loads a row of matrix M
  - Loads a column of matrix N
  - Perform one multiply and addition for each pair of M and N elements
  - Compute to off-chip memory access ratio close to 1:1
    - Not that good, acceptable for now…

- Size of matrix limited by the number of threads allowed in a thread block
Step 1: Matrix Multiplication
A Simple Host Code in C

// Matrix multiplication on the (CPU) host in double precision;

void MatrixMulOnHost(const Matrix M, const Matrix N, Matrix P)
{
    for (int i = 0; i < M.height; ++i) {
        for (int j = 0; j < N.width; ++j) {
            double sum = 0;
            for (int k = 0; k < M.width; ++k) {
                double a = M.elements[i * M.width + k]; // you'll see a lot of this…
                double b = N.elements[k * N.width + j]; // and of this as well…
                sum += a * b;
            }
            P.elements[i * N.width + j] = sum;
        }
    }
}
Step 2: Matrix Multiplication, Host-side. Main Program Code

```c
int main(void) {
    // Allocate and initialize the matrices.
    // The last argument in AllocateMatrix: should an initialization with
    // random numbers be done? Yes: 1. No: 0 (everything is set to zero)
    Matrix  M  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  N  = AllocateMatrix(WIDTH, WIDTH, 1);
    Matrix  P  = AllocateMatrix(WIDTH, WIDTH, 0);

    // M * N on the device
    MatrixMulOnDevice(M, N, P);

    // Free matrices
    FreeMatrix(M);
    FreeMatrix(N);
    FreeMatrix(P);

    return 0;
}
```
Step 3: Matrix Multiplication

Host-side code

```c
// Matrix multiplication on the device
void MatrixMulOnDevice(const Matrix M, const Matrix N, Matrix P) {
    // Load M and N to the device
    Matrix Md = AllocateDeviceMatrix(M);
    CopyToDeviceMatrix(Md, M);
    Matrix Nd = AllocateDeviceMatrix(N);
    CopyToDeviceMatrix(Nd, N);

    // Allocate P on the device
    Matrix Pd = AllocateDeviceMatrix(P);

    // Setup the execution configuration
    dim3 dimGrid(1, 1);
    dim3 dimBlock(WIDTH, WIDTH);

    // Launch the kernel on the device
    MatrixMulKernel<<<dimGrid, dimBlock>>>(Md, Nd, Pd);

    // Read P from the device
    CopyFromDeviceMatrix(P, Pd);

    // Free device matrices
    FreeDeviceMatrix(Md);
    FreeDeviceMatrix(Nd);
    FreeDeviceMatrix(Pd);
}
```
Step 4: Matrix Multiplication- Device-side Kernel Function

// Matrix multiplication kernel – thread specification
__global__ void MatrixMulKernel(Matrix M, Matrix N, Matrix P)
{
    // 2D Thread Index. In the business of computing P[ty][tx]...
    int tx = threadIdx.x;
    int ty = threadIdx.y;

    // Pvalue will end up storing the value of P[ty][tx]. That is,
    // P.elements[ty * P.width + tx] = Pvalue
    float Pvalue = 0;
    for (int k = 0; k < M.width; ++k) {
        float Melement = M.elements[ty * M.width + k];
        float Nelement = N.elements[k * N.width + tx];
        Pvalue += Melement * Nelement;
    }

    // Write the matrix to device memory;
    // each thread writes one element
    P.elements[ty * P.width + tx] = Pvalue;
}
Step 5: Some Loose Ends

// Allocate a device matrix of same size as M.
Matrix AllocateDeviceMatrix(const Matrix M) {
    Matrix Mdevice = M;
    int size = M.width * M.height * sizeof(float);
    cudaMalloc((void**)&Mdevice.elements, size);
    return Mdevice;
}

// Copy a host matrix to a device matrix.
void CopyToDeviceMatrix(Matrix Mdevice, const Matrix Mhost) {
    int size = Mhost.width * Mhost.height * sizeof(float);
    cudaMemcpy(Mdevice.elements, Mhost.elements, size,
                cudaMemcpyHostToDevice);
}

// Copy a device matrix to a host matrix.
void CopyFromDeviceMatrix(Matrix Mhost, const Matrix Mdevice) {
    int size = Mdevice.width * Mdevice.height * sizeof(float);
    cudaMemcpy(Mhost.elements, Mdevice.elements, size,
                cudaMemcpyDeviceToHost);
}

// Free a device matrix.
void FreeDeviceMatrix(Matrix M) {
    cudaFree(M.elements);
}

void FreeMatrix(Matrix M) {
    free(M.elements);
}
The Common Pattern to CUDA Programming

- **Phase 1**: Allocate memory on the device and copy to the device the data required to carry out computation on the GPU.

- **Phase 2**: Let the GPU crunch the numbers based on the kernel that you defined.

- **Phase 3**: Bring back the results from the GPU. Free memory on the device (clean up...). You’re done.
Timing Your Application

- Timing support – part of the API
  - You pick it up as soon as you include `<cuda.h>`

- Why is good to use
  - Provides cross-platform compatibility
  - Deals with the asynchronous nature of the device calls by relying on events and forced synchronization

- Resolution: milliseconds.
  - From NVIDIA CUDA Library Documentation:
    - Computes the elapsed time between two events (in milliseconds with a resolution of around 0.5 microseconds). If either event has not been recorded yet, this function returns `cudaErrorInvalidValue`. If either event has been recorded with a non-zero stream, the result is undefined.
# Timing Example

Timing a query of device 0 properties

```cpp
#include<iostream>
#include<cuda.h>

int main() {
    cudaEvent_t startEvent, stopEvent;
    cudaEventCreate(&startEvent);
    cudaEventCreate(&stopEvent);
    cudaEventRecord(startEvent, 0);

    cudaDeviceProp deviceProp;
    const int currentDevice = 0;
    if (cudaGetDeviceProperties(&deviceProp, currentDevice) == cudaSuccess)
        printf("Device %d: %s\n", currentDevice, deviceProp.name);
    cudaMemcpy(deviceProp, &deviceProp, sizeof(deviceProp), cudaMemcpyDeviceToHost);

    cudaEventRecord(stopEvent, 0);
    cudaEventSynchronize(stopEvent);
    float elapsedTime;
    cudaEventElapsedTime(&elapsedTime, startEvent, stopEvent);

    std::cout << "Time to get device properties: " << elapsedTime << " ms\n";
    cudaEventDestroy(startEvent);
    cudaEventDestroy(stopEvent);
    return 0;
}
```
The CUDA API wrap up
Memory Layout in CUDA
February 10, 2011
Before We Get Started...

- Last time
  - API related issues
  - Memory allocation, copying, freeing, etc.
  - Simple matrix multiplication example
  - Discussed the typical kernel invocation sequence

- Today
  - Wrap up CUDA API discussion
  - Start discussion of memory hierarchy in NVIDIA’s GPU and CUDA support

- HW
  - HW2: due today at 23:59 PM
  - HW3 has been posted. Due date: 02/15
  - There is assigned reading
Application Programming Interface (API)
~Taking a Step Back~

- CUDA runtime API: exposes a set of extensions to the C language
  - See Section 3.2 and Appendix B of “NVIDIA CUDA C Programming Guide”
    - Keep in mind the 20/80 rule

- It consists of:
  - Language extensions
    - To target portions of the code for execution on the device
  - A runtime library split into:
    - A common component providing built-in vector types and a subset of the C runtime library in both host and device codes
      - Callable both from device and host
    - A host component to control and access one or more devices from the host
      - Callable from the host only
    - A device component providing device-specific functions
      - Callable from the device only
Language Extensions: Variable Type Qualifiers

<table>
<thead>
<tr>
<th>Qualifiers</th>
<th>Memory</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>device</strong> <strong>local</strong></td>
<td>int LocalVar;</td>
<td>local</td>
<td>thread</td>
</tr>
<tr>
<td><strong>device</strong> <strong>shared</strong></td>
<td>int SharedVar;</td>
<td>shared</td>
<td>block</td>
</tr>
<tr>
<td><strong>device</strong></td>
<td>int GlobalVar;</td>
<td>global</td>
<td>grid</td>
</tr>
<tr>
<td><strong>device</strong> <strong>constant</strong></td>
<td>int ConstantVar;</td>
<td>constant</td>
<td>grid</td>
</tr>
</tbody>
</table>

- __device__ is optional when used with __local__, __shared__, or __constant__

- Automatic variables without any qualifier reside in a register
  - Except arrays, which reside in local memory (unless they are small and of known constant size)
Common Runtime Component

- “Common” above refers to functionality that is provided by the CUDA API and is common both to the device and host.

- Provides:
  - Built-in vector types
  - A subset of the C runtime library supported in both host and device codes
**Common Runtime Component:**

**Built-in Vector Types**

  - Structures accessed with `x`, `y`, `z`, `w` fields:
    ```c
    uint4 param;
    int dummy = param.y;
    ```

- `dim3`
  - Based on `uint3`
  - Used to specify dimensions
  - You see a lot of it when defining the execution configuration of a kernel (any component left uninitialized assumes default value 1)

See Appendix B in
“NVIDIA CUDA C Programming Guide”
Common Runtime Component: Mathematical Functions

- `pow`, `sqrt`, `cbrt`, `hypot`
- `exp`, `exp2`, `expm1`
- `log`, `log2`, `log10`, `log1p`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `atan2`
- `sinh`, `cosh`, `tanh`, `asinh`, `acosh`, `atanh`
- `ceil`, `floor`, `trunc`, `round`
- etc.

- When executed on the host, a given function uses the C runtime implementation if available
- These functions are only supported for scalar types, not vector types
Host Runtime Component

- Provides functions available only to the host to deal with:
  - **Device** management (including multi-device systems)
  - **Memory** management
  - **Error** handling

- Examples:
  - cudaHostAlloc, cudaHostFree, cudaMemcpyAsync, etc.

- Quick Remark, Device Management
  - A host thread can invoke device code on only one device
    - Multiple host threads required to run on multiple devices
Host Runtime Component: Memory Management

- Device memory allocation
  - cudaMalloc(), cudaFree()

- Memory copy from host to device, device to host, device to device
  - cudaMemcpy(), cudaMemcpy2D(), cudaMemcpyToSymbol(), cudaMemcpyFromSymbol()

- Memory addressing – returns the address of a device variable
  - cudaGetSymbolAddress()
Device Runtime Component: Mathematical Functions

- Some mathematical functions (e.g. $\sin(x)$) have a less accurate, but faster device-only version (e.g. __sin(x))
  - __pow
  - __log, __log2, __log10
  - __exp
  - __sin, __cos, __tan
End API discussion
…… transitioning into...
Memory Layout discussion
Terminology Review

- **Kernel** = GPU program executed by each parallel thread in a block
- **Block** = a 3D collection of threads that can access the block’s shared memory and can synchronize during execution
- **Grid** = 2D array of blocks of threads that execute a kernel
- **Device** = GPU = set of stream multiprocessors
- **Stream Multiprocessor** (SM) = set of scalar processors & shared memory
- **Scalar Processor** (SP) = also called CUDA processor, shader processor is where instructions are executed

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached</th>
<th>Access</th>
<th>Who</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>One thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A - resident</td>
<td>Read/write</td>
<td>All threads in a block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>Read/write</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>Read</td>
<td>All threads + host</td>
</tr>
</tbody>
</table>

Off-chip means on-device; i.e., slow access time.
GPU: Underlying Hardware
[Tesla C1060]

- The hardware organized as follows:
  - One Stream Processor Array (SPA)…
    - … has a collection of Texture Processor Clusters (TPC, ten of them on C1060) …
      - …and each TPC has three Stream Multiprocessors (SM) …
        - …and each SM is made up of eight Stream or Scalar Processor (SP)
The CUDA Memory Ecosystem
Access Times [Tesla C1060]

- Register – dedicated HW - single cycle
- Shared Memory – dedicated HW - single cycle
- Local Memory – DRAM, no cache - *slow*
- Global Memory – DRAM, no cache - *slow*
- Constant Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Texture Memory – DRAM, cached, 1…10s…100s of cycles, depending on cache locality
- Instruction Memory (invisible) – DRAM, cached
Matrix Multiplication Example, Revisited

- **Purpose**
  - See an example where the use of multiple blocks of threads plays a central role
  - Emphasize the role of the shared memory
  - Emphasize the need for the _syncthreads() function call
Why Revisiting the Matrix Multiplication Example?

● In the naïve first implementation the ratio of arithmetic computation to memory transaction very low
  ● Each arithmetic computation required one fetch from global memory
  ● The matrix M (its entries) is copied from global memory to the device N.width times
  ● The matrix N (its entries) is copied from global memory to the device M.height times

● What matters when you implement the solution of a numerical problem is going through the chain of computations as fast as possible
  ● You don’t get ahead moving data around but only computing things
A Common Programming Pattern
BRINGING THE SHARED MEMORY INTO THE PICTURE

- Local and global memory reside in device memory (DRAM) - much slower access than shared memory

- An advantageous way of performing computation on the device is to partition ("tile") data to take advantage of fast shared memory:
  - Partition data into data subsets (tiles) that each fits into shared memory
  - Handle each data subset (tile) with one thread block by:
    - Loading the tile from global memory to shared memory, using multiple threads to exploit memory-level parallelism
    - Performing the computation on the tile from shared memory; each thread can efficiently multi-pass over any data element
    - Copying results from shared memory back to global memory
Multiply Using Several Blocks

- One block computes one square sub-matrix \( C_{\text{sub}} \) of size Block_Size

- One thread computes one element of \( C_{\text{sub}} \)

- Assume that the dimensions of \( A \) and \( B \) are multiples of Block_Size and square shape
  - Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest

NOTE: Similar example provided in the CUDA Programming Guide 3.2
- Available on the class website
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication func.
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C)
{
    int size;

    // Load A and B to the device
    float* Ad; size = hA * wA * sizeof(float);
    cudaMalloc((void**)&Ad, size);
    cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
    float* Bd; size = wA * wB * sizeof(float);
    cudaMalloc((void**)&Bd, size);
    cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

    // Allocate C on the device
    float* Cd; size = hA * wB * sizeof(float);
    cudaMalloc((void**)&Cd, size);

    // Compute the execution configuration assuming
    // the matrix dimensions are multiples of BLOCK_SIZE
    dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
    dim3 dimGrid( wB/dimBlock.x , hA/dimBlock.y );

    // Launch the device computation
    Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

    // Read C from the device
    cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(Ad);
    cudaFree(Bd);
    cudaFree(Cd);
}
// Device multiplication function called by Mul()
// Compute C = A * B
// wA is the width of A
// wB is the width of B
__global__ void Muld(float* A, float* B, int wA, int wB, float* C) 
{
    // Block index
    int bx = blockIdx.x; // the B (and C) matrix sub-block column index
    int by = blockIdx.y; // the A (and C) matrix sub-block row index

    // Thread index
    int tx = threadIdx.x; // the column index in the sub-block
    int ty = threadIdx.y; // the row index in the sub-block

    // Index of the first sub-matrix of A processed by the block
    int aBegin = wA * BLOCK_SIZE * by;

    // Index of the last sub-matrix of A processed by the block
    int aEnd = aBegin + wA - 1;

    // Step size used to iterate through the sub-matrices of A
    int aStep = BLOCK_SIZE;

    // Index of the first sub-matrix of B processed by the block
    int bBegin = BLOCK_SIZE * bx;

    // Step size used to iterate through the sub-matrices of B
    int bStep = BLOCK_SIZE * wB;

    // The element of the block sub-matrix that is computed
    // by the thread
    float Csub = 0;

    // Loop over all the sub-matrices of A and B required to
    // compute the block sub-matrix
    for (int a = aBegin, b = bBegin;
         a <= aEnd;
         a += aStep, b += bStep) {
        // Shared memory for the sub-matrix of A
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

        // Shared memory for the sub-matrix of B
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Load the matrices from global memory to shared memory;
        // each thread loads one element of each matrix
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];

        // Synchronize to make sure the matrices are loaded
        __syncthreads();

        // Multiply the two matrices together;
        // each thread computes one element
        // of the block sub-matrix
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];

        // Synchronize to make sure that the preceding
        // computation is done before loading two new
        // sub-matrices of A and B in the next iteration
        __syncthreads();

        // Write the block sub-matrix to global memory;
        // each thread writes one element
        int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
        C[c + wB * ty + tx] = Csub;
    }
}
“They have computers, and they may have other weapons of mass destruction.”

Janet Reno, former Attorney General of the United States
Before We Get Started…

- **Last time**
  - Wrapped up CUDA API short overview
  - Started discussion on memory ecosystem on the GPU card
  - Started example of tiled matrix-matrix multiplication
    - Vehicle for introducing the concept of shared memory and thread synchronization

- **Today**
  - Wrap up tiled matrix-matrix multiplication
  - Discuss thread scheduling for execution on the GPU

- **HW**
  - HW4 has been posted. Due date: 02/17, 11:59 PM
  - Please indicate your preference for midterm project on the forum
Here’s Euler, in Diapers…

- Andrew and Hammad the delivery doctors on duty
- 32 Fermi GPUs
- Eight compute nodes, each with two quad core Intel Xeon 5520
- Hopefully operational upon your return from Spring break
- Hopefully you’ll be able to use authentication credentials from Newton to log into Euler
Multiply Using Several Blocks

- One **block** computes one square sub-matrix $C_{\text{sub}}$ of size $\text{Block\_Size}$
- One **thread** computes one element of $C_{\text{sub}}$
- Assume that the dimensions of $A$ and $B$ are multiples of $\text{Block\_Size}$ and square shape
  - Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest

NOTE: Similar example provided in the CUDA Programming Guide 3.2
- Available on the class website
// Thread block size
#define BLOCK_SIZE 16

// Forward declaration of the device multiplication func.
__global__ void Muld(float*, float*, int, int, float*);

// Host multiplication function
// Compute C = A * B
// hA is the height of A
// wA is the width of A
// wB is the width of B
void Mul(const float* A, const float* B, int hA, int wA, int wB, float* C) {
  int size;

  // Load A and B to the device
  float* Ad; size = hA * wA * sizeof(float);
  cudaMalloc((void**)&Ad, size);
  cudaMemcpy(Ad, A, size, cudaMemcpyHostToDevice);
  float* Bd; size = wA * wB * sizeof(float);
  cudaMalloc((void**)&Bd, size);
  cudaMemcpy(Bd, B, size, cudaMemcpyHostToDevice);

  // Allocate C on the device
  float* Cd; size = hA * wB * sizeof(float);
  cudaMalloc((void**)&Cd, size);

  // Compute the execution configuration assuming
  // the matrix dimensions are multiples of BLOCK_SIZE
  dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
  dim3 dimGrid( wB/dimBlock.x , hA/dimBlock.y );

  // Launch the device computation
  Muld<<<dimGrid, dimBlock>>>(Ad, Bd, wA, wB, Cd);

  // Read C from the device
  cudaMemcpy(C, Cd, size, cudaMemcpyDeviceToHost);

  // Free device memory
  cudaFree(Ad);
  cudaFree(Bd);
  cudaFree(Cd);
}
__global__ void Muld(float* A, float* B, int wA, int wB, float* C) {
// Block index
int bx = blockIdx.x; // the B (and C) matrix sub-block column index
int by = blockIdx.y; // the A (and C) matrix sub-block row index

// Thread index
int tx = threadIdx.x; // the column index in the sub-block
int ty = threadIdx.y; // the row index in the sub-block

// Index of the first sub-matrix of A processed by the block
int aBegin = wA * BLOCK_SIZE * by;

// Index of the last sub-matrix of A processed by the block
int aEnd = aBegin + wA - 1;

// Step size used to iterate through the sub-matrices of A
int aStep = BLOCK_SIZE;

// Index of the first sub-matrix of B processed by the block
int bBegin = BLOCK_SIZE * bx;

// Step size used to iterate through the sub-matrices of B
int bStep = BLOCK_SIZE * wB;

// The element of the block sub-matrix that is computed
// by the thread
float Csub = 0;

// Shared memory for the sub-matrix of A
__shared__ float As[BLOCK_SIZE][BLOCK_SIZE];

// Shared memory for the sub-matrix of B
__shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

// Loop over all the sub-matrices of A and B required to
// compute the block sub-matrix
for (int a = aBegin, b = bBegin;
a <= aEnd;
a += aStep, b += bStep) {
    // Load the matrices from global memory to shared memory;
    // each thread loads one element of each matrix
    As[ty][tx] = A[a + wA * ty + tx];
    Bs[ty][tx] = B[b + wB * ty + tx];

    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();

    // Multiply the two matrices together;
    // each thread computes one element
    // of the block sub-matrix
    for (int k = 0; k < BLOCK_SIZE; ++k)
        Csub += As[ty][k] * Bs[k][tx];

    // Synchronize to make sure that the preceding
    // computation is done before loading two new
    // sub-matrices of A and B in the next iteration
    __syncthreads();

    // Write the block sub-matrix to global memory;
    // each thread writes one element
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;
}
Synchronization Function

- It’s a device lightweight runtime API function
  - void __syncthreads();

- Synchronizes all threads in a block (acts as a barrier for all threads of a block)

- Once all threads have reached this point, execution resumes normally

- Used to avoid RAW/WAR/WAW hazards when accessing shared or global memory

- Allowed in conditional constructs only if the conditional is uniform across the entire thread block
The Shared Memory in the Context of the SM Memory Architecture [NVIDIA G80]

- Threads in a Block:
  - Cooperate through data accessible to all of them both in Global Memory and Shared Memory
  - Synchronize at barrier instruction

- Shared Memory is very good
  - Keeps data close to processor (low latency)
  - Minimize trips to global memory
  - Dynamically allocated at the SM level to each Block
  - One of the limiting resources

Courtesy: John Nickolls, NVIDIA
The Three Most Important Parallel Memory Spaces

- **Register**: per-thread basis
  - Private per thread
  - Can spill into local memory (perf. hit)
- **Shared Memory**: per-block basis
  - Shared by threads of the same block
  - Used for: Inter-thread communication
- **Global Memory**: per-application basis
  - Available for use to all threads
  - Used for: Inter-thread communication
  - Also used for inter-grid communication
SM Register File (RF) [Tesla C1060]

- Register File (RF)
  - 64 KB (16,384 four byte words)
  - Provides 4 operands/clock cycle
  - Note: typical CPU has less than 20 registers per core

- TEX pipe can also read/write RF
  - 3 SMs share 1 TEX

- Global Memory Load/Store pipe can also read/write RF
Programmer View of Register File

- Number of 32 bit registers in one SM:
  - 8K registers in each SM in G80
  - 16K on Tesla C1060
  - 32K on Tesla C2050

- This is an implementation decision, not part of CUDA

- Registers are dynamically partitioned across all Blocks assigned to the SM

- Once assigned to a Block, these registers are NOT accessible by threads in other Blocks

- Each thread in the same Block only access registers assigned to itself

Possible per-block partitioning scenarios of the RF available on the SM
If each Block has 16x16 threads and each thread uses 20 registers, how many threads can run on each SM?

- Each Block requires 20*256 = 5120 registers
- \(16384 = 3 \times 5120 + \text{change}\)
- So, three blocks can run on an SM as far as registers are concerned

What if each thread increases the use of registers by 2?

- Each Block now requires 22*256 = 5632 registers
- \(16384 < 16896 = 5632 \times 3\)
- Only two Blocks can run on an SM, about 33% reduction of parallelism!!!

Example shows why understanding the underlying hardware is essential if you want to squeeze performance out of parallelism.
Dynamic partitioning gives more flexibility to compilers/programmers

- One can run a smaller number of threads that require many registers each, or run a large number of threads that require few registers each
  - This allows for finer grain threading than traditional CPU threading models.

- The compiler can tradeoff between instruction-level parallelism and thread level parallelism
Constant Memory

- This comes handy when all threads use the same *constant* value in their computation
  - Example: $\pi$, some spring force constant, $e=2.7173$, etc.

- Constants are stored in DRAM but cached on chip
  - There is a limited amount of L1 cache per SM
  - Might run into slow access if for example have a large number of constants used to compute some complicated formula (might overflow the cache…)

- A constant value can be broadcast to all threads in a Warp
  - Extremely efficient way of accessing a value that is common for all threads in a Block!
  - When all threads in a warp read the same constant memory address this is as fast as a register
Example, Use of Constant Memory
[For compute capability 2.0 (GTX480, C2050) – due to use of “printf”]

```c
#include <stdio.h>

// Declare the constant device variable outside the body of any function
__device__ __constant__ float dansPI;

// Some dummy function that uses the constant variable
__global__ void myExample() {
    float circum = 2.f*dansPI*threadIdx.x;
    printf("Hello thread %d, Circ=%5.2f\n", threadIdx.x, circum);
}

int main(int argc, char **argv) {
    float somePI = 3.141579f;
    cudaMemcpyToSymbol(dansPI, &somePI, sizeof(float));
    myExample<<<1, 16>>>();
    cudaThreadSynchronize();
    return 0;
}
```

Hello thread 0, Circ= 0.00
Hello thread 1, Circ= 6.28
Hello thread 2, Circ=12.57
Hello thread 3, Circ=18.85
Hello thread 4, Circ=25.13
Hello thread 5, Circ=31.42
Hello thread 6, Circ=37.70
Hello thread 7, Circ=43.98
Hello thread 8, Circ=50.27
Hello thread 9, Circ=56.55
Hello thread 10, Circ=62.83
Hello thread 11, Circ=69.11
Hello thread 12, Circ=75.40
Hello thread 13, Circ=81.68
Hello thread 14, Circ=87.96
Hello thread 15, Circ=94.25
Memory Issues Not Addressed Yet...

- Not all global memory accesses are equivalent
  - How can you optimize memory accesses?
  - Very relevant question

- Not all shared memory accesses are equivalent
  - How can optimize shared memory accesses?
  - Moderately relevant questions

- To do justice to these topics we’ll need to talk first about scheduling threads for execution
  - Coming up next…
Thread Execution Scheduling

- GPU Architecture Paradigm: Single Instruction Multiple Data (SIMD)
  - CUDA perspective: Single Program Multiple Threads

- What’s the overall software (application) development model?
  - CUDA integrated CPU + GPU application C program
    - Serial C code executes on CPU
    - Parallel Kernel C code executes on GPU thread blocks

```
Grid 0
GPU Parallel Kernel
KernelA<<< nBlkA, nTidA >>>(args);

CPU Serial Code

Grid 1
GPU Parallel Kernel
KernelB<<< nBlkB, nTidB >>>(args);

CPU Serial Code
```

17
CUDA Thread Block

[We already know this...]

- In relation to a Block, the programmer decides:
  - Block size: from 1 to 512 concurrent threads
  - Block dimension (shape): 1D, 2D, or 3D
  - # of threads in each dimension

- Threads have thread idx numbers within Block
- Threads within Block share data and may synchronize while each is doing its work
- Thread program uses thread idx to select work and address shared data
- Beyond the concept of thread idx we brought into the picture the concept of thread id and how to compute a thread id based on the thread index
GeForce-8 Series HW Overview
Scheduling on the HW

- Grid is launched on the SPA
- Thread Blocks are serially distributed to all the SMs
  - Potentially >1 Thread Block per SM
- Each SM launches Warps of Threads
- SM schedules and executes Warps that are ready to run
- As Warps and Thread Blocks complete, resources are freed
  - SPA can launch next Block[s] in line

NOTE: Two levels of scheduling:
- For running [desirably] a large number of blocks on a small number of SMs (30/16/14/etc.)
- For running up to 24 (or 32, on Tesla C1060) warps of threads on the 8 SPs available on each SM
Thread Scheduling/Execution

- Each Thread Block is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model

- Warps are the basic scheduling units in SM

- If 3 blocks are processed by an SM and each Block has 256 threads, how many Warps are managed by the SM?
  - Each Block is divided into 256/32 = 8 Warps
  - There are 8 * 3 = 24 Warps
  - At any point in time, only *one* of the 24 Warps will be selected for instruction fetch and execution.
SM Warp Scheduling

- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected

- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp in G80

- How is this relevant?
  - Suppose your code has one global memory access every six simple instructions
  - Then, a minimum of 17 Warps are needed to fully tolerate 400-cycle memory latency:

\[
400/(6 \times 4) = 16.6667 \Rightarrow 17 \text{ Warps}
\]
SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - From instruction L1 cache
  - Into any instruction buffer slot

- Issue one “ready-to-go” warp instruction per 4 cycles
  - From any warp - instruction buffer slot
  - Operand scoreboard used to prevent hazards

- Issue selection based on round-robin/age of warp

- SM broadcasts the same instruction to 32 Threads of a Warp
Scoreboarding

- Used to determine whether a thread is ready to execute

- A **scoreboard** is a table in hardware that tracks
  - Instructions being fetched, issued, executed
  - Resources (functional units and operands) needed by instructions
  - Which instructions modify which registers

- Old concept from CDC 6600 (1960s) to separate memory and computation
Scoreboarding from Example

- Consider three separate instruction streams: warp1, warp3 and warp8

<table>
<thead>
<tr>
<th>Warp</th>
<th>Current Instruction</th>
<th>Instruction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>42</td>
<td>Computing</td>
</tr>
<tr>
<td>Warp 3</td>
<td>95</td>
<td>Computing</td>
</tr>
<tr>
<td>Warp 8</td>
<td>11</td>
<td>Operands ready to go</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Mary Hall, U-Utah
Scoreboarding from Example

- Consider three separate instruction streams: warp1, warp3 and warp8

<table>
<thead>
<tr>
<th>Warp</th>
<th>Current Instruction</th>
<th>Instruction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>42</td>
<td>Ready to write result</td>
</tr>
<tr>
<td>Warp 3</td>
<td>95</td>
<td>Computing</td>
</tr>
<tr>
<td>Warp 8</td>
<td>11</td>
<td>Computing</td>
</tr>
</tbody>
</table>

Schedule at time $k+1$
Scoreboarding

- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - Status becomes “ready” after the needed values are deposited
  - Prevents hazards
  - Cleared instructions are eligible for issue

- Decoupled Memory/Processor pipelines
  - Any thread can continue to issue instructions until scoreboarding prevents issue

Instruction:

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>TB1 W1</td>
<td>TB2 W1</td>
<td>TB3 W1</td>
<td>TB3 W2</td>
<td>TB2 W1</td>
<td>TB1 W1</td>
<td>TB1 W2</td>
<td>TB1 W3</td>
</tr>
<tr>
<td>TB2, W1 stall</td>
<td>TB3, W2 stall</td>
<td>TB1, W1 stall</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

TB = Thread Block, W = Warp
Granularity Considerations

[NOTE: Specific to Tesla C1060]

- For Matrix Multiplication, should I use 8X8, 16X16 or 32X32 tiles?

  - For 8X8, we have 64 threads per Block. Since each Tesla C1060 SM can manage up to 1024 threads, it could take up to 16 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!

  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1024 threads, it can take up to 4 Blocks and achieve full capacity unless other resource considerations overrule.

  - For 32X32, we have 1024 threads per Block. This is not an option anyway (we need less then 512 per block)

- NOTE: this type of thinking should be invoked for your target hardware (from where the need for auto-tuning software…)
ILP vs. TLP Example

- Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 20 registers.
- Also, assume global loads have 400 cycles.
  - 3 Blocks can run on each SM.

- If a Compiler can use two more registers to change the dependence pattern so that 8 independent instructions exist (instead of 4) for each global memory load.
  - Only two blocks can now run on each SM.
  - However, one only needs 400 cycles/(8 instructions * 4 cycles/instruction) \( \approx 13 \) Warps to tolerate the memory latency.
  - Two Blocks have 16 Warps. The performance can be actually higher!
Summing It Up…

- When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity.

- The threads of a block execute concurrently on one SM, and multiple blocks (up to 8) can execute concurrently on one SM.

- When a thread block finishes, a new block is launched on the vacated SM.
A Word on HTT

[Detour: slide 1/2]

- The traditional host processor (CPU) may stall due to a cache miss, branch misprediction, or data dependency

- Hyper-threading Technology (HTT): an Intel-proprietary technology used to improve parallelization of computations (doing multiple tasks at once)

- For each processor core that is physically present, the operating system addresses two virtual processors, and shares the workload between them when possible.

- HT works by duplicating certain sections of the processor—those that store the architectural state—but not duplicating the main execution resources.
  - This allows a hyper-threading processor to appear as two "logical" processors to the host operating system, allowing the operating system to schedule two threads or processes simultaneously.

- Similar to the use of multiple warps on the GPU to hide latency
  - The GPU has an edge, since it can handle simultaneously up to 32 warps (on Tesla C1060)
Streamlining SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors as a reply to AMD's 3DNow!

- SSE contains 70 new instructions

**Example**

- Old school, adding two vectors. Corresponds to four x86 FADD instructions in the object code

  \[
  \begin{align*}
  &\text{vec\_res.x} = v1.x + v2.x; \\
  &\text{vec\_res.y} = v1.y + v2.y; \\
  &\text{vec\_res.z} = v1.z + v2.z; \\
  &\text{vec\_res.w} = v1.w + v2.w; 
  \end{align*}
  \]

- SSE pseudocode: a single 128 bit 'packed-add' instruction can replace the four scalar addition instructions

  \[
  \begin{align*}
  &\text{movaps xmm0, address-of-v1 ; xmm0} = v1.w \mid v1.y \mid v1.x \\
  &\text{addps xmm0, address-of-v2 ; xmm0} = v1.w+v2.w \mid v1.y+v2.y \mid v1.x+v2.x \text{ movaps address-of-vec\_res,xmm0}
  \end{align*}
  \]
ME964
High Performance Computing for Engineering Applications

Execution Scheduling in CUDA
Revisiting Memory Issues in CUDA
February 17, 2011

“Computers are useless. They can only give you answers.”
Pablo Picasso
Before We Get Started…

- **Last time**
  - Wrapped up tiled matrix-matrix multiplication using shared memory
    - Shared memory used to reduce some of the pain associated with global memory accesses
  - Discussed thread scheduling for execution on the GPU

- **Today**
  - Wrap up discussion about execution scheduling on the GPU
  - Discuss global memory access issues in CUDA

- **Other issues**
  - HW3 due tonight at 11:59 PM
    - Use Learn@UW drop-box to submit homework
    - Note that timing should be done as shown on slide 30 of the pdf posted for lecture 02/08
    - Building of CUDA code in Visual Studio: issues related to “include” of a file that hosts the kernel definition (either don’t include, or force linking)
  - HW4 was posted. Due date: 02/22, 11:59 PM
  - Please indicate your preference for midterm project on the forum
Thread Scheduling/Execution

- Each Thread Block is divided in 32-thread Warps
  - This is an implementation decision, not part of the CUDA programming model

- Warps are the basic scheduling units in SM

- Example (draws on figure at right):
  - Assume 2 blocks are processed by an SM and each Block has 512 threads, how many Warps are managed by the SM?
    - Each Block is divided into \( \frac{512}{32} = 16 \) Warps
    - There are \( 16 \times 2 = 32 \) Warps
    - At any point in time, only *one* of the 32 Warps will be selected for instruction fetch and execution.
SM Warp Scheduling

- SM hardware implements zero-overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected

- 4 clock cycles needed to dispatch the same instruction for all threads in a Warp on C1060

- How is this relevant?
  - Suppose your code has one global memory access every six simple instructions
  - Then, a minimum of 17 Warps are needed to fully tolerate 400-cycle memory latency:

  \[
  \frac{400}{(6 \times 4)} = 16.6667 \Rightarrow 17 \text{ Warps}
  \]
SM Instruction Buffer – Warp Scheduling

- Fetch one warp instruction/cycle
  - From instruction L1 cache
  - Into any instruction buffer slot

- Issue one “ready-to-go” warp instruction per 4 cycles
  - From any warp - instruction buffer slot
  - Operand scoreboard used to prevent hazards

- Issue selection based on round-robin/age of warp

- SM broadcasts the same instruction to 32 Threads of a Warp
Scoreboarding

- Used to determine whether a thread is ready to execute

- A **scoreboard** is a table in hardware that tracks
  - Instructions being fetched, issued, executed
  - Resources (functional units and operands) needed by instructions
  - Which instructions modify which registers

- Old concept from CDC 6600 (1960s) to separate memory and computation
Scoreboarding from Example

Consider three separate instruction streams: warp1, warp3 and warp8

<table>
<thead>
<tr>
<th>Warp</th>
<th>Current Instruction</th>
<th>Instruction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>42</td>
<td>Computing</td>
</tr>
<tr>
<td>Warp 3</td>
<td>95</td>
<td>Computing</td>
</tr>
<tr>
<td>Warp 8</td>
<td>11</td>
<td>Operands ready to go</td>
</tr>
</tbody>
</table>

Mary Hall, U-Utah
Scoreboarding from Example

- Consider three separate instruction streams: warp1, warp3 and warp8

<table>
<thead>
<tr>
<th>Warp</th>
<th>Current Instruction</th>
<th>Instruction State</th>
</tr>
</thead>
<tbody>
<tr>
<td>Warp 1</td>
<td>42</td>
<td>Ready to write result</td>
</tr>
<tr>
<td>Warp 3</td>
<td>95</td>
<td>Computing</td>
</tr>
<tr>
<td>Warp 8</td>
<td>11</td>
<td>Computing</td>
</tr>
</tbody>
</table>

Mary Hall, U-Utah
Scoreboarding

- All register operands of all instructions in the Instruction Buffer are scoreboarded
  - Status becomes “ready” after the needed values are deposited
  - Prevents hazards
  - Cleared instructions are eligible for issue
- Decoupled Memory/Processor pipelines
  - Any thread can continue to issue instructions until scoreboarding prevents issue

Instruction: 1 2 3 4 5 6 1 2 1 2 3 4 7 8 1 2 1 2 3 4

TB = Thread Block, W = Warp
Granularity Considerations

[NOTE: Specific to Tesla C1060]

- For Matrix Multiplication, should I use 8X8, 16X16 or 32X32 tiles?
  
  - For 8X8, we have 64 threads per Block. Since each Tesla C1060 SM can manage up to 1024 threads, it could take up to 16 Blocks. However, each SM can only take up to 8 Blocks, only 512 threads will go into each SM!

  - For 16X16, we have 256 threads per Block. Since each SM can take up to 1024 threads, it can take up to 4 Blocks unless other resource considerations overrule.

  - For 32X32, we have 1024 threads per Block. This is not an option anyway (we need less then 512 per block)

- NOTE: this type of thinking should be invoked for your target hardware (from where the need for auto-tuning software…)}
ILP vs. TLP Example

- Assume that a kernel has 256-thread Blocks, 4 independent instructions for each global memory load in the thread program, and each thread uses 20 registers.
- Also, assume global loads have an associated overhead of 400 cycles.
  - 3 Blocks can run on each SM.
- If a Compiler can use two more registers to change the dependence pattern so that 8 independent instructions exist (instead of 4) for each global memory load.
  - Only two blocks can now run on each SM.
  - However, one only needs 400 cycles/(8 instructions *4 cycles/instruction) ≈ 13 Warps to tolerate the memory latency.
  - Two Blocks have 16 Warps. The performance can be actually higher!
Summing It Up…

- When a CUDA program on the host CPU invokes a kernel grid, the blocks of the grid are enumerated and distributed to SMs with available execution capacity.

- The threads of a block execute concurrently on one SM, and multiple blocks (up to 8) can execute concurrently on one SM.

- When a thread block finishes, a new block is launched on the vacated SM.
A Word on HTT

[Detour: slide 1/2]

- The traditional host processor (CPU) may stall due to a cache miss, branch misprediction, or data dependency.

- Hyper-threading Technology (HTT): an Intel-proprietary technology used to improve parallelization of computations.

- For each processor core that is physically present, the operating system addresses two virtual processors, and shares the workload between them when possible.

- HT works by duplicating certain sections of the processor—those that store the architectural state—but not duplicating the main execution resources.
  - This allows a hyper-threading processor to appear as two "logical" processors to the host operating system, allowing the operating system to schedule two threads or processes simultaneously.

- Similar to the use of multiple warps on the GPU to hide latency.
  - The GPU has an edge, since it can handle simultaneously up to 32 warps (on Tesla C1060).
Streaming SIMD Extensions (SSE) is a SIMD instruction set extension to the x86 architecture, designed by Intel and introduced in 1999 in their Pentium III series processors as a reply to AMD's 3DNow!

- SSE contains 70 new instructions

**Example**

- Old school, adding two vectors. Corresponds to four x86 FADD instructions in the object code

```plaintext
vec_res.x = v1.x + v2.x;
vec_res.y = v1.y + v2.y;
vec_res.z = v1.z + v2.z;
vec_res.w = v1.w + v2.w;
```

- SSE pseudocode: a single 128 bit 'packed-add' instruction can replace the four scalar addition instructions

```plaintext
movaps xmm0, address-of-v1 ; xmm0=v1.w | v1.y | v1.x
addps xmm0, address-of-v2 ; xmm0=v1.w+v2.w | v1.y+v2.y | v1.x+v2.x
movaps address-of-vec_res, xmm0
```
Finished Discussion Execution Scheduling

Revisit Memory Accessing Topic
Important Point

- In GPU computing, memory operations are perhaps most relevant in determining the overall efficiency (performance) of your code.
The Device Memory Ecosystem

[Quick Review]

The significant change from 1.x to 2.x device capability was the caching of the local and global memory accesses

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location on/off chip</th>
<th>Cached</th>
<th>Access</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>On</td>
<td>n/a</td>
<td>R/W</td>
<td>1 thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Local</td>
<td>Off</td>
<td>†</td>
<td>R/W</td>
<td>1 thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On</td>
<td>n/a</td>
<td>R/W</td>
<td>All threads in block</td>
<td>Block</td>
</tr>
<tr>
<td>Global</td>
<td>Off</td>
<td>†</td>
<td>R/W</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
<tr>
<td>Constant</td>
<td>Off</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
<tr>
<td>Texture</td>
<td>Off</td>
<td>Yes</td>
<td>R</td>
<td>All threads + host</td>
<td>Host allocation</td>
</tr>
</tbody>
</table>

† Cached only on devices of compute capability 2.x.
Global Memory and Memory Bandwidth

- These change from card to card. On Tesla C1060:
  - 4 GB in GDDR3 RAM
  - Memory clock speed: 800 MHz
  - Memory interface: 512 bits
  - Peak Bandwidth: $800 \times 10^6 \times (512/8) \times 2 = 102.4$ GB/s

- Effective bandwidth of your application: very important to gauge
  - Formula, effective bandwidth ($B_r$ - bytes read, $B_w$ - bytes written)
    
    \[ \text{Effective bandwidth} = \frac{(B_r + B_w)/10^9}{\text{time}} \]
  
  - Example: kernel copies a $2048 \times 2048$ matrix from global memory, then copies matrix back to global memory. Does it in a certain amount of “time” [seconds]
    
    \[ \text{Effective bandwidth} = \frac{(2048^2 \cdot 4 \cdot 2)/10^9}{\text{time}} \]
  
  - 4 above comes from four bytes per float, 2 from the fact that the matrix is both read from and written to the global memory. The $10^9$ used to get an answer in GB/s.
Global Memory

- The global memory space is not cached on Tesla C1060 (1.3)
  - Very important to follow right access pattern to get maximum memory throughput

- Two aspects of global memory access are relevant when fetching data into shared memory and/or registers
  - The layout of the access to global memory (the pattern of the access)
  - The size/alignment of the data you try to fetch from global memory
The Memory Access Layout

- The important concept here is that of “coalesced memory access”

- The basic idea:
  - Suppose each thread in a half-warp accesses a global memory address for a load operation at some point in the execution of the kernel
  - These threads can access global memory data that is either (a) neatly grouped or (b) scattered all over the place
  - Case (a) is called a “coalesced memory access”; if you end up with (b) this will adversely impact the overall program performance

- Analogy
  - Can send one semi truck on six different trips to bring back each time a bundle of wood
  - Alternatively, can send semi truck to one place and get it back fully loaded with wood
Global Memory Access
Compute Capability 1.3

- A global memory request for a warp is split in two memory requests, one for each half-warp
- The following 5-stage protocol is used to determine the memory transactions necessary to service all threads in a half-warp

**Stage 1**: Find the memory segment that contains the address requested by the lowest numbered active thread. The memory segment size depends on the size of the words accessed by the threads:
  - 32 bytes for 1-byte words,
  - 64 bytes for 2-byte words,
  - 128 bytes for 4-, 8- and 16-byte words.

**Stage 2**: Find all other active threads whose requested address lies in the same segment

**Stage 3**: Reduce the transaction size, if possible:
  - If the transaction size is 128 bytes and only the lower or upper half is used, reduce the transaction size to 64 bytes;
  - If the transaction size is 64 bytes (originally or after reduction from 128 bytes) and only the lower or upper half is used, reduce the transaction size to 32 bytes.

**Stage 4**: Carry out the transaction and mark the serviced threads as inactive.

**Stage 5**: Repeat until all threads in the half-warp are serviced.
Memory Issues in CUDA
February 22, 2011

“Once a new technology rolls over you, if you’re not part of the steamroller, you’re part of the road.”
Stewart Brand
Before We Get Started…

- **Last time**
  - Wrapped up discussion about execution scheduling on the GPU
  - Discussed global memory access issues in CUDA

- **Today**
  - Examples, global memory accesses
  - Discuss shared memory accesses in CUDA
  - A couple of comments on HW4

- **Other issues**
  - HW4 due tonight at 11:59 PM
    - Use Learn@UW drop-box to submit homework
  - HW5 posted, due on March 1, 11:59 PM
  - Please take a look at the latest version of the syllabus, has been updated recently
  - Thursday, Feb. 24
    - TAs Toby Heyn and Arman Pazouki will provide an overview of two Midterm Project topics: Discrete Element Method (DEM) and Collision Detection, respectively
  - Wednesday, Feb 23: no office hours – I will be traveling (leaving on Wd at noon, returning Th evening)
Global Memory Access
Compute Capability 1.3

- A global memory request for a warp is split in two memory requests, one for each half-warp.
- The following 5-stage protocol is used to determine the memory transactions necessary to service all threads in a half-warp.

  **Stage 1**: Find the memory segment that contains the address requested by the lowest numbered active thread. The memory segment size depends on the size of the words accessed by the threads:
  - 32 bytes for 1-byte words,
  - 64 bytes for 2-byte words,
  - 128 bytes for 4-, 8- and 16-byte words.

  **Stage 2**: Find all other active threads whose requested address lies in the same segment.

  **Stage 3**: Reduce the transaction size, if possible:
  - If the transaction size is 128 bytes and only the lower or upper half is used, reduce the transaction size to 64 bytes;
  - If the transaction size is 64 bytes (originally or after reduction from 128 bytes) and only the lower or upper half is used, reduce the transaction size to 32 bytes.

  **Stage 4**: Carry out the transaction and mark the serviced threads as inactive.

  **Stage 5**: Repeat until all threads in the half-warp are serviced.
Examples

- Look at an example that deals with 32 bit words (4 bytes)
- This is the case when handling integers or floats
- Various scenarios are going to be considered to illustrate how the two factors (layout of access & alignment) come into play when accessing global memory
- Note that when handling 32 bit words, “segment size” represents 128 byte data chunks (all aligned at multiples of 128)
  - In what follows, a different color is associated with each 128 byte memory segment
  - In other words, two rows of the same color represent a 128-byte aligned segment
Example: Scenario 1

- Coalesced access in which all threads but one access the corresponding word in a segment.

- This access pattern results in a single 64-byte transaction, indicated by the red rectangle.

- Note that even though one word is not requested, all data in the segment are fetched.

- If accesses by threads were permuted within this segment, still one 64-byte transaction would be performed on Tesla C1060.
Example: Scenario 2

- Sequential threads in a half warp access memory that is sequential but not aligned with the segments.

- Given that the addresses fall within a 128-byte segment, a single 128-byte transaction is performed on Tesla C1060.
Example: Scenario 3

- A half warp accesses memory that is sequential but split across two 128-byte segments. Note that the request spans two different memory segments.

- On Tesla C1060, two transactions are performed: one 64-byte transaction and one 32-byte transaction result.
Example: Scenario 4

- Strided access to global memory, as shown in the code snippet below:

  ```c
  __global__ void strideCopy(float *odata, float* idata, int stride)
  {
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
  }
  ```

- Although a stride of 2 above results in a single transaction, note that half the elements in the transaction are not used and represent wasted bandwidth.
Example: Scenario 4

[Cndtd.]

- Strided access to global memory, as shown in the code snippet below:

```c
__global__ void strideCopy(float *odata, float* idata, int stride)
{
    int xid = (blockIdx.x*blockDim.x + threadIdx.x)*stride;
    odata[xid] = idata[xid];
}
```

- As the stride increases, the effective bandwidth decreases until the point where 16 transactions are issued for the 16 threads in a half warp, as shown in the plot.

![Copy with Stride](image)
Looking Beyond Tesla C1060

- Tesla C1060 represents compute capability 1.3. How about other compute capabilities?

- Look at the same example as before
  - Accessing floats or integers for global memory transactions

- Example 1: access is aligned and sequential

![Diagram showing aligned and sequential addresses and threads with compute capability and memory transactions details.]
Looking Beyond Tesla C1060

- **Example 2:** Aligned but non-sequential

  ![Aligned and non-sequential diagram]

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute capability:</td>
<td>1.0 and 1.1</td>
<td>1.2 and 1.3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Cached</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 128</td>
<td>1 x 64B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 192</td>
<td>8 x 32B at 224</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 224</td>
<td>1 x 128B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Example 3:** Misaligned and sequential

  ![Misaligned and sequential diagram]

<table>
<thead>
<tr>
<th>Addresses:</th>
<th>96</th>
<th>128</th>
<th>160</th>
<th>192</th>
<th>224</th>
<th>256</th>
<th>288</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads:</td>
<td>0</td>
<td>...</td>
<td>31</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute capability:</td>
<td>1.0 and 1.1</td>
<td>1.2 and 1.3</td>
<td>2.0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory transactions:</td>
<td>Uncached</td>
<td>Cached</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 x 32B at 128</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 160</td>
<td>1 x 64B at 192</td>
<td>1 x 128B at 128</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 192</td>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8 x 32B at 224</td>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1 x 32B at 256</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Think about this…

- Say you use in your program complex data constructs that could be organized using C-structures

- Based on what we learned today, how is it more advantageous to store data in global memory?
  - Alternative A: as an array of structures
  - Alternative B: as a structure of arrays
## Technical Specifications and Features

### [Short Detour]

### Compute Capability

<table>
<thead>
<tr>
<th>Technical Specifications</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum x- or y-dimension of a grid of thread blocks</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum x- or y-dimension of a block</td>
<td>512</td>
<td>1024</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum z-dimension of a block</td>
<td>64</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Warp size</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>24</td>
<td>32</td>
<td>48</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>768</td>
<td>1024</td>
<td>1536</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8 K</td>
<td>16 K</td>
<td>32 K</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum amount of shared memory per multiprocessor</td>
<td>16 KB</td>
<td>48 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Number of shared memory banks</td>
<td>16</td>
<td>32</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Amount of local memory per thread</td>
<td>16 KB</td>
<td>512 KB</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Constant memory size</td>
<td>64 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Cache working set per multiprocessor for constant memory</td>
<td>8 KB</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Maximum number of instructions per kernel</td>
<td>2 million</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Feature Support

(Unlisted features are supported for all compute capabilities)

<table>
<thead>
<tr>
<th>Feature Supporting in Section</th>
<th>1.0</th>
<th>1.1</th>
<th>1.2</th>
<th>1.3</th>
<th>2.x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer atomic functions operating on 32-bit words in global memory</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>yes</td>
</tr>
<tr>
<td>Integer atomic functions operating on 64-bit words in global memory</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Integer atomic functions operating on 32-bit words in shared memory</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Warp vote functions</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Double-precision floating-point numbers</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>Floating-point atomic addition operating on 32-bit words in global and shared memory</td>
<td>No</td>
<td></td>
<td></td>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>__ballot() (Section B.12)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>__threadfence_system() (Section B.5)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>__syncthreads_count(), __syncthreads_and(), __syncthreads_or() (Section B.6)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Surface functions (Section B.9)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Vector Reduction with Bank Conflicts
(assume 1024 vector entries stored in shared memory; one block of 512 threads carries out the reduction)

Array elements (floats)
Discuss Shared Memory Issues
Shared Memory: Syntax & Semantics

- You can statically declare shared memory like in the code snippet below:

```c
__global__ void coalescedMultiply(float *a, float* b, float *c,
int N)
{
__shared__ float aTile[TILE_DIM][TILE_DIM];

int row = blockIdx.y * blockDim.y + threadIdx.y;
int col = blockIdx.x * blockDim.x + threadIdx.x;
float sum = 0.0f;
aTile[threadIdx.y][threadIdx.x] = a[row*TILE_DIM+threadIdx.x];
for (int i = 0; i < TILE_DIM; i++) {
    sum += aTile[threadIdx.y][i]* b[i*N+col];
}
c[row*N+col] = sum;
}
```

- **NOTE:** this makes the variable aTile visible to all threads in each block, and **only** to those threads
- The thread that executes the kernel above sees the aTile declaration and understands that all its brother-threads in the block are going to see it too. They will together share this variable
- The same thread, when it sees the variable “row” it understands that it has sole ownership of this variable (variable stored most likely in a register)
Shared Memory
[Tesla C1060]

- Each SM has 16 KB of Shared Memory
  - Physically organized as 16 banks of 4 byte words
  - Note that shared memory can store less data than the registers (16 vs. 64 KB)

- The 16 banks of the Shared Memory are organized like benches in a movie theater
  - You have 256 rows of benches. Each row has 16 benches, in each bench you can “seat” a family of four (bytes). Note that a bank represents a column of benches in the movie theater

- CUDA uses Shared Memory as shared storage visible to all threads in a thread block
  - All threads in the block have read & write access
Shared Memory: Transaction Rules

- For compute capability 1.x (Newton works with 1.3), the shared memory is organized as 16 banks.
- Each warp access is split in two → Only half warp accesses shared memory at a time.

