Welcome!
Today’s Agenda:

- Introduction to GPUs
- GPU Processing Architecture
- Kernels
- SIMT Execution Model
Introduction to GPUs

Heterogeneous Processing

Modern PCs contain a CPU and a GPU

We have optimized the CPU code, but left the vast amount of GPU power untouched

*Graphics Processing Unit*
Introduction to GPUs

Consistent approach

1. Determine optimization requirements
2. Profile: determine hotspots
3. Analyze hotspots: determine scalability
4. Apply high-level optimizations to hotspots
5. Profile again
6. Parallelize / vectorize / use GPGPU
7. Profile again
8. Apply low level optimizations to hotspots
9. Repeat steps 7 and 8 until time runs out
10. Report

https://rog.asus.com/motherboards/rog-strix/rog-strix-b560-a-gaming-wifi-model/
Introduction to GPUs

AMD:
RX 6900 XT  
$1000  
512 GB/s  
23.04 TFLOPS

NVidia:
GTX3090  
$1500  
936 GB/s  
35.58 TFLOPS

Intel:
i9-11900k  
$880  
50 GB/s  
0.857 TFLOPS

AMD:
Ryzen 9 5900x  
$549  
50 GB/s  
0.908 TFLOPS

NVidia GTX4080: 48.7 TFLOPS

5000  
10  
100  
1000  
10000  
100000  
1000000  
10000000  
2008  
2010  
2012  
2014  
2016  

Theoretical Peak Performance, Single Precision

Introduction to GPUs

A Brief History of GPGPU

NVidia NV-1 (Diamond Edge 3D)
1995

3Dfx – Diamond Monster 3D
1996

NVidia NV-1
(Diamond Edge 3D)
1995

NVidia NV-1
(Diamond Edge 3D)
1995

- Ordinary VGA Quake
  Resolution: 320x200
  Colors: 256
  Frame-rate: 30fps

- OpenGL Quake on 3Dfx
  Resolution: 640x480
  Colors: 65,536
  Frame-rate: 30fps
Introduction to GPUs

A Brief History of GPGPU

Nvidia claims to have made the first ‘real’ GPU, capable of processing over 10 million triangles per second.

Nvidia ‘256’

"a single chip processor with integrated transform, lighting, triangle setup/clipping, and rendering engines that is capable of processing a minimum of 10 million polygons per second."

- Nvidia, August 31, 1999
Metro Exodus looks a lot like Mad Max in these stunning new 4K screenshots.
Introduction to GPUs

A Brief History of GPGPU

Fixed Function Pipeline

https://www.khronos.org/opengl/wiki/Rendering_Pipeline_Overview

https://blog.griffy.com/2021/01/14/shader-programming/
Introduction to GPUs

A Brief History of GPGPU

GPUs perform well on tasks that require massive parallelism

CPUs perform well on very complex single-threaded tasks

CPU: Designed to run one thread as fast as possible.

- Use caches to minimize memory latency
- Use pipelines and branch prediction
- Multi-core processing: task parallelism

GPU: Designed to combat latencies by using many threads

- Hide latency by computation
- Maximize parallelism
- SIMT
Today's Agenda:

- Introduction to GPUs
- GPU Processing Architecture
- Kernels
- SIMT Execution Model
GPU Processing Architecture

CPU Architecture

- 4 cores
- L1 and L2 cache on chip
- L3 cache shared
- RAM separate
- Integrated graphics chip

https://tweakers.net/reviews/4162/4/de-skylake-architectuur-de-techniek-achter-de-nieuwste-processors-de-architectuur-een-overzicht.html
GPU Processing Architecture

- Hundreds to thousands of cores!
- Level 1 and 2 cache on chip
- RAM separate
Full GPU with 84 streaming multiprocessors
GPU Processing Architecture

Today we will delve deeper into the architecture of a GPU

A NVidia GPU consist of these components

- Graphics Processing Clusters
- Texture Processing Clusters
- Streaming Multiprocessors
- Raster Operators (we will not discuss these)

