Intro to GPU’s for Parallel Computing

Goals for Rest of Course

• Learn how to program massively parallel processors and achieve
  – high performance
  – functionality and maintainability
  – scalability across future generations
• Acquire technical knowledge required to achieve the above goals
  – principles and patterns of parallel programming
  – processor architecture features and constraints
  – programming API, tools and techniques
• Overview of architecture first, then introduce architecture as we go
Equipment

• Your own, if CUDA-enabled; will use CUDA SDK in C
  – Compute Unified Device Architecture
  – NVIDIA G80 or newer
  – G80 emulator won’t quite work
• Lab machine – uaa-csetesla.duckdns.org
  – Ubuntu
  – two Intel Xeon E5-2609 @2.4Ghz, each four cores
  – 128 Gb memory
  – Two nVidia Quadro 4000’s
    • 256 CUDA Cores
    • 1 Ghz Clock
    • 2 Gb memory

Why Massively Parallel Processors

• A quiet revolution and potential build-up
  – 2006 Calculation: 367 GFLOPS vs. 32 GFLOPS
  – G80 Memory Bandwidth: 86.4 GB/s vs. 8.4 GB/s
  – Until recently, programmed through graphics API

  – GPU in every PC and workstation – massive volume and potential impact

![Graph showing GFLOPS over time](image.png)
CPUs and GPUs have fundamentally different design philosophies

Architecture of a CUDA-capable GPU

32 SM’s each with 8 SP’s on one Quadro 4000
GT200 Characteristics

- 1 TFLOPS peak performance (25-50 times of current high-end microprocessors)
- 265 GFLOPS sustained for apps such as Visual Molecular Dynamics (VMD)
- Massively parallel, 128 cores, 90W
- Massively threaded, sustains 1000s of threads per app
- 30-100 times speedup over high-end microprocessors on scientific and media applications: medical imaging, molecular dynamics

“I think they're right on the money, but the huge performance differential (currently 3 GPUs ~ 300 SGI Altix Itanium2s) will invite close scrutiny so I have to be careful what I say publicly until I triple check those numbers.”

- John Stone, VMD group, Physics UIUC

Future Apps Reflect a Concurrent World

- Exciting applications in future mass computing market have been traditionally considered “supercomputing applications”
  - Molecular dynamics simulation, Video and audio coding and manipulation, 3D imaging and visualization, Consumer game physics, and virtual reality products
  - These “Super-apps” represent and model physical, concurrent world
- Various granularities of parallelism exist, but...
  - programming model must not hinder parallel implementation
  - data delivery needs careful management
Sample of Previous GPU Projects

<table>
<thead>
<tr>
<th>Application</th>
<th>Description</th>
<th>Source</th>
<th>Kernel</th>
<th>% time</th>
</tr>
</thead>
<tbody>
<tr>
<td>H.264</td>
<td>SPEC ’06 version, change in guess vector</td>
<td>34,811</td>
<td>194</td>
<td>35%</td>
</tr>
<tr>
<td>LBM</td>
<td>SPEC ’06 version, change to single precision and print fewer reports</td>
<td>1,481</td>
<td>285</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>RC5-72</td>
<td>Distributed.net RC5-72 challenge client code</td>
<td>1,979</td>
<td>218</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>FEM</td>
<td>Finite element modeling, simulation of 3D graded materials</td>
<td>1,874</td>
<td>146</td>
<td>99%</td>
</tr>
<tr>
<td>RPES</td>
<td>Rye Polynomial Equation Solver, quantum chem, 2-electron repulsion</td>
<td>1,104</td>
<td>281</td>
<td>99%</td>
</tr>
<tr>
<td>PNS</td>
<td>Petri Net simulation of a distributed system</td>
<td>322</td>
<td>160</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>SAXPY</td>
<td>Single-precision implementation of saxpy, used in Linpack’s Gaussian elim. routine</td>
<td>952</td>
<td>31</td>
<td>&gt;99%</td>
</tr>
<tr>
<td>TRACF</td>
<td>Two Point Angular Correlation Function</td>
<td>536</td>
<td>98</td>
<td>96%</td>
</tr>
<tr>
<td>FDTD</td>
<td>Finite-Difference Time Domain analysis of 2D electromagnetic wave propagation</td>
<td>1,365</td>
<td>93</td>
<td>16%</td>
</tr>
<tr>
<td>MRI-Q</td>
<td>Computing a matrix Q, a scanner’s configuration in MRI reconstruction</td>
<td>490</td>
<td>33</td>
<td>&gt;99%</td>
</tr>
</tbody>
</table>

