ME964
High Performance Computing for Engineering Applications

Using CMake & Building on Euler
GPU Computing: Execution Configuration in CUDA
CUDA API

February 14, 2012

“In theory there is no difference between theory and practice. In practice there is.”
Yogi Berra
Before We Get Started…

- **Last time**
  - Started GPU computing segment of the class
    - GPU hardware
    - GPGPU computing (General Purpose GPU) and the role of CUDA
    - Discussed first example that invoked a kernel launched on the GPU
    - GPU kernels are executed asynchronously

- **Today**
  - Quick overview, CMake and
    - How to use it to build an executable that runs on the CPU & GPU
  - Discuss about execution configuration on the GPU
  - Comments on CUDA API

- **Assignments**
  - HW 3 due on Th, at 11:59 PM
CMake
~ A Build System for Build Systems ~
Role of Build Systems

- Find dependencies
  - “Where is MPI? CUDA? What flags enable OpenMP?”

- Generate/compile/link code
  - Generate via preprocessors (“macro magic”)
    - #define, Yacc, Bison, Protobuf, …
  - Compile code
  - Link with libraries, generate executables

- Test & install
  - Run unit tests
  - Generate installer?
A Few Build Systems...

- (hand-written) makefiles
  - Portability depends on author

- Autotools (GNU build system)
  - Most familiar: `./configure && make && make install`
  - But there’s more: aclocal, autoheader, automake, autoconf,…
  - Require Cygwin or MSYS for Windows

- Eclipse, Visual Studio
  - Tied to IDE
  - Complex setup for large projects
...and Two More

- **SCons**
  - Builds defined as Python scripts
  - Used by Blender, Doom3, NumPy, SciPy

- **CMake**
  - Can generate Eclipse projects, Visual Studio solutions, Makefiles, Xcode projects, etc.
  - What we’ll use for the rest of ME964
Intro to CMake

- Projects are defined in simple text (txt) files
  - No more digging through stacks of config dialogs
  - Easy to diff
  - Easy to maintain under revision control (Mercurial, SVN, etc.)

- User-configurable options set in the ccmake/cmake-gui programs

- Once configured, project files are generated for your system’s native build environment
CMake Lingo

- CMakeLists.txt
  - Text file in which you define the bulk of the project

- Generator
  - Converts CMakeLists.txt to a project file for your IDE

- Cache
  - Stores environment-specific and user-configurable options

- Build type
  - Set of compiler/linker options
  - Some predefined setups:
    - debug, release, release with debug symbols, space-optimized release
CMake Workflow

- Write CMakeLists.txt
- Select build directory in ccmake/cmake-gui
- Choose generator for your environment
  - Eclipse, Visual Studio, Makefiles, etc
- Configure project options, saved in cache
- Generate project files
- Build
CMakeLists.txt

- Defines the entire project and build process

- Watch out: name must be exactly CMakeLists.txt

- Contents themselves are case insensitive
  - But **be consistent**
  - Commonly found in recent projects:
    - functions()
    - VARIABLES

- 20/80 rule: 20% of commands do 80% of what you’ll need

- Documentation (CMake 2.8):
  [http://cmake.org/cmake/help/cmake-2-8-docs.html](http://cmake.org/cmake/help/cmake-2-8-docs.html)
CMakeLists.txt: A Few Other Functions

- configure_file: do find/replace on files
- ExternalProject: require an external project to be built before building your own
- find_package(foo): see if package foo is available on this system
  - This makes setting up CUDA and MPI relatively painless
  - But, FindFoo.cmake script must already be written
- math: perform arbitrary math operations
- {add,remove}_definitions: set/remove preprocessor definitions
# Set the required version of CMake
cmake_minimum_required(VERSION 2.8)

# Set your project title
project(ME964)

# Look for CUDA and set up the build environment
# Flag ‘REQUIRED’ forces us to set up CUDA correctly before building
find_package("CUDA" REQUIRED)

# Finally, we would like to create a program ‘foo’
# from the files ‘foo.cu’ and ‘bar.cu’
# Using cuda_add_executable tells CMake to use with nvcc instead of gcc
cuda_add_executable(foo foo.cu bar.cu)
CMake for ME964

- A template is in **hg**, under **CMakeTemplate**
  - Has macros for CUDA, MPI, and OpenMP projects
  - To use:
    - Copy to your source directory
    - Uncomment relevant sections of CMakeLists.txt
    - Modify for your assignments

- Useful command: **add_subdirectory**
  - Allows you to have a single main CMakeLists.txt with assignment-specific ones in subdirs
# Minimum version of CMake required. Don't touch.
cmake_minimum_required(VERSION 2.8)

# Set the name of your project
project(ME964)

# Include macros from the SBEL utils library
include(SBELUtils.cmake)

## Example CUDA program
enable_cuda_support()
if(LIBCUTIL AND LIBSHRUTIL)
    cuda_add_executable(bandwidthTest bandwidthTest.cu)
target_link_libraries(bandwidthTest ${LIBCUTIL} ${LIBSHRUTIL})
else()
    message(FATAL_ERROR "Could not find libcutil or libshrutil. Please check CUDA_SDK_LIB_DIR and verify that these libraries have been built."
endif()
What This Shows...

