Tutorial on GPU computing

With an introduction to CUDA

Felipe A. Cruz

University of Bristol, Bristol, United Kingdom.
The GPU evolution

- The **Graphic Processing Unit** (GPU) is a processor that was **specialized** for processing graphics.

- The GPU has recently **evolved** towards a **more flexible** architecture.

- **Opportunity**: We can implement *any algorithm*, not only graphics.

- **Challenge**: obtain **efficiency** and **high performance**.
Overview of the presentation

• Motivation
• The Buzz: GPU, Teraflops, and more!
• The reality (my point of view)
GPU computing - key ideas:

• Massively parallel.
• Hundreds of cores.
• Thousands of threads.
• Cheap.
• Highly available.
• Programable: CUDA
CUDA: Compute Unified Device Architecture

- Introduced by Nvidia in late 2006.
- CUDA is a **compiler and toolkit** for programming NVIDIA GPUs.
- CUDA API **extends the C** programming language.
- Runs on **thousands of threads**.
- It is an **scalable model**.
- Objectives:
  - Express parallelism.
  - Give a high level **abstraction from hardware**.
NVIDIA: GPU vendor

- **GPU market:** multi-billion dollars! (Nvidia +30% market)
- **Sold hundreds of millions** of CUDA-capable GPUs.
  - HPC market is tiny in comparison.
- **New GPU generation every ~18 months.**
- **Strong support** to GPU computing:
  - Hardware side: developing flexible GPUs.
  - Software side: releasing and improving development tools.
  - Community side: support to academics.

How a GPU looks like?

• Most computers have one.

• Billions of transistors.

• Computing:
  • 1 Teraflop (Single precision)
  • 100 Gflops (Double precision)

• Also:
  • A heater for winter time!

• Supercomputer for the masses?
Applications

- Many can be found at the NVIDIA site!

Felipe A. Cruz
Ok… after the buzz

• Question 1: Why accelerator technology today? If it has been around since the 70’s!

• Question 2: Can I really get 100x in my application?

• Question 3: CUDA? vendor dependent?

• Question 4: GPU computing = General-purpose on GPU?
Why accelerator technology today?

- Investment on GPU technology makes **more sense today** than in 2004.

- **CPU uni-processor speed is not doubling** every 2 years anymore!

- Case: **investing in an accelerator** that gives a ~10x speedup:
  
  - **2004** speedup 1.52x per year: 10x today would be **1.3x** acceleration in 5 years.
  
  - **TODAY** speedup 1.15x per year: 10x today would be **4.9x** acceleration in 5 years.
  
  - Consider the point that **GPU parallel performance is doubling** every 18 months!
Can I get 100x speedups?

- You can get hundred-fold speedup for some algorithms.

- It depends on the non-parallel part: Amdahl’s law.

- Complex application normally make use of many algorithms.

- Look for alternative ways to perform the computations that are more parallel.

- **Significance**: An accelerated program is going to be as fast as its serial part!
CUDA language is vendor dependent?

• Yes, and nobody wants to be locked to a single vendor.

• OpenCL is going to become an industry standard. (Some time in the future.)

• OpenCL is a low level specification, more complex to program with than CUDA C.

• CUDA C is more mature and currently makes more sense (to me).

• However, OpenCL is not “that” different from CUDA. Porting CUDA to OpenCL should be easy in the future.

• Personally, I’ll wait until OpenCL standard & tools are more mature.
GPU computing = General-purpose GPU?

- With CUDA you can program in C but with some restrictions.
- Next CUDA generation will have full support C/C++ (and much more.)
- However, GPU are still highly specialized hardware.
- Performance in the GPU does not come from the flexibility...
GPU computing features

- **Fast GPU cycle**: New hardware every ~18 months.
- Requires special programming but similar to C.
- CUDA code is forward compatible with future hardware.
- **Cheap** and available hardware (£200 to £1000).
- **Number crunching**: 1 card ~= 1 teraflop ~= small cluster.
- **Small factor** of the GPU.
- Important factors to consider: **power** and **cooling**!
CUDA introduction
with images from CUDA programming guide
What’s better?

Scooter

Sport car
What’s better?

Many scooters

Sport car
What’s better?

Many scooters

Deliver many packages within a reasonable timescale.

Sport car

Deliver a package as soon as possible
What do you need?

**High throughput and reasonable latency**

- Compute many jobs within a reasonable timeframe.

**Low latency and reasonable throughput**

- Compute a job as fast as possible.
## NVIDIA GPU Architecture

<table>
<thead>
<tr>
<th>GPU</th>
<th>G80</th>
<th>GT200</th>
<th>Fermi</th>
</tr>
</thead>
<tbody>
<tr>
<td>Transistors</td>
<td>681 million</td>
<td>1.4 billion</td>
<td>3.0 billion</td>
</tr>
<tr>
<td>CUDA Cores</td>
<td>128</td>
<td>240</td>
<td>512</td>
</tr>
<tr>
<td>Double Precision Floating Point Capability</td>
<td>None</td>
<td>30 FMA ops / clock</td>
<td>256 FMA ops /clock</td>
</tr>
<tr>
<td>Single Precision Floating Point Capability</td>
<td>128 MAD ops/clock</td>
<td>240 MAD ops / clock</td>
<td>512 FMA ops /clock</td>
</tr>
<tr>
<td>Special Function Units (SFUs) / SM</td>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Warp schedulers (per SM)</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Shared Memory (per SM)</td>
<td>16 KB</td>
<td>16 KB</td>
<td>Configurable 48 KB or 16 KB</td>
</tr>
<tr>
<td>L1 Cache (per SM)</td>
<td>None</td>
<td>None</td>
<td>Configurable 16 KB or 48 KB</td>
</tr>
<tr>
<td>L2 Cache</td>
<td>None</td>
<td>None</td>
<td>768 KB</td>
</tr>
<tr>
<td>ECC Memory Support</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Concurrent Kernels</td>
<td>No</td>
<td>No</td>
<td>Up to 16</td>
</tr>
<tr>
<td>Load/Store Address Width</td>
<td>32-bit</td>
<td>32-bit</td>
<td>64-bit</td>
</tr>
</tbody>
</table>

