High-Performance Computing Using GPUs

Luca Caucci
caucci@email.arizona.edu

Center for Gamma-Ray Imaging

The University of Arizona

November 7, 2012
Why GPUs? What is CUDA?
The CUDA programming model
Anatomy of a CUDA program
An application in medical imaging
Conclusions
Why GPUs? What is CUDA?

- GPU stands for “graphics processing unit”
- A GPU is a specialized computing device that offloads and accelerates graphics rendering from the CPU
- GPUs are very efficient at manipulating computer graphics and they are highly parallel (hundreds of cores)
- Originally designed for the entertainment industry (video coding, 3D rendering, games, . . . ), they have become suited for general-purpose complex algorithms as well
### Why GPUs? What is CUDA?

<table>
<thead>
<tr>
<th>GPU Card</th>
<th>Number Cores</th>
<th>Total Memory</th>
<th>Single Prec. GFLOP/s</th>
<th>Price</th>
</tr>
</thead>
<tbody>
<tr>
<td>GeForce GTX 480</td>
<td>480</td>
<td>1.5 GB</td>
<td>1344</td>
<td>$200</td>
</tr>
<tr>
<td>GeForce GTX 580</td>
<td>512</td>
<td>3 GB</td>
<td>1581</td>
<td>$530</td>
</tr>
<tr>
<td>GeForce GTX 690</td>
<td>3072</td>
<td>4 GB</td>
<td>5622</td>
<td>$1000</td>
</tr>
<tr>
<td>Tesla C2075</td>
<td>448</td>
<td>6 GB</td>
<td>1288</td>
<td>$2000</td>
</tr>
<tr>
<td>Tesla K10</td>
<td>3072</td>
<td>8 GB</td>
<td>5340</td>
<td>$3400</td>
</tr>
<tr>
<td>Tesla K20</td>
<td>— To Be Announced —</td>
<td>— —</td>
<td>— —</td>
<td>— —</td>
</tr>
</tbody>
</table>
Why GPUs? What is CUDA?
Why GPUs? What is CUDA?

http://www.nvidia.com/
Why GPUs? What is CUDA?

Theoretical GB/s

http://www.nvidia.com/
To satisfy the need of a convenient way to develop code for execution on a GPU device, NVIDIA developed CUDA.

CUDA stands for “compute unified device architecture”

CUDA gives developers access to the instruction set and memory of the parallel computational elements of GPUs.

CUDA is a minimal extension to C/C++

Many threads run in parallel slowly (rather than executing a single thread very fast, as on a CPU)

CUDA SDK includes CUFFT, CUBLAS libraries, sparse matrices, random numbers, ...
Why GPUs? What is CUDA?

**GPU:**
- Threads are extremely lightweight
- Very little creation/scheduling overhead
- Threads scheduled by the hardware
- 100’s or 1000’s threads for full efficiency

**CPU:**
- Usually, OS schedules threads
- Multi-core CPUs need only a few threads

http://www.nvidia.com/
Data-parallel portions of an application are executed as *kernels*, which run in parallel on many *threads*.

Threads are organized in a hierarchy of *grids* of thread *blocks*.

Blocks can have up to 3 dimensions and contain up to 1024 threads. Threads in the same block can share data via shared memory.

Grids can have up to 2 dimensions and $65535 \times 65535$ blocks. No communication between blocks.
The CUDA programming model

Thread

Block

Grid
In CUDA, threads execute on a physically separated *device*, that operates as a coprocessor to the *host*.

Both the host and the device have separate memory spaces, called *host memory* and *device memory*.

Device memory includes *shared*, *global*, *constant*, *texture*, ... memories.

A thread can only access device memory.