- For compute capability 2.x (Fermi), the shared memory is organized as 32 banks.
- There is no splitting of the warp, all threads in a warp attempt to access shared memory simultaneously.
Q: Is 16K of Shared Memory Enough? Revisit the Matrix Multiplication Example

- One **block** computes one square sub-matrix \( C_{\text{sub}} \) of size \( \text{Block}_\text{Size} \)
- One **thread** computes one element of \( C_{\text{sub}} \)
- Assume that the dimensions of \( A \) and \( B \) are multiples of \( \text{Block}_\text{Size} \) and square shape
  - Doesn’t have to be like this, but keeps example simpler and focused on the concepts of interest
Matrix Multiplication: Shared Memory Usage

- Each Block requires $2 \times \text{WIDTH}^2 \times 4$ bytes of shared memory storage
  - For $\text{WIDTH} = 16$, each BLOCK requires 2KB, up to 8 Blocks can fit into the Shared Memory of an SM
  - Since each SM can only take 1024 threads, each SM can only take 4 Blocks of 256 threads each
  - Shared memory size is not a limitation for our implementation of the Matrix Multiplication
Shared Memory Architecture

- Common sense observation: in a parallel machine many threads access memory at the same time
  - To service more than one thread, memory is divided into banks
  - Essential to achieve high bandwidth

- Each bank can service one address per cycle
  - The shared memory can service as many simultaneous accesses as it has banks

- Multiple simultaneous accesses to a bank result in a bank conflict
  - Conflicting accesses are serialized
Bank Addressing Examples

- No Bank Conflicts
  - Linear addressing stride == 1

- No Bank Conflicts
  - Random 1:1 Permutation
Bank Addressing Examples

- 2-way Bank Conflicts
  - Linear addressing stride == 2

- 8-way Bank Conflicts
  - Linear addressing stride == 8
Shared Memory Bank Conflicts

- If there are no bank conflicts
  - Shared memory access is as fast as registers
  - Latency is roughly 100x lower than global memory latency

- Share memory access, the fast case:
  - If all threads of a half-warp access different banks, there is no bank conflict
  - If all threads of a half-warp access an identical address for a fetch operation, there is no bank conflict (broadcast)

- Share memory access, the slow case:
  - Bank Conflict: multiple threads in the same half-warp access the same bank
  - Must serialize the accesses
  - Cost = max # of simultaneous accesses to a single bank
How addresses map to banks on Tesla C1060

- Each bank has a bandwidth of 32 bits per clock cycle
- Successive 32-bit words are assigned to successive banks
- Tesla C1060 has 16 banks
  - Bank you work with = address % 16
  - Same as the number of threads in a half-warp
    - NOTE: There is no such thing as bank conflicts between threads belonging to different half-warps; this issue only relevant for threads from within a single half-warp
Linear Addressing

- Given:
  ```
  __shared__ float sharedM[256];
  float foo = sharedM[baseIndex + s * threadIdx.x];
  ```

- This is bank-conflict-free if `s` shares no common factors with the number of banks
  - 16 on C1060, so `s` must be odd
The Math Beyond Bank Conflicts

- We are in a half-warp, and the question is if thread \( t_1 \) and thread \( t_2 > t_1 \) might access the same bank of shared memory.
- Let \( b \) be the base of the array (the “shareM” pointer on previous slide).
- How should you not choose \( s \)?

\[
\begin{align*}
  b + st_2 &= b + st_1 + 16k, \quad \text{for some positive integer } k \\
  0 &< t_2 - t_1 \leq 15
\end{align*}
\]

\[
\begin{align*}
  16k &= s(t_2 - t_1) \\
  0 &< t_2 - t_1 \leq 15
\end{align*}
\]

- If \( s=2 \), take \( k=1 \), and then any threads \( t_1 \) and \( t_2 \) which are eight apart satisfy the condition above and will have a bank conflict ([0,8], [1,9], etc.) – two way conflict.
- If \( s=4 \), take \( k=2 \), any threads \( t_1 \) and \( t_2 \) which are four apart will have a bank conflict ([0,4,8,12], [1,5,9,13], etc.) – four way conflict.
- **NOTE**: you can’t get a bank conflict is \( s \) is odd (no quartet \( k, s, t_1, t_2 \) satisfies the bank conflict condition above). So take stride \( s=1,3,5, \) etc.
Data types and bank conflicts

- No conflicts below if `shared` is a 32-bit data type:
  
  ```
  foo = shared[baseIndex + threadIdx.x]
  ```

- But not if the data type is smaller
  - 4-way bank conflicts:
    ```
    __shared__ char shared[];
    foo = shared[baseIndex + threadIdx.x];
    ```
  
  - 2-way bank conflicts:
    ```
    __shared__ short shared[];
    foo = shared[baseIndex + threadIdx.x];
    ```
Structs and Bank Conflicts

- Struct assignments compile into as many memory accesses as there are struct members:

  ```
  struct vector { float x, y, z; }
  struct myType {
      float f;
      int c;
  }

  __shared__ struct vector vectors[64];
  __shared__ struct myType myTypes[64];
  ```

- This has no bank conflicts for vector; struct size is 3 words
  - 3 accesses per thread, contiguous banks (no common factor with 16)

  ```
  struct vector v = vectors[baseIndex + threadIdx.x];
  ```

- This has 2-way bank conflicts for my Type; (2 accesses per thread)

  ```
  struct myType m = myTypes[baseIndex + threadIdx.x];
  ```
Common Array Bank Conflict Patterns 1D

- Each thread loads 2 elements into shared memory:
  - 2-way-interleaved loads result in 2-way bank conflicts:

```c
int tid = threadIdx.x;
shared[2*tid] = global[2*tid];
shared[2*tid+1] = global[2*tid+1];
```

- This makes sense for traditional CPU threads, locality in cache line usage and reduced sharing traffic.
  - Not in shared memory usage where there is no cache line effects but banking effects
A Better Array Access Pattern

- Each thread loads one element in every consecutive group of blockDim elements.

\[
\text{shared}[\text{tid}] = \text{global}[\text{tid}]; \\
\text{shared}[\text{tid} + \text{blockDim}.x] = \text{global}[\text{tid} + \text{blockDim}.x];
\]
Vector Reduction **without** Bank Conflicts
(assume 1024 vector entries stored in shared memory; one block of 512 threads carries out the reduction)
Discrete Element Method

Midterm Project: Option 2
Motivation

- **Industries:**
  - Mining
  - Food & Pharmaceutics
  - Film & Game
  - etc.

- **Problem examples:**
  - Collapsing Silos
  - Mars Rover
  - etc.
Discrete Element Method

- Collision detection determines pairs of colliding bodies
- Contact forces computed based on constitutive relation (spring-damper model)
- Requires small time-steps
- Newton’s Second Law used to compute accelerations
- Numerical integration (e.g., Velocity Verlet) used to compute velocity, position of all bodies
Discrete Element Method

Loop
$t_{start}$ to $t_{end}$

Particle Initialization → Collision Detection → Contact Force Calculation → Newton’s 2nd Law → Velocity and Position Analysis → Output Data

Next time step
DEM

- Spatial Subdivision

- 2 particles \( \mathbf{r}_i; \mathbf{r}_j \)

\[ \mathbf{r}_{ij} = \mathbf{r}_i - \mathbf{r}_j \]

- If \( r_{ij} \leq d \)

\[ \delta_{ij} = d - r_{ij} \]

\[ \mathbf{n}_{ij} = \frac{\mathbf{r}_{ij}}{r_{ij}} \]

- Otherwise no collis
Collision detection code will be provided to you

Input: Arrays of sphere positions and radii

Output: Array of collision data
Contact force components
- **normal**
- tangential

Four different categories:
- Continuous potential models
- **Linear viscoelastic models**
- Non-linear viscoelastic models
\[ v_{ij} = v_i - v_j \]

\[ v_{nij} = (v_{ij} \cdot n_{ij})n_{ij} \]

\[ m_{eff} = \frac{m_im_j}{m_i + m_j} \]

- Normal Force \( F_{nij} \) computed as:

\[ F_{nij} = f \left( \frac{\delta_{ij}}{d} \right) \left( k_n \delta_{ij}n_{ij} - \gamma_n m_{eff}v_{nij} \right) \]

\( k_n \) – spring stiffness

\( \gamma_n \) – damping coefficient
DEM

- Force on one particle is the sum of its contact forces and gravity:

\[ \mathbf{F}_{i}^{\text{tot}} = m_i \mathbf{g} + \sum_j \left( \mathbf{F}_{nj} \right) \]

- Calculation acceleration:

\[ \mathbf{F}_{i}^{\text{tot}} = m_i \mathbf{a}_i \rightarrow \mathbf{a}_i = \frac{\mathbf{F}_{i}^{\text{tot}}}{m_i} \]
DEM

- Use explicit numerical integration methods like Explicit Euler or Velocity Verlet Integration

- Explicit Euler:

\[ \mathbf{r}_i(t + \Delta t) = \mathbf{r}_i(t) + \mathbf{v}_i(t) \Delta t \]

\[ \mathbf{v}_i(t + \Delta t) = \mathbf{v}_i(t) + \mathbf{a}_i(t) \Delta t \]
Parallelism

- Parallel collision detection (provided)
- (Per-contact): Compute collision forces
- (Per-body): Reduction to resultant force per body
- (Per-body): Solution of Newton’s Second Law, time integration
Example

- 1 million spheres
- 0.5 sec long simulation
- ~12,000 sec computational time
- GPU
Suggested Code Structure

- **Class ParticleSystem**
  - void initializeSim()
  - void performCD()
  - void computeForces()
  - void integrate()
  - void getGPUdata()
  - void outputState()
void initializeSim()

- Set initial conditions of all bodies
- Copy state data from host to device

void performCD()

- Call GPU CD function (provided) to determine pairs of colliding spheres
- Returns array of contact_data structs
  - data members: objectIdA, objectIdB
void computeForces()

- Compute contact force for each contact
- Compute resultant force acting on each body
- Compute and add reaction force for contact with boundary planes

void integrate()

- Compute acceleration of each body
- Update velocity and position of each body
void getGPUdata()

- Copy state data back to host

void outputState()

- Output sphere positions and radii to a text file
int main(int argc, char* argv[]) {
    float t_curr=0.0f;
    float t_end=1.0f;
    float h=0.00005f;
    ParticleSystem *psystem = new ParticleSystem(…);
    psystem->initializeSim();
    while(t_curr<=t_end) {
        psystem->performCD();
        psystem->computeForces();
        psystem->integrate();
        t_curr+=h;
    }
    delete psystem;
    return 0;
}
Other Tips (Force computation)

1. Compute force for each contact with one thread per contact
   - Store key-value array with body ID as key, force as value
   - Note each contact should create a force on two bodies

2. Sort by key (body ID)
   - thrust::sort_by_key(…)

Other Tips (Force computation)

3. Sum all forces acting on a single body
   - thrust::reduce_by_key(…)
   - One thread per entry in output, copy to appropriate place in net force list

4. Add gravity force to each body’s net force
   - One thread per body
Other Tips (Force computation)

5. Contact with planes
   - Assume infinite planes
   - A plane is defined by a point \((p)\) and normal direction \((N)\)
   - One thread per sphere (at position \(r)\)
     - Compute \(d = N \cdot (r - p)\)
     - Contact if \(d < \text{radius}\)
     - Compute force as before, add to net force
Parallel Collision Detection
Overview

- **Method 1: Brute Force**
  - Easier implementation
  - $O(N^2)$ Complexity

- **Method 2: Parallel Binning**
  - More involved
  - $O(N)$ Complexity
Brute Force Approach

- Three Steps:
  - Run preliminary pass to understand the memory requirements by figuring out the number of contacts present
  - Allocate on the device the required amount of memory to store the desired collision information
  - Run actual collision detection and populate the data structure with the information desired
Step 1: Search for contacts

- Create on the device an array of unsigned integers, equal in size to the number $N$ of bodies in the system
  - Call this array $dB$, initialize all its entries to zero
  - Array $dB$ to store in entry $j$ the number of contacts that body $j$ will have with bodies of higher index
    - If body 5 collides with body 9, no need to say that body 9 collides with body 5 as well

Do in parallel, one thread per body basis

```plaintext
for body $j$, loop from $k=j+1$ to $N$
  if bodies $j$ and $k$ collide, $dB[j] += 1$
endloop
eendDo```

24
Step 1, cont.
Step 2: Parallel Scan Operation

- Allocate memory space for the collision information
  - Step 2.1: Define first a structure that might help (this is not the most efficient approach, but we’ll go along with it…)
    ```c
    struct collisionInfo {
      float3 r_A;
      float3 r_B;
      float3 normal;
      unsigned int indxA;
      unsigned int indxB;
    }
    ```
  - Step 2.2: Run a parallel inclusive prefix scan on $d_B$, which gets overwritten during the process
    ![Prefix Scan Diagram]
  - Step 2.3: Based on the last entry in the $d_B$ array, which holds the total number of contacts, allocate from the host on the device the amount of memory required to store the desired collision information. To this end you’ll have to use the size of the “struct” collisionInfo. Call this array $dCollisionInfo$. 

26
Step 3

- Parallel pass on a per body basis (one thread per body – similar to step 1)
  - Thread $j$ (associated with body $j$), computes its number of contacts as $dB[j]-dB[j-1]$, and sets the variable $contactsProcessed=0$
  - Thread $j$ runs a loop for $k=j+1$ to $N$
  - If body $j$ and $k$ are in contact, populate entry $dCollisionInfo[dB[j-1]+contactsProcessed]$ with this contact’s info and increment $contactsProcessed++$
  - Note: you can break out of the look after $k$ as soon as $contactsProcessed==dB[j]-dB[j-1]$
Concluding Remarks, Brute Force

- Level of effort for discussed approach
  - Step 1, $O(N^2)$ (checking body against the rest of the bodies)
  - Step 2: prefix scan is $O(N)$
  - Step 3, $O(N^2)$ (checking body against the rest of the bodies, basically a repetition of Step 1)

- No use of the atomicAdd, which is a big performance bottleneck

- Numerous versions of this can be contrived to improve the overall performance
  - Not discussed here for this brute force idea, rather moving on to a different approach altogether, called “binning”
Parallel Binning
Collision Detection: Binning

- Very similar to the idea presented by LeGrand in GPU-Gems 3
- 30,000 feet perspective:
  - Do a spatial partitioning of the volume occupied by the bodies
    - Place bodies in bins (cubes, for instance)
  - Do a brute force for all bodies that are touching a bin
  - Taking the bin to be small means that chances are you’ll not have too many bodies inside any bin for the brute force stage
  - Taking the bins to be small means you’ll have a lot of them
Example: 2D collision detection, bins are squares

Body 4 touches bins A4, A5, B4, B5
Body 7 touches bins A3, A4, A5, B3, B4, B5, C3, C4, C5
In proposed algorithm, bodies 4 and 7 will be checked for collision several times: by threads associated with bin A4, A5, B4.
CD: Binning

The method draws on

- Parallel Sorting
  - Implemented with $O(N)$ work (NVIDIA tech report, also SDK particle simulation demo)

- Parallel Exclusive Prefix Scan
  - Implemented with $O(N)$ work (NVIDIA SDK example)

The extremely fast binning operation for the simple convex geometries that we’ll be dealing with

- On a rectangular grid it is very easy to figure out where the CM (center of mass) of a simple convex geometry will land
Binning: The Method

- **Notation Use:**
  - $N$ – number of bodies
  - $N_b$ – number of bins
  - $N_a$ – number of active bins
  - $p_i$ - body $i$
  - $b_j$ – bin $j$

- **Stage 1: body parallel**
  - Parallelism: one thread per body
  - Kernel arguments: grid definition
    - $x_{\text{min}}, x_{\text{max}}, y_{\text{min}}, y_{\text{max}}, z_{\text{min}}, z_{\text{max}}$
    - $h_x, h_y, h_z$ (grid size in 3D)
    - Can also be placed in constant memory, will end up cached
Stage 1: # Bin-Body Contacts

- **Purpose:** find the number of bins touched by each body in the problem
- **Store results in the “T”, array of N integers**
- **Key observation:** it’s easy to bin bodies

![Diagram showing a grid with numbered cells, representing bins, and an example of storing results in the T-array.]
Stage 2: Parallel Exclusive Scan

- Run a parallel exclusive scan on the array $T$
  - Save to the side the number of bins touched by the last body, needed later, otherwise overwritten by the scan operation. Call this value $b_{last}$
  - In our case, if you look carefully, $b_{last} = 6$

- Complexity of Stage: $O(N)$, based on parallel scan algorithm of Harris, see GPU Gem 3 and CUDA SDK

- Purpose: determine the amount of entries $M$ needed to store the indices of all the bins touched by each body in the problem
Stage 3: Determine body-&-bin association

- Allocate an array \( B \) of \( M \) pairs of integers.
  - The key (first entry of the pair), is the bin index
  - The value (second entry of pair) is the body that touches that bin
  - Stage is parallel, on a per-body basis
Stage 4: Sort

- In parallel, run a radix sort to order the B array according to the keys
- Work load: $O(N)$
Stage 5-8: Find # of Bodies/Bin

- Purpose: Find the number of bodies per each active bin and the location of the active bins in B.

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

B-array

| A1 | A2 | A2 | A3 | A3 | A3 | A4 | A4 | A5 | A5 | B1 | B1 |

The Value

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>...</th>
</tr>
</thead>
</table>

C-array

| 1  | 2  | 3  | 2  | 2  | 2  | ... |

The Key

| A1 | A2 | A3 | A4 | A5 | B1 | ... |
Stage 5-8: Find # of Bodies/Bin

- **Stage 5:** Host allocates $C$, an array of unsigned integers of length $N_b$, on device and initializes it by the largest possible integer.
  Run in parallel, on a per bin basis, find the start location of each sequence. Write the location to the corresponding entry of $C$-value.
- **Stage 6:** Run parallel radix sort to sort $C$-value.
- **Stage 7:** Find the location of the first inactive bin.
  - To save memory, $C$ can be resized.
- **Stage 8:** Find out $nbpb_k$ (number of bodies per bin $k$) and store it in entry $k$ of $C$, as the key associated with this pair.
Stage 9: Sort C for Load Balancing

- Do a parallel radix sort on the array C based on the key
- Purpose: balance the load during next stage
- NOTE: this stage might or might not be carried out if the load balancing does not offset the overhead associated with the sorting job
- Effort: $O(N^a)$
Stage 10: Investigate Collisions in each Bin

- Carried out in parallel, one thread per bin

To store information generated during this stage, host needs to allocate an unsigned integer array $D$ of length $N_b$:
- Array $D$ stores the number of actual contacts occurring in each bin
- $D$ is in sync with (linked to) $C$, which in turn is sync with (linked to) $B$

Parallelism: one thread per bin:
- Thread $k$ reads the pair key-value in entry $k$ of array $C$
- Thread $k$ reads does rehearsal for brute force collision detection
- Outcome: the number $s$ of active collisions taking place in a bin
  - Value $s$ stored in $k^{th}$ entry of the $D$ array
Stage 10, details...

- In order to carry out this stage you need to keep in mind how C is organized, which is a reflection of how B is organized.

The drill: thread 0 relies on info at C[0], thread 1 relies on info at C[1], etc.

Let’s see what thread 2 (goes with C[2]) does:
- Read the first 2 bodies that start at offset 6 in B.
  - These bodies are 4 and 7, and as B indicates, they touch bin A4.
  - Bodies 4 and 7 turn out to have 1 contact in A4, which means that entry 2 of D needs to reflect this.
Stage 10, details…

- In order to carry out this stage you need to keep in mind how C is organized, which is a reflection of how B is organized.

- The drill: thread 0 relies on info at C[0], thread 1 relies on info at C[1], etc.

- Let’s see what thread 2 (goes with C[2]) does:
  - Read the first 2 bodies that start at offset 6 in B.
    - These bodies are 4 and 7, and as B indicates, they touch bin A4.
    - Bodies 4 and 7 turn out to have 1 contact in A4, which means that entry 2 of D needs to reflect this.

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>3</td>
<td>2</td>
<td>5</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>4</td>
<td>7</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>

A1 A2 A2 A3 A3 A3 A4 A4 A5 A5 B1 B1 ...
```

```
<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>6</th>
<th>8</th>
<th>10</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>
```

Bin vs. Body Touching This Bin

- Shows up 2 since there are two bodies (4 & 7) in bin with offset 6 (A4)

```
\[\text{D-array (Length: } N_b)\]

\[\begin{array}{cccccccc}
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
A1 & A2 & A4 & A5 & B1 & A3 & ... \\
\end{array}\]

Bin offset in B and number of bodies touching that bin
Stage 10, details

- Brute Force CD rehearsal
  - Carried out to understand the memory requirements associated with collisions in each bin
    - Finds out the total number of contacts owned by a bin
  - Key question: which bin does a contact belong to?
    - Answer: It belongs to bin containing the CM of the Contact Volume (CMCV)
Stage 10, Comments

- Two bodies can have multiple contacts, which is ok

- Easy to define the CMCV for two spheres, two ellipsoids, and a couple of other simple geometries
  - In general finding CMCV might be tricky
    - Notice picture below, CM of 4 is in A5, CM of 7 is in B4 and CMCV is in A4
  - Finding the CMCV is the subject of the so called “narrow phase collision detection”
    - It’ll be simple in our case since we are going to work with simple geometry primitives
Stage 11: Exclusive Prefix Scan

- Save to the side the number of contacts in the last bin (last entry of $D$) $d_{\text{last}}$
  - Last entry of $D$ will get overwritten

```
0 1 2 3 4 5 ...
0 0 1 0 0 0 ...
(A1) (A2) (A4) (A5) (B1) (A3) ...
```

- Run parallel exclusive prefix scan on $D$:

```
0 1 2 3 4 5 ...
0 0 0 1 1 1 ...
(A1) (A2) (A4) (A5) (B1) (A3) ...
```

- Total number of actual collisions:

$$N_c = D[N_b] + d_{\text{last}}$$
Stage 12: Populate Array E

- From the host, allocate on the device memory for array $E$
  - Array $E$ stores the required collision information: normal, two tangents, etc.
  - Number of entries in the array: $N_c$ (see previous slide)

- In parallel, on a per bin basis (one thread/bin):
  - Populate the $E$ array with required info

- Not discussed in greater detail, this is just like Stage 7, but now you have to generate actual collision info (stage 7 was the rehearsal)

- Thread for A4 will generate the info for contact “c”
- Thread for C2 will generate the info for “i” and “d”
- Etc.
Stage 12, details

- B, C, D required to populate array E with collision information

- C and B are needed to compute the collision information
- D is needed to understand where the collision information will be stored in E
Stage 12, Comments

- In this stage, parallelism is on a per bin basis
  - Each thread picks up one entry in the array C
  - Based on info in C you pick up from B the bin id and bodies touching this bin
  - Based on info in B you run brute force collision detection
    - You run brute force CD for as long as necessary to find the number of collisions specified by array D
    - Note that in some cases there are no collisions, so you exit without doing anything
  - As you compute collision information, you store it in array E
Parallel Binning: Summary of Stages

- Stage 1: Find number of bins touched by each body, populate $T$ (body parallel)
- Stage 2: Parallel exclusive scan of $T$ (length of $T$: $N$)
- Stage 3: Determine body-to-bin association, populate $B$ (body parallel)
- Stage 4: Parallel sort of $B$ (length of $B$: $M$)
- Stage 5: Find active bins, populate $C$-value (bin parallel)
- Stage 6: Parallel sort of $C$-value (bin parallel)
- Stage 7: Find and remove inactive bins (bin parallel)
- Stage 8: Find number of bodies per active bin (bin parallel)
- Stage 9: Parallel sort of $C$ for load balancing (length of $C$: $N_a$)
Parallel Binning: Summary of Stages

- Stage 10: Determine # of collisions in each bin, store in D (bin parallel)
- Stage 11: Parallel prefix scan of D (length of D: Na)
- Stage 12: Run collision detection and populate E with required info (bin parallel)
Parallel Binning – Concluding Remarks

Some unaddressed issues:

- How big should the bins be?
- Can you have bins of variable size?
- How should the computation be organized such that memory access is not trampled upon?
- Can you eliminate stage 5 (the binary search) and use info from the sort of stage 4?
- Do you need stage 9 (sort for load balancing)?
- Does it make sense to have a second sort for load balancing (as we have right now)?
Parallel Binning – Concluding Remarks

- At the cornerstone of the proposed approach is the fact that one can very easily find the bins that a simple geometry intersects
  - First, it’s easy to bin bodies
  - Second, if you find a contact, it’s easy to allocate it to a bin and avoid double counting

- Method scales very well on multiple GPUs
  - Each GPU handles a subvolume of the volume occupied by the bodies

- CD algorithm relies on two key algorithms: sorting and prefix scan
  - Both these operations require O(N) on the GPU
  - NOTE: a small number of basic algorithms used in many applications.
Outlining Midterm Projects

Topic 3: GPU-based FEA
Topic 4: GPU Direct Solver for Sparse Linear Algebra

March 01, 2011

“The real problem is not whether machines think but whether men do.”
B. F. Skinner
Before We Get Started…

● Last time
  ○ Midterm Project topics 1 and 2
    ● Discrete Element Method on the GPU. Area coordinator: Toby Heyn
    ● Collision Detection on the GPU. Area coordinator: Arman Pazouki

● Today
  ○ Midterm Project topics 3 and 4
    ● Finite Element Method on the GPU. Area coordinators: Prof. Suresh and Naresh Khude
    ● Sparse direct solver on the GPU (Cholesky). Area coordinator: Dan Negrut

● Midterm Project Related Issues
  ○ Midterm Project is due on 04/13 at 11:59 PM (use Learn@UW drop-box)
  ○ Intermediate report due on 03/22 at 11:59 PM (use the same Learn@UW drop-box)
  ○ Each area coordinator
    ● Will provide a test problem for you to test your GPU implementation
    ● Will also assist you with questions related to the non-programming aspects (the “theory”) behind the topic you chose
  ○ You can continue your Midterm Project (MP) and have it become your Final Project (FP)
    ● In this case you will be expected to show how the FP implementation is superior to your MP implementation

● Other issues
  ○ HW5 due tonight at 11:59 PM
    ● Use Learn@UW drop-box to submit homework

Midterm Project is due on 04/13 at 11:59 PM (use Learn@UW drop-box)
Finite Element Analysis on the GPU?

Krishnan Suresh
suresh@engr.wisc.edu
Associate Professor
Finite Element Analysis

- Computer simulation of engineering models
- **Physics:**
  - Structural, thermal, fluid, …
- **Mode:**
  - Static, modal, transient
  - Linear, non-linear, multi-physics
Why GPU?

Hours or even days of CPU time.

[Gordon; JPL]
Can one exploit graphics programmable units (GPU) to speed-up Finite Element analysis?
Structural Static FEA

\[ K = \sum K_e \]

\[ f = \sum f_e \]

\[ Ku = f \]
FEA: Variations

\[ K = \sum K_e \]
\[ f = \sum f_e \]
\[ Ku = f \]
FEA: Challenges

\[ K = \sum K_e \]
\[ f = \sum f_e \]
\[ Ku = f \]

1. Accuracy
2. Automation
3. Speed
Typical Bottleneck

\[ K = \sum K_e \]
\[ f = \sum f_e \]
\[ Ku = f \]
GPU & Engineering Analysis

Discretization
- Data: Small b-rep (+)
- Logic: Complex (-)
- Threads: Few (-)

Not a good candidate for GPU!?
Element Stiffness

- Data: O(N) (+/-)
- Logic: Simple (+)
- Threads: N (+)

$K_e$

$\mathbf{f}_e$

Hex 2$^{nd}$ Order

Hex Hybrid
**Stiffness: Hex 2nd Order**

\[ K_e = [ ]_{(M,M)} \]

- 8 Corners~100 Bytes Data (x y z)
- 27 Nodes~ M = 81 DOF (u v w)
- \( k_{ij} \sim \text{Gaussian integration} \)
- 30 flops

\[ Flops \approx N(15M^2) \]

\[ N = 200000, M = 81 \]

\[ T_{CPU} \approx 4 \text{ sec} \]
Typical Bottleneck

Model → Discretize → Element Stiffness → Assemble/Solve

\[ K = \sum K_e \]
\[ f = \sum f_e \]
\[ Ku = f \]
Direct vs. Iterative

\[ Ku = f \]

- **Direct**
  
  \[ K = LDL^T \]
  
  \[ u = L^{-1}D^{-1}L^T f \]

- **Iterative**

  \[ u^{i+1} = u^i + B(f - Ku^i) \]

  \(B : \text{Preconditioner of } K\)

  (GPU Variation: Assembly-free)

Note: Nvidia offers CuBLAS-3 dense matrix library
Direct Sparse on GPU (1)

Graphics Processing Units as Fast Co-Processors for Scientific Computing

<table>
<thead>
<tr>
<th>CPU</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel Pentium D, 2 x 3.4 GHz</td>
<td>GeForce 8800 GTX, CUBLAS</td>
</tr>
</tbody>
</table>

Matthias-M. Christen
Forschungsgruppe Burkhart,
Dept. Informatik, Uni Basel

(2006)
Direct Sparse on GPU (1)

GPU acceleration for a sparse linear solver

- Parallel direct solver „PARDISO“ (developed by Olaf Schenk) for large, sparse systems of linear equations
  - Uses supernode techniques for LU decomposition, i.e. columns of equal sparsity pattern below diagonal block are grouped together
    ⇒ BLAS-3 routines could be used
  - PARDISO is integrated into Intel MKL
- Minimally invasive GPU integration: replace BLAS-3 routine calls in LU decomposition by GPU equivalents if enough ops

\[ K u = f \]
Direct Sparse on GPU (1)

\[ Ku = f \]
Direct Sparse on GPU (2)

\[ Ku = f \]

Accelerating the ANSYS Direct Sparse Solver with GPUs

Géraud P. Krawezik
Acceleware Corp.
Calgary, AB, Canada

Gene Poole
ANSYS Inc.
Canonsburg, PA, USA

The program was tested with a NVIDIA Tesla C1060 compute card, based on a GT200 GPU connected to 4 GB of memory. The host contains two Intel Xeon 5335 processors
$Ku = f$
Concurrent Number Cruncher: An Efficient Sparse Linear Solver on the GPU

(2008)

Luc Buatois¹, Guillaume Caumon², and Bruno Lévy³

- Jacobi preconditioned conjugate gradient
- ATI GPU
- Speed-up 3.5.
Iterative Sparse on GPU (2)

Efficient Sparse Matrix-Vector Multiplication on CUDA

Nathan Bell* and Michael Garland†

December 11, 2008

- Double precision real world SpMv
  - CPU (2.3 GHz Dual Xeon): 1 GFLOPS
  - GPU (GTX 280): 16 GFLOPS
  - Speedup ~ 16
FEA/GPU Class Projects?

1. Complete < 6 weeks
2. Important (publishable)
3. Pilot code
1. GPU Friendly **Preconditioners** for *Thin Structures*
   - Research papers
   - OpenCL and ViennaCL Pilot Code

2. Topology Optimization
   - Research papers
   - CUDA code

3. Others
   - Can discuss …
Thin Structure?
Thin Structure?

Large K
Preconditioners?

\[ Ku = f \]
\[ u^{i+1} = u^i + B(f - Ku^i) \]
\[ B : \text{Preconditioner of } K \]

- **Iterative Methods:**
  - GPU methods available for \( K\u \)
  - Typical preconditioners: simple Jacobi, …

- **Poor preconditioner … slow convergence**

- **Objective:**
  - GPU friendly preconditioner for thin structures
GPU-friendly Preconditioners for Efficient 3-D Finite Element Analysis of Thin Structures

Vikalp Mishra and Krishnan Suresh
Basic Idea

Restriction (via dual-representation)

Prolongation (via dual-representation)

1-D Coarse Mesh (via dual-representation)
Algorithm:

Finite Element Assembly

Finite Element Assembly

\[ K_{3D}; f_{3D} \]

\[ u_{3D} = 0 \]

\[ \varepsilon_{\text{prev}} = 1.0 \]

Iterate N times (CG, …)

\[ u_{3D} \]

\[ \varepsilon_{\text{prev}} = \|r_{3D}\| \]

Correction

\[ \varepsilon_{3D} = P^r K_{DR}^{-1} P r_{3D} \]

Yes

No

\[ \|r_{3D}\| < \varepsilon_{\text{prev}}? \]

Residual

No

Yes

\[ r_{3D} = f_{3D} - K_{3D} u_{3D} \]

Other Preconditioners (Optional)

\[ u_{3D} = u_{3D} + c_{3D} \]

Terminate

Yes

No

\[ \|r_{3D}\| < \varepsilon_{\text{desired}}? \]
Why Preconditioner?
Why Double Precision?
How Expensive is Preconditioner?
GPU Friendly

Speed-up without Preconditioner

Speed-up with Preconditioner
1. **GPU Friendly Preconditioners** for *Thin Structures*  
   - Research papers  
   - OpenCL and ViennaCL Pilot Code

2. **Topology Optimization**  
   - Research papers  
   - CUDA code

3. **Others**  
   - Can discuss …
Topology Optimization

Stiffest topology for a given volume?
Where to remove material?

$$\min_{\Omega \subset D} J$$

$$|\Omega| = V_0$$

V = 50%

[Sigmund 2001]

Multi Objective + Topology Optimization = MOTO

$$\min_{\Omega \subset D} \{ J, V_0 \}$$
Demo

Matlab code [www.ersl.wisc.edu](http://www.ersl.wisc.edu)
Pareto Optimal Designs

Purely pareto optimal
Comparison

<table>
<thead>
<tr>
<th>$\nu$</th>
<th>Proposed Method</th>
<th>SIMP</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9</td>
<td>J=46.33; #FEA=18</td>
<td>J=46.22; #FEA=44</td>
</tr>
<tr>
<td>0.8</td>
<td>J=49.8; #FEA=34</td>
<td>J=49.6; #FEA=135</td>
</tr>
<tr>
<td>0.7</td>
<td>J=54.3; #FEA=50</td>
<td>J=54.8; #FEA=280</td>
</tr>
<tr>
<td>0.6</td>
<td>J=61.77; #FEA=64</td>
<td>J=62.2; #FEA=422</td>
</tr>
<tr>
<td>0.5</td>
<td>J=73.13; #FEA=82</td>
<td>J=73; #FEA=720</td>
</tr>
</tbody>
</table>

$D$
3-D
Multi-grid Topology Optimization on the GPU
(IDETC conf. 2011)
Motivation for Topic 4: Sparse Direct Solver

- Solve the following optimization problem

\[
\begin{cases}
\text{minimize} & q(x) = \frac{1}{2} x^T M x - x^T k \\
\text{subject to} & D x = c
\end{cases}
\]

- \(x \in \mathbb{R}^n\) – the set of variables over which the cost function needs to be minimized
- \(M \in \mathbb{R}^{n \times n}\) – given to you, it is a symmetric and positive matrix.
- \(k \in \mathbb{R}^n\) – given to you
- \(D \in \mathbb{R}^{m \times n}\) – given to you, it is a sparse rectangular matrix
- \(c \in \mathbb{R}^m\) – given to you
- The above quadratic optimization problem with equality constraints requires the solution of the following linear system:

\[
\begin{bmatrix}
M_{n \times n} & D_{m \times n}^T \\
D_{m \times n} & 0_{m \times m}
\end{bmatrix}
\begin{bmatrix}
x \\
\lambda
\end{bmatrix}_{n+m} =
\begin{bmatrix}
k \\
c
\end{bmatrix}_{n+m}
\]

- NOTE: \(\lambda\) is the Lagrange Multiplier associated with the optimization problem at hand
Nomenclature
&
Simplifying Assumptions

• The coefficient matrix $\begin{bmatrix} M & D^T \\ D & 0 \end{bmatrix}$ is called the KKT matrix

• The matrix $E = DM^{-1}D^T$ is called the Schur complement of the KKT matrix

• Simplifying assumptions for ME964 project
  
  – The matrix $M$ is diagonal
  – The matrix $D$ does not have in any of its rows more than two entries and a row could sometime have only one entry
  – The matrix $D$ is full row rank (the equality constraints are independent)

• Midterm Project 4 has to do with finding the Cholesky factorization $E = LL^T$ on the GPU and then solving a system of the form $E\lambda = b$ using forward/backward substitution
The Schur Complement Problem in Multi-Body Dynamics Applications

- This problem is also encountered in a large number of other applications
- One such application is multi-body dynamics
  - Here the $x_n$ are the set of body accelerations and $\lambda_m$ are the constraint forces in the joints of the mechanical system
Formulation Framework

- Position: \( \mathbf{r}_i = [x_i, y_i, z_i]^T \)

- Orientation: Euler parameters, \( \mathbf{p}_i = [e_i^0, e_i^1, e_i^2, e_i^3]^T \)

- Translational Velocity: \( \dot{\mathbf{r}}_i = [\dot{x}_i, \dot{y}_i, \dot{z}_i]^T \)

- Angular velocities \( \overline{\omega}_i = [\overline{\omega}_i^x, \overline{\omega}_i^y, \overline{\omega}_i^y]^T \)
Constrained Equations of Motion

\[ \Phi(r, p, t) = 0 \]

\[ \Phi_\eta(r, p, t)\dot{r} + \Phi_\rho(r, p, t)\dot{\omega} = -\Phi_t(r, p, t) \]

\[ \Phi_\eta(r, p, t)\ddot{r} + \Phi_\rho(r, p, t)\ddot{\omega} = \tau(\dot{r}, \dot{\omega}, r, p, t) \]

\[
\begin{bmatrix}
M & 0 \\
0 & \bar{J}
\end{bmatrix}
\begin{bmatrix}
\ddot{r} \\
\ddot{\omega}
\end{bmatrix}
+ \begin{bmatrix}
\Phi_\eta^T(r, p, t) \\
\Phi_\rho^T(r, p, t)
\end{bmatrix} \lambda = \begin{bmatrix}
F(\dot{r}, \dot{\omega}, r, p, t) \\
\hat{n}(\dot{r}, \dot{\omega}, r, p, t)
\end{bmatrix}
\]
Numerical Solution of the Newton-Euler Constrained Equations of Motion

- One has to solve a set of Differential Algebraic Equations (DAEs) to find the time evolution of a mechanical system.

- Most often the numerical solution of the DAEs requires the solution of a linear system of the form:

\[
\begin{bmatrix}
M & 0 & \Phi_\eta^T \\
0 & \bar{J} & \Phi_\rho^T \\
\Phi_\eta & \Phi_\rho & 0
\end{bmatrix}
\begin{bmatrix}
\ddot{r} \\
\dot{\omega} \\
\lambda
\end{bmatrix}
= 
\begin{bmatrix}
\dot{F} \\
\dot{\mathbf{n}} \\
\tau
\end{bmatrix}
\]
Approach Followed

- First solve the "Reduced System" for $\lambda$:

$$
\begin{bmatrix}
  \Phi_\eta & \Phi_\rho
\end{bmatrix}
\begin{bmatrix}
  M^{-1} & 0 \\
  0 & \bar{J}^{-1}
\end{bmatrix}
\begin{bmatrix}
  \Phi_\eta^T \\
  \Phi_\rho^T
\end{bmatrix}
\lambda = b
$$

- Then recover accelerations

$$
\ddot{r} = M^{-1} (F - \Phi_\eta^T \lambda) \\
\ddot{\omega} = \bar{J}^{-1} (\hat{n} - \Phi_\rho^T \lambda)
$$
Iterative Solution of the Reduced System

- Define positive definite Reduced Matrix $E$

\[
E = \begin{bmatrix}
\Phi_\eta & \Phi_\rho
\end{bmatrix}
\begin{bmatrix}
M^{-1} & 0 \\
0 & J^{-1}
\end{bmatrix}
\begin{bmatrix}
\Phi_\eta^T \\
\Phi_\rho^T
\end{bmatrix}
\]

- Preconditioned Conjugate Gradient
  - requires computation at time $t_n$ of $E_n \lambda^{(k)}$
  - requires preconditioning: $E_{old} \lambda = b$
A thread is associated with each body

We’ll look at how thread 9 does its share of work to compute $\mathbf{e}^3$
How **Thread-9** Does its Work

**S1.** Compute *reaction forces* acting on me:

\[
F_9^C = (\Phi_9^3)^T \lambda_3 + (\Phi_9^5)^T \lambda_5 + (\Phi_9^6)^T \lambda_6
\]

**S2.** Compute my *constraint acceleration*

\[
a_9^C = M_9^{-1} \cdot F_9^C
\]

**S3.** Project my *constraint acceleration*

\[
\Pi_9^3 = \Phi_9^3 \cdot a_9^C \quad \Pi_9^5 = \Phi_9^5 \cdot a_9^C \quad \Pi_9^6 = \Phi_9^6 \cdot a_9^C
\]

Finally,

\[
e^3 = \Pi_9^3 + \Pi_{12}^3
\]
## Iteration Operation Count for **Body 9** (Thread-9)

<table>
<thead>
<tr>
<th>Step</th>
<th>Multiplications</th>
<th>Additions</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>$6 \cdot C_9$</td>
<td>$6 \cdot (C_9 - 1)$</td>
</tr>
<tr>
<td>S2</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>S3</td>
<td>$6 \cdot C_9$</td>
<td>$5 \cdot C_9$</td>
</tr>
</tbody>
</table>
The algorithm scales very well: one thread for each body

Each thread only interacts with adjacent joints

Load balance is obtained when the bodies have similar topology index
Direct Solution of the Reduced System
The Sparse Direct Solver

- Amounts to finding the solution of the reduced system \( E\lambda = b \)

- In general, there is a connectivity graph associated with each linear system
  
  - In particular, for the reduced system, this connectivity graph is easily visualized for mechanical systems by considering the joints and bodies present in the mechanical system
    
    Recall: joints connect bodies. We are after finding the value of the Lagrange Multipliers \( \lambda \). Each joint \( i \) that connects two bodies \( L_i \) and \( U_i \) has one and only one Lagrange Multiplier \( \lambda_i \) associated with it

    - In the graph representation of the mechanism, the joints (or constraints) in the mechanism represent the edges in the graph; i.e., the Lagrange Multipliers. The vertices are the bodies that are connected by the joints (constraints)

    - Moreover, if a edge \( i \) connects two vertices, say \( L_i \) and \( U_i \), it means that in the governing equation of \( \lambda_i \) will show up all the \( \lambda \)'s that connect \( L_i \) or \( U_i \) with other bodies in the mechanism

- The important remark: finding the parallel factorization has a graphical interpretation that brings into the picture the associated connectivity graph for the matrix \( E_{m\times m} \)

- The factorization is translated into a parallel elimination of edges in the connectivity graph
The Direct Solver: How Things Get Done

- In the reduced linear system $\mathbf{E}\lambda = \mathbf{b}$, each constraint induces an equation.

- Example: constraint 3 induced equation:

$\mathbf{E}_{32}\lambda_2 + \mathbf{E}_{33}\lambda_3 + \mathbf{E}_{35}\lambda_5 + \mathbf{E}_{36}\lambda_6 = \mathbf{b}_3$

- Since $\mathbf{E}$ is positive definite, $\mathbf{E}_{33}$ is also positive definite.

- Fundamental Idea:
  - Solve for $\lambda_3$ and substitute it in all the equations where it shows up.
First Example: Seven-Body Mechanism
The fundamental question is this: what should be the sequence in which the unknowns (the edges of the graph) are eliminated?
- Different elimination sequences result in different levels of effort

The question becomes more complicated since you are interested in a parallel elimination sequence
- You would like to limit the amount of synchronization barriers that you impose in the implementation

In the end, although it’s formulated like solving a system, the problem becomes that starting with a graph and eliminating its edges in parallel
- Similar to a Mikado, or “pick-up sticks”, game that you want to play in parallel
### Second Example: HMMWV Model

<table>
<thead>
<tr>
<th>Elim. Sequence</th>
<th>A</th>
<th>M</th>
<th>I</th>
<th>F</th>
<th>NNZ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bad</td>
<td>1240</td>
<td>1336</td>
<td>195</td>
<td>96</td>
<td>99</td>
</tr>
<tr>
<td>Good</td>
<td>459</td>
<td>469</td>
<td>109</td>
<td>10</td>
<td>99</td>
</tr>
<tr>
<td><strong>Index Reduction</strong></td>
<td>220</td>
<td>233</td>
<td>90</td>
<td>13</td>
<td>77</td>
</tr>
</tbody>
</table>

![Diagrams of HMMWV Model](image)
“A computer will do what you tell it to do, but that may be much different from what you had in mind”

Joseph Weizenbaum
Before We Get Started…

● Last time
  ● Discussed Midterm Project topics 3 and 4
    ● Finite Element Method on the GPU. Area coordinators: Prof. Suresh and Naresh Khude
    ● Sparse direct solver on the GPU (Cholesky). Area coordinator: Dan Negrut

● Today
  ● Thread divergence on the GPU
  ● Execution Configuration Optimization
  ● Instruction Optimization

● Other issues
  ● HW6 posted (due 03/22): deals with a parallel prefix scan operation
  ● Midterm Projects: Four new discussion threads started, one for each topic
    ● You will have to submit a one paragraph document by 11:59 PM today to commit to a project topic
      • Use “MidtermProject” drop-box in Learn@UW
    ● My advice: work on Project 3 or 4 only if you want to make it your Final Project
    ● Brute force collision detection is the easiest way out
How thread blocks are partitioned

- Each thread block is partitioned into warps
  - Thread indices (indexes?) within a warp are consecutive and increasing
    - Remember: In multidimensional blocks, the x thread index runs first, followed by the y thread index, and finally followed by the z thread index
  - Warp 0 starts with Thread Idx 0

- Partitioning of threads in warps is always the same
  - You can use this knowledge in control flow
  - So far, the warp size of 32 has been kept constant from device to device and CUDA version to CUDA version

- While you can rely on ordering among threads, DO NOT rely on any ordering among warps
  - Remember, the concept of warp is not something you control through CUDA
  - If there are any dependencies between threads, you must __syncthreads() to get correct results
Main performance concern with branching is *divergence*

- Threads within a single warp take different paths
- Different execution paths are serialized
  - The control paths taken by the threads in a warp are traversed one at a time until there is no more.
- NOTE: Don’t forget that divergence can manifest *only* at the warp level. You can not discuss this concept in relation to code executed by threads in different warps.
A common case: branch condition is a function of thread ID

- Example with divergence:
  - If (threadIdx.x > 2) { }
  - This creates two different control paths for threads in a block
  - Branch granularity < warp size; threads 0 and 1 follow different path than the rest of the threads in the first warp

- Example without divergence:
  - If (threadIdx.x / WARP_SIZE >= 2) { }
  - Also creates two different control paths for threads in a block
  - Branch granularity is a whole multiple of warp size; all threads in any given warp follow the same path
Control Flow Instructions

- *if, switch, for, while* – can significantly impact the effective instruction throughput when threads of the same warp diverge.

- If this happens, the execution is serialized:
  - This increases the number of instructions executed for this warp.
  - When all the different execution paths have completed, the threads converge back to the same execution path.
  - Not only that you execute more instructions, but you also need logic associated with this process (book-keeping).
Parallel reduction is a *very* common problem
- Given an array of values, “reduce” them in parallel to a single value

Examples
- Sum reduction: sum of all values in the array
- Max reduction: maximum of all values in the array
- Min reduction: minimum of all values in the array

One example where you can run into it:
- Find the infinity norm of a very large array – used as a convergence test for an iterative solver, for instance

Typically parallel implementation:
- Recursively halve the number of threads, deal with two values per thread
- Takes log(n) steps for n elements, requires n/2 threads
A Vector Reduction Example

- Assume an in-place reduction using shared memory
  - We are in the process of summing up a 512 element array
  - The shared memory used to hold a partial sum vector
  - Each iteration brings the partial sum vector closer to the final sum
  - The final sum will be stored in element 0
A simple implementation

- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

```c
unsigned int t = threadIdx.x;
for (unsigned int stride = 1; stride < blockDim.x; stride *= 2)
{
    __syncthreads();
    if (t % (2*stride) == 0) partialSum[t] += partialSum[t+stride];
}
```
The “Branch Divergence” Aspect

Array elements

Thread 0
Thread 2
Thread 4
Thread 6
Thread 8
Thread 10

0 1 2 3 4 5 6 7 8 9 10 11

0...3
4..7
8..11

0+1
2+3
4+5
6+7
8+9
10+11

 HK-UIUC

iterations
Some Observations

- In each iteration, two control flow paths will be sequentially traversed for each warp
  
  - Threads that perform addition and threads that do not
  
  - Threads that do not perform addition may cost extra cycles depending on the implementation of divergence
Some Observations (cont.)

- No more than half of the threads will be executing at any time
  - All odd index threads are disabled right from the beginning!
  - On average, less than ¼ of the threads will be activated for all warps over time.
  - After the 5th iteration, entire warps in each block will be disabled, poor resource utilization but no divergence.
    - This can go on for a while, up to 4 more iterations (512/32=16= $2^4$), where each iteration only has one thread activated until all warps retire
A Better Implementation

- Assume we have already loaded array into
  - `__shared__ float partialSum[]`