Ordered in a hierarchical fashion
GPU Processing Architecture

Graphics Processing Clusters (GPCs)

A GPC is a high-level hardware block with all key graphics processing units residing inside.
Graphics Processing Clusters (GPCs)

A GPC is a high-level hardware block with all key graphics processing units residing inside:

- 2 ROP partitions (containing 8 ROPS each)
- 6 TPCs (containing 2 SMs each)
GPU Processing Architecture

Texture Processing Clusters (TPCs)
GPU Processing Architecture

Texture Processing Clusters (TPCs)

- Polymorph engine
- 2 Streaming Multiprocessors
GPU Processing Architecture

Streaming Multiprocessors (SMs)

A SM contains the logical cores for performing operations on data

SMs on modern NVidia GPUs contain:
- 4 processing blocks containing:
  - 1x 16 FP32 units
  - 1x 16 FP32 and 16 INT32 units
  - 4 Tensor Cores
  - L0 i-cache, warp scheduler, instruction dispatch
- 1 Ray Tracing Core
- L1 Data cache/Shared Memory
- 4 Texture Units

In total, each SM contains 16 * 2 * 4 = 128 cores that can operate simultaneously on FP data.
GPU Processing Architecture

Hierarchical order

TPC

PolyMorph Engine
Today's Agenda:

- Introduction to GPUs
- GPU Processing Architecture
- Kernels
- SIMT Execution Model
**DISCLAIMER!**

During the course we discuss the GPGPU programming model using OpenCL

**Why OpenCL?**
- Cross platform
- Relatively easy to setup and use
- Maintained by the Khronos group (they also maintain OpenGL and Vulkan)

This is not a course on OpenCL programming! We discuss how the architecture of a GPU translates to GPGPU programming.
Kernels

Several GPGPU languages

- OpenCL (cross-platform)
- OpenGL/Vulkan (compute shaders)
- CUDA (Nvidia)
- Metal (Apple specific, high-performance shaders)
Kernels

Kernel execution

For many tasks we are used to execute loop-bodies

Example: array multiplication

```c
void mulArrays(float* a, float* b, float* c, int N) {
    for (int i = 0; i < N; i++)
        c[i] = a[i] * b[i];
}
```
OpenCL programming model

Kernel execution

For a GPU we define a task per thread; a *kernel*

Example: array multiplication

```c
__kernel void mulArrays(__global float* a, __global float* b, __global float* c) {
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
}
```
Kernels

Kernel execution

In GPGPU we write kernels that are executed per-thread

- We pass parameters from host
- Explicitly allocate memory (GPGPU 2)
- Retrieve index from thread-number
- Enqueue kernels from host

// Device code

__kernel void mulArrays(__global float* a, __global float* b, __global float* c) {
    int id = get_global_id(0);
    c[id] = a[id] * b[id];
}

// Host code

...
Kernels

Kernel execution

When we enqueue a thread, we must indicate a global work-size, a local work-size and the working dimensions.

- **Global ws**: total # of threads
- **Local ws**: # of threads per block

**total # of blocks** = \( \frac{\text{global ws}}{\text{local ws}} \)

Working dimensions indicate how our threads are ‘grouped’: 1-, 2- or 3-dimensional. Example:

<table>
<thead>
<tr>
<th>w-dim</th>
<th>Global ws</th>
<th>Local ws</th>
<th># blocks</th>
<th># threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.048.576</td>
<td>512</td>
<td>2048</td>
<td>1.048.576</td>
</tr>
<tr>
<td>2</td>
<td>(1024, 1024)</td>
<td>(32, 16)</td>
<td>2048</td>
<td>1.048.576</td>
</tr>
<tr>
<td>3</td>
<td>(256, 256, 64)</td>
<td>(8, 8, 8)</td>
<td>2048</td>
<td>1.048.576</td>
</tr>
</tbody>
</table>
Kernels

