Multi-core Parallel Computing Using OpenMP

October 30, 2015
Quote of the Day

“Books are useless! I only ever read one book, To Kill A Mockingbird, and it gave me absolutely no insight on how to kill mockingbirds!”

-- Homer Simpson, safety inspector
[ more widely known since 1989 ]
Before We Get Started

- Issues covered last time:
  - CUDA streams [wrap up]
  - CUDA Unified Memory

- Today’s topics
  - CUDA libraries
  - Multi-core parallel computing w/ OpenMP – get started

- Assignment:
  - HW07 - due on Wd, Nov. 4 at 11:59 PM
CUDA Libraries...
CUDA Libraries

- Math, Numerics, Statistics
- Dense & Sparse Linear Algebra
- Algorithms (sort, etc.)
- Image Processing
- Signal Processing
- Finance

- In addition to these widely adopted libraries, several less established ones available in the community

cuBLAS: Dense linear algebra on GPUs

- Complete BLAS implementation plus useful extensions
  - Supports all 152 standard routines for single, double, complex, and double complex
  - Levels 1, 2, and 3 BLAS
Speedups Compared to Multi-threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
cuSPARSE: Sparse linear algebra routines

- Sparse matrix-vector multiplication & triangular solve
  - APIs optimized for iterative methods

- New features in 4.1:
  - Tri-diagonal solver with speedups up to 10x over Intel MKL
  - ELL-HYB format offers 2x faster matrix-vector multiplication

\[
\begin{bmatrix}
  y_1 \\
  y_2 \\
  y_3 \\
  y_4 \\
\end{bmatrix} = \alpha \begin{bmatrix}
  2 & & & -1 \\
  4 & -1 & & \\
  5 & 9 & 1 & \\
 -1 & & 8 & \\
\end{bmatrix} \begin{bmatrix}
  -1 \\
  2 \\
  1 \\
  3 \\
\end{bmatrix} + \beta \begin{bmatrix}
  2 \\
  0 \\
 -1 \\
  2 \\
\end{bmatrix}
\]
cuSolver

- Matrix factorization, eigenvalue solvers
- Matrix refactoring, sparse and dense LAPACK routines

- Based on cuBLAS and cuSPARSE libraries
cuFFT: Multi-dimensional FFTs

- Algorithms based on Cooley-Tukey and Bluestein
- Simple interface, similar to FFTW
- Streamed asynchronous execution
- 1D, 2D, 3D transforms of complex and real data
- Double precision (DP) transforms
- 1D transform sizes up to 128 million elements
- Batch execution for doing multiple transforms
- In-place and out-of-place transforms
Speedups Compared to Multi-Threaded CPU Execution

- CUDA 4.1 on Tesla M2090, ECC on
- MKL 10.2.3, TYAN FT72-B7015 Xeon x5680 Six-Core @ 3.33 GHz
cuRAND: Random Number Generation

- Pseudo- and Quasi-RNGs
  - Supports several output distributions
  - Statistical test results reported in documentation

- New RNGs in CUDA 4.1:
  - MRG32k3a RNG
  - MTGP11213 Mersenne Twister RNG
NPP: NVIDIA Performance Primitives

- Arithmetic, Logic, Conversions, Filters, Statistics, Signal Processing, etc.
- This is where GPU computing shines
- 1,000+ new image primitives in 4.1
Development, Debugging, and Deployment Tools

[Rounding Up the CUDA Ecosystem]
Programming Languages & APIs

- HMPP Compiler
- Python for CUDA
- NVIDIA C Compiler
- CUDA Fortran
- OpenCL
- NVIDIA CUDA
- OpenGL
- Microsoft DirectX
- Microsoft AMP C/C++
- PGI Accelerator
Debugging Tools

- NVIDIA Parallel Nsight for Visual Studio
- NVIDIA CUDA-MEMCHECK for Linux & Mac
- Allinea DDT with CUDA Distributed Debugging Tool
- NVIDIA CUDA-GDB for Linux & Mac
- TotalView for CUDA for Linux Clusters
MPI & CUDA Support

Platform MPI

Announced beta at SC2011

As of OFED 1.5.2

GPUDirect™

InfiniBand

MVAPICH

Announced pre-release at SC2011

Platform MPI

Announced beta at SC2011

NVIDIA [C. Woolley]
Multicore Parallel Computing with OpenMP
On Computing Speeds

- 1961:
  - One would have to combine 17 million IBM-1620 computers to reach 1 Mflops
  - At $64K apiece, when adjusted for inflation this would be ½ the 2015 US national debt

- 2000:
  - About $1,000

- 2015 (January):
  - $0.08
OpenMP: Target Hardware

- CUDA: targeted parallelism on the GPU

- OpenMP: targets parallelism on SMP architectures
  - Handy when
    - You have a machine that has 16 cores
    - You have a good amount of shared memory, say 64 GB

- MPI: targeted parallelism on a cluster (distributed computing)
  - Note that MPI implementation can handle transparently an SMP architecture such as a workstation with two hexcore CPUs that draw on a good amount of shared memory
Feature Length on a Chip: Moore’s Law at Work

- 2013 – 22 nm
- 2015 – 14 nm
- 2017 – 10 nm
- 2019 – 7 nm
- 2021 – 5 nm
- 2023 – ??? (carbon nanotubes, black phosphorous?)
What Does This Mean?

