Welcome!
Today's Agenda:

- Recap
- GPU Memory Architecture
- Memory Allocation
- Synchronization
Recap

Recap – Processing Architecture

The processing components of a GPU are structured hierarchically
Recap

Recap - Processing Architecture

TPC

PolyMorph Engine
Recap

Recap – Thread Execution

Thread or work-item

Scalar processor

Scalar processor

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
Recap

Recap – SIMT

Threads are organized in groups of 32 called warps (or wavefront for AMD)

The threads in a warp execute in lockstep

Each thread executes the same instruction at any given cycle

Branching temporarily disables threads who do not traverse the branch

<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>i2</td>
<td>i3</td>
</tr>
<tr>
<td>...</td>
<td>i0</td>
<td>i1</td>
<td>i2</td>
<td>i3</td>
</tr>
<tr>
<td>31</td>
<td>i0</td>
<td>i1</td>
<td>i2</td>
<td>i3</td>
</tr>
</tbody>
</table>

<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>i3</td>
<td></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>
Recap

Recap – SIMT

Most latencies on a GPU arise from memory operations

Latencies are hidden by swapping blocked warps by active warps

This is done in hardware by the warp scheduler

Queuing more threads per work group provides more opportunity for warp swapping

However, threads in a work group share resources available on a SM
Today’s Agenda:

- Recap
- GPU Memory Architecture
- Memory Allocation
- Synchronization
GPU Memory Architecture

Memory

- Caching hierarchy (l1, l2, and l3)
- RAM on motherboard, not on chip
- Memory hierarchy is implicit

CPU

- Caching hierarchy (l1, l2, and l3)
- RAM on motherboard, not on chip
- Memory hierarchy is implicit

GPU

- Only 2 levels of cache
- DRAM on graphics board, not on chip
- Memory hierarchy is explicit! (except for caching)

[Image of CPU and GPU memory architectures]

https://www.bsc.es/research-development/research-areas/computer-architecture-and-codesign/memory-hierarchy-gpu
GPU Memory Architecture

The memory system on a GPU is different from the CPU

- Memory architecture
- Explicit allocation

We must specifically tell our GPU what kind of memory we would like to use!
GPU Memory Architecture

There are multiple memory locations*:

- DRAM
- Level-2 Cache
- Level-1 Cache
- Register memory
GPU Memory Architecture

**DRAM**

- Slowest type of memory
- On the graphics board, not on chip
- Accessible by all SMs
- Multiple GBs

Texture memory and the Constant Data Cache also reside in DRAM. The difference is in how they are accessed.
GPU Memory Architecture

Level-2 Cache

- On chip
- Faster than DRAM
- Shared by all SMs
- Size: ~40MB*

* Sizes vary per device
GPU Memory Architecture

Level-1 Data Cache

* Sizes vary per device
GPU Memory Architecture

Level-1 Data Cache

- On chip
- Second fastest memory
- Unique for each SM (no synchronization)
- Size: 128KB*

The actual size depends on its configuration, more on this later.
GPU Memory Architecture

- Fastest type of memory
- Accessible per thread
- 64K registers per SM (128 cores)

Each thread receives its own registers to work on. We typically assign more than 128 threads per SM (e.g., 1024 threads*).

---

GPU Memory Architecture

Latencies

<table>
<thead>
<tr>
<th>Memory Type</th>
<th>Latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registers</td>
<td>0 - 1 cycles</td>
</tr>
<tr>
<td>Level-1 data cache</td>
<td>1 – 32 cycles</td>
</tr>
<tr>
<td>Level-2 cache</td>
<td>~200 cycles</td>
</tr>
<tr>
<td>DRAM</td>
<td>400 – 600 cycles</td>
</tr>
</tbody>
</table>

High latencies for data access on the GPU!

