Welcome!
Today’s Agenda:

- Recap
- OpenCL Programming Model
  - Kernel execution
  - Memory allocation
  - Thread synchronization
- GPU profiling
Recap GPGPU-1

Hierarchical order

TPC

PolyMorph Engine
Recap GPGPU-1

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 $16 \times 2 \times 4 = 128$ cores that can operate simultaneously on FP data.
Recap GPGPU-1

Explicit memory allocation

- Register memory
- Shared memory
- Global memory
- Local memory
- Texture memory
- Constant memory
Recap GPGPU-1

Explicit memory allocation

<table>
<thead>
<tr>
<th>Location</th>
<th>Accessibility</th>
<th>Size*</th>
<th>Latency (cycles)</th>
<th>Cached</th>
<th>Constant</th>
<th>Location (on/off chip)</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</td>
</tr>
<tr>
<td>Local</td>
<td>Thread</td>
<td>?</td>
<td>400~600</td>
<td>Yes</td>
<td>No</td>
<td>Off</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>On (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>Off</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>Off</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>Off</td>
</tr>
</tbody>
</table>

*These values may vary per device!
Recap GPGPU-1

Quick overview of thread execution

Thread or work-item

Scalar processor

0 1 2

0 1 2

block or work-group

SMs

Grid of blocks/work-groups

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 GPGPU-1

SIMT Execution

- Threads are executed in groups of 32 called ‘warps’ (wavefronts on AMD)
- Warps execute in lockstep
- At each cycle, all threads in a warp must execute the same instruction with different data

<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>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>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>

/* https://en.wikipedia.org/wiki/Xorshift */

```c
float rnd(uint seed) {
    seed ^= seed << 13; // i0
    seed ^= seed >> 17; // i1
    seed ^= seed << 5;  // i2
    return seed / UINT_MAX; // i3
}
```
Recap GPGPU-1

SIMT Execution

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>
<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>
Introduction

CPU vs. GPU

**CPUs hide latencies via**
- Super-scalar execution
- Out-of-order execution
- Branch prediction
- Cache hierarchy
- Speculative prefetching

**GPUs hide latencies via**
- Swapping blocked warps
Today's Agenda:

- Recap
- **OpenCL Programming Model**
  - Kernel execution
  - Memory allocation
  - Thread synchronization
- GPU profiling
OpenCL programming model

GPGPU programming

- OpenGL/Vulkan (compute shaders)
- CUDA (Nvidia)
- OpenCL (cross-platform)
OpenCL programming model

GPGPU programming

Today we will discuss GPGPU programming using OpenCL

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

Note: this is not a lecture on OpenCL! Today we discuss how the architecture of a GPU translates to GPGPU programming.
OpenCL programming model

Host and Device

The host is the component that controls our device: CPU

The device is the component that performs our operations: GPU

The host is responsible for setting up and maintaining our GPU application:

- Selecting a compatible device (GPU)
- Reserving/releasing memory
- Creating Kernels
- ‘Managing’ execution queues
OpenCL programming model

Setting up our environment

Before executing GPGPU code, we must first prepare a programming environment:

- Select platform (NVIDIA, AMD, Intel)  \( \text{clGetPlatformIDs}(\ldots) \)
- Select a device from the platform  \( \text{clGetDeviceIDs}(\ldots) \)
- Setup a context with this device  \( \text{clCreateContext}(\ldots) \)
OpenCL programming model

Next, we can create all that is necessary for running our GPGPU code using our context:

- Loading and compiling our OpenCL code
- Create kernels
- Reserve memory
- Create command queues
- Enqueuing commands

More on these later in the lecture
Today's Agenda:

- Recap
- OpenCL Programming Model
  - Kernel execution
  - Memory allocation
  - Thread synchronization
- GPU profiling
OpenCL programming model

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

In the previous lectures we have seen how to convert our scalar code to SIMD code

Example: array multiplication

```c
void mulArrays(float* a, float* b, float* c, int N) {
    __m128 *a4 = (__m128*)a;
    __m128 *b4 = (__m128*)b;
    __m128 *c4 = (__m128*)c;
    for (int i = 0; i < N / 4; i++)
        c4[i] = _mm_mul_ps (a4[i], b4[i]);
}
```
OpenCL programming model

Kernel execution

For SIMT 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];
}
```
OpenCL programming model

Kernel execution

Array multiplication SIMD