- One of two things:
  - You either increase the computational power and/or smarts of the chip since you have more transistors, or
  - You can keep the number of transistors constant but decrease the size of the chip
Increasing the Number of Transistors: Multicore is Here to Stay

- What does that buy you?
- More computational units

October 2015:
- Intel Xeon w/ 18 cores – 5.7 billion transistors (E7-8890 v3, $7200)
- Intel Xeon Phi: about 60 lesser cores

Projecting ahead (not going to happen, “dark silicon” phenomenon):
- 2017: about 36 cores
- 2019: about 60 cores
- 2021: about 100 cores
Decreasing the Area of the Chip

- Decreasing the chip size: imagine that you want to pack the power of today’s 12 core chip on tomorrow’s wafer

- Size of chip – assume a square of length “L”
  - 2013: L is about 20 mm
  - 2015: L ≈ 14 mm
  - 2017: L ≈ 10 mm
  - 2019: L ≈ 7 mm
  - 2021: L ≈ 5 mm → a fifth of an inch fits on your phone
Dennard’s Scaling: Making Moore’s Law Useful

- Dennard scaling: dictates how the voltage/current, frequency should change in response to our etching ability

- Voltage lower and lower, static power losses more prevalent

- Transistors are too close
  - Many of them per unit area

- Amount of power dissipated grows to high levels
  - Thermal runaway
  - Computing not reliable anymore: what’s noise and what’s information?
Power Issues: The Dark Side of the Story

- Dark Silicon: transistors that manufacturers cannot afford to turn on

- Too many transistors on a chip – cannot afford to power them lest the chip melts

Illustrating Hardware

- Intel Haswell
  - Released in June 2013
  - 22 nm technology
  - Transistor budget: 1.4 billions
    - Tri-gate, 3D transistors
  - Typically comes in four cores
  - Has an integrated GPU
  - Deep pipeline – 16 stages
  - Sophisticated infrastructure for ILP acceleration
  - Superscalar
  - Supports HTT (hyper-threading technology)

Good source of information for these slides: http://www.realworldtech.com/
Illustrating Hardware

- Actual layout of the chip

![Image of chip layout]

- Schematic of the chip organization

- LLC: last level cache (L3)

- Three clocks:
  - A core’s clock ticks at 2.7 to 3.0 GHz but adjustable up to 3.7-3.9 GHz
  - Graphics processor ticking at 400 MHz but adjustable up to 1.3 GHz
  - Ring bus and shared L3 cache - a frequency that is close to but not necessarily identical to that of the cores
Caches

- **Data:**
  - L1 – 32 KB per *core*
  - L2 – 512 KB or 1024 KB per *core*
  - L3 – 8 MB per *CPU*

- **Instruction:**
  - L0 – room for about 1500 microoperations (uops) per core
    - See H/S primer, online
  - L1 – 32 KB per core

- **Cache is a black hole for transistors**
  - Example: 8 MB of L3 translates into:
    - $8 \times 1024 \times 1024 \times 8$ (bits) * 6 (transistors per bit, SRAM) = 402 million transistors out of 1.4 billions
Fermi Specifics

- There are two schedulers that issue warps of “ready-to-go” threads.
- One warp issued at each clock cycle by each scheduler.
- During no cycle can more than 2 warps be dispatched for execution on the four functional units.
- Scoreboarding is used to figure out which warp is ready.
Haswell Microarchitecture
[30,000 Feet]

- Microarchitecture components:
  - Instruction pre-fetch support (purple)
  - Instruction decoding support (orange)
    - CISC into uops
      - Turning CISC to RISC
  - Instruction Scheduling support (yellowish)
  - Instruction execution
    - Arithmetic (blue)
    - Memory related (green)

- More details - see primer posted online:
  http://sbel.wisc.edu/Courses/ME964/Literature/primerHW-SWinterface.pdf
Haswell Microarchitecture: The HTT Feature

- Employs the so-called Hyper-Threading Technology (HTT) on each core of the chip

- Somewhat similar to what NVIDIA does with the SM
  - The SM is overcommitted – there are up to 64 warps waiting to execute to hide latencies

- HTT: two threads active at any given time and the scheduler tries to issue instructions from both
  - The HW upgrades to support HTT are relatively minor
    - Required to save execution state of a thread when it’s idle and not running for lack of data of lack of functional units available
A processor core to maintain two architectural states, each of which can support its own thread.

Many of the internal microarchitectural hardware resources are shared between the two threads.
Essentially, one physical core shows up as two virtual cores.

Why is it good?
- More functional units end up being used at any given time.
- Instruction dependencies are reduced since instructions executed belong to different threads.

