TDT4200 Parallel Programming

Bart van Blokland

Lecture 11

For November 15th:

Send me topics you want me to explain once more!

Link:

https://docs.google.com/forms/d/e/1FAIpQLSfEOE4N6FD_q0ujHKvXEP3SupPcHHoYWf21J0WIcfzorer-5A/viewform

If you want me to talk about a topic you still don’t quite understand, let me know by using the link in the slide!

About last time..

A note on memory loads/stores

Warp-level intrinsics

I didn’t quite get the time last week to talk about some topics, so I want to go into them a bit deeper this time.

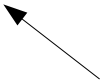
A note on memory loads and stores

```
unsigned int threadIndex = blockDim.x * blockIdx.x + threadIdx.x;
float4 vertex = vertexArray[threadIndex];
```

So first I need to mention something that I didn’t actually knew existed.

A note on memory loads and stores

```
unsigned int threadIndex = blockDim.x * blockIdx.x + threadIdx.x;
float4 vertex = vertexArray[threadIndex];
```



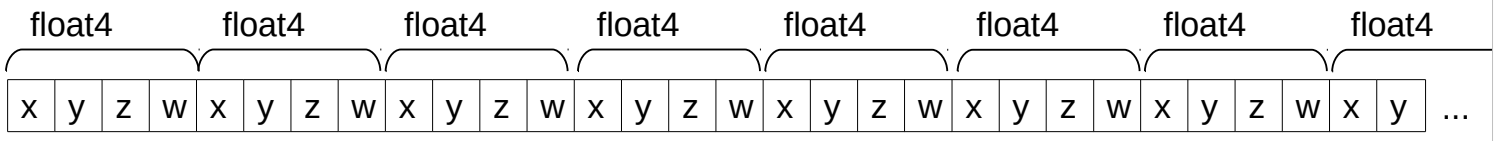
```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

I talked last time about how loading a struct into memory is essentially loading each of its fields separately. Therefore, the float4 read shown in the slide really requires four separate memory reads.

A note on memory loads and stores

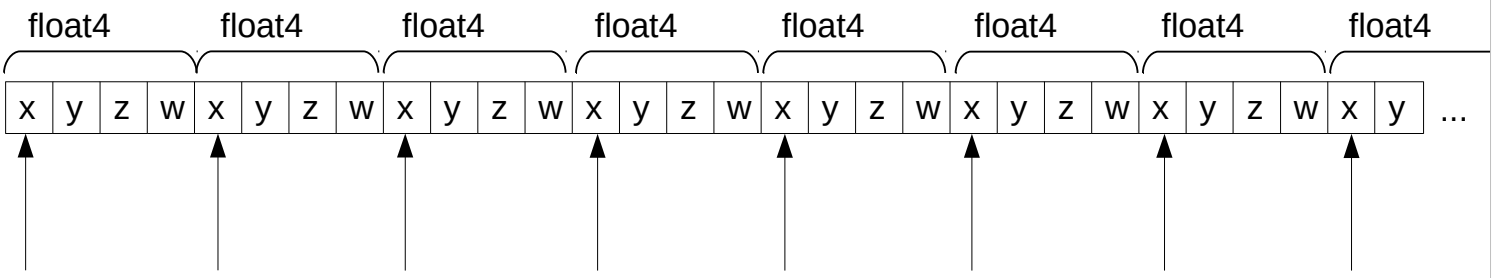
```
unsigned int threadIndex = blockDim.x * blockIdx.x + threadIdx.x;
float4 vertex = vertexArray[threadIndex];
```

```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```



When you look at the layout of this array of structs in memory, you can see that each of the fields are laid out consectively.

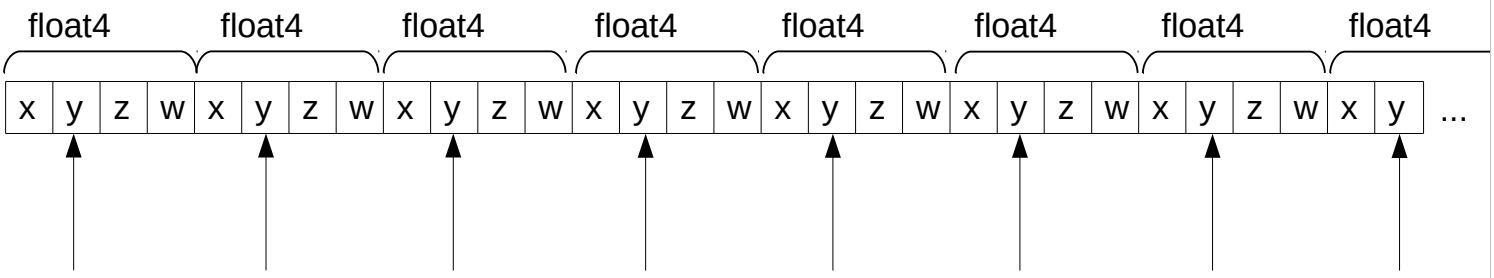
A note on memory loads and stores




```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

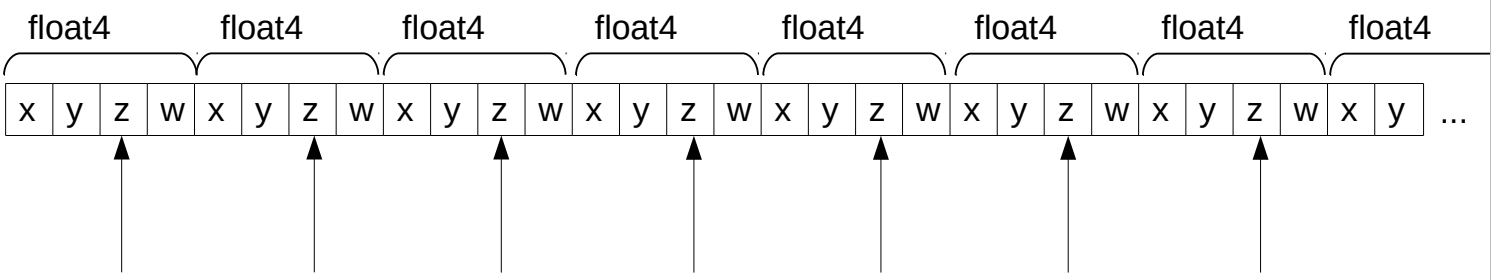
When each field is loaded individually, the access pattern produced by all threads in the warp requesting one 4-byte value has a lot of space between individual requests.

A note on memory loads and stores