```c
unsigned int t = threadIdx.x;
for (unsigned int stride = blockDim.x; stride >= 1; stride >>= 1) {
   __syncthreads();
   if (t < stride)
      partialSum[t] += partialSum[t+stride];
}
```
No Divergence until < 16 sub-sums
Some Observations About the New Implementation

- Only the last 5 iterations will have divergence

- Entire warps will be shut down as iterations progress
  - For a 512-thread block, 4 iterations to shut down all but one warp in the block. Here’s how the thread count shapes up:
    - 512 (1\text{st} iteration), 256 (2\text{nd} iteration), 128 (3\text{rd} iteration), 64 (4\text{th} iteration)

  - Better resource utilization, will likely retire warps and thus block executes faster

- Recall, no bank conflicts either
Predicated Execution Concept
[Looking Under the Hood]

- The thread divergence can be avoided in some cases by using the concept of predication

  `<p1> LDR r1, r2, 0`

- If `p1` is TRUE, the assembly code instruction above executes normally

- If `p1` is FALSE, instruction treated as NOP
  - NOP – “no operation”
Predication Example

\[
\text{if (x == 10)} \\
\quad c = c + 1;
\]

\[
\text{LDR r5, X} \\
p1 <- r5 \text{ eq 10} \\
\text{LDR r1 <- C} \\
\text{ADD r1, r1, 1} \\
\text{STR r1 -> C}
\]
Predication Helpful for If-Else
If-else example

:p1, p2 <- r5 eq 10
<p1> inst 1 from B
<p1> inst 2 from B
<p1> :
:p2> inst 1 from C
<p2> inst 2 from C

This is what gets scheduled

The cost is extra instructions will be issued each time the code is executed. However, there is no branch divergence.
Instruction Predication
[Tesla C1060]

- A comparison instruction sets a condition code (CC)

- Instructions can be predicated to write results only when CC meets criterion (CC != 0, CC >= 0, etc.)

- Compiler tries to predict if a branch condition is likely to produce many divergent warps
  - If that’s the case, go ahead and predicate if the branch has <7 instructions
  - If that’s not the case, only predicate if the branch has <4 instructions
  - Note: it’s pretty bad if you predicate when it was obvious that there would have been no divergence
ALL predicated instructions take execution cycles
- Those with false conditions don’t write their output, and do not evaluate addresses or read operands
- Saves branch instructions, so can be cheaper than serializing divergent paths

If all this business is confusing, remember this:
- Avoid thread divergence

It’s not 100% clear to me, but I believe that there is no cost if a subset of threads belonging to a warp sits there and does nothing while the other warp threads are all running the same instruction
End: Control Flow in CUDA

Begin: Execution Configuration Optimization
Thread instructions are executed sequentially, so executing other warps is the only way to hide latencies and keep the hardware busy

**Occupancy** = Number of warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently

Can have 32 warps on Tesla C1060

**Limited by resource usage:**
- Registers
- Shared memory

Source: NVIDIA
Blocks per Grid Heuristics

# of blocks > # of multiprocessors
- So all multiprocessors have at least one block to execute

# of blocks / # of multiprocessors > 2
- Multiple blocks can run concurrently in a multiprocessor
- Blocks that aren’t waiting at a __syncthreads() keep the hardware busy
- Subject to resource availability – registers, shared memory

# of blocks > 100 to scale to future devices
- Blocks executed in pipeline fashion
- 1000’s of blocks per grid will scale across multiple generations

Source: NVIDIA
Register Dependency

Read-after-write register dependency

Instruction’s result can be read ~24 cycles later

Scenarios:

CUDA:

x = y + 5;
z = x + 3;
s_data[0] += 3;

PTX (Parallel Thread eXecution ISA):

add.f32 $f3, $f1, $f2
add.f32 $f5, $f3, $f4
ld.shared.f32 $f3, [$r31+0]
add.f32 $f3, $f3, $f4

To completely hide the latency:

Run at least 192 threads (6 warps) per multiprocessor
At least 25% occupancy (1.0/1.1), 18.75% (1.2/1.3)
Threads do not have to belong to the same thread block

Source: NVIDIA
Register Pressure

Hide latency by using more threads per SM

Limiting Factors:
- Number of registers per kernel
  - 8K/16K per multiprocessor, partitioned among concurrent threads
- Amount of shared memory
  - 16KB per multiprocessor, partitioned among concurrent threadblocks

Compile with --ptxas-options=-v flag

Use --maxrregcount=N flag to NVCC
- N = desired maximum registers / kernel
- At some point “spilling” into local memory may occur
  - Reduces performance – local memory is slow

Source: NVIDIA
Occupancy Calculator

CUDA GPU Occupancy Calculator

Click here for detailed instructions on how to use this occupancy calculator.

For more information on NVIDIA CUDA, visit http://developer.nvidia.com/cuda

Your chosen resource usage is indicated by the red triangle on the graphs. The other data points represent the range of possible block sizes, register counts, and shared memory allocation.

Source: NVIDIA
Optimizing Threads Per Block

Choose threads per block as a multiple of warp size
- Avoid wasting computation on under-populated warps
- Facilitates coalescing

Want to run as many warps as possible per multiprocessor (hide latency)
Multiprocessor can run up to 8 blocks at a time

Heuristics
- Minimum: 64 threads per block
  - Only if multiple concurrent blocks
- 192 or 256 threads a better choice
  - Usually still enough regs to compile and invoke successfully
- This all depends on your computation, so experiment!

Source: NVIDIA
Occupancy != Performance

Increasing occupancy does not necessarily increase performance

**BUT...**

Low-occupancy multiprocessors cannot adequately hide latency on memory-bound kernels

- In the end, it all comes down to arithmetic intensity and available parallelism

Source: NVIDIA
Parameterize Your Application

Parameterization helps adaptation to different GPUs

GPUs vary in many ways
- # of SMs (stream multiprocessors)
- Memory bandwidth
- Shared memory size
- Register file size
- Max. threads per block

You can even make apps self-tuning (like FFTW and ATLAS)
- “Experiment” mode discovers and saves optimal configuration

Source: NVIDIA
End: Execution Configuration Optimization

Begin: Instruction Optimizations
CUDA Instruction Performance

Instruction cycles (per warp) = sum of
- Operand read cycles
- Instruction execution cycles
- Result update cycles

Therefore instruction throughput depends on
- Nominal instruction throughput
- Memory latency
- Memory bandwidth

“Cycle” refers to the multiprocessor clock rate
- 1.3 GHz on the Tesla C1060, for example

Source: NVIDIA
Maximizing Instruction Throughput

Maximize use of high-bandwidth memory
- Maximize use of shared memory
- Minimize accesses to global memory
- Maximize coalescing of global memory accesses

Optimize performance by overlapping memory accesses with HW computation
- You need many warps running on the SM to this end…
- Another thing that’s helpful: high arithmetic intensity programs
  - i.e. high ratio of math to memory transactions

Source: NVIDIA
Arithmetic Instruction Throughput

- **int and float add, shift, min, max and float mul, mad:** 4 cycles per warp
  - **int multiply (×) is by default 32-bit**
    - requires multiple cycles / warp
  - **Use __mul24() / __umul24() intrinsics for 4-cycle 24-bit int multiply**

- **Integer divide and modulo are more expensive**
  - Compiler will convert literal power-of-2 divides to shifts
    - But it has been documented to miss some cases
  - Be explicit in cases where compiler can’t tell that divisor is a power of 2!
  - **Useful trick:** `foo % n == foo & (n - 1)` if `n` is a power of 2
Runtime Math Library

There are two types of runtime math operations in single precision

__funcf(): direct mapping to hardware ISA
- Fast but lower accuracy (see prog. guide for details)
- Examples: __sinf(x), __expf(x), __powf(x, y)

funcf(): compile to multiple instructions
- Slower but higher accuracy (5 ulp or less)
- Examples: sinf(x), expf(x), powf(x, y)

The -use_fast_math compiler option forces every funcf() to compile to __funcf()
GPU Results May Not Match CPU

Many variables: hardware, compiler, optimization settings

CPU operations aren’t strictly limited to 0.5 ulp
- Sequences of operations can be more accurate due to 80-bit extended precision ALUs
- ULP: “Unit in the Last Place” is the spacing between floating-point numbers, i.e., the value that the least significant bit (lsb) represents if it is 1. It is used as a measure of precision in numeric calculations

Floating-point arithmetic is not associative!

Source: NVIDIA
FP Math is Not Associative!

In symbolic math, \((x+y)+z == x+(y+z)\)

This is not necessarily true for floating-point addition

Try \(x = 10^{30}, y = -10^{30}\) and \(z = 1\) in the above equation

When you parallelize computations, you potentially change the order of operations

Parallel results may not exactly match sequential results

This is not specific to GPU or CUDA – inherent part of parallel execution

Source: NVIDIA
Control Flow Instructions

Main performance concern with branching is divergence
- Threads within a single warp take different paths
- Different execution paths must be serialized

Avoid divergence when branch condition is a function of thread ID

Example with divergence:
```c
if (threadIdx.x > 2) { }
```
- Branch granularity < warp size

Example without divergence:
```c
if (threadIdx.x / WARP_SIZE > 2) { }
```
- Branch granularity is a whole multiple of warp size

Source: NVIDIA
“There are two ways of constructing a software design: one way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies. The first method is far more difficult.”

Sir C. A. R. Hoare
Before We Get Started…

- Last time
  - Thread divergence on the GPU
  - Execution Configuration Optimization
  - Instruction Optimization

- Today
  - Summarize lessons learned in a collection of optimization rules of thumb
  - Discuss parallel prefix scan on the GPU
    - This is your next homework, due 03/22

- Other issues
  - The “easy way out” Midterm Project was emailed to you
  - There are three students who haven’t indicated their Midterm Project choice
  - HW6 was posted, due on 03/22
    - Please remember that you have reading assigned as well (see online Syllabus)
Performance Optimization

- Performance optimization revolves around three basic strategies:
  - Maximizing parallel execution
  - Optimizing memory usage to achieve maximum memory bandwidth
  - Optimizing instruction usage to achieve maximum instruction throughput

- Writing CUDA software is a craft
  - Sometimes having to deal with conflicting requirements
  - A list of recommendations is provided. Sections that are referenced are as in the CUDA C Best Practices Guide Version 3.2
Writing CUDA Software: High-Priority Recommendations

1. To get the maximum benefit from CUDA, focus first on finding ways to parallelize sequential code. (Section 1.1.3)

2. Use the effective bandwidth of your computation as a metric when measuring performance and optimization benefits. (Section 2.2)

3. Minimize data transfer between the host and the device, even if it means running some kernels on the device that do not show performance gains when compared with running them on the host CPU. (Section 3.1)

4. Ensure global memory accesses are coalesced whenever possible. (Section 3.2.1)

5. Minimize the use of global memory. Prefer shared memory access where possible. (Section 5.2)

6. Avoid different execution paths within the same warp. (Section 6.1)
1. Accesses to shared memory should be designed to avoid serializing requests due to bank conflicts. (Section 3.2.2.1)

2. To hide latency arising from register dependencies, maintain sufficient numbers of active threads per multiprocessor (i.e., sufficient occupancy). (Sections 3.2.6 and 4.3)

3. The number of threads per block should be a multiple of 32 threads, because this provides optimal computing efficiency and facilitates coalescing. (Section 4.4)
4. Use the fast math library whenever speed trumps precision. (Section 5.1.4)

5. Prefer faster, more specialized math functions over slower, more general ones when possible. (Section 5.1.4)

6. Use signed integers rather than unsigned integers as loop counters. (Section 6.3)
Writing CUDA Software: Low-Priority Recommendations

1. For kernels with long argument lists, place some arguments into constant memory to save shared memory. (Section 3.2.2.4)

2. Use shift operations to avoid expensive division and modulo calculations. (Section 5.1.1)

3. Avoid automatic conversion of doubles to floats. (Section 5.1.3)

4. Make it easy for the compiler to use branch predication in lieu of loops or control statements. (Section 6.2)

End Optimization Issues
Start Parallel Prefix Scan
Objectives of This Exercise

- The vehicle for the software design exercise: implementation of a parallel prefix sum operation
  - Recall first assignment, also the topic of assignment #6

- Goal 1: Putting your CUDA knowledge to work

- Goal 2: Understand that
  - Different algorithmic designs lead to different performance levels
  - Different constraints dominate in different applications and/or design solutions
  - Case studies help to establish intuition, idioms and ideas

- Goal 3: Identify parallel algorithm patterns that can result in superior performance
  - Understand that there are patterns and it’s worth being aware of them
  - If you want, these are the tricks of the trade
  - When considering patterns, you can’t lose sight of the underlying hardware
You come to rely on compiler to figure out the parallelism in a piece of code and then map it to an underlying hardware
- VERY hard, the holy grail in parallel computing

You rely on parallel libraries built for a specific underlying hardware
- Very convenient, the way to go when such libraries are available

You rely on language extensions to facilitate the process of generating a parallel executable
- This is where you are with CUDA
- Presents a great opportunity to screw up or to generate some tailored code that takes care of your *specific* application in an efficient fashion
Parallel Prefix Sum (Scan)

- **Definition:**
  The all-prefix-sums operation takes a binary associative operator $\oplus$ with identity $I$, and an array of $n$ elements
  \[
  [a_0, a_1, \ldots, a_{n-1}]
  \]
  and returns the ordered set
  \[
  [I, a_0, (a_0 \oplus a_1), \ldots, (a_0 \oplus a_1 \oplus \ldots \oplus a_{n-2})].
  \]

- **Example:**
  if $\oplus$ is addition, then scan on the set
  \[
  [3 \ 1 \ 7 \ 0 \ 4 \ 1 \ 6 \ 3]
  \]
  returns the set
  \[
  [0 \ 3 \ 4 \ 11 \ 11 \ 15 \ 16 \ 22]
  \]
  (From Blelloch, 1990, “Prefix Sums and Their Applications)
void scan( float* scanned, float* input, int length) {
    scanned[0] = 0;
    for(int i = 1; i < length; ++i) {
        scanned[i] = scanned[i-1] + input[i-1];
    }
}

- Just add each element to the sum of the elements before it
- Trivial, but sequential
- Exactly $n-1$ adds: optimal in terms of work efficiency
Applications of Scan

- Scan is a simple and useful parallel building block
  - Convert recurrences from sequential ...
    ```cpp
    for(j=1; j<n; j++)
        out[j] = out[j-1] + f(j);
    ...
    ... into parallel:
    forall(j) in parallel
        temp[j] = f(j);
    scan(out, temp);
    ```

- Useful in implementation of several parallel algorithms:
  - Radix sort
  - Quicksort
  - String comparison
  - Lexical analysis
  - Stream compaction
  - Polynomial evaluation
  - Solving recurrences
  - Tree operations
  - Histograms
  - Etc.
Parallel Scan Algorithm: Solution One
Hillis & Steele (1986)

- Note that an implementation of the algorithm shown in picture requires two buffers of length $n$ (shown is the case $n=8=2^3$).
- Assumption: the number $n$ of elements is a power of 2: $n=2^M$
The Plain English Perspective

- First iteration, I go with stride $1=2^0$
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have $2^M-1$ additions

- Second iteration, I go with stride $2=2^1$
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have $2^M-2^1$ additions

- Third iteration: I go with stride $4=2^2$
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have $2^M-2^2$ additions

- … (and so on)
The Plain English Perspective

- Consider the $k^{th}$ iteration (where $1 < k < M - 1$): I go with stride $2^{k-1}$
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have $2^M - 2^{k-1}$ additions

- ...

- $M^{th}$ iteration: I go with stride $2^{M-1}$
  - Start at $x[2^M]$ and apply this stride to all the array elements before $x[2^M]$ to find the mate of each of them. When looking for the mate, the stride should not land you before the beginning of the array. The sum replaces the element of higher index.
    - This means that I have $2^M - 2^{M-1}$ additions

- NOTE: There is no $(M+1)^{th}$ iteration since this would automatically put me beyond the bounds of the array (if you apply an offset of $2^M$ to "&$x[2^M]$ " it places you right before the beginning of the array – not good…)
Hillis & Steele Parallel Scan Algorithm

- Algorithm looks like this:

```plaintext
for d := 0 to M-1 do
   forall k in parallel do
      if k - 2^d ≥ 0 then
         x[out][k] := x[in][k] + x[in][k - 2^d]
      else
         x[out][k] := x[in][k]
   endforall
   swap(in,out)
endfor
```

Double-buffered version of the sum scan
Operation Count
Final Considerations

- The number of operations tally:
  - \((2^M-2^0) + (2^M-2^1) + \ldots + (2^M-2^{k-1}) + \ldots + (2^M-2^{M-1})\)
  - Final operation count:
    \[
    M \cdot 2^M - (2^0 + \ldots + 2^{M-1}) = M \cdot 2^M - 2^M + 1 = n(\log(n) - 1) + 1
    \]
  - This is an algorithm with \(O(n \cdot \log(n))\) work

- This scan algorithm is not that work efficient
  - Sequential scan algorithm only needs \(n-1\) additions
  - A factor of \(\log(n)\) might hurt: 20x more work for \(10^6\) elements!
    - Homework requires a scan of about 16 million elements
    - Additionally, you need two buffers…

- A parallel algorithm can be slow when execution resources are saturated due to low algorithm efficiency
__global__ void scan(float *g_odata, float *g_idata, int n) {
    extern __shared__ float temp[]; // allocated on invocation

    int thid = threadIdx.x;
    int pout = 0, pin = 1;

    // load input into shared memory.
    // Exclusive scan: shift right by one and set first element to 0
    temp[thid] = (thid > 0) ? g_idata[thid-1] : 0;
    __syncthreads();

    for (int offset = 1; offset < n; offset <<= 1 )
    {
        pout = 1 - pout; // swap double buffer indices
        pin = 1 - pout;

        if (thid >= offset)
            temp[pout*n+thid] += temp[pin*n+thid - offset];
        else
            temp[pout*n+thid] = temp[pin*n+thid];

        __syncthreads();
    }

    g_odata[thid] = temp[pout*n+thid1]; // write output
}
The kernel is very simple, which is good

Note the nice trick that was used to swap the buffers

The kernel only works when the entire array is processed by one block

One block in CUDA has 512 threads, which means I can have up to 1024 elements (short of 16 million, which is your assignment)

This needs to be improved upon…
Improving Efficiency

- A common parallel algorithm pattern: 
  
  *Balanced Trees*
  
  - Build a balanced binary tree on the input data and sweep it to and then from the root
  - Tree is not an actual data structure, but a concept to determine what each thread does at each step

- For scan:
  
  - Traverse down from leaves to root building partial sums at internal nodes in the tree
    - Root holds sum of all leaves (this is a reduction algorithm!)
  - Traverse back up the tree building the scan from the partial sums
Picture and Pseudocode
~ Reduction Step ~

for k=0 to M-1
    offset = 2^k
    for j=1 to 2^{M-k-1} in parallel do
        x[j \cdot 2^{k+1}-1] = x[j \cdot 2^{k+1}-1] + x[j \cdot 2^{k+1}-2^k-1]
    endfor
endfor

\[ \begin{array}{cccccccc}
    d=0 & x_0 & \Sigma(x_0,x_1) & x_2 & \Sigma(x_0,x_3) & x_4 & \Sigma(x_4,x_5) & x_6 & \Sigma(x_4,x_7) \\
    d=1 & x_0 & \Sigma(x_0,x_1) & x_2 & \Sigma(x_0,x_3) & x_4 & \Sigma(x_4,x_5) & x_6 & \Sigma(x_4,x_7) \\
    d=2 & x_0 & \Sigma(x_0,x_1) & x_2 & \Sigma(x_2,x_3) & x_4 & \Sigma(x_4,x_5) & x_6 & \Sigma(x_4,x_7) \\
    d=3 & x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\
\end{array} \]

\[ \begin{array}{cccccccc}
    j \cdot 2^{k+1} - 1 = & 1 & 3 & 5 & 7 \\
    & 3 & 7 & -1 & -1 \\
    & 7 & -1 & -1 & -1 \\
\end{array} \]

\[ \begin{array}{cccccccc}
    j \cdot 2^{k+1} - 2^k - 1 = & 0 & 2 & 4 & 6 \\
    & 1 & 5 & -1 & -1 \\
    & 3 & -1 & -1 & -1 \\
\end{array} \]
Operation Count, Reduce Phase

for $k=0$ to $M-1$
  offset = $2^k$
  for $j=1$ to $2^{M-k-1}$ in parallel do
    $x[j \cdot 2^{k+1}-1] = x[j \cdot 2^{k+1}-1] + x[j \cdot 2^{k+1}-2^k-1]$
  endfor
endfor

By inspection:

$$\sum_{k=0}^{M-1} 2^{M-k-1} = 2^M - 1 = n - 1$$

Looks promising…
The Down-Sweep Phase

for k=M-1 to 0
  offset = 2^k
  for j=1 to 2^{M-k-1} in parallel do
    dummy = x[j \cdot 2^{k+1} - 2^k - 1]
    x[j \cdot 2^{k+1} - 2^k - 1] = x[j \cdot 2^{k+1} - 1] + dummy
  endfor
endfor

NOTE: This is just a mirror image of the reduction stage. Easy to come up with the indexing scheme…
Down-Sweep Phase, Remarks

- Number of operations for the down-sweep phase:
  - Additions: n-1
  - Swaps: n-1 (each swap shadows an addition)

- Total number of operations associated with this algorithm
  - Additions: 2n-2
  - Swaps: n-1
    - Looks very comparable with the work load in the sequential solution

- The algorithm is convoluted though, it won’t be easy to implement
  - Kernel shown on next slide
__global__ void prescan(float *g_odata, float *g_idata, int n)
{
    extern __shared__ float temp[];  // allocated on invocation

    int thid = threadIdx.x;
    int offset = 1;

    temp[2*thid] = g_idata[2*thid];  // load input into shared memory
    temp[2*thid+1] = g_idata[2*thid+1];

    for (int d = n>>1; d > 0; d >>= 1) // build sum in place up the tree
    {
        __syncthreads();

        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            temp[bi] += temp[ai];
        }

        offset <<= 1;  //multiply by 2 implemented as bitwise operation
    }

    if (thid == 0) { temp[n - 1] = 0; } // clear the last element

    for (int d = 1; d < n; d *= 2) // traverse down tree & build scan
    {
        offset >>= 1;
        __syncthreads();

        if (thid < d)
        {
            int ai = offset*(2*thid+1)-1;
            int bi = offset*(2*thid+2)-1;

            float t = temp[ai];
            temp[ai] = temp[bi];
            temp[bi] += t;
        }

        __syncthreads();
    }

    g_odata[2*thid] = temp[2*thid];  // write results to device memory
    g_odata[2*thid+1] = temp[2*thid+1];
}
Bank Conflicts

- No penalty if all threads access different banks
  - Or if all threads read from the exact same address (broadcasting)

- This is not the case here: multiple threads access the same shared memory bank with different addresses; i.e. different rows of a bank
  - We have something like \(2^{k+1} \cdot j-1\)
    - \(k=0\): two way bank conflict
    - \(k=1\): four way bank conflict
    - ...

- Recall that shared memory accesses with conflicts are serialized
  - N-bank memory conflicts lead to a set of N successive shared memory transactions
Initial Bank Conflicts on Load

- Each thread loads two shared mem data elements
- Tempting to interleave the loads (see lines 9 & 10, and 46 & 47)
  \[
  \text{temp}[2 \times \text{thid}] = \text{g\_idata}[2 \times \text{thid}]; \\
  \text{temp}[2 \times \text{thid} + 1] = \text{g\_idata}[2 \times \text{thid} + 1];
  \]
  - Thread 0 accesses banks 0 and 1
  - Thread 1 accesses banks 2 and 3
  - ...  
  - Thread 8 accesses banks 16 and 17. Oops, that’s 0 and 1 again…
    - Two way bank conflict, can’t be easily eliminated
- Better to load one element from each half of the array
  \[
  \text{temp}[	ext{thid}] = \text{g\_idata}[	ext{thid}]; \\
  \text{temp}[	ext{thid} + (n/2)] = \text{g\_idata}[	ext{thid} + (n/2)];
  \]
- Solution above is helping with the global memory bandwidth as well…
Bank Conflicts in the Tree Algorithm

- When we build the sums, during the first iteration of the algorithm each thread in a half-warp reads two shared memory locations and writes one.
- We have bank conflicts: Threads (0 & 8) access bank 0 at the same time, and then bank 1 at the same time.

First iteration: 2 threads access each of 8 banks.
Bank Conflicts in the tree algorithm

- 2\textsuperscript{nd} iteration: even worse!
  - 4-way bank conflicts; for example:
    Th(0,4,8,12) access bank 1, Th(1,5,9,13) access Bank 5, etc.

\[
\begin{array}{cccccccccccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
\hline
3 & 4 & 7 & 4 & 4 & 5 & 6 & 9 & 5 & 13 & 2 & 2 & 3 & 6 & 1 & 10 \\
\end{array}
\]

2\textsuperscript{nd} iteration: 4 threads access each of 4 banks.

Each \( \bullet \) corresponds to a single thread.

Like-colored arrows represent simultaneous memory accesses
Managing Bank Conflicts in the Tree Algorithm

- Use padding to prevent bank conflicts
  - Add a word of padding every 16 words.
    - Now you work with a virtual 17 bank shared memory layout
  - Within a 16-thread half-warp, all threads access different banks
    - They are aligned to a 17 word memory layout
  - It comes at a price: you have memory words that are wasted
  - Keep in mind: you should also load data from global into shared memory using the virtual memory layout of 17 banks
Use Padding to Reduce Conflicts

- After you compute a ShMem address like this:
  \[
  \text{address} = 2 \times \text{stride} \times \text{thid};
  \]

- Add padding like this:
  \[
  \text{address} += (\text{address} >> 4); // \text{divide by NUM\_BANKS}
  \]

- This removes most bank conflicts
  - Not all, in the case of deep trees

- Material posted online will contain a discussion of this “deep tree” situation along with a proposed solution
Managing Bank Conflicts in the Tree Algorithm

Original scenario.

Virtual Bank:

Note that only arrows with the same color happen simultaneously.
Concluding Remarks, Parallel Scan

- Intuitively, the scan operation is not the type of procedure ideally suited for parallel computing
- Even if it doesn’t fit like a glove, leads to nice speedup:

<table>
<thead>
<tr>
<th># elements</th>
<th>CPU Scan (ms)</th>
<th>GPU Scan (ms)</th>
<th>Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>1024</td>
<td>0.002231</td>
<td>0.079492</td>
<td>0.03</td>
</tr>
<tr>
<td>32768</td>
<td>0.072663</td>
<td>0.106159</td>
<td>0.68</td>
</tr>
<tr>
<td>65536</td>
<td>0.146326</td>
<td>0.137006</td>
<td>1.07</td>
</tr>
<tr>
<td>131072</td>
<td>0.726429</td>
<td>0.200257</td>
<td>3.63</td>
</tr>
<tr>
<td>262144</td>
<td>1.454742</td>
<td>0.326900</td>
<td>4.45</td>
</tr>
<tr>
<td>524288</td>
<td>2.911067</td>
<td>0.624104</td>
<td>4.66</td>
</tr>
<tr>
<td>1048576</td>
<td>5.900097</td>
<td>1.118091</td>
<td>5.28</td>
</tr>
<tr>
<td>2097152</td>
<td>11.848376</td>
<td>2.099666</td>
<td>5.64</td>
</tr>
<tr>
<td>4194304</td>
<td>23.835931</td>
<td>4.062923</td>
<td>5.87</td>
</tr>
<tr>
<td>8388688</td>
<td>47.390906</td>
<td>7.987311</td>
<td>5.93</td>
</tr>
<tr>
<td>16777216</td>
<td>94.794598</td>
<td>15.854781</td>
<td>5.98</td>
</tr>
</tbody>
</table>

Source: 2007 paper of Harris, Sengupta, Owens
Concluding Remarks, Parallel Scan

- The Hillis-Steele (HS) implementation is simple, but suboptimal

- The Harris-Sengupta-Owen (HSO) solution is convoluted, but scales like $O(n)$
  - The complexity of the algorithm due to an acute bank-conflict situation

- Finally, we have not solved the problem yet: we only looked at the case when our array has up to 1024 elements
  - You will have to think how to handle the $16,777,216 = 2^{24}$ elements case
  - Likewise, it would be fantastic if you implement as well the case when the number of elements is not a power of 2
“Software is like entropy: It is difficult to grasp, weighs nothing, and obeys the Second Law of Thermodynamics; i.e., it always increases.”

– Norman Augustine
Before We Get Started…

- Last time
  - CUDA optimization rules of thumb
  - Discussed two parallel implementations of the prefix scan operation

- Today
  - Asynchronous Concurrent Execution in CUDA
  - Using a CUDA stream and asynchronous mem copy to decouple CPU and GPU execution
  - Handling Multiple Streams in CUDA as a means to enable task parallelism

- Other issues
  - Syllabus firmed up, we'll have three guest lecturers later in the semester
Concurrent Execution between Host and Device

- In order to facilitate concurrent execution between host and device, some function calls are asynchronous
  - Control is returned to the host thread before the device has completed the requested task

- Examples of asynchronous calls
  - Kernel launches
  - Device ↔ device memory copies
  - Host ↔ device memory copies of a memory block of 64 KB or less
  - Memory copies performed by functions that are suffixed with Async

- NOTE: When an application is run via a CUDA debugger or profiler (cuda-gdb, CUDA Visual Profiler, Parallel Nsight), all launches are synchronous
Concurrent Kernel Execution
[CUDA 3.2]

- Feature allows up to 16 kernels to be run on the device at the same time
- When is this useful?
  - Devices of compute capability 2.x are pretty wide (large number of SMs)
  - Sometimes you launch kernels whose execution configuration is smaller than the GPU’s “width”
  - Then, two or three independent kernels can be “squeezed” on the GPU at the same time
- Represents one of GPU’s attempts to look like a MIMD architecture
**Host-Device Data Transfer Issues**

- In general, host ↔ device data transfers using `cudaMemcpy()` are blocking
  - Control is returned to the host thread only after the data transfer is complete

- There is a non-blocking variant, `cudaMemcpyAsync()`
  ```
cudaMemcpyAsync(a_d, a_h, size, cudaMemcpyHostToDevice, 0);
kernel<<<grid, block>>>(a_d);
cpuFunction();
  ```
  - The host does not wait on the device to finish the mem copy and the kernel call for it to start execution of `cpuFunction()` call
  - The launch of “kernel” only happens after the mem copy call finishes

- **NOTE:** the asynchronous transfer version requires pinned host memory (allocated with `cudaHostAlloc()`), and it contains an additional argument (a stream ID)
When is this overlapping useful?
- Imagine a kernel executes on the device and only works with the lower half of the device memory
- Then, you can copy data from host to device in the upper half of the device memory?
- These two operations can take place simultaneously

Note that there is a issue with this idea:
- The device execution stack is FIFO, one function call on the device is not serviced until all the previous device function calls completed
- This would prevent overlapping execution with data transfer

This issue was addressed by the use of CUDA “streams”
CUDA Streams: Overview

- A programmer can manage concurrency through *streams*

- A stream is a sequence of CUDA commands that execute in order
  - Look at a stream as a queue of GPU operations
  - The execution order in a stream is identical to the order in which the GPU operations are added to the stream

- One host thread can define multiple CUDA streams
  - Think of a stream as a task that gets executed by the GPU. You can have multiple tasks and sometimes the GPU can execute parts of these tasks simultaneously

- What are the typical operations in a stream?
  - Invoking a data transfer
  - Invoking a kernel execution
  - Handling events
CUDA Streams: Overview

With respect to each other, different CUDA streams execute their commands as they see fit
- Inter-stream relative behavior is not guaranteed and should therefore not be relied upon for correctness (e.g. inter-kernel communication for kernels allocated to different streams is undefined)
- Another way to look at it: streams can by synchronized at barrier points, but correlation of sequence execution within different streams is not supported

When thinking about the typical GPU operations in the stream (see previous slide), remember that the GPU hardware has two types of engines
- One or more engines for copy operations
- One engine to execute kernels

The fact that there are two hardware engines becomes relevant in relation to how you organize the queuing of GPU operations in a stream
- For improved efficiency you want to have these two engines work simultaneously...
CUDA Streams: Creation

- A stream is defined by creating a stream object and specifying it as the stream parameter to a sequence of kernel launches and host ↔ device memory copies.

- The following code sample creates two streams and allocates an array “hostPtr” of float in page-locked memory
  - hostPtr will be used in asynchronous host ↔ device memory transfers

```c
cudaStream_t stream[2];
for (int i = 0; i < 2; ++i)
    cudaStreamCreate(&stream[i]);
float* hostPtr;
cudaMallocHost(&hostPtr, 2 * size);
```
CUDA Streams: Making of Use of Them

- In the code below, each of two streams is defined as a sequence of:
  - One memory copy from host to device,
  - One kernel launch, and
  - One memory copy from device to host

```c
for (int i = 0; i < 2; ++i) {
    cudaMemcpyAsync(inputDevPtr + i * size, hostPtr + i * size, size, cudaMemcpyHostToDevice, stream[i]);
    MyKernel<<<100, 512, 0, stream[i]>>>(outputDevPtr + i * size, inputDevPtr + i * size, size);
    cudaMemcpyAsync(hostPtr + i * size, outputDevPtr + i * size, size, cudaMemcpyDeviceToHost, stream[i]);
}
```

- There are some wrinkles to it, we’ll revisit shortly…
CUDA Streams: Clean Up Phase

- Streams are released by calling cudaStreamDestroy()

```c
for (int i = 0; i < 2; ++i)
    cudaStreamDestroy(stream[i]);
```

- Note that cudaStreamDestroy() waits for all preceding commands in the given stream to complete before destroying the stream and returning control to the host thread
CUDA Streams: Caveats

- Two commands from different streams cannot run concurrently if either one of the following operations is issued in-between them by the host thread:
  - A page-locked host memory allocation,
  - A device memory allocation,
  - A device memory set,
  - A device ↔ device memory copy,
  - Any CUDA command to stream 0 (including kernel launches and host ↔ device memory copies that do not specify any stream parameter)
  - A switch between the L1/shared memory configurations
CUDA Streams: Synchronization Aspects

- cudaThreadSynchronize() waits until all preceding commands in all streams have completed.

- cudaStreamSynchronize() takes a stream as a parameter and waits until all preceding commands in the given stream have completed. It can be used to synchronize the host with a specific stream, allowing other streams to continue executing on the device.

- cudaStreamWaitEvent() takes a stream and an event as parameters and makes all the commands added to the given stream after the call to cudaStreamWaitEvent() delay their execution until the given event has completed. The stream can be 0, in which case all the commands added to any stream after the call to cudaStreamWaitEvent() wait on the event.

- cudaStreamQuery() provides applications with a way to know if all preceding commands in a stream have completed.

- NOTE: To avoid unnecessary slowdowns, all these synchronization functions are usually best used for timing purposes or to isolate a launch or memory copy that is failing.
Example 1: Using One GPU Stream

- Example draws on material presented in the “CUDA By Example” book
  - J. Sanders and E. Kandrot, authors

- What is the purpose of this example?
  - Shows an example of using page-locked (pinned) host memory
  - Shows one strategy that you should invoke when dealing with applications that require more memory than you can accommodate on the GPU
  - [Most importantly] Shows a strategy that you can follow to get things done on the GPU without blocking the CPU (host)
    - While the GPU works, the CPU works too

- Remark:
  - In this example the magic happens on the host side. Focus on host code, not on the kernel executed on the GPU (the kernel code is basically irrelevant)
Kernel

- Computes an average, it’s not important, simply something that gets done and allows us later on to gauge efficiency gains when using *multiple* streams (for now dealing with one stream only)

```c
#include "../common/book.h"

#define N (1024*1024)
#define FULL_DATA_SIZE (N*20)

__global__ void kernel( int *a, int *b, int *c ) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        int idx1 = (idx + 1) % 256;
        int idx2 = (idx + 2) % 256;
        float as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
        float bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
        c[idx] = (as + bs) / 2;
    }
}
```
The “main()” Function

```c
int main( void ) {
    cudaEvent_t start, stop;
    float elapsedTime;
    cudaStream_t stream;
    int *host_a, *host_b, *host_c;
    int *dev_a, *dev_b, *dev_c;

    // start the timers
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );

    // initialize the stream
    HANDLE_ERROR( cudaStreamCreate( &stream ) );

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c, N * sizeof(int) ) );

    // allocate host locked memory, used to stream
    HANDLE_ERROR( cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
    HANDLE_ERROR( cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
    HANDLE_ERROR( cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );

    for (int i=0; i<FULL_DATA_SIZE; i++) {
        host_a[i] = rand();
        host_b[i] = rand();
    }
}
```
The “main()” Function

[Cntd.]

```c
HANDLE_ERROR( cudaEventRecord( start, 0 ) );
// now loop over full data, in bite-sized chunks
for (int i=0; i<FULL_DATA_SIZE; i+= N) {
    // copy the locked memory to the device, async
    HANDLE_ERROR( cudaMemcpyAsync( dev_a, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream ) );

    kernel<<N/256,256,0,stream>>>( dev_a, dev_b, dev_c );
}
```

```c
// copy result chunk from locked to full buffer
HANDLE_ERROR( cudaStreamSynchronize( stream ) );
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime, start, stop ) );
printf( "Time taken: %3.1f ms\n", elapsedTime );
```

```c
// cleanup the streams and memory
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a ) );
HANDLE_ERROR( cudaFree( dev_b ) );
HANDLE_ERROR( cudaFree( dev_c ) );
HANDLE_ERROR( cudaStreamDestroy( stream ) );
return 0;
```

The “main()” Function

[Cntd.]
Example 1, Summary

- Stage 1 sets up the events needed to time the execution of the program.
- Stage 2 allocates page-locked memory on the host side so that we can fall back on asynchronous memory copy operations between host and device.
- Stage 3 enqueues the set of GPU operations that need to be undertaken (the “chunkification”).
- Stage 4 needed for timing reporting.
- Stage 5: clean up time.
Example 2: Using Multiple Streams

[Version 1]

- Implement the same example but use two streams to this end

- Why would you want to use multiple streams?
  - For devices that are capable of overlapping execution with host↔device data movement, you might hide this data movement and improve overall performance

- Two ideas underlie the process
  - The idea of “chunkification” of the computation
    - Large computation is broken into pieces that are queued up for execution on the device (we already saw this in Example 1, which uses one stream)
  - The idea of overlapping execution with host↔device data movement
Overlapping Execution and Data Transfer: The Ideal Scenario

Timeline of intended application execution using two independent streams

- **Observations:**
  - “memcpy” actually represents an asynchronous cudaMemcpyAsync() memory copy call
  - White (empty) boxes represent time when one stream is waiting to execute an operation that it cannot overlap with the other stream’s operation
  - The goal: keep both GPU engine types (execution and mem copy) busy
    - Note: recent hardware allows two copies to take place simultaneously: one from host to device, at the same time one goes on from device to host (you have two copy engines)
Kernel

- Note that the kernel actually doesn’t change...

```c
#include "../common/book.h"
#define N   (1024*1024)
#define FULL_DATA_SIZE   (N*20)

__global__ void kernel(int *a, int *b, int *c) {
    int idx = threadIdx.x + blockIdx.x * blockDim.x;
    if (idx < N) {
        int idx1 = (idx + 1) % 256;
        int idx2 = (idx + 2) % 256;
        float as = (a[idx] + a[idx1] + a[idx2]) / 3.0f;
        float bs = (b[idx] + b[idx1] + b[idx2]) / 3.0f;
        c[idx] = (as + bs) / 2;
    }
}
```
The “main()” Function, 2 Streams

```c
int main( void ) {
    cudaDeviceProp prop;
    int whichDevice;
    HANDLE_ERROR( cudaGetDevice( &whichDevice ) );
    HANDLE_ERROR( cudaGetDeviceProperties( &prop, whichDevice ) );
    if (!prop.deviceOverlap) {
        printf( "Device will not handle overlaps, so no speed up from streams\n" );
        return 0;
    }

    cudaEvent_t start, stop;
    float elapsedTime;
    cudaStream_t stream0, stream1;
    int *host_a, *host_b, *host_c;
    int *dev_a0, *dev_b0, *dev_c0;
    int *dev_a1, *dev_b1, *dev_c1;

    // start the timers
    HANDLE_ERROR( cudaEventCreate( &start ) );
    HANDLE_ERROR( cudaEventCreate( &stop ) );

    // initialize the streams
    HANDLE_ERROR( cudaStreamCreate( &stream0 ) );
    HANDLE_ERROR( cudaStreamCreate( &stream1 ) );

    // allocate the memory on the GPU
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a0, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b0, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c0, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_a1, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_b1, N * sizeof(int) ) );
    HANDLE_ERROR( cudaMalloc( (void**)&dev_c1, N * sizeof(int) ) );

    // allocate host locked memory, used to stream
    HANDLE_ERROR( cudaHostAlloc( (void**)&host_a, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
    HANDLE_ERROR( cudaHostAlloc( (void**)&host_b, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
    HANDLE_ERROR( cudaHostAlloc( (void**)&host_c, FULL_DATA_SIZE * sizeof(int), cudaHostAllocDefault ) );
}
```
The “main()” Function, 2 Streams

[Cndt.]

```
for (int i=0; i<FULL_DATA_SIZE; i++) {
    host_a[i] = rand();
    host_b[i] = rand();
}

HANDLE_ERROR( cudaEventRecord( start, 0 ) );

// now loop over full data, in bite-sized chunks
for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
    // copy the locked memory to the device, async
    HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i, N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );

    kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );

    HANDLE_ERROR( cudaMemcpy( host_c+i, dev_c0, N * sizeof(int), cudaMemcpyDeviceToHost, stream0 ) );

    // copy the locked memory to the device, async
    HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );

    kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );

    HANDLE_ERROR( cudaMemcpy( host_c+i+N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1 ) );
}
```
The "main()" Function, 2 Streams

[Cntd.]

```c
HANDLE_ERROR( cudaStreamSynchronize( stream0 ) );
HANDLE_ERROR( cudaStreamSynchronize( stream1 ) );
HANDLE_ERROR( cudaEventRecord( stop, 0 ) );
HANDLE_ERROR( cudaEventSynchronize( stop ) );
HANDLE_ERROR( cudaEventElapsedTime( &elapsedTime, start, stop ) );
printf( "Time taken: %3.1f ms\n", elapsedTime );

// cleanup the streams and memory
HANDLE_ERROR( cudaFreeHost( host_a ) );
HANDLE_ERROR( cudaFreeHost( host_b ) );
HANDLE_ERROR( cudaFreeHost( host_c ) );
HANDLE_ERROR( cudaFree( dev_a0 ) );
HANDLE_ERROR( cudaFree( dev_b0 ) );
HANDLE_ERROR( cudaFree( dev_c0 ) );
HANDLE_ERROR( cudaFree( dev_a1 ) );
HANDLE_ERROR( cudaFree( dev_b1 ) );
HANDLE_ERROR( cudaFree( dev_c1 ) );
HANDLE_ERROR( cudaStreamDestroy( stream0 ) );
HANDLE_ERROR( cudaStreamDestroy( stream1 ) );
```

return 0;

```
Example 2 [Version 1], Summary

- Stage 1 ensures that your device supports your attempt to overlap kernel execution with host↔device data transfer (ok in devices of compute capability 1.1 and higher)

- Stage 2 sets up the events needed to time the execution of the program

- Stage 3 allocates page-locked memory on the host side so that we can fall back on asynchronous memory copy operations between host and device and initializes data

- Stage 4 enques the set of GPU operations that need to be undertaken (the “chunkification”)

- Stage 5 takes care of timing reporting and clean up
Comments, Using 2 Streams
[Version 1]

- Timing results provided by “CUDA by Example: An Introduction to General-Purpose GPU Programming,”
  - Sanders and Kandrot reported results on NVIDIA GTX285

- Using one stream (in Example 1): 62 ms

- Using two streams (this example, version 1): 61 ms

- Lackluster performance goes back to the way the two GPU engines (kernel execution and copy) are scheduled
The Two Stream Example, Version 1
Looking Under the Hood

At the left:
- An illustration of how the work queued up in the streams ends up being assigned by the CUDA driver to the two GPU engines (copy and execution)

At the right:
- Image shows dependency that is implicitly set up in the two streams given the way the streams were defined in the code
- The queue in the Copy Engine, combined with the dependencies defined determines the scheduling of the Copy and Kernel Engines (see next slide)
Note that due to the *specific* way in which the streams were defined (depth first), basically there is no overlap of copy & execution...

- This explains the no net-gain in efficiency compared to the one stream example

Remedy: go breadth first, instead of depth first

- In the current version, execution on the two engines was inadvertently blocked by the way the streams have been organized and the existing scheduling and lack of dependency checks available in the current version of CUDA
The 2 Stream Example
[Version 2: A More Effective Implementation]

- Old way (the depth first approach):
  - Assign the copy of a, copy of b, kernel execution, and copy of c to stream0. Subsequently, do the same for stream1

- New way (the breadth first approach):
  - Add the copy of a to stream0, and then add the copy of a to stream1
  - Next, add the copy of b to stream0, and then add the copy of b to stream1
  - Next, enqueue the kernel invocation in stream0, and then enqueue one in stream1.
  - Finally, enqueue the copy of c back to the host in stream0 followed by the copy of c in stream1.
The 2 Stream Example
A 20% More Effective Implementation (48 vs. 61 ms)

```c
for (int i=0; i<FULL_DATA_SIZE; i+= N*2) {
    // enqueue copies of a in stream0 and stream1
    HANDLE_ERROR( cudaMemcpyAsync( dev_a0, host_a+i,   N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_a1, host_a+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );
    // enqueue copies of b in stream0 and stream1
    HANDLE_ERROR( cudaMemcpyAsync( dev_b0, host_b+i,   N * sizeof(int), cudaMemcpyHostToDevice, stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( dev_b1, host_b+i+N, N * sizeof(int), cudaMemcpyHostToDevice, stream1 ) );
    // enqueue kernels in stream0 and stream1
    kernel<<<N/256,256,0,stream0>>>( dev_a0, dev_b0, dev_c0 );
    kernel<<<N/256,256,0,stream1>>>( dev_a1, dev_b1, dev_c1 );
    // enqueue copies of c from device to locked memory
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i, dev_c0,   N * sizeof(int), cudaMemcpyDeviceToHost, stream0 ) );
    HANDLE_ERROR( cudaMemcpyAsync( host_c+i+N, dev_c1, N * sizeof(int), cudaMemcpyDeviceToHost, stream1 ) );
}
```

Execution timeline of the breadth-first approach
(blue line shows dependency)
Using Streams, Lessons Learned

- Streams provide a basic mechanism that enables task-level parallelism in CUDA C applications

- Two requirements underpin the use of streams in CUDA C
  - `cudaHostAlloc()` should be used to allocate memory on the host so that it can be used in conjunction with a `cudaMemcpyAsync()` non-blocking copy command
    - The use of pinned (page-locked) host memory improves data transfer performance even if you only work with one stream
  - Effective latency hiding of kernel execution with memory copy operations requires a breadth-first approach to enqueuing operations in different streams
    - This is a consequence of the two engine setup associated with a GPU
CUDA 4.0
Application Porting Made Simpler

Rapid Application Porting
Unified Virtual Addressing

Faster Multi-GPU Programming
NVIDIA GPUDirect™ 2.0

Easier Parallel Programming in C++
Thrust
CUDA 4.0: Highlights

Easier Parallel Application Porting
- Share GPUs across multiple threads
- Single thread access to all GPUs
- No-copy pinning of system memory
- New CUDA C/C++ features
- Thrust templated primitives library
- NPP image/video processing library
- Layered Textures

Faster Multi-GPU Programming
- NVIDIA GPUDirect™ v2.0
  - Peer-to-Peer Access
  - Peer-to-Peer Transfers
- Unified Virtual Addressing

New & Improved Developer Tools
- Auto Performance Analysis
- C++ Debugging
- GPU Binary Disassembler
- cuda-gdb for MacOS
No-copy Pinning of System Memory

Reduce system memory usage and CPU memcpy() overhead
- Easier to add CUDA acceleration to existing applications
- Just register malloc’d system memory for async operations and then call cudaMemcpy() as usual

<table>
<thead>
<tr>
<th>Before No-copy Pinning</th>
<th>With No-copy Pinning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Extra allocation and extra copy required</td>
<td>Just register and go!</td>
</tr>
<tr>
<td><strong>malloc(a)</strong></td>
<td></td>
</tr>
<tr>
<td>cudaMallocHost(b)</td>
<td></td>
</tr>
<tr>
<td>memcpy(b, a)</td>
<td>cudaMemcpy()</td>
</tr>
<tr>
<td>cudaMemcpy() to GPU, launch kernels, cudaMemcpy()</td>
<td>cudaHostRegister(a)</td>
</tr>
<tr>
<td>memcpy(a, b)</td>
<td></td>
</tr>
<tr>
<td>cudaFreeHost(b)</td>
<td>cudaHostUnregister(a)</td>
</tr>
</tbody>
</table>

All CUDA-capable GPUs on Linux or Windows
- Requires Linux kernel 2.6.15+ (RHEL 5)
C++ Templatized Algorithms & Data Structures (Thrust)

- Powerful open source C++ parallel algorithms & data structures
- Similar to C++ Standard Template Library (STL)
- Automatically chooses the fastest code path at compile time
- Divides work between GPUs and multi-core CPUs
- Parallel sorting @ 5x to 100x faster

Data Structures
- thrust::device_vector
- thrust::host_vector
- thrust::device_ptr
- Etc.

Algorithms
- thrust::sort
- thrust::reduce
- thrust::exclusive_scan
- Etc.
CUDA 4.0: Highlights

Easier Parallel Application Porting
- Share GPUs across multiple threads
- Single thread access to all GPUs
- No-copy pinning of system memory
- New CUDA C/C++ features
- Thrust templated primitives library
- NPP image/video processing library
- Layered Textures

Faster Multi-GPU Programming
- NVIDIA GPUDirect™ v2.0
  - Peer-to-Peer Access
  - Peer-to-Peer Transfers
- Unified Virtual Addressing

New & Improved Developer Tools
- Auto Performance Analysis
- C++ Debugging
- GPU Binary Disassembler
- cuda-gdb for MacOS
Before NVIDIA GPUDirect™ v2.0

Required Copy into Main Memory

Two copies required:
1. cudaMemcpy(GPU2, sysmem)
2. cudaMemcpy(sysmem, GPU1)
NVIDIA GPUDirect™ v2.0: Peer-to-Peer Communication

Direct Transfers between GPUs

Only one copy required:
1. cudaMemcpy(GPU2, GPU1)
Unified Virtual Addressing
Easier to Program with Single Address Space

No UVA: Multiple Memory Spaces

UVA : Single Address Space
Unified Virtual Addressing

- One address space for all CPU and GPU memory
  - Determine physical memory location from pointer value
  - Enables libraries to simplify their interfaces (e.g. cudaMemcpy)

Supported on Tesla 20-series and other Fermi GPUs
- 64-bit applications on Linux and Windows TCC

<table>
<thead>
<tr>
<th>Before UVA</th>
<th>With UVA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Separate options for each permutation</td>
<td>One function handles all cases</td>
</tr>
<tr>
<td>cudaMemcpyHostToDevice</td>
<td>cudaMemcpyDefault</td>
</tr>
<tr>
<td>cudaMemcpyHostToDeviceToHost</td>
<td>(data location becomes an implementation detail)</td>
</tr>
</tbody>
</table>
CUDA 4.0: Highlights

**Easier Parallel Application Porting**
- Share GPUs across multiple threads
- Single thread access to all GPUs
- No-copy pinning of system memory
- New CUDA C/C++ features
- Thrust templated primitives library
- NPP image/video processing library
- Layered Textures

**Faster Multi-GPU Programming**
- NVIDIA GPUDirect™ v2.0
  - Peer-to-Peer Access
  - Peer-to-Peer Transfers
- Unified Virtual Addressing

**New & Improved Developer Tools**
- Auto Performance Analysis
- C++ Debugging
- GPU Binary Disassembler
- cuda-gdb for MacOS
## CUDA Features Overview

### New in CUDA 4.0

- **GPUDirect™ (v 2.0)**
  - Peer-Peer Communication

### Hardware Features

- ECC Memory
- Double Precision
- Native 64-bit Architecture
- Concurrent Kernel Execution
- Dual Copy Engines
- 6GB per GPU supported

### Operating System Support

- MS Windows 32/64
- Linux 32/64
- Mac OS X 32/64

### Designed for HPC

- Cluster Management
- GPU Direct
- Tesla Compute Cluster (TCC)
  - Multi-GPU support

### Fortran support

- CUDA Fortran (PGI)

### C support

- NVIDIA C Compiler
- CUDA C Parallel Extensions
- Function Pointers
- Recursion
- Atomics
  - mallocl/free

### C++ support

- Classes/Objects
- Class Inheritance
- Polymorphism
- Operator Overloading
- Class Templates
- Function Templates
- Virtual Base Classes
- Namespaces

### Fortran support

- CUDA Fortran (PGI)

### GPU Computing SDK

- NVML
- CUPTI

### 3rd Party Developer Tools

- Allinea DDT
- RogueWave / Totalview
- Vampir
- Tau
- CAPS HMPP

### NVIDIA Library Support

- Complete math.h
- Complete BLAS Library (1, 2 and 3)
- Sparse Matrix Math Library
- RNG Library
- FFT Library (1D, 2D and 3D)
- Video Decoding Library (NVCUVID)
- Video Encoding Library (NVCUVENC)
- Image Processing Library (NPP)
- Video Processing Library (NPP)

### Thrust C++ Library

- Templated Performance Primitives Library

### 3rd Party Math Libraries

- CULA Tools (EM Photonics)
- MAGMA Heterogeneous LAPACK
- IMSL (Rogue Wave)
- VSIPL (GPU VSIPL)

### NVIDIA Developer Tools

- Parallel Nsight
  - for MS Visual Studio cuda-gdb Debugger
  - with multi-GPU support
- CUDA/OpenCL Visual Profiler
- CUDA Memory Checker
- CUDA Disassembler
- GPU Computing SDK
- NVML
- CUPTI

### 3rd Party Developer Tools

- Allinea DDT
- RogueWave / Totalview
- Vampir
- Tau
- CAPS HMPP

© NVIDIA Corporation 2011
CUDA Developer Resources from NVIDIA

**Development Tools**

- **CUDA Toolkit**
  Complete GPU computing development kit
- **cuda-gdb**
  GPU hardware debugging
- **cuda-memcheck**
  Identifies memory errors
- **cuobjdump**
  CUDA binary disassembler

**Visual Profiler**

- GPU hardware profiler for CUDA C and OpenCL

**Parallel Nsight Pro**

- Integrated development environment for Visual Studio

**SDKs and Code Samples**

- **GPU Computing SDK**
  CUDA C/C++, DirectCompute, OpenCL code samples and documentation

- **Books**
  CUDA by Example
  GPU Computing Gems Programming Massively Parallel Processors
  Many more...

- **Optimization Guides**
  Best Practices for GPU computing and graphics development

**Libraries and Engines**

- **Math Libraries**
  CUFFT, CUBLAS, CUSPARSE, CURAND, math.h

- **3rd Party Libraries**
  CULA LAPACK, VSIPL

- **NPP Image Libraries**
  Performance primitives for imaging

- **App Acceleration Engines**
  Ray Tracing: Optix, iRay

- **Video Encoding / Decoding**
  NVCUVENC / VCUVID

PGI CUDA x86 Compiler

Benefits
- Deploy CUDA apps on legacy systems without GPUs
- Less code maintenance for developers

Timeline
- April/May 1.0 initial release
  Develop, debug, test functionality
- Aug 1.1 performance release
  Multicore, SSE/AVX support
CUDA 3rd Party Ecosystem

Cluster Tools
- Cluster Management
  - Platform HPC
  - Platform Symphony
  - Bright Cluster manager
  - Ganglia Monitoring System
  - Moab Cluster Suite
  - Altair PBS Pro
- Job Scheduling
  - Altair PBSpro
  - TORQUE
  - Platform LSF
- MPI Libraries
  - Coming soon...

Parallel Language Solutions & APIs
- PGI CUDA Fortran
- PGI Accelerator (C/Fortran)
- PGI CUDA x86
- CAPS HMPP
- pyCUDA (Python)
- Tidepowerd GPU.NET (C#)
- JCuda (Java)
- Khronos OpenCL
- Microsoft DirectCompute

3rd Party Math Libraries
- CULA Tools (EM Photonics)
- MAGMA Heterogeneous LAPACK
- IMSL (Rogue Wave)
- VSIPL (GPU VSIPL)
- NAG

Parallel Tools
- Parallel Debuggers
  - MS Visual Studio with Parallel Nsight Pro
  - Allinea DDT Debugger
  - TotalView Debugger
- Parallel Performance Tools
  - ParaTools VampirTrace
  - TauCUDA Performance Tools
  - PAPI
  - HPC Toolkit

Compute Platform Providers
- Cloud Providers
  - Amazon EC2
  - Peer 1
- OEM’s
  - Dell
  - HP
  - IBM
- Infiniband Providers
  - Mellanox
  - QLogic

© NVIDIA Corporation 2011
ME964
High Performance Computing for Engineering Applications

CUDA Optimization Tips (Hammad Mazhar)
Schedule related issues
March 22, 2011

“Adding manpower to a late software project makes it later”.
Fred Brooks
Overview

- General Guidelines
  - What to do and What not to do
  - Debugging Tips
  - Compiler
  - Assembly
- Texture usage
- Using the profiler
What to do and what not to do

GENERAL GUIDELINES
What to do

- Use fast math operations when possible
- Waste a register rather than divide the same value multiple times
- When multiplying/dividing by powers of two use bitshifting
- Unroll loops that have a known size
- Inline simple (1/2 line) functions
What to do

- Max # of registers set to 32 by default
  - Properties for cuda wizard or build rule
    - `maxrregcount=N`
  - Forces compiler to use less or more registers
  - Extra registers spill to local memory
  - **Good**: use 32 registers rather than 33
    - More occupancy, usually faster
  - **Bad**: use 32 registers rather than 60
    - Too much local memory usage
What not to do

- Avoid double precision math where single precision is satisfactory

- Avoid division / modulo operators if possible

- Avoid static array declarations, compiler will (almost) always use lmem
  - Used shared memory if possible
What not to do

- Avoid Inlining large pieces of code, will cause local memory to be used unnecessarily.