Speedup of Applications

- GeForce 8800 GTX vs. 2.2GHz Opteron 248
- 10× speedup in a kernel is typical, as long as the kernel can occupy enough parallel threads
- 25× to 400× speedup if the function’s data requirements and control flow suit the GPU and the application is optimized
GPU History
CUDA

Graphics Pipeline Elements

1. A scene description: vertices, triangles, colors, lighting
2. Transformations that map the scene to a camera viewpoint
3. “Effects”: texturing, shadow mapping, lighting calculations
4. Rasterizing: converting geometry into pixels
5. Pixel processing: depth tests, stencil tests, and other per-pixel operations.
A Fixed Function GPU Pipeline

Texture Mapping Example

Texture mapping example: painting a world map texture image onto a globe object.
Anti-Aliasing Example

Triangle Geometry

Aliased

Anti-Aliased

Programmable Vertex and Pixel Processors

An example of separate vertex processor and fragment processor in a programmable graphics pipeline

3/21/2017
GeForce 8800 GPU

• 2006 – Mapped the separate programmable graphics stages to an array of unified processors
  – Logical graphics pipeline visits processors three times with fixed-function graphics logic between visits
  – Load balancing possible; different rendering algorithms present different loads among the programmable stages
    • Dynamically allocated from unified processors

• Functionality of vertex and pixel shaders identical to the programmer
  – geometry shader to process all vertices of a primitive instead of vertices in isolation

Unified Graphics Pipeline GeForce 8800
What is (Historical) GPGPU?

- General Purpose computation using GPU and graphics API 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
  - Game effects (FX) physics, image processing
  - Physical modeling, computational engineering, matrix algebra, convolution, correlation, sorting

Previous 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
  - Scatter a[i] = p
Tesla GPU

- NVIDIA developed a more general purpose GPU
- Can programming it like a regular processor
- Must **explicitly** declare the data parallel parts of the workload
  - Shader processors → fully programming processors with instruction memory, cache, sequencing logic
  - Memory load/store instructions with random byte addressing capability
  - Parallel programming model primitives; threads, barrier synchronization, atomic operations

CUDA

- “Compute Unified Device Architecture”
- General purpose programming model
  - User kicks off batches of threads on the GPU
  - GPU = dedicated super-threaded, massively data parallel co-processor
- 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
  - Data sharing with OpenGL buffer objects
  - Guaranteed maximum download & readback speeds
  - Explicit GPU memory management
Parallel Computing on a GPU

- 8-series GPUs deliver 25 to 200+ GFLOPS on compiled parallel C applications
  - Available in laptops, desktops, and clusters

- GPU parallelism is doubling every year
- Programming model scales transparently

- Programmable in C with CUDA tools
- Multithreaded SPMD model uses application data parallelism and thread parallelism

Overview

- CUDA programming model – basic concepts and data types

- CUDA application programming interface - basic

- Simple examples to illustrate basic concepts and functionalities

- Performance features will be covered later
CUDA – C with no shader limitations!