```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

A note on memory loads and stores

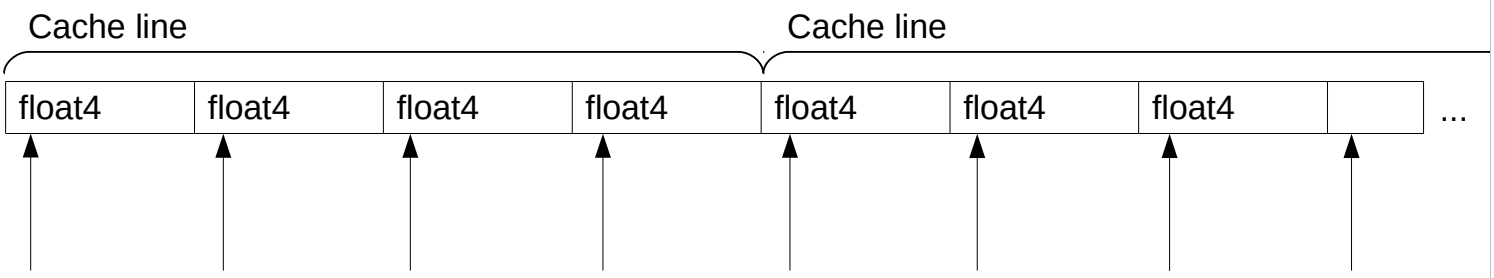



```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

The problem with this access pattern is that the requests of each load instruction within the warp spans 4 cache lines. That means that for every field of the struct to be read, you'd need to fetch 4 cache lines from global memory in order to get all data you need. And repeat that for each field in the struct.

Since cache lines don't tend to live very long on the GPU due to the high volume of memory accesses, this means you'd both need to perform additional memory transfers to fetch all the cache lines you need. In addition, for each cache line you do fetch, you're only effectively using 25% of it, meaning that 75% of the memory bandwidth from the memory banks is effectively wasted.

A note on memory loads and stores



Vector loads and stores can read and write 64 or 128-bit values in a single instruction

```
float4 vertex;
vertex.x = vertexArray[threadIndex].x;
vertex.y = vertexArray[threadIndex].y;
vertex.z = vertexArray[threadIndex].z;
vertex.w = vertexArray[threadIndex].w;
```

Now if the compiler recognises that a struct is a power of two in size, it can use a special hardware feature that loads in 16 bytes in one instruction. This means the entirety of the cache line is used, even though the original memory access pattern in theory is not ideal.

A note on memory loads and stores

Vector loads and stores can read and write 64 or 128-bit values in a single instruction

- NVCC will automatically apply them on structs whose size is a power of 2
- Just like SSE, you need to load an entire struct at once
- Use CUDA's built-in vector types (int2, float4, etc) wherever possible