```c
void mulArrays(float* a, float* b, float* c, int N) {
  __m128 *a4 = (__m128*)a;
  __m128 *b4 = (__m128*)b;
  __m128 *c4 = (__m128*)c;
  for (int i = 0; i < N / 4; i++)
    c4[i] = _mm_mul_ps (a4[i], b4[i]);
}
```

Array multiplication SIMT

- No loop body
- Get index from thread id

```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];
}
```
OpenCL programming model

Kernel execution

In GPGPU we write kernels that are executed per-thread

- We pass parameters from host
- Explicitly allocate memory
- 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
...
```

```
//pass kernel arguments.
clSetKernelArg(kernel, 0, sizeof(cl_mem), a_buffer);
clSetKernelArg(kernel, 1, sizeof(cl_mem), b_buffer);
clSetKernelArg(kernel, 2, sizeof(cl_mem), c_buffer);
...
clEnqueueNDRangeKernel(..., kernel, ..., global_size, local_size, ...);
```
OpenCL programming model

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.
- **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!
OpenCL programming model

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, local size – 1]
3. **get_group_id(int dim)**, retrieves the work-group (block) id within the grid [0, local size]
OpenCL programming model

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:

`clGetDeviceInfo(device, CL_DEVICE_MAX_WORK_GROUP_SIZE, ...);`
OpenCL programming model

Kernel execution

The thread-scheduler swaps blocked-warps by active warps to hide latencies

Must we always assign the largest possible number of threads per block?

No: threads within a block share resources!
OpenCL programming model

Kernel execution

A block is always executed on the same streaming multiprocessor (SM)

Three important properties:
1. The warp-schedulers cycle between active warps within the blocks assigned to the SM
2. All threads in a block share the available registers on a SM
3. All threads in a block can access the same shared memory and share the same l1-cache
OpenCL programming model

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)

So how do we select the optimal number of threads per block?

No mathematical model, find a balance by tweaking parameters (profile performance!)
Today's Agenda:

- Recap
- **OpenCL Programming Model**
  - Kernel execution
  - Memory allocation
  - Thread synchronization
- GPU profiling
OpenCL programming model

Memory allocation

On a GPU we have seen several memory locations:

- DRAM, l2-cache, l1-cache/shared memory, and registers

We must explicitly indicate where to store our data
OpenCL programming model

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
OpenCL programming model

__private

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

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

__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];
}
OpenCL programming model

__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: retrieve data from global memory, work on it, and write back.

__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) {
  ...
}
OpenCL programming model

__local

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

```c
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.
OpenCL programming model

__global

Memory that has been declared using the __global keyword is stored in 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;
}
OpenCL programming model

__constant

Memory that has been declared using the __constant keyword is also stored 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!
OpenCL programming model

Memory allocation

Not only threads can read and write to global memory, but the host can as well.

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:

```c
clReleaseMemObject(buffer);
```
OpenCL programming model

Memory allocation

Section 2.5: matrix multiplication using shared memory
Today’s Agenda:

- Recap
- OpenCL Programming Model
  - Kernel execution
  - Memory allocation
  - Thread synchronization
- GPU profiling
OpenCL programming model

Thread synchronization

Sometimes it is important that we can synchronize our threads

Synchronization can be achieved within our kernels or from the host
OpenCL programming model

Thread synchronization – Host

OpenCL uses command queues to enqueue and execute kernels

The host is responsible for creating queues, enqueuing kernels and executing kernels
OpenCL programming model

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>Device</th>
<th>Render frame n-1</th>
<th>Frame n+1</th>
<th>Render frame n</th>
</tr>
</thead>
</table>

Time
OpenCL programming model

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)
OpenCL programming model
OpenCL programming model

Thread synchronization - Host

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

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

Thread synchronization - Device

OpenCL also provides some tools to synchronize from within the Device

- **Volatile**
- **Barrier/memory fences**
- **Atomics**
OpenCL programming model

Volatile

In the example we do two memory accesses

The compiler optimizes memory-access by re-using old memory-accesses

ref2 will never become 2!