- Avoid complex kernels that need many registers
  - Keep kernels simple
  - Split complex kernels to reduce register pressure
Tips For debugging

- If card is compute 2.0 use printf on device
  - cuPrintf might be useful for cards <2.0
    - look in SDK for code and example

- “Invalidate” code by putting:
  \[
  \text{If(} \text{threadIdx.x} == -1) \{ \ldots \text{code here} \ldots \} \\
  \]
  - Prevents compiler from optimizing away code
  - Move statement until problem found
Tips For debugging

- Checking for execution errors:
  - `CUDA_SAFE_CALL(...);`
    - Will terminate code with reference to line of code
    - Means that something before this call went wrong
  
  - `CUT_CHECK_ERROR("ERROR MESSAGE");`
    - Prints out user specified string if something went wrong.
Compiler Info

- Compiler is smart about optimizing code
  - Takes care of register reuse
  - Combining math operations
    - Fused multiply add (MAD)
  - Delay global memory access until variable is actually used
  - Remove unused code
    - If a variable is computed but never used it gets removed at compile time
Compiler Info

- Compiler is not perfect
  - Reorganizing complex code manually can help
- Use `--ptxas-options=-v` for extra info
  - Shows info at compile time:

```
Compiling entry function '_Z8kernel_exPi' for 'sm_13'
Used 16 registers, 4 bytes lmem, 4+16 bytes smem, 4 bytes cmem[1]
```

- Useful when optimizing register usage
  - don’t need to run code to see changes
Cuda Disassembler

- Look at what the compiler actually does
  - Assembly code is a bit tricky but can be followed
- `cuobjdump.exe –dump-sass prog.exe >out.txt`
  - Write assembly to out.txt
- Useful for making sure that memory reads and writes are optimized, fast math functions are used etc.
Example kernel

- Load 4 integers in single 128 bit (16 byte) load
- Do some math in a loop
- Store 4 integers in single 128 bit write

```c
__global__ void kernel (int4* A, int reps){
    uint index=blockIdx.x*blockDim.x+threadIdx.x.x;
    for(int i=0; i<reps; i++){
        int4 temp=A[index];
        temp.x=temp.y*temp.z*temp.w;
        A[index]=temp;
    }
}
```
Example Assembly (1.0)

Function: _Z8kernelP4int4i

/*0000*/ ISET.S32.C0 o [0x7f], g [0x5], R124, LE;
/*0008*/ RET C0.NE;
/*0010*/ MOV.U16 R0H, g [0x1].U16;
/*0018*/ I2I.U32.U16 R1, R0L;
/*0020*/ IMAD.U16 R0, g [0x6].U16, R0H, R1;
/*0028*/ SHL R0, R0, 0x4;
/*0030*/ IADD R5, g [0x4], R0;
/*0038*/ IADD32I R0, R5, 0xc;
/*0040*/ GLD.U32 R4, global14 [R0];
/*0048*/ MOV R6, R124;
/*0050*/ GLD.S128 R0, global14 [R5];
/*0058*/ IMUL32.U16.R3, R0L, R1H;
/*005c*/ IMUL32.U16.R7, R4L, R2H;
/*0060*/ IMAD.U16.R3, R0H, R1L, R3;
/*0068*/ IMAD.U16.R7, R4H, R2L, R7;
/*0070*/ SHL R3, R3, 0x10;
/*0078*/ SHL R7, R7, 0x10;
/*0080*/ IMAD.U16.R0, R0L, R1L, R3;

/*0088*/ IMAD.U16.R3, R4L, R2L, R7;
/*0090*/ IMUL.U16.R7, R0L, R3H;
/*0098*/ IMAD.U16.R7, R0H, R3L, R7;
/*00a0*/ SHL.R7, R7, 0x10;
/*00a8*/ IADD32I.R6, R6, 0x1;
/*00b0*/ IMAD.U16.R0, R0L, R3L, R7;
/*00b8*/ MOV.R3, R4;
/*00c0*/ ISET.S32.C0 o [0x7f], g [0x5], R6, NE;
/*00c8*/ GST.S128.global14 [R5], R0;
/*00d0*/ BRA C0.NE, 0x50;
/*00d8*/ NOP;
Example Assembly (1.3)

Function: _Z8kernelP4int4i
/*0000*/ ISET.S32.C0 o [0x7f], g [0x5], R124, LE;
/*0008*/ RET C0.NE;
/*0010*/ G2R.U16 R0H, g [0x1].U16;
/*0018*/ I2I.U32.U16 R1, R0L;
/*0020*/ IMAD.U16 R0, g [0x6].U16, R0H, R1;
/*0028*/ SHL R0, R0, 0x4;
/*0030*/ IADD R5, g [0x4], R0;
/*0038*/ IADD32I R0, R5, 0xc;
/*0040*/ GLD.U32 R4, global14 [R0];
/*0048*/ MOV.SFU R6, R124;
/*0050*/ GLD.S128 R0, global14 [R5];
/*0058*/ IMUL32.U16.U16 R3, R0L, R1H;
/*005c*/ IMUL32.U16.U16 R7, R4L, R2H;
/*0060*/ IMAD.U16 R3, R0H, R1L, R3;
/*0068*/ IMAD.U16 R7, R4H, R2L, R7;
/*0070*/ SHL R3, R3, 0x10;
/*0078*/ SHL R7, R7, 0x10;
/*0080*/ IMAD.U16 R0, R0L, R1L, R3;
/*0088*/ IMAD.U16 R3, R4L, R2L, R7;
/*0090*/ IMUL.U16.U16 R7, R0L, R3H;
/*0098*/ IMAD.U16 R7, R0H, R3L, R7;
/*00a0*/ SHL R7, R7, 0x10;
/*00a8*/ IADD32I R6, R6, 0x1;
/*00b0*/ IMAD.U16 R0, R0L, R3L, R7;
/*00b8*/ MOV R3, R4;
/*00c0*/ ISET.S32.C0 o [0x7f], g [0x5], R6, NE;
/*00c8*/ GST.S128 global14 [R5], R0;
/*00d0*/ BRA C0.NE, 0x50;
/*00d8*/ NOP;
Branching Example

```c
__global__ void kernel(int* data) {
    if (threadIdx.x == 0) {
        data[threadIdx.x] = 1;
    }
    else if (threadIdx.x == 1) {
        data[threadIdx.x] = 2;
    }
}
```
Branching Assembly

Function: _Z8kernelPi

/*0000*/ I2I.U32.U16.C0 R0, R0L;
/*0008*/ BRA C0.NE, 0x38;
/*0010*/ SHL R1, R0, 0x2;
/*0018*/ MVI R0, 0x1;
/*0020*/ IADD R1, g [0x4], R1;
/*0028*/ GST.U32 global14 [R1], R0;
/*0030*/ RET;
/*0038*/ ISET.C0 o [0x7f], R0, c [0x1] [0x0], NE;
/*0040*/ RET C0.NE;
/*0048*/ SHL R1, R0, 0x2;
/*0050*/ MVI R0, 0x2;
/*0058*/ IADD R1, g [0x4], R1;
/*0060*/ GST.U32 global14 [R1], R0;
Easy way to speed up code

TEXTURE CACHE
The texture processor cluster

- Each TPC has several SM’s and it’s own texture memory
- No more TPC on Fermi Architecture
Texture Memory

- A method of caching global memory reads
- Uses texture cache next to SMs
- Cannot write to a texture
  - i.e writing to global memory cannot be cached

- Useful if data access is random but data is reused by different threads in the same texture processor cluster or SM
“Binding” a texture (the simple way)

- Map global memory to a texture
- Allows mapped memory to be cached
- Keyword: `cudaBindTexture(…)`
  - `cudaUnbindTexture` to free

- Memory needs to be a linear array
  - 2D arrays/textures more complicated
Simple Example:

texture<int> texData; //global scope
...
int *devData
cudaMalloc((void**) & devData, size);
cudaBindTexture( NULL, texData, devData, size);
... //Run kernel
cudaUnbindTexture(texData);
cudaFree(devData);

__global__ void kernel(...){
...=tex1Dfetch(texData,index); //access
}
Complicated method Part 1

- Necessary if using 2D textures
- Useful for image processing
  - Image is essentially a 2D matrix
- Look in SDK for more examples

```cpp
//Variable needs to be at global scope
texture<float, 2, cudaReadModeElementType> texData;

__global__ void kernel(...){
  ...=tex2D(texData, u, v);  //access element (u,v)
}```
Complicated Method Part 2

// allocate array and copy data
cudaChannelFormatDesc channelDesc = 
cudaCreateChannelDesc
(32, 0, 0, 0, cudaChannelFormatKindFloat);

cudaArray* cu_array;
cudaMallocArray( &cu_array, &channelDesc, width, height);

cudaMemcpyToArray( cu_array, 0, 0, h_data, size, 
cudaMemcpyHostToDevice);
Complicated Method Part 3

// set texture parameters
texData.addressMode[0] = cudaAddressModeWrap;
texData.addressMode[1] = cudaAddressModeWrap;
texData.filterMode = cudaFilterModeLinear;
//access with normalized texture coordinates
texData.normalized = true;

// Bind the array to the texture
cutilSafeCall( cudaBindTextureToArray(texData, cu_array, channelDesc));

//Texture read to use!!
Using the Compute Visual Profiler

PROFILING CODE
Compute Visual Profiler

- Included in CUDA SDK
- Useful tool for profiling code
- Uses the GPU’s built in counters
- Needs multiple passes
  - Each pass computes different parameters
- Only one SM is profiled
  - Some variables extrapolated
User Interface

- Plots
- Profiler Views
- Sessions
- Main Prof View
Profiler Output View

- GPU Timestamp
- Function Name
- GPU time
- CPU time
- Occupancy
- Grid/Block Size
- Shared Memory used per block
- Registers used
- Branched instructions
- Total Instructions
## Summary Table

- Shows the amount of relative time each kernel took

<table>
<thead>
<tr>
<th>Method</th>
<th>#Calls</th>
<th>GPU time</th>
<th>%GPU time</th>
<th>gld efficiency</th>
<th>gst efficiency</th>
<th>instruction throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>kernel_a</td>
<td>1</td>
<td>413750</td>
<td>60.57</td>
<td>0.516387</td>
<td>0.491375</td>
<td>0.355267</td>
</tr>
<tr>
<td>kernel_b</td>
<td>1</td>
<td>269039</td>
<td>39.38</td>
<td>0.983576</td>
<td>0.919356</td>
<td>0.54636</td>
</tr>
<tr>
<td>memcpyHtoD</td>
<td>5</td>
<td>259.584</td>
<td>0.03</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Instruction Throughput

- Alternative to Occupancy
- Ratio of achieved instruction rate to peak single issue instruction rate.
- Calculated as:
  - $\text{gpu\_time} \times \text{clock\_frequency} / \text{(instructions)}$
- Can be more than 1
Kernel, Memcopy Table Views

- Function Name
- # of Calls
- Grid / Block Size
- Shared memory per block
- Registers per thread
- Memory Transfer Type
- Memory Transfer Size
Plots

- GPU Time Summary Plot
- GPU Time Height Plot
- GPU Time Width Plot
- Comparison Plot
- Cuda API Trace Plot
End: CUDA Optimization Tips

Begin: Schedule Related Issues
Summary of Important Dates

- 03/29 – Midterm Project progress report due
- 04/06 – A one to two page PDF due, states your Final Project topic
- 04/11 – Three slides outlining your Final Project are due
- 04/13 – Midterm Project is due
  - Sample 2008 Midterm Project report available online
- 04/19 – Midterm Exam
- 05/09 – Final Project is due
- 05/10 – Individual presentations of Final Project results/outcomes
Midterm Project: Progress Report
[What’s Needed…]

- You will have to provide an overview of the algorithm that you plan to implement.

- Things that I'm interested in:
  - Flow diagrams
  - Data structures that you plan to use
  - Explain how your algorithm maps upon the underlying SIMD architecture
  - Possible limiting factors that work against your solution implementation (for instance, if all threads executing a kernel need to synchronize, or to perform atomic operations, etc.)
  - Etc.

- Indicate the use of any third party CUDA libraries such as thrust, for instance.
  - The use of existing libraries is encouraged as long as they don't completely solve your problem...
Final Project Related

- Initial plan called for each one of you to make a five minute presentation of the Final Project topic you chose

- I will be out travelling on April 12, there will no class that Tuesday

- We will have a makeup class on May 3
  - Developer from MathWorks (MATLAB) will have a two hour lecture
    - First hour: GPU Computing in MATLAB
    - Second hour: Parallel Computing Toolbox and MDCS (MATLAB Distributed Computing Server)
Final Project Related

- One to two page PDF doc with your proposal for the Final Project due on 04/06
  - Use the Learn@UW drop-box for submission

- It should contain:
  - Problem statement
  - Reasons for choosing this Final Project topic and any preliminary results
  - Summary of outcomes and deliverables at the end of the semester (your contract with me)

- Prepare a presentation that has *three slides*:
  - First slide: your name, department, and problem statement
  - Reasons for choosing this Final Project topic and any preliminary results
  - Summary of outcomes and deliverables at the end of the semester (your contract with me)

- NOTE: I will compile all your presentations in one big presentation that I will go through on April 14 (20X3=60 slides)
  - It’s important to use the same theme for the presentation, use the one I’ve been using throughout the semester (download a pptx from the class website and use it to generate your three slides…).
Schedule Highlights

- Two lectures dedicated to parallel computing using MPI
- Two lectures dedicated to parallel computing using OpenMP
- One lecture for Final Project discussions
- Midterm Exam 04/19
- Guest lectures at the end of the semester:
  - Matt Knepley – U of Chicago researcher, MPI (PETSC) related
  - Brian Davis – using cMake & Debugging CUDA
  - Ginger Ross – USAF researcher, discussion of HPC hardware, including a 500 TFlops machine USAF operates
  - Narfi Stefansson – MathWorks, GPU in MATLAB
  - Rob Farber – Pacific Northwest National Lab, GPU Computing
“A state-of-the-art calculation requires 100 hours of CPU time on the state-of-the-art computer, independent of the decade.” -- Edward Teller
Before We Get Started…

- Last lecture
  - CUDA Optimization Tips
  - Marked the end of the GPU Computing segment of the course
  - Summary of important dates

- Today
  - Start discussion of Message Passing Interface (MPI) standard
    - Discussion to be wrapped up in one more lecture
  - Learn how to run an MPI executable on Newton

- Other issues
  - Please take care of the reading assignments, they are related to CUDA programming and optimization
Distributed Memory Systems

- Individual nodes consist of a CPU, RAM, and a network interface
  - A hard disk is typically not necessary; mass storage can be supplied using NFS
- Information is passed between nodes using the network
- No need for special cache coherency hardware
- More difficult to write programs for distributed memory systems since the programmer must keep track of memory usage
- Traditionally, this represents the hardware setup that supports MPI-enabled parallel computing

Includes material from Adam Jacobs presentation
Shared Memory Systems

- Memory resources are shared among processors
- Relatively easy to program for since there is a single unified memory space
- Scales poorly with system size due to the need for cache coherence

Example:
  - Symmetric Multi-Processors (SMP)
    - Each processor has equal access to RAM

Traditionally, this represents the hardware setup that supports OpenMP-enabled parallel computing

Includes material from Adam Jacobs presentation
Overview of Large Multiprocessor Hardware Configurations

Larger multiprocessors

Shared address space
- Symmetric shared memory (SMP)
  Examples: IBM eserver, SUN Sunfire
- Distributed shared memory (DSM)

Distributed address space
- Commodity clusters: Beowulf and others
- Custom cluster
  - Uniform cluster: IBM BlueGene
  - Constellation cluster of DSMs or SMPs
    SGI Altix, ASC Purple

Cache coherent: ccNUMA: SGI Origin/Altix
Noncache coherent: Cray T3E, X1

Newton

© 2007 Elsevier, Inc. All rights reserved.

Courtesy of Elsevier, Computer Architecture, Hennessey and Patterson, fourth edition
Newton
~ Hardware Configurations ~
Hardware Relevant in the Context of MPI
Three Components of Newton that are Important

- **CPU**: Intel Xeon E5520 Nehalem 2.26GHz
  - Quad-Core Processor
  - 4 x 256KB L2 Cache
  - 8MB L3 Cache
  - LGA 1366 (Intel CPU Socket B)
  - 80W

- **HCA**: 40Gbps Mellanox MHQH19B-XTR Infiniband interconnect
  - Pretty large bandwidth compared to PCI-Ex16, yet the latency is poor
  - This is critical factor in a cluster

- **Switch**: QLogic Switch
MPI: The 30,000 Feet Perspective

- A pool of processes is started on a collection of cores
- The code executes on the CPU core (no direct role for the GPU)
- The approach is characteristic of the SPMD (Single Program Multiple Data) computing paradigm
- What differentiates processes is their rank: processes with different ranks do different things ("branching based on the process rank")
  - Very similar to GPU computing, where one thread did work based on its thread index
Why Care about MPI?

- Today, MPI is what makes the vast majority of the supercomputers tick at TFlops and PFlops rates.

- Examples of architectures relying on MPI:
  - IBM Blue Gene L/P/Q
  - Cray supercomputers (Jaguar, etc.)

- MPI known for portability and performance.

- MPI has FORTRAN, C, and C++ bindings.
MPI is a Standard

- MPI is an API for parallel programming on distributed memory systems. Specifies a set of operations, but says nothing about the implementation
  - MPI is a standard

- Popular because it many vendors support (implemented) it, therefore code that implements MPI-based parallelism is very portable

- Most common implementation: MPICH
  - The CH comes from Chameleon, the portability layer used in the original MPICH to provide portability to the existing message-passing systems
  - First MPICH implementation overseen at Argonne National Lab by Bill Gropp and Rusty Lusk
MPI Implementations

- MPI implementations available for free
  - Appleseed (UCLA)
  - CRI/EPCC (Edinburgh Parallel Computing Centre)
  - LAM/MPI (Indiana University)
  - MPI for UNICOS Systems (SGI)
  - MPI-FM (University of Illinois); for Myrinet
  - **MPICH (ANL)**
  - MVAPICH (Infiniband)
  - SGI Message Passing Toolkit
  - OpenMPI

- NOTE: MPI is available in Microsoft Visual Studio
  - The implementation offered by VS is the MPICH one

- A detailed list of MPI implementations with features can be found at [http://www.lam-mpi.org/mpi/implementations/](http://www.lam-mpi.org/mpi/implementations/)

Includes material from Adam Jacobs presentation
Where Can We Use Message Passing?

Message passing can be used wherever it is possible for processes to exchange messages:

- Distributed memory systems
- Networks of Workstations
- Even on shared memory systems
CUDA vs. MPI

- When would you use CPU/GPU computing and when would you use MPI-based parallel programming?
  - Use CPU/GPU
    - If your data fits the memory constraints associated with GPU computing
    - You have parallelism at a fine grain so that you the SIMD paradigm applies
    - Example:
      - Image processing
  - Use MPI-enabled parallel programming
    - If you have a very large problem, with a lot of data that needs to be spread out across several machines
    - Example:
      - Solving large heterogeneous multi-physics problems
      - The typical application: nuclear physics (DOE’s playground)

- In large scale computing the future likely to belong to heterogeneous architecture
  - A collection of CPU cores that communicate through MPI, each or which farming out work to an accelerator (GPU)
The Message-Passing Model

- A process is (traditionally) a program counter and address space
- Processes may have multiple threads (program counters and associated stacks) sharing a single address space
- Message passing is for communication among processes, which have separate address spaces
- Interprocess communication consists of
  - Synchronization
  - Movement of data from one process’s address space to another’s

Credit: Rusty Lusk
A First MPI Program

```c
#include "mpi.h"
#include <iostream>
#include <winsock2.h>

int main(int argc, char **argv) {
    int my_rank, n;
    char hostname[128];

    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &n);

    gethostname(hostname, 128);
    if (my_rank == 0) { /* master */
        printf("I am the master: %s\n", hostname);
    }
    else { /* worker */
        printf("I am a worker: %s (rank=%d/%d)\n", hostname, my_rank, n-1);
    }

    MPI_Finalize();
    exit(0);
}
```

Has to be called first, and once

Has to be called last, and once

Includes material from Allan Snively presentation
Program Output
Example out of Pacheco’s book:

- “Parallel Programming with MPI”
- Good book, on reserve at Wendt
# include "mpi.h"
#include <stdio.h>
#include <string.h>

int main(int argc, char* argv[]) {
    int my_rank; /* rank of process */
    int p; /* number of processes */
    int source; /* rank of sender */
    int dest; /* rank of receiver */
    int tag = 0; /* tag for messages */
    char message[100]; /* storage for message */
    MPI_Status status; /* return status for receive */

    MPI_Init(&argc, &argv); // Start up MPI
    MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); // Find out process rank
    MPI_Comm_size(MPI_COMM_WORLD, &p); // Find out number of processes

    if (my_rank != 0) {
        /* Create message */
        sprintf(message, "Greetings from process %d!", my_rank);
        dest = 0;
        /* Use strlen+1 so that '\0' gets transmitted */
        MPI_Send(message, strlen(message)+1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
    }
    else { /* my_rank == 0 */
        for (source = 1; source < p; source++) {
            MPI_Recv(message, 100, MPI_CHAR, source, tag, MPI_COMM_WORLD, &status);
            printf("%s\n", message);
        }
    }

    MPI_Finalize(); // Shut down MPI
    return 0;
} /* main */
Program Output
MPI Operations: Basic Set
[The 20/80 Rule – This set gets you very far…]

MPI can be thought of as a small specification, because any complete implementation need only provide the following operations:

- MPI_INIT
- MPI_COMM_SIZE
- MPI_COMM_RANK
- MPI_SEND
- MPI_RECV
- MPI_FINALIZE
MPI Data Types

MPI specifies data types explicitly in the message. This is needed since you might have a cluster of heterogeneous machines ⇒ word sizes and data formats may differ.

- MPI_INT
- MPI_FLOAT
- MPI_BYTE
- MPI_CHAR
- etc…
Example: Approximating $\pi$

$$\int_0^1 \frac{4}{1 + x^2} = 4 \cdot \tan^{-1}(1) = \pi$$

Numerical Integration: Midpoint rule

$$\int_0^1 \frac{4}{1 + x^2} \approx \sum_{i=1}^{n} \frac{1}{n} f \left( (i - 0.5) \cdot h \right)$$
Example: Approximating $\pi$

- Use 4 MPI process (rank 0 through 3)
- Here, $n=13$
- Sub-intervals are assigned to ranks in a round-robin manner
  - Rank 0: 1,5,9,13
  - Rank 1: 2,6,10
  - Rank 2: 3,7,11
  - Rank 3: 4,8,12
- Each rank computes the area in its associated sub-intervals
- $\text{MPI}_{\text{Reduce}}$ is used to sum the areas computed by each rank, giving final approximation to $\pi$

Credit: Toby Heyn
Code for Approximating $\pi$

// MPI_PI.cpp : Defines the entry point for the console application.  
//

#include "mpi.h"
#include <math.h>
#include <iostream>

using namespace std;

int main(int argc, char *argv[])
{
    int n, rank, size, i;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int namelen;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    MPI_Get_processor_name(processor_name, &namelen);

    cout << "Hello from process " << rank << " of " << size << " on " << processor_name << endl;
if (rank == 0) {
    //cout << "Enter the number of intervals: (0 quits) ";
    //cin >> n;
    if (argc<2 || argc>2)
        n=0;
    else
        n=atoi(argv[1]);
}

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
if (n>0) {
    h = 1.0 / (double) n;
    sum = 0.0;
    for (i = rank + 1; i <= n; i += size) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;

    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
    if (rank == 0)
        cout << "pi is approximately " << pi << ", Error is " << fabs(pi - PI25DT) << endl;
}

MPI_Finalize();
return 0;
Compiling/Running MPI under Windows with VS 2008

- On a Windows machine with VS2008, in this order, do the following:

  - Download/Install HPC Pack 2008 R2 Client Utilities Redistributable Package with Service Pack 1

  - Download/Install HPC Pack 2008 R2 SDK with Service Pack 1

  - Download/Install HPC Pack 2008 R2 MS-MPI Redistributable Package with Service Pack 1
Compiling Code

- Compile like any other project in MS Visual Studio

- Additional Include Directories:
  - C:\Program Files\Microsoft HPC Pack 2008 R2\Inc

- Additional Library Directories:
  - C:\Program Files\Microsoft HPC Pack 2008 R2\Lib\i386

- Additional Dependencies:
  - msmpi.lib

- Compile in Release mode
  - MPI_PI.exe

Credit: Toby Heyn
Overview of MPI
[Part 2 of 2]
March 29, 2011

“There are three rules to follow when parallelizing large codes. Unfortunately, no one knows what these rules are.”
W. Somerset Maugham and Gary Montry
Before We Get Started…

- Last time
  - Start discussion of Message Passing Interface (MPI) standard
  - Mentioned the components that you need to install on your machine to run MPI code

- Today
  - Learn how to run an MPI executable on Newton
  - Point-to-Point Communication with MPI
  - Collective Communication in MPI
  - Wrap up discussion of Message Passing Interface (MPI) standard

- Other issues
  - Midterm Project: progress report due tonight
  - Syllabus has been updated, up-to-date info regarding deadlines, topics covered, etc.
Running on Local Machine

mpiexec -n 4 MPI_PI.exe 13

(number of processes)  
(command line argument)  
(number of sub-intervals)

(run this in the directory where you have your executable)

Credit: Toby Heyn
Running on Newton (1 node)
Running on Newton (2 nodes)

Request exactly two nodes
Running on Newton (2 nodes)

Two cores per node

Use exactly two nodes

Credit: Toby Heyn
Running on Newton (2 nodes)

Hello from process 1 of 4 on NEWTON04.ad.engr.wisc.edu
Hello from process 0 of 4 on NEWTON06.ad.engr.wisc.edu
Hello from process 3 of 4 on NEWTON06.ad.engr.wisc.edu
Hello from process 2 of 4 on NEWTON06.ad.engr.wisc.edu
pi is approximately 3.14209, Error is 0.000493096

Note 4 processes on two nodes (NEWTON04 and NEWTON06)

Credit: Toby Heyn
Debugging MPI Code
Under VS 2008, 32 Bit

- Set MPIRun Command to “C:\Program Files\Microsoft HPC Pack 2008 R2\Bin\mpiexec.exe” (or wherever your mpiexec.exe is stored)
- Set MPIShim Location to “C:\Program Files\Microsoft Visual Studio 9.0\Common7\IDE\Remote Debugger\x86\mpishim.exe” (or wherever your mpishim.exe is stored)
Compiling MPI Code, Known Issue...

- Why do I get a compilation error "catastrophic error: #error directive: SEEK_SET is #defined but must not be for the C++ binding of MPI" when I compile C++ application?
  - Define the MPICH_IGNORE_CXX_SEEK macro at compilation stage to avoid this issue. For instance,
    - mpicc -DMPICH_IGNORE_CXX_SEEK

- Why?
  - There are name-space clashes between stdio.h and the MPI C++ binding. MPI standard requires SEEK_SET, SEEK_CUR, and SEEK_END names in the MPI namespace, but stdio.h defines them to integer values. To avoid this conflict make sure your application includes the mpi.h header file before stdio.h or iostream.h or undefine SEEK_SET, SEEK_CUR, and SEEK_END names before including mpi.h.
Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Closing Remarks
Example: Sending and Receiving
[Point-To-Point Communication]

```c
#include <mpi.h>
#include <stdio.h>

int main(int argc, char **argv) {
    int i, my_rank, nprocs, x[4];

    MPI_Init(&argc,&argv); MPI_Comm_rank(MPI_COMM_WORLD, &my_rank); MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

    if (my_rank == 0) { /* master */
        for (i=1;i<nprocs;i++)
            MPI_Send(x, 4, MPI_INT, i, 0, MPI_COMM_WORLD);
    }
    else { /* worker */
        MPI_Status status;
        MPI_Recv(x, 4, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
        if ( my_rank==1 ) {
            printf("I am worker rank=%d/%d\n", my_rank,nprocs-1);
            printf("Here's the data I recently received: %d, %d, %d, %d", x[0], x[1], x[2], x[3]);
        }
    }

    MPI_Finalize();
    exit(0);
}
```

Contains material from Allan Snavely’s presentation
Program Output

```
C:\Users\negrut\Temp\MPI\Debug>
C:\Users\negrut\Temp\MPI\Debug>
C:\Users\negrut\Temp\MPI\Debug> mpiexec.exe -n 8 MPI.exe
I am worker rank=1/8
Here's the data I recently received: 42, 43, 44, 45
C:\Users\negrut\Temp\MPI\Debug>
```
MPI Basic Send/Receive (Point-to-Point Communication)

- We need to fill in the details in

Process 0
Send(data)

Process 1
Receive(data)

- Things that need specifying:
  - How will “data” be described?
  - How will processes be identified?
  - How will the receiver recognize/screen messages?
  - What will it mean for these operations to complete?
Point-to-Point Communication

- Data to be communicated is described by three attributes:
  - address
  - data type of the message
  - length of the message

- Involved processes are described by two attributes
  - communicator
  - rank

- A message is identified by a user chosen “tag” (integer)

Credit: Allan Snavely
MPI Communicators

(default communicator)
MPI_COMM_WORLD

User-created Communicator

User-created Communicator

Credit: Allan Snavely
MPI Tags

- Messages are sent with an accompanying user-defined integer *tag*, to assist the receiving process in identifying the message.

- Messages can be screened at the receiving end by means of a specific tag, or not screened by using `MPI_ANY_TAG` as the tag in a receive.

- Some non-MPI message-passing systems have called tags “message types”. MPI calls them tags to avoid confusion with data types.
Tags and Contexts

- Separation of messages used to be accomplished by use of tags, but
  - this requires libraries to be aware of tags used by other libraries.
  - this can be defeated by use of “wild card” tags.

- Contexts are different from tags
  - no wild cards allowed
  - allocated dynamically by the system when a library sets up a communicator for its own use.

- User-defined tags still provided in MPI for user convenience in organizing application

- MPI_Comm_split can be used to create new communicators
MPI Point-to-Point Communication

- Two **modes** of communication:
  - Blocking: Communication does not complete until the message has been received
  - Non-blocking: Completes as soon as the message is “on its way”, and hopefully it gets to destination

- MPI provides four **versions** for these two modes
  - synchronous, buffered, standard, ready

Contains material from Allan Snavely’s presentation
Synchronous/Buffered Sending in MPI

- **Synchronous with MPI_Ssend**
  - In synchronous mode, a send will not complete until a matching receive is posted.
    - Copy data to the network, wait for an acknowledgement
    - The sender has to wait for a receive to be posted
    - No buffering of data

- **Buffered with MPI_Bsend**
  - The send completes once the message has been buffered internally by MPI
    - Buffering incurs an extra memory copy
    - Does not require a matching receive to be posted
    - May cause buffer overflow if many bsends and no matching receives have been posted yet

Contains material from Allan Snavely’s presentation
Standard/Ready Send

- **Standard with MPI_Send**
  - Up to MPI to decide whether to do synchronous or buffered, for performance reasons
  - The rationale is that a correct MPI program should not rely on buffering to ensure correct execution

- **Ready with MPI_Rsend**
  - May be started *only* if the matching receive has been posted
  - Can be done efficiently on some systems as no hand-shaking is required

Contains material from Allan Snavely’s presentation
There is only one MPI_Recv, which returns when the data has been received

- Performs a blocking receive operation
- C Function prototype:

```c
void MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status);
```

- buf - the initial address of the receive buffer (choice) (OUT)
- count - the number of elements to be received (integer) (IN)
- datatype - the data type of each receive buffer element (handle) (IN)
- source - the rank of the source task in comm or MPI_ANY_SOURCE (integer) (IN)
- tag - the message tag or MPI_ANY_TAG (positive integer) (IN)
- comm - the communicator (handle) (IN)
- status - the status object (INOUT).
Deadlock Issues

- Deadlock situations: appear when due to a certain sequence of commands the execution hangs

```
... PROCESS 0
MPI_Ssend()
MPI_Recv()
...
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
...

PROCESS 1
...
MPI_Ssend()
MPI_Recv()
...
...
MPI_Buffer_attach()
MPI_Bsend()
MPI_Recv()
...
...
MPI_Ssend()
MPI_Recv()
...
```

Contains material from Allan Snavely’s presentation
Recall that MPI_Send is either synchronous or buffered.

Example, on a certain machine running MPICH v1.2.1:

- **Process 0**
  - \( \ldots \)
  - `MPI_Send()`
  - `MPI_Send()`
  - `MPI_Recv()`
  - `MPI_Recv()`
  - \( \ldots \)

- **Process 1**
  - \( \ldots \)
  - `MPI_Send()`
  - `MPI_Send()`
  - `MPI_Recv()`
  - `MPI_Recv()`
  - `MPI_Recv()`
  - \( \ldots \)

Deadlock

- Data size > 127999 bytes
- Data size < 128000 bytes

No Deadlock

Contains material from Allan Snavely’s presentation
Avoiding Deadlocking

- Easy way to eliminate deadlock is to pair MPI_Ssend and MPI_Recv operations the right way:

  **PROCESS 0**
  
  ```
  ... 
  MPI_Ssend() 
  MPI_Recv() 
  ... 
  ```

  **PROCESS 1**
  
  ```
  ... 
  MPI_Recv() 
  MPI_Ssend() 
  ... 
  ```

- Conclusion: understand how the implementation works and what its pitfalls/limitations are
The Buffered Send

- This subroutine is a blocking buffered mode send operation

- It is a **local** operation. It does not depend on the occurrence of a matching receive in order to complete

- If a send operation is started and no matching receive is posted, the outgoing message is buffered to allow the send call to complete

- Return from an MPI_Bsend does not guarantee the message was sent

- Message may remain in the buffer until a matching receive is posted. MPI_Buffer_Detach() will block until all messages are received
The Buffered Send

Make sure you have enough buffer space available. An error occurs if the message must be buffered and there is not enough buffer space.

The amount of buffer space needed to be safe depends on the expected peak of pending messages. The sum of the sizes of all of the pending messages at that point plus (MPI_BSEND_OVERHEAD*number_of_messages) should be sufficient. You attach a buffer to a process using the following function:

\[
\text{int MPI_Buffer_attach(\text{void* buffer, int size});}
\]

This subroutine provides MPI a buffer in the user's memory which is used for buffering outgoing messages. This buffer is used only by messages sent in buffered mode, and only one buffer is attached to a task at any time.

MPI_Bsend adds overhead because it requires an extra memory-to-memory copy of the outgoing data.
Non-blocking Communications

- So far only blocking communication have been discussed:
  - The call returns whenever its operation is complete (MPI_SSEND returns once the message has been received, MPI_BSEND returns once the message has been buffered, etc.)

- MPI provides non-blocking communication: the call returns immediately and there is another call that can be used to check on completion.

- Rationale: Non-blocking calls let the sender/receiver do something useful while waiting for completion of the operation (without playing with threads, etc.).
Non-blocking Communication

- MPI_Issend, MPI_Ibsend, MPI_Isend, MPI_Irsend, MPI_Irecv

```c
MPI_Request request;
MPI_Isend(&x,1,MPI_INT,dest,tag,communicator,&request);
MPI_Irecv(&x,1,MPI_INT,src,tag,communicator,&request);
```

- Functions to check on completion: MPI_Wait, MPI_Test, MPI_Waitany, MPI_Testany, MPI_Waitall, MPI_Testall, MPI_Waitsome, MPI_Testsome.

```c
MPI_Status status;
MPI_Wait(&request, &status) /* blocks */
MPI_Test(&request, &status) /* doesn’t block */
```
MPI Barrier

- Synchronization of the calling processes
  - The MPI_Barrier call blocks until all of the processes belonging to certain communicator have placed the call

- No data is exchanged

```c
... MPI_Barrier(MPI_COMM_WORLD) ...
... 
```
Example: Non-blocking Point-To-Point Communication

```c
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv) {
    int my_rank, x;
    MPI_Status status;
    MPI_Request request;
    MPI_Init(&argc,&argv);
    MPI_Comm_rank(MPI_COMM_WORLD,&my_rank);
    if (my_rank == 0) { /* Process 0 */
        x=42;
        MPI_Isend(&x,1,MPI_INT,1,0,MPI_COMM_WORLD,&request);
        //..// Do some work here, but don't change x (since might not have been sent yet)
    } else if (my_rank == 1) { /* Process 1 */
        int y;
        // Do some work here, you don't need x here
        MPI_Irecv(&x,1,MPI_INT,0,0,MPI_COMM_WORLD,&request);
        // Do some work here, don't count on x yet, might not be here yet
        //....
        // Now I need x, i have to have it
        MPI_Wait(&request, &status);
        y = x/6;
        printf("I am worker rank=%d. The value of y: %d\n", my_rank, y);
    }
    MPI_Finalize();
    return 0;
}
```
Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Closing Remarks
Collective Communication

- Operations that allow more than two processes to communicate simultaneously
  - MPI broadcast
  - MPI scatter
  - MPI gather
  - MPI reduce

- All these can be built using point-to-point communications, but typical MPI implementations have optimized them, and it’s a good idea to use them.

- In all of these, all processes place the same call (in good SPMD fashion), although depending on the process, some arguments may not be used.

Contains material from Allan Snively’s presentation.
MPI Broadcast

- One-to-many communication

- Note that multicast can be implemented via the use of communicators (i.e., to create processor groups)

```c
... MPI_Bcast(x, 4, MPI_INT, 0, MPI_COMM_WORLD) ...
```

Rank of the root

Credit: Allan Snavely
MPI Scatter

- One-to-many communication
- Not sending the same message to all

```
MPI_Scatter(x, 100, MPI_INT, y, 100, MPI_INT, 0, MPI_COMM_WORLD)
```

Credit: Allan Snavely
MPI Gather

- Many-to-one communication
- Not sending the same message to the root

\[ \text{MPI\_Scatter}(x, 100, \text{MPI\_INT}, y, 100, \text{MPI\_INT}, 0, \text{MPI\_COMM\_WORLD}) \]

Credit: Allan Snavely
Gather-to-all

- Many-to-many communication
- Each process sends the same message to all
- Different processes send different messages

\[
\text{MPI\_Allgather}(x, 100, \text{MPI\_INT}, y, 100, \text{MPI\_INT}, \text{MPI\_COMM\_WORLD})
\]
All-to-all

- Many-to-many communication
- Each process sends a different message to each other process

```c
MPI_Alltoall(x, 100, MPI_INT, y, 100, MPI_INT, MPI_COMM_WORLD)
```

Credit: Allan Snavely
MPI Reduction Operations

- Used to compute a result from data that is distributed across processes
  - Often what a user wants to do anyway - why not provide the functionality as a single API call rather than having people keep re-implementing the same things

- Example: you have an large array and the first process stores the first 1000 entries, the second stores the next 1000, etc.
  - You are interested in the sum of all the elements in the array

- Predefined operations:
  - MPI_MAX, MPI_MIN, MPI_SUM, etc.

- It is possible to have user-defined operations
MPI_Reduce, MPI_Allreduce

- MPI_Reduce: result is sent out to the root
  - The operation is applied element-wise for each element of the input arrays on each processor
- MPI_Allreduce: result is sent out to everyone

```c
... MPI_Reduce(x, r, 10, MPI_INT, MPI_MAX, 0, MPI_COMM_WORLD) ...
```

input array  output array  array size

```c
... MPI_Allreduce(x, r, 10, MPI_INT, MPI_MAX, MPI_COMM_WORLD) ...
```

Credit: Allan Snavely
MPI_Reduce example

MPI_Reduce(sbuf, rbuf, 6, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD)

sbuf

P0  3 4 2 8 12 1

P1  5 2 5 1 7 11

P2  2 4 4 10 4 5

P3  1 6 9 3 1 1

rbuf

P0  11 16 20 22 24 18
**MPI_Scan: Prefix reduction**

- Process i receives data reduced on process 0 to i.

```latex
\textbf{MPI\_Scan}\texttt{(sbuf, rbuf, 6, MPI\_INT, MPI\_SUM, MPI\_COMM\_WORLD)}
```

Credit: Allan Snavely
And more...

- Most broadcast operations come with a version that allows for a stride (so that blocks do not need to be contiguous)
  - MPI_Gatherv(), MPI_Scatterv(), MPI_Allgatherv(), MPI_Alltoallv()

- MPI_Reduce_scatter(): functionality equivalent to a reduce followed by a scatter

- All the above have been created as they are common in scientific applications and save code

- All details on the MPI Webpage

Credit: Allan Snively
Outline

- Introduction to message passing and MPI
- Point-to-Point Communication
- Collective Communication
- MPI Closing Remarks
In some MPI implementations there are more than 300 MPI functions. Not all of them part of the MPI standard though, some vendor specific.

Recall the 20/80 rule: six basic calls is probably what you need to implement a decent MPI code:

- MPI_Init, MPI_Comm_Size, MPI_Comm_Rank, MPI_Send, MPI_Recv, MPI_Finalize
The PETSc Library

- PETSc: Portable, Extensible Toolkit for Scientific Computation
  - One of the most successful libraries built on top of MPI
  - Intended for use in large-scale application projects,
  - Developed at Argonne National Lab (Barry Smith)

- PETSc provides routines for the parallel solution of systems of equations that arise from the discretization of PDEs
  - Linear systems
  - Nonlinear systems
  - Time evolution

- PETSc also provides routines for
  - Sparse matrix assembly
  - Distributed arrays
  - General scatter/gather (e.g., for unstructured grids)
Structure of PETSc

- **PETSc PDE Numerical Solution Utilities**
  - ODE Integrators
  - Visualization
  - Nonlinear Solvers, Unconstrained Minimization
  - Interface
  - Linear Solvers
  - Preconditioners + Krylov Methods
  - Grid Management
  - Object-Oriented Matrices, Vectors, Indices

- **Profiling Interface**
- **Computation and Communication Kernels**
  - MPI, MPI-IO, BLAS, LAPACK
# PETSc Numerical Components

## Nonlinear Solvers
- Newton-based Methods
- Line Search
- Trust Region
- Other

## Time Steppers
- Euler
- Backward Euler
- Pseudo Time Stepping
- Other

## Krylov Subspace Methods
- GMRES
- CG
- CGS
- Bi-CG-STAB
- TFQMR
- Richardson
- Chebychev
- Other

## Preconditioners
- Additive Schwartz
- Block Jacobi
- Jacobi
- ILU
- ICC
- LU (Sequential only)
- Others

## Matrices
- Compressed Sparse Row (AIJ)
- Blocked Compressed Sparse Row (BAIJ)
- Block Diagonal (BDIAG)
- Dense
- Matrix-free
- Other

## Distributed Arrays

## Index Sets
- Indices
- Block Indices
- Stride
- Other

## Vectors

---

47
Flow Control for PDE Solution

Main Routine

Timestepping Solvers (TS)

Nonlinear Solvers (SNES)

Linear Solvers (SLES)

PC

KSP

PETSc

Application Initialization

Function Evaluation

Jacobian Evaluation

Post-Processing

User code

PETSc code
Parallel Computing using OpenMP

[Part 1 of 2]

March 31, 2011

“The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague.”

Edsger W. Dijkstra
Before We Get Started...

- **Last time**
  - Learn how to run an MPI executable on Newton
  - Point-to-Point Communication with MPI
  - Collective Communication in MPI

- **Today**
  - Parallel Computing using OpenMP, part 1 of 2.

- **Other issues**
  - Assignment 7 was posted on the class website, due on April 7
  - Class website includes link to the OpenMP 3.0 Application Programming Interface
Acknowledgements

- The overwhelming majority of slides used for discussing OpenMP issues are from Intel’s library of presentations for promoting OpenMP
  - The slides are used herein with permission

- Credit is given where due by a “Credit: IOMPP” or “Includes material from IOMPP” message at the bottom of the slide
  - IOMPP stands for “Intel OpenMP Presentation”
Data vs. Task Parallelism

- **Data parallelism**
  - You have a large amount of data elements and each data element (or possibly a subset of elements) needs to be processed to produce a result
  - When this processing can be done in parallel, we have data parallelism
  - Example:
    - Adding two long arrays of doubles to produce yet another array of doubles

- **Task parallelism**
  - You have a collection of tasks that need to be completed
  - If these tasks can be performed in parallel you are faced with a task parallel job
  - Examples:
    - Reading the newspaper, drinking coffee, and scratching your back
    - The breathing your lungs, beating of your heart, liver function, controlling the swallowing, etc.
Objectives

- Understand OpenMP at the level where you can
  - Implement data parallelism
  - Implement task parallelism
Work Plan

● What is OpenMP?
  Parallel regions
  Work sharing
  Data environment
  Synchronization

● Advanced topics
OpenMP: Target Hardware

- **CUDA**: targeted parallelism on the GPU

- **MPI**: targeted parallelism on a cluster (distributed computing)
  - Note that MPI implementation can handle transparently a SMP architecture such as a workstation with two hexcore CPUs that use a large amount of shared memory

- **OpenMP**: targets parallelism on SMP architectures
  - Handy when
    - You have a machine that has 12 cores, probably 24 if HTT is accounted for
    - You have a large amount of shared memory that is backed by a 64 bit OS
OpenMP: What to Expect

- If you have 12 cores available to you, it is *highly* unlikely to get a speedup of more than 12 (superlinear)

- Recall the trick that helped the GPU hide latency
  - Overcommitting the SPs and hiding memory access latency with warp execution

- This mechanism of hiding latency by overcommitment does not *explicitly* exist for parallel computing under OpenMP beyond what’s offered by HTT
OpenMP: What Is It?

- Portable, shared-memory threading API
  - Fortran, C, and C++
  - Multi-vendor support for both Linux and Windows
- Standardizes task & loop-level parallelism
- Supports coarse-grained parallelism
- Combines serial and parallel code in single source
- Standardizes ~ 20 years of compiler-directed threading experience

- Current spec is OpenMP 3.0
  - http://www.openmp.org
  - 318 Pages
“pthreads”: An OpenMP Precursor

- Before there was OpenMP, a common approach to support parallel programming was by use of pthreads
  - “pthread”: POSIX thread
  - POSIX: Portable Operating System Interface [for Unix]

- pthreads
  - Available originally under Unix and Linux
  - Windows ports are also available some as open source projects

- Parallel programming with pthreads: relatively cumbersome, prone to mistakes, hard to maintain/scale/expand
  - Moreover, not envisioned as a mechanism for writing scientific computing software
```c
int main(int argc, char *argv[]) {
    parm *arg;
    pthread_t *threads;
    pthread_attr_t pthread_custom_attr;
    int n = atoi(argv[1]);
    threads = (pthread_t *) malloc(n * sizeof(*threads));
    pthread_attr_init(&pthread_custom_attr);
    barrier_init(&barrier1); /* setup barrier */
    finals = (double *) malloc(n * sizeof(double)); /* allocate space for final result */

    arg=(parm *)malloc(sizeof(parm)*n);
    for( int i = 0; i < n; i++) { /* Spawn thread */
        arg[i].id = i;
        arg[i].nproc = n;
        pthread_create(&threads[i], &pthread_custom_attr, cpi, (void *)(arg+i));
    }

    for( int i = 0; i < n; i++) /* Synchronize the completion of each thread. */
        pthread_join(threads[i], NULL);

    free(arg);
    return 0;
}
```
# include <stdio.h>
# include <math.h>
# include <time.h>
# include <sys/types.h>
# include <pthread.h>
# include <sys/time.h>

#define SOLARIS 1
#define ORIGIN 2
#define OS SOLARIS

typedef struct {
    int id;
    intnoproc;
    intdim;
} parm;

typedef struct {
    intcur_count;
    pthread_mutex_t barrier_mutex;
    pthread_cond_t barrier_cond;
} barrier_t;

void barrier_init(barrier_t * mybarrier) {
    /* barrier */
    /* must run before spawning the thread */
    pthread_mutexattr_t attr;

    # if (OS==ORIGIN)
    pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);
    pthread_mutexattr_setprioceiling(&attr, 0);
    pthread_mutex_init(&(mybarrier->barrier_mutex), &attr);
    # elif (OS==SOLARIS)
    pthread_mutex_init(&(mybarrier->barrier_mutex), NULL);
    # else
    # error "undefined OS"
    # endif
    pthread_cond_init(&(mybarrier->barrier_cond), NULL);
    mybarrier->cur_count = 0;
}

void barrier(int numproc, barrier_t * mybarrier) {
    pthread_mutex_lock(&mybarrier->barrier_mutex);
    mybarrier->cur_count++;
    if (mybarrier->cur_count!=numproc) {
        pthread_cond_wait(&mybarrier->barrier_cond, &mybarrier->barrier_mutex);
    } else {
        mybarrier->cur_count=0;
        pthread_cond_broadcast(&mybarrier->barrier_cond);
    }
    pthread_mutex_unlock(&mybarrier->barrier_mutex);
}

typedef struct {
    int id;
    intnoproc;
    intdim;
} parm;

typedef struct {
    intcur_count;
    pthread_mutex_t barrier_mutex;
    pthread_cond_t barrier_cond;
} barrier_t;

void barrier_init(barrier_t * mybarrier) {
    /* barrier */
    /* must run before spawning the thread */
    pthread_mutexattr_t attr;

    # if (OS==ORIGIN)
    pthread_mutexattr_setprotocol(&attr, PTHREAD_PRIO_INHERIT);
    pthread_mutexattr_setprioceiling(&attr, 0);
    pthread_mutex_init(&(mybarrier->barrier_mutex), &attr);
    # elif (OS==SOLARIS)
    pthread_mutex_init(&(mybarrier->barrier_mutex), NULL);
    # else
    # error "undefined OS"
    # endif
    pthread_cond_init(&(mybarrier->barrier_cond), NULL);
    mybarrier->cur_count = 0;
}

void barrier(int numproc, barrier_t * mybarrier) {
    pthread_mutex_lock(&mybarrier->barrier_mutex);
    mybarrier->cur_count++;
    if (mybarrier->cur_count!=numproc) {
        pthread_cond_wait(&mybarrier->barrier_cond, &mybarrier->barrier_mutex);
    } else {
        mybarrier->cur_count=0;
        pthread_cond_broadcast(&mybarrier->barrier_cond);
    }
    pthread_mutex_unlock(&mybarrier->barrier_mutex);
}

void * cpi(void *arg) {
    parm *p = (parm *) arg;
    int myid = p->id;
    int numprocs = p->noproc;
    double PI25DT = 3.141592653589793238462643;
    double mypi, pi, h, sum, x, a;
    double startwtime, endwtime;

    if (myid == 0) { 
        startwtime = clock();
    }
    barrier(numprocs, &barrier1);
    if (rootn==0)
        finals[myid]=0;
    else {
        h = 1.0 / (double) rootn;
        sum = 0.0;
        for(int i = myid + 1; i <=rootn; i += numprocs) {
            x = h * ((double) i - 0.5);
            sum += f(x);
        } 
        mypi = h * sum;
    }
    finals[myid] = mypi;
    barrier(numprocs, &barrier1);

    if (myid == 0) { 
        pi = 0.0;
        for(int i=0; i < numprocs; i++) pi += finals[i];
        endtime = clock();
        printf("pi is approx %.16f, Error is %.16f\n", pi, fabs(pi - PI25DT));
        printf("wall clock time = %f\n",
               (endtime - startwtime) / CLOCKS_PER_SEC);
    }
    return NULL;
}
“pthreads”: Moving Away…

- Looking at the previous example (which is not the best written piece of code, lifted from the web…)
  - Code displays platform dependency (not portable)
  - Code is cryptic, low level, hard to read (not simple)
  - Requires busy work: fork and joining threads, etc.
    - Burdens the developer
    - Probably in the way of the compiler as well: rather low chances that the compiler will be able to optimize the implementation

- Long time experience with “pthreads” suggested that a higher level approach to SMP parallel computing for *scientific applications* was in order
OpenMP Programming Model

- **Master thread** spawns a **team of threads** as needed
  - Managed transparently on your behalf
  - It still relies on thread fork/join methodology to implement parallelism
    - The developer is spared the details

- Parallelism is added incrementally: that is, the sequential program evolves into a parallel program

Includes material from IOMPP
OpenMP: 20+ Library Routines

- Runtime environment routines:
  - Modify/check the number of threads
    - `omp_[set|get]_num_threads()`
    - `omp_get_thread_num()`
    - `omp_get_max_threads()`
  - Are we in a parallel region?
    - `omp_in_parallel()`
  - How many processors in the system?
    - `omp_get_num_procs()`
  - Explicit locks
    - `omp_[set|unset]_lock()`
  - And several more...
A Few Syntax Details to Get Started

- Most of the constructs in OpenMP are compiler directives or pragmas
  - For C and C++, the pragmas take the form:
    
    #pragma omp construct [clause [clause]...]
  
  - For Fortran, the directives take one of the forms:
    
    C$OMP construct [clause [clause]...]  
    !$OMP construct [clause [clause]...]  
    *$OMP construct [clause [clause]...]  

- Header file or Fortran 90 module
  
  #include “omp.h”  
  use omp_lib
Why Compiler Directive and/or Pragmas?

- One of OpenMP’s design principles was to have the same code, with no modifications and have it run either on one core machine, or a multiple core machine.
- Therefore, you have to “hide” all the compiler directives behind Comments and/or Pragmas.
- These hidden directives would be picked up by the compiler only if you instruct it to compile in OpenMP mode.
  - Example: Visual Studio – you have to have the /openmp flag on in order to compile OpenMP code.
  - Also need to indicate that you want to use the OpenMP API by having the right header included: #include <omp.h>.

---

Step 1: Go here

Step 2: Select /openmp
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data environment
  - Synchronization

- Advanced topics
Most OpenMP constructs apply to structured blocks

- Structured block: a block with one point of entry at the top and one point of exit at the bottom
- The only “branches” allowed are STOP statements in Fortran and exit() in C/C++

```c
#pragma omp parallel
{
    int id = omp_get_thread_num();
    more: res[id] = do_big_job(id);
    if (conv (res[id]) goto more;
}
printf ("All done\n");
```

```c
if (go_now()) goto more;
#pragma omp parallel
{
    int id = omp_get_thread_num();
    more: res[id] = do_big_job(id);
    if (conv (res[id]) goto done;
        goto more;
}
```

A structured block

```c
done: if (!really_done()) goto more;
```

Not a structured block
```c
#include <stdio.h>
#include <omp.h>

int main() {
    #pragma omp parallel
    {
        int myId = omp_get_thread_num();
        int nThreads = omp_get_num_threads();

        printf("Hello World. I'm thread %d out of %d.\n", myId, nThreads);
        for (int i=0; i<2 ;i++ )
            printf("Iter:%d\n",i);
    }
    printf("GoodBye World\n");
}
```

Example: Hello World on my Machine

• Here’s my machine (12 core machine)

Two Intel Xeon X5650 Westmere 2.66GHz
12MB L3 Cache LGA 1366 95Watts Six-Core Processors

Credit: OpenMP code from IOMPP
One of the key tenets of OpenMP is that of data independence across parallel jobs.

Specifically, when distributing work among parallel threads it is assumed that there is no data dependency.

Since you place the `omp parallel` directive around some code, it is your responsibility to make sure that data dependency is ruled out.

Compilers are not smart enough and sometimes it is outright impossible to rule out data dependency between what might look as independent parallel jobs.
Work Plan

- What is OpenMP?
  Parallel regions
  Work sharing – Parallel For
  Data environment
  Synchronization
- Advanced topics
Work Sharing

● **Work sharing** is the general term used in OpenMP to describe distribution of work across threads

● Three categories of worksharing in OpenMP:
  ● “omp for” construct
  ● “omp sections” construct
  ● “omp task” construct

Automatically divides work among threads
"omp for" construct

// assume N=12
#pragma omp parallel
#pragma omp for
    for(i = 1, i < N+1, i++)
        c[i] = a[i] + b[i];

- Threads are assigned an independent set of iterations
- Threads must wait at the end of work-sharing construct
Combining Constructs

- These two code segments are equivalent

```c
#pragma omp parallel
{
    #pragma omp for
    for (int i=0; i< MAX; i++) {
        res[i] = huge();
    }
}
```

```c
#pragma omp parallel for
for (int i=0; i< MAX; i++) {
    res[i] = huge();
}
```
The Private Clause

- Reproduces the variable for each task
  - Variables are un-initialized; C++ object is default constructed
  - Any value external to the parallel region is undefined
  - By declaring a variable as being private it means that each thread will have a private copy of that variable
    - The value that thread 1 stores in x is different than the value that thread 2 stores in the variable x

