

# GPU Acceleration

Jonas Sys

Supervisors: Prof. Christophe Scholliers, Prof. Elisa Gonzalez Boix

November 22, 2024

# Outline

- 1 GPUs
- 2 CUDA Workflow
- 3 Example
  - Kernel
  - Code
- 4 Extras
  - Errors
  - Memory
  - Synchronization
  - CUDA and OpenGL

# Outline

## 1 GPUs

## 2 CUDA Workflow

## 3 Example

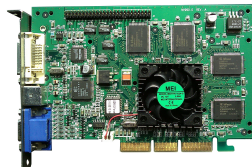
- Kernel
- Code

## 4 Extras

- Errors
- Memory
- Synchronization
- CUDA and OpenGL

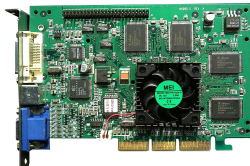
# GPUs

- Originally: accelerate graphics (for games)



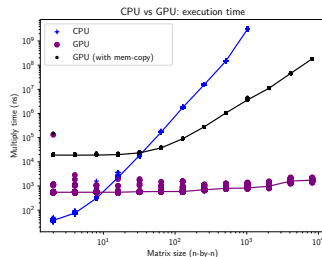
# GPUs

- Originally: accelerate graphics (for games)
- Programmable shaders: flexibility



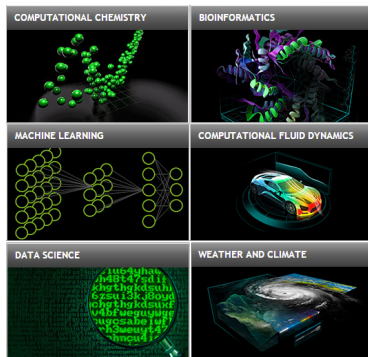
# GPUs

- Originally: accelerate graphics (for games)
- Programmable shaders: flexibility
- Much more performant than CPUs

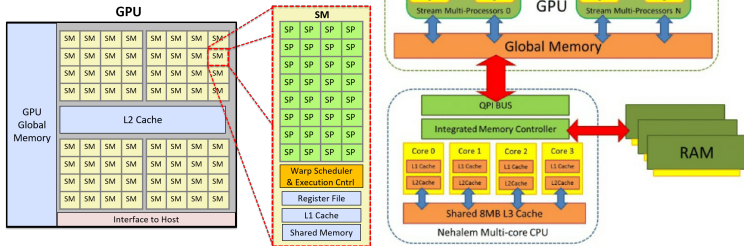


# GPUs

- Originally: accelerate graphics (for games)
- Programmable shaders: flexibility
- Much more performant than CPUs
- General purpose (CUDA, 2007)



# GPUs





# Outline

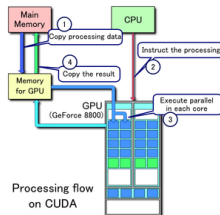
- 1 GPUs
- 2 **CUDA Workflow**
- 3 Example
  - Kernel
  - Code
- 4 Extras
  - Errors
  - Memory
  - Synchronization
  - CUDA and OpenGL

# CUDA Workflow

- Identify parallel parts of workload

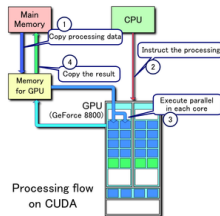
# CUDA Workflow

- Identify parallel parts of workload
- Allocate input/output buffers on GPU



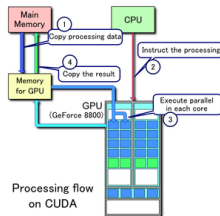
# CUDA Workflow

- Identify parallel parts of workload
- Allocate input/output buffers on GPU
- Copy input from CPU to GPU



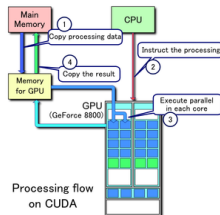
# CUDA Workflow

- Identify parallel parts of workload
- Allocate input/output buffers on GPU
- Copy input from CPU to GPU
- Start GPU computation