```
float4 = vertices[index];
```

```
float4 vertex;
vertex.x = vertices[index].x;
vertex.y = vertices[index].y;
vertex.z = vertices[index].z;
vertex.w = vertices[index].w;
```

The NVCC compiler will generally only apply this optimisation on the built-in vector (float4, int2, etc) datatypes.

A note on memory loads and stores

When a warp is reading a value from memory, all requests should land in the same cache line.

cudaMalloc() guarantees the start of the allocated memory block is cache line aligned.

Perfectly coalesced memory transactions of a single warp should all land on the same cache line. Note that the location within the cache line doesn’t matter there. So if your threads are reading addresses in reverse, that’s totally fine too.

Also, cudaMalloc() guarantees that the start of a chunk of memory is located at the start of a cache line. You can use this fact to ensure that your memory transactions indeed land within similar cache lines.

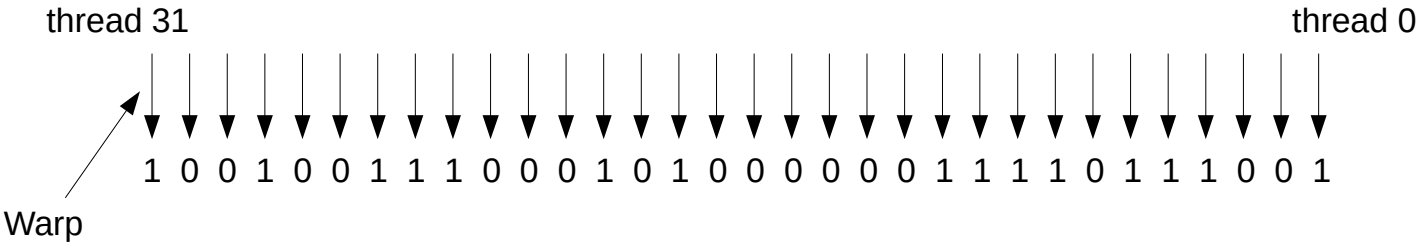
Once more: Warp Voting

32 threads in a warp, 32 bits in an unsigned int.

The “ballot” instruction lets you set one bit per thread in a warp.

Can be used to communicate between threads.

Completely inside the core, so no need for external memory.



To some extent, the requirement that GPUs only execute threads in a warp can be seen as a limitation. However, the warp voting and shuffle instructions turn this around, and use it as an advantage instead.

First off, warp voting allows all threads in the warp to “vote” with a boolean value. All votes are concatenated, and because a warp has 32 threads, the result is a 32-bit unsigned integer.

You can for example use this for figuring out which threads have work to do when having threads in a warp cooperate on a task. The other big advantage here is that because everything happens in-core, this means of communication is super fast. The same is true for shuffle instructions.

Once more: Warp Voting

```
#include <stdio.h>

__global__ void ballotExample() {
    bool needBathroom = threadIdx.x % 3 == 0;
    unsigned int votes = __ballot_sync(0xFFFFFFFF, needBathroom);
    unsigned int count = __popc(votes);
    if(threadIdx.x == 0) {
        printf("%i threads need a break.\n", count);
    }
}

int main() {
    ballotExample<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

The __popc() function is a “population count”: it returns the number of bits set. In this case, that will tell you how many threads votes “true” with their __ballot_sync() call.

Once more: Shuffle Instructions

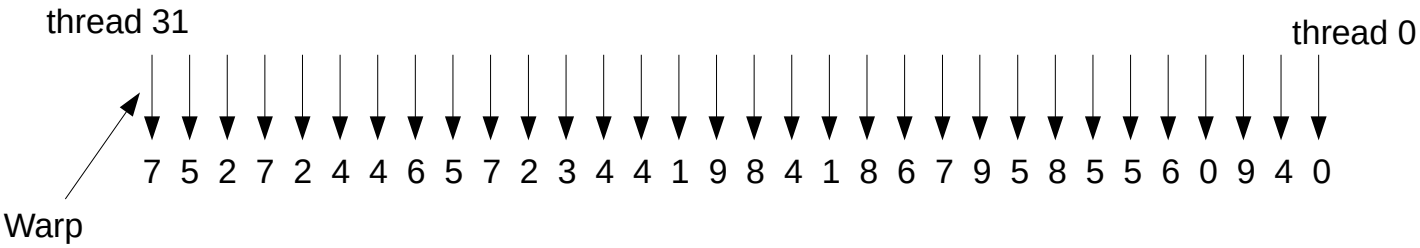
Read the contents of registers of other threads in the warp

Can save a lot of memory requests when using registers as temporary buffer.

Shuffle instructions can be a bit confusing at first, but the main idea is that threads within a warp can exchange the values of their variables (registers, to be more precise).

This allows you to, in a way, use registers as a way to temporarily store data you need fast access to. As long as one thread has a particular value, other threads can get hold of it if they need it.

