GPU programming

Dr. Bernhard Kainz
Overview

• About myself
• Motivation
• GPU hardware and system architecture
• GPU programming languages
• GPU programming paradigms
• Pitfalls and best practice
• Reduction and tiling examples
• State-of-the-art applications
About myself

• Born, raised, and educated in Austria

• PhD in interactive medical image analysis and visualisation

• Marie-Curie Fellow, Imperial College London, UK

• Senior research fellow King‘s College London

• Lecturer in high-performance medical image analysis at DOC

• > 10 years GPU programming experience
History
GPUs

**GPU** = graphics processing unit

**GPGPU** = General Purpose Computation on Graphics Processing Units

**CUDA** = Compute Unified Device Architecture

**OpenCL** = Open Computing Language

Images: www.geforce.co.uk

Dr Bernhard Kainz
History

First dedicated GPUs

1998

Other (graphics related) developments

Brook

2004

2007

CUDA

2008

Modern interfaces to CUDA and OpenCL (python, Matlab, etc.)

2008

you

Now

programmable shader

Dr Bernhard Kainz
Why GPUs became popular

http://www.computerhistory.org/timeline/graphics-games/
Why GPUs became popular for computing

Intel CPU Trends
(sources: Intel, Wikipedia, K. Olukotun)

Dual-Core Itanium 2
Pentium 4
Pentium
386

Sandy Bridge Haswell

© HerbSutter „The free lunch is over“
Motivation
parallelisation

for (int i = 0; i < N; ++i)
    c[i] = a[i] + b[i];

Thread 0

\[
\begin{array}{ccc}
1 & + & 1 \\
\vdots & & \vdots \\
\vdots & & \vdots \\
\vdots & & \vdots \\
\vdots & & \vdots \\
\end{array}
\begin{array}{ccc}
& & \\
& & \\
& & \\
& & \\
& & \\
\end{array}
= \begin{array}{ccc}
2 \\
\vdots \\
\vdots \\
\vdots \\
\vdots \\
\end{array}
\]
parallelisation

\[ \begin{array}{ccc}
1 & + & 1 \\
\vdots & & \vdots \\
\hline
\end{array} = \begin{array}{ccc}
2 \\
\vdots \\
\hline
\end{array} \]

Thread 0
\[
\text{for } (\text{int } i = 0; i < N/2; ++i) \\
c[i] = a[i] + b[i];
\]

Thread 1
\[
\text{for } (\text{int } i = N/2; i < N; ++i) \\
c[i] = a[i] + b[i];
\]
parallelisation

\[
1 + 1 = 2
\]

Thread 0
\[
\text{for (int } i = 0; i < \text{N}/3; ++i) \\
\quad c[i] = a[i] + b[i];
\]

Thread 1
\[
\text{for (int } i = \text{N}/3; i < 2*\text{N}/3; ++i) \\
\quad c[i] = a[i] + b[i];
\]

Thread 2
\[
\text{for (int } i = 2*\text{N}/3; i < \text{N}; ++i) \\
\quad c[i] = a[i] + b[i];
\]
multi-core CPU

\begin{center}
\begin{tikzpicture}
\node[draw] (control) at (0,0) {Control};
\node[draw] (alu1) at (1,0) {ALU};
\node[draw] (alu2) at (2,0) {ALU};
\node[draw] (alu3) at (1,-1) {ALU};
\node[draw] (alu4) at (2,-1) {ALU};
\node[draw] (cache) at (0,-2) {Cache};
\node[draw] (dram) at (0,-3) {DRAM};
\end{tikzpicture}
\end{center}
parallelisation

\[
\begin{array}{ccc}
1 & + & 1 \\
... & & ...
\end{array}
\quad =
\begin{array}{ccc}
2 \\
... & & ...
\end{array}
\]

\[
\begin{align*}
c[0] &= a[0] + b[0]; \\
c[1] &= a[1] + b[1]; \\
c[N-1] &= a[N-1] + b[N-1]; \\
c[N] &= a[N] + b[N];
\end{align*}
\]
multi-core GPU

DRAM
Terminology
Host vs. device

CPU (host)

GPU w/ local DRAM (device)
current schematic
Nvidia Maxwell architecture
Streaming Multiprocessors (SM, SMX)

- single-instruction, multiple-data (SIMD) hardware
- 32 threads form a warp
- Each thread within a warp must execute the same instruction (or be deactivated)
- 1 instruction $\rightarrow$ 32 values computed
- handle more warps than cores to hide latency
Differences CPU-GPU

• Threading resources
  - Host currently ~32 concurrent threads
  - Device: smallest executable unit of parallelism: “Warp”: 32 thread
  - 768-1024 active threads per multiprocessor
  - Device with 30 multiprocessors: > 30,000 active threads
  - Devices can hold billions of threads

• Threads
  - Host: heavyweight entities, context switch expensive
  - Device: lightweight threads
  - If the GPU processor must wait for one warp of threads, it simply begins executing work on another Warp.