We hide these latencies by fetching data simultaneously while working on other threads.
Today’s Agenda:

- Recap
- GPU Memory Architecture
- Memory Allocation
- Synchronization
Explicit Memory Allocation

- On a CPU we allocate memory using for example `malloc` or `new`
- After allocation we do not know where our memory resides (RAM, L1-$\$, L2-$\$, L3-$\$, ...)

```c
int * buffer = (int*)malloc(sizeof(int) * 1024);
// Operate on buffer ...
free(buffer);

int* myArray = new int[1024];
// Operate on array ...
delete[] myArray;
```
GPU Memory Allocation

Explicit Memory Allocation

- Allocating memory on a GPU is explicit
- There are several “memory spaces” mapped to the memory hierarchy as discussed previously
  - Global memory
  - Shared memory
  - Local memory
  - Register memory
  - And others...

- In this lecture we discuss the various memory locations and the analogous OpenCL allocations
GPU Memory Allocation

OpenCL Memory Allocation

We can explicitly store data in 5 locations: registers, shared memory, global memory, constant memory, and texture memory.

We will not discuss texture memory today

In OpenCL we get 4 address specifiers which point to the various memory locations:

__private, __local, __global and __constant
GPU Memory Allocation

Global Memory

Located in DRAM

Global memory is accessible by all threads

Also accessible by the host (CPU) for data transfer between the GPU and CPU.

Very slow memory, not being cached
GPU Memory Allocation

__global

In OpenCL we use the __global keyword to allocate memory in global memory (DRAM)

This data is accessible by all threads

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

__kernel void kernel_func(...) {
    // pointer in private address space, pointing towards data in global space
    __global int* p;
}
GPU Memory Allocation

__global__

The host can read and write global memory too

We can define memory buffers on the host which can be read and written

```c
cl_mem buffer = clCreateBuffer(context, cl_mem_flags, size, ...);
```

Memory reserved by these buffers is always located in DRAM (or on the host)

Similar to C++ memory allocation, we must also deallocate memory: `clReleaseMemObject(buffer)`
GPU Memory Allocation

__constant

Memory that has been declared using the __constant keyword is also located in DRAM

This data is different from global memory

- Read-only
- Being cached!

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

Note: we can also declare
const __global float* x
this is different from __constant memory!
GPU Memory Allocation

Shared Memory

The memory per SM is shared by the l1 cache and the shared memory.

Shared memory can be accessed by all threads in a block (or work group)

Lasts while the block is ‘alive’

We may have various configurations:

- 128 KB L1 + 0 KB Shared Memory
- 120 KB L1 + 8 KB Shared Memory
- 112 KB L1 + 16 KB Shared Memory
- 96 KB L1 + 32 KB Shared Memory
- 64 KB L1 + 64 KB Shared Memory
- 28 KB L1 + 100 KB Shared Memory
- 16 KB L1 + 112 KB Shared Memory
GPU Memory Allocation

__local

Memory that has been declared using the __local keyword is stored in shared memory

This memory is accessible by all threads within the same block

Main purpose: “cache” coherent data between threads in a work group for quicker look-up

__kernel void kernel_func(...) {
    __local float localData[1024];
    ...
}

// Host
clSetKernelArg(kernel, 0, sizeof(cl_float) * length, NULL);

// Device
__kernel void kernel_func(__local float* localData) {
    ...
}
GPU Memory Allocation

__local

OpenCL provides a function to asynchronously prefetch data from global memory into local memory or write back to global memory:

async_work_group_copy(...)

We can then use the ‘wait_group_events(...)’ function to wait for the event to finish.

Warps can only continue beyond the wait_group_events(...) call after each thread in a work group has encountered this call.
GPU Memory Allocation

Register Memory

Only visible to the thread that wrote this data

Lasts as long as that the thread is ‘alive’

0 – 1 cycles
GPU Memory Allocation

__private

Memory that has been declared using the __private keyword is stored in the registers

Variables declared within a kernel without any address qualifier are automatically stored within the register-space

Remember: we have about 64K registers available per SM

__kernel void mulArrays(__global float* a, __global float* b, __global float* c, int n) {
    int id = get_global_id(0);
    if (id >= n) return;
    c[id] = a[id] * b[id];
}
GPU Memory Allocation

Local Memory

Our local memory is also located in DRAM*

Similar to register memory (only visible to the thread)

Implicitly allocated

Used as an overflow buffer for our registers

Very slow, but cached!