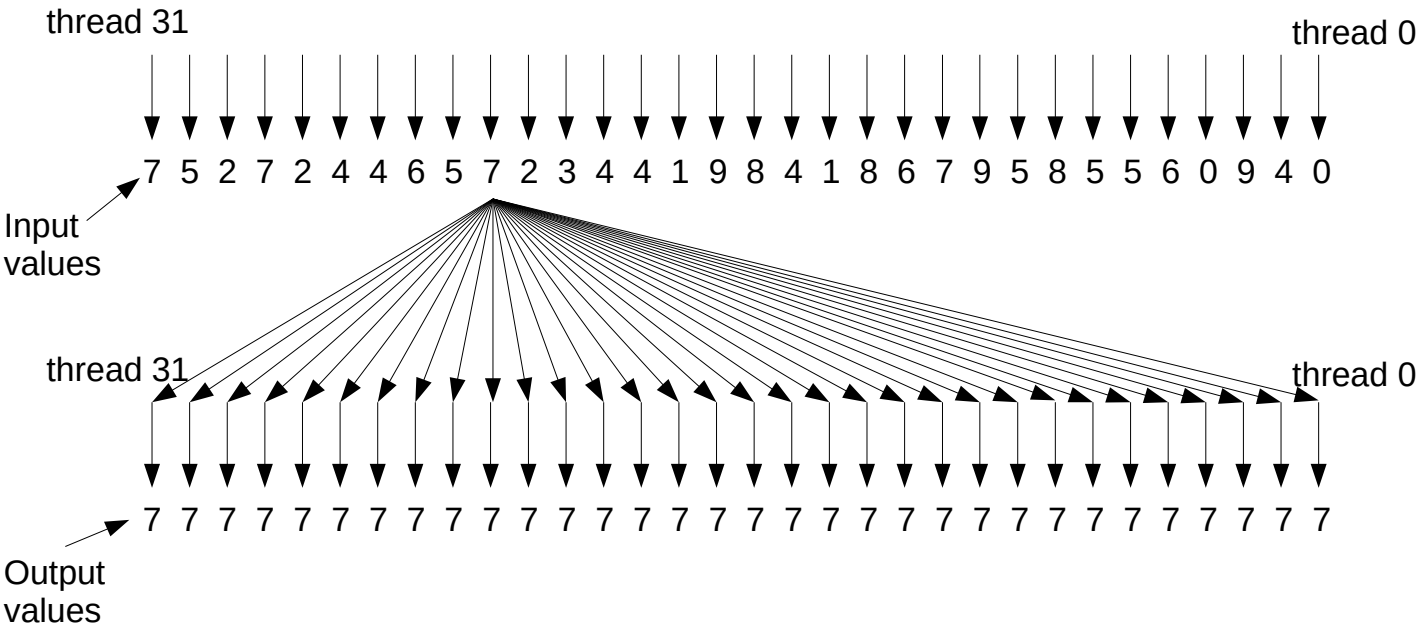
Once more: Shuffle Instructions



Note that thread 0 is usually located on the right hand side when it comes to warp operations.

With shuffle instructions, each thread “advertises” a value.

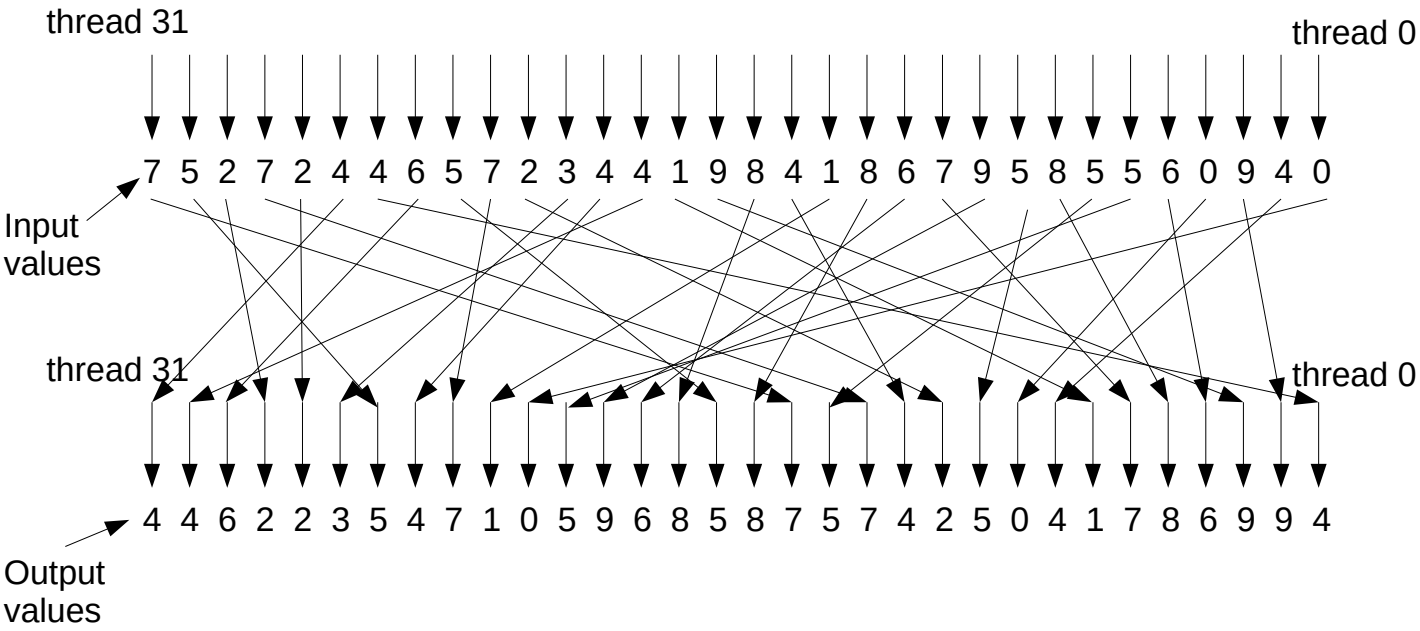
Once more: Shuffle Instructions



Each thread within the warp can subsequently grab the value which one of the other threads has advertised.