# CUDA Workflow

- Identify parallel parts of workload
- Allocate input/output buffers on GPU
- Copy input from CPU to GPU
- Start GPU computation
- Copy output from GPU to CPU

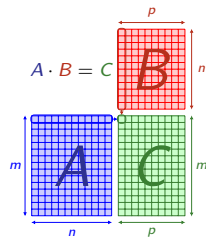


# Outline

- 1 GPUs
- 2 CUDA Workflow
- 3 Example**
  - Kernel
  - Code
- 4 Extras
  - Errors
  - Memory
  - Synchronization
  - CUDA and OpenGL

# Example

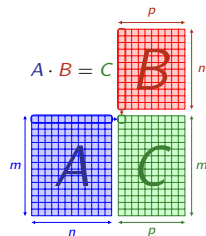
- Naive matrix multiply





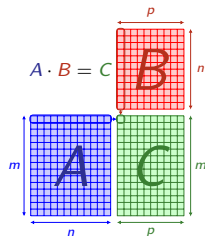
# Example

- Naive matrix multiply
- Each element in  $C$  is independently computed



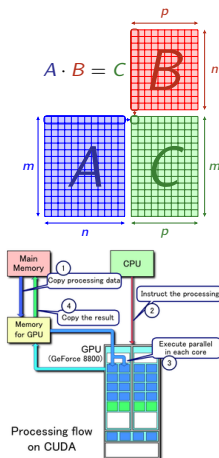
# Example

- Naive matrix multiply
- Each element in  $C$  is independently computed
- Steps:
  - 1 Starting C++ code



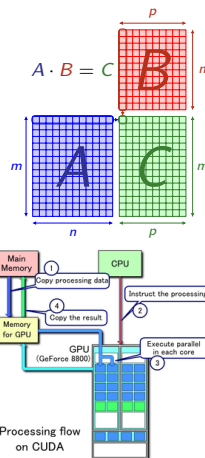
# Example

- Naive matrix multiply
- Each element in  $C$  is independently computed
- Steps:
  - 1 Starting C++ code
  - 2 Allocate GPU memory



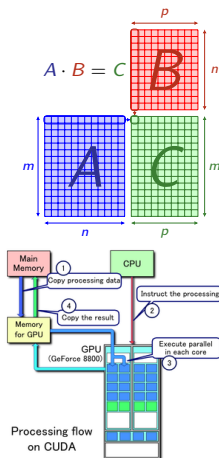
# Example

- Naive matrix multiply
- Each element in  $C$  is independently computed
- Steps:
  - 1 Starting C++ code
  - 2 Allocate GPU memory
  - 3 Copy  $A$ ,  $B$  to GPU



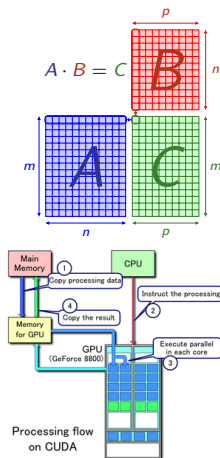
# Example

- Naive matrix multiply
- Each element in  $C$  is independently computed
- Steps:
  - 1 Starting C++ code
  - 2 Allocate GPU memory
  - 3 Copy  $A$ ,  $B$  to GPU
  - 4 Start kernel (using  $m \cdot p$  GPU threads)



# Example

- Naive matrix multiply
- Each element in  $C$  is independently computed
- Steps:
  - 1 Starting C++ code
  - 2 Allocate GPU memory
  - 3 Copy  $A$ ,  $B$  to GPU
  - 4 Start kernel (using  $m \cdot p$  GPU threads)
  - 5 Copy  $C$  to CPU



# Kernel

- CUDA needs to know where functions are run

# Kernel

- CUDA needs to know where functions are run  
    Host Only on CPU (“normal” functions)



# Kernel

- CUDA needs to know where functions are run
  - Host** Only on CPU (“normal” functions)
  - Device** Only on GPU

