Parallel Computing with CUDA

Diogo R. Ferreira
Graphics cards
GPU

• The Graphics Processing Unit (GPU)
  • computer graphics and image processing
  • massively parallel and more powerful than the CPU
  • CUDA makes the GPU accessible for general-purpose programming
CPU vs. GPU

CPU: Multiple Cores

GPU: Thousands of Cores

---

Intel Core™ i7

NVIDIA GeForce GTX

---
CPU vs. GPU

• CPU
  • higher clock speed
  • sequential tasks done in a certain order
  • ad-hoc operations on small arrays

• GPU
  • lower clock speed
  • independent tasks done in any order
  • same operation on very large arrays
CUDA

- CUDA was developed by NVIDIA
  - and works on NVIDIA GPUs
  - a more standard (but less popular) alternative is OpenCL
CUDA

- CUDA = architecture + programming model
  - architecture: hardware structure and design
  - programming model: how to develop software on top of that hardware
    - for maximum performance, software must be tuned to hardware

```c
__global__ void kernel(int* d_array, int size) {
    ... 
}

kernel<<<blocks, threads>>>(d_array, size);
```
CUDA architecture

- **Tesla** (2008): 1.0–1.3
- **Fermi** (2010): 2.0–2.1
- **Kepler** (2012): 3.0–3.7
- **Maxwell** (2014): 5.0–5.3
- **Pascal** (2016): 6.0–…

Normalized Performance

Anatomy of a CUDA GPU

GeForce GTX 750 Ti
Maxwell (5.0)
GM107 chip
The Streaming Multiprocessor

(only upper half is shown)

1 warp = 32 threads

32 cores running “in sync”
CUDA architecture

- CUDA architecture version 5.x (“Maxwell”)
  - a GPU has one or more **graphics processing clusters** (GPC)
  - each GPC has one or more **streaming multiprocessors** (SM)
  - each SM has one or more processing units (4)
  - each processing unit has a number of cores (32)
CUDA architecture

• Some examples
  • GTX 750 Ti
    • 1 GPC
    • 5 SMs
    • 4x32 cores per SM
    • 1x5x4x32 = 640 cores (total)
  • GTX 980
    • 4 GPCs
    • 4 SMs per GPC
    • 4x32 cores per SM
    • 4x4x4x32 = 2048 cores (total)
CUDA programming model

- Basic ideas
  - run multiple threads
  - prefer many short-lived threads over fewer, longer threads (why?)
  - what happens when (number of threads) > (number of cores) ?
CUDA programming model

- Threads are organized into **blocks**
  - each SM receives a block for processing
  - when finished, receives another block (if there are more)

- Questions
  - how many blocks?
  - what is the block size?
CUDA programming model

- Example
  - GPU has 5 SMs, 128 cores per SM
  - we need to run 2000 threads
  - what is the ideal block size?

<table>
<thead>
<tr>
<th>threads</th>
<th>block size</th>
<th>no. blocks</th>
<th>no. blocks</th>
<th>no. threads</th>
<th>wasted</th>
</tr>
</thead>
<tbody>
<tr>
<td>2000</td>
<td>128</td>
<td>15.6</td>
<td>16</td>
<td>2048</td>
<td>48</td>
</tr>
<tr>
<td>2000</td>
<td>256</td>
<td>7.8</td>
<td>8</td>
<td>2048</td>
<td>48</td>
</tr>
<tr>
<td>2000</td>
<td>384</td>
<td>5.2</td>
<td>6</td>
<td>2304</td>
<td>304</td>
</tr>
<tr>
<td>2000</td>
<td>512</td>
<td>3.9</td>
<td>4</td>
<td>2048</td>
<td>48</td>
</tr>
<tr>
<td>2000</td>
<td>640</td>
<td>3.1</td>
<td>4</td>
<td>2560</td>
<td>560</td>
</tr>
<tr>
<td>2000</td>
<td>768</td>
<td>2.6</td>
<td>3</td>
<td>2304</td>
<td>304</td>
</tr>
<tr>
<td>2000</td>
<td>896</td>
<td>2.2</td>
<td>3</td>
<td>2688</td>
<td>688</td>
</tr>
<tr>
<td>2000</td>
<td>1024</td>
<td>2.0</td>
<td>2</td>
<td>2048</td>
<td>48</td>
</tr>
</tbody>
</table>
Part 2
Kernels
Host vs. device

host
host memory (i.e. RAM)
host functions

device
device memory (i.e. GPU mem)
device functions
Host vs. device

called on the host kernel executed on the device
(a.k.a. global function)
Host vs. device