For example, if all threads choose to copy the value from one particular thread, the effect would essentially be a broadcast of that particular value within the thread.

Once more: Shuffle Instructions



But each thread can choose which thread it would like to read the value from on its own.

Once more: Shuffle Instructions

```
#include <stdio.h>

__global__ void shuffleExample() {
    int threadID = threadIdx.x;
    // Read value of threadID from thread 12
    int otherThreadID = __shfl_sync(0xFFFFFFFF, threadID, 12);
    if(threadIdx.x == 0) {
        printf("Thread 12's ID is: %i.\n", otherThreadID);
    }
}

int main() {
    shuffleExample<<<1, 32>>>();
    cudaDeviceSynchronize();
    return 0;
}
```

An example of a shuffle instruction used in a snippet of code. The threadID passed into __shfl_sync is the value advertised by the particular thread. The last parameter is the index of the thread which the advertised value should be read from. The function will then return that specific value.

Using registers as cache

```
#include <stdio.h>

__global__ void shuffleExample(int* array) {
    int threadIndex = threadIdx.x + blockDim.x * threadIdx.y;
    array[threadIndex] = threadIndex;
    int currentValue = array[threadIndex];

    otherValue = __shfl_sync(0xFFFFFFFF, currentValue, threadIdx.x + 1);
    array[threadIndex] = otherValue - currentValue;
    printf("Thread %i computed %i.\n", threadIndex, array[threadIndex]);
}

int main() {
    int* array;
    cudaMalloc(&array, 128*sizeof(int));
    dim3 blockDimensions(32, 4, 1);
    shuffleExample<<<1, blockDimensions>>>(array);
    cudaDeviceSynchronize();
    return 0;
}
```

Here's a neat way to use shuffle instructions: we can compute the difference between each subsequent element of an array, while only having to request the values from memory once.

Each thread loads an index from the array, and copies the next value in the array from its neighbour.

Small problem here: thread 31 will try to read a value from a thread that doesn't exist. So we need to solve the problem differently for those specific cases.

Using registers as cache

```
#include <stdio.h>

__global__ void shuffleExample(int* array) {
    int threadIndex = threadIdx.x + blockDim.x * threadIdx.y;
    array[threadIndex] = threadIndex;
    int currentValue = array[threadIndex];
    __shared__ int exchange[5];
    if(threadIdx.x == 0) { exchange[threadIdx.y] = threadIndex; }
    __syncthreads();
    int otherValue = __shfl_sync(0xFFFFFFFF, currentValue, threadIdx.x + 1);
    if(threadIdx.x == 31 && threadIdx.y != blockDim.y - 1) {
        otherValue = exchange[threadIdx.y + 1];
        printf("Thread %i read %i.\n", threadIndex, otherValue);
    }
    array[threadIndex] = otherValue - currentValue;
    printf("Thread %i computed %i.\n", threadIndex, array[threadIndex]);
}

int main() {
    int* array;
    cudaMalloc(&array, 128*sizeof(int));
    dim3 blockDimensions(32, 4, 1);
    shuffleExample<<<1, blockDimensions>>>(array);
    cudaDeviceSynchronize();
    return 0;
}
```

Here's the complete program. We'll use some shared memory to store the starting values of the next warp.

So, thread 0 of each warp writes its own value to shared memory. Next, we use a barrier (__syncthreads()) to make sure all threads have completed their writes. The last thread in each warp can subsequently read the starting value of the next warp, thereby solving the problem from the previous slide.

I'll leave this for you to figure out:

- Why is the shuffle instruction not in the else clause of the if statement below it?
- Is there still an issue left in the code snippet?

Putting it all together:

Warp-level intrinsitcs

I also wanted to show an example putting these warp operations together. The application here is to dynamically append values to an array.

```cpp
#include <stdio.h>
__global__ void shuffleExample(int* array, int* nextIndex) {
    bool needToDoSomething = threadIdx.x % 5 == 0;
    int myValue = blockIdx.x * 100 + threadIdx.x;
    unsigned int votes = __brev(__ballot_sync(0xFFFFFFFF, needToDoSomething));
    unsigned int count = __popc(votes);
    unsigned int startIndex;
    if(threadIdx.x == 0) { startIndex = atomicAdd(nextIndex, count); }
    startIndex = __shfl_sync(0xFFFFFFFF, startIndex, 0);
    int howManyCameBeforeMe = __popc(votes >> (32 - threadIdx.x));
    if(needToDoSomething) {
        printf("%i -> %i\n", startIndex + howManyCameBeforeMe, myValue);
        array[startIndex + howManyCameBeforeMe] = myValue;
    }
}

int main() {
    int* array; int* nextIndex;
    cudaMalloc(&array, 32 * sizeof(int)); cudaMalloc(&nextIndex, sizeof(int));
    cudaMemset(nextIndex, sizeof(int), 0);
    shuffleExample<<<3, 32>>>(array, nextIndex);
    cudaDeviceSynchronize();
    return 0;
}
```

Here's an example showing how ballot and shuffle instructions can be used. The values of needToDoSomething and myValue are just arbitrary input. We first perform a vote. This tells us how many elements in the input array to reserve. Here only one thread is performing this action, and is reserving spots in the array for all threads at once. Next, because each thread needs to know which starting index thread 0 reserved, we broadcast the start index using a shuffle instruction to all threads. Now each thread computes which index they should write to, by computing how many threads voted "true" before them [by thread index in the warp]. We finally write out the value to the array.