```c
void* work(float* c, int N) {
    float x, y; int i;
    #pragma omp parallel for private(x,y)
    for(i=0; i<N; i++) {
        x = a[i]; y = b[i];
        c[i] = x + y;
    }
}
```
Example: Parallel Mandelbrot

- Objective: create a parallel version of Mandelbrot using OpenMP work sharing clauses to parallelize the computation of Mandelbrot.

Includes material from IOMPP
Example: Parallel Mandelbrot
[The Important Function; Includes material from IOMPP]

```c
int Mandelbrot (float z_r[][JMAX], float z_i[][JMAX], float z_color[][JMAX], char gAxis ){
    float xinc = (float)XDELTA/(IMAX-1);
    float yinc = (float)YDELTA/(JMAX-1);

    #pragma omp parallel for private(i,j) schedule(static,8)
    for (int i=0; i<IMAX; i++) {
        for (int j=0; j<JMAX; j++) {
            z_r[i][j] = (float) -1.0*XDELTA/2.0 + xinc * i;
            z_i[i][j] = (float) 1.0*YDELTA/2.0 - yinc * j;
            switch (gAxis) {
                case 'V':
                    z_color[i][j] = CalcMandelbrot(z_r[i][j], z_i[i][j] ) /1.0001;
                    break;
                case 'H':
                    z_color[i][j] = CalcMandelbrot(z_i[i][j], z_r[i][j] ) /1.0001;
                default:
                    break;
            }
        }
    }
    return 1;
}
```
The schedule Clause

- The `schedule` clause affects how loop iterations are mapped onto threads

**schedule(static [,chunk])**
- Blocks of iterations of size “chunk” to threads
- Round robin distribution
- Low overhead, may cause load imbalance

**schedule(dynamic [,chunk])**
- Threads grab “chunk” iterations
- When done with iterations, thread requests next set
- Higher threading overhead, can reduce load imbalance

**schedule(guided [,chunk])**
- Dynamic schedule starting with large block
- Size of the blocks shrink; no smaller than “chunk”
schedule Clause Example

```
#pragma omp parallel for schedule (static, 8)
for( int i = start; i <= end; i += 2 )
{
    if ( TestForPrime(i) ) gPrimesFound++;
}
```

- Iterations are divided into chunks of 8
- If start = 3, then first chunk is

\[ i = \{3, 5, 7, 9, 11, 13, 15, 17\} \]
Work Plan

- What is OpenMP?
  Parallel regions
  Work sharing – Parallel Sections
  Data environment
  Synchronization

- Advanced topics
Function Level Parallelism

a = alice();
b = bob();
s = boss(a, b);
c = cy();
printf ("%.2f\n", bigboss(s,c));

alice, bob, and cy can be computed in parallel.
omp sections

- `#pragma omp sections`
- Must be inside a parallel region
- Precedes a code block containing $N$ sub-blocks of code that may be executed concurrently by $N$ threads
- Encompasses each omp section

- `#pragma omp section`
- Precedes each sub-block of code within the encompassing block described above
- Enclosed program segments are distributed for parallel execution among available threads

Credit: IOMPP
#pragma omp parallel sections
{
    #pragma omp section
    double a = alice();
    #pragma omp section
    double b = bob();
    #pragma omp section
    double c = cy();

    double s = boss(a, b);
    printf("%6.2f\n", bigboss(s, c));
}
Advantage of Parallel Sections

- Independent sections of code can execute concurrently – reduce execution time

```c
#pragma omp parallel sections
{
#pragma omp section
  phase1();
#pragma omp section
  phase2();
#pragma omp section
  phase3();
}
```

Credit: IOMPP
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing – Tasks
  - Data environment
  - Synchronization

- Advanced topics
New Addition to OpenMP

- **Tasks** – Main change for in the latest 3.0 version of OpenMP

- Allows parallelization of irregular problems
  - Unbounded loops
  - Recursive algorithms
  - Producer/consumer

Credit: IOMPP
Tasks: What Are They?

- Tasks are independent units of work
- A thread is assigned to perform a task
- Tasks might be executed immediately or might be deferred
  - The runtime system decides which of the above
- Tasks are composed of
  - **code** to execute
  - **data** environment
  - **internal control variables** (ICV)

Credit: IOMPP
Simple Task Example

```c
#include <omp.h>

#define NODE_SIZE 8

node* head_of_list = NULL;
node* end_of_list = NULL;

int main() {
    #pragma omp parallel
    // assume 8 threads
    {
        #pragma omp single private(p)
        {
            // some computation here...
            node *p = head_of_list;
            while( p != end_of_list ) {
                #pragma omp task
                {
                    processwork(p);
                }
                p = p->next;
            }
        }
    }
    return 0;
}
```

A pool of 8 threads is created here

Only one thread gets to execute the while loop

The single “while loop” thread creates a task for each instance of processwork()
Task Construct – Explicit Task View

- A team of threads is created at the `omp parallel` construct
- A single thread is chosen to execute the while loop – call this thread “L”
- Thread L operates the while loop, creates tasks, and fetches next pointers
- Each time L crosses the `omp task` construct it generates a new task and has a thread assigned to it
- Each task runs in its own thread
- All tasks complete at the barrier at the end of the parallel region’s construct

```
#pragma omp parallel
{
    #pragma omp single
    {
        node *p = head_of_list;
        while (p) {
            #pragma omp task private(p)
            process(p);
            p = p->next;
        }
    }
}
```

Credit: IOMPP
Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```c
#pragma omp parallel
{
  #pragma omp single
  { // block 1
    node *p = head_of_list;
    while (p) { //block 2
      #pragma omp task private(p)
      process(p);
      p = p->next; //block 3
    }
  }
}
```

Includes material from IOMPP
Tasks: Synchronization Issues

- **Setup:**
  - Assume Task B specifically relies on completion of Task A
  - You need to be in a position to guaranteed completion of Task A before invoking the execution of Task B

- Tasks are guaranteed to be complete at thread or task barriers:
  - At the directive: `#pragma omp barrier`
  - At the directive: `#pragma omp taskwait`
Task Completion Example

```
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

- Multiple foo tasks created here – one for each thread
- All foo tasks guaranteed to be completed here
- One bar task created here
- bar task guaranteed to be completed here

Task Completion Example

bar task guaranteed to be completed here

Multiple foo tasks created here – one for each thread

All foo tasks guaranteed to be completed here

One bar task created here

bar task guaranteed to be completed here
Parallel Computing using OpenMP

[Part 2 of 2]

April 5, 2011

“The inside of a computer is as dumb as hell but it goes like mad!”

Richard Feynman
Before We Get Started…

● Last time
  ● General intro, OpenMP
  ● Parallel regions
  ● Work sharing under OpenMP
    ● omp for
    ● omp sections
    ● omp tasks

● Today
  ● Parallel Computing using OpenMP, part 2 of 2.

● Other issues
  ● Assignment 7 due on April 7
  ● Thursday I’ll finish what I planned to lecture for ME964
  ● Beyond that:
    ● No class next Tuesday
    ● Recall that you have to send me a PPT with your Final Project topic (see syllabus for due date)
    ● We’ll have several guest lecturers
    ● Midterm Exam on April 19
Why are tasks useful?

Have potential to parallelize irregular patterns and recursive function calls

```c
#pragma omp parallel
{
    #pragma omp single
    {
        // block 1
        node *p = head_of_list;
        while (p) {
            #pragma omp task private(p)
            process(p);
            p = p->next;  //block 3
        }
    }
}
```

Includes material from IOMPP
Tasks: Putting Things in Perspective

- Upper pic: sequential. Lower pic: parallel
Tasks: Synchronization Issues

- **Setup:**
  - Assume Task B specifically relies on completion of Task A
  - You need to be in a position to guaranteed completion of Task A before invoking the execution of Task B

- Tasks are guaranteed to be complete at thread or task barriers:
  - At the directive: `#pragma omp barrier`
  - At the directive: `#pragma omp taskwait`
Task Completion Example

```c
#pragma omp parallel
{
    #pragma omp task
    foo();
    #pragma omp barrier
    #pragma omp single
    {
        #pragma omp task
        bar();
    }
}
```

- Multiple `foo()` tasks created here – one for each thread
- All `foo()` tasks guaranteed to be completed here
- One `bar()` task created here
- `bar()` task guaranteed to be completed here

Credit: IOMPP
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data scoping
  - Synchronization

- Advanced topics
Data Scoping – What’s shared

- OpenMP uses a shared-memory programming model

- **Shared variable** - a variable that can be read or written by multiple threads

- Shared clause can be used to make items explicitly shared
  - Global variables are shared by default among tasks
  - Other examples of variables being shared among threads
    - File scope variables
    - Namespace scope variables
    - Variables with const-qualified type having no mutable member
    - Static variables which are declared in a scope inside the construct

Includes material from IOMPP
Data Scoping – What’s Private

- Not everything is shared...
  - Examples of implicitly determined PRIVATE variables:
    - Stack (local) variables in functions called from parallel regions
    - Automatic variables within a statement block
    - Loop iteration variables
    - Implicitly declared private variables within tasks will be treated as firstprivate

- firstprivate
  - Specifies that each thread should have its own instance of a variable, and that the variable should be initialized with the value of the variable, because it exists before the parallel construct

Includes material from IOMPP
A Data Environment Example

float A[10];
main () {
    int index[10];
    #pragma omp parallel
    {
        Work (index);
    }
    printf ("%d\n", index[1]);
}

extern float A[10];
void Work (int *index) {
    float temp[10];
    static integer count;
    <...>
}

A, index, and count are shared by all threads, but temp is local to each thread

Includes material from IOMPP
Data Scoping Issue: fib Example

Values of the private variables not available outside of tasks

```
int fib ( int n ) {

    int x, y;
    if ( n < 2 ) return n;
#pragma omp task
    x = fib(n-1);
#pragma omp task
    y = fib(n-2);
#pragma omp taskwait

    return x+y
}
```

- **n** is private in both tasks
- **x** is a private variable
- **y** is a private variable

What’s wrong here?

Credit: IOMPP
Data Scoping Issue: fib Example

```c
int fib ( int n ) {
    int x, y;
    if ( n < 2 ) return n;
    #pragma omp task shared(x)
    x = fib(n-1);
    #pragma omp task shared(y)
    y = fib(n-2);
    #pragma omp taskwait

    return x+y
}
```

- **n** is private in both tasks
- **x** & **y** are shared

**Good solution**
we need both values to compute the sum

The values of the **x** & **y** variables will be available outside each task construct – after the taskwait

Credit: IOMPP
Discussion: Variable Scoping Aspects

- Consider parallelizing the following code

```c
int main() {
    const int n=20;
    int a[n];
    for( int i=0; i<n; i++ )
        a[i] = i;

    //this is the part that needs to
    //be parallelized
    caller(a, n);

    for( int i=0; i<n; i++ )
        printf("a[%d]=%d\n", i, a[i]);
    return 0;
}

void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```
Program Output

- Looks good
  - The value of the counter increases each time you hit the “callee” subroutine

- If you run the executable 20 times, you get the same results 20 times
First Attempt to Parallelize

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}
void caller(int *a, int n) {
    int i, j, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```

<table>
<thead>
<tr>
<th>Var</th>
<th>Scope</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>n</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>i</td>
<td>private</td>
<td>Parallel loop index</td>
</tr>
<tr>
<td>j</td>
<td>shared</td>
<td>Declared outside parallel construct</td>
</tr>
<tr>
<td>m</td>
<td>shared</td>
<td>Constant decl. outside parallel construct</td>
</tr>
<tr>
<td>k</td>
<td>private</td>
<td>Automatic variable/parallel region</td>
</tr>
<tr>
<td>x</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*x</td>
<td>shared</td>
<td>(actually a)</td>
</tr>
<tr>
<td>y</td>
<td>private</td>
<td>Passed by value</td>
</tr>
<tr>
<td>*y</td>
<td>private</td>
<td>(actually k)</td>
</tr>
<tr>
<td>z</td>
<td>private</td>
<td>(actually j)</td>
</tr>
<tr>
<td>ii</td>
<td>private</td>
<td>Local stack variable in called function</td>
</tr>
<tr>
<td>cv</td>
<td>shared</td>
<td>Declared static (like global)</td>
</tr>
</tbody>
</table>
Program Output, First Attempt to Parallelize

- Looks bad…
  - The values in array “a” are all over the map
  - The value of the counter “cv” changes chaotically within “callee”
  - The function “callee” gets hit a random number of times (should be hit 100 times). Example:
    ```
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 70
    ```
- If you run executable 20 times, you get different results
- One of the problems is that “j” is shared
Second Attempt to Parallelize

- Declare the inner loop variable “j” as a private variable within the parallel loop

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, j, m=3;
    #pragma omp parallel for private(j)
    for (i=0; i<n; i++) {
        int k=m;
        for (j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```
Program Output, Second Attempt to Parallelize

- Looks better
  - The values in array “a” are correct
  - The value of the counter “cv” changes strangely within the “callee” subroutine
  - The function “callee” gets hit 100 times:
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 100

- If you run executable 20 times, you get good results for “a”, but the static variable will continue to behave strangely (it’s shared)
  - Fortunately, it’s not used in this code for any subsequent computation

- Conclusion: be careful when you work with static or some other global variables in parallel programming
  - In general, dealing with such variables is bad programming practice
Slightly Better Solution…

- Declare the inner loop index “j” only inside the parallel segment
  - After all, it’s only used there
  - You get rid of the “private” attribute, less constraints on the code, increasing the opportunity for code optimization at compile time

```c
void callee(int *x, int *y, int z) {
    int ii;
    static int cv=0;
    cv++;
    for (ii=1; ii<z; ii++) {
        *x = *x + *y + z;
    }
    printf("Value of counter: %d\n", cv);
}

void caller(int *a, int n) {
    int i, m=3;
    #pragma omp parallel for
    for (i=0; i<n; i++) {
        int k=m;
        for (int j=1; j<=5; j++) {
            callee(&a[i], &k, j);
        }
    }
}
```

Used here, then you should declare here (common sense…)}
Program Output, Parallelized Code

- Looks good
  - The values in array “a” are correct
  - The value of the counter “cv” changes strangely within the “callee” subroutine
  - The function “callee” gets hit 100 times:
    ```
    # parallelGood.exe | grep "Value of counter" | wc -l
    # 100
    ```

- If you run executable 20 times, you get good results for “a”, but the static variable will continue to behave strangely (it’s shared)

- What surprised me: the value of the counter was indeed 100
  - In other words, although shared, no trashing of this variable…
Work Plan

What is OpenMP?
Parallel regions
Work sharing
Data environment
Synchronization

Advanced topics
Implicit Barriers

- Several OpenMP constructs have implicit barriers
  - parallel – necessary barrier – cannot be removed
  - for
  - single

- Unnecessary barriers hurt performance and can be removed with the `nowait` clause
  - The `nowait` clause is applicable to:
    - for clause
    - single clause
Nowait Clause

- Use when threads unnecessarily wait between independent computations

```c
#pragma omp for nowait
for(...)
{ [... ] };
```

```c
#pragma omp for schedule(dynamic,1) nowait
for(int i=0; i<n; i++)
    a[i] = bigFunc1(i);
```

```c
#pragma omp for schedule(dynamic,1)
for(int j=0; j<m; j++)
    b[j] = bigFunc2(j);
```

Credit: IOMPP
Barrier Construct

- Explicit barrier synchronization
- Each thread waits until all threads arrive

```c
#pragma omp parallel shared(A, B, C)
{
    DoSomeWork(A,B); // Processed A into B
    #pragma omp barrier
    DoSomeWork(B,C); // Processed B into C
}
```
Atomic Construct

- Applies only to simple update of memory location
- Special case of a critical section, to be discussed shortly

```c
#pragma omp parallel for shared(x, y, index, n)
for (i = 0; i < n; i++) {
    #pragma omp atomic
    x[index[i]] += work1(i);
    y[i] += work2(i);
}
```

index[0] = 2;
index[1] = 3;
index[2] = 4;
index[3] = 0;
index[4] = 5;
index[5] = 5;
index[6] = 5;
index[7] = 1;

Credit: IOMPP
```c
float dot_prod(float* a, float* b, int N)
{
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
    return sum;
}
```

What is Wrong?
Race Condition

- A *race condition* is nondeterministic behavior caused by the times at which two or more threads access a shared variable.

- For example, suppose both Thread A and Thread B are executing the statement:
  
  ```
  area += 4.0 / (1.0 + x*x);
  ```

*Credit: IOMPP*
Two Possible Scenarios

Order of thread execution causes non-determinant behavior in a data race

Credit: IOMPP
Protect Shared Data

- Must protect access to shared, modifiable data

```c
float dot_prod(float* a, float* b, int N) {
    float sum = 0.0;
    #pragma omp parallel for shared(sum)
    for(int i=0; i<N; i++) {
        #pragma omp critical sum += a[i] * b[i];
    }
    return sum;
}
```

Credit: IOMPP
OpenMP Critical Construct

```
#pragma omp critical [(lock_name)]

- Defines a critical region on a structured block

float RES;
#pragma omp parallel
{
  #pragma omp for
    for(int i=0; i<niters; i++){
      float B = big_job(i);
      #pragma omp critical (RES_lock)
      consum(B, RES);
    }
}
```

Threads wait their turn – only one at a time calls consum() thereby protecting RES from race conditions.

Naming the critical construct RES_lock is optional but highly recommended.

Good Practice – Name all critical sections

Includes material from IOMPP
OpenMP Reduction Clause

reduction (op : list)

- The variables in “list” must be shared in the enclosing parallel region

- Inside parallel or work-sharing construct:
  - A PRIVATE copy of each list variable is created and initialized depending on the “op”
  - These copies are updated locally by threads
  - At end of construct, local copies are combined through “op” into a single value and combined with the value in the original SHARED variable
Reduction Example

- Local copy of \textit{sum} for each thread
- All local copies of \textit{sum} added together and stored in “global” variable

```
#pragma omp parallel for reduction(+:sum)
    for(i=0; i<N; i++) {
        sum += a[i] * b[i];
    }
```
OpenMP Reduction Example: Numerical Integration

\[ \int_{0}^{1} \frac{4.0}{(1+x^2)} \, dx = \pi \]

\begin{verbatim}
static long num_steps=100000;
double step, pi;

void main() {
    int i;
    double x, sum = 0.0;

    step = 1.0/(double) num_steps;
    for (i=0; i< num_steps; i++){
        x = (i+0.5)*step;
        sum = sum + 4.0/(1.0 + x*x);
    }
    pi = step * sum;
    printf("Pi = %f\n",pi);
}
\end{verbatim}
OpenMP Reduction Example: Numerical Integration

```c
#include <stdio.h>
#include <stdlib.h>
#include "omp.h"

int main(int argc, char* argv[]) {
    int num_steps = atoi(argv[1]);
    double step = 1.0/(double)(num_steps);
    double sum;

#pragma omp parallel for reduction(+:sum)
{
    for(int i=0; i<num_steps; i++) {
        double x = (i + .5)*step;
        sum += 4.0/(1.0 + x*x);
    }
}

double my_pi = sum*step;

return 0;
}
```

This didn’t work for me in VS2008, no support for reduction there…
A range of associative operands can be used with reduction

Initial values are the ones that make sense mathematically
OpenMP: Concluding Remarks & Wrap up
OpenMP Summary

- Shared memory, thread-based parallelism
- Explicit parallelism (parallel regions)
- Fork/join model

- Industry-standard shared memory programming model
  - First version released in 1997
- OpenMP Architecture Review Board (ARB) determines additions and updates to standard
  - The draft of OpenMP Version 3.1 has been released for public comments on 02/07/2011
  - The final specification of Version 3.1 is expected for June 2011

Include material from Rebecca Hartman-Baker’s presentation
The OpenMP API

- Application Programmer Interface (API) is combination of
  - Directives
    - Example: `#pragma omp task`
  - Runtime library routines
    - Example: `int omp_get_thread_num(void)`
  - Environment variables
    - Example: `setenv OMP_SCHEDULE "guided, 4"`
The OpenMP API

[Cntd.]

- API falls into three categories
  - Expression of parallelism (flow control)
    - Example: `#pragma omp parallel for`
  - Data sharing among threads (communication)
    - Example: `#pragma omp parallel for private(x,y)`
  - Synchronization (coordination or interaction)
    - Example: `#pragma omp barrier`

Include material from Rebecca Hartman-Baker’s presentation
OpenMP: Environment Variables

- **OMP_SCHEDULE**
  - Example: `setenv OMP_SCHEDULE "guided, 4"`

- **OMP_NUM_THREADS**
  - Sets the maximum number of threads to use during execution.
  - Example: `setenv OMP_NUM_THREADS 8`

- **OMP_DYNAMIC**
  - Enables or disables dynamic adjustment of the number of threads available for execution of parallel regions. Valid values are TRUE or FALSE
  - Example: `setenv OMP_DYNAMIC TRUE`

- **OMP_NESTED**
  - Enables or disables nested parallelism. Valid values are TRUE or FALSE
  - Example: `setenv OMP_NESTED TRUE`
OpenMP: Environment Variables
[New ones in 3.0 Release]

- **OMP_STACKSIZE**
  - Controls the size of the stack for created (non-Master) threads.

- **OMP_WAIT_POLICY**
  - Provides a hint to an OpenMP implementation about the desired behavior of waiting threads.

- **OMP_MAX_ACTIVE_LEVELS**
  - Controls the maximum number of nested active parallel regions. The value of this environment variable must be a non-negative integer. Example:
    - `setenv OMP_MAX_ACTIVE_LEVELS 2`

- **OMP_THREAD_LIMIT**
  - Sets the number of OpenMP threads to use for the whole OpenMP program
    - Example:
      - `setenv OMP_THREAD_LIMIT 8`
OpenMP 3.0: Summary of Run-Time Library OpenMP Routines

1. OMP_SET_NUM_THREADS  
2. OMP_GET_NUM_THREADS  
3. OMP_GET_MAX_THREADS  
4. OMP_GET_THREAD_NUM  
5. OMP_GET_THREAD_LIMIT  
6. OMP_GET_NUM_PROCS  
7. OMP_IN_PARALLEL  
8. OMP_SET_DYNAMIC  
9. OMP_GET_DYNAMIC  
10. OMP_SET_NESTED  
11. OMP_GET_NESTED  
12. OMP_SET_SCHEDULE  
13. OMP_GET_SCHEDULE  
14. OMP_SET_MAX_ACTIVE_LEVELS  
15. OMP_GET_MAX_ACTIVE_LEVELS  
16. OMP_GET_LEVEL  
17. OMP_GET_ANCESTOR_THREAD_NUM  
18. OMP_GET_TEAM_SIZE  
19. OMP_GET_ACTIVE_LEVEL  
20. OMP_INIT_LOCK  
21. OMP_DESTROY_LOCK  
22. OMP_SET_LOCK  
23. OMP_UNSET_LOCK  
24. OMP_TEST_LOCK  
25. OMP_INIT_NEST_LOCK  
26. OMP_DESTROY_NEST_LOCK  
27. OMP_SET_NEST_LOCK  
28. OMP_UNSET_NEST_LOCK  
29. OMP_TEST_NEST_LOCK  
30. OMP_GET_WTIME  
31. OMP_GET_WTICK
30+ Library Routines

- Runtime environment routines:
  - Modify/check the number of threads
    - `omp_[set|get]_num_threads()`
    - `omp_get_thread_num()`
    - `omp_get_max_threads()`
  - Are we in a parallel region?
    - `omp_in_parallel()`
  - How many processors in the system?
    - `omp_get_num_procs()`
  - Explicit locks
    - `omp_[set|unset]_lock()`
OpenMP API

- Get the thread number within a team
  ```c
  int omp_get_thread_num(void);
  ```
- Get the number of threads in a team
  ```c
  int omp_get_num_threads(void);
  ```
- Usually not needed for OpenMP codes
  - Can lead to code not being serially consistent
  - Does have specific uses (debugging)
  - Must include a header file
  ```c
  #include <omp.h>
  ```
OpenMP
The 30,000 Feet Perspective

OpenMP language extensions

- parallel control structures
- work sharing
- data environment
- synchronization
- runtime functions, env. variables

- governs flow of control in the program
- parallel directive
- do/parallel do and section directives
- distributes work among threads
- scopes variables
- shared and private clauses
- coordinates thread execution
- critical and atomic directives
- barrier directive
- runtime environment
- omp_set_num_threads()
- omp_get_thread_num()
- OMP_NUM_THREADS
- OMP_SCHEDULE
Attractive Features of OpenMP

- Parallelize small parts of application, one at a time (beginning with most time-critical parts)
- Can implement complex algorithms
- Code size grows only modestly
- Expression of parallelism flows clearly, code is easy to read
- Single source code for OpenMP and non-OpenMP
  - Non-OpenMP compilers simply ignore OMP directives
OpenMP, Some Caveats

- I’m not familiar with various OpenMP distributions, but it seems that there is a lag caused by the vendors to support the latest specifications
  - Intel probably is most up to speed although I haven’t used their compilers

- OpenMP threads are heavy
  - Good for handling parallel tasks
  - Not so good at handling fine large scale grain parallelism
Further Reading, OpenMP

- Michael Quinn (2003) Parallel Programming in C with MPI and OpenMP
- LLNL OpenMP Tutorial, https://computing.llnl.gov/tutorials/openMP/
- OpenMP.org, http://openmp.org/
- OpenMP 3.0 API Summary Cards:
  - C/C++:
“Part of the inhumanity of the computer is that, once it is competently programmed and working smoothly, it is completely honest.”

Isaac Asimov
Before We Get Started...

- **Last Time**
  - OpenMP wrap up
    - Variable scoping
    - Synchronization issues

- **Today**
  - Parallel programming patterns
  - One slide summary of ME964

- **Other issues:**
  - No class on April 12
  - Assignment 7 due tonight
  - Assignment 8 (last ME964 assignment) posted on the class website, due next Th
  - Midterm exam on April 19
    - Review session the evening before
  - From now on only guest lectures and such, time to concentrate on your projects

- Used for this presentation
- A good overview of challenges, best practices, and common techniques in all aspects of parallel programming
- Book is on reserve at Wendt Library
Objective

- Get exposed to techniques & best practices that have emerged as useful in the parallel programming practice
- They are expected to facilitate and/or help you with
  - Thinking/Visualizing your problem as being solved in parallel
  - Addressing functionality and performance issues in your parallel program design
  - Discussing your design decisions with others
  - Selecting an appropriate platform that helps you express & implement the parallel design for the solution you identified
Parallel Computing: When and Why.

- **Parallel computing, prerequisites**
  - The problem can be decomposed into sub-problems that can be independently solved at the same time.
  - The part of the problem that is concurrent is large enough to justify an approach that exploits this concurrency.
    - Recall Amdhal’s law.

- **Parallel computing, goals**
  - Solve problems in less time
  - and/or
  - Solve bigger problems.
Parallel Computing, Caveats

- Performance can be drastically reduced by many factors
  - Overhead of parallel processing
  - Load imbalance among processor elements
  - Inefficient data sharing patterns
  - Saturation of critical resources such as memory bandwidth
Implementing a Parallel Solution to Your Problem: Key Steps

1) Find the concurrency in the problem

2) Structure the algorithm so that concurrency can be exploited

3) Implement the algorithm in a suitable programming environment

4) Tune the performance of the code on the target parallel system

NOTE: The reality is that these have not been separated into levels of abstractions that can be dealt with independently.
What’s Next?

Focus on this for a while

Finding Concurrency

Algorithm Structure

Supporting Structures

Implementation Mechanisms

From “Patterns for Parallel Programming”
Finding and Nurturing Concurrency

- Dependence kills concurrency. Dependencies need to be identified and managed
  - Dependencies prevent parallelism, at least the “embarrassingly parallel” flavor
  - Often times, dependencies end up requiring synchronization barriers (not good…)

- Concurrency has some caveats
  - In sequential execution: One step feeds result to the next steps ⇒ a unique way moving from A to Z
  - In parallel execution: numeric accuracy may be affected by ordering steps that parallel with each other ⇒ platform dependent, OS dependent possibly even dependent on the state of the system

- Finding and exploiting concurrency often requires looking at the problem from a non-obvious angle
Finding Concurrency in Problems

- Goal: Identify a decomposition of the problem into sub-problems that can be solved simultaneously

- In order to meet this goal:
  - Perform a task decomposition to identify tasks that can execute concurrently
  - Carry out data decomposition to identify data local to each task
  - Identify a way of grouping tasks and ordering the groups to satisfy temporal constraints
  - Carry out an analysis of the data sharing patterns among the concurrent tasks to avoid any race condition issues and optimize memory access
  - Perform a design evaluation that assesses the quality of the choices made in all the steps
Finding Concurrency – The Process

This is typically an iterative process, like an optimization process that has to negotiate several constraints.
Find Concurrency 1: Decomp. Stage: Task Decomposition

- Many large problems have natural independent tasks
  - The number of tasks used should be adjustable to the execution resources available.
  - Each task must include sufficient work in order to compensate for the overhead of managing their parallel execution.
  - Tasks should maximize reuse of sequential program code to minimize design/implementation effort.

“In an ideal world, the compiler would find tasks for the programmer. Unfortunately, this almost never happens.”
- Mattson, Sanders, Massingill
Example: Task Decomposition
Square Matrix Multiplication

- \( P = M \times N \) of \( \text{WIDTH} \times \text{WIDTH} \)
  - One natural task (sub-problem) produces one element of \( P \)
  - All tasks can execute in parallel
Find Concurrency 2: Decomp. Stage: Data Decomposition

- The most compute intensive parts of many large problems manipulate a large data structure
  - Similar operations are being applied to different parts of the data structure, in a mostly independent manner.
  - This is what CUDA is optimized for.

- The data decomposition should lead to
  - Efficient data usage by each Unit of Execution (UE) within the partition
  - Few dependencies between the UEs that work on different partitions
  - Adjustable partitions that can be varied according to the hardware characteristics
Sometimes several tasks in problem can be grouped to improve efficiency

- Reduced synchronization overhead – because when task grouping there is supposedly no need for synchronization
- All tasks in the group efficiently share data loaded into a common on-chip, shared storage (Shared Memory)
Task Grouping Example - Square Matrix Multiplication

- Tasks calculating a P sub-block
  - Extensive input data sharing, reduced memory bandwidth using Shared Memory
  - All synched in execution
Find Concurrency 4: Dependence Analysis - Task Ordering

- Identify the data required by a group of tasks before they can execute
  - Find the task group that creates it (look upwind)
  - Determine a temporal order that satisfy all data constraints
Task Ordering Example: Finite Element Analysis

1. Node List
2. External Forces
3. Element Internal Force
4. Acceleration of each deformable beam
5. Update position and velocity of each beam
6. Next Time Step
Data sharing can be a double-edged sword

- An algorithm that calls for excessive data sharing can drastically reduce advantage of parallel execution

- Localized sharing can improve memory bandwidth efficiency

- Use the execution of task groups to interleave with (mask) global memory data accesses

- Read-only sharing can usually be done at much higher efficiency than read-write sharing, which often requires a higher level of synchronization
Data Sharing Example
Matrix Multiplication on the GPU

- Each task group will finish usage of each sub-block of N and M before moving on
  - N and M sub-blocks loaded into Shared Memory for use by all threads of a P sub-block
  - Amount of on-chip Shared Memory strictly limits the number of threads working on a P sub-block

- Read-only shared data can be efficiently accessed as Constant or Texture data (on the GPU)
  - Frees up the shared memory for other uses
Find Concurrency 6: Design Evaluation

- Key questions to ask
  - How many Units of Execution (UE) can be used?
  - How are the data structures shared?
  - Is there a lot of data dependency that leads to excessive synchronization needs?
  - Is there enough work in each UE between synchronizations to make parallel execution worthwhile?
Implementing a Parallel Solution to Your Problem: Key Steps

1) Find the concurrency in the problem

2) Structure the algorithm so that concurrency can be exploited

3) Implement the algorithm in a suitable programming environment

4) Execute and tune the performance of the code on a parallel system
What’s Next?

Focus on this for a while

Finding Concurrency

Algorithm Structure

Supporting Structures

Implementation Mechanisms

From “Patterns for Parallel Programming”
Algorithm: Definition

- A step by step procedure that is guaranteed to terminate, such that each step is precisely stated and can be carried out by a computer
  - Definiteness – the notion that each step is precisely stated
  - Effective computability – each step can be carried out by a computer
  - Finiteness – the procedure terminates
- Multiple algorithms can be used to solve the same problem
  - Some require fewer steps and some exhibit more parallelism
Algorithm Structure - Strategies

Organize by Tasks
- Task Parallelism
- Divide and Conquer

Organize by Data
- Geometric Decomposition
- Recursive Data Decomposition

Organize by Flow
- Pipeline
- Event Condition
Choosing Algorithm Structure

Algorithm Design

Organize by Task
- Linear
  - Task Parallelism
- Recursive
  - Divide and Conquer

Organize by Data
- Linear
  - Geometric Decomposition
- Recursive
  - Recursive Data Decmp.

Organize by Data Flow
- Regular
  - Pipeline
- Irregular
  - Event Driven
Alg. Struct. 1: Organize by Structure
Linear Parallel Tasks

- Common in the context of distributed memory models

- These are the algorithms where the work needed can be represented as a collection of decoupled or loosely coupled tasks that can be executed in parallel
  - The tasks don’t even have to be identical
  - Load balancing between UE can be an issue (dynamic load balancing?)
  - If there is no data dependency involved there is no need for synchronization: the so called “embarrassingly parallel” scenario
Examples

- Imagine a car that needs to be painted:
  - One robot paints the front left door, another one the rear left door, another one the hood, etc.
  - The car is parceled up with a collection of UEs taking care of subtasks

- Other: ray tracing, a good portion of the N-body problem, Monte Carlo type simulation
Valid when you can break a problem into a set of decoupled or loosely coupled smaller problems, which in turn can be broken etc…
  - This pattern is applicable when you can solve concurrently and with little synchronization the small problems (the leaves)

In some case you need synchronization when dealing with this balanced tree type algorithm
  - Often required by the merging step (assembling the result from the “subresults”)

Examples: FFT, Linear Algebra problems (see FLAME project), the vector reduce operation
Computational ordering can have major effects on memory bank conflicts and control flow divergence.
Alg. Struct. 3: Organize by Data
~ Linear (Geometric Decomposition) ~

- This is the case where the UEs gather and work around a big chunk of data with little or no synchronization.

- This is exactly the algorithmic approach enabled best by the GPU & CUDA.

- Examples: Matrix multiplication, matrix convolution, image processing.
This scenario comes into play when the data that you operate on is structured in a recursive fashion
- Balanced Trees
- Graphs
- Lists

Sometimes you don’t even have to have the data represented as such
- See example with prefix scan
- You choose to look at data as having a balanced tree structure

Problems that seem inherently sequential can be approached in this framework
- This is typically associated with an net increase in the amount of work you have to do
- Work goes up from $O(n)$ to $O(n \log(n))$ (see for instance Hillis and Steele algorithm)
- The key question is whether parallelism gained brings you ahead of the sequential alternative
Example: The Prefix Scan
~ Reduction Step~

for k=0 to M-1
    offset = 2^k
    for j=1 to 2^{M-k-1} in parallel do
        x[j \cdot 2^{k+1} - 1] = x[j \cdot 2^{k+1} - 1] + x[j \cdot 2^{k+1} - 2^k - 1]
    endfor
endfor
Example: The array reduction (the bad choice)

Array elements

<table>
<thead>
<tr>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0+1</td>
<td>2+3</td>
<td>4+5</td>
<td>6+7</td>
<td>8+9</td>
<td>10+11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0..3</td>
<td>4..7</td>
<td>8..11</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0..7</td>
<td>8..15</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Tasks are parallel, but there is a high degree of synchronization (coordination) between the UEs
- The input of one UE is the output of an upwind UE (the “pipeline”)
- The time elapsed between two pipeline ticks dictated by the slowest stage of the pipeline (the bottleneck)

Commonly employed by sequential chips

For complex problems having a deep pipeline is attractive
- At each pipeline tick you output one set of results

Examples:
- CPU: instruction fetch, decode, data fetch, instruction execution, data write are pipelined.
- The Cray vector architecture drawing heavily on this algorithmic structure when performing linear algebra operations
Alg. Struct. 6: Organize by Data Flow
~ Irregular: Event Driven Scenarios ~

- The furthest away from what the GPU can support today
- Well supported by MIMD architectures
- You coordinate UEs through asynchronous events
- Critical aspects:
  - Load balancing – should be dynamic
  - Communication overhead, particularly in real-time applications
- Suitable for action-reaction type simulations
- Examples: computer games, traffic control algorithms, server operation (amazon, google)
Implementing a Parallel Solution to Your Problem: Key Steps

1) Find the concurrency in the problem

2) Structure the algorithm so that concurrency can be exploited

3) **Implement the algorithm in a suitable programming environment**

4) Execute and tune the performance of the code on a parallel system
What’s Comes Next?

Finding Concurrency

Algorithm Structure

Supporting Structures

Implementation Mechanisms

Focus on this for a while
Above are the models for which parallel software/hardware combos provide good support nowadays. If you don’t fall in one of the above there’ll be no sailing, you’ll have to row.
Data Models

- **Shared Data**
  - All threads share a major data structure
  - This is what CUDA and GPU computing support the best

- **Shared Queue**
  - All threads see a “thread safe” queue
  - Very relevant in conjunction with the Master/Worker scenarios
  - OpenMP is very helpful here if your problem fits on one machine
    - If not, MPI can help

- **Distributed Array**
  - Decomposed and distributed among threads
  - Limited support in CUDA Shared Memory but the direction where libraries are going (thrust, for instance)
  - Good library support under MPI (this is how things get done in PETSc)
  - OpenMP: doesn’t apply
Program Models

- Master/Worker
  - A Master thread sets up a pool of worker threads and a bag of tasks
  - Workers execute concurrently, removing tasks until done
  - Common in OpenMP

- Loop Parallelism
  - Loop iterations execute in parallel
  - FORTRAN do-all (truly parallel), do-across (with dependence)
  - Very common in OpenMP

- Fork/Join
  - Most general way of creation of threads (the POSIX standard)
  - Can be regarded as a very low level approach in which you use the OS to manage parallelism
Program Models

- SPMD (Single Program, Multiple Data)
  - All PE’s (Processor Elements) execute the same program in parallel
  - Each PE has its own data
  - Each PE uses a unique ID to access its portion of data
  - Different PE can follow different paths through the same code

SPMD is by far the most commonly used pattern for structuring parallel programs.
More on SPMD

- Dominant coding style of scalable parallel computing
  - MPI code is mostly developed in SPMD style
  - Almost exclusively used as the pattern in GPU computing
  - Much OpenMP code is also in SPMD
  - Particularly suitable for algorithms based on data parallelism, geometric decomposition, divide and conquer.

- Main advantage
  - Tasks and their interactions visible in one piece of source code, no need to correlated multiple sources
Typical SPMD Program Phases

- **Initialize**
  - Establish localized data structure and communication channels

- **Obtain a unique identifier**
  - Each thread acquires a unique identifier, typically in the range from 0 to N-1, where N is the number of threads.
  - OpenMP, MPI, and CUDA have built-in support for this

- **Distribute Data**
  - Decompose global data into chunks and localize them, or
  - Sharing/replicating major data structure using thread ID to associate subset of the data to threads

- **Run the core computation**
  - More details in next slide…

- **Finalize**
  - Reconcile global data structure, prepare for the next major iteration
Core Computation Phase

- Thread IDs are used to differentiate behavior of threads
  - CUDA: Indx.x, Indx.y, Indx.z (also block ids)
  - MPI: rank of a process
  - OpenMP: get_thread_num() – gets id associated with a specific thread in a parallel region

- Use thread ID in loop index calculations to split loop iterations among threads

- Use conditions based on thread ID to branch to their specific actions
## Algorithm Structures [in columns] vs. Program Models [in rows]

<table>
<thead>
<tr>
<th>Algorithm Structures</th>
<th>Task Parallel</th>
<th>Divide/Conquer</th>
<th>Geometric Decomp.</th>
<th>Recursive Data</th>
<th>Pipeline</th>
<th>Event-based</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPMD</td>
<td>☺☺☺☺</td>
<td>☺☺</td>
<td>☺☺☺☺</td>
<td>☺</td>
<td>☺☺☺☺</td>
<td>☺</td>
</tr>
<tr>
<td>Loop Parallel</td>
<td>☺☺☺</td>
<td>☺</td>
<td>☺☺☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Master/Worker</td>
<td>☺☺☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
<td>☺</td>
</tr>
<tr>
<td>Fork/Join</td>
<td>☺☺</td>
<td>☺☺☺</td>
<td>☺☺☺</td>
<td>☺☺☺</td>
<td>☺☺☺☺</td>
<td>☺☺☺</td>
</tr>
</tbody>
</table>

Four smilies is the best (Source: Mattson, et al.)
Parallel Programming Support [in columns]  
vs.  
Program Models [in rows]

<table>
<thead>
<tr>
<th></th>
<th>OpenMP</th>
<th>MPI</th>
<th>CUDA</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SPMD</strong></td>
<td>☺☺☺</td>
<td>☺☺☺☺</td>
<td>☺☺☺☺</td>
</tr>
<tr>
<td><strong>Loop Parallel</strong></td>
<td>☺☺☺☺</td>
<td>☺</td>
<td></td>
</tr>
<tr>
<td><strong>Master/Slave</strong></td>
<td>☺☺</td>
<td>☺☺☺</td>
<td></td>
</tr>
<tr>
<td><strong>Fork/Join</strong></td>
<td>☺☺</td>
<td></td>
<td>☺☺☺</td>
</tr>
</tbody>
</table>
Sequential computing hit three walls: power, memory, and ILP walls

Moore’s law scales at least for one more decade
- Parallel computing poised to pick up where sequential computing left

Moore’s law brought us powerful CPUs and GPUs
- Use OpenMP and/or CUDA (or OpenCL) to leverage this hardware

Large problems do not always fit inside one workstation
- Not enough memory, or if enough memory not enough crunching number power

Large problem handled well by clusters or massively parallel computers
- MPI helps you here

When possible, use parallel libraries (thrust, PETSc, TBB, MKL, etc.)
- However, writing your own code for your own small problem sometimes pays off nicely
“The first 90 percent of the code accounts for the first 90 percent of the development time. The remaining 10 percent of the code accounts for the other 90 percent of the development time.”

—Tom Cargill
Before We Get Started…

- Last Time
  - Parallel programming patterns
  - One slide summary of ME964

- Today
  - Quickly go through your Final Project proposal presentations

- Other issues:
  - Assignment 8 (last ME964 assignment) due today at 11:59 PM
  - Midterm exam on April 19
    - Review session the evening before
    - Syllabus called for closed books, it’s ok to bring whatever source of information you want with you
  - From now on only guest lectures and such, time to concentrate on your projects
  - IMPORTANT: Final Project Proposal PDF doc due on Tu, at 11:59 PM
    - You submitted a proposal and/or provided a 3 slide presentation on the topic of your choice
    - If you don’t hear back from me it means that I’m ok with the topic you proposed and you should get going on your Final Project
    - I will use Doodle to allow you to enter the time/day when you choose to present your Final Project work
Discretized Poisson Sparse Matrix Solver with Conjugate Gradient Method

Spring 2011

Lakshman Anumolu
Department of Mechanical Engineering
University of Wisconsin, Madison
Motivation & Illustration

- Fluid flow simulations require the solution of Poisson equation.
- Solving Poisson equation is computationally expensive for high fidelity simulations.
- Discretized Poisson matrix contains at most 5 non-zero elements in each row.

\[
A = \begin{bmatrix}
4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
-1 & 4 & -1 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & -1 & 4 & 0 & 0 & -1 & 0 & 0 & 0 \\
-1 & 0 & 0 & 4 & -1 & 0 & -1 & 0 & 0 \\
0 & -1 & 0 & -1 & 4 & -1 & 0 & -1 & 0 \\
0 & 0 & -1 & 0 & -1 & 4 & 0 & 0 & -1 \\
0 & 0 & 0 & -1 & 0 & 0 & 4 & -1 & 0 \\
0 & 0 & 0 & 0 & -1 & 0 & -1 & 4 & -1 \\
0 & 0 & 0 & 0 & 0 & -1 & 0 & -1 & 4
\end{bmatrix}
\]

(Example: For 5x5 nodes with known boundary conditions)\(^1\)

Goal

- **Midterm project:**
  - Implementation of Conjugate Gradient method for solving above system on GPU.

- **Final project:**
  - Extending above implementation by modifying the method to use symmetric nature of discretized matrix.
  - Comparing the performance with sparse matrix solver “SpeedIT_classic²”.

Isotropic Projection Gridding for MRI Applications

Problem: Grid 3D radial data onto a Cartesian grid so that an FFT can be performed to obtain our final image.

Mike Loecher
Reasoning and prior work

- Gridding is the biggest time sink in the full image reconstruction
- Large data sets (~10,000,000 points per image x multiple timeframes)
- Potential to get a full reconstruction to a doctor while the patient is still in the scanner
- Some prior work by another author showed a speedup of around 100x, but in 2D and with a different trajectory
Outcomes

• Implement gridding for 3D isotropic projection data
• Obtain images with very little error compared to CPU gridding
• Optimize handling of variable sampling density (probably by separating into dense and sparse regions and handling differently)
Modeling the effects of applied stresses on granular materials with the discrete element method

Ian C. Olson

Current GPU DEM code:
- Normal Stiffness $K_n$
- Fixed boundaries
- Fixed particle size

New GPU DEM code:
- Add particle friction $F_f$
- Add capacity for variability in particle size
- Fixed, smooth side boundaries
- Smooth top / bottom boundaries apply stress to highest / lowest particles
- Output force distribution in system
- Automated variation in $\sigma$ when system reaches equilibrium
- Output changes in system height $h$ with corresponding applied stress
Final Project Problem Statement

- Develop GPU friendly code to compute a preconditioner used in the finite element analysis of thin structures.
GPU Friendly Preconditioners

Why?
- Many research projects, including my own, involve the solution of a linear system that is either impractical or impossible to solve directly. Using a preconditioner in iterative solvers can greatly reduce computation time and the number of iterations to solve the system but they are not easily implemented on a GPU.

Preliminary Results
- Dr. Suresh has already developed parts of the CUDA code and theory to implement a preconditioner on a GPU, as well as Abhirami and I have already implemented the calculation of the general metric over a triangular mesh and are validating and investigating the accuracy of our code.
GPU Friendly Preconditioners

- **Deliverables**
  - CUDA C code to complete the following tasks
    - Accurately calculate the general metric over the elements of a 3D triangular mesh
    - Calculate the general metric of a 3D structure for the volume inside of a bounding box
    - Calculate the stiffness matrix of a 3D structure using a general metric formulation
Final Project

- Brian J. Davis
- Medical Physics / BME
- Conebeam Backprojection Reconstruction for C-Arm CT and associated algorithms
- Need for real time cone-beam backprojections of projection images for four dimensional digital subtraction angiography (4D DSA)

Example of C-Arm CT - Siemens Artis Zeego shown
Rational

- Allow for 4D (3D time resolved volumes) to be viewed and rotated to any angle by the Radiologist.
- Allow selection of Region of Interest (ROI) to be selected showing perfusion, pulse, and Time of Arrival by the Radiologist.
Summary of Outcomes (Deliverables)

- Decrease backprojection reconstruction by the greatest degree possible. Current 512x512x397 reconstruction ~10 min on Tesla c1060. Matlab version was 2.5 days. Goal < 1 minute.
- Port other stages of recon to GPU currently in MATLAB with Mex interface. Log subtract, Parker Weights, Cosine Weighting, 4D-DSA, Time Interval Difference (TID), Color mapping, rendering (VTK), etc.
- Forward Projection (time allotting) – Reverse of back projection. Requires solving a sparse matrix of a linear system of equations.
Final Project Proposal

“Fast Normalized Cross Correlation”

ME 964

Kwang Won Choi
Abstract

- The correlation between two signals (cross correlation) is a typical approach to feature detection.
- The basic idea of this algorithm is finding the correlation coefficient between template feature (also called kernel) and input images.
Description of Algorithm

- The normalized cross-correlation between two signals of length N is defined as:

\[
 r_{xy} = \frac{\sum_{n=1}^{N} \{x(n) - \bar{x}\} \{y(n) - \bar{y}\}}{\sqrt{\sum_{n=1}^{N} \{x(n) - \bar{x}\}^2 \sum_{n=1}^{N} \{y(n) - \bar{y}\}^2}}
\]

- The result is that \( r_{xy} \) approaches 1 only when the region of image contained the template.
Usage in research field

- Elastography in the field of bio-research is a non-invasive technique in which stiffness or strain images of soft tissue are used to detect or distinguish tumours or tissues that have problems.
- We can detect and track a specific tissue out of whole tissue by using Cross Correlation.
Fig 1. Normalized Cross Correlation used in the area of elastography to detect a specific tissue out of ultrasound image of whole tissue.
Optical Lens Optimization

- Kuya Takami
- Mechanical Engineering

- Optical fiber fused lens coupling optimization using CUDA programing for ray tracing.
path length, $p = 80$ mm
$L_t = 5$ [mm]
$L_r = 16$ [mm]
By the end of the semester...

- Provide lens parameter optimizer program

- Based on user input (machining tolerance, parameter constraints) output optimal transmission lens design parameters.
Problem Statement
  • Implement a direct parallel solver to solve very large sparse linear systems (millions of unknowns)

Reasons
  • Solving sparse linear systems is needed in optimization, mechanics, fluidics, etc. The parallel direct solver remains an open problem, and there is so far no open source GPU implementation.

Preliminary results
  • Investigated the SPIKE algorithm.
The SPIKE Algorithm

Goal: Solving $AX=b$, where $A$ is sparse

- Use some recording technique, transform $A$ to a banded matrix
- Partition $A$ into $p$ blocks (in this example, $p=3$)

- Factorize $A$, $A=DS$, where $D=\text{diag}(A_1, \ldots, A_p)$
Problem becomes solving $DSX=b$, can be divided into two steps:
(a) $DG=b$ can be solved directly
(b) $SX=G$ can be further reduced and solved either directly or iteratively depending on the problem type.

- **Expected Outcomes**
  - Implement SPIKE algorithm on GPU or other parallel scheme
  - Optimize the code
Sparse Matrix Reordering

Spring 2011

Andrew Seidl

Department of Mechanical Engineering
University of Wisconsin, Madison
Sparse Matrix Reordering

- **Goal**: rearrange elements so that matrix becomes a banded matrix
  - Allows a partitioning of a large matrix as required by the SPIKE algorithm (see previous presentation)
  - This in turn enables the use of multiple MPI-connected nodes in solving the large sparse linear system

- **ParMETIS**
  - MPI-based library implementing multilevel nested dissection algorithm
  - Balanced elimination trees: allows for parallel direct factorization
Sparse Matrix Reordering

$nz = 9240$
ME964 Final Project: GPU Implementation of Absolute Nodal Coordinate Formulation (ANCF)

Dan Melanz – Mechanical Engineering Simulation-Based Engineering Lab
What is ANCF?

- Used to carry out the dynamics analysis of flexible bodies that undergo large rotation and large deformation

- Consistent with nonlinear theory of continuum mechanics, yet easy to implement

- Can be combined with the Discrete Element Method to handle friction and contact between beams
Applications of ANCF

- Hair simulations
- Polymer simulations
- Grass/Terrain modeling
- Material modeling
Parallelizing Exhaustive Search of Feasible Aperiodic Task Start

● Rehan Ahmed
Realtime Tasks

- Tasks in which the correctness of a computed output is dependent not only on the logical correctness but also time at which the output is delivered
- Realtime tasks have a notion of deadline
- Depending on the type of system, missing a deadline can have catastrophic consequences.

Types Of Tasks
- **Periodic Tasks**: Repeat After every Time Period.
- **Aperiodic Tasks**: can come at arbitrary time instant
Problem Definition

- In a given thermally constrained system, a static schedule for periodic tasks can be constructed such that their deadlines are met and thermal constraints for the system are met.
- Aperiodic tasks can be scheduled in idle times as long as the processor temperature does not go beyond a certain threshold level.
- Goal of the scheduler is to exhaustively search for the earliest start time for the aperiodic task.
- For this project, I intend to parallelize the exhaustive search phase of this algorithm.
  - Each thread can evaluate temperature impact at a different start time.
  - Earliest feasible start time is accepted
Deliverables

- Code for scheduling periodic/aperiodic tasks along with instructions (readme.txt)
  - Both serial and parallel versions
- Design Document:
  - Explanation of temperature estimation scheme.
  - Algorithm for making admission control decision of aperiodic tasks.
  - Performance analysis (serial vs. parallel)
Beam and plate structures are used extensively in structural engineering. Analysis of such structures involves the construction of a beam (or plate) stiffness matrix.
The parallelization achievable in GPUs enables the use of simpler algorithms at various stages involved in the analysis, and also provides significant speedup in the computations.

- Weighted volume computation has been implemented on the GPU and verified using various test structures and cases.
- Also part of the Midterm Project is to complete the calculation of the weighted volume of a model within a given bounding box.
SUMMARY OF DELIVERABLES

- For the final project, this framework is to be extended in order to calculate the beam stiffness matrix of any arbitrary structure, given a triangulated representation of it and 1-D beam elements.

- The accuracy and precision of the computation for various parameter choices, and also the performance of the algorithm in comparison to the time taken on a CPU are to be evaluated.
Jacobi solver on a GPU to solve Poisson equation

Spring 2011

Sarangarajan.V.Iyengar

Department of Mechanical Engineering
University of Wisconsin, Madison
Motivation

- Solving CFD problems are computationally expensive
- One of the important steps which takes the major share in computational time is solving the Poisson equation
- Since there is no analytic solution for the Navier-Stokes equation we opt for Iterative schemes.
- Some common techniques used are
  - Jacobi method
  - Gauss-Seidel method
  - Successive over relaxation method
  - Conjugate gradient method
Goal

- **Midterm project:**
  - Implementation of GPU based Jacobi solver.