Calls to the *CUDA runtime* (a library) allow manage device memory, and data transfer between host and device memory.
Value type qualifiers:
- **__device__**: declares a variable that resides in the device memory
- **__host__**: declares a variable that resides in the host memory (default)
- **__shared__**: declares a variable that resides in the shared memory space of a thread block
- **__constant__**: declares a variable that resides in constant memory space on the device

Function type qualifiers:
- **__global__**: runs on device, called from host code
- **__device__**: runs on device, called from device code
- **__host__**: runs on host, called from host code (default)
### The CUDA programming model

<table>
<thead>
<tr>
<th>Memory</th>
<th>Location</th>
<th>Cached?</th>
<th>Access</th>
<th>Scope</th>
<th>Lifetime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Local</td>
<td>Off-chip</td>
<td>No</td>
<td>R/W</td>
<td>One thread</td>
<td>Thread</td>
</tr>
<tr>
<td>Shared</td>
<td>On-chip</td>
<td>N/A</td>
<td>R/W</td>
<td>All threads in block</td>
<td>Block</td>
</tr>
<tr>
<td>Global</td>
<td>Off-chip</td>
<td>No</td>
<td>R/W</td>
<td>All threads and host</td>
<td>Application</td>
</tr>
<tr>
<td>Constant</td>
<td>Off-chip</td>
<td>Yes</td>
<td>R</td>
<td>All threads and host</td>
<td>Application</td>
</tr>
<tr>
<td>Texture</td>
<td>Off-chip</td>
<td>Yes</td>
<td>R</td>
<td>All threads and host</td>
<td>Application</td>
</tr>
</tbody>
</table>

- Use shared memory to improve performance (trade-off: 16 or 48 KB of shared memory per block)
- Global memory very slow; recalculation instead of retrieve
Built-in vector types: char\(_n\), uchar\(_n\), short\(_n\), ushorr\(_n\), int\(_n\), uint\(_n\), long\(_n\), ulong\(_n\), float\(_n\), double\(_{1,2}\), for \(n = 1, \ldots, 4\)

These are 1D, 2D, 3D, 4D vector types. They are structures with .\(x\), .\(y\), .\(z\), .\(w\) fields

Built-in variables: gridDim, blockIdx, blockDim, threadIdx. They are of type dim3, which is the same as uint3

Built-in variables used by threads to calculate indexes in arrays, matrices, etc.
The CUDA programming model

- `<<<...>>>` is used to invoke a kernel from the host code.
- The `<<<...>>>` CUDA syntax instructs the hardware to generate and run threads on the device.
- For example, `my_kernel<<<N, M>>>(...)`
  - `N`, of type `dim3` or `int`, tells the grid size.
  - `M`, of type `dim3` or `int`, tells the block size.
- Thread scheduling is performed automatically by the hardware.
General idea:
- Execute serial code (such load data from disk) on host
- Allocate memory on device
- Copy data from host memory to device memory
- Call kernel using `<<<...>>>` syntax
- Copy result from device memory to host memory
- Release memory allocated on device

Include `cuda.h` for the CUDA runtime

Compile using the NVIDIA `nvcc` compiler
Anatomy of a CUDA program

Luca Caucci caucci@email.arizona.edu

High-Performance Computing Using GPUs
An example of a CUDA application: add two vectors $a$ and $b$ together to produce $c$

The kernel code is shown below

```c
__global__ void add_kernel(float *a, float *b, float *c, int n) {
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if(i < n) {
    c[i] = a[i] + b[i];
  }
  return;
}
```
#include <cuda.h>

__global__ void add_kernel(float *a, float *b, float *c, int n);

int main(int argv, char **argv) {
    float a[1024], b[1024], c[1024];
    float *a_dev, *b_dev, *c_dev;
    int i;

    ... // Fill out arrays a and b

cudaMalloc((void **) (& a_dev), 1024 * sizeof(float));
cudaMalloc((void **) (& b_dev), 1024 * sizeof(float));
cudaMalloc((void **) (& c_dev), 1024 * sizeof(float));
cudaMemcpy(a_dev, a, 1024 * sizeof(float), cudaMemcpyHostToDevice);
cudaMemcpy(b_dev, b, 1024 * sizeof(float), cudaMemcpyHostToDevice);
add_kernel<<<4, 256>>>(a_dev, b_dev, c_dev, 1024);
cudaMemcpy(c, c_dev, 1024 * sizeof(float), cudaMemcpyDeviceToHost);
cudaFree(a_dev); cudaFree(b_dev); cudaFree(c_dev);