# Kernel

- CUDA needs to know where functions are run
  - Host** Only on CPU (“normal” functions)
  - Device** Only on GPU
  - Global** Special case: entry point from CPU to GPU

# Kernel

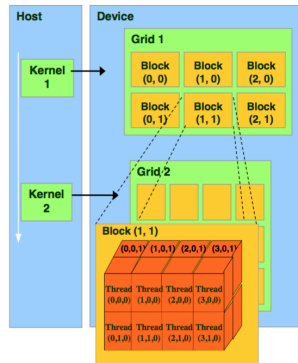
- CUDA needs to know where functions are run
  - Host** Only on CPU (“normal” functions)
  - Device** Only on GPU
  - Global** Special case: entry point from CPU to GPU
- Host and device can be combined (same code runs on CPU and GPU)

# Kernel

- Call from CPU to GPU using global function

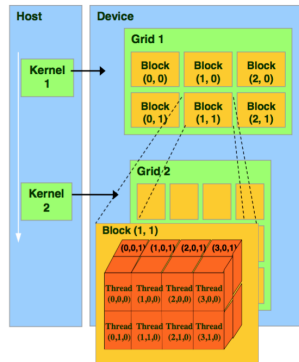
# Kernel

- Call from CPU to GPU using global function
- When calling, specify number of threads



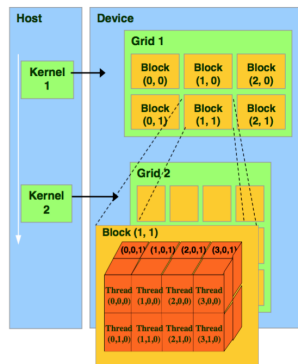
# Kernel

- Call from CPU to GPU using global function
- When calling, specify number of threads
  - Threads are grouped in blocks



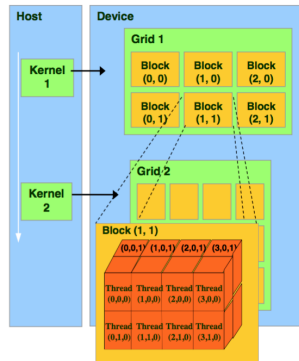
# Kernel

- Call from CPU to GPU using global function
- When calling, specify number of threads
  - Threads are grouped in blocks
  - All blocks form one grid



# Kernel

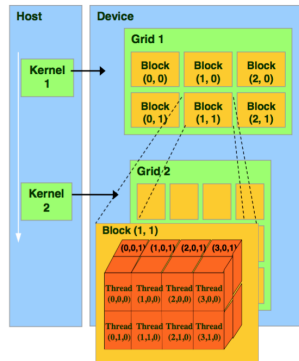
- Call from CPU to GPU using global function
- When calling, specify number of threads
  - Threads are grouped in blocks
  - All blocks form one grid
  - Sizes can be specified in 1D, 2D, or 3D





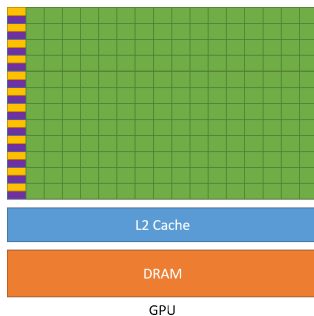
# Kernel

- Call from CPU to GPU using global function
- When calling, specify number of threads
  - Threads are grouped in blocks
  - All blocks form one grid
  - Sizes can be specified in 1D, 2D, or 3D
  - At runtime: 32 threads form a warp



# Kernel

- Call from CPU to GPU using global function
- When calling, specify number of threads
  - Threads are grouped in blocks
  - All blocks form one grid
  - Sizes can be specified in 1D, 2D, or 3D
  - At runtime: 32 threads form a warp
  - All warps: same instruction

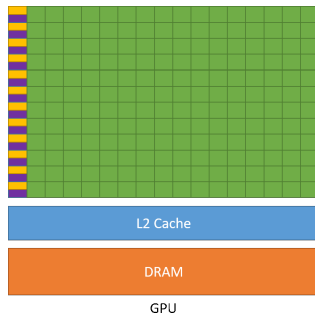


# Kernel

- Call from CPU to GPU using global function
- When calling, specify number of threads
  - At runtime: 32 threads form a warp
  - All warps: same instruction

## Branches

For performance: make sure all warps are full, i.e. threads are taking the same branch in groups of 32. This can be across warps, since they will be re-grouped if need be.