```c
// myArray is an array of non-zero integers
__global int myArray[...];
...
__kernel void myKernel(__global int* result) {
    int id = get_global_id( 0 );
    int ref1 = myArray[tid] * 1;
    myArray[tid + 1] = 2;
    int ref2 = myArray[tid] * 1;
    result[tid] = ref1 * ref2;
}
```
**OpenCL programming model**

**Volatile**

We can change this behavior by declaring ‘myArray’ as volatile

The compiler assumes that any variable declared as volatile, can be changed by any thread at any given moment

```c
__volatile__ __global__ int myArray[...];
...
__kernel void myKernel(__global int* result) {
    int id = get_global_id(0);
    int ref1 = myArray[tid] * 1;
    myArray[tid + 1] = 2;
    int ref2 = myArray[tid] * 1;
    result[tid] = ref1 * ref2;
}
```
OpenCL programming model

Barriers and memory fences

There is another problem! Remember: warps execute in lockstep.

Threads from different warps have no guarantee to finish the same instruction before continuing!

The first thread in the warp is not guaranteed to read the correct value for ref2 as thread[tid – 1] is in a different warp!

```c
// myArray is an array of non-zero integers
__volatile__ __global int myArray[...];
...
__kernel void myKernel(__global int* result) {
  int id = get_global_id(0);
  int ref1 = myArray[tid] * 1;
  myArray[tid + 1] = 2;
  int ref2 = myArray[tid] * 1;
  result[tid] = ref1 * ref2;
}
```
Barriers and memory fences

OpenCL implements barriers and fences to synchronize within a work-group

- `mem_fence(cl_mem_fence_flags flags)`
- `read_mem_fence(cl_mem_fence_flags flags)`
- `write_mem_fence(cl_mem_fence_flags flags)`
- `barrier(cl_mem_fence_flags flags)`

The fences and barrier are a bit different from each other

```c
__volatile __global int myArray[...];
...
__kernel void myKernel(__global int* result) {
    int id = get_global_id(0);
    int ref1 = myArray[tid] * 1;
    myArray[tid + 1] = 2;
    int ref2 = myArray[tid + 1] = 2;
    int ref2 = myArray[tid] * 1;
    result[tid] = ref1 * ref2;
}
```
OpenCL programming model

Barriers and memory fences

Memory fences enforce that all read and/or writes preceding the fence will be committed to memory before any loads and stores following fence

Threads continue to execute regularly until a memory operation is encountered

- mem_fence(cl_mem_fence_flags flags)
- read_mem_fence(cl_mem_fence_flags flags)
- write_mem_fence(cl_mem_fence_flags flags)
OpenCL programming model

Barriers and memory fences

All work-items in a work-group must execute this function, before any are allowed to continue execution beyond the barrier!

The barrier enforces a memory fence for the specified address space and ensures the correct ordering of memory operations

As opposed to fences, all threads block all execution until all threads within the block reached the barrier!

- barrier(cl_mem_fence_flags flags)

```
// myArray is an array of non-zero integers
__volatile __global int myArray[...];
...
__kernel void myKernel(__global int* result) {
    int id = get_global_id(0);
    int ref1 = myArray[tid] * 1;
    myArray[tid + 1] = 2;
    int ref2 = myArray[tid] = 1;
    result[tid] = ref1 * ref2;
}
```
OpenCL programming model

Thread synchronization

We must pass an argument to the barrier and fence calls to specify for which type of memory operations to enforce a barrier

- CLK_LOCAL_MEM_FENCE
- CLK_GLOBAL_MEM_FENCE

```c
volatile __global int myArray[...];
...
__kernel void myKernel(__global int* result) {
    int id = get_global_id(0);
    int ref1 = myArray[tid] * 1;
    myArray[tid + 1] = 2;
    int ref2 = myArray[tid] * 1;
    result[tid] = ref1 * ref2;
}
```
Thread synchronization

How do we solve the synchronization issues here?

- Where do we place the barriers/fence?
- Which type of fence/barrier do we use?
- Which address space must be used?

However, we did not actually solve our problem...

Memory fences and barriers only apply to a work-group. There is no build-in method to synchronize between work-groups

// myArray is an array of non-zero integers
__volatile __global int myArray[...];
...
__kernel void myKernel(__global int* result) {
    int id = get_global_id(0);
    int ref1 = myArray[tid] * 1;
    myArray[tid + 1] = 2;
    int ref2 = myArray[tid] * 1;
    result[tid] = ref1 * ref2;
}
OpenCL programming model

Atomics

GPUs can also perform atomic operations

GPUs are optimized to perform atomic operations!

We can use atomics e.g., to enqueue new tasks or to create our own barrier

For an overview of the available atomic operations, see:
https://www.khronos.org/registry/OpenCL/sdk/1.2/docs/man/xhtml/atomicFunctions.html

