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

- 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 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
- 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

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`)

```cpp
// 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 * N * 1 threads
    int numBlocks = 1;
    dim3 threadsPerBlock(N, N);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
}
```
Something to think about…

- 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 times 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

\(n\).

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

```
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 cudaMemcpyDeviceToHost are symbolic constants

```c
cudaMemcpy(Md.elements, M.elements, size, cudaMemcpyHostToDevice);
cudaMemcpy(M.elements, Md.elements, size, cudaMemcpyDeviceToHost);
```
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`)