Comparison of NVIDIA GPU generations. Current generation: GT200. Table from NVIDIA Fermi whitepaper.
CUDA architecture

- Support of languages: C, C++, OpenCL.
- Windows, Linux, OS X compatible.

![CUDA Architecture Diagram]

Felipe A. Cruz
Strong points of CUDA

• Abstracting from the hardware
  
  • Abstraction by the CUDA API. You don’t see every little aspect of the machine.
  
  • Gives flexibility to the vendor. Change hardware but keep legacy code.
  
• Forward compatible.

• Automatic Thread management (can handle +100k threads)
  
  • Multithreading: hides latency and helps maximize the GPU utilization.
  
  • Transparent for the programmer (you don’t worry about this.)
  
  • Limited synchronization between threads is provided.

• Difficult to dead-lock. (No message passing!)
Programmer effort

- Analyze algorithm for **exposing parallelism**:
  - Block size
  - Number of threads

- **Tool**: pen and paper

- **Challenge**: Keep machine busy (with limited resources)
  - Global data set (Have efficient data transfers)
  - Local data set (Limited on-chip memory)
  - Register space (Limited on-chip memory)

- **Tool**: Occupancy calculator
Outline

• Memory hierarchy.
• Thread hierarchy.
• Basic C extensions.
• GPU execution.
• Resources.
Thread hierarchy

- Kernels are executed by **thread**.
- A kernel is a **simple C** program.
- Each thread has its own **ID**.
- **Thousands** of threads execute the same kernel.
- Threads are grouped into **blocks**.
  - Threads in a block can **synchronize** execution.
- Blocks are grouped in a **grid**.
  - Blocks are **independent** (Must be able to be executed in any order.)
Memory hierarchy

- Three **types** of memory in the graphic card:
  - Global memory: 4GB
  - Shared memory: 16 KB
  - Registers: 16 KB

- **Latency**:
  - Global memory: 400-600 cycles
  - Shared memory: Fast
  - Register: Fast

- **Purpose**:
  - Global memory: IO for grid
  - Shared memory: thread collaboration
  - Registers: thread space
Basic C extensions

Function modifiers

- `__global__` : to be called by the host but executed by the GPU.
- `__host__` : to be called and executed by the host.

Kernel launch parameters

- Block size: (x, y, z). \(x\times y\times z\) = Maximum of 768 threads total. (Hw dependent)
- Grid size: (x, y). Maximum of thousands of threads. (Hw dependent)

Variable modifiers

- `__shared__` : variable in shared memory.
- `__syncthreads()` : sync of threads within a block.

Check CUDA programming guide for all the features!
Example: device

• Simple example: add two arrays

• Not strange code: It is C with extensions.

```c
// Device code
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

• Example from CUDA programming guide
Example: device

• Simple example: add two arrays

• Not strange code: It is C with extensions.

```c
// Device code
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}
```

• Example from CUDA programming guide
```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMemcpy((void**)&d_A, size);
    float* d_B;
    cudaMemcpy((void**)&d_B, size);
    float* d_C;
    cudaMemcpy((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```
Example: host

```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc(&d_A, size);
    float* d_B;
    cudaMalloc(&d_B, size);
    float* d_C;
    cudaMalloc(&d_C, size);

    // Copy vectors from host memory to device memory
    // cudaMemcpy : copy from host to device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // cudaMemcpy : copy from device to host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```
Example: host

```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```
Example: host

```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}```
Example: host

```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc(void**)&d_A, size);
    float* d_B;
    cudaMalloc(void**)&d_B, size);
    float* d_C;
    cudaMalloc(void**)&d_C, size);

    // Copy vectors from host memory to device memory
    cudaMemcp(y(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);
}
```
Example: host

```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```
Example: host

```c
// Host code
int main()
{
    // Allocate vectors in device memory
    size_t size = N * sizeof(float);
    float* d_A;
    cudaMalloc((void**)&d_A, size);
    float* d_B;
    cudaMalloc((void**)&d_B, size);
    float* d_C;
    cudaMalloc((void**)&d_C, size);

    // Copy vectors from host memory to device memory
    // h_A and h_B are input vectors stored in host memory
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // Invoke kernel
    int threadsPerBlock = 256;
    int threadsPerGrid =
        (N + threadsPerBlock - 1) / threadsPerBlock;
    VecAdd<<<threadsPerGrid, threadsPerBlock>>>(d_A, d_B, d_C);

    // Copy result from device memory to host memory
    // h_C contains the result in host memory
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

    // Free device memory
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);
}
```
Work flow

Memory allocation

Memory copy: Host -> GPU

Kernel call

Memory copy: GPU -> Host

Free GPU memory

// Device code
__global__ void VecAdd(float* A, float* B, float* C)
{
    int i = threadIdx.x;
    if (i < N)
        C[i] = A[i] + B[i];
}