Kernel execution

When we enqueue a thread, we must indicate a global work-size, a local work-size and the working dimensions:

- Global ws: total # of threads
- Local ws: # of threads per block
- total # of blocks = \( \frac{\text{global ws}}{\text{local ws}} \)

Working dimensions indicate how our threads are ‘grouped’: 1-, 2- or 3-dimensional. Example:

<table>
<thead>
<tr>
<th>w-dim</th>
<th>Global ws</th>
<th>Local ws</th>
<th># blocks</th>
<th># threads</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1.048.576</td>
<td>512</td>
<td>2048</td>
<td>1.048.576</td>
</tr>
<tr>
<td>2</td>
<td>(1024, 1024)</td>
<td>(32, 16)</td>
<td>2048</td>
<td>1.048.576</td>
</tr>
<tr>
<td>3</td>
<td>(256, 256, 64)</td>
<td>(8, 8, 8)</td>
<td>2048</td>
<td>1.048.576</td>
</tr>
</tbody>
</table>

Working dimensions do not affect performance, local work-size does!*
Kernels

Thread execution

Thread or work-item

0 1 2 ...

block or work-group

0 1 2 ...

Grid of blocks/work-groups

0 1 2 ...

Threads are executed by a scalar processor

Blocks are executed on SMs

They are always executed on the same SM

A grid of blocks is always executed on the same device
**Kernels**

**Kernel execution**

In our kernel we can retrieve three types of id's

1. **get_global_id(int dim)**, retrieves the global thread id
2. **get_local_id(int dim)**, retrieves the thread id within the work-group \([0, \text{local size} - 1]\)
3. **get_group_id(int dim)**, retrieves the work-group (block) id within the grid \([0, \frac{\text{global size}}{\text{local size}}]\)
**Kernels**

Kernel execution

Work-group sizes are capped: sizes vary per device (compute capability for Nvidia)

In OpenCL we can retrieve the maximum block size for a device by calling:

```
cGetDeviceInfo(device, CL_DEVICE_MAX_WORK_GROUP_SIZE, ...);
```
Today's Agenda:

- Introduction to GPUs
- GPU Processing Architecture
- Kernels
- SIMT Execution Model
SIMT Execution Model

The GPU execution model is referred to as SIMT: Single Instruction, Multiple Threads

This is similar to SIMD – Single Instruction, Multiple Data – on the CPU

Threads are executed on a GPU in small groups, called ‘warps’ (‘wavefronts’ for AMD)

A warp consists of 32 threads that operate in lockstep – comparable to SIMD
SIMT Execution Model

**Warps**

- Warps consist of 32 threads
- One SM contains 128 cores, meaning that one SM can execute 128 / 32 = 4 warps simultaneously
- Warps always execute on the same SM
SIMT Execution Model

Warps – Lockstep

Warps in a thread execute in lockstep:

- At each cycle, all threads in a warp must execute the same instruction with different data

```
float rnd(uint seed) {
    seed ^= seed << 13;  // i0
    seed ^= seed >> 17;  // i1
    seed ^= seed << 5;   // i2
    return seed / UINT_MAX;  // i3
}
```
SIMT Execution Model

Warps – Conditional Code

What happens when we have conditional code?

Threads for which the conditional code is not executed are temporarily disabled, therefore reducing GPU occupancy. Note the similarity to SIMD code!

<table>
<thead>
<tr>
<th>Thread</th>
<th>cycle 0</th>
<th>cycle 1</th>
<th>cycle 2</th>
<th>cycle 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i0</td>
<td>i1</td>
<td>i2</td>
<td>i3</td>
</tr>
<tr>
<td>1</td>
<td>i0</td>
<td>i1</td>
<td></td>
<td>i3</td>
</tr>
<tr>
<td>2</td>
<td>i0</td>
<td>i1</td>
<td>i2</td>
<td>i3</td>
</tr>
<tr>
<td>...</td>
<td>i0</td>
<td>i1</td>
<td></td>
<td>i3</td>
</tr>
<tr>
<td>31</td>
<td>i0</td>
<td>i1</td>
<td></td>
<td>i3</td>
</tr>
</tbody>
</table>

```c
int a = get_local_id(0); // i0
if (a % 2 == 0) { // i1
  a = 0;
} // i2
return a; // i3
```
SIMT Execution Model

Warps – Conditional Code

What happens when we have conditional code?