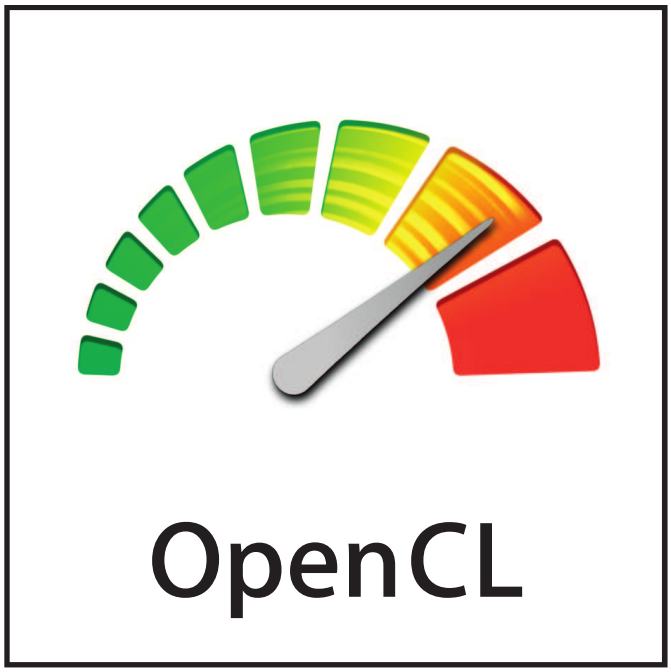
In this way, we can append items to an array using only a single atomic operation.

```cpp
#include <stdio.h>
__global__ void shuffleExample(int* array, int* nextIndex) {
    bool needToDoSomething = threadIdx.x % 5 == 0;
    int myValue = blockIdx.x * 100 + threadIdx.x;
    unsigned int votes = __brev(__ballot_sync(0xFFFFFFFF, needToDoSomet
    unsigned int count = __popc(votes);
    unsigned int startIndex;
    if(threadIdx.x == 0) { startIndex = atomicAdd(nextIndex, count); }
    startIndex = __shfl_sync(0xFFFFFFFF, startIndex, 0);
    int howManyCameBeforeMe = __popc(votes >> (32 - threadIdx.x));
    if(needToDoSomething) {
        printf("%i -> %i\n", startIndex + howManyCameBeforeMe, myValue)
        array[startIndex + howManyCameBeforeMe] = myValue;
    }
}

int main() {
    int* array; int* nextIndex;
    cudaMalloc(&array, 32 * sizeof(int)); cudaMalloc(&nextIndex, sizeof
    cudaMemset(nextIndex, sizeof(int), 0);
    shuffleExample<<<3, 32>>>(array, nextIndex);
    cudaDeviceSynchronize();
    return 0;
}
```

```
14 -> 0
15 -> 5
16 -> 10
17 -> 15
18 -> 20
19 -> 25
20 -> 30
7 -> 100
8 -> 105
9 -> 110
10 -> 115
11 -> 120
12 -> 125
13 -> 130
0 -> 200
1 -> 205
2 -> 210
3 -> 215
4 -> 220
5 -> 225
6 -> 230
```

The output of the program. As you can see the order in which items are appended depends on the order in which the atomicAdd function is executed. However, no element is overwritten twice, and we only increment nextIndex once in each warp.

Today:



Moving on to the topic of the day: OpenCL

What is it, and why do we bother?

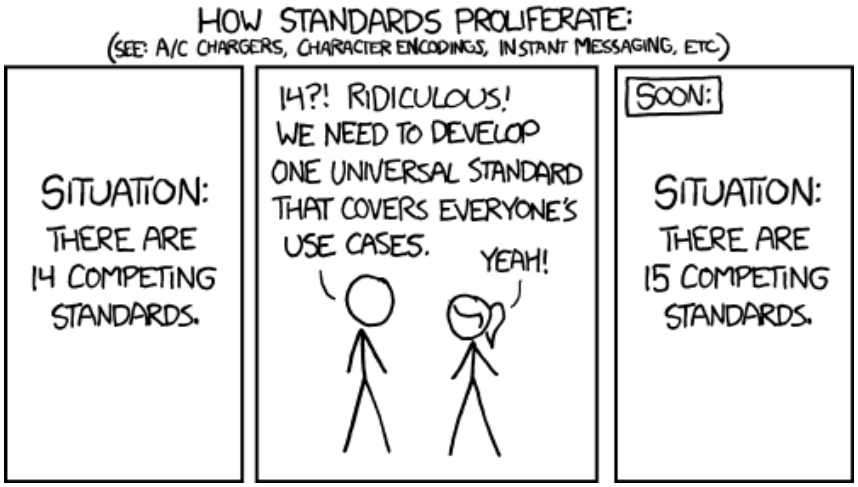
And of course we need to start with understanding why this API is even relevant.

