CUDA

Jacob Lauzon, Ryan McLaughlin
Agenda

1. Background
2. Overview
3. Basic GPU architecture
4. CUDA Programming Model
5. CUDA Example, Modeling the heart
Background

Computationally expensive problems illustrate the need for parallel processing.

Some of the key solutions today:
CPU: MPI
GPU: CUDA, OpenCL
Background cont.

(Note that now, General Purpose GPU ~ GPU)
Before OpenGL, Accelerators (GPUs) could only perform image operations.

1992 - OpenGL brought shaders to programmers, but required extensive knowledge of the GPUs, the shader usage, and the implementation language

1999 - nVIDIA GeForce 256 is “world’s first GPU” and first to let developers control computations, albeit only polygons and shaders using shader language.

2006 - CUDA unveiled, after absorbing the efforts of Ian Buck (Brooke programming model - 2003 ) to bring GPU programming to C language.
Overview, What is CUDA?

- **CUDA** - Compute Unified Device Architecture
- Parallel computing platform created by NVIDIA
- Released June of 2007
- Implemented on nVIDIA Graphic Processing Units (GPUs)
- CUDA allows for GPUs to be used as a general purpose processor, this is known as GPGPU
Overview - GPUs

- GPUs have architectures that promote many “slower” concurrent threads rather than a single very fast thread
- Optimized for data-parallel computations with very high throughput
- Can tolerate higher memory latency than CPUs
- Hardware focused heavily on computation
nVIDIA GPU Architecture

- Made up of two main components:
  - Global Memory
    - Like RAM for a CPU
    - Shared among all cores of the GPU and the CPU
    - Usually 1 - 12 GBs per GPU
    - High bandwidth
  - Streaming Multiprocessors (SMs) (SMX on Kepler architecture)
    - Where the computation is performed
    - Contains their own control units, registers, execution pipelines, and caches
    - Contains "CUDA cores"/Streaming Processors (SIMD) which contain logic units and computation units, including units dedicated to IEEE 754-2008 Floating Point arithmetic
    - Also contains Warp Schedulers which schedule groups of 32 parallel threads
Architecture cont. (Kepler)
Memory System (Kepler)

- Each SMX has the following:
  - 64 KB of configurable memory which can be split between L1 cache and shared memory
  - 48 KB of Read-Only cache (Kepler only)

- Each GPU has 1536 KB of L2 cache
  - Shared among all SMs and is primary point to share data between the SMs

- Each GPU also has its main system memory
  - Much like RAM to a CPU
Processing Flow

1. Copy data from CPU memory (main memory) to GPU memory
2. CPU instructs the GPU on how to process
3. Parallel processing on GPU
4. Results are copied from GPU memory to CPU memory
Programming Model

- CUDA libraries exist for most programming languages. (C, C++, Java, Python…)
- Keywords exist for each language that map to various CUDA “commands”
- CPU acts as “host”
- CUDA functions are created to run on the GPU but sequential code still runs on CPU
- Parallel portion of application is called a **kernel**
Kernels

- The parallel portion of the application that is to be executed by the entire GPU
- Kernels are divided into many threads that execute the same code
- Each thread has an ID (much like MPI)
- Threads are grouped into blocks and blocks are grouped into a grid
- The grid is what gets executed by the GPU
Kernel Execution
Communication and Synchronization

- Since blocks of threads run on a single SM, they can communicate through the shared memory through cooperative loads/stores and can use barrier synchronization.
- Thread blocks are completely independent can execute sequentially or concurrently and in any order.
  - This allows for scalability to any number of SMs.
Advantages

- Massively Parallel - compute many independent values simultaneously
- Highly Specialized to nVIDIA - Achieves higher efficiencies than OpenCL
- System is commercially available for less than a many processor system
- Easily add more ‘systems’ by adding GPGPU cards
- Multiple warps allow latency & ALU delay to be hidden
- Shared memory access for warps
- Full support for integer and floating point operations
Limitations

- Limited to only nVIDIA cards
- High specialization requires intricate details OpenCL doesn’t need
- Suffers greatly in thread divergence - if statements/sequential
- No built in exception handling - thread divergence is too severe
- Substantial performance degradation when communicating with host
- Memory is not paged like host memory, so is quickly limited.
  - Can use host paged memory, but with large performance loss
Example - Modeling the Heart

In a similar fashion the Neuron, the Action Potential (differential voltage in and outside a cell) may be modeled mathematically.

The voltage is controlled by various voltages, diffusions, gates, and ion concentrations.
Fox Model

States/Variables such as currents, concentrations, gate variables.

Sometimes such models are referred to by the number of current variables - i.e. the Fox model is a 14 current model, while the Mitchell model is only 2 currents.

These models may be used to represent 1D, 2D, 3D simulations.
State Variables

Every state variable for each cell must be stored. For a 2D simulation, there must be a 2D grid for each state variable, with 1 for each cell. A 60 variable model would need to store 60 Height x Width arrays.

Internally, this data must be precise (double) while it may be decreased for space optimizations when examining a given iteration for observation (output). These states may be saved in order to resume from the prior iteration.

Because the memory size only changes with number and size of states, the program is limited by the GPU’s memory and the number/size of states, but not the time being simulated.
Processing

Computation Step:
For a given coordinate, update the state based on the prior states and update equations. This may include integrating, differentiating, multiplication, division, etc. The equations are specified by the model (Fox)

Diffusion Step:
In the second step, neighbors diffuse voltages to the given coordinate’s voltage. This could be 5 point, 9 point, etc.

Afterwards, the algorithm repeats, utilizing the new state values.
Implementation Details (CUDA C)

Number & distribution (in blocks) of threads is calculated based on problem size. Each of the state variable grids are allocated both on the host and the GPU. The state variables are then initialized on the host, and copied to the GPU. The kernel, denoted using `<<<>>>` notation, operates by using the grid/block details and x, y, z coordinates to calculate & store the appropriate value. Afterwards, the data is copied back to the host, before deallocating on GPU. Along the way, various low accuracy outputs are copied back and stored as short integers for checking/observation. The Compute/Diffusion steps are called iteratively from host for each time step. Boundary conditions - mirroring method (could use wrap around).
Problems & Solutions

Keeping all data on hand is most convenient, but limits the simulation size.

Data size
Data Comparison/Checking/Visualization
If statements discouraged -> special cases, boundary conditions
Diffusion step requires accessing multiple addresses per cell, every iteration
Visualization Examples (3D)

See 3D PDF example
Results

From the results on hand, using the same simulation settings:
For 1 Million Computation Nodes
  GPU: 400 seconds/simulated second
  Sequential: 27000 seconds/simulated second
  Speedup= 67.5
Questions?
References

http://www.cc.gatech.edu/~vetter/keeneland/tutorial-2011-04-14/02-cuda-overview.pdf
Cuda by Example, Jason Sanders & Edward Kandrot