- Including commands from another file
- Running a macro (no arguments)
- Checking if two libraries have been found
  - Actually, checking if those two variables are defined
- Adding a CUDA executable to build
- Linking the executable to two libraries
- Outputting messages
- SBELUtils.cmake has more, see comments
User-configurable options are set here
  Similar to running ~/.configure

Set source and build directories
  Must decide between in-source v. out-of-source build

New build dir/cleared cache: nothing there
  Hit ‘Configure’ to select generator & start configuring

New/changed options are shown in red
  Modify if need be, then keep hitting configure until done

‘Generate’ creates the project files

Feel free to venture into ‘Advanced’ options
  Can manually set compiler/linker options here
  Remember this: do a “File > Delete Cache” if something gets messed up
cmlake-gui gotchas

- If you need a library/path/variable, make sure it is found
  - Will show up as `{FOO}_NOT_FOUND` in the config options
  - Can be manually set if need be
    - But you should probably first determine why it’s not being done automatically

- Option not showing up? Hit Configure again, check advanced

- Strange issues? Clear the cache
  - Same as make `distclean`
In-source v. Out-of-source Builds

- **In-source builds**
  - Binaries & project files generated alongside source code
  - Need to pay attention if using version control
  - IDEs (Eclipse) prefer this method
    - See [http://www.cmake.org/Wiki/Eclipse_CDT4_Generator](http://www.cmake.org/Wiki/Eclipse_CDT4_Generator)

- **Out-of-source builds**
  - Binaries & project files in separate directory
  - Easy to clean – just delete it
  - Only need to `checkin/commit` the source directory
Using Projects, Compiling

- After generating the project files, open in your IDE
  - Eclipse: File > Import Project
  - Visual Studio: open the solution
  - Makefiles/Eclipse: make (make -j for parallel build)

- Source code should be in there, even if using out-of-source (linked to the source directory)

- **CMake** will automatically run when building to update project/make files
  - No need to open cmake-gui again unless changing options
  - Visual Studio may ask to reload the project; do it
CMake Ecosystem

- **CPack**
  - Generates package files for installation
    - RPM, deb, tgz, Windows NSIS, OS X packages

- **CTest**
  - Unit testing framework
  - Generates a 'make test' target
  - Similar to Metronome (but not quite as automated/integrated)

- **CDash**
  - Dashboard for showing build & testing results

- **Our lab (SBEL) in the process of moving to this build/run/test environment**
Job Submission w/ Torque
Torque

- Euler uses Torque to manage jobs
  - Maybe Condor this summer?

- Detailed guide will come later
  - For now, enough to do the homework

- See the Torque documentation at:
  http://www.adaptivecomputing.com/resources/docs/torque/3-0-3/2.1jobsubmission.php
Job Submission

- Two modes: batch and interactive

- Batch:
  - Compute tasks written as shell script, with Torque/PBS-specific comments

- Interactive:
  - Given an interactive shell on a compute node
Job Submission - batch

**bandwidthTest.sh**

```bash
#!/bin/bash

#PBS –n bandwidthTest
#PBS –l nodes=1:gpus=1
#PBS –d $PBS_O_WORKDIR

./bandwidthTest
```

Shell script  Name of job  Resource selection  Set working directory

Run!

Submit with:

```
$ qsub bandwidthTest.sh
```

Output placed in bandwidthTest.{o,e}[0-9]*
Job Submission - Interactive

me964@euler $ qsub -l -l nodes=1:gpus=1

me964@node $ ./bandwidthTest

“qsub -Eye -ell blah”
Resource Selection

- Follows –l (-ell)
- One node with one GPU
  - nodes=1:gpus=1
- Two nodes with one GPU/node
  - nodes=2:gpus=1
- Two nodes with three processors/node
  - Nodes=2:ppn=3

- Note: must request GPUs for GPU jobs
End Build Tools/Approaches
Go Back to CUDA
Calling a Kernel Function, Details

- A kernel function must be called with an execution configuration:

  ```
  __global__ void kernelFoo(...); // declaration
  
dim3 DimGrid(100, 50);  // 5000 thread blocks
  dim3 DimBlock(4, 8, 8); // 256 threads per block
  
  kernelFoo<<< DimGrid, DimBlock >>>(...your arg list comes here...);
  ```

- Any call to a kernel function is asynchronous
  - By default, execution on host doesn’t wait for kernel to finish
The host call below instructs the GPU to execute the function (kernel) “foo” using 25,600 threads.
- Two arguments are passed down to each thread executing the kernel “foo”

```c
foo<<<100,256>>>(pMyMatrixD, pMyVecD)
```