```
// myArray is an array of non-zero integers
__volatile__ __global__ int myArray[...];
...
__kernel void myKernel(__global int* result){
    int id = get_global_id(0);
    int ref1 = myArray[tid] * 1;
    atomic_inc(counter);
    while (*counter < NUM_THREADS) {}
    myArray[tid + 1] = 2;
    atomic_inc(counter);
    while (*counter < (NUM_THREADS * 2) - 1) {}
    int ref2 = myArray[tid] * 1;
    result[tid] = ref1 * ref2;
}
```
OpenCL programming model

Thread synchronization

Synchronization using atomics in this example is silly of course!

It is best to separate the kernel in multiple smaller kernels and enqueue these in-order.
OpenCL programming model

- We write kernels to execute GPGPU code
- Kernels are enqueued by specifying a local and a global work size
- Local work-sizes correspond to the threads in a block which are executed per SM
- We explicitly specify where to store our memory: __private, __local, __global, __constant
- OpenCL offers various methods to synchronize between threads:
  - Command queues
  - Volatile
  - Memory fences and buffers
  - Atomic operations
Today's Agenda:

- Recap
- OpenCL Programming Model
  - Kernel execution
  - Memory allocation
  - Thread synchronization
- GPU profiling
GPU Profiling

GPU profiling is different from CPU profiling. On the CPU, we have hundreds of metrics that we can measure.
GPU Profiling

GPU profiling is different from CPU profiling

On the CPU we have hundreds of metrics that we can measure

A GPU does not have branch prediction, extensive caching, SIMD, superscalar pipelines, etc., therefore there are fewer metrics to profile
OpenCL programming model

GPU Profiling

Useful metrics that we can measure are

- Kernel execution duration
- Host-device memory operations
- Memory usage
OpenCL programming model

GPU Profiling

There are various available tools for profiling our GPU code

- AMD’s GPU Open profiler (https://gpuopen.com/rgp/)
- Nvidia’s Nsight and nvprof
- OpenCL profiler: CLtracer (1-month free trial)

OpenCL enables us to measure our own performance
GPU Profiling

We can setup our own profiling calls by using events:

1. Enable profiling when creating our command queue
   
   ```
   cl_command_queue queue = clCreateCommandQueue(context, device, CL_QUEUE_PROFILING_ENABLE, &err);
   ```

2. Include an event when enqueuing operations
   
   ```
   cl_event profEvent;
   clEnqueueNDRangeKernel (queue, kernel, wdim, 0, gsize, lsize, 0, 0, &profEvent);
   ```

3. Wait for the queue to finish
   
   ```
   clFinish(queue); // OR wait for event to finish: clWaitForEvents(1, &profEvent);
   ```

4. Retrieve the profiling information
   
   ```
   cl_ulong startTime, endTime;
   clGetProfilingInfo(profEvent, CL_PROFILING_COMMAND_START, sizeof(cl_ulong), &startTime, NULL);
   clGetProfilingInfo(profEvent, CL_PROFILING_COMMAND_END, sizeof(cl_ulong), &startTime, NULL);
   double time = (endTime - startTime) * 1e-6; // convert from ns to ms
   ```
OpenCL programming model

GPU Profiling

We can retrieve four different values for our profiling information

- **CL_PROFILING_COMMAND_QUEUED**
  Time when the command was enqueued by the host.

- **CL_PROFILING_COMMAND_SUBMIT**
  Time when the command was submitted to the device

- **CL_PROFILING_COMMAND_START**
  Time when the command identified by the event starts

- **CL_PROFILING_COMMAND_END**
  Time when the command identified by the event finishes
OpenCL programming model

GPU Profiling

We will not go any deeper into GPU profiling during this course.

Most latencies occur from memory-access as these result in stalls for the warps in our work-groups.
Introduction
Practical

Today’s Practical: basic GPGPU conversion

1. Convert loop-body to kernel
2. Create device context and build application
   - If failed to compile, fix errors and retry
3. Create buffers
4. Pass parameters to kernel
5. Copy memory host-to-device (optional)
6. Enqueue kernels
7. Copy memory device-to-host (optional)