• Memory
  - Host: equally accessible to all code
  - Device: divided virtually and physically into different types
Flynn’s Taxonomy

- **SISD**: single-instruction, single-data (single core CPU)
- **MIMD**: multiple-instruction, multiple-data (multi core CPU)
- **SIMD**: single-instruction, multiple-data (data-based parallelism)
- **MISD**: multiple-instruction, single-data (fault-tolerant computers)
Amdahl’s Law

- Sequential vs. parallel
- Performance benefit

\[ S(N) = \frac{1}{(1 - P) + \frac{P}{N}} \]

- P: parallelizable part of code
- N: # of processors
SM Warp Scheduling

- SM hardware implements zero overhead Warp scheduling
  - Warps whose next instruction has its operands ready for consumption are eligible for execution
  - Eligible Warps are selected for execution on a prioritized scheduling policy
  - All threads in a Warp execute the same instruction when selected
- Currently: ready-queue and memory access scoreboarding
- Thread and warp scheduling are active topics of research!
Programming GPUs
Programming languages

• OpenCL (Open Computing Language):
  - **OpenCL is an open, royalty-free, standard for cross-platform, parallel programming of modern processors**
  - An Apple initiative approved by Intel, Nvidia, AMD, etc.
  - Specified by the Khronos group (same as OpenGL)
  - It intends to unify the access to heterogeneous hardware accelerators
    - CPUs (Intel i7, …)
    - GPUs (Nvidia GTX & Tesla, AMD/ATI 58xx, …)
  - **What’s the difference to other languages?**
    - Portability over Nvidia, ATI, S3… platforms + CPUs
    - Slow or no implementation of new/special hardware features
Programming languages

• CUDA:
  - “Compute Unified Device Architecture”
    - Nvidia GPUs only!
    - Open source announcement
    - Does not provide CPU fallback
    - NVIDIA CUDA Forums – 26,893 topics
      - AMD OpenCL Forums – 4,038 topics
    - Stackoverflow CUDA Tag – 1,709 tags
      - Stackoverflow OpenCL Tag – 564 tags
    - Raw math libraries in NVIDIA CUDA
      - CUBLAS, CUFFT, CULA, Magma
    - new hardware features immediately available!
Installation

- Download and install the newest driver for your GPU!
- OpenCL: get SDK from Nvidia or AMD
- CUDA nvcc complier -> easy access via CMake and .cu files
- OpenCL -> no special compiler, runtime evaluation
- Integrated Intel something graphics -> No No No!
Writing parallel code

• Current GPUs have > 3000 cores (GTX TITAN, Tesla K80 etc.)

• Need more threads than cores (warp scheduler)

• Writing different code for 10000 threads / 300 warps?

→ Single-program, multiple-data (SPMD = SIMD) model
  - Write one program that is executed by all threads
CUDA C is C (C++) with additional keywords to control parallel execution

```c
__device__ float x;
__global__ void func(int* mem)
{
    __shared__ int y[32];
    ...
    y[threadIdx.x] = blockIdx.x;
    __syncthreads();
}
...
cudaMalloc(&d_mem, bytes);
func<<<<<10,10>>>>(d_mem);
```

**Type qualifiers**
- `__device__`
- `__global__`
- `__constant__`
- `__shared__`
- `__device__`

**Keywords**
- threadIdx
- blockIdx
- cudaMalloc
- __any()
- __syncthreads()

**Intrinsics**
- GPU code (device code)
- CPU code (host code)

**Runtime API**

**GPU function launches**

Dr Bernhard Kainz
Kernel

- A function that is executed on the GPU
- Each started thread is executing the same function

```c
__global__ void myfunction(float *input, float *output) {
  *output = *input;
}
```

- Indicated by `__global__`
- must have return value `void`
Parallel Kernel

- Kernel is split up in blocks of threads
Launching a kernel

- A function that is executed on the GPU
- Each started thread is executing the same function

```c
dim3 blockSize(32,32,1);
dim3 gridSize((iSpaceX + blockSize.x - 1)/blockSize.x,
  (iSpaceY + blockSize.y - 1)/blockSize.y), 1)
myfunction<<<gridSize, blockSize>>>(input, output);
```

- Indicated by `__global__`
- must have return value `void`
Distinguishing between threads

- using `threadIdx` and `blockIdx` execution paths are chosen
- with `blockDim` and `gridDim` number of threads can be determined

```cpp
__global__ void myfunction(float *input, float* output)
{
    uint bid = blockIdx.x + blockIdx.y * gridDim.x;
    uint tid = bid * (blockDim.x * blockDim.y) +
                (threadIdx.y * blockDim.x) + threadIdx.x;
    output[tid] = input[tid];
}
```
Distinguishing between threads

- **blockId** and **threadId**
Grids, Blocks, Threads

Dr Bernhard Kainz
Blocks

• Threads within one block…
  - are executed together
  - can be synchronized
  - can communicate efficiently
  - share the same local cache

→ can work on a goal cooperatively
Blocks

- Threads of different blocks...
  - may be executed one after another
  - cannot synchronize on each other
  - can only communicate inefficiently
  ➔ should work independently of other blocks
Block Scheduling

• Block queue feeds multiprocessors
• Number of available multiprocessors determines number of concurrently executed blocks
Blocks to warps

- On each multiprocessor each block is split up in warps. Threads with the lowest id map to the first warp.

<table>
<thead>
<tr>
<th></th>
<th>0,1</th>
<th>1,1</th>
<th>2,1</th>
<th>3,1</th>
<th>4,1</th>
<th>5,1</th>
<th>6,1</th>
<th>7,1</th>
<th>8,1</th>
<th>9,1</th>
<th>10,1</th>
<th>11,1</th>
<th>12,1</th>
<th>13,1</th>
<th>14,1</th>
<th>15,1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0,0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0,7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

warp 0
warp 1
warp 2
warp 3
Where to start

- CUDA programming guide:
  https://docs.nvidia.com/cuda/cuda-c-programming-guide/
- OpenCL
  http://www.nvidia.com/content/cudazone/download/opencl/nvidia_opencl_programmingguide.pdf
GPU programming

Dr. Bernhard Kainz