- Integrated host+device app C program
  - Serial or modestly parallel parts in **host** C code
  - Highly parallel parts in **device** SPMD/SIMT kernel C code

```
Serial Code (host)

Parallel Kernel (device)
KernelA<<< nBlk, nTid >>>(args);

Serial Code (host)

Parallel Kernel (device)
KernelB<<< nBlk, nTid >>>(args);
```

CUDA Devices and Threads

- A compute **device**
  - Is a coprocessor to the CPU or **host**
  - Has its own DRAM (**device memory**)
  - Runs many **threads in parallel**
  - Is typically a **GPU** but can also be another type of parallel processing **device**

- Data-parallel portions of an application are expressed as device **kernels** which run 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
G80 CUDA mode – A Device Example

- Processors execute computing threads
- New operating mode/HW interface for computing

Extended C

- **Type Qualifiers**
  - `global`, `device`, `shared`, `local`, `host`
  ```
  __device__ float filter[N];
  __global__ void convolve (float *image) {
      __shared__ float region[M];
      ...
      region[threadIdx] = image[i];
      ...
      __syncthreads()
      ...
      image[j] = result;
  }
  ```

- **Keywords**
  - `threadIdx`, `blockIdx`
  ```
  region[threadIdx] = image[i];
  ```

- **Intrinsics**
  - `__syncthreads`
  ```
  __syncthreads()
  ```

- **Runtime API**
  - Memory, symbol, execution management
  ```
  // Allocate GPU memory
  void *myimage = cudaMalloc(bytes)
  ```

- **Function launch**
  ```
  // 100 blocks, 10 threads per block
  convolve<<<100, 10>>>(myimage);
  ```
CUDA Platform

CUDA Platform
Arrays of Parallel Threads

• A CUDA kernel is executed by an array of threads
  – All threads run the same code (SPMD)
  – Each thread has an ID that it uses to compute memory addresses and make control decisions

Thread Blocks: Scalable Cooperation

• Divide monolithic thread array into multiple blocks
  – Threads within a block cooperate via shared memory, atomic operations and barrier synchronization
  – Threads in different blocks cannot cooperate
  – Up to 65535 blocks, 512 threads/block
Block IDs and Thread IDs

- We launch a “grid” of “blocks” of “threads”
- Each thread uses IDs to decide what data to work on
  - Block ID: 1D, 2D, or 3D
    - Usually 1D or 2D
  - Thread ID: 1D, 2D, or 3D
- Simplifies memory addressing when processing multidimensional data
  - Image processing
  - Solving PDEs on volumes
  - ...

CUDA Memory Model Overview

- Global memory
  - Main means of communicating R/W Data between host and device
  - Contents visible to all threads
  - Long latency access
- We will focus on global memory for now
  - Constant and texture memory will come 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

CUDA Device Memory Allocation (cont.)

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

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

cudaMalloc((void**)&Md, size);
cudaFree(Md);
```
CUDA Host-Device Data Transfer

- cudaMemcpy()
  - memory data transfer
  - Requires four parameters
    - Pointer to destination
    - Pointer to source
    - Number of bytes copied
    - Type of transfer
      - Host to Host
      - Host to Device
      - Device to Host
      - Device to Device
  - Non-blocking/asynchronous transfer

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

\[
\text{cudaMemcpy(Md, M, size, cudaMemcpyHostToDevice);}
\]
\[
\text{cudaMemcpy(M, Md, size, cudaMemcpyDeviceToDevice);}
\]
CUDA Keywords

CUDA Function Declarations

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

- `__global__` defines a kernel function
  - Must return `void`
CUDA Function Declarations (cont.)

- **__device__*/ functions cannot have their address taken
- For functions executed on the device:
  - No recursion
  - No static variable declarations inside the function
  - No variable number of arguments

Calling a Kernel Function – Thread Creation

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

  ```
  __global__ void KernelFunc(...);
  dim3 DimGrid(100, 50);  // 5000 thread blocks
  dim3 DimBlock(4, 8, 8); // 256 threads per block
  size_t SharedMemBytes = 64; // 64 bytes of shared memory
  KernelFunc<<< DimGrid, DimBlock, SharedMemBytes >>>(...);
  ```

- Any call to a kernel function is asynchronous from CUDA 1.0 on, explicit synch needed for blocking
Next Time

• Code example