- **Final project:**
  - Solving 2D Incompressible Laminar N.S. equations using the GPU based Jacobi solver.
ME964 Final Project: Monte Carlo Ray Tracing using CUDA

Zigfried Hampel-Arias
Final Project Proposal
14 April, 2011
Cerenkov Water Tank

Introduction:
Cosmic Rays produce
\( N = O(10^{10}) \) particles in air
\( N \sim \text{energy} \rightarrow \text{estimated from Cerenkov light in water tank} \)

Motivation for Work:
Need faster tank sim (always...)
Monte Carlo

Detector Function:
Relativistic particles from air showers emit Cerenkov light in water tank
GPU Implementation

Simulation:
Photons are created independently and non-interacting
  -> Embarrassingly Parallel!!!
  -> Threads don’t share information
  -> 1 photon/thread

Method:
- Ray trace photons in tank....wait for it...
  - Reflections off tank walls NOT specular
  - Water absorption/scattering non-deterministic
- Detection by Photomultiplier Tube stochastic
  ....Distributions, distributions, distributions....
GPU Implementation

Proposed Solution:
- Photon’s life has three states (device functions):
  - Reflection
  - Absorption (dead)
  - Detection

- Best to group photons in similar life-states
- Readout detector response after all photon’s die
- DO PHYSICS!!!
Modeling of a Magneto-Rheological Suspension in Parallel – Ben Wilson

Background

- Magneto-Rheological (MR) Fluid is a fluid with magnetic particles suspended in it
  - When a magnetic field is applied, properties of the fluid change (i.e. viscosity)
- Common application today: Magneride shocks

Modeling aspects

- Can treat each magnetic particle as a sphere
- Each particle is acted on by a magnetic force, hydrodynamic force, and a wall force
- This becomes an N-Body simulation
Current Techniques used to reduce computation time

- Taking advantage of the fact that $\mathbf{F}_{ij} = -\mathbf{F}_{ji}$
  - Enables interaction between each particle to be calculated only once
- Use of neighbor list
  - Determines which particles are “neighbors” to a particular particle
    - Indicates which particles might interact with a particular particle later in the simulation
  - Only iterate over the “neighbor” particles when calculating the force
    - Opposed to iterating over every particle to see which are within range
  - If calculated too frequently code slows down
    - Simply just iterating over every particle anyway
  - If calculated too infrequently, may not include particles that are interacting with a particular particle later in the simulation
Goals of this project

- Use knowledge of CUDA when calculating the forces between particles.
  - Starting with a cluster of particles, observe how they interact when including only magnetic forces
- Determine the speedup of the code when compared with sequential code.
- Time permitting, include periodic boundary condition, forces associated with the system walls, and hydrodynamic drag.
ME964
Final Project: CFD
Andrew Kokemoor
2D Navier-Stokes Solver

- Conservation Equations
  - Mass
  - X-momentum
  - Y-momentum

- Numerics
  - $2^{nd}$ order Central Difference derivatives
  - Adams-Bashforth time integration
Physical Setup

- Rectangular domain, grid
- Uniform inlet flow, advective outflow
- Shear-free walls parallel to flow
- Gaussian vortex initial condition
Parallel Numerics

- Embarrassingly Parallel
  - Initialization
  - Time integration
- Pressure-velocity coupling
  - Solve matrix using Poisson solver
  - Iterative methods can be parallelized
    - Not necessarily deterministic
PETSc: Platform for Scientific Computing

Matthew Knepley

Computation Institute
University of Chicago

ME 964: High Performance Computing for Engineering Applications
University of Wisconsin – Madison
April 21, 2011
1 Introduction

- Who uses and develops PETSc?
- Stuff for Windows
- How can I get PETSc?
- How do I Configure PETSc?
- How do I Build PETSc?
- How do I run an example?
- How do I get more help?

2 Version Control

3 Vector Algebra

4 Matrix Algebra
What I Need From You

- Tell me if you do not understand
- Tell me if an example does not work
- Suggest better wording or figures
- Followup problems at petsc-maint@mcs.anl.gov
Ask Questions!!!

- Helps me understand what you are missing
- Helps you clarify misunderstandings
- Helps others with the same question
How We Can Help at the Tutorial

- Point out relevant documentation
- Quickly answer questions
- Help install
- Guide design of large scale codes
- Answer email at petsc-maint@mcs.anl.gov
Point out relevant documentation
Quickly answer questions
Help install
Guide design of large scale codes
Answer email at petsc-maint@mcs.anl.gov
How We Can Help at the Tutorial

- Point out relevant documentation
- Quickly answer questions
- Help install
- Guide design of large scale codes
- Answer email at petsc-maint@mcs.anl.gov
How We Can Help at the Tutorial

- Point out relevant documentation
- Quickly answer questions
- Help install
- Guide design of large scale codes
- Answer email at petsc-maint@mcs.anl.gov
PETSc was developed as a Platform for Experimentation

We want to experiment with different
- Models
- Discretizations
- Solvers
- Algorithms
  - which blur these boundaries
Developing parallel, nontrivial PDE solvers that deliver high performance is still difficult and requires months (or even years) of concentrated effort.

**PETSc is a toolkit that can ease these difficulties and reduce the development time, but it is not a black-box PDE solver, nor a silver bullet.**

— Barry Smith
What is PETSc?

A freely available and supported research code

- Free for everyone, including industrial users
- Hyperlinked manual, examples, and manual pages for all routines
- Hundreds of tutorial-style examples
- Support via email: petsc-maint@mcs.anl.gov
- Usable from C, C++, Fortran 77/90, and Python
What is PETSc?

- Portable to any parallel system supporting MPI, including:
  - Tightly coupled systems
    - Cray XT5, BG/Q, NVIDIA Fermi, Earth Simulator
  - Loosely coupled systems, such as networks of workstations
    - IBM, Mac, iPad/iPhone, PCs running Linux or Windows

PETSc History

- Begun September 1991
- Over 60,000 downloads since 1995 (version 2)
- Currently 400 per month

PETSc Funding and Support

- Department of Energy
  - SciDAC, MICS Program, AMR Program, INL Reactor Program
- National Science Foundation
  - CIG, CISE, Multidisciplinary Challenge Program
**What Can We Handle?**

- PETSc has run implicit problems with over 500 billion unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media

- PETSc has run on over 224,000 cores efficiently
  - UNIC on the IBM BG/P Intrepid at ANL
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL

- PETSc applications have run at 22 Teraflops
  - Kaushik on XT5
  - LANL PFLOTRAN code
What Can We Handle?

- PETSc has run implicit problems with over 500 billion unknowns
  - UNIC on BG/P and XT5
  - PFLOTRAN for flow in porous media

- PETSc has run on over 224,000 cores efficiently
  - UNIC on the IBM BG/P Intrepid at ANL
  - PFLOTRAN on the Cray XT5 Jaguar at ORNL

- PETSc applications have run at 22 Teraflops
  - Kaushik on XT5
  - LANL PFLOTRAN code
PETSc has run implicit problems with over 500 billion unknowns
- UNIC on BG/P and XT5
- PFLOTRAN for flow in porous media

PETSc has run on over 224,000 cores efficiently
- UNIC on the IBM BG/P Intrepid at ANL
- PFLOTRAN on the Cray XT5 Jaguar at ORNL

PETSc applications have run at 22 Teraflops
- Kaushik on XT5
- LANL PFLOTRAN code
New Model for Scientific Software

- sympy
- symbolics
- FFC/SyFi
- eqn. definition
- data structures
- integration/assembly
- solvers
- petsc4py
- numpy
- PyCUDA
- PETSc
- CUDA OpenCL

Figure: Schematic for a generic scientific application
1 Introduction

- Who uses and develops PETSc?
  - Stuff for Windows
  - How can I get PETSc?
  - How do I Configure PETSc?
  - How do I Build PETSc?
  - How do I run an example?
  - How do I get more help?
Who Uses PETSc?

Computational Scientists

- **Earth Science**
  - PyLith (CIG)
  - Underworld (Monash)
  - Magma Dynamics (LDEO, Columbia)

- **Subsurface Flow and Porous Media**
  - STOMP (DOE)
  - PFLOTRAN (DOE)
Who Uses PETSc?

Computational Scientists

- CFD
  - OpenFOAM
  - freeCFD
  - OpenFVM

- MicroMagnetics (MagPar)

- Fusion (NIMROD)
Who Uses PETSc?

Algorithm Developers

- **Iterative methods**
  - Deflated GMRES
  - LGMRES
  - QCG
  - SpecEst

- **Preconditioning researchers**
  - Prometheus (Adams)
  - ParPre (Eijkhout)
  - FETI-DP (Klawonn and Rheinbach)
Who Uses PETSc?

Algorithm Developers

- Finite Elements
  - PETSc-FEM
  - libMesh
  - Deal II
  - OOFEM
- Fast Multipole Method (PetFMM)
- Radial Basis Function Interpolation (PetRBF)
- Eigensolvers (SLEPc)
- Optimization (TAO)
The PETSc Team

Bill Gropp
Barry Smith
Satish Balay

Jed Brown
Matt Knepley
Lisandro Dalcin

Hong Zhang
Victor Eijkhout
Dmitry Karpeev
Introduction

Who uses and develops PETSc?

Stuff for Windows

How can I get PETSc?

How do I Configure PETSc?

How do I Build PETSc?

How do I run an example?

How do I get more help?
Questions for Windows Users

- Have you installed cygwin?
  - Need python, make, and build-utils packages

- Will you use the GNU compilers?
  - If not, remove `link.exe`
  - If MS, check compilers from `cmd window` and use `win32fe`

- Which MPI will you use?
  - You can use `-with-mpi=0`
  - If MS, need to install MPICH2
  - If GNU, can use `-download-mpich`
Outline

1 Introduction
  - Who uses and develops PETSc?
  - Stuff for Windows
  - How can I get PETSc?
  - How do I Configure PETSc?
  - How do I Build PETSc?
  - How do I run an example?
  - How do I get more help?
The latest tarball is on the PETSc site
  - We no longer distribute patches (everything is in the distribution)

There is a Debian package

There is a FreeBSD Port

There is a Mercurial development repository
Cloning PETSc

- The full development repository is open to the public
  - http://petsc.cs.iit.edu/petsc/petsc-dev
  - http://petsc.cs.iit.edu/petsc/BuildSystem

- Why is this better?
  - You can clone to any release (or any specific ChangeSet)
  - You can easily rollback changes (or releases)
  - You can get fixes from us the same day

- We also make release repositories available
  - http://petsc.cs.iit.edu/petsc/releases/petsc-3.1
  - http://petsc.cs.iit.edu/petsc/releases/BuildSystem-3.1
Unpacking PETSc

- **Just clone development repository**
  - `hg clone http://petsc.cs.iit.edu/petsc/petsc-dev petsc-dev`
  - `hg clone -rrelease-3.1 petsc-dev petsc-3.1`

- **Unpack the tarball**
  - `tar xzf petsc.tar.gz`
Exercise 1

Download and Unpack PETSc!
Outline

1 Introduction
- Who uses and develops PETSc?
- Stuff for Windows
- How can I get PETSc?
- How do I Configure PETSc?
- How do I Build PETSc?
- How do I run an example?
- How do I get more help?
Configuring PETSc

- Set `$PETSC_DIR` to the installation root directory
- Run the configuration utility
  - `$PETSC_DIR/configure`
  - `$PETSC_DIR/configure -help`
  - `$PETSC_DIR/configure -download-mpich`
  - `$PETSC_DIR/configure -prefix=/usr`
- There are many examples on the installation page
- Configuration files are in `$PETSC_DIR/$PETSC_ARCH/conf`
  - Configure header is in `$PETSC_DIR/$PETSC_ARCH/include`
  - `$PETSC_ARCH` has a default if not specified
You can easily reconfigure with the same options
- ./$PETSC_ARCH/conf/reconfigure-$PETSC_ARCH.py

Can maintain several different configurations
- ./configure -PETSC_ARCH=linux-fast
  -with-debugging=0

All configuration information is in the logfile
- ./$PETSC_ARCH/conf/configure.log
- ALWAYS send this file with bug reports
Automatic Downloads

- Starting in 2.2.1, some packages are automatically
  - Downloaded
  - Configured and Built (in $PETSC_DIR/externalpackages)
  - Installed with PETSc
- Currently works for
  - petsc4py
  - PETSc documentation utilities (Sowing, lgrind, c2html)
  - BLAS, LAPACK, BLACS, ScaLAPACK, PLAPACK
  - MPICH, MPE, OpenMPI
  - ParMetis, Chaco, Jostle, Party, Scotch, Zoltan
  - MUMPS, Spooles, SuperLU, SuperLU_Dist, UMFPack, pARMS
  - BLOPEX, FFTW, SPRNG
  - Prometheus, HYPRE, ML, SPAI
  - Sundials
  - Triangle, TetGen
  - FIAT, FFC, Generator
  - Boost
Configure your downloaded PETSc.
Outline

1 Introduction

- Who uses and develops PETSc?
- Stuff for Windows
- How can I get PETSc?
- How do I Configure PETSc?
- How do I Build PETSc?
- How do I run an example?
- How do I get more help?
Building PETSc

- Uses recursive make starting in `cd $PETSC_DIR`
  - `make`
  - `make install` if you configured with `--prefix`
  - Check build when done with `make test`

- Or `./config/builder.py` which handles dependencies

- Complete log for each build is in logfile
  - `./$PETSC_ARCH/conf/make.log`
  - ALWAYS send this with bug reports

- Can build multiple configurations
  - `PETSC_ARCH=linux-fast make`
  - Libraries are in `$PETSC_DIR/$PETSC_ARCH/lib`

- Can also build a subtree
  - `cd src/snes; make`
  - `cd src/snes; make ACTION=libfast tree`
Build your configured PETSc.
Exercise 4

Reconfigure PETSc to use ParMetis.

1. `linux-c-debug/conf/reconfigure-linux-c-debug.py`
   - `PETSC_ARCH=linux-parmetis`
   - `download-parmetis`

2. `PETSC_ARCH=linux-parmetis` make

3. `PETSC_ARCH=linux-parmetis` make test
Outline

1 Introduction

- Who uses and develops PETSc?
- Stuff for Windows
- How can I get PETSc?
- How do I Configure PETSc?
- How do I Build PETSc?
- How do I run an example?
- How do I get more help?
Try running PETSc examples first
  cd $PETSC_DIR/src/snes/examples/tutorials

Build examples using make targets
  make ex5

Run examples using the make target
  make runex5

Can also run using MPI directly
  mpirun ./ex5 -snes_max_it 5
  mpiexec ./ex5 -snes_monitor
Using MPI

- The **Message Passing Interface** is:
  - a library for parallel communication
  - a system for launching parallel jobs (mpirun/mpiexec)
  - a community standard

- Launching jobs is easy
  - `mpiexec -n 4 ./ex5`

- You should never have to make MPI calls when using PETSc
  - Almost never
Common Viewing Options

- Gives a text representation
  - `-vec_view`

- Generally views subobjects too
  - `-snes_view`

- Can visualize some objects
  - `-mat_view_draw`

- Alternative formats
  - `-vec_view_binary, -vec_view_matlab, -vec_view_socket`

- Sometimes provides extra information
  - `-mat_view_info, -mat_view_info_detailed`
Common Monitoring Options

- Display the residual
  - `-ksp_monitor`, **graphically** `-ksp_monitor_draw`
- Can disable dynamically
  - `-ksp_monitors_cancel`
- Does not display subsolvers
  - `-snes_monitor`
- Can use the true residual
  - `-ksp_monitor_true_residual`
- Can display different subobjects
  - `-snes_monitor_residual`, `-snes_monitor_solution`, `-snes_monitor_solution_update`
  - `-snes_monitor_range`
  - `-ksp_gmres_krylov_monitor`
- Can display the spectrum
  - `-ksp_monitor_singular_value`
Run SNES Example 5 using come custom options.

1. cd $PETSC_DIR/src/snes/examples/tutorials
2. make ex5
3. mpiexec ./ex5 -snes_monitor -snes_view
4. mpiexec ./ex5 -snes_type tr -snes_monitor -snes_view
5. mpiexec ./ex5 -ksp_monitor -snes_monitor -snes_view
6. mpiexec ./ex5 -pc_type jacobi -ksp_monitor -snes_monitor -snes_view
7. mpiexec ./ex5 -ksp_type bicg -ksp_monitor -snes_monitor -snes_view
Exercise 6

Create a new code based upon SNES Example 5.

1. Create a new directory
   
   ```
   mkdir -p /home/knepley/proj/newsim/src
   ```

2. Copy the source
   
   ```
   cp ex5.c /home/knepley/proj/newsim/src
   Add myStuff.c and myStuff2.F
   ```

3. Create a PETSc makefile
   
   ```
   bin/ex5: src/ex5.o src/myStuff.o src/myStuff2.o
   ${CLINKER} -o $@ ^ ${PETSC_SNES_LIB}
   include ${PETSC_DIR}/conf/variables
   include ${PETSC_DIR}/conf/rules
   ```

To get the project ready-made

```
hg clone http://petsc.cs.iit.edu/petsc/tutorials/SimpleTutorial newsim
```
Introduction

- Who uses and develops PETSc?
- Stuff for Windows
- How can I get PETSc?
- How do I Configure PETSc?
- How do I Build PETSc?
- How do I run an example?
- How do I get more help?
Getting More Help

- http://www.mcs.anl.gov/petsc
- Hyperlinked documentation
  - Manual
  - Manual pages for every method
  - HTML of all example code (linked to manual pages)
- FAQ
- Full support at petsc-maint@mcs.anl.gov
- High profile users
  - David Keyes
  - Marc Spiegelman
  - Richard Katz
  - Brad Aagaard
  - Aron Ahmadia
Outline

1. Introduction
2. Version Control
3. Vector Algebra
4. Matrix Algebra
5. Algebraic Solvers
6. SNES
7. DA
8. PCFieldSplit
9. PETSc-GPU
Location and Retrieval
“Where’s the Tarball”

Version Control
- Mercurial, Git, Subversion

Hosting
- BitBucket, GitHub, Launchpad

Community involvement
- arXiv, PubMed
CVS/SVN manage a single repository
  - Versioned data
  - Local copy for modification and checkin

Mercurial manages many repositories
  - Identified by URLs
  - No one Master

Repositories communicate by ChangeSets
  - Use push and pull to move changesets
  - Can move arbitrary changes with patch queues
Figure: Single Repository
Project Workflow

Figure: Master Repository with User Clones
Figure: Project with Release and Bugfix Repositories
What are PETSc vectors?

- Fundamental objects representing
  - solutions
  - right-hand sides
  - coefficients

- Each process locally owns a subvector of contiguous global data
How do I create vectors?

- VecCreate(
  MPI_Comm, Vec *)
- VecSetSizes(Vec, PetscInt n, PetscInt N)
- VecSetType(Vec, VecType typeName)
- VecSetFromOptions(Vec)

  Can set the type at runtime
A PETSc Vec

- Supports all vector space operations
  - VecDot(), VecNorm(), VecScale()
- Has a direct interface to the values
  - VecGetArray(), VecGetArrayF90()
- Has unusual operations
  - VecSqrtAbs(), VecStrideGather()
- Communicates automatically during assembly
- Has customizable communication (VecScatter)
Processes may set an arbitrary entry
  - Must use proper interface
Entries need not be generated locally
  - Local meaning the process on which they are stored
PETSc automatically moves data if necessary
  - Happens during the assembly phase
Vector Assembly

- A three step process
  - Each process sets or adds values
  - Begin communication to send values to the correct process
  - Complete the communication

VecSetValues($\text{Vec } v$, $\text{PetscInt } n$, $\text{PetscInt } \text{rows}[]$, $\text{PetscScalar } \text{values}[]$, $\text{InsertMode } \text{mode}$)

- Mode is either $\text{INSERT\_VALUES}$ or $\text{ADD\_VALUES}$
- Two phases allow overlap of communication and computation
  - VecAssemblyBegin($\text{Vec } v$)
  - VecAssemblyEnd($\text{Vec } v$)
One Way to Set the Elements of a Vector

VecGetSize(x, &N);
MPI_Comm_rank(PETSC_COMM_WORLD, &rank);
if (rank == 0) {
    val = 0.0;
    for (i = 0; i < N; ++i) {
        VecSetValues(x, 1, &i, &val, INSERT_VALUES);
        val += 10.0;
    }
}

/* These routines ensure that the data is distributed to the other processes */
VecAssemblyBegin(x);
VecAssemblyEnd(x);
VecGetOwnershipRange(x, &low, &high);
val = low*10.0;
for (i = low; i < high; ++i) {
    VecSetValues(x, 1, &i, &val, INSERT_VALUES);
    val += 10.0;
}
/* No data will be communicated here */
VecAssemblyBegin(x);
VecAssemblyEnd(x);
## Selected Vector Operations

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>VecAXPY(Vec y, PetscScalar a, Vec x)</td>
<td>( y = y + a \times x )</td>
</tr>
<tr>
<td>VecAYPX(Vec y, PetscScalar a, Vec x)</td>
<td>( y = x + a \times y )</td>
</tr>
<tr>
<td>VecWAYPX(Vec w, PetscScalar a, Vec x, Vec y)</td>
<td>( w = y + a \times x )</td>
</tr>
<tr>
<td>VecScale(Vec x, PetscScalar a)</td>
<td>( x = a \times x )</td>
</tr>
<tr>
<td>VecCopy(Vec y, Vec x)</td>
<td>( y = x )</td>
</tr>
<tr>
<td>VecPointwiseMult(Vec w, Vec x, Vec y)</td>
<td>( w_i = x_i \times y_i )</td>
</tr>
<tr>
<td>VecMax(Vec x, PetscInt *idx, PetscScalar *r)</td>
<td>( r = \max r_i )</td>
</tr>
<tr>
<td>VecShift(Vec x, PetscScalar r)</td>
<td>( x_i = x_i + r )</td>
</tr>
<tr>
<td>VecAbs(Vec x)</td>
<td>( x_i =</td>
</tr>
<tr>
<td>VecNorm(Vec x, NormType type, PetscReal *r)</td>
<td>( r =</td>
</tr>
</tbody>
</table>
It is sometimes more efficient to directly access local storage of a Vec.

- PETSc allows you to access the local storage with
  - `VecGetArray(Vec, double *[])`

- You must return the array to PETSc when you finish
  - `VecRestoreArray(Vec, double *[])`

- Allows PETSc to handle data structure conversions
  - Commonly, these routines are fast and do not involve a copy
Vec v;
PetscScalar *array;
PetscInt n, i;
PetscErrorCode ierr;

VecGetArray(v, &array);
VecGetLocalSize(v, &n);
PetscSynchronizedPrintf(PETSC_COMM_WORLD,
    "First element of local array is %f\n", array[0]);
PetscSynchronizedFlush(PETSC_COMM_WORLD);
for(i = 0; i < n; ++i) {
    array[i] += (PetscScalar) rank;
}
VecRestoreArray(v, &array);
```c
#include "finclude/petsc.h"

Vec v;
PetscScalar array(1)
PetscOffset offset
PetscInt n, i
PetscErrorCode ierr

call VecGetArray(v, array, offset, ierr)
call VecGetLocalSize(v, n, ierr)
do i=1,n
    array(i+offset) = array(i+offset) + rank
end do
call VecRestoreArray(v, array, offset, ierr)
```
#include "finclude/petsc.h90"

Vec v;
PetscScalar pointer :: array(:)
PetscInt n, i
PetscErrorCode ierr

call VecGetArrayF90(v, array, ierr)
call VecGetLocalSize(v, n, ierr)
do i=1,n
   array(i) = array(i) + rank
end do
call VecRestoreArrayF90(v, array, ierr)
with v as a:
    for i in range(len(a)):
        a[i] = 5.0*i
## Outline

1. Introduction
2. Version Control
3. Vector Algebra
4. **Matrix Algebra**
5. Algebraic Solvers
6. SNES
7. DA
8. PCFieldSplit
What are PETSc matrices?

- Fundamental objects for storing stiffness matrices and Jacobians
- Each process locally owns a contiguous set of rows
- Supports many data types
  - AIJ, Block AIJ, Symmetric AIJ, Block Matrix, etc.
- Supports structures for many packages
  - MUMPS, Spooles, SuperLU, UMFPack, DSCPack
How do I create matrices?

- `MatCreate(MPI_Comm, Mat *)`
- `MatSetSizes(Mat, PetscInt m, PetscInt n, M, N)`
- `MatSetType(Mat, MatType typeName)`
- `MatSetFromOptions(Mat)`

  - Can set the type at runtime

- `MatSeqAIJPreallocation(Mat, PetscInt nz, const PetscInt nnz[])`
- `MatMPIAIJPreallocation(Mat, dnz, dnz[], onz, onz[])`
- `MatSetValues(Mat, m, rows[], n, cols[], values[], InsertMode)`

  - **MUST** be used, but does automatic communication
Matrix Polymorphism

The PETSc Mat has a single user interface,

- Matrix assembly
  - `MatSetValues()`
- Matrix-vector multiplication
  - `MatMult()`
- Matrix viewing
  - `MatView()`

but multiple underlying implementations.

- AIJ, Block AIJ, Symmetric Block AIJ,
- Dense
- Matrix-Free
- etc.

A matrix is defined by its **interface**, not by its **data structure**.
Matrix Algebra

Matrix Assembly

- A three step process
  - Each process sets or adds values
  - Begin communication to send values to the correct process
  - Complete the communication
- MatSetValues(Mat m, m, rows[], n, cols[], values[], mode)
  - mode is either INSERT_VALUES or ADD_VALUES
  - Logically dense block of values
- Two phase assembly allows overlap of communication and computation
  - MatAssemblyBegin(Mat m, MatAssemblyType type)
  - MatAssemblyEnd(Mat m, MatAssemblyType type)
  - type is either MAT_FLUSH_ASSEMBLY or MAT_FINAL_ASSEMBLY
Matrix Algebra

One Way to Set the Elements of a Matrix
Simple 3-point stencil for 1D Laplacian

```c
v[0] = -1.0; v[1] = 2.0; v[2] = -1.0;
if (rank == 0) {
    for (row = 0; row < N; row++) {
        cols[0] = row-1; cols[1] = row; cols[2] = row+1;
        if (row == 0) {
            MatSetValues(A, 1, &row, 2, &cols[1], &v[1], INSERT_VALUES);
        } else if (row == N-1) {
            MatSetValues(A, 1, &row, 2, cols, v, INSERT_VALUES);
        } else {
            MatSetValues(A, 1, &row, 3, cols, v, INSERT_VALUES);
        }
    }
}
MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
```

M. Knepley ()
PETSc
UW '11 67 / 118
Matrix Storage Layout

- Each process locally owns a submatrix of contiguous global rows
- Each submatrix consists of diagonal and off-diagonal parts

```
MatGetOwnershipRange(Mat A, int *start, int *end)
```

- `start`: first locally owned row of global matrix
- `end-1`: last locally owned row of global matrix
v[0] = -1.0; v[1] = 2.0; v[2] = -1.0;
MatGetOwnershipRange(A,&start,&end);
for (row = start; row < end; row++) {
    cols[0] = row-1; cols[1] = row; cols[2] = row+1;
    if (row == 0) {
        MatSetValues(A,1,&row,2,&cols[1],&v[1],INSERT_VALUES);
    } else if (row == N-1) {
        MatSetValues(A,1,&row,2,cols,v,INSERT_VALUES);
    } else {
        MatSetValues(A,1,&row,3,cols,v,INSERT_VALUES);
    }
}
MatAssemblyBegin(A, MAT_FINAL_ASSEMBLY);
MatAssemblyEnd(A, MAT_FINAL_ASSEMBLY);
No one data structure is appropriate for all problems
- Blocked and diagonal formats provide performance benefits
- PETSc has many formats
- Makes it easy to add new data structures

Assembly is difficult enough without worrying about partitioning
- PETSc provides parallel assembly routines
- High performance still requires making most operations local
- However, programs can be incrementally developed.
- MatPartitioning and MatOrdering can help

Matrix decomposition in contiguous chunks is simple
- Makes interoperation with other codes easier
- For other ordering, PETSc provides “Application Orderings” (AO)
Outline

1. Introduction
2. Version Control
3. Vector Algebra
4. Matrix Algebra
5. Algebraic Solvers
6. SNES
7. DA
8. PCFieldSplit
9. PETSc-GPU
Solver Types

- **Explicit:**
  - Field variables are updated using local neighbor information

- **Semi-implicit:**
  - Some subsets of variables are updated with global solves
  - Others with direct local updates

- **Implicit:**
  - Most or all variables are updated in a single global solve
Using PETSc linear algebra, just add:
- `KSPSetOperators(KSP ksp, Mat A, Mat M, MatStructure flag)`
- `KSPSolve(KSP ksp, Vec b, Vec x)`

Can access subobjects
- `KSPGetPC(KSP ksp, PC *pc)`

Preconditioners must obey PETSc interface
- Basically just the KSP interface

Can change solver dynamically from the command line
- `-ksp_type bicgstab`
Using PETSc linear algebra, just add:

- \text{SNESSetFunction} (\text{SNES} \text{ snes}, \text{Vec} \text{ r}, \text{residualFunc}, \text{void} *\text{ctx})
- \text{SNESSetJacobian} (\text{SNES} \text{ snes}, \text{Mat} \text{ A}, \text{Mat} \text{ M}, \text{jacFunc}, \text{void} *\text{ctx})
- \text{SNESSolve} (\text{SNES} \text{ snes}, \text{Vec} \text{ b}, \text{Vec} \text{ x})

Can access subobjects

- \text{SNESGetKSP} (\text{SNES} \text{ snes}, \text{KSP} *\text{ksp})

Can customize subobjects from the cmd line

- Set the subdomain preconditioner to ILU with \text{-sub_pc_type ilu}
Use `SNESSetFromOptions()` so that everything is set dynamically

- Set the type
  - Use `-snes_type` (or take the default)
- Override the tolerances
  - Use `-snes_rtol` and `-snes_atol`
- View the solver to make sure you have the one you expect
  - Use `-snes_view`
- For debugging, monitor the residual decrease
  - Use `-snes_monitor`
  - Use `-ksp_monitor` to see the underlying linear solver
Complete table of solvers

1. Sequential LU
   - ILUDT (SPAREREKIT2, Yousef Saad, U of MN)
   - EUCLID & PILUT (Hypre, David Hysom, LLNL)
   - ESSL (IBM)
   - SuperLU (Jim Demmel and Sherry Li, LBNL)
   - Matlab
   - UMFPACK (Tim Davis, U. of Florida)
   - LUSOL (MINOS, Michael Saunders, Stanford)

2. Parallel LU
   - MUMPS (Patrick Amestoy, IRIT)
   - SPOOLES (Cleve Ashcroft, Boeing)
   - SuperLU_Dist (Jim Demmel and Sherry Li, LBNL)

3. Parallel Cholesky
   - DSCPACK (Padma Raghavan, Penn. State)
   - MUMPS (Patrick Amestoy, Toulouse)
   - CHOLMOD (Tim Davis, Florida)

4. XYTlib - parallel direct solver (Paul Fischer and Henry Tufo, ANL)
3rd Party Preconditioners in PETSc

Complete table of solvers

1. Parallel ICC
   - BlockSolve95 (Mark Jones and Paul Plassman, ANL)

2. Parallel ILU
   - PaStiX (Faverge Mathieu, INRIA)

3. Parallel Sparse Approximate Inverse
   - Parasails (Hypre, Edmund Chow, LLNL)
   - SPAI 3.0 (Marcus Grote and Barnard, NYU)

4. Sequential Algebraic Multigrid
   - RAMG (John Ruge and Klaus Steuben, GMD)
   - SAMG (Klaus Steuben, GMD)

5. Parallel Algebraic Multigrid
   - Prometheus (Mark Adams, PPPL)
   - BoomerAMG (Hypre, LLNL)
   - ML (Trilinos, Ray Tuminaro and Jonathan Hu, SNL)
Outline

1. Introduction
2. Version Control
3. Vector Algebra
4. Matrix Algebra
5. Algebraic Solvers
6. SNES
7. DA
8. PCFieldSplit
9. PETSc-GPU

M. Knepley ()
Flow Control for a PETSc Application

Main Routine

- Timestepping Solvers (TS)
- Nonlinear Solvers (SNES)
- Linear Solvers (KSP)
- Preconditioners (PC)

Application Initialization
Function Evaluation
Jacobian Evaluation
Postprocessing

PETSc
The SNES interface is based upon callback functions

- `FormFunction()`, set by `SNESSetFunction()`
- `FormJacobian()`, set by `SNESSetJacobian()`

When PETSc needs to evaluate the nonlinear residual $F(x)$,

- Solver calls the **user's** function
- User function gets application state through the `ctx` variable
  - PETSc **never** sees application data
Topology Abstractions

- **DA**
  - Abstracts Cartesian grids in any dimension
  - Supports stencils, communication, reordering
  - Nice for simple finite differences

- **Mesh**
  - Abstracts general topology in any dimension
  - Also supports partitioning, distribution, and global orders
  - Allows arbitrary element shapes and discretizations
Assembly Abstractions

- **DM**
  - Abstracts the logic of multilevel (multiphysics) methods
  - Manages allocation and assembly of local and global structures
  - Interfaces to DMMG solver

- **Section**
  - Abstracts functions over a topology
  - Manages allocation and assembly of local and global structures
  - Will merge with DM somehow
SNES Function

User provided function calculates the nonlinear residual:

```c
PetscErrorCode (*func)(SNES snes, Vec x, Vec r, void *ctx)
```

- **x**: The current solution
- **r**: The residual
- **ctx**: The user context passed to `SNESSetFunction()`
  - Use this to pass application information, e.g. physical constants
User provided function calculates the Jacobian:

\[(\*func)(\text{SNES } \text{snes}, \text{Vec } \text{x}, \text{Mat } \star \text{J}, \text{Mat } \star \text{M}, \text{MatStructure } \star \text{flag}, \text{void } \star \text{ctx})\]

**x**: The current solution

**J**: The Jacobian

**M**: The Jacobian preconditioning matrix (possibly J itself)

**ctx**: The user context passed to **SNESSetJacobian**()

- Use this to pass application information, e.g. physical constants

  - Possible **MatStructure** values are:
    - **SAME_NONZERO_PATTERN**
    - **DIFFERENT_NONZERO_PATTERN**

Alternatively, you can use

- matrix-free finite difference approximation, **-snes_mf**
- finite difference approximation with coloring, **-snes_fd**
SNES Variants

- Line search strategies
- Trust region approaches
- Picard iteration
- Variational inequality approaches
PETSc can compute and explicitly store a Jacobian via 1st-order FD

- **Dense**
  - Activated by `-snes_fd`
  - Computed by `SNESDefaultComputeJacobian()`

- **Sparse via colorings**
  - Coloring is created by `MatFDColoringCreate()`
  - Computed by `SNESDefaultComputeJacobianColor()`

Can also use Matrix-free Newton-Krylov via 1st-order FD

- Activated by `-snes_mf` without preconditioning
- Activated by `-snes_mf_operator` with user-defined preconditioning
  - Uses preconditioning matrix from `SNESSetJacobian()`
SNES Example
Driven Cavity

Solution Components

- Velocity-vorticity formulation
- Flow driven by lid and/or buoyancy
- Logically regular grid
  - Parallelized with DA
- Finite difference discretization
- Authored by David Keyes

$PETCS_DIR/src/snes/examples/tutorials/ex19.c
typedef struct {
    /*----- basic application data -----*/
    PetscReal lid_velocity;
    PetscReal prandtl;
    PetscReal grashof;
    PetscBool draw_contours;
} AppCtx;

$PETCS_DIR/src/snes/examples/tutorials/ex19.c
Residual(SNES snes, Vec X, Vec F, void *ptr) {
    AppCtx *user = (AppCtx *) ptr;

    /* local starting and ending grid points */
    PetscInt istart, iend, jstart, jend;
    PetscScalar *f; /* local vector data */
    PetscReal grashof = user->grashof;
    PetscReal prandtl = user->prandtl;
    PetscErrorCode ierr;

    /* Code to communicate nonlocal ghost point data */
    VecGetArray(F, &f);
    /* Code to compute local function components */
    VecRestoreArray(F, &f);
    return 0;
}
ResLocal(DALocalInfo *info, PetscScalar **x, PetscScalar **f, void *ctx)
{
    for (j = info->ys; j < info->ys+info->ym; ++j) {
        for (i = info->xs; i < info->xs+info->xm; ++i) {
            u = x[j][i];
            if (i==0 || j==0 || i == M || j == N) {
                f[j][i] = u; continue;
            }
            u_xx = (2.0 *u - x[j][i-1] - x[j][i+1])*hydhx;
            u_yy = (2.0 *u - x[j-1][i] - x[j+1][i])*hxdhy;
            f[j][i] = u_xx + u_yy - hx*hy*lambda*exp(u);
        }
    }
}
What is a DA?

**DA** is a topology interface on structured grids

- Handles parallel data layout
- Handles local and global indices
  - `DAGetGlobalIndices()` and `DAGetAO()`
- Provides local and global vectors
  - `DAGetGlobalVector()` and `DAGetLocalVector()`
- Handles ghost values coherence
  - `DAGetGlobalToLocal()` and `DAGetLocalToGlobal()`
The DA interface is based upon \textit{local} callback functions

- \texttt{FormFunctionLocal()}, \textbf{set by} \texttt{DASetLocalFunction()}
- \texttt{FormJacobianLocal()}, \textbf{set by} \texttt{DASetLocalJacobian()}

When PETSc needs to evaluate the nonlinear residual $F(x)$,

- Each process evaluates the local residual
- PETSc assembles the global residual automatically
  - \textbf{Uses} \texttt{DALocalToGlobal()} method
To evaluate a local function $f(x)$, each process requires
- its local portion of the vector $x$
- its **ghost values**, bordering portions of $x$ owned by neighboring processes
## DA Global Numberings

### Natural numbering

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>25 26 27</td>
<td>28 29</td>
<td>10 11 12</td>
<td>13 14</td>
</tr>
<tr>
<td>20 21 22</td>
<td>23 24</td>
<td>5 6 7</td>
<td>8 9</td>
</tr>
<tr>
<td>15 16 17</td>
<td>18 19</td>
<td>0 1 2</td>
<td>3 4</td>
</tr>
</tbody>
</table>

### PETSc numbering

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>21 22 23</td>
<td>28 29</td>
<td>6 7 8</td>
<td>13 14</td>
</tr>
<tr>
<td>18 19 20</td>
<td>26 27</td>
<td>3 4 5</td>
<td>11 12</td>
</tr>
<tr>
<td>15 16 17</td>
<td>24 25</td>
<td>0 1 2</td>
<td>9 10</td>
</tr>
</tbody>
</table>
**Global**: Each vertex has a unique id belongs on a unique process

**Local**: Numbering includes vertices from neighboring processes
- These are called *ghost* vertices

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>12</td>
<td>13</td>
</tr>
<tr>
<td>14</td>
<td>15</td>
</tr>
<tr>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>10</td>
<td>11</td>
</tr>
<tr>
<td>11</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

Local numbering

<table>
<thead>
<tr>
<th>Proc 2</th>
<th>Proc 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>22</td>
</tr>
<tr>
<td>23</td>
<td>28</td>
</tr>
<tr>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>20</td>
<td>26</td>
</tr>
<tr>
<td>15</td>
<td>16</td>
</tr>
<tr>
<td>17</td>
<td>24</td>
</tr>
<tr>
<td>24</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>8</td>
<td>13</td>
</tr>
<tr>
<td>11</td>
<td>12</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>9</td>
</tr>
</tbody>
</table>

Global numbering

<table>
<thead>
<tr>
<th>Proc 0</th>
<th>Proc 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Proc 0</td>
<td>Proc 1</td>
</tr>
</tbody>
</table>

M. Knepley ( )

PETSc

UW ’11 95 / 118
DA Local Function

The user provided function which calculates the nonlinear residual in 2D has signature

\[(*lfunc)(DALocalInfo *info, PetscScalar **x, PetscScalar **r, void *ctx)\]

- **info**: All layout and numbering information
- **x**: The current solution
  - Notice that it is a multidimensional array
- **r**: The residual
- **ctx**: The user context passed to `DASetLocalFunction()`

The local DA function is activated by calling

\[SNESSetFunction(snes, r, SNESDADAFormFunction, ctx)\]
\[ \Delta u + \lambda e^u = 0 \]

Bratu Residual Evaluation

```c
ResLocal(DALocalInfo *info,
         PetscScalar **x, PetscScalar **f, void *ctx) {
    for(j = info->ys; j < info->ys+info->ym; ++j) {
        for(i = info->xs; i < info->xs+info->xm; ++i) {
            u = x[j][i];
            if (i==0 || j==0 || i == M || j == N) {
                f[j][i] = u; continue;
            }
            u_xx = (2.0 *u - x[j][i-1] - x[j][i+1])*hydhx;
            u_yy = (2.0 *u - x[j-1][i] - x[j+1][i])*hxdhy;
            f[j][i] = u_xx + u_yy - hx*hy*lambda*exp(u);
        }
    }
}
```

PETSc_DIR/src/snes/examples/tutorials/ex5.c
The user provided function which calculates the Jacobian in 2D has signature

\[(*1\text{func})(\text{DALocalInfo } *\text{info}, \text{PetscScalar } **x, \text{Mat } J, \text{void } *\text{ctx})\]

**info**: All layout and numbering information

**x**: The current solution

**J**: The Jacobian

**ctx**: The user context passed to \text{DASetLocalJacobian}()

The local DA function is activated by calling

\text{SNESSetJacobian}(\text{snes, } J, J, \text{SNESDADecomputeJacobian}, \text{ctx})
# Bratu Jacobian Evaluation

```c
JacLocal(DALocalInfo *info, PetscScalar **x, Mat jac, void *ctx) {
    for (j = info->ys; j < info->ys + info->ym; j++) {
        for (i = info->xs; i < info->xs + info->xm; i++) {
            row.j = j; row.i = i;
            if (i == 0 || j == 0 || i == mx-1 || j == my-1) {
                v[0] = 1.0;
                MatSetValuesStencil(jac, 1, &row, 1, &row, v, INSERT_VALUES);
            } else {
                v[0] = -(hx/hy); col[0].j = j-1; col[0].i = i;
                v[1] = -(hy/hx); col[1].j = j; col[1].i = i-1;
                v[2] = 2.0*(hy/hx+hx/hy) - hx*hy*lambda*PetscExpScalar(x[j][i]);
                v[3] = -(hy/hx); col[3].j = j; col[3].i = i+1;
                v[4] = -(hx/hy); col[4].j = j+1; col[4].i = i;
                MatSetValuesStencil(jac, 1, &row, 5, col, v, INSERT_VALUES);
            }
        }
    }
}
```

$PETCS_DIR/src/snes/examples/tutorials/ex5.c
A DA contains **topology**, **geometry**, and an implicit Q1 **discretization**.

It is used as a template to create
- Vectors (functions)
- Matrices (linear operators)
The DA object contains only layout (topology) information.
- All field data is contained in PETSc Vecs.

Global vectors are parallel.
- Each process stores a unique local portion.
- `DACreateGlobalVector(DA da, Vec *gvec)`

Local vectors are sequential (and usually temporary).
- Each process stores its local portion plus ghost values.
- `DACreateLocalVector(DA da, Vec *lvec)`
- includes ghost values!
Updating Ghosts

Two-step process enables overlapping computation and communication

- **DAGlobalToLocalBegin**(da, gvec, mode, lvec)
  - gvec provides the data
  - mode is either INSERT_VALUES or ADD_VALUES
  - lvec holds the local and ghost values

- **DAGlobalToLocalEnd**(da, gvec, mode, lvec)
  - Finishes the communication

The process can be reversed with **DALocalToGlobal()**.
Both the box stencil and star stencil are available.
PETSc provides

\[
\text{MatSetValuesStencil}(\text{Mat } A, m, \text{ MatStencil } \text{idxm}[], n, \text{ MatStencil } \text{idxn}[], \text{PetscScalar} \text{ values}[], \text{InsertMode} \text{ mode})
\]

- Each row or column is actually a MatStencil
  - This specifies grid coordinates and a component if necessary
  - Can imagine for unstructured grids, they are vertices
- The values are the same logically dense block in row/col
Creating a DA

```
DACreate2d(comm, wrap, type, M, N, m, n, dof, s, lm[], ln[], DA *da)
```

**wrap:** Specifies periodicity
- DA_NONPERIODIC, DA_XPERIODIC, DA_YPERIODIC, or DA_XYPERIODIC

**type:** Specifies stencil
- DA_STENCIL_BOX or DA_STENCIL_STAR

**M/N:** Number of grid points in x/y-direction

**m/n:** Number of processes in x/y-direction

**dof:** Degrees of freedom per node

**s:** The stencil width

**lm/n:** Alternative array of local sizes
- Use PETSC_NULL for the default
The **PCFieldSplit** interface

- extracts functions/operators corresponding to each physics
  - \texttt{VecScatter} and \texttt{MatGetSubMatrix()} for efficiency
- assemble functions/operators over all physics
  - Generalizes \texttt{LocalToGlobal()} mapping
- is composable with **ANY** PETSc solver and preconditioner
  - This can be done recursively
The **PCFieldSplit** interface

- extracts functions/operators corresponding to each physics
  - *VecScatter* and *MatGetSubMatrix()* for efficiency
- assemble functions/operators over all physics
  - Generalizes *LocalToGlobal()* mapping
- is composable with **ANY** PETSc solver and preconditioner
  - This can be done recursively

**FieldSplit** provides the **buildings blocks** for multiphysics preconditioning.
MultiPhysics Paradigm

The **PCFieldSplit** interface

- extracts functions/operators corresponding to each physics
  - *VecScatter* and *MatGetSubMatrix()* for efficiency

- assemble functions/operators over all physics
  - Generalizes *LocalToGlobal()* mapping

- is composable with **ANY** PETSc solver and preconditioner
  - This can be done recursively

Notice that this works in exactly the same manner as

- multiple resolutions (MG, FMM, Wavelets)
- multiple domains (Domain Decomposition)
- multiple dimensions (ADI)
Several varieties of preconditioners can be supported:

- Block Jacobi or Block Gauss-Siedel
- Schur complement
- Block ILU (approximate coupling and Schur complement)
- Dave May’s implementation of Elman-Wathen type PCs which only require actions of individual operator blocks

Notice also that we may have any combination of

- “canned” PCs (ILU, AMG)
- PCs needing special information (MG, FMM)
- custom PCs (physics-based preconditioning, Born approximation)

since we have access to an algebraic interface
Outline

1. Introduction
2. Version Control
3. Vector Algebra
4. Matrix Algebra
5. Algebraic Solvers
6. SNES
7. DA
8. PCFieldSplit
Thrust is a CUDA library of parallel algorithms

- Interface similar to C++ Standard Template Library
- Containers (vector) on both host and device
- Algorithms: sort, reduce, scan
- Freely available, part of PETSc configure (-with-thrust-dir)
Cusp is a CUDA library for sparse linear algebra and graph computations

- Builds on data structures in Thrust
- Provides sparse matrices in several formats (CSR, Hybrid)
- Includes some preliminary preconditioners (Jacobi, SA-AMG)
- Freely available, part of PETSc configure (`-with-cusp-dir`)
Strategy: Define a new Vec implementation

- Uses Thrust for data storage and operations on GPU
- Supports full PETSc Vec interface
- Inherits PETSc scalar type
- Can be activated at runtime, \(-\text{vec\_type\_cuda}\)
- PETSc provides memory coherence mechanism
PETSc Objects now hold a coherence flag

<table>
<thead>
<tr>
<th>PETSC_CUDA_UNALLOCATED</th>
<th>No allocation on the GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>PETSC_CUDA_GPU</td>
<td>Values on GPU are current</td>
</tr>
<tr>
<td>PETSC_CUDA_CPU</td>
<td>Values on CPU are current</td>
</tr>
<tr>
<td>PETSC_CUDA_BOTH</td>
<td>Values on both are current</td>
</tr>
</tbody>
</table>

**Table:** Flags used to indicate the memory state of a PETSc CUDA *Vec* object.
Also define new \texttt{Mat} implementations

- Uses \texttt{Cusp} for data storage and operations on GPU
- Supports full PETSc \texttt{Mat} interface, some ops on CPU
- Can be activated at runtime, \texttt{-mat_type aijcuda}
- Notice that parallel matvec necessitates off-GPU data transfer
Solvers come for Free

- All linear algebra types work with solvers
- Entire solve can take place on the GPU
  - Only communicate scalars back to CPU
- GPU communication cost could be amortized over several solves
- Preconditioners are a problem
  - Cusp has a promising AMG
Installation

PETSc only needs

```sh
# Turn on CUDA
--with-cuda
# Specify the CUDA compiler
--with-cudac='nvcc -m64'
# Indicate the location of packages
# --download-* will also work soon
--with-thrust-dir=/PETSc3/multicore/thrust
--with-cusp-dir=/PETSc3/multicore/cusp
# Can also use double precision
--with-precision=single
```
Example
Driven Cavity Velocity-Vorticity with Multigrid

ex19 -da_vec_type seqcuda
  -da_mat_type aijcuda -mat_no_inode
  -da_grid_x 100 -da_grid_y 100
  -pc_type none -dmmg_nlevels 1
  -preload off -cuda_synchronize
  -log_summary # Setup types
  # Set grid size
  # Setup solver
  # Setup run
Flow Solver
32 × 32 × 32 grid

<table>
<thead>
<tr>
<th>Routine</th>
<th>Time (s)</th>
<th>MFlops</th>
<th>MFlops/s</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>CPU</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KSPSolve</td>
<td>8.3167</td>
<td>4370</td>
<td>526</td>
</tr>
<tr>
<td>MatMult</td>
<td>1.5031</td>
<td>769</td>
<td>512</td>
</tr>
<tr>
<td><strong>GPU</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>KSPSolve</td>
<td>1.6382</td>
<td>4500</td>
<td>2745</td>
</tr>
<tr>
<td>MatMult</td>
<td>0.3554</td>
<td>830</td>
<td>2337</td>
</tr>
</tbody>
</table>

P. Lichtner, G. Hammond, R. Mills, B. Phillip
ME964
CMake & CUDA
Debugging

Spring 2011

Brian J. Davis
M.S. / PhD. Candidate Biomedical Engineering
Research Assistant
School of Medicine and Public Health
Research Area Medical Physics
bdavis5@wisc.edu
CMake

- Build tools – What do they do?
- What is CMake?
- Why use CMake?
- What can CMake do?
- Example Project
- Don’t listen to me go to the source… hey man I am just spreading the word.
  - Google Tech Talk
    - CMake/CPack/CTest/CDash Open Source Tools to Build Test and Deploy C++ Software
      - http://www.youtube.com/watch?v=8Ut9o4OdSC0&feature=youtube_gdata
  - I am not affiliated in any way with Kitware. Just a poor schlep trying to get his code to compile.
The Source

- CMake by Kitware
  - [http://www.CMake.org/](http://www.CMake.org/)
  - Documentation
    - [http://www.cmake.org/cmake/help/cmake-2-8-docs.html](http://www.cmake.org/cmake/help/cmake-2-8-docs.html)
  - FAQ
    - [http://www.CMake.org/Wiki/CMake_FAQ](http://www.CMake.org/Wiki/CMake_FAQ)
  - Download
    - [http://www.CMake.org/CMake/resources/software.html](http://www.CMake.org/CMake/resources/software.html)
  - Source Repository
    - [http://www.CMake.org/Wiki/CMake/Git](http://www.CMake.org/Wiki/CMake/Git)
  - Tutorial
    - [http://www.CMake.org/CMake/help/CMake_tutorial.html](http://www.CMake.org/CMake/help/CMake_tutorial.html)

"Go right to the source and ask the horse. He'll give you the answer that you'll endorse. He's always on a steady course. Talk to Mr. Ed" – Theme song to Mr. Ed, 1961-1966
Build tools what do they do?

- Check dependencies
  - You changed a source file and the .obj file needs to be rebuilt which then rebuilds the dll or exe.
  - Controls what gets built in what order

- Execute generators
  - Such as the compiler which converts source to output files
  - Generate configuration files
  - Specify install and testing locations and put all files in correct locations.
  - Generate code
    - Lexx Yacc or Flex and Bison
    - Extended Backus–Naur Form (EBNF) – context-free grammars
    - SWIG C/C++/C# (Mono) Language integration in Linux

- Check system for build dependencies (some do VS not so much)
  - CUDA, nvcc,
  - Library dependencies (VTK, ITK, etc)
  - What are capabilities/versions of C and C++ stdlib etc.

- Whatever you command them to do
  
  ```
  MESSAGE( "Executing world domination script now" )
  execute_process(
    COMMAND world_domination.bat
    WORKING_DIRECTORY ./world_domination_scripts/milky_way_galaxy/earth
    INPUT_FILE world_domination.cfg
    OUTPUT_FILE world_domination_result.log
    ERROR_FILE world_domination_error.log
  )
  ```
Build Tools

- What have I used? What comparisons can I make?
  - BorlandBuilder 5.02/Builder C++ 4, and 6
  - MS VisualStudio
  - GNU Make, autoconf, and friends
  - BJam – BoostBuildV2 part of Boost C++
    - Perforce Software (not related to Boost) Jam Tutorial – only put here due to use of Jam
      - I have not used the Perforce version only Boost’s version
  - CMake
Visual Studio as a build tool

>>> MY OPINIONS <<<

- Based on What? – ~10 years experience
- Great for simple projects
- Unwieldy for complex projects
- Modal Dialog boxes that can’t be resized?
- Build Spec is not searchable
- Which spec am I changing? all, debug, release? Is this for 32 or x64?
- What did I change that made the build break? I know I can diff the .sln and .prj files. We will take a look at those (next slide)
- All seems hidden behind GUI which is difficult access, change, and maintain. Especially with broad sweeping changes. Ok there is project inheritance.
- Is this the best Microsoft can do? Well no they do have nmake. Of course laying out a path to (project) destruction is no help either.
- Wizards are no Merlin.
- Good for wear leveling of your mouse buttons through increased use of right click

Registry Key use saddens me. The very concept (the registry) is an attack on my sensibilities. I must now cleanse my thoughts with a couple of pages of Linux kernel code and thoughts of apple pie :-) - Brian J. Davis – CMake Forums : CUDA, CMake, and an attempt to build nbody, 2010
Looking at (diffing) VS project files WHEN things go wrong: .sln

Microsoft Visual Studio Solution File, Format Version 10.00
# Visual Studio 2008
Project("[8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942]") = "CudaCollision", "CudaCollision\CudaCollision.vcproj", "[08C6F311-7AA6-46EB-BFB9-7F947F5DD014]
EndProject
Project("[8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942]") = "BulletValidation", "BulletValidation\BulletValidation.vcproj", "[8644F016-E5EF-432D-98C6-91D27D459746]
EndProject
Project("[8BC9CEB8-8B4A-11D0-8D11-00A0C91BC942]") = "DataGeneration", "DataGeneration\DataGeneration.vcproj", "[A0B424D7-6CCD-465C-A90E-D5589E73954E]
EndProject

Global
  GlobalSection(SolutionConfigurationPlatforms) = preSolution
    Debug|Win32 = Debug|Win32
    Debug|x64 = Debug|x64
    Release|Win32 = Release|Win32
    Release|x64 = Release|x64
  EndGlobalSection
  GlobalSection(ProjectConfigurationPlatforms) = postSolution
    [08C6F311-7AA6-46EB-BFB9-7F947F5DD014].Debug|Win32.ActiveCfg = Debug|Win32
    [08C6F311-7AA6-46EB-BFB9-7F947F5DD014].Debug|x64.ActiveCfg = Debug|x64
    [08C6F311-7AA6-46EB-BFB9-7F947F5DD014].Debug.x64.Build.0 = Debug|x64
    [08C6F311-7AA6-46EB-BFB9-7F947F5DD014].Release|Win32.ActiveCfg = Release|Win32
    [08C6F311-7AA6-46EB-BFB9-7F947F5DD014].Release|x64.ActiveCfg = Release|x64
    [08C6F311-7AA6-46EB-BFB9-7F947F5DD014].Release.x64.Build.0 = Release|x64
  EndGlobalSection
  GlobalSection(SolutionProperties) = preSolution
    HideSolutionNode = FALSE
  EndGlobalSection
EndGlobal
Looking at (diffing) VS project files WHEN things go wrong: .sln

```xml
<VisualStudioProject
    ProjectType="Visual C++"
    Version="9.00"
    Name="CudaCollision"
    ProjectGUID="(08C6F311-7AA6-46EB-BFB9-7F947F5DD014)"
    RootNamespace="CudaCollision"
    Keyword="x64Proj"
    TargetFrameworkVersion="0"
>
    <Platforms>
        <Platform Name="x64"/>
        <Platform Name="Win32"/>
    </Platforms>
    <ToolFiles>
        <DefaultToolFile FileName="NvCudaRuntimeApi.v3.2.rules"/>
        ...
        ... Blah Blah Blah GOBS and GOBS more where this came from
        ...
        <File RelativePath="\.cuda_timer.cu"/>
    </ToolFiles>
</VisualStudioProject>
```
What does VS do?