What is it, and why do we bother?

A standard for parallel computing.

Because OpenCL is yet another standard for parallel computation.

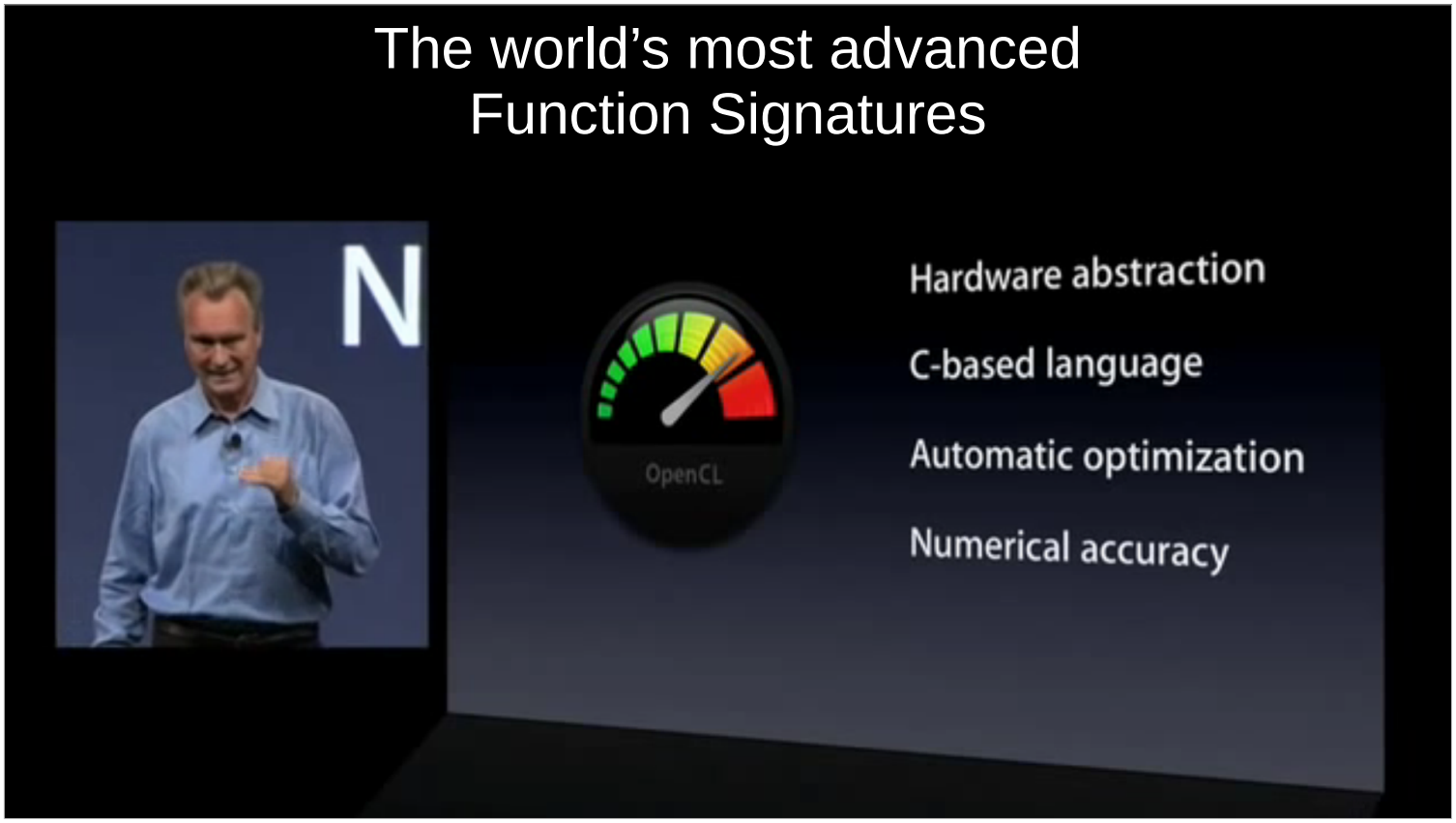
We've already seen MPI, OpenMP, and CUDA. Why exactly do we need anything else?



So why exactly do we need another standard when most of our use cases are already covered by other standards?

The answer lies in its original announcement.

And before I tell you who that was, does anyone know who initially developed it?



Apple originally released OpenCL in 2009 for doing computations on the GPU's of their computers. But released it as an open standard.

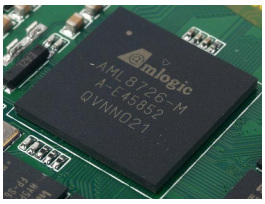
Did this standard take off, you might ask?