- The host (CPU) calls kernels
  - the CPU “coordinates” the sequence of tasks
- The device (GPU) executes kernels
  - the GPU parallelizes (i.e. “accelerates”) each task
Host vs. device

- Kernels operate on data in GPU (device) memory
- Data must be transferred between host and device
CUDA programming

- Typical CUDA programming
  - allocate memory on device
  - transfer data from host to device
  - execute kernel(s)
  - transfer results from device to host
  - free memory on device
CUDA programming

• Typical CUDA programming
  • cudaMalloc(...)  
  • cudaMemcpy(...)  
  • kernel<<blocks, threads>>>(...) 
  • cudaMemcpy(...)  
  • cudaFree()
CUDA programming

- **malloc**
  - host memory
  - input data
  - output data
  - free

- cudaMalloc
  - device memory
  - input data
  - threads
  - output data

- cudaMemcpy
  - kernel<<<...,...>>>
Kernels and threads

- A kernel is replicated into many threads
  - recap: threads are organized into blocks

```
kernel<<<blocks, threads>>>(...)
```

- number of blocks \( \times \) number of threads per block
  (according to hardware architecture)
Thread indexing

- How to identify threads

\[ i = blockIdx \times blockDim + threadIdx \]
Thread indexing

- Threads in a 2D grid

\[
i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} \\
j = \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y}
\]
Thread indexing

- Threads in a 3D grid

\[
i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x} \\
j = \text{blockIdx.y} \times \text{blockDim.y} + \text{threadIdx.y} \\
k = \text{blockIdx.z} \times \text{blockDim.z} + \text{threadIdx.z}
\]

- CUDA supports 1D, 2D, 3D grids
  - in the following example we will use 1D

\[
i = \text{blockIdx.x} \times \text{blockDim.x} + \text{threadIdx.x}
\]
Example

- Squaring numbers

```c
void square(double* A, double* B, int N) {
    for(int i=0; i<N; i++)
    {
        B[i] = A[i]*A[i];
    }
}
```

```c
__global__ void kernel_square(double* d_A, double* d_B, int N) {
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    if (i < N)
    {
        d_B[i] = d_A[i]*d_A[i];
    }
}
```

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.643</td>
<td>0.413</td>
</tr>
<tr>
<td>0.308</td>
<td>0.095</td>
</tr>
<tr>
<td>0.302</td>
<td>0.091</td>
</tr>
<tr>
<td>0.943</td>
<td>0.890</td>
</tr>
<tr>
<td>0.197</td>
<td>0.039</td>
</tr>
<tr>
<td>0.061</td>
<td>0.004</td>
</tr>
<tr>
<td>0.321</td>
<td>0.103</td>
</tr>
<tr>
<td>0.528</td>
<td>0.279</td>
</tr>
<tr>
<td>0.180</td>
<td>0.033</td>
</tr>
<tr>
<td>0.056</td>
<td>0.003</td>
</tr>
<tr>
<td>0.389</td>
<td>0.151</td>
</tr>
<tr>
<td>0.596</td>
<td>0.356</td>
</tr>
</tbody>
</table>
Example

• Squaring numbers

<table>
<thead>
<tr>
<th>N</th>
<th>square()</th>
<th>kernel_square()</th>
</tr>
</thead>
<tbody>
<tr>
<td>$10^1$</td>
<td>0.0000000</td>
<td>0.000019</td>
</tr>
<tr>
<td>$10^2$</td>
<td>0.0000000</td>
<td>0.000019</td>
</tr>
<tr>
<td>$10^3$</td>
<td>0.0000004</td>
<td>0.000020</td>
</tr>
<tr>
<td>$10^4$</td>
<td>0.0000042</td>
<td>0.000021</td>
</tr>
<tr>
<td>$10^5$</td>
<td>0.000393</td>
<td>0.000078</td>
</tr>
<tr>
<td>$10^6$</td>
<td>0.003152</td>
<td>0.000119</td>
</tr>
<tr>
<td>$10^7$</td>
<td>0.029324</td>
<td>0.000678</td>
</tr>
<tr>
<td>$10^8$</td>
<td>0.289635</td>
<td>0.006197</td>
</tr>
</tbody>
</table>