- Ok well .prj is better than .sln not that I don’t like looking at hash codes or anything.
- I counted 698 lines (ok well SciTE http://www.scintilla.org/SciTE.html line numbering did) of good wholesome xml
- Seriously 698 lines of xml to specify 1 executable (CollisionDetection) to generate a handful of command lines????
- Who’s selling harddrives these days… I need to invest!
- The point here is that this is the text output which can be viewed and diffed with diff tools and should be preferably clean and it should be a relatively short time to figure out what changed and broke the build.
What is CMake?

- From http://www.CMake.org/
  - “Welcome to CMake, the cross-platform, open-source build system. CMake is a family of tools designed to build, test and package software. CMake is used to control the software compilation process using simple platform and compiler independent configuration files. CMake generates native makefiles and workspaces that can be used in the compiler environment of your choice.”
  - Product of Kitware due to the need for a cross-platform build environment for Insight Segmentation and Registration Toolkit (ITK) as part of the Visible Human Project
Meta build

- Meta build tool
  - A built tool that builds build files defined by build specifications (CMakeLists.txt files)
  - CMakeLists.txt files which generate Make (Linux and friends and Nmake on Windows) or project files (VS):
  - First experience was with cygwin where CMake bootstraped itself, built itself which was then used to generate the build files for ITK which were then used to build ITK using GNU Make. This resulted in a 4 stage to compile…. Awesome!
    - ./bootstrap
    - make
    - make install
    - Run CMake to generate Makefiles
    - Build VTK using gnu make and makefiles
  - I was flabbergasted and my command prompt cursor must have been exhausted. I had never seen the little guy do so much tearing across the screen to build a 3rdParty Package. Thankfully he was still blinking at the same rate as when he started, tough little bugger.
  - I vowed never to use CMake after that… Well then 2009 rolled around and the need to use VTK, ITK, dcmtk, boost etc all of witch had versions which used CMake. Quoting Homer (Simpson not Greek poet of Iliad) “D’oh!”
What else is CMake?

- What is meant by "family of tools"?
  - CTest and CDash
    - Automated test (CTest) and reporting (CDash)
  - CPack
    - Package software for distributions
- We will focus on CMake in this talk
Project life cycle

- Code
- Build
- Run tests
- Report test results
- Bug reports
- Consume coffee or other caffeinated beverage of choice
- Rinse, wash, repeat.
- Finally after project completion:
  - Requirements Analysis 😊 i.e. what should we have done in the beginning – keeps us programmers employed!
Why use CMake?

- Cross platform support
- Plain text files for build specification which can be tracked and diffed (WinMerge/Meld) easily with source control tools (git)
- Code generation, configuration file generation, and text manipulation with configure_file(…)
- Powerful scripting
- Create your own generator
- Regular Expressions
- Not an exhaustive list!
What does CMake Look Like?

# The name of our project is "HELLO". CMakeLists files in this project can
# refer to the root source directory of the project as ${HELLO_SOURCE_DIR} and
# to the root binary directory of the project as ${HELLO_BINARY_DIR}.
cmake_minimum_required (VERSION 2.6)

project (HELLO)

# Recurse into the "Hello" and "Demo" subdirectories. This does not actually
# cause another CMake executable to run. The same process will walk through
# the project's entire directory structure.
add_subdirectory (Hello)
add_subdirectory (Demo)

Source: http://www.CMake.org/CMake/help/examples.html
What does CMake Look Like?

# Make sure the compiler can find include files from our Hello library.
include_directories (${HELLO_SOURCE_DIR}/Hello)

# Make sure the linker can find the Hello library once it is built.
link_directories (${HELLO_BINARY_DIR}/Hello)

# Add executable called "helloDemo" that is built from the source files
# "demo.cxx" and "demo_b.cxx". The extensions are automatically found.
add_executable (helloDemo demo.cxx demo_b.cxx)

# Link the executable to the Hello library.
target_link_libraries (helloDemo Hello)

Source: http://www.CMake.org/CMake/help/examples.html
What does CMake Look Like?

- CMake provides a GUI – CMake GUI
- Processes root CMakeLists.txt file
  - **Not** CMakeLists.txt or CMakelists.txt or CMakelist.txt or CMakeList.txt or CMakelists.txt or CMakelists.Txt – remember cross platform (Linux)
- Allows user to interact with the build and change build parameters
- Build spec creator can specify their own parameters which appear in the GUI
- Where to build the binaries – recommended out of source builds
- Delete Cache button was moved to File->Delete Cache in newer versions 2.6.
- When generating build specs I use copious amounts of Delete Cache and wish it were still a button (speed).
- Kitware needs to update their website with screen shots of 2.8!

http://www.CMake.org/CMake/help/runningCMake.html

"I posted a desirement in the CMake Mantis bug tracker only to find out that all one has to do in CMake GUI is File->"Delete Cache" then config, config, generate, wait for CMake VS Macros to notice something is awry and update the projects... I still wish there were a button. If I could get a heart rate monitor to sense my level of frustration and automatically rerun a script to delete the cache I think this would be the optimal solution." – Brian J. Davis http://comments.gmane.org/gmane.comp.lib.boost.CMake/821, 2010
What Does CMake Circa 2.8 Look Like?
CMake cache

- Cache is generated when all build parameters have been set in GUI.
- Cache generation can require multiple configure steps as when build parameters change others can be activated which require user to ok or allow the user to change.
- Changes are in red until accepted and new ones based on user changes will appear in red.
- Clearing the cache
  - File->Delete Cache
CMake cache

- Why a cache?
  - Speed – no need to reparse all CMakeLists.txt unless they change.
- There is a dependency on CMakeLists.txt files.
  - Consequences:
    - Do not track VS project files or Makefiles (CMake output) with source control management (SCM).
      - Show of hands, Who is Using a SCM tool? Should be everyone!!
      - Windows file explorer copy directory does not count nor does zip and copy!!!
    - VS projects and Makefiles will be regenerated.
    - Never Change CMake output!
      - Except: from my understanding you can change the cache. Though not a good idea unless careful what you change.
      - Cache is loaded when CMake is loaded to acquire previous build settings.
CMake cache

- Cache and SET(..)
  - To set variables in CMake SET(..) is used

```cmake
set(<variable> <value>
    [[CACHE <type> <docstring> [FORCE]] | PARENT_SCOPE])
```

- Example

```cmake
SET( MY_STR "HELLO" CACHE STRING "hello text" FORCE )
```

- What does FORCE do?
  - No matter what is typed in the GUI in an attempt to change the variable the variable will always be "HELLO"
  - Be careful with FORCE as remember the cache gets reloaded when CMake is run if you decide to change a value from FORCE to not FORCED then you need to delete the cache and regenerate

- A non FORCE example with use of booleans

```cmake
SET( MY_BOOL YES CACHE BOOL "my boolean" )
```

- Allows the user to change MY_BOOL in the GUI and have the modified value change in the cache
- Can also use: option( MY_BOOL YES )
Example Project

- Checkout using SVN
  - I was forced to use SVN 😞
- Use CAE username and password
- Create your own CMakeLists.txt
- Rename CMakeLists.txt.example to CMakeLists.txt to see example use of SAP - my (mis)use of CMake
- Run go.bat
  - Extracts CMake from zip archive – downloaded from CMake website. It is not modified.
  - Launches CMake
  - Simply for ease of use to get started

svn+ssh://bdavis@svn.caewisc.edu/filespace/people/b/bdavis/svn/me964cmake/trunk
Example Project

- Click Configure
- Click Generate
Example Project

- Browse to location of top (trunk directory)
- CMakeLists.txt file then to
  - trunk\build\ME964.sln (project)
  - trunk\source\cpp\project1 (source)
- Very simple example project is generated.
- Uses SAP (my CMake code)
  - Submitted as feature request
- NEVER USE SEMICOLONs AFTER COMMANDS IN CMake—can lead to all kinds of confusion as to where the error is

```cmake
add_project_executable(
  # the name of your executable
  project1
  # Defines if you need any
  DEFINES
    MY_DEF=1
  # A list of .cu sources
  CU_SOURCES
    project1.cu
  # A list of .cpp sources
  CPP_SOURCES
    main.cpp
  INCLUDE_DIRECTORIES
    ../include
  INSTALL_DIRECTORIES
    bin)
```

http://www.CMake.org/Bug/view.php?id=11807
Example Project

- What did I add?
  - `add_project` – not fully implemented… it’s a WIP – really just a placeholder for now. Goal was to have namespace resolution at least a form that would be possible in Cmake by prepending variables with project name.
  - `add_project_configuration` – allows specification of a configuration which can be inherited
  - `add_project_executable` - creates a executable and can inherit project configurations (multiple)
  - `add_project_library` – creates a library which compiles C/C++/CUDA and can inherit configurations (multiple)
  - `patch` – patches files
  - `unpack` – unzips .tar, .bz2, and zip

- Remember Cmake was unmodified these scripts are in the CMake directory at root of tree. CMake build tool and related source is found in platform/3rdParty/tools directory when extracted from zip archive.
Example Project Using Vanilla CMake

- Browse to location of `CMAKE_HOME_DIRECTORY` which is the path to top of source tree where root CMakeLists.txt file is located
- Copy CMakeLists.txt.vanilla to CMakeLists.txt file
- Delete cache if necessary (if you tried previous project)
- Configure and Generate project
- Browse to
  - `trunk\build\ME964.sln` (project)
  - `trunk\source\cpp\project1_vanilla` (source)
- Very simple example project is generated.
# Root CMakeLists.txt file

# check required version of CMake
CMAKE_MINIMUM_REQUIRED(VERSION 2.0)
#IF(CMAKE_BACKWARDS_COMPATIBILITY GREATER 2.0.6)
#  SET(CMAKE_BACKWARDS_COMPATIBILITY 2.0.6 CACHE STRING "Latest version of CMake when this project was released." FORCE)
#ENDIF(CMAKE_BACKWARDS_COMPATIBILITY GREATER 2.0.6)

if(COMMAND cmake_policy)
  cmake_policy(SET CMP0003 NEW)
endif(COMMAND cmake_policy)

# Declare the project
PROJECT(ME964)

SET( MY_STR “HELLO” CACHE STRING “hello text” FORCE )
SET( MY_BOOL YES CACHE BOOL “my boolean” )

add_subdirectory( source/cpp/project_vanilla/src project_vanilla )
Example Project Using Vanilla CMake

- **cmake_policy**
  
  "As CMake evolves it is sometimes necessary to change existing behavior in order to fix bugs or improve implementations of existing features. The CMake Policy mechanism is designed to help keep existing projects building as new versions of CMake introduce changes in behavior. Each new policy (behavioral change) is given an identifier of the form "CMP<NNNN>" where "<NNNN>" is an integer index. Documentation associated with each policy describes the OLD and NEW behavior and the reason the policy was introduced. Projects may set each policy to select the desired behavior. When CMake needs to know which behavior to use it checks for a setting specified by the project. **If no setting is available the OLD behavior is assumed and a warning is produced requesting that the policy be set.**"

  - Want to know more then see policies section of reference listed below

- **project(<projectname> [languageName1 languageName2 ... ] )**
  
  - Sets language to be used
  - Default C/C++
  - Not what I was expecting as projects can have multiple subprojects

- **find_package(<package> [version] [EXACT] [QUIET] [[REQUIRED|COMPONENTS] [components...]] [NO_POLICY_SCOPE])**
  
  - FIND_PACKAGE( CUDA )

- **add_subdirectory(source_dir [binary_dir] [EXCLUDE_FROM_ALL])**
  
  - add_subdirectory( source/cpp/project_vanilla/src project_vanilla )
    - Adds the subdirectory source/cpp/project_vanilla/src containing a CMakeLists.txt file for the exe and library.
    - Example CMakeLists.txt file to follow.
    - EXCLUDE_FROM_ALL removes from all target such as “make all” requiring a manual build command specifying target

Source: http://www.CMake.org/CMake/help/CMake-2-8-docs.html
Example Project Using Vanilla
CMake


Fairly self explanatory which is nice!



Override where CMake would like to
put the files default C:\Program
Filles\ME964

# OVERRIDE WHERE CMake WOULD LIKE TO INSTALL THE FILES
SET(
CMAKE_INSTALL_PREFIX
${CMAKE_HOME_DIRECTORY}/install
CACHE STRING "" FORCE
)
include_directories( ../include )



Include directories

SET( MY_LIB_NAME libv )
SET( MY_APP_NAME project_vanilla )



Need to use CUDA_ADD_LIBRARY
which is from part of FindCUDA
when we called find_package(
CUDA)

# for normal C/C++ code add_library would be used,
# but since this contains a .cu file
# CUDA_ADD_LIBRARY must be used



Set link flags to export function in dll



Specify where to install the app and
lib in ${CMAKE_INSTALL_PREFIX
}/bin and
${CMAKE_INSTALL_PREFIX }/lib
respectively



Configure a file which uses a
variable in the file throug use of
syntax ${SOME_VAR}.


When Configured the place holder in
the file will be replaced with the value.

CUDA_ADD_LIBRARY(
${MY_LIB_NAME}
project_vanilla.cu
libv.cpp
)
# need to export the function
set_target_properties(
${MY_LIB_NAME}
PROPERTIES
LINK_FLAGS /export:my_entry_function
)
add_executable(
${MY_APP_NAME}
main.cpp
)
target_link_libraries( ${MY_APP_NAME} ${MY_LIB_NAME} )
install( TARGETS ${MY_APP_NAME} DESTINATION bin )
install( TARGETS ${MY_LIB_NAME} DESTINATION lib )
SET( SOME_VAR "This is what gets put in run_program.bat when configured" )
configure_file( run_program.txt ${CMAKE_INSTALL_PREFIX}/bin/run_program.bat )

29


CMake 20/80

- Along the lines of Dan’s 20/80 rule
- This is more like the 80/5 rule
  - 80 percent of the functions – there aren’t that many to do 5% of your work which is the build specification.
  - And some of these are paired like if(), else(), elseif() etc so maybe it’s like the 40/5 (half as much) rule with the remaining 95 percent (of the number in denominator) going to coding in C++ which is what we should be doing anyway.
  - If you look at the example there are even fewer used, but this is a simple example. CMake allows things to get much, ... much more complicated which is good.
  - Complexity when you need it simplicity when you don’t

```
add_custom_command
add_custom_target
add_definitions
add_dependencies
add_executable
add_library
add_subdirectory
break
configure_file
cmake_policy
code
else
elseif
endforeach
endforeach
endif
endfunction
endmacro
endwhile
execute_process
export
file
find_file
find_package
function
if
include
include_directories
install
link_directories
macro
message
option
project
return
set
string
target_link_libraries
while
add_custom_command
```
CMake project regeneration

- If VS is open and CMake regenerates project files the dialog to the left will appear.
- CMake cannot be used (it is locked up) until this dialog in VS is accepted (“Yes”) and any remaining “Regenerate project” dialogs that may appear.
Back To Vanilla CMake

- Directory listing
  - Want find and grep then install Cygwin or GNUWin32 Utils.
  - Gates did not kill the command prompt and as we all know with out massive amounts of data compression 640k just isn’t going to cut it.
Vanilla CMake Example Cont

- **Main.cpp (below)**
- **project_vanilla.cu (left)**

```cpp
#include <iostream>
#include <project_vanilla.h>

int main( void )
{
    my_entry_function();
    std::cout << "all is well in the universe\n";
}
```

```cpp
#include <project_vanilla.h>
#include <iostream>
#include <cuda.h>

__global__ void project1( float* data )
{
    int index = threadIdx.x + blockDim.x * threadIdx.y;
    *(data + index) = index;
}

#define BLOCK_SIZE 8

_EXPORT_FUNCTION void my_entry_function( void )
{
    float* dev_data;
    size_t size = BLOCK_SIZE * BLOCK_SIZE * sizeof( float);
    float host_data[BLOCK_SIZE * BLOCK_SIZE];
    cudaMalloc( &dev_data, size );
    std::cout << "entry_foo has been entered\n";
    dim3 numThreads(BLOCK_SIZE, BLOCK_SIZE);
    dim3 numBlocks(1, 1, 1);
    project1<<<numBlocks, numThreads>>>(dev_data);

    cudaMemcpy( host_data, dev_data, size, cudaMemcpyDeviceToHost);
    for( int row = 0; row < BLOCK_SIZE; row++ )
    {
        for( int col = 0; col < BLOCK_SIZE; col++ )
        {
            std::cout << "\t" << host_data[col + row * BLOCK_SIZE];
        }
        std::cout << std::endl;
    }
    cudaFree( dev_data );
}
```
Vanilla CMake Example Cont

- run_program.txt which becomes run_program.bat after configure_file
- Note `${SOME_VAR}` usage. Will be replaced with value defined in CMake when .bat file is generated
- cmd /k
  - Just keeps command window from disappearing so commands can be typed after double clicking .bat file in file explorer
- PWD sets present working directory to current directory

```
SET PWD=%CD%

echo ${SOME_VAR}

cmd /k
```
CMake Macros/Functions and Parse Arguments

- When writing your own functions and macros you’ll likely need the parse_arguments function
  - [http://www.itk.org/Wiki/CMakeMacroParseArguments](http://www.itk.org/Wiki/CMakeMacroParseArguments)
  - You can probably guess what it is good for.

```cmake
SET(arguments hello OPTION3 world LIST3 foo bar OPTION2 LIST1 fuz baz )
PARSE_ARGUMENTS(ARG "LIST1;LIST2;LIST3" "OPTION1;OPTION2;OPTION3" ${arguments})
```

- The parameters are then dereferenced with syntax `${ARG_LIST1}` and `${ARG_OPTION2}` as an example
CMake Commands

- `message([STATUS|WARNING|AUTHOR_WARNING|FATAL_ERROR|SEND_ERROR] "message to display" ...)

- Output shows up in Cmake GUI output window.

- Very handy when trouble shooting build scripts
CMake Commands

- **if() else() endif()**
  
  if(expression)
  
  # then section.
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
  elseif(expression2)
  # elseif section.
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
  else(expression)
  # else section.
  COMMAND1(ARGS ...)
  COMMAND2(ARGS ...)
  ...
  endif(expression)

- **You might see this expression where beginning if and else need same expression. I never do this and only ever put expression in first if and not in trailing else, elseif, and endif etc.**

- **From CMake FAQ:** “As of CMake 2.6.0 the ELSE() and ENDIF() constructs can be empty.”

- **There are more than those shown below. Just showing most commonly ones I use. Documentation is omitted see online reference manual for more.**
  
  if(<constant>)
  if(<variable>)
  if(NOT <expression>)
  if(<expr1> AND <expr2>)
  if(<expr1> OR <expr2>)
  if(TARGET target-name)
  if(EXISTS file-name)
  if(EXISTS directory-name)
  if(IS_DIRECTORY directory-name)
  if(IS_ABSOLUTE path)
  if(<variable|string> MATCHES regex)
  if(<variable|string> LESS <variable|string>)
  if(<variable|string> GREATER <variable|string>)
  if(<variable|string> EQUAL <variable|string>)
  if(<variable|string> STRLESS <variable|string>)
  if(<variable|string> STRGREATER <variable|string>)
  if(<variable|string> STREQUAL <variable|string>)
  if(DEFINED <variable>)
  if((expression) AND (expression OR (expression)))
CMake Commands

- `file(WRITE filename "message to write")`
- `file(APPEND filename "message to write")`
- `file(READ filename variable [LIMIT numBytes] [OFFSET offset] [HEX])`
- `file(STRINGS filename variable [LIMIT_COUNT num] [LIMIT_INPUT numBytes] [LIMIT_OUTPUT numBytes] [LENGTH_MINIMUM numBytes] [LENGTH_MAXIMUM numBytes] [NEWLINE_CONSUME] [REGEX regex] [NO_HEX_CONVERSION])`
- `file(GLOB variable [RELATIVE path] [globbing expressions])`
- `file(GLOB_RECURSE variable [RELATIVE path] [FOLLOW_SYMLINKS] [globbing expressions])`
- `file(RENAME <oldname> <newname>)`
- `file(REMOVE [file1 ...])`
- `file(REMOVE_RECURSE [file1 ...])`
- `file(MAKE_DIRECTORY [directory1 directory2 ...])`
- `file(RELATIVE_PATH variable directory file)`
- `file(TO_CMAKE_PATH path result)`
- `file(TO_NATIVE_PATH path result)`
- `file(DOWNLOAD url file [TIMEOUT timeout] [STATUS status] [LOG log] [EXPECTED_MD5 sum] [SHOW_PROGRESS])`
CMake Commands

- String regex – regular expression
- String comparisons
- To Upper/Lower case

```
string(REGEX MATCH <regular_expression> <output variable> <input> [<input>...])
string(REGEX MATCHALL <regular_expression> <output variable> <input> [<input>...])
string(REGEX REPLACE <regular_expression> <replace_expression> <output variable> <input> [<input>...])
string(REPLACE <match_string> <replace_string> <output variable> <input> [<input>...])
string(COMpare EQUAL <string1> <string2> <output variable>)
string(COMpare NOTEQUAL <string1> <string2> <output variable>)
string(COMpare LESS <string1> <string2> <output variable>)
string(COMpare GREATER <string1> <string2> <output variable>)
string(ASCII <number> [<number>...] <output variable>)
string(CONFIGURE <string1> <output variable> [@ONLY] [ESCAPE_QUOTES])
string(ToUpper <string1> <output variable>)
string(Tolower <string1> <output variable>)
string(LENGTH <string> <output variable>)
string(SUBSTRING <string> <begin> <length> <output variable>)
string(STRIp <string> <output variable>)
string(RANDOM [LENGTH <length>] [ALPHABET <alphabet>] [RANDOM_SEED <seed>] <output variable>)
```
ExternalProject_ADD

- Adds external 3rdParty packages to your projects
- Can download source from repositories: SVN, GIT, CVS
- Build and install source that it CMake friendly

- Haven’t figured out how to get it to mow my lawn yet or make coffee, but look at the number of parameters… it’s huge … there’s got to be a way!
- Once it was elusive and it’s documentation could only be accessed by the command prompt.
Some points about CMake

- Flexible and powerful through use of a few commands
- There are more CMAKE_ variables see doc for more info
- Build settings are local to the directory.
  - What does this mean?
    - Build settings specified are “Global” to the directory unless set_target_properties is used
    - set_target_properties useful when you have 2 libs which use same source file with #ifdefs and you want to build them with different settings.
- Settings can have build configuration specific settings such as
  - LINK_FLAGS_<CONFIG>
  - LINK_FLAGS_DEBUG
  - LINK_FLAGS_RELEASE
- Beware of CACHE FORCE and clear cache if you think something is not quite right.
- IMO - Needs the concept of namespaces as CMake variables can grow in large projects
- project() is not what I expected
- add_project – does not work for third party packages such as vtk, dcmtk, ITK etc and ExternalProject_Add must be used.
  - This IMO does not allow to build for only what you need. - No true dependency checking across files. What gets build is all or none.
- Use fully qualified names for paths NOT relative.
  - Try relative first, but when it doesn't work switch to fully qualified names.
- Tar was supported in CMake –E, but not zip – used for unpacking source zip and tarballs.
- Watch for deprecated functions. Mostly what I see is consolidation (consolidating functions into one and increasing parameters) which makes sense.
CUDA Debugging

- Bug types
- Why use a debugger?
- Where did printf go?
- Current state of tools
- NSight
- cuda-gdb
- OpenCL

“I might at this point be better off programming an 256x256x256 array of industrial robots to move around beads on a matched series of abacuses. At least I could physically see where the problem was occurring.” -- Brian J. Davis, NVIDIA Developer Zone posting: [NSIGHT Confused by ? shows ??? I know I am confused] - 2011
Bug types

- **Bohr bug** – “A repeatable bug; one that manifests reliably under a possibly unknown but well-defined set of conditions. Antonym of **heisenbug**.”

- **Heisenbug** - “A bug that disappears or alters its behavior when one attempts to probe or isolate it. (This usage is not even particularly fanciful; the use of a debugger sometimes alters a program’s operating environment significantly enough that buggy code, such as that which relies on the values of uninitialized memory, behaves quite differently.) Antonym of **Bohr bug**.”

- **Mandelbug** (Mandelbrot) – “A bug whose underlying causes are so complex and obscure as to make its behavior appear chaotic or even non-deterministic.”

- **Schroedinbug** (Schroedinger's Cat thought-experiment) – “A design or implementation bug in a program that doesn't manifest until someone reading source or using the program in an unusual way notices that it never should have worked, at which point the program promptly stops working for everybody until fixed. Though (like bit rot) this sounds impossible, it happens; some programs have harbored latent schroedinbugs for years.”

- **Phase of the Moon bug** – “The phase of the moon is sometimes spouted as a silly parameter on which a bug might depend, such as when exasperated after trying to isolate the true cause. The Jargon File documents two rare instances in which data processing problems were actually caused by phase-of-the-moon timing.” Think Y2K. **Yes computers do manifest certain weird behavior based on the alignment of the planets**

- **Statistical (Stat) bug** – “Statistical bugs can only be detected in aggregates and not in single runs of a section of code. These are bugs that usually affect code that is supposed to produce random or pseudo-random output.”

Source:
- Jargon File - http://www.catb.org/jargon/
Why use a debugger?

- Clean code – no need to sprinkle with printf, #ifdef _DEBUG… #endif, or macros
- Zero in on the thread, block, and grid (ID) that is causing the problem and see state of variables.
- Set watch points and if equals, less, and more (boolean) operations etc
- If it (Language) does not have a debugger in this day and age I don’t use it. i.e. I don’t waste my time (except for playing Angry Birds, but that’s a choice).
- SAVE TIME! SAVE TIME! SAVE TIME! That can be better utilized playing Angry Birds or sailing.
Where did printf go?

- Removed prior to 3.0? (not sure exactly) release
  - NVIDIA CUDA Linux Release Notes Version 3.1
    - Added the ability to call printf() from kernels. This feature is supported only on the Fermi architecture.

- Even a better question why printf in the first place?
  - Let’s think about this for a second.
  - Which thread in what block int the grid is this printf running?
    - When does it run?
      - Might need it in the future so add a #define #ifdef #endif. Result = messy code
  - Without if statements this can generate a lot of text.
  - How do I read this text? This text has to be copied over to the CPU.
  - Who’s going to read all that text?
  - Even armed with grep… not me.
  - Should be optimizing GPU code for calculation throughput not printf statements
Current Tools


- There are likely more as this is not meant to be an exhaustive list and discussion of every debugger

- This talk will focus on 2
  - Parallel Nsight
  - cuda-gdb

- Visual Profiler was covered in a previous talk
Parallel Nsight

- CUDA C/C++ Debugging
- CUDA Kernel Trace/Profiling
- Data breakpoints for CUDA C/C++ code
- OpenCL Kernel Trace/Profiling – OK but what about debugging?
- Now with VS 2010 support

Parallel Nsight - Features

<table>
<thead>
<tr>
<th>Feature</th>
<th>No-cost Download</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integrated into</td>
<td>✔</td>
</tr>
<tr>
<td>Visual Studio 2008 SP1 or Visual Studio 2010</td>
<td>✔</td>
</tr>
<tr>
<td>CUDA C/C++ Debugging</td>
<td>✔</td>
</tr>
<tr>
<td>DirectX 10/11 Shader Debugging</td>
<td>✔</td>
</tr>
<tr>
<td>DirectX 10/11 Frame Debugging</td>
<td>✔</td>
</tr>
<tr>
<td>DirectX 10/11 Frame Profiling</td>
<td>✔</td>
</tr>
<tr>
<td>CUDA Kernel Trace/Profiling</td>
<td>✔</td>
</tr>
<tr>
<td>OpenCL Kernel Trace/Profiling</td>
<td>✔</td>
</tr>
<tr>
<td>DirectX 10/11 API &amp; HW Trace</td>
<td>✔</td>
</tr>
<tr>
<td>Data breakpoints for CUDA C/C++ code</td>
<td>✔</td>
</tr>
<tr>
<td>Analyzer/System Trace</td>
<td>✔</td>
</tr>
<tr>
<td>Tesla Compute Cluster (TCC) Support</td>
<td>✔</td>
</tr>
<tr>
<td>Forum Support</td>
<td>✔</td>
</tr>
<tr>
<td>All Version Upgrades</td>
<td>✔</td>
</tr>
</tbody>
</table>
Nsight Monitor and Debugger

- Nsight monitor must be started before the program to be debugged can be launched
- Nsight debugger attaches to monitor
- Secured Connections
  - Allows only certain computers to connect
- File synchronization
  - Needed if remote debugging and dlls, config files, etc are needed.
Nsight Configuration

- **Nsight Monitor**
  - **Headed Mode - now called Local Mode**
    - Can be used when computer has more than 1 GPUs installed
    - WDDM TDR
      - “TDR stands for Timeout Detection and Recovery. This is a feature of the Windows operating system which detects response problems from a graphics card, and recovers to a functional desktop by resetting the card. If the operating system does not receive a response from a graphics card within a certain amount of time (default is 2 seconds), the operating system resets the graphics card. Before TDR existed, problems of this nature would have resulted in a system freeze and required a reboot of the operating system. If TDR is enabled and you see the TDR error message "Display driver stopped responding and has recovered", this means that the Windows operating system reset the display driver.” – reference Nsight User manual
      - TDR crashes will also be seen on long running kernels.
  - **Headless Mode - now called Remote Mode**
    - Used when there is no display
    - Connections are made remotely from client running Nsight – I have not tried this… yet as I run headed mode
Nsight Requirements

  - OS
    - Windows® Vista (32 or 64-bit) with SP1, or
    - Windows® 7 (32 or 64-bit), or
    - Windows HPC Server 2008 (32 or 64-bit)
  - Local debugging (Headed Mode) (host and target on same machine)
    - 2 GPUs, each must be either a G92, GT200, or GT400 GPU. See below for supported graphics cards.
  - Remote debugging (Headless Mode) (host and target on different machines)
    - On the target machine:
      - 1 GPU on target machine: must be a G92, GT200, or GT400 GPU*
    - On the host machine (with Visual Studio):
      - 1 GPU on host machine: can be any GPU
  - Current supported cards at left
Nsight Requirements

  - Disable D3D acceleration for WPF (applies to local debugging only) Open Windows Explorer.
    - Browse to the Common folder:
      - On a Windows 32-bit system browse to: C:\Program Files\NVIDIA Parallel Nsight 1.51\Common
      - On a Windows 64-bit system browse to: C:\Program Files (x86)\NVIDIA Parallel Nsight 1.51\Common
    - Double-click on the file named:
      - DisableWpfHardwareAcceleration.reg

- Known working from experience:
  - Foxconn destroyer motherboard with onboard NVIDIA® 780a SLI Chipset and 4 Teslac1050s works just fine.
  - Also now Remote Desktop debugging works without device enumeration issue
Nsight in Action - Debugger

NVIDIA Instructional Video

Nsight in Action - Profiler

cuda-gdb

- More evolved than Nsight - though v2.0 is promising
- Released before Nsight
- DDD with cuda-gdb
  - ddd --debugger cuda-gdb
- Emacs – below text is from the url at the bottom of the slide.
  CUDA-GBD works with GUD in Emacs and XEmacs. No extra step is required besides pointing to the right binary.
  To use cuda-gdb, the 'gud-gdb-command-name' variable must be set to "cuda-gdb --annotate=3". Use M-x customize-variable to set the variable. Ensure that cuda-gdb is present in the Emacs/XEmacs $PATH.

cuda-gdb

- Remote ssh – for instance I remote debug my single gpu on my laptop using a remote ssh session
- Very good success using this method
  - ssh –X username@wherever
  - Stop your display manager (Ubuntu 10.10 x64 cmd shown below using gnome)
    - sudo stop gdm
    - set PATH and LD_LIBRARY_PATH as necessary
    - ddd –debugger cuda-gdb app_name
OpenCL – er what?

- Q: Is there a OpenCL debugger?
  
  There is some support for OpenCL in Parallel Nsight.

- Remember what I said. If it doesn't have a debugger I don't use it.

- When OpenCL does then I’ll switch.

High Performance Computing for Engineering Applications

University of Wisconsin – Madison
Guest Lecture – April 28, 2011

Virginia W. Ross, Ph. D.
Air Force Research Laboratory/Information Directorate
Virginia.Ross@rl.af.mil
315-330-4384

DISTRIBUTION STATEMENT A. Approved for public release; distribution unlimited.
Case numbers 88ABW-2011-1884 & 88ABW-2010-4976
Outline

• 500 TeraFLOPS Heterogeneous HPC Cluster
  – HPC Introduction
  – Hardware
  – Applications

• Cloud Computing for DoD
  – Cloud Computing Background
  – Federal Government Cloud Computing
  – Organizational Benefits Gained from Cloud Computing

• Conclusions
500 TeraFLOPS Heterogeneous Cluster
Outline

• 500 TeraFLOPS Heterogeneous HPC Cluster
  – HPC Introduction
  – Hardware
  – Applications

• Cloud Computing for DoD
  – Cloud Computing Background
  – Federal Government Cloud Computing
  – Organizational Benefits Gained from Cloud Computing

• Conclusions
HPC Introduction

• This system put AFRL/RI in the lead for hosting the largest interactive High Performance Computer (HPC) for the Department of Defense.

• The Cell BE cluster was transitioned to another facility.

• These machines are freely available to government researchers and their contractors.
What makes this advance possible?

• As the server market drove price-performance improvements that the HPC community leveraged over the past decade, now the gaming marketplace may deliver 10x-20x improvements (power as well).
  
  — $3800 3.2 GHz dual-quad core Xeon®, 96 Gflops (DP)- baseline system, Power 1000 Watts
  
  — $380 3.2 GHz PS3® with Cell Broadband Engine® 153 Gflops (SP), power 135 Watts
    • 1.6X Flops/board, 1/10th cost
  
  — $2000 NVIDIA Tesla C2050 (515Gflops (DP), 1.03Tflops (SP)), Power 225 Watts
    • 1/10th cost, 1/20th the power
Outline

• 500 TeraFLOPS Heterogeneous HPC Cluster
  – High Performance Computing (HPC) Introduction
  – Hardware
  – Applications

• Cloud Computing for DoD
  – Cloud Computing Background
  – Federal Government Cloud Computing
  – Organizational Benefits Gained from Cloud Computing

• Conclusions
PlayStation3 Fundamentals

- $380
- Cell BE® processor
- 256 MB RDRAM (only)
- 160 GB hard drive
- Gigabit Ethernet (only)
- 153 Gflops Single Precision Peak
  - 380 TFLOPS/$M
- Sony Hypervisor
- Fedora Core 7 or 9 Linux or YDL 6.2
- IBM CELL SDK 3.1

- 6 of 8 SPEs available
- 25.6 GB/sec to RDRAM
- ~110 Watts
AFRL/RIT Horus Cluster

10 - 1U Rack Servers

- **26 Tflops**
- Supports TTCP efforts
- 18 General Purpose Graphical Processor Units (GPGPUs) Cluster

NVIDIA C2050

1.1 TFLOPS SP
515 GFLOPS DP
Key Questions

• Which codes could scale given these constraints?
• Can a hybrid mixture of PS3s and traditional servers mitigate the weaknesses of the PS3s alone and still deliver outstanding price-performance?
• What level of effort is required to deliver a reasonable percentage of the enormous peak throughput?
• A case study approach is being taken to explore these questions
Early Access System Approach

• A 53 TeraFLOPS cluster of PlayStation® 3s was built at AFRL Information Directorate in Rome, NY to provide early access to the IBM Cell Broadband Engine® chip technology included in the low priced commodity gaming consoles.

• A heterogeneous cluster with powerful subcluster headnodes is used to balance the architecture in light of PS3 memory and input/output constraints
  – 14 subclusters each with 24 PS3s and a headnode

• Interactive usage

• Used by HPCMP community for experimentation
• The Cell Cluster has a peak performance of 51.5 Teraflops from 336 PS3s and additional 1.4 TF from the headnodes on its 14 subclusters.
  • Cost: $361K ($257K from HPCMP)
    • PS3s 37% of cost
  • Price Performance: 147 TFLOPS/$M
• The 24 PS3s in aggregate contain 6 GB of memory and 960 GB of disk. The dual quad-core Xeon headnodes have 32 GB of DRAM and 4 TB of disk each.
500 TFLOPS Architecture (2010)

- ~300 Tflops from 2000 PS3s
- ~200 Tflops from GPGPUs on subcluster headnodes
- Cost: ~$2M
500 TFLOPS Architecture

CONDOR CLUSTER
Online: December 2010

- Approx. 270 TFLOPS from 1,760 PS3s
  - 153 GFLOPS/PS3
  - 80 subclusters of 22 PS3s
- Approx. 230 TFLOPS from subcluster headnodes
  - 2 GPGPU (2.1 TFLOPS / headnode)
  - 84 headnodes (Intel Nehalem 5660 dual socket Hexa (12 cores))
  - *Horus Cluster (~26 Tflops)
- Cost: Approx. $2M
- Total Power 300KW
CONDOR Node (Dual Nahlem x5650, 24 GB Ram, 2TB HD, 1200W PS, 2 Tesla GPGPU, 40Gb/s Inf, Dual 10Gb (2.5 Tflops SP or 1.2 Tflops DP)

Legend

- 2U Compute Node 2.5 TF/s
- 10GbE/1GbE switch
- PS3
This project provides the HPCMP community with early access to HPC scale commodity multicore through a 336 node cluster of PS3 gaming consoles (53 TF).

Applications leveraging the >10X price-performance advantage include:

large scale simulations of neuromorphic computing models
GOTCHA radar video SAR for wide area persistent surveillance
Real-time PCID image enhancement for space situational awareness

Dr. Richard Linderman, AFRL/RI, Rome, NY

... but beginning to perceive that the handcuffs were not for me and that the military had so far got ...

Neuromorphic example:
Robust recognition of occluded text

Gotcha SAR

PCID Image Enhancement
Outline

• 500 TeraFLOPS Heterogeneous HPC Cluster
  – HPC Introduction
  – Hardware
  – Applications

• Cloud Computing for DoD
  – Cloud Computing Background
  – Federal Government Cloud Computing
  – Organizational Benefits Gained from Cloud Computing

• Conclusions
19

Neuromorphic Computing Architecture Simulation Case

• The driving application behind developing a 53 TF class cluster was to support basic research into alternative neuromorphic computing architectures.

• The first of these to be optimized for the PS3 was the “Brain-State-In –A-Box” (BSB)—looking for 1M BSBs simulating in real time

• Optimized the BSB for the PS3 and achieved 18 GFLOPS on each core of the PS3 [6]. Across the 6 cores, 108 GFLOPS/PS3, over 70% of peak was sustained.
  — 12 staff week effort for first PS3 optimization experience

• Constructing hybrid simulations with BSBs and “Confabulation” models
Minicolumn Model
Hybrid: Attractor + Geometric Receptors

Mechanisms identified during initial effort are being applied to a closely neuromorphic columnar model we are emulating on a Cell-BE Cluster.

Literature reviews: minicolumn anatomy, cortical anatomy, cortical modeling, Cog Psyc, Neural Sci.

Explored attractors (BSB, Willshaw, PINN, Sparse Distributed Memory, Limit cycle) & arrays of (Erzatz Brain, Liquid State Machines).

Assessment of Confabulation: algorithm complexity, efficacy, acceleration.

Development of Hybrid model:
Simple/Complex cell minicolumn, functional columns, full scale V1.

Spiky Neuron Dynamical Modeling: emulation exercise – 64 minicolumns assembled as a functional column.
Neuromorphic Vision System

SONY EVI-HD1 Cameras
- 1080p @ 29.97 fps (60Hz)
- 10x Optical / 4x Digital Zoom

JBCC
- Pointing (FOV Modeling)
- Stereo Vision Disparity (MAE)
- Slice Storage
- Object Recognition (MAE Threshold)

Sensor to HPC

100K JBI Pub/Sub
- Publishing 60 fps
- 3,236 msgs/frame
- Payload: 80 MB/frame
- 37.5 gigabits/sec

196 PS3s using BSB models to compute:
- Orientation lines
- Color
- Light intensity
Mapping V1 Model to HPC
One Subfield per PS3 Node

A “neighborhood” of 6 subfields.

- Full V1 (196 Subfields)
- 196 PS3 nodes emulating a Full V1

Connectivity pattern of a Functional Column
Intense local communications within 3 mm

1 Subfield:
- 2 Field of Views
- 64 Functional Columns per FOV

1 Functional Column:
- 64 minicolumns

1 Minicolumn:
- 32 Element BSB
- “Simple & complex cells
- “Readout” cells

Consensus of percepts

196 Subfields x 2 FOV x 64 FC’s x 64 MC’s = 1,605,632 Minicolumns
Results: Neuromorphic Modelling

- Cell BE Cluster networking infrastructure is more than up to the challenge of handling the I/O from most I/O intensive models under examination (400-500 Hz update far exceeds 100 Hz real-time need)
...but beginning to perceive that the handcuffs were not for me and that the military had so far got....
Confabulation Architecture
Core & Processor Level Parallelism

Core level parallelism
Multi-threading, Shared memory

Processor level parallelism
Loosely synchronous, MPI communication

8-core Xeon

Word Level Confab.
Sentence Confab.

2009

24 PS3s

Dispatch Images
Receive Letters

12-core Intel Xeon Processor

22 PS3s

Performance Monitor

BSB BSB BSB

Sub-cluster 1
## Performance Evaluation

<table>
<thead>
<tr>
<th></th>
<th>PS3 node 1 Cell Processor</th>
<th>Sub-cluster 1 head-node + 24 PS3</th>
<th>HPC cluster 14 sub-clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computing power from Cell processors (GFLOPS)</td>
<td>75</td>
<td>1800</td>
<td>25200</td>
</tr>
<tr>
<td>Character recognition peak performance (characters / sec)</td>
<td>48</td>
<td>1152</td>
<td>16128</td>
</tr>
<tr>
<td>Word confabulation peak performance (words / sec)</td>
<td>N/A</td>
<td>30</td>
<td>420</td>
</tr>
<tr>
<td>Sentence confabulation peak performance (sentences / sec)</td>
<td>N/A</td>
<td>160</td>
<td>2240</td>
</tr>
<tr>
<td>Overall typical text recognition performance (sentences / sec)</td>
<td>N/A</td>
<td>4.3</td>
<td>59.9</td>
</tr>
</tbody>
</table>
Video Synthetic Aperture Radar Backprojection Case

• This algorithm is expensive computationally, but allows SAR radar images to focus each pixel independently, accounting for variations in elevation.

• This algorithm was accelerated >300X over original XEON code and achieved 40% of peak (60 GFLOPS sustained) on each PS3.

• 8 PS3s and headnode flown in 2007 for 1.5km spot

• 96 PS3s demonstrated 5km spot processing in Lab in May 08

• 9 1U Servers (8 with dual GPGPUs)
  — 2km x10km swath, 2.2 TFLOPS sustained

• 20 KM spot-72 TFLOPS, 40 KM spot 350 TFLOPS
Results: Gotcha VideoSAR Scalability

- At 256 PS3s, each send 6 MB/sec and receives 8.4 MB/sec while headnodes each receive 200 MB/sec and send 140 MB/sec
**SAR Image Formation using GPGPUs**

**9 KM SPOT**  
**500Hz PRF**  
**16 Tesla Boards**

64 MB Pulse Packs (500 pulse segments)

Master Nodes: Pulse Compression, Form, Store Image, publish frames

Node 9: Collects aux data, merges radar signals and sends to Master Nodes

Node 9 merges segments and sends them via TCP

Each Node does Pulse compression
High Definition (HD) Video Processing Case

- This case study employed 3 PS3s and a headnode to process 1080p grayscale HD video (52 MB/sec at 1080x1920) in real time.

- The sum of absolute differences algorithm was first optimized to process 64 frames/second at HD resolution (1920x1080). 21 GFLOPS was sustained on one PS3.

- Flux tensor optimized to 60 GFLOPS (40% of peak), 92X Xeon, in 3 staff months.

- The headnode played a key role of both archiving and disseminating the HD video to the PS3s in parallel streams.
Large Matrix-Matrix Multiply Case

- Matrix multiplication runs near peak for small matrices on a single PS3—but can it scale across a cluster and still have excellent performance?
- In theory, yes!
  - Source portions of the A matrix from local disk
  - Multicast the B matrix in on gigabit ethernet
- Progress to date:
  - Extended Dresden square MM to rectangular MM
  - Tested Ethernet (~90 MB/s) and Disk (35 MB/s) capabilities
  - Combining the pieces (where theory meets practice)
I/O Rates for Peak MM Performance Given Working Memory Available

Potential region for peak performance
Outline

• 500 TeraFLOPS Heterogeneous HPC Cluster
  – HPC Introduction
  – Hardware
  – Applications

• Cloud Computing for DoD
  – Cloud Computing Background
  – Federal Government Cloud Computing
  – Organizational Benefits Gained from Cloud Computing

• Conclusions
Cloud Computing for DoD
Outline

• 500 TeraFLOPS Heterogeneous HPC Cluster
  – HPC Introduction
  – Hardware
  – Applications
• Cloud Computing for DoD
  – Cloud Computing Background
  – Federal Government Cloud Computing
  – Organizational Benefits Gained from Cloud Computing
• Conclusions
Cloud Computing Background

• Rapid computer usage growth and Internet expansion

• Desire for computing application solution that is:
  ─ Cost-effective
  ─ Able to meet consumer needs, especially adaptability and availability
  ─ Reliable
  ─ Secure
Problem or Opportunity

- Large IT investment for computing
  - Financial
  - Manpower
- Centralization of function via cloud computing
  - Economy of scale
  - Efficient resource usage
  - Availability to large user base
  - Agility to meet changing needs
What is Cloud Computing?

• Computing resources held by provider
• Internet access to resources via PCs, laptops, smart phones, and PDAs
• Access to programs, storage, processing, and applications development
• Precursors include:
  – Thin clients
  – Grid computing
  – Utility computing
• Cloud computing is

- a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage, applications and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.

  (Mell & Grance, 2009, p. 8)
Increased Parallelism
- New Moore’s Law - 2X processors per chip generation
- Parallel software industries emerging to address challenges
- Redundant networks and storage increasing performance

Increased Virtualization
- Processing, Storage, Bandwidth, Delivery

Commodity Components
- X86 servers, consumer hard drives, ethernet
- Open Source SW – Freedom to customize and adapt

Increased Outsourcing of Core Elements
- “By 2012, 80 percent of Fortune 1000 companies will pay for some cloud computing service, and 30 percent of them will pay for cloud computing infrastructure.” Gartner

(Greenfield, 2009)
Four Cloud Deployment Models

• Internal (private) cloud
  — Enterprise owned or leased

• Community cloud
  — Shared infrastructure for specific community

• Public cloud
  — Sold to the public, mega-scale infrastructure

• Hybrid cloud
  — Composed of two or more clouds
Cloud Delivery Models

- **Software as a Service (SaaS)**
  - Using provider’s applications over a network

- **Platform as a Service (PaaS)**
  - Deploying customer applications to a cloud

- **Infrastructure as a Service (IaaS)**
  - Lease processing, storage, network, and other computing resources

- Services above are all deployed on a cloud infrastructure
Cloud Computing Growth

• Four hour non-parallel session dedicated to cloud computing at ISC
• Extensive emphasis on cloud computing by IEEE Computer Society
• NSF cloud computing
• NIST support of cloud computing
• Rapid growth in cloud computing in both public and private sector
Outline

• 500 TeraFLOPS Heterogeneous HPC Cluster
  – HPC Introduction
  – Hardware
  – Applications
• Cloud Computing for DoD
  – Cloud Computing Background
  – Federal Government Cloud Computing
  – Organizational Benefits Gained from Cloud Computing
• Conclusions
Federal Cloud Computing Goals

• White House wants “fundamental re-examination of investments in the technology infrastructure.” (Hoover, 2009)

• Federal CIO Vivek Kundra and NIST advocate cloud computing in government. (Hoover, 2009)

• GSA – Significant cost savings anticipated in the long term. (Hoover, 2009)

• GSA launched Apps.gov to provide cloud computing information and services for civilian agencies.
Federal Cloud Computing

- Federal Cloud Computing Initiative (FCCI)
  - The FCCI focuses on implementing cloud computing solutions for the Federal Government that increase operational efficiencies, optimize common services and solutions across organizational boundaries and enable transparent, collaborative and participatory government.
  - Cloud computing definition (NIST)
  - Hosting cloud computing summit
  - Released IaaS RFI and RFQ
  - Launching cloud computing storefront: apps.gov
  - Steering Committee, Advisory Council, Working Groups
Apps.gov

• Http://apps.gov by GSA for civilian agencies
  – Federal CIO - Promoting President’s agenda to modernize Federal IT
• Business applications
• Productivity applications
• Cloud IT services (IaaS: contract awarded)
  – Storage, virtual machines, web hosting
• Social media applications
Air Force Cloud Computing

• IBM effort to Design and Demonstrate Mission-Oriented Cloud Architecture for Cyber Security (2010)

• Air Force Research Laboratory/Information Directorate
  – University Center of Excellence (UCoE) in Assured Cloud Computing - to be awarded soon
  – Cloud Computing Collaboration with Cornell University
  – Ad-hoc Cloud Computing Collaborative Research with Harvard
  – STTR - Innovative Approaches to On-Demand Cloud Computing over Ad-Hoc Wireless Networks
  – HPC facility operates like cloud computing
NASA Nebula Platform

- Cloud computing pilot program at NASA Ames
- Integrates open-source components into seamless, self-service platform
- Provides high-capacity computing, storage, and network connectivity
- Virtualized, scalable approach
- Cost and energy efficient
- Mission support
- Education and public outreach

(NASA Nebula, 2010)
NSF Supported Cloud Research

- Cluster Exploratory (CLuE) program
- $5 M to 14 Universities
- IBM/Google Cloud Computing University Initiative
- Employ SW and services on IBM/Google cloud to explore innovative research ideas in data-intensive computing, including:
  - Image processing
  - Large scale data analysis
  - Internet improvement studies
  - Human genome sequencing
Cloud Comparison to DoD HPCMP

- DoD High Performance Computing Modernization Program (HPCMP) has supported:
  - Grid Computing
  - Centralized computing resources
  - Centralized authentication and security

- DoD HPCMP currently emphasizes support for parallel computing jobs with users knowledgeable about parallel processing

- DREN/SDREN network support
Significance to the DoD

- Cost savings
- Meet growing computational needs
- Enhance reliability
- Centralize security
- 24/7 availability
- Adaptable to changing needs
Outline

• 500 TeraFLOPS Heterogeneous HPC Cluster
  – HPC Introduction
  – Hardware
  – Applications

• Cloud Computing for DoD
  – Cloud Computing Background
  – Federal Government Cloud Computing
  – Organizational Benefits Gained from Cloud Computing

• Conclusions
Distributed Test Events

• Joint Mission Environment Test Capability (JMETC) provides infrastructure to link distributed facilities

• Incorporate Test and Training Enabling Architecture (TENA)

• Opportunity for cloud computing to capitalize on:
  – Network
  – Software infrastructure
  – Distributed test tools
  – Data management solutions
  – Reuse repository (Norman, 2010)
Cloud Computing Advantages

- Support for data intensive computing
  - Index and Parallelize large data sets
  - Support low BW transmission by data preformatting
- Ease of use – transfer complexity to cloud host
- Multi-user data access from large distributed cloud databases
- Default backup and cost effective archival for large data sets.
- Accessible any time, anywhere at low cost
Programming Models
What’s the right fit for DoD?

App-components-as-a-service

Software-platform-as-a-service

Virtual-Infrastructure-as-a-Service

Physical infrastructure

Hardware Resources

Compute

Storage

Networking

GCDS Akamai

Data Intensive
Amazon Hadoop, Public Data Sets, Simple DB

Amazon Elastic Compute Cloud (Amazon EC2) - Beta

Google App Engine

Azure

(Redfield, 2009, p 20)
## Storage data charges of cloud computing providers (SaaS)

<table>
<thead>
<tr>
<th>Vendor</th>
<th>Usage</th>
<th>Data transfer out</th>
<th>Data transfer in</th>
<th>No of requests</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amazon S3</td>
<td>$0.15/GB</td>
<td>$0.17/GB</td>
<td>No restrictions</td>
<td>$0.01/1000 requests</td>
</tr>
<tr>
<td>AT&amp;T Synaptic</td>
<td>$0.25/GB</td>
<td>$0.1/GB</td>
<td>$0.1/GB</td>
<td>Nil</td>
</tr>
<tr>
<td>GoGrid</td>
<td>$0.15/GB</td>
<td>No restrictions</td>
<td>No restrictions</td>
<td>No restrictions</td>
</tr>
<tr>
<td>Rackspace</td>
<td>$0.15/GB</td>
<td>No restrictions</td>
<td>No restrictions</td>
<td>No restrictions</td>
</tr>
</tbody>
</table>

- Prices shown for lowest usage tier and reduce with higher usage.