# Starting Code

```
1  struct matrix {
2      float &at(int row, int col) {
3          return data[row * cols + col];
4      }
5      float *data; int rows; int cols;
6  };
7  void dot(matrix a, matrix b, matrix c, int row, int col) {
8      c.at(row, col) = 0;
9      for(int k = 0; k < a.cols; k++)
10         c.at(row, col) += a.at(row, k) * b.at(k, col);
11  }
12  void matmul_cpu(matrix a, matrix b, matrix c) {
13      for(int row = 0; row < c.rows; row++) {
14          for(int col = 0; col < c.cols; col++)
15             dot(a, b, c, row, col);
16      }
17  }
18  // ...
100 matrix a, b, c;
101 // Initialize a, b here
102 // ...
103 matmul_cpu(a, b, c);
```

# Steps

Recall steps from before:

- 1 Starting C++ code

# Steps

Recall steps from before:

- 1 Starting C++ code
- 2 Allocate GPU memory

# Steps

Recall steps from before:

- 1 Starting C++ code
- 2 Allocate GPU memory
- 3 Copy  $A$ ,  $B$  to GPU

# Steps

Recall steps from before:

- 1 Starting C++ code
- 2 Allocate GPU memory
- 3 Copy  $A$ ,  $B$  to GPU
- 4 Start kernel (using  $m \cdot p$  GPU threads)

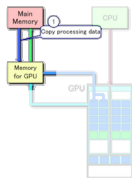


# Steps

Recall steps from before:

- 1 Starting C++ code
- 2 Allocate GPU memory
- 3 Copy  $A$ ,  $B$  to GPU
- 4 Start kernel (using  $m \cdot p$  GPU threads)
- 5 Copy  $C$  to CPU

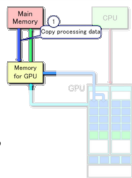
# Allocate GPU memory



```
100 matrix a, b, c;
101 // Assume a, b, c are initialized
102 matrix a_gpu, b_gpu, c_gpu;
103 a_gpu.rows = a.rows; a_gpu.cols = a.cols;
104 cudaMalloc(&a_gpu.data, a.rows * a.cols * sizeof(float));
105 b_gpu.rows = b.rows; b_gpu.cols = b.cols;
106 cudaMalloc(&b_gpu.data, b.rows * b.cols * sizeof(float));
107 c_gpu.rows = c.rows; c_gpu.cols = c.cols;
108 cudaMalloc(&c_gpu.data, c.rows * c.cols * sizeof(float));
```

`cudaMalloc(void **devPtr, size_t bytes)` allocates bytes bytes of GPU memory, storing the pointer in devPtr.

# Copy $A$ , $B$ to GPU



```
109 cudaMemcpy(a_gpu.data, a.data, a.rows * a.cols * sizeof(float),  
    ↪ cudaMemcpyDefault);  
110 cudaMemcpy(b_gpu.data, b.data, b.rows * b.cols * sizeof(float),  
    ↪ cudaMemcpyDefault);
```

`cudaMemcpy(void *dst, const void *src, size_t bytes, cudaMemcpyKind kind)` copies bytes bytes from src to dst. The final argument, `kind`, specifies the direction.

On recent CUDA versions (CUDA 4+), this can be inferred using `cudaMemcpyDefault`.