• $10^8$ is approaching the limit of GPU memory
  • $8 \times 10^8 \sim 800$ MB to store each array
CPU version

```c
int N = ...

double* A = (double*)malloc(N*sizeof(double));
double* B = (double*)malloc(N*sizeof(double));

for(int i=0; i<N; i++)
{
    A[i] = ...
}

square(A, B, N);

free(A);
free(B);

void square(double* A, double* B, int N)
{
    for(int i=0; i<N; i++)
    {
        B[i] = A[i]*A[i];
    }
}
```
double* d_A;
double* d_B;

cudaMalloc(&d_A, N*sizeof(double));
cudaMalloc(&d_B, N*sizeof(double));

cudaMemcpy(d_A, A, N*sizeof(double), cudaMemcpyHostToDevice);
...

GPU version
double* d_A;
double* d_B;

cudaMalloc(&d_A, N*sizeof(double));
cudaMalloc(&d_B, N*sizeof(double));

cudaMemcpy(d_A, A, N*sizeof(double), cudaMemcpyHostToDevice);

dim3 threads(128);
dim3 blocks((int)ceil((double)N/(double)threads.x));

kernel_square<<<blocks, threads>>>(d_A, d_B, N);
...

GPU version
double* d_A;
double* d_B;
cudaMalloc(&d_A, N*sizeof(double));
cudaMalloc(&d_B, N*sizeof(double));
cudaMemcpy(d_A, A, N*sizeof(double), cudaMemcpyHostToDevice);
dim3 threads(128);
dim3 blocks((int)ceil((double)N/(double)threads.x));
kernel_square<<<blocks, threads>>>(d_A, d_B, N);
...
double* d_A;
double* d_B;

cudaMalloc(&d_A, N*sizeof(double));
cudaMalloc(&d_B, N*sizeof(double));

cudaMemcpy(d_A, A, N*sizeof(double), cudaMemcpyHostToDevice);

dim3 threads(128);
dim3 blocks((int)ceil((double)N/(double)threads.x));

kernel_square<<<blocks, threads>>>(d_A, d_B, N);

cudaMemcpy(B, d_B, N*sizeof(double), cudaMemcpyDeviceToHost);

cudaFree(d_A);
cudaFree(d_B);
Part 3
Libraries
CUDA libraries

- cuBLAS
- cuDNN
- CUDA Math Library
- cuFFT
- cuRAND
- cuSOLVER
- cuSPARSE
- NPP
- NVBI0
- nvGRAPH
- Thrust
Fourier Transform

- Fourier Transform of a periodic signal
Discrete Fourier Transform (DFT)

- Fourier Transform of a discrete and periodic signal
Discrete Fourier Transform (DFT)

- Signal
- DFT (real part)
- DFT (magnitude)
- DFT (imaginary part)
Fast Fourier Transform (FFT)
Fast Fourier Transform (FFT)
cuFFT

• The cuFFT library
  • performs complex-to-complex DFTs
  • requires setting up a “plan” (for a given N)
  • calculates FFT by executing the plan with data
  • the data (signal) must be in device memory
  • the result (FFT) will be in device memory
  • memory allocation and transfers
cuFFT

**Typical cuFFT programming**

- use data type: cufftComplex
- allocate and copy signal: cudaMalloc/cudaMemcpy
- allocate space for result: cudaMalloc
- set up the plan: cufftPlan1d()
- execute the plan: cufftExecC2C()
- copy result to host: cudaMemcpy
Example

- Calculating the FFT of a sine wave
  - signal as A
  - FFT as B
  - both A and B complex
  - x, y are the real, imaginary parts

```c
struct float2 {
    float x, y;
};
typedef float2 cufftComplex;
```
Example