Source:
http://www.thecloudtutorial.com/cloudcomparison.html
Cost Savings

• Centralized resources and management yield economy of scale

• Cost reduction of 5-7x for power, network, operations, software, and hardware

• Reduced energy usage and higher utilization for green computing

• Capitalize on low cost locations
Need for Cloud Computing

- Provide resources not available to individual users
- Minimize up-front user expenses
- On demand availability, ability to handle surges
- Provider manages security
- Handle data intensive applications
- Mobile interactive applications
- Large parallel computing jobs
- Serve countries/organizations with limited resources
- Medical research
- Online gaming
Cloud Computing Reliability

- Software reliability is critical
  - Malware
  - Bugs
- Quality of service approach beneficial
- Reliable Internet connectivity
Security Effectiveness

- Data integrity
- Commingling of data
- Virtualization
- Cost versus risk issues
- Multicore for data separation
- Social engineering and human error
- Remote access/authentication
- Strong, enforced security posture
Availability and Usability

- Proprietary software
- Portability between vendors
- Standards
Cloud Adoption Study Implications

• Non-technical issues influence cloud computing adoption decisions.

• Consideration of the overall organizational impact of cloud computing is important.

• A complex interaction between vendors and potential customers, considering factors such as security, need, reliability, and cost, could maximize customer benefit. (Ross, 2010)
Outline

• 500 TeraFLOPS Heterogeneous HPC Cluster
  – HPC Introduction
  – Hardware
  – Applications

• Cloud Computing for DoD
  – Cloud Computing Background
  – Federal Government Cloud Computing
  – Organizational Benefits Gained from Cloud Computing

• Conclusions
Conclusions

• Commercial market brings economy of scale for HPC and cloud computing

• World-class 500 TFLOPS interactive HPC at RRS capitalizes on PS3s with powerful headnodes in a subcluster configuration with dual GPGPUs

• Several applications scaling very well and achieving significant percentage of Cell BE peak performance

• SAR backprojection algorithm scales on GPGPUs

• Cloud computing offers numerous potential benefits to the DoD

• AFRL is a leader in successful DoD cloud computing ventures


• NASA Nebula (2010). Retrieved from nebula.nasa.gov


Parallel Computing with MATLAB

Narfi Stefansson
Parallel Computing Development Manager
MathWorks
Agenda

- Products and terminology
- GPU capabilities
- Multi-process capabilities
- How are customers using this?
Parallel Computing with MATLAB on CPU

- **Parallel Computing Toolbox**
- **MATLAB Distributed Computing Server**
- **MATLAB Workers**

**User’s Desktop**

**Compute Cluster**
Evolving With Technology Changes

- GPU
- Single processor
- Multicore
- Multiprocessor
- Cluster
- Grid, Cloud
Why GPUs and why now?

- Double support
  - Single/double performance inline with expectations
- Operations are IEEE Compliant
- Cross-platform support now available
What came in R2010b?

- Parallel Computing Toolbox
  - GPU support
  - Broader distributed array algorithm support (QR, rectangular \\)

- MATLAB Distributed Computing Server
  - GPU support
  - Run as user with MathWorks job manager
  - Non-shared file system support

- Simulink®
  - Real-Time Workshop® support with PCT and MDCS
What came in R2011a?

- Parallel Computing Toolbox
  - Deployment of local workers
  - More GPU support
  - More distributed array algorithm support

- MATLAB Distributed Computing Server
  - Enhanced support for Microsoft HPC Server
  - More GPU support
  - Remote service start in Admin Center
GPU Support

- Call GPU(s) from MATLAB or toolbox/server worker
- Support for CUDA 1.3 enabled devices and up
Programming Parallel Applications

Level of control

Minimal

Some

Extensive

Required effort

None

Straightforward

Involved
Summary of Options for Targeting GPUs

**Level of control**

- Minimal
- Some
- Extensive

**Parallel Options**

- Use GPU arrays with MATLAB built-in functions
- Execute custom functions on elements of the GPU array
- Create kernels from existing CUDA code and PTX files
GPU Array Functionality

- Array data stored in GPU device memory
- Algorithm support for over 100 functions
- Integer, single, double, real and complex support
Example:

GPU Arrays

```matlab
>> A = someArray(1000, 1000);
>> G = gpuArray(A); % Push to GPU memory
...
>> F = fft(G);
>> x = G\b;
...
>> z = gather(x); % Bring back into MATLAB
```
GPUArray Function Support

- >100 functions supported
  - `fft`, `fft2`, `ifft`, `ifft2`
  - Matrix multiplication `(A*B)`
  - Matrix left division `(A\b)`
  - LU factorization
  - `' `.
  - `abs`, `acos`, ..., `minus`, ..., `plus`, ..., `sin`, ...
  - `conv`, `conv2`, `filter`
  - indexing
GPU Array benchmarks

<table>
<thead>
<tr>
<th>A\b*</th>
<th>Tesla C1060</th>
<th>Tesla C2050 (Fermi)</th>
<th>Quad-core Intel CPU</th>
<th>Ratio (Fermi:CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>191</td>
<td>250</td>
<td>48</td>
<td>5:1</td>
</tr>
<tr>
<td>Double</td>
<td>63.1</td>
<td>128</td>
<td>25</td>
<td>5:1</td>
</tr>
<tr>
<td>Ratio</td>
<td>3:1</td>
<td>2:1</td>
<td>2:1</td>
<td></td>
</tr>
</tbody>
</table>

* Results in Gflops, matrix size 8192x8192. Limited by card memory. Computational capabilities not saturated.
## GPU Array benchmarks

<table>
<thead>
<tr>
<th>MTIMES</th>
<th>Tesla C1060</th>
<th>Tesla C2050 (Fermi)</th>
<th>Quad-core Intel CPU</th>
<th>Ratio (Fermi:CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>365</td>
<td>409</td>
<td>59</td>
<td>7:1</td>
</tr>
<tr>
<td>Double</td>
<td>75</td>
<td>175</td>
<td>29</td>
<td>6:1</td>
</tr>
<tr>
<td>Ratio</td>
<td>4.8:1</td>
<td>2.3:1</td>
<td>2:1</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>FFT</th>
<th>Tesla C1060</th>
<th>Tesla C2050 (Fermi)</th>
<th>Quad-core Intel CPU</th>
<th>Ratio (Fermi:CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Single</td>
<td>50</td>
<td>99</td>
<td>2.29</td>
<td>43:1</td>
</tr>
<tr>
<td>Double</td>
<td>22.5</td>
<td>44</td>
<td>1.47</td>
<td>30:1</td>
</tr>
<tr>
<td>Ratio</td>
<td>2.2:1</td>
<td>2.2:1</td>
<td>1.5:1</td>
<td></td>
</tr>
</tbody>
</table>
Example:

arrayfun: Element-Wise Operations

```matlab
>> y = arrayfun(@foo, x);  % Execute on GPU
```

```matlab
function y = foo(x)
y = 1 + x.*(1 + x.*(1 + x.*(1 + x.*(1 + x.*(1 + x.*(1 + x.*(1 + x.*(1 + x.*(1 + x.*(...
    x.*(1 + x.*(1 + x.*(1 + x.*(1 + x.*(...
    x.*(1 + x./9)./8)./7)./6)./5)./4)./3)./2);
```
Some `arrayfun` benchmarks

Black-Scholes benchmark performance

- Baseline, CPU[4]
- CPU[1]
- Tesla C1060
- Tesla C2050

Point-in-polygon benchmark performance

- Baseline, CPU[4]
- CPU[1]
- Tesla C1060
- Tesla C2050

Note: Due to memory constraints, a different approach is used at N=15 and above.

`CPU[4] = multhithreading enabled`

`CPU[1] = multhithreading disabled`
% Setup
kern = parallel.gpu.CUDAKernel('myKern.ptx', cFcnSig)

% Configure
kern.ThreadBlockSize=[512 1];
kern.GridSize=[1024 1024];

% Run
[c, d] = feval(kern, a, b);
Example:

**Corner Detection on the CPU**

```matlab
dx = cdata(2:end-1,3:end) - cdata(2:end-1,1:end-2);
dy = cdata(3:end,2:end-1) - cdata(1:end-2,2:end);
dx2 = dx.*dx;
dy2 = dy.*dy;
dxy = dx.*dy;

gaussHalfWidth = max(1, ceil(2*gaussSigma));
ssq = gaussSigma^2;
t = -gaussHalfWidth : gaussHalfWidth;
gaussianKernel1D = exp(-(t.*t)/(2*ssq))/(2*pi*ssq);
% The Gaussian 1D filter
gaussianKernel1D = gaussianKernel1D / sum(gaussianKernel1D);
smooth_dx2 = conv2( gaussianKernel1D, gaussianKernel1D, dx2, 'valid' );
smooth_dy2 = conv2( gaussianKernel1D, gaussianKernel1D, dy2, 'valid' );
smooth_dxy = conv2( gaussianKernel1D, gaussianKernel1D, dxy, 'valid' );

det = smooth_dx2 .* smooth_dy2 - smooth_dxy .* smooth_dxy;
trace = smooth_dx2 + smooth_dy2;
score = det - 0.25*edgePhobia*(trace.*trace);
```

1. Calculate derivatives
2. Smooth using convolution
3. Calculate score
Example:

Corner Detection on the GPU

```matlab
cdata = gpuArray( cdata );

dx = cdata(2:end-1,3:end) - cdata(2:end-1,1:end-2);
dy = cdata(3:end,2:end-1) - cdata(1:end-2,2:end-1);
dx2 = dx.*dx;
dy2 = dy.*dy;
dxy = dx.*dy;

gaussHalfWidth = max( 1, ceil( 2*gaussSigma ) );
ssq = gaussSigma^2;
t = -gaussHalfWidth : gaussHalfWidth;
gaussianKernel1D = exp(-(t.*t)/(2*ssq))/(2*pi*ssq);       % The Gaussian 1D filter
gaussianKernel1D = gaussianKernel1D / sum(gaussianKernel1D);
smooth_dx2 = conv2( gaussianKernel1D, gaussianKernel1D, dx2, 'valid' );
smooth_dy2 = conv2( gaussianKernel1D, gaussianKernel1D, dy2, 'valid' );
smooth_dxy = conv2( gaussianKernel1D, gaussianKernel1D, dxy, 'valid' );

det = smooth_dx2 .* smooth_dy2 - smooth_dxy .* smooth_dxy;
trace = smooth_dx2 + smooth_dy2;
score = det - 0.25*edgePhobia*(trace.*trace);
score = gather( score );
```

0. Move data to GPU

4. Bring data back
arrayfun

Can execute entire scalar programs on the GPU
  (while, if, for, break, &, &&, ...)

function [logCount,t] = mandelbrotElem( x0, y0, r2, maxIter)
  % Evaluate the Mandelbrot function for a single element
  z0 = complex( x0, y0 );
  z = z0;
  count = 0;
  while count <= maxIter && (z*conj(z) <= r2)
    z = z*z + z0;
    count = count + 1;
  end
  % . . . Etc. . . .
Summary of Options for Targeting GPUs

**Level of control**
- Minimal
- Some
- Extensive

**Parallel Options**
- Use GPU arrays with MATLAB built-in functions
- Execute custom functions on elements of the GPU array
- Create kernels from existing CUDA code and PTX files
Parallel Computing enables you to …

Larger Compute Pool

Speed up Computations

Larger Memory Pool

Work with Large Data
Programming Parallel Applications

**Level of control**

- Minimal
- Some
- Extensive

**Parallel Options**

- Support built into Toolboxes

  - High-Level Programming Constructs:
    (e.g. parfor, batch, distributed)

  - Low-Level Programming Constructs:
    (e.g. Jobs/Tasks, MPI-based)
Parallel Computing with MATLAB on CPU

MATLAB
SIMULINK
TOOLBOXES
BLOCKSETS

Worker
Worker
Worker
Worker
Worker
Worker
Worker
Worker
Worker
Parallel Support in Optimization Toolbox

- **Functions:**
  - `fmincon`  
    - Finds a constrained minimum of a function of several variables
  - `fminimax`  
    - Finds a minimax solution of a function of several variables
  - `fgoalattain`  
    - Solves the multiobjective goal attainment optimization problem

- Functions can take finite differences in parallel in order to speed the estimation of gradients
Tools with Built-in Support

- Optimization Toolbox
- Global Optimization Toolbox
- Statistics Toolbox
- SystemTest
- Simulink Design Optimization
- Bioinformatics Toolbox
- Model-Based Calibration Toolbox
- ...


Directly leverage functions in Parallel Computing Toolbox
Programming Parallel Applications

Level of control

- Minimal
- Some
- Extensive

Parallel Options

- Support built into Toolboxes
- High-Level Programming Constructs: (e.g. parfor, batch, distributed)
- Low-Level Programming Constructs: (e.g. Jobs/Tasks, MPI-based)
Running Independent Tasks or Iterations

- Ideal problem for parallel computing
- No dependencies or communications between tasks
- Examples include parameter sweeps and Monte Carlo simulations
Example: Parameter Sweep of ODEs

- Solve a 2\textsuperscript{nd} order ODE
  \[ m\ddot{x} + b \dot{x} + k \ x = 0 \]
  \[ 1,2,\ldots \quad 1,2,\ldots \]

- Simulate with different values for $b$ and $k$

- Record peak value for each run

- Plot results
Summary of Example

- Mixed task-parallel and serial code in the same function
- Ran loops on a pool of MATLAB resources
The Mechanics of `parfor` Loops

```matlab
a = zeros(10, 1);
parfor i = 1:10
    a(i) = i;
end
a
```

Pool of MATLAB Workers
Parallel Computing enables you to …

- Speed up Computations
- Work with Large Data
Client-side Distributed Arrays

Remotely Manipulate Array from Desktop

Distributed Array Lives on the Cluster
# Enhanced MATLAB Functions That Operate on Distributed Arrays

<table>
<thead>
<tr>
<th>Type of Function</th>
<th>Function Names</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data functions</td>
<td><code>cumprod</code>, <code>cumsum</code>, <code>fft</code>, <code>max</code>, <code>min</code>, <code>prod</code>, <code>sum</code></td>
</tr>
<tr>
<td>Data type functions</td>
<td><code>cast</code>, <code>cell2mat</code>, <code>cell2struct</code>, <code>celldisp</code>, <code>cellfun</code>, <code>char</code>, <code>double</code>, <code>fieldnames</code>, <code>int16</code>, <code>int32</code>, <code>int64</code>, <code>int8</code>, <code>logical</code>, <code>num2cell</code>, <code>rmfield</code>, <code>single</code>, <code>struct2cell</code>, <code>swapbytes</code>, <code>typen_cast</code>, <code>uint16</code>, <code>uint32</code>, <code>uint64</code>, <code>uint8</code></td>
</tr>
<tr>
<td>Elementary and trigonometric functions</td>
<td><code>abs</code>, <code>acos</code>, <code>acosd</code>, <code>acosh</code>, <code>acot</code>, <code>acotd</code>, <code>acoth</code>, <code>acsc</code>, <code>acscd</code>, <code>acsch</code>, <code>angle</code>, <code>asec</code>, <code>asecd</code>, <code>asech</code>, <code>asin</code>, <code>asind</code>, <code>asinh</code>, <code>atan</code>, <code>atan2</code>, <code>atand</code>, <code>atanh</code>, <code>ceil</code>, <code>complex</code>, <code>conj</code>, <code>cos</code>, <code>cosd</code>, <code>cosh</code>, <code>cot</code>, <code>cotd</code>, <code>coth</code>, <code>csc</code>, <code>cscd</code>, <code>csch</code>, <code>exp</code>, <code>expm1</code>, <code>fix</code>, <code>floor</code>, <code>hypot</code>, <code>imag</code>, <code>isreal</code>, <code>log</code>, <code>log10</code>, <code>log1p</code>, <code>log2</code>, <code>mod</code>, <code>nextpow2</code>, <code>nthroot</code>, <code>pow2</code>, <code>real</code>, <code>reallog</code>, <code>realpow</code>, <code>realsqrt</code>, <code>rem</code>, <code>round</code>, <code>sec</code>, <code>secd</code>, <code>sech</code>, <code>sign</code>, <code>sin</code>, <code>sind</code>, <code>sinh</code>, <code>sqrt</code>, <code>tan</code>, <code>tand</code>, <code>tanh</code></td>
</tr>
<tr>
<td>Elementary matrices</td>
<td><code>cat</code>, <code>diag</code>, <code>eps</code>, <code>find</code>, <code>isempty</code>, <code>isequal</code>, <code>isequalwithnan</code>, <code>isfinite</code>, <code>isinf</code>, <code>isnan</code>, <code>length</code>, <code>ndims</code>, <code>size</code>, <code>tril</code>, <code>triu</code></td>
</tr>
<tr>
<td>Matrix functions</td>
<td><code>chol</code>, <code>eig</code>, <code>lu</code>, <code>norm</code>, <code>normest</code>, <code>svd</code></td>
</tr>
<tr>
<td>Array operations</td>
<td><code>all</code>, <code>and</code>, <code>any</code>, <code>bitand</code>, <code>bitor</code>, <code>bitxor</code>, <code>ctranspose</code>, <code>end</code>, <code>eq</code>, <code>ge</code>, <code>gt</code>, <code>horzcat</code>, <code>ldivide</code>, <code>le</code>, <code>lt</code>, <code>minus</code>, <code>mldivide</code>, <code>mrdivide</code>, <code>mtimes</code>, <code>ne</code>, <code>not</code>, <code>or</code>, <code>plus</code>, <code>power</code>, <code>rdivide</code>, <code>subsasgn</code>, <code>subsindex</code>, <code>subsref</code>, <code>times</code>, <code>transpose</code>, <code>uminus</code>, <code>uplus</code>, <code>vertcat</code>, <code>xor</code></td>
</tr>
<tr>
<td>Sparse matrix functions</td>
<td><code>full</code>, <code>issparse</code>, <code>nnz</code>, <code>nonzeros</code>, <code>nmax</code>, <code>sparse</code>, <code>spfun</code>, <code>spzeros</code></td>
</tr>
<tr>
<td>Special functions</td>
<td><code>dot</code></td>
</tr>
</tbody>
</table>
Programming Parallel Applications

**Level of control**

- Minimal
- Some
- Extensive

**Parallel Options**

- **Support built into Toolboxes**
- **High-Level Programming Constructs:** (e.g. parfor, batch, distributed)
- **Low-Level Programming Constructs:** (e.g. Jobs/Tasks, MPI-based)
spmd blocks

```matlab
spmd
  \% single program across workers
end
```

- Mix parallel and serial code in the same function
- Run on a pool of MATLAB resources
- **Single Program** runs simultaneously across workers
  - Distributed arrays, message-passing
- **Multiple Data** spread across multiple workers
  - Data stays on workers
Client-side Distributed Arrays and SPMD

- Client-side distributed arrays
  - Class `distributed`
  - Can be created and manipulated directly from the client.
  - Simpler access to memory on labs
  - Client-side visualization capabilities

- `spmd`
  - Block of code executed on workers
  - Worker specific commands
  - Explicit communication between workers
  - Mixture of parallel and serial code
**MPI-Based Functions in Parallel Computing Toolbox™**

Use when a high degree of control over parallel algorithm is required

- High-level abstractions of MPI functions
  - `labSendReceive`, `labBroadcast`, and others
  - Send, receive, and broadcast any data type in MATLAB

- Automatic bookkeeping
  - Setup: communication, ranks, etc.
  - Error detection: deadlocks and miscommunications

- Pluggable
  - Use any MPI implementation that is *binary*-compatible with MPICH2
Scheduling Jobs and Tasks

MATLAB
SIMULINK
TOOLBOXES
BLOCKSETS

Scheduler

Job
Results

Worker

Task
Result

Worker

Task
Result

Worker

Task
Result

Worker
Support for Schedulers

Direct Support

Open API for others
Programming Parallel Applications

Level of control

- Minimal
- Some
- Extensive

Parallel Options

- Support built into Toolboxes
  - High-Level Programming Constructs: (e.g. parfor, batch, distributed)
  - Low-Level Programming Constructs: (e.g. Jobs/Tasks, MPI-based)
Research Engineers Advance Design of the International Linear Collider with MathWorks Tools

Challenge
Design a control system for ensuring the precise alignment of particle beams in the International Linear Collider

Solution
Use MATLAB, Simulink, Parallel Computing Toolbox, and Instrument Control Toolbox software to design, model, and simulate the accelerator and alignment control system

Results
- Simulation time reduced by an order of magnitude
- Development integrated
- Existing work leveraged

“Using Parallel Computing Toolbox, we simply deployed our simulation on a large group cluster. We saw a linear improvement in speed, and we could run 100 simulations at once. MathWorks tools have enabled us to accomplish work that was once impossible.”

Dr. Glen White
Queen Mary, University of London
Edwards Air Force Base Accelerates Flight Test Data Analysis Using MATLAB and MathWorks Parallel Computing Tools

Challenge
Accelerate performance and flying qualities flight test data analysis for unmanned reconnaissance aircraft

Solution
Use MathWorks parallel computing tools to execute MATLAB flight data processing algorithms on a 16-node computer cluster

Results
- Analysis completed 16 times faster
- Application parallelized in minutes
- Program setup time reduced from weeks to days

Parallel Computing Toolbox and MATLAB Distributed Computing Server provided a one-for-one time savings with the number of processors used. For example, with a 16-processor cluster, throughput was 16 times higher, enabling Edwards AFB engineers to accomplish in hours tasks that used to take days.

Link to user story
Argonne National Laboratory Develops Powertrain Systems Analysis Toolkit with MathWorks Tools

Challenge
Evaluate designs and technologies for hybrid and fuel cell vehicles

Solution
Use MathWorks tools to model advanced vehicle powertrains and accelerate the simulation of hundreds of vehicle configurations

Results
- Distributed simulation environment developed in one hour
- Simulation time reduced from two weeks to one day
- Simulation results validated using vehicle test data

“We developed an advanced framework and scalable powertrain components in Simulink, designed controllers with Stateflow, automated the assembly of models with MATLAB scripts, and then distributed the complex simulation runs on a computing cluster – all within a single environment.”

Sylvain Pagerit
Argonne National Laboratory

Link to user story
Supercomputing for the Masses: Killer-Apps, Parallel Mappings, Scalability and Application Lifespan

Rob Farber

Senior Scientist (PNNL) -> Visiting scientist ICHEC
Killer Apps

occur when personal vision matches technical capability to fulfill a market demand

• Graphics processors and games: killer apps that created a huge market
Technical capability

- Market forces evolved GPUs into massively parallel GPGPUs (General Purpose GPUs).

- 250+ million (1/4 billion) CUDA-enabled GPUs says it all!

- CUDA: put supercomputing in the hands of the masses.
  - December 1996, ASCI Red the first teraflop supercomputer
  - Today: kids buy GPUs with flop rates comparable to systems available to scientists with supercomputer access in the mid to late 1990s.
    - GeForce 480 1.35 TF/s peak 32-bit
    - Newegg.com: $299

Remember that Finnish kid who wrote some software to understand operating systems? Inexpensive commodity hardware enables:

- New thinking
- A large educated base of developers
A perfect storm of opportunities and technology

(Summary of Farber, Scientific Computing, “Realizing the Benefits of Affordable Teraflop-capable Hardware”)

• **Multi-threaded software is a must-have** because manufacturers were forced to move to multi-core CPUs
  o The failure of Dennard’s scaling laws meant processor manufacturers had to add cores to increase performance and entice customers

• Multi-core is disruptive to single-threaded legacy apps
  o Businesses and research efforts will not benefit from new hardware unless they invest in multi-threaded software
  o **Lack of investment risks stagnation and losing to the competition**

• Competition is fierce, the new technology is readily available and it is inexpensive!
  o **Which software and models? Look to those that are:**
    • Widely adopted and have withstood the test of time
    • Look at CUDA and the CUDA model
CUDA is not the only game in town

(but will be a focus of this talk)

- Android/Iphone - mobile is huge


Augmented Reality

Jen-Hsun with RTT at 2009 GTC
CUDA is a game changer!

- CUDA enables orders of magnitude faster apps:
  - 10x can make computational workflows more interactive (even poorly performing GPU apps are useful).
  - 100x is disruptive and has the potential to fundamentally affect scientific research by removing time-to-discovery barriers.
  - 1000x and greater achieved through the use of the NVIDIA SFU (Special Function Units) or multiple GPUs … Whooo Hoooo!

- In a few slides: examine CUDA + Graphics = Wow!
CUDA was adopted amazingly fast!

- February 2007: The initial CUDA SDK was made public.

- Now: CUDA-based GPU Computing is part of the curriculum at over 360 universities.
  - MIT, Harvard, Cambridge, Oxford, the Indian Institutes of Technology, National Taiwan University, and the Chinese Academy of Sciences.
The numbers have changed!

Redefining

What is Possible

Perfect storm of opportunities delivers fresh approaches

Rob Farber

General purpose graphics processor unit (GPGPU) technology has arrived during a perfect storm of opportunities. Multi-threaded software is now a necessity as x86 and other conventional processor designs have been forced to adopt a multi-core approach. From dual core cell phones to IBM Power 7 systems that will support well over a million concurrent threads of execution, parallelism is now the path to performance.

Legacy applications and research efforts that do not invest in multi-threaded software will not benefit from modern multi-core processors, because single-threaded and poorly scaling software will not be able to utilize extra processor cores. As a result, computational performance will plateau at or near current levels, placing the projects that depend on these legacy applications at risk of both stagnation and loss of competitiveness.

Graphic processors have matured into general purpose computational devices at exactly the right time to be considered in this industry-wide retooling to utilize multi-threaded parallelism. To put this in very concrete terms, any teenager (or research effort) from Beijing, China, to New Delhi, India, can purchase a teraflop capable graphics processor and start developing and testing massively parallel applications. Table 1 shows two inexpensive teraflop-capable offerings from AMD and NVIDIA that are available now for purchase.

These devices represent a peak floating-point capability that was beyond anything available for the most advanced high performance computing (HPC) users until Sandia National Laboratory performed a trillion floating-point operations per second in December 1996 on the ASCI Red supercomputer. I wonder how many of those proposals for leading-edge research using a teraflop supercomputer can be performed today by students anywhere in the world using a few GPGPUs in a workstation with a fast RAID disk subsystem and a decent amount of host system memory.

From personal experience, current GPGPU flop rates meet or exceed the computational capability to which I had access as a scientist in the theoretical division at Los Alamos National laboratory in the late 1990s. In addition, the machines I used were shared with other users, while current GPGPUs are inexpensive enough to be dedicated for use by a single individual. Installing four high-end GPUs in a workstation can create a machine with a peak flop rate comparable to the large MPP2 supercomputer that Pacific Northwest National Laboratory (PNNL) made available to users just a few years ago.

Competition is fierce in both commercial and academic circles, which is why commodity supercomputing in the hands of the masses is going to have a huge impact on both commercial products and scientific research. Plus, GPGPU technology has made the competition global and

Application speed says it all!
(fastest 100 apps in the NVIDIA Showcase Sept. 8, 2010)

Ranked speedup by project (best to worst)

Orders of magnitude increased performance in an extraordinary number of fields

• Spanning a wide-range of computational, data driven, and real-time applications:
  o Computational finance
  o Medical
  o Quantum chemistry simulations
  o Molecular modeling and electrostatic potentials
  o Diffusion
  o Fluid flow
  o Systems of differential equations
  o Data driven problems such as microscopy

• Many can be considered killer apps in their field.
An example: the Metropolis algorithm 300x – 1000x

- Among the ten algorithms that have had the greatest influence on the development and practice of science and engineering in the 20th century (Beichl, Sullivan, 2000).

- Plays a significant role in statistics, econometrics, physics and computing science.

  - For some applications, MCMC simulation is the only known general approach for providing a solution within a reasonable time (Diaconis, Saloff-Coste, 1995).

- CUDA version reported to be 300x to 1000x faster (Alerstam, Svensson, Engels, 2008).
Three rules for fast GPU codes

1. Get the data on the GPU (and keep it there!)
   - PCIe x16 v2.0 bus: 8 GiB/s in a single direction
   - 20-series GPUs: 140-200 GiB/s

2. Give the GPU enough work to do
   - Assume 10 µs latency and 1 TF device
   - Can waste \((10^{-6} \times 10^{12}) = 1M\) operations

3. Reuse and locate data to avoid global memory bandwidth bottlenecks
   - \(10^{12}\) flop hardware delivers \(10^{10}\) flop when global memory limited
   - Can cause a 100x slowdown!

Corollary: Avoid malloc/free!
Application lifespan
SIMD: a key from the past

Farber: general SIMD mapping from the 1980s

- Acknowledgements: Work performed at or funded by the Santa Fe Institute, the theoretical division at Los Alamos National Laboratory and various NSF, DOE and other funding sources including the Texas Advance Computer Center.

The Connection Machine

This mapping for Neural Networks …

“Most efficient implementation to date” (Singer 1990), (Thearling 1995)

Results presented at SC09 (courtesy TACC)

Observed Peak Effective Rate vs. Number of Ranger Cores

<table>
<thead>
<tr>
<th>Number of Barcelona cores</th>
<th>Effective Rate (TF/s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>60,000 cores</td>
<td>363 TF/s measured</td>
</tr>
<tr>
<td>62,796 cores</td>
<td>386 TF/s (projected)</td>
</tr>
</tbody>
</table>
The Parallel Mapping

\[ \text{energy} = \text{objFunc}(p_1, p_2, \ldots p_n) \]

Optimization Method
(Powell, Conjugate Gradient, Other)

Step 1
Broadcast parameters

Step 2
Calculate partials

Step 3
Sum partials to get energy

GPU 1
\( p_1, p_2, \ldots p_n \)
Examples
0, N-1

GPU 2
\( p_1, p_2, \ldots p_n \)
Examples
N, 2N-1

GPU 3
\( p_1, p_2, \ldots p_n \)
Examples
2N, 3N-1

GPU 4
\( p_1, p_2, \ldots p_n \)
Examples
3N, 4N-1
Principle Components Analysis (PCA)

- A widely used technique in data-mining and data reduction
  - Demonstrate a method proposed by Sanger (1989)

  - Scales according to data
  - Extends to Nonlinear PCA (NLPCA)
    - E. Oja, J. Karhunen, L. Wang, and R. Vigario, 1995
This is a general mapping
(think of your own applications!)

- Optimization
- Locally Weighted Linear Regression (LWLR)
- Neural Networks
- Naive Bayes (NB)
- Gaussian Discriminative Analysis (GDA)
- k-means
- Logistic Regression (LR)
- Independent Component Analysis (ICA)
- Expectation Maximization (EM)
- Support Vector Machine (SVM)
- Others: (MDS, Ordinal MDS, etcetera)
Results for a PCA analysis

The Connection Machine

\[ \neq \quad * \quad C_{\text{NVIDIA}} \]

(where \(C_{\text{NVIDIA}} \gg 1\))

What is \(C_{\text{NVIDIA}}\) for modern x86_64 machines?

<table>
<thead>
<tr>
<th></th>
<th>Linear PCA</th>
<th>Nonlinear PCA</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Average 100 iterations (sec)</td>
<td>Average 100 iterations (sec)</td>
</tr>
<tr>
<td>8x core*</td>
<td>0.578</td>
<td>0.4389</td>
</tr>
<tr>
<td>C2050 **</td>
<td>0.00482</td>
<td>0.0076</td>
</tr>
<tr>
<td>speedup</td>
<td>15x</td>
<td>58x</td>
</tr>
<tr>
<td>vs. 1 core</td>
<td>120x (measured)</td>
<td>425x (measured)</td>
</tr>
</tbody>
</table>

* 2x Intel (quadcore) E5540s @ 2.53 GHz, openmp, SSE enabled via g++

** includes all data transfer overhead (“Effective Flops”)
Time includes all overhead!

(effective rate or “honest flops”)

\[
\text{EffectiveRate} = \frac{\text{TotalOpCount}}{T_{\text{broadcastParam}} + T_{\text{func}} + T_{\text{reduce}}}
\]

• Memory bandwidth is key
  o More SMP cores does not translate to faster performance!
  o Previous results: single-core was faster than 1/8\(^{th}\) of an 8-core run

---

PCA

NLPCA
Scalability across GPU/CPU cluster nodes
(In collaboration with Ted Hromadka – a UCSD grad student)

TACC Longhorn GPU Scaling
(max and min over 5 runs 500/384 are within 10%)

Oak Ridge National Laboratory looks to NVIDIA “Fermi” architecture for new supercomputer
NERSC experimental GPU cluster: Dirac
EMSL experimental GPU cluster: Barracuda

Tihane-1a
7,168
NVIDIA®
Tesla™
M2050
GPUs
(4.04 MW)

Nebulæe
(星雲)
Shenzhen,
China
Looking into my crystal ball

I predict long life for GPGPU applications

- Efficient CUDA codes will stay around
  - SIMD/SPMD/MIMD mapping translate well to new architectures
  - CUDA is an excellent way to create these codes
  - Previous SIMD example is still solving important problems

Will these applications always be written in CUDA?

Data-parallel extensions are hot!
Thrust: a very good thing!
(Now a standard API in CUDA 4.0)

- Pre-cuda 4.0 download from: http://code.google.com/p/thrust/
- The primary developers of Thrust are:
  - Jared Hoberock
  - Nathan Bell
- Others acknowledged
  - Mark Harris
  - Michael Garland
  - Nadathur Satish
  - Shubho Sengupta

Example from website

```c
int main(void) {
    // generate random data on the host
    thrust::host_vector<int> h_vec(100);
    thrust::generate(h_vec.begin(), h_vec.end(), rand);

    // transfer to device and compute sum
    thrust::device_vector<int> d_vec = h_vec;
    int x = thrust::reduce(d_vec.begin(), d_vec.end(), 0, thrust::plus<int>())
    return 0;
}
```

Expect good things from Copperhead: the data-parallel Python project!
int main() {
    const int N=1000000;

    // task 1: create the array
    thrust::device_vector<int> a(N);

    // task 2: fill the array
    thrust::sequence(a.begin(), a.end(), 0);

    // task 3: calculate the sum of the array
    int sumA = thrust::reduce(a.begin(), a.end(), 0);

    // task 4: calculate the sum of 0 .. N-1
    int sumCheck=0;
    for(int i=0; i < N; i++) sumCheck += i;

    // task 5: check the results agree
    if(sumA == sumCheck) cout << "Test Succeeded!" << endl;
    else { cerr << "Test FAILED!" << endl; return(1); }

    return(0);
}
Async kernel execution allows task-parallelism

```cpp
int main()
{
    const int N=1000000;

    // task 1: create the array
    thrust::device_vector<int> a(N);

    // task 2: fill the array
    thrust::sequence(a.begin(), a.end(), 0);

    // task 4: calculate the sum of 0 .. N-1
    int sumCheck=0;
    for(int i=0; i < N; i++) sumCheck += i;

    // task 3: calculate the sum of the array
    int sumA= thrust::reduce(a.begin(),a.end(), 0);

    // task 5: check the results agree
    if(sumA == sumCheck) cout << "Test Succeeded!" << endl;
    else { cerr << "Test FAILED!" << endl; return(1);}

    return(0);
}
```
Multi-GPU and Hybrid for more parallelism

Multi-GPU scale by $N$ GPUs
Really Exciting! Hybrid Codes

• Magma (Matrix Algebra on GPU and Multicore Architectures)
  o “A dense linear algebra library similar to LAPACK but for heterogeneous/hybrid architectures, starting with current "Multicore+GPU" systems.” [http://icl.cs.utk.edu/magma/](http://icl.cs.utk.edu/magma/)

• The MAGMA team has made the conclusion that dense linear algebra methods are now a better fit on GPU architectures instead of traditional multicore architectures
  o (Nath, Stanimire, & Dongerra, 2010).

• MAGMA BLAS libraries up to 838 Gflop/s
  o 33% occupancy and 2 thread blocks per SM (Volkov, 2010).
Returning to Thrust: CUDA made simple

- Most of the actual code from an example that scales to 500 GPUs and delivers 100-times speedup over a single-core (32-bit) Xeon core.

```cpp
...  
FcnOfInterest objFcn(input);

energy = thrust::transform_reduce(
    thrust::counting_iterator<int>(0),
    thrust::counting_iterator<int>(nExamples),
    objFcn,
    0.0f,
    thrust::plus<Real>())
```
An ANN neural network functor

CalcError( const Real* _examples, const Real* _p,
            const int _nInput, const int _exLen)
  : examples(_examples), p(_p), nInput(_nInput), exLen(_exLen) {}

__device__ __host__
Real operator()(unsigned int tid)
{
    const register Real* in = &examples[tid * exLen];
    register int index=0;
    register Real h1 = p[index++];
    register Real o = p[index++];

    h1 += in[0] * p[index++];
    h1 += in[1] * p[index++];
    h1 = G(h1);

    o += in[0] * p[index++];
    o += in[1] * p[index++];
    o += h1 * p[index++];

    // calculate the square of the diffs
    o -= in[nInput];
    return o * o;
};


Thrust for both host and GPU

- Can specify **nvcc** command-line options ... **SLOW!**
- Use OpenMP

```c
Real objFunc(Real *p) {
    if(nExamples == 0) { cerr << "data not set" << endl; exit(1); }

    double startTime=omp_get_wtime();
    Real sum = 0.;
    CalcError getError(&h_data[0], p, nInput, exLen);

    #pragma omp parallel for reduction(+ : sum)
    for(int i=0; i < nExamples; ++i) {
        Real d = getError(i);
        sum += d;
    }

    objFuncCallTime += (omp_get_wtime() - startTime);
    objFuncCallCount++;
    return(sum);
}
```
## Speedup over a quad core

<table>
<thead>
<tr>
<th>OS</th>
<th>Machine</th>
<th>Opt method</th>
<th>Precision</th>
<th>Ave obj func time</th>
<th>% func time</th>
<th>Speedup over quad-core</th>
<th>Speedup over single-core</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>NVIDIA C2070</td>
<td>Nelder-Mead</td>
<td>32</td>
<td>0.00532</td>
<td>100.0</td>
<td>85</td>
<td>341</td>
</tr>
<tr>
<td>Win7</td>
<td>NVIDIA C2070</td>
<td>Nelder-Mead</td>
<td>32</td>
<td>0.00566</td>
<td>100.0</td>
<td>81</td>
<td>323</td>
</tr>
<tr>
<td>Linux</td>
<td>NVIDIA GTX280</td>
<td>Nelder-Mead</td>
<td>32</td>
<td>0.01109</td>
<td>99.2</td>
<td>41</td>
<td>163</td>
</tr>
<tr>
<td>Linux</td>
<td>NVIDIA C2070</td>
<td>Nelder-Mead</td>
<td>64</td>
<td>0.01364</td>
<td>100.0</td>
<td>40</td>
<td>158</td>
</tr>
<tr>
<td>Win7</td>
<td>NVIDIA C2070</td>
<td>Nelder-Mead</td>
<td>64</td>
<td>0.01612</td>
<td>100.0</td>
<td>22</td>
<td>87</td>
</tr>
<tr>
<td>Linux</td>
<td>NVIDIA C2070</td>
<td>Levenberg-Marquardt</td>
<td>32</td>
<td>0.04313</td>
<td>2.7</td>
<td>10</td>
<td>38</td>
</tr>
<tr>
<td>Linux</td>
<td>NVIDIA C2070</td>
<td>Levenberg-Marquardt</td>
<td>64</td>
<td>0.08480</td>
<td>4.4</td>
<td>6</td>
<td>23</td>
</tr>
<tr>
<td>Linux</td>
<td>Intel e5630</td>
<td>Levenberg-Marquardt</td>
<td>32</td>
<td>0.41512</td>
<td>21.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux</td>
<td>Intel e5630</td>
<td>Levenberg-Marquardt</td>
<td>64</td>
<td>0.49745</td>
<td>20.8</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux</td>
<td>Intel e5630</td>
<td>Nelder-Mead</td>
<td>32</td>
<td>0.45312</td>
<td>100.0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Linux</td>
<td>Intel e5630</td>
<td>Nelder-Mead</td>
<td>64</td>
<td>0.53872</td>
<td>100.0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
PCA/NLPCA with Nelder-Mead Optimization

PCA

NLPCA
Parallel Nsight with NVTX library
(10M examples)
1M Examples

Transform Reduce


__tools_Flush  __tools_Flush

cudaMalloc  functor  cudaFree
10k examples
Moral of the story: Avoid allocations!
(tough to do in C++ and Thrust)

Write your own reduction

<table>
<thead>
<tr>
<th>Data size</th>
<th>Speedup over thrust</th>
</tr>
</thead>
<tbody>
<tr>
<td>10M</td>
<td>2%</td>
</tr>
<tr>
<td>100k</td>
<td>14%</td>
</tr>
<tr>
<td>10k</td>
<td>24%</td>
</tr>
</tbody>
</table>

Parallel Nsight is the only tool that showed the bottleneck
## cudaprof: The Visual Profiler

<table>
<thead>
<tr>
<th>Method</th>
<th>#Calls</th>
<th>GPU time (us)</th>
<th>CPU time (us)</th>
<th>%GPU time</th>
<th>glob mem read throughput</th>
<th>glob mem write throughput</th>
<th>IPC</th>
<th>l1 gld hit rate %</th>
</tr>
</thead>
<tbody>
<tr>
<td>launch_closure_by_value-2</td>
<td>58</td>
<td>49919.4</td>
<td>50240.4</td>
<td>83.86</td>
<td>10.6109</td>
<td>0.10275</td>
<td>1.77138</td>
<td>94.4175</td>
</tr>
<tr>
<td>launch_closure_by_value-3</td>
<td>57</td>
<td>325.248</td>
<td>572.552</td>
<td>0.54</td>
<td>33.4966</td>
<td>12.1854</td>
<td>0.658475</td>
<td>0</td>
</tr>
<tr>
<td>launch_closure_by_value-0</td>
<td>1</td>
<td>5.312</td>
<td>8</td>
<td>0</td>
<td>27.3012</td>
<td>0.0783133</td>
<td>0.240059</td>
<td>0</td>
</tr>
<tr>
<td>launch_closure_by_value-1</td>
<td>1</td>
<td>2.144</td>
<td>6</td>
<td>0</td>
<td>64.3582</td>
<td>0.134328</td>
<td>0.541946</td>
<td>0</td>
</tr>
<tr>
<td>memcpyHtoD</td>
<td>117</td>
<td>1897.09</td>
<td>2406</td>
<td>3.18</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memcpyDtoD</td>
<td>2</td>
<td>181.792</td>
<td>208.744</td>
<td>0.3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>memcpyDtoH</td>
<td>57</td>
<td>88.896</td>
<td>48717</td>
<td>0.14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Cudaprof: automated analysis

Analysis for kernel launch_closure_by_value-2 on device Tesla C2070

Summary profiling information for the kernel:
Number of calls: 58
Minimum GPU time(us): 794.37
Maximum GPU time(us): 951.78
Average GPU time(us): 860.68
GPU time (%): 83.86
Grid size: [14 1 1]
Block size: [960 1 1]

Limiting Factor
Achieved Instruction Per Byte Ratio: 45.92 ( Balanced Instruction Per Byte Ratio: 3.58 )
Achieved Occupancy: 0.63 ( Theoretical Occupancy: 0.62 )
IPC: 1.77 ( Maximum IPC: 2 )
Achieved global memory throughput: 10.71 ( Peak global memory throughput(GB/s): 143.42 )

Hint(s)
The achieved instructions per byte ratio for the kernel is greater than the balanced instruction per byte ratio for the device. Hence, the kernel is likely compute bound. For details, click on Instruction Throughput Analysis.
Massive GPU hardware parallelism achieved via replication of streaming multiprocessors
Programming a GPU is programming multiple Streaming Multiprocessors

- CUDA requires an execution configuration `Kernal<<<nBlk,nThreadPerBlk>>>()`
- The gigathread scheduler allocates one or more block per thread.
- The SM schedules warps and resources
- Highly scalable as the gigathread scheduler only needs to know when an SM is busy.
Thread level parallelism

- Provide as many warps (groups of threads) as possible.
- Warps execute in SIMD fashion
- SM scheduler detects when SIMD instruction is ready to run (no dependency)

- Lots of warps implies a good chance that a SIMD instruction will be ready to run ... hides latency!
Instruction Level parallelism (ILP)

- TLP is wasteful: for(int i=0; i<n; i++) ...
  - Uses 512 registers when threads per block are 512
  - Only 64 when threads per block are 64

- Registers are valuable
  - Only memory fast enough for peak performance
  - (Volkov 2010) Register 8 TB/s; shared mem 1.3 TB/s: Global mem 140 GB/s

<table>
<thead>
<tr>
<th></th>
<th>GF100</th>
<th>GF200</th>
</tr>
</thead>
<tbody>
<tr>
<td>20 at 100% occupancy</td>
<td>63 at 33% occupancy</td>
<td>3x more registers per thread</td>
</tr>
<tr>
<td>16 at 100% occupancy</td>
<td>≈ 128 at 12.5% occupancy</td>
<td>8x more registers per thread</td>
</tr>
</tbody>
</table>
Multiple independent operations per thread

```c
#ifdef ILP4
#pragma unroll 16
for(int i=0; i < NUM_ITERATIONS; i++) {
    a = a * b + c;     d = d * b + c;     e = e * b + c;     f = f * b + c;
}
#else
#pragma unroll 16
for(int i=0; i < NUM_ITERATIONS; i++) {
    a = a * b + c;
}
#endif
```
More registers per thread and greater efficiency
**Instruction Level parallelism (ILP)**

- ILP exploits SM warp scheduling
  - Dual execution paths in compute 2.0 devices
  - SFU (transcendental functions) scheduled separately
- Compute 2.1 superscalar operation

**Use Little’s law**

\[
\text{Needed parallelism} = \text{Latency} \times \text{Throughput}
\]
ILP now part of the standard libraries

<table>
<thead>
<tr>
<th>Large SGEMM</th>
<th>CUBLAS 1.1</th>
<th>CUBLAS 2.0</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Threads per block</td>
<td>512</td>
<td>64</td>
<td>8x smaller thread blocks</td>
</tr>
<tr>
<td>Occupancy (Compute 1.0)</td>
<td>67%</td>
<td>33%</td>
<td>2x lower occupancy</td>
</tr>
<tr>
<td>Performance (Compute 1.0)</td>
<td>128 Gflop/s</td>
<td>204 Gflop/s</td>
<td>1.6x faster performance</td>
</tr>
</tbody>
</table>

| Threads per block | 256 | 64         | 4x smaller thread blocks                   |
| Occupancy (Compute 1.0) | 33% | 17%        | 2x lower occupancy                        |
| Performance (Compute 1.0) | 45 GFlop/s | 93 Gflop/s | 2x faster performance                     |
cuobjdump & PTX kernels

code for sm_20
Function: __Z6kernelff
/*0000*/ /*0x00005de428004404*/ MOV R1, c [0x1] [0x100];
/*0008*/ /*0x80001de428004000*/ MOV R0, c [0x0] [0x20];
/*0010*/ /*0xfc009de428000000*/ MOV R2, RZ;
/*0018*/ /*0x9000dee428004000*/ MOV R3, c [0x0] [0x24];
/*0020*/ /*0x40209c034000c000*/ IADD R2, R2, 0x10;
/*0028*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0030*/ /*0x0421dc231a8e4000*/ ISETP.NE.AND P0, pt, R2, c [0x10] [0x0], pt;
/*0038*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0040*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0048*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0050*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0058*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0060*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0068*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0070*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0078*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0080*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0088*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0090*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*0098*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*00a0*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*00a8*/ /*0xa0301c0030008000*/ FFMA R0, R3, R0, c [0x0] [0x28];
/*00b0*/ /*0xc000e5e428000600*/ @PO BRA 0x18:
/*00b8*/ /*0x84009c042c0c00000*/ S2R R2, SR_Tid_X;
/*00c0*/ /*0x00000d0e428001800*/ MOV R3, c [0xe] [0x0];
/*00c8*/ /*0x10211c03207c000*/ IMAD.U32.U32 R4.CC, R2, 0x4, R3;
/*00d0*/ /*0x10209c0435000000*/ IMUL.U32.U32.HI R2, R2, 0x4;
/*00d8*/ /*0x10215c0434007800*/ IADD.X R5, R2, c [0xe] [0x4];
/*00e0*/ /*0x00401c8590000000*/ ST.E [R4]. R0;
/*00e8*/ /*0x000001de800000000*/ EXIT;

/*
 * PTX is equivalent to the following kernel:
 *
 * __global__ void myKernel(int *data)
 * {
 *     int tid = blockIdx.x * blockDim.x + threadIdx.x;
 *     data[tid] = tid;
 * }
 */

char myPtx[] = "
.version 1.4
.target sm_10, map_f64_to_f32
.entry _Z8myKernelPi (.param .u64 __cudaparm__Z8myKernelPi_data)
{
    .reg .u16 %rh<4>;
    .reg .u32 %r<5>;
    .reg .u64 %rd<6>;
    cvt.u32.u16 %r1, %tid.x;
    mov.u16 %r1, %ctaid.x;
    mov.u16 %r2, %ntid.x;
    mul.wide.u16 %r2, %r1, %r2;
    add.u32 %r3, %r1, %r2;
    ld.param.u64 %rd1, __cudaparm__Z8myKernelPi_data;
    cvt.s64.s32 %rd2, %r3;
    mul.wide.s32 %rd3, %r3, 4;
    add.u64 %rd4, %rd1, %rd3;
    st.global.s32 [%rd4+0], %r3;
    exit;
}
";
Ocelot
CUDA + Graphics

(a potent combination!)

- **Primitive restart**: define an index value to be used as a tag that tells OpenGL that the next vertex starts a new OpenGL primitive of the same type
  
  - Keep the data on the GPU
  - Avoids the PCIe bottleneck
  - Variable length data works great!
  - A feature of OpenGL 3.1

- Output only: visualization, rendering and games
- Combine with vision recognition: Augmented Reality!
A primitive restart virtual world example

- Perlin Noise

  Important for games and the movie industry: Ken Perlin won an Academy Award for this noise generator

- Primitive restart can be 100 FPS faster than other rendering methods and delivers higher quality images

<table>
<thead>
<tr>
<th>Card/OS</th>
<th>Observed FPS</th>
<th>Rough Average</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GTX 280/Linux</td>
<td>550-590</td>
<td>560</td>
</tr>
<tr>
<td>C2050/Linux</td>
<td>2720 - 2740</td>
<td>2730</td>
</tr>
</tbody>
</table>

Farber DDJ Part 18
Primitive Restart generates better quality images (June 15th NIVIDA Webinar)

- Define an index to specify “restart” of graphics primitive

| Line1(x1,y1,z1) | Line1(x2,y2,z2) | 1000 | Line2(x1,y1,z1) | Line2(x2,y2,z2) | Line2(x3,y3,z3) |

- Rendering performance can be optimized by arranging the indices to achieve the highest reuse in the texture units.
- Higher quality images can be created by alternating the direction of tessellation
  - Old
    ![Old Tessellation Diagram]
  - New
    ![New Tessellation Diagram]
Parallel Nsight shows the speed

- Primitive restart: around 60 µs.
- Multidraw: around 3,900 µs.
- Iteratively drawing each triangle fan: approximately 1,100,000 µs.

Generate a 512x512 heightmap using Perlin noise

Farber DDJ Part 20
Spend your time swapping buffers
Avoid the PCI bus

<table>
<thead>
<tr>
<th>Seconds</th>
<th>2.3308109</th>
<th>2.3408109</th>
<th>2.3508109</th>
<th>2.3608109</th>
<th>2.3708109</th>
<th>2.3808109</th>
<th>2.3908109</th>
<th>2.4008109</th>
<th>2.4108109</th>
<th>2.4208109</th>
<th>2.4308109</th>
<th>2.4408109</th>
<th>2.4508109</th>
</tr>
</thead>
<tbody>
<tr>
<td>Thread State</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function Calls</td>
<td>Swap...</td>
<td>SwapB...</td>
<td>SwapBu...</td>
<td>SwapBu...</td>
<td>SwapBu...</td>
<td>SwapBu...</td>
<td>SwapBu...</td>
<td>SwapBu...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Tools Extensi...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Level 1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 0.0% [3...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread State</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 0.0% [3...</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CUDA</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context 0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Context 1 [0]</td>
<td>Driver API</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Memory</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Compute</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Farber DDJ Part 20

Host — GPU
If you remember one thing from this talk
(Three rules for fast GPU codes)

1. **Get the data on the GPU (and keep it there!)**
   - PCIe x16 v2.0 bus: 8 GiB/s in a single direction
   - 20-series GPUs: 140-200 GiB/s

2. **Give the GPU enough work to do**
   - Assume 10 µs latency and 1 TF device
   - Can waste \((10^{-6} \times 10^{12}) = 1M\) operations

3. **Reuse and locate data to avoid global memory bandwidth bottlenecks**
   - 10^{12} flop hardware delivers 10^{10} flop when global memory limited
   - Can cause a 100x slowdown!

*Corollary: Avoid malloc/free!*
Predicting future killer apps?

Humility: five years ago I would not have believed:

- Adding four PCIe devices will give my workstation roughly the same peak flop rate as the largest PNNL supercomputer.

- It is now possible to get the full 3D wiring diagram for the entire brain of a cat or mouse.
Killer apps: when personal vision meets technical capability

- The Connectome project: A Galilean first opportunity for scientists to examine the detailed schematic diagram that nature uses for vision and cognition.

- SC09: computers can simulate an entire cat brain
  - "The cat is out of the bag: cortical simulations with $10^9$ neurons, $10^{13}$ synapses", Ananthanarayanan, Esser, Simon, and Modha (2009).

- My prediction: combining detailed brain models with sufficient computational capability will be a killer app.
  - People studied birds and (eventually) created supersonic aircraft
  - With nature’s wiring diagram for vision & language, (eventually) …?

WHAT IS YOUR VISION?