*DRAM
GPU Memory Allocation

Memory

- Caching is implicit, but works differently from the CPU*
- On the GPU we must explicitly indicate where we want to store our data
- Data locations have different levels of accessibility

* [https://rastergrid.com/blog/gpu-tech/2021/01/understanding-gpu-caches/](https://rastergrid.com/blog/gpu-tech/2021/01/understanding-gpu-caches/)
INFOMOV – Lecture 7 – “GPGPU 2”

GPU Memory Allocation

Reads

Section 2.5: matrix multiplication using shared memory

Understanding GPU caches
https://rastergrid.com/blog/gpu-tech/2021/01/understanding-gpu-caches/
Today’s Agenda:

- Recap
- GPU Memory Architecture
- Memory Allocation
- Synchronization
Synchronization

Thread synchronization

It is important that we can synchronize our threads

Synchronization can be achieved within our kernels or from the host
Synchronization

Thread synchronization – Host

OpenCL uses command queues to enqueue and execute kernels

The host is responsible for creating queues, enqueueing kernels and executing kernels
Synchronization

Thread synchronization – Host

Command queues are executed asynchronously to the CPU

In a game engine for example we can concurrently render the previous frame, while updating the game-world for the current frame.

<table>
<thead>
<tr>
<th>Host</th>
<th>Update AI</th>
<th>Physics</th>
<th>Networking</th>
<th>Copy buffers</th>
<th>Enqueue kernels</th>
<th>Update AI</th>
<th>Physics</th>
<th>Networking</th>
<th>Copy buffers</th>
<th>Enqueue kernels</th>
</tr>
</thead>
<tbody>
<tr>
<td>Device</td>
<td>Render frame n-1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Render frame n</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

`time`
## Synchronization

### Thread synchronization – Host

Command queues are created via `clCreateCommandQueue(...)` function.

We can specify whether we want to execute the kernels in our queue in-order or out-of-order.

Multiple queues can be defined and executed simultaneously (e.g., a render queue and a particle queue).

### Diagram

<table>
<thead>
<tr>
<th>frame n</th>
<th>frame n+1</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Host</strong></td>
<td><strong>Update AI</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Physics</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Networking</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Copy buffers</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Enqueue kernels</strong></td>
</tr>
<tr>
<td><strong>Update AI</strong></td>
<td><strong>Physics</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Networking</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Copy buffers</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Enqueue kernels</strong></td>
</tr>
<tr>
<td><strong>Queue A</strong></td>
<td><strong>Render frame (n-1)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Update particles (n)</strong></td>
</tr>
<tr>
<td><strong>Queue B</strong></td>
<td><strong>Render frame (n)</strong></td>
</tr>
<tr>
<td></td>
<td><strong>Update particles (n + 1)</strong></td>
</tr>
</tbody>
</table>

**Survive:** SurviveProbability, diffuse | estimation: doing it properly, clearly

**Radiance:** SampleLight, Ambient, I, AI, IBl, IBL, x + radiance, y + radiance, z > 0

E: (weight + costThetaRef) / directTheta

Random walk: done properly, closely follow time
Synchronization

Thread synchronization – Host

Queues run asynchronously to the host; we cannot know when a queue has finished

OpenCL implements the clFinish(queue) function to solve this issue
Synchronization

Thread synchronization – Device

OpenCL also provides some tools to synchronize from within the Device

- Atomics
- Volatile (not discussed today)
- Barrier/memory fences (not discussed today)
Synchronization

Thread synchronization – Atomics

It is not uncommon in GPGPU to queue data requests or other data for further processing.

Let’s consider the example on the right:

```c
__kernel void traverse_scene(
    __global int* indices,
    __global int* counter,
    /* other params */
) {
    int idx = get_global_id(0);
    // Traverse scene
    while (traverse(idx, ...)) {
        // Shading
    }
}
```

```c
int idx = get_global_id(0);
// Traverse scene
while (traverse(idx, ...)) {
    // Shading
}
```
**Synchronization**

Thread synchronization – Atomics

It is not uncommon in GPGPU to queue data requests or other data for further processing

Instead: use compaction to avoid gaps in warps

What goes wrong in this example?