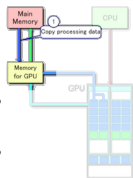
# Copy A, B to GPU

```
109 cudaMemcpy(a_gpu.data, a.data, a.rows * a.cols * sizeof(float),  
    ↪ cudaMemcpyDefault);  
110 cudaMemcpy(b_gpu.data, b.data, b.rows * b.cols * sizeof(float),  
    ↪ cudaMemcpyDefault);
```

`cudaMemcpy(void *dst, const void *src, size_t bytes, cudaMemcpyKind kind)` copies bytes bytes from src to dst. The final argument, kind, specifies the direction.

On older CUDA versions, this should be one of

- `cudaMemcpyHostToDevice`: CPU to GPU
- `cudaMemcpyDeviceToHost`: GPU to CPU
- `cudaMemcpyDeviceToDevice`: GPU to GPU



# Start kernel

```

7  // TODO: we will implement this later
8  __global__ void matmul_gpu(matrix a, matrix b, matrix c);

...

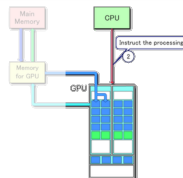
112 // specify grid (thread and block sizes)
113 dim3 threads = dim3(1024, 1024); // CUDA prefers powers of two
114 dim3 blocks = dim3(c.rows / 1024 + 1, c.cols / 1024 + 1);
115 // launch kernel
116 matmul_gpu<<<blocks, threads>>>(a_gpu, b_gpu, c_gpu);
117 cudaDeviceSynchronize();

```

The `__global__` attribute specifies `matmul_gpu` as a global (CPU to GPU) function.

Recall that we had

- CPU-only (host) functions (using `__host__`)
- GPU-only (device) functions (using `__device__`)
- GPU to GPU entry point functions (using `__global__`)



# Start kernel

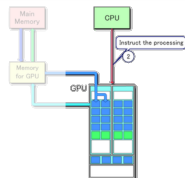
```

7  // TODO: we will implement this later
8  __global__ void matmul_gpu(matrix a, matrix b, matrix c);

...

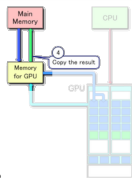
112 // specify grid (thread and block sizes)
113 dim3 threads = dim3(1024, 1024); // CUDA prefers powers of two
114 dim3 blocks = dim3(c.rows / 1024 + 1, c.cols / 1024 + 1);
115 // launch kernel
116 matmul_gpu<<<blocks, threads>>>(a_gpu, b_gpu, c_gpu);
117 cudaDeviceSynchronize();

```



- Invoking global functions is always done by the <<<blocks, threads>>> syntax.
- Make sure none of the arguments have pointers to CPU memory!
- Kernel launches are asynchronous, we use `cudaDeviceSynchronize()` to have the CPU wait until the GPU is finished.

# Copy C to CPU



```
118 cudaMemcpy(c.data, c_gpu.data, c.rows * c.cols * sizeof(float),  
    ↪ cudaMemcpyDefault);
```

Same as before, `cudaMemcpy(void *dst, const void *src, size_t size, cudaMemcpyKind kind)` copies data.

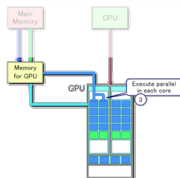
Again, `cudaMemcpyDefault` is used to infer direction (older systems should use `cudaMemcpyDeviceToHost`).

# Implement kernel

```

1  struct matrix {
2      __host__ __device__ float &at(int row, int col) {
3          return data[row * cols + col];
4      }
5      float *data; int rows; int cols;
6  };
7  __host__ __device__ void dot(matrix a, matrix b, matrix c, int row, int
  ↪ col) {
8      c.at(row, col) = 0;
9      for(int k = 0; k < a.cols; k++)
10         c.at(row, col) += a.at(row, k) * b.at(k, col);
11  }

```



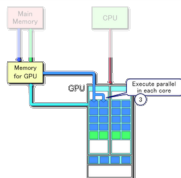
- We will use `matrix::at` and `dot` on both CPU and GPU
- To avoid code duplication, we make the compiler generate both from one definition
  - If we use `__host__`, we only get CPU
  - If we use `__device__`, we only get GPU
  - If we combine `__host__ __device__`, we get both