    ... // Use data in array c

return(0);
}
Simple PET setup

- Gamma-ray photons emitted by object
- Points $\hat{R}_1^{(j)}$ and $\hat{R}_2^{(j)}$ estimated from PMT data
- Define $\hat{A}^{(j)} = (\hat{R}_1^{(j)}, \hat{R}_2^{(j)})$
- Build list $\hat{A} = \{\hat{A}^{(1)}, \ldots, \hat{A}^{(J)}\}$
We want to reconstruct object $f$ from list $\hat{A}$

Use the list-mode (LM) MLEM algorithm:

$$\hat{f}_n^{(k+1)} = \hat{f}_n^{(k)} \left\{ \frac{1}{\tau} \sum_{j=1}^{J} \frac{pr(\hat{A}^{(j)} | n)}{\sum_{n'=1}^{N} pr(\hat{A}^{(j)} | n') s_{n'} \hat{f}_{n'}^{(k)}} \right\}$$

Need to calculate probability of $\hat{A}^{(j)}$ given emission within $n^{th}$ voxel:

$$pr(\hat{A}^{(j)} | n) = \frac{\mu_{2e}^2}{4\pi Z_{\text{max}}^2} \times$$

$$\times \int_{D_1} \text{pr}(\hat{R}_1^{(j)} | R_1^{(j)}) \frac{e^{-\mu_{\text{tot}} \Delta_1(R_1^{(j)}; r_n)}}{|R_1^{(j)} - r_n|^2} \int_{D_2} \text{pr}(\hat{R}_2^{(j)} | R_2^{(j)}) e^{-\mu_{\text{tot}} \Delta_2(R_2^{(j)}; r_n)} \times$$

$$\times \int_{-\infty}^{\infty} \psi_{D_2}(R_1^{(j)} + (r_n - R_1^{(j)})\ell) \delta_{\text{Dir}}(R_2^{(j)} - R_1^{(j)} - (r_n - R_1^{(j)})\ell) d\ell \ d^3 R_2^{(j)} \ d^3 R_1^{(j)}$$

Integrals amenable to GPU computation
<table>
<thead>
<tr>
<th>Computing Platform</th>
<th>Task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intel® Xeon® L5506 2.13 GHz</td>
<td>pr($\hat{A}(j) \mid n$)</td>
</tr>
<tr>
<td></td>
<td>9347.17 s</td>
</tr>
<tr>
<td></td>
<td>10.70 events/s</td>
</tr>
<tr>
<td>NVIDIA Tesla C2050, 2 devices</td>
<td>30.30 s (308.49×)</td>
</tr>
<tr>
<td></td>
<td>3311.45 events/s</td>
</tr>
<tr>
<td>NVIDIA Tesla C2050, 4 devices</td>
<td>15.52 s (602.27×)</td>
</tr>
<tr>
<td></td>
<td>6442.47 events/s</td>
</tr>
<tr>
<td>NVIDIA Tesla C2050, 6 devices</td>
<td>11.22 s (833.08×)</td>
</tr>
<tr>
<td></td>
<td>8911.21 events/s</td>
</tr>
<tr>
<td>NVIDIA Tesla C2050, 8 devices</td>
<td>9.74 s (959.67×)</td>
</tr>
<tr>
<td></td>
<td>10264.06 events/s</td>
</tr>
</tbody>
</table>

Luca Caucci  caucci@email.arizona.edu  High-Performance Computing Using GPUs
- $^{18}$F-NaF bone scan for a normal mouse
- $J = 2280715$ elements in list $\hat{A}$
- $88$ mm $\times$ $88$ mm $\times$ $32$ mm FOV size
- $550$ $\mu$m $\times$ $550$ $\mu$m $\times$ $500$ $\mu$m voxel size
GPU hardware viable for high-performance computing (many $\times 100$’s speedup)

- Many products on the market
- Prices constantly falling; performance constantly increasing
- CUDA: minimal extensions to C/C++
- CUDA programming model is easy and scales well
- Tools to use GPUs with Matlab, Mathematica, …
Enormous potential for number crunching applications:

- SPECT, PET, CT, MRI, ...
- Physics, chemistry, biology, material science, ...
- Matrix operations
- Video/audio manipulation
- ...

Lots of resources:

- www.nvidia.com
- NVIDIA CUDA™ C Programming Guide
- Examples
- Books
- Forums
- ...
Questions?