• Initializing

```c
int N = ...
cuftComplex* A = (cuftComplex*)malloc(N*sizeof(cuftComplex));
cuftComplex* B = (cuftComplex*)malloc(N*sizeof(cuftComplex));
for(int i=0; i<N; i++)
{
    A[i].x = sin(2*PI*i/N);
    A[i].y = 0.0;
}
...
```
Example

- Allocate and transfer data

```c
... 

cufftComplex* d_A;
cufftComplex* d_B;

cudaMalloc(&d_A, N*sizeof(cufftComplex));
cudaMalloc(&d_B, N*sizeof(cufftComplex));

cudaMemcpy(d_A, A, N*sizeof(cufftComplex), cudaMemcpyHostToDevice);
... 
```
Example

• Create plan and execute FFT

```c
... 
cufftHandle plan;
cufftPlan1d(&plan, N, CUFFT_C2C, 1)
cufftExecC2C(plan, d_A, d_B, CUFFT_FORWARD);
... 
```

<table>
<thead>
<tr>
<th>A.x</th>
<th>A.y</th>
<th>B.x</th>
<th>B.y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.309</td>
<td>0.0</td>
<td>0.0</td>
<td>-10.0</td>
</tr>
<tr>
<td>0.588</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.809</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>0.951</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1.000</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.309</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.588</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.809</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.951</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>-1.000</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.951</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.809</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.588</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>-0.309</td>
<td>0.0</td>
<td>0.0</td>
<td>10.0</td>
</tr>
</tbody>
</table>
Example

• Transfer results back to host

```c
... 
cudaMemcpy(B, d_B, N*sizeof(cufftComplex), cudaMemcpyDeviceToHost);
for(int i=0; i<N; i++)
{
    printf("%f %f %f %f\n", A[i].x, A[i].y, B[i].x, B[i].y);
}
... 
```
Example

• Free resources

```c
...  
cufftDestroy(plan);
cudaFree(d_A);
cudaFree(d_B);
free(A);
free(B);
...  
```
Part 4
Applications
Plasma reflectometry

- Probing the plasma with microwave frequencies
Plasma reflectometry

- I/Q signal magnitude
Plasma reflectometry

- Computing the beat frequencies
  - FFT on a sliding-window segment
Plasma reflectometry

• Spectrogram
Plasma reflectometry

- Density profile
Plasma reflectometry

• Computing the density profile

I/Q signal

- independent segments, FFTs, peak detection, etc.
- parallelizable

beat frequencies

- iterative, sequential algorithm where $r_{i+1}$ depends on $r_i$
- non-parallelizable

density profile

(total time)
Plasma reflectometry

• Computing the beat frequencies with CUDA
  • build all segments at once (kernel 1)
  • compute the FFT for each segment (cuFFT)
  • compute the magnitude of each FFT (kernel 2)
  • find the peak of each FFT (kernel 3)
Plasma reflectometry

• Computing the density profile – results

<table>
<thead>
<tr>
<th>Run time (s)</th>
<th>C version</th>
<th>CUDA version</th>
<th>Performance gain</th>
</tr>
</thead>
<tbody>
<tr>
<td>sweep #6019</td>
<td>0.011204</td>
<td>0.003446</td>
<td>3.3x</td>
</tr>
<tr>
<td>sweep #10019</td>
<td>0.011178</td>
<td>0.003402</td>
<td>3.3x</td>
</tr>
<tr>
<td>sweep #73185</td>
<td>0.011247</td>
<td>0.003429</td>
<td>3.3x</td>
</tr>
</tbody>
</table>

Table 1: Processing time for three sample sweeps from pulse #86903
Conclusion

• GPU Computing is having a profound impact in many fields
  • speedups of 10x~100x are common
  • GPUs are more powerful than CPUs
  • scientific computing can leverage GPUs
• GPU parallelization requires a different mindset
  • use large arrays and many threads
  • check for existing/applicable libraries
  • optimize for hardware
Further reading

• CUDA programming

• Applications
  • D. R. Ferreira, P. J. Carvalho, H. Fernandes, L. Meneses, “Towards real-time density profile reconstruction with CUDA”, 1st EPS Conference on Plasma Diagnostics, Frascati, Italy, April 14-17, 2015