# Implement kernel

```
7  __global__ void matmul_gpu(matrix a, matrix b, matrix c) {  
8      int row = blockIdx.x * blockDim.x + threadIdx.x;  
9      int col = blockIdx.y * blockDim.y + threadIdx.y;  
10     if(row >= c.rows || col >= c.cols) // out-of-bounds  
11         return;  
12  
13     dot(a, b, c, row, col);  
14 }
```

- `matmul_gpu` has to be called from the CPU, but run on the GPU, so it should be `__global__`

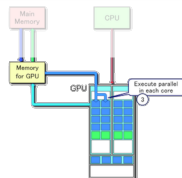


# Implement kernel

```

7  __global__ void matmul_gpu(matrix a, matrix b, matrix c) {
8      int row = blockIdx.x * blockDim.x + threadIdx.x;
9      int col = blockIdx.y * blockDim.y + threadIdx.y;
10     if(row >= c.rows || col >= c.cols) // out-of-bounds
11         return;
12
13     dot(a, b, c, row, col);
14 }

```



- We launched  $(\frac{c.rows}{1024} + 1, \frac{c.cols}{1024} + 1)$  blocks of (1024, 1024) threads each
- We can get indices using builtin variables:
  - `threadIdx` holds the 3D thread index within a block
  - `blockIdx` holds the 3D block index within the grid
  - `blockDim` holds the 3D size of the block
- These indices can be out of bounds, so we need to check!

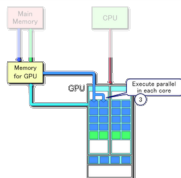
# Implement kernel

```

7  __global__ void matmul_gpu(matrix a, matrix b, matrix c) {
8      int row = blockIdx.x * blockDim.x + threadIdx.x;
9      int col = blockIdx.y * blockDim.y + threadIdx.y;
10     if(row >= c.rows || col >= c.cols) // out-of-bounds
11         return;
12
13     dot(a, b, c, row, col);
14 }

```

- By making dot (and matrix::at) available on the GPU (using `__device__`), we can simply re-use the code.



# Final code

```
1  struct matrix {
2      __host__ __device__ float &at(int row, int col) {
3          return data[row * cols + col];
4      }
5      float *data; int rows; int cols;
6  };
7  __host__ __device__ void dot(matrix a, matrix b, matrix c, int row, int
↪ col) {
8      c.at(row, col) = 0;
9      for(int k = 0; k < a.cols; k++)
10         c.at(row, col) += a.at(row, k) * b.at(k, col);
11 }
12 __global__ void matmul_gpu(matrix a, matrix b, matrix c) {
13     int row = blockIdx.x * blockDim.x + threadIdx.x;
14     int col = blockIdx.y * blockDim.y + threadIdx.y;
15     if(row >= c.rows || col >= c.cols) // out-of-bounds
16         return;
17
18     dot(a, b, c, row, col);
19 }
```

# Final code

```
100 matrix a, b, c;
101 // Assume a, b, c are initialized
102 matrix a_gpu, b_gpu, c_gpu;
103 a_gpu.rows = a.rows; a_gpu.cols = a.cols;
104 cudaMalloc(&a_gpu.data, a.rows * a.cols * sizeof(float));
105 b_gpu.rows = b.rows; b_gpu.cols = b.cols;
106 cudaMalloc(&b_gpu.data, b.rows * b.cols * sizeof(float));
107 c_gpu.rows = c.rows; c_gpu.cols = c.cols;
108 cudaMalloc(&c_gpu.data, c.rows * c.cols * sizeof(float));
109 cudaMemcpy(a_gpu.data, a.data, a.rows * a.cols * sizeof(float),
    ↪ cudaMemcpyDefault);
110 cudaMemcpy(b_gpu.data, b.data, b.rows * b.cols * sizeof(float),
    ↪ cudaMemcpyDefault);
111 // specify grid (thread and block sizes)
112 dim3 threads = dim3(1024, 1024); // CUDA prefers powers of two
113 dim3 blocks = dim3(c.rows / 1024 + 1, c.cols / 1024 + 1);
114 // launch kernel
115 matmul_gpu<<<blocks, threads>>>(a_gpu, b_gpu, c_gpu);
116 cudaDeviceSynchronize();
117 cudaMemcpy(c.data, c_gpu.data, c.rows * c.cols * sizeof(float),
    ↪ cudaMemcpyDefault);
```