- In this execution configuration, the host instructs the device that it is supposed to run 100 blocks each having 256 threads in it.
- The concept of block is important, since it represents the entity that gets executed by an SM (stream multiprocessor).
More on the Execution Model
[Some Constraints]

- There is a limitation on the number of blocks in a grid:
  - The grid of blocks can be organized as a 3D structure: max of 65535 by 65535 by 65535 grid of blocks (about 280,000 billion blocks)

- Threads in each block:
  - The threads can be organized as a 3D structure (x,y,z)
  - The total number of threads in each block cannot be larger than 1024
# CUDA, Simple Example

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

__global__ void simpleKernel(int* data)
{
    // write something trivial to the global memory...
    data[threadIdx.x] = blockIdx.x + threadIdx.x;
}

int main()
{
    int hostArray[4], *devArray;
    // allocate memory on the device (GPU)
    cudaMalloc((void**)&devArray, sizeof(int)*4);

    // invoke GPU kernel, with one block that has four threads
    simpleKernel<<<1,4>>>(devArray);

    // bring the result back from the GPU into the hostArray
    cudaMemcpy(&hostArray, devArray, sizeof(int)*4, cudaMemcpyDeviceToHost);

    // print out the result to confirm that things are looking good
    std::cout << "Values stored in hostArray: ", ";
    std::cout << hostArray[0] << ", ";
    std::cout << hostArray[1] << ", ";
    std::cout << hostArray[2] << ", ";
    std::cout << hostArray[3] << std::endl;

    // release the memory allocated on the GPU
    cudaFree(devArray);

    return 0;
}
```
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

Exercise:
- How was the grid defined for this pic?
  - I.e., how many blocks in X and Y directions?
- How was a block defined in this pic?
A Couple of Built-In Variables
[Critical in supporting the SIMD paradigm]

- 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
  - blockDim (dim3) – contains the dimension of the grid
  - [warpSize (uint) – provides warp size, we’ll talk about this later…]
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 3D layout?
  - Simplifies memory addressing when processing multidimensional data
    - Handling matrices
    - Solving PDEs on subdomains
    - …

Courtesy: NVIDIA
Thread Index vs. Thread ID
[critical in understanding how SIMD is supported in CUDA & understanding the concept of “warp”]

- 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 \((x, y, z)\), called the thread index, is a high-level representation of a thread in the economy of a block. Under the hood, the same thread has a simplified and unique id, which is computed as \( t_{id} = x + y \times D_x + z \times D_x \times D_y \). You can regard this as a ”projection” to a 1D representation. The concept of thread id is important in understanding how threads are grouped together in warps (more on ”warps” later).

- 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 PDEs in 3D you might want to have all three block dimensions nonzero.
Example: Adding Two Matrices

- You have two matrices A and B of dimension $N \times N$ (N=32)
- You want to compute $C = A + B$ in parallel
- Code provided below (some details omitted, such as `#define N 32`)

```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);
}
```
Exercise

- 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...

  \[
  C[j][i] = A[j][i] + B[j][i]
  \]
Combining Threads and Blocks

- Recall that there is a limit on the number of threads you can organize in a block.

- In the vast majority of applications you need to use many blocks, each containing the same number of threads.

- Example: your assignment, when adding the two large vectors.
Indexing Arrays with Blocks and Threads
[important to grasp]

- No longer as simple as using only threadIdx.x
  - Consider indexing into an array, one thread accessing one element
  - Assume you have $M=8$ threads/block and the array has 32 entries

```
threadIdx.x threadIdx.x threadIdx.x threadIdx.x
0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
```

- With $M$ threads/block a unique index for each thread is given by:

```
int index = threadIdx.x + blockIdx.x * M;
```

- NOTE: Identical to finding the offset in 1-dimensional storage of a 2-dimensional matrix

&NVIDIA
Example: Indexing Arrays

- Which thread will operate on the red element?

```
int index = threadIdx.x + blockIdx.x * M;
= 5 + 2 * 8;
= 21;
```
Vector Addition

[with Blocks and Threads: relevant in your assignment]

- Use the built-in variable `blockDim.x` for threads per block
  - This basically gives you the value of $M$ on previous slide
    
    ```
    int index = threadIdx.x + blockIdx.x * blockDim.x;
    ```

- When it comes to launching the kernel, you’ll have to compute how many blocks you have to deal with:

  ```
  add<<<N/THREADS_PER_BLOCK, THREADS_PER_BLOCK>>>(d_a, d_b, d_c);
  ```

- How would you deal with a vector whose length $N$ is not a multiple of the number of threads $M$ in a block?

  ```
  add<<<(N + M-1) / M, M>>>(d_a, d_b, d_c, N)
  ```
Timing Your Application
[useful for your assignment]

- Timing support – part of the CUDA 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

- Reports time in milliseconds with resolution of about 0.5 microseconds
  - 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.
#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);

    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;
}