We introduced a racing condition

```c
__kernel void _traverse_scene(
    __global int* indices,
    __global int* counter;
    /* other params */) {
    // Fetch index from array.
    int idx = get_global_id(0);
    idx = indices[idx];
    // Trave scene
    if (traverse(idx, ...)) {
        // Store ray in array.
        indices[*counter++] = idx;
        return;
    }
    // Shading
}
```
Synchronization

Thread synchronization – Atomics

Similar to CPU multithreading we may encounter racing conditions between threads

On a GPU this is even worse, why?

We have 32 threads in a warp that execute the same instruction at the same cycle!

```c
__kernel void _traverse_scene(
    __global int* indices,
    __global int* counter,
    /* other params */
) {

    // Fetch index from array.
    int idx = get_global_id(0);
    idx = indices[idx];

    // Traverse scene
    if (traverse(idx, ...)) {
        // Store ray in array.
        indices[*counter++] = idx;
        return;
    }

    // Shading
}
```
Synchronization

Thread synchronization – Atomics

GPUs employ atomics to deal with global racing conditions

Atomic operations remove racing conditions between threads when working on the same data

Many atomic operations available:

- Increment
- Subtraction
- Exchange
- Addition
- Decrement
- Compare-exchange
- Min
- Max
- And
- OR
- XOR

```c
__kernel void _traverse_scene(
    __global int* indices,
    __global int* counter,
    /* other params */)
{
    int idx = get_global_id(0);
    idx = indices[idx];
    if (traverse(idx, ...)) {
        indices[*counter++] = idx;
        return;
    }
}
```
Synchronization

Thread synchronization – Atomics

We may solve this problem using atomic increment function

We have no avoided any global racing conditions

```c
__kernel void _traverse_scene(
  __global int* indices,
  __global int* counter,
  /* other params */ ) {

  // Fetch index from array.
  int idx = get_global_id( 0 );
  idx = indices[idx];

  // Traverse scene
  if (traverse(idx, ...))
  {
    // Store ray in array.
    int i = atomic_inc( counter );
    indices[i] = idx;
    return;
  }

  // Shading
}
```
Today's Agenda:

- Recap
- GPU Memory Architecture
- Memory Allocation
- Synchronization
# GPU Memory Overview

## Overview GPU memory

<table>
<thead>
<tr>
<th>Location</th>
<th>Accessibility</th>
<th>Size*</th>
<th>Latency (cycles)</th>
<th>Cached</th>
<th>Constant</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>Thread</td>
<td>64k 32-bit registers per SM</td>
<td>0~1</td>
<td>-</td>
<td>No</td>
<td>On Streaming Multiprocessor</td>
</tr>
<tr>
<td>Local</td>
<td>Thread</td>
<td>-</td>
<td>400~600</td>
<td>Yes</td>
<td>No</td>
<td>DRAM</td>
</tr>
<tr>
<td>Shared</td>
<td>Block</td>
<td>0KB – 100KB per SM</td>
<td>1~32</td>
<td>-</td>
<td>No</td>
<td>L1 data-cache</td>
</tr>
<tr>
<td>Global</td>
<td>All threads + host</td>
<td>Multiple GB</td>
<td>400~600</td>
<td>No</td>
<td>No</td>
<td>DRAM</td>
</tr>
<tr>
<td>Texture</td>
<td>All threads + host</td>
<td>Multiple GB</td>
<td>400~600</td>
<td>Yes</td>
<td>Yes</td>
<td>DRAM</td>
</tr>
<tr>
<td>Constant</td>
<td>All threads + host</td>
<td>64KB</td>
<td>400~600</td>
<td>Yes</td>
<td>Yes</td>
<td>DRAM</td>
</tr>
</tbody>
</table>

*These values vary per device.
END of “GPGPU (2)”
Demonstration

In the working lecture we will have a look at memory in the Mandelbrot application from last week.