# Outline

- 1 GPUs
- 2 CUDA Workflow
- 3 Example
  - Kernel
  - Code
- 4 Extras
  - Errors
  - Memory
  - Synchronization
  - CUDA and OpenGL

# Errors

- CUDA functions return `cudaError_t`, useful with `cudaGetErrorString()`

```
#define CHECK(x) do { \  
    const auto _ = (x); \  
    if(_ != cudaSuccess) { \  
        /* ... handle error using cudaGetErrorString(_) */ \  
    } \  
} while(false)
```

# Memory

- Manage GPU memory
- Allocate with `cudaMalloc(void **dev, size_t bytes)`
- Read/Write CPU-side with `cudaMemcpy(void *dst, const void *src, size_t bytes, cudaMemcpyKind kind)`
  - `cudaMemcpyDefault`: Inferred based on pointers (CUDA 4+)
  - `cudaMemcpyHostToDevice`: CPU → GPU
  - `cudaMemcpyDeviceToHost`: GPU → CPU
  - `cudaMemcpyDeviceToDevice`: GPU → GPU
  - `cudaMemcpyHostToHost`: CPU → CPU
- Unified Memory: `cudaMallocManaged(void **dev, size_t bytes)` (accessible on CPU & GPU)
- Free/Clean up with `cudaFree(void *dev)`



# Synchronization

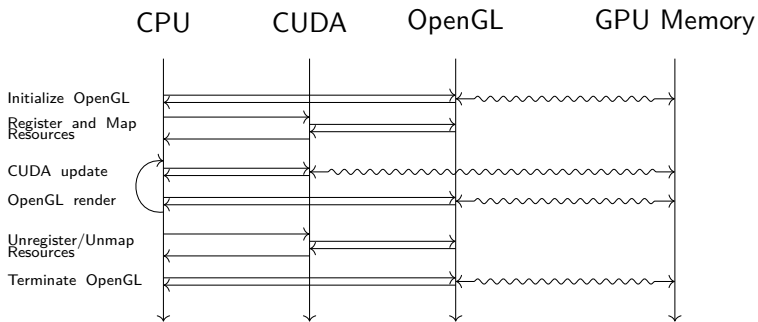
- Atomic functions like `atomicAdd`
- Across threads/warps in block: `__syncthreads()`
- Make CPU wait for GPU: `cudaDeviceSynchronize()`
- More advanced:
  - Selected threads using `cuda::barrier`
  - All threads across blocks: `this_grid().sync()` (requires Cooperative Groups and launch using `cudaLaunchCooperativeKernel`)

# CUDA and OpenGL



- High-performance GPU compute
  - Real-time 3D graphics
- If both are on the same GPU, they can communicate!
- CUDA can read from/write to OpenGL buffers
  - OpenGL will use the updated data to render
  - That all without using the CPU or any extensive memory copying
  - (It's how I made the intro)

# CUDA and OpenGL



# CUDA and OpenGL

- `__device__` functions can access OpenGL data

```
#include <cuda_gl_interop.h>
// set up on CPU
cudaGraphicsResource *resource; T *dev_ptr; size_t size;
cudaGraphicsGLRegisterBuffer(&resource, handle,
↪  cudaGraphicsRegisterFlagsNone);
cudaGraphicsMapResources(1, &resource);
cudaGraphicsResourceGetMappedPointer(
    reinterpret_cast<void **>(&dev_ptr), &size, resource
);
// ... do stuff with dev_ptr on GPU
// clean up on CPU
cudaGraphicsUnmapResources(1, &resource);
cudaGraphicsUnregisterResource(resource);
```