Threads for which the conditional code is not executed are temporarily disabled, therefore reducing GPU occupancy. Note the similarity to SIMD code!

It gets even worse for if-else statements!

<table>
<thead>
<tr>
<th>Thread</th>
<th>cycle 0</th>
<th>cycle 1</th>
<th>cycle 2</th>
<th>cycle 3</th>
<th>cycle 4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>i0</td>
<td>i1</td>
<td>i2</td>
<td></td>
<td>i4</td>
</tr>
<tr>
<td>1</td>
<td>i0</td>
<td>i1</td>
<td></td>
<td>i3</td>
<td>i4</td>
</tr>
<tr>
<td>2</td>
<td>i0</td>
<td>i1</td>
<td>i2</td>
<td></td>
<td>i4</td>
</tr>
<tr>
<td>...</td>
<td>i0</td>
<td>i1</td>
<td>...</td>
<td>...</td>
<td>i4</td>
</tr>
<tr>
<td>31</td>
<td>i0</td>
<td>i1</td>
<td></td>
<td>i3</td>
<td>i4</td>
</tr>
</tbody>
</table>

int a = get_local_id(0); // i0
if (a % 2 == 0) // i1
    a = 0;
else a = 1; // i3
return a; // i4
SIMT Execution Model

Warps – Hiding Latencies

Execution of a warp can be stalled by e.g., memory access

GPUs hide latencies by swapping out blocked warps

```
int id = get_global_id(0); // i0
int a = my_buffer[id]; // i1
return a >> 2 + 15; // i2
```
SIMT Execution Model

Warps – Hiding Latencies

- GPUs hide latencies by swapping out blocked warps
- Each SM contains 4 Warp Schedulers

- To be able to hide latencies we must feed the GPU with enough tasks to perform

#threads > cores (SMs) * 4 * 32

Many resources available online to provide an indication of the number of threads you need. In general, more is better!
Kernels

Kernel execution

A SM has 64K 32-bit registers to distribute among each thread within a block

For a block-size of 1024, each thread may use 64 registers

Registers overflow in global-memory (DRAM)

In this case, fewer threads are better. But how many?

No mathematical model, find a balance by tweaking parameters (profile performance!)

<table>
<thead>
<tr>
<th>Technical specifications</th>
<th>Compute capability (version)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum number of threads per block</td>
<td>512</td>
</tr>
<tr>
<td>Warp size</td>
<td>1624</td>
</tr>
<tr>
<td>Maximum number of resident blocks per multiprocessor</td>
<td>24 32 46 64</td>
</tr>
<tr>
<td>Maximum number of resident warps per multiprocessor</td>
<td>768 1024 1536</td>
</tr>
<tr>
<td>Maximum number of resident threads per multiprocessor</td>
<td>2048 1024 2048 1536</td>
</tr>
<tr>
<td>Number of 32-bit registers per multiprocessor</td>
<td>8K 16K 32K 64K 128K 64K</td>
</tr>
</tbody>
</table>
Brief Summary

- We create tasks (kernels) per thread
- Threads are organized in work groups
- Work groups always execute on the same SM
- Threads in a work group execute as warps
- Warps operate in lockstep
- Branching and memory operations incur latencies
- Latencies from e.g., memory access are hidden by swapping out blocked warps
- Threads are managed and scheduled by the hardware (Warp Scheduler)
END of “GPGPU 1”
Demonstration

- In the working lecture, I will demonstrate how we may convert the Mandelbrot project from the Low-Level lecture to GPGPU.