When one thread has a cache miss, branch mispredict, or any other pipeline stall, the other thread continues processing instructions at nearly the same rate as a single thread running on the core.
Moving from HW to SW
Acknowledgements

- Majority of slides used for discussing OpenMP issues are from Intel’s library of presentations for promoting OpenMP
  - Slides used herein with permission

- Credit given where due: IOMPP
  - IOMPP stands for “Intel OpenMP Presentation”
OpenMP and Symmetric Multi-Processing

- Threads have access to a pool of shared memory
- Threads can have private data
  - Not accessible by other threads
- Data transfer/access transparent to programmer
- Synchronization is implicit but can be made explicit as well
Data vs. Task Parallelism

- **Data parallelism**
  - You have a large amount of data elements and each data element 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
  - Example, task parallelism:
    - Prepare a soup, make a salad, bake a cake, microwave popcorn

- **NOTE**: A task might actually be “execute this instruction” → what ILP is
  - In example above, all tasks used different “functional units”
Objectives

- Understand OpenMP at the level where you can
  - Implement data parallelism
  - Implement task parallelism
Work Plan: What We’ll Cover

● What is OpenMP?
  Parallel regions
  Work sharing
  Data environment
  Synchronization

● Advanced topics
OpenMP: What’s Reasonable to Expect

- If you have more than 16 cores or so in an SMP setup, it’s pretty unlikely that you can get a speed-up on that scale. All sorts of overheads kick in to slow you down
  - Beyond 16: law of diminishing return

- Some reasons: no overcommitment of HW, false cache sharing, etc.

- A word on lack of overcommitment
  - Recall the trick that helped the GPU hide latency
    - Overcommitting an SM and hoping to hide 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
- Very good at coarse-grained parallelism
- Combines serial and parallel code in single source
- Standardizes ~ 25 years of compiler-directed threading experience

- Current spec is OpenMP 4.0
  - Released in 2013
  - [http://www.openmp.org](http://www.openmp.org)
  - More than 300 Pages
OpenMP Programming Model

- **Master thread** spawns a **team of threads** as needed
  - Managed transparently on your behalf
  - It relies on low-level thread fork/join methodology to implement parallelism
    - The developer is spared the details

- Leveraging OpenMP in an existing code: Parallelism is added incrementally: that is, the sequential program evolves into a parallel program

![Diagram showing OpenMP programming model](image)
OpenMP: Library Support

- Runtime environment routines:
  - Modify/check the number of threads
    
    ```c
    omp_[set|get]_num_threads()
    omp_get_thread_num()
    omp_get_max_threads()
    ```
  - Are we in a parallel region?
    ```c
    omp_in_parallel()
    ```
  - How many processors in the system?
    ```c
    omp_get_num_procs()
    ```
  - Explicit locks
    ```c
    omp_[set|unset]_lock()
    ```
  - Many more...

https://computing.llnl.gov/tutorials/openMP/
A Few Syntax Details to Get Started

● Picking up the API - header file in C, or Fortran 90 module
  ```
  #include "omp.h"
  use omp_lib
  ```

● Most OpenMP constructs 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]...]
    ```
Why Compiler Directive and/or Pragmas?

- One of OpenMP’s design principles: the same code, with no modifications, can run either on an one core machine or a multiple core machine.

- Therefore, you have to “hide” all the compiler directives behind Comments and/or Pragmas.

- These directives 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
OpenMP, Compiling Using the Command Line

- Method depends on compiler

- GCC:
  
  ```
  $ g++ -o integrate_omp integrate_omp.c -fopenmp
  ```

- ICC:
  
  ```
  $ icc -o integrate_omp integrate_omp.c -openmp
  ```

- MSVC (not in the express edition):
  
  ```
  $ cl /openmp integrate_omp.c
  ```
OpenMP Odds and Ends...

- **Controlling the number of threads**
  - The default number of threads that a program uses when it runs is the number of processors on the machine
  - For the C Shell: `setenv OMP_NUM_THREADS number`
  - For the Bash Shell: `export OMP_NUM_THREADS=number`

- **Timing:**
  ```c
  #include <omp.h>
  stime = omp_get_wtime();
  mylongfunction();
  etime = omp_get_wtime();
  total=etime-stime;
  ```
Work Plan

- What is OpenMP?
  - Parallel regions
  - Work sharing
  - Data environment
  - Synchronization

- Advanced topics
Parallel Region & Structured Blocks (C/C++)

- Most OpenMP constructs apply to **structured blocks**
  - **structured block**, definition: a block with one point of entry at the top and one point of exit at the bottom
  - The only “branches” allowed are `exit()` function calls

### A structured block

```c
#pragma omp parallel
{
    int id = omp_get_thread_num();

    more: res[id] = do_big_job (id);

    if ( not_conv(res[id]) ) goto more;
}
printf ("All done\n");
```

### Not a structured block

```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;
}
done: if (!really_done()) goto more;
```

There is an implicit barrier at the right “}” curly brace and that’s the point at which the other worker threads complete execution and either go to sleep or spin or otherwise idle.
Example: Hello World

```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("All done here...\n");
}
```

Here’s my laptop:
Intel Core i5-3210M @ 2.50GHz 3 MB L3 Cache, TDP 35 Watts, Two-Core Four-Thread Processors