ARM CPU’s

Intel CPU’s

AMD CPU’s

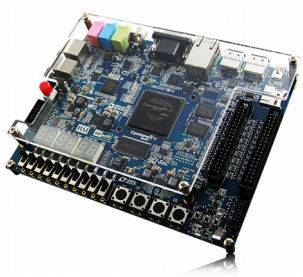
Intel Xeon Phi



ARM GPU’s

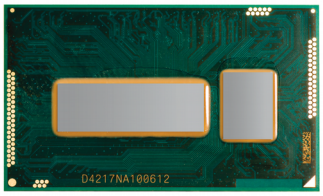
Nvidia GPU’s

AMD GPU’s

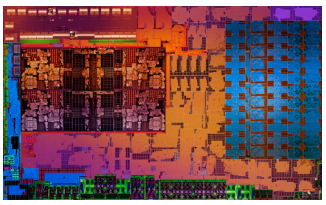


Altera FPGA’s

IBM POWER CPU’s



Intel Integrated Graphics



AMD Integrated Graphics

You could say so, yes.

Currently OpenCL is the most cross-platform parallel computing language out there.

Anything that can run a thread (including weaving machines) can pretty much run OpenCL.

If you’re an application developer, that can be a really big deal. Your single program will run on anything.

It basically runs on everything!

Except Apple hardware.. (soon)

Apple Deprecates OpenGL & OpenCL

Written by Michael Larabel in Standards on 4 June 2018 at 04:31 PM EDT. 107 Comments

With today's announcement of macOS 10.14 Mojave, Apple quietly confirmed they are deprecating OpenGL and OpenCL within macOS.

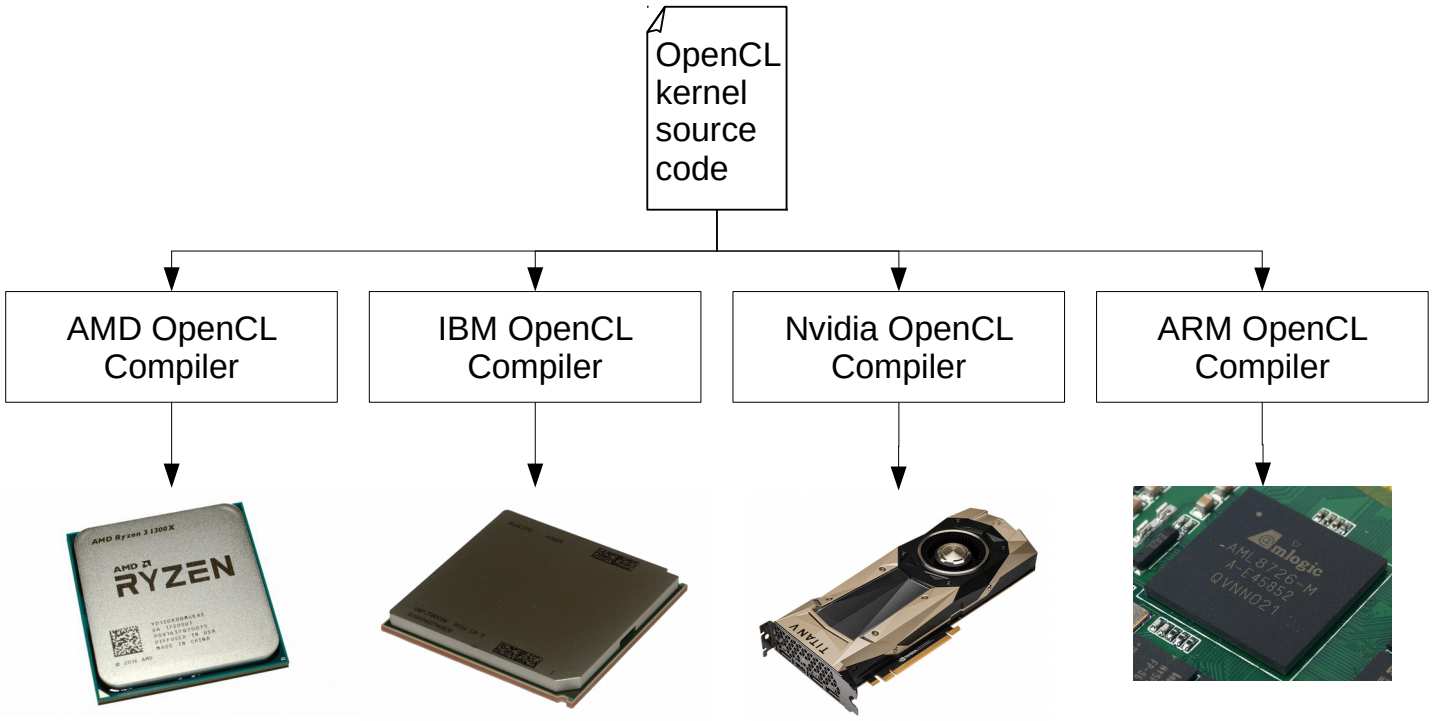
Apple deprecating OpenCL and OpenGL hardly comes as a surprise given in the past few years they have been pushing their Metal API for graphics and compute across macOS and iOS. Additionally, their OpenGL stack hasn't been updated well in years and has lagged behind the OpenGL 4.x upstream advancements out of The Khronos Group.

Sadly, this deprecation doesn't come because of supporting Vulkan, but just their vendor-

Anything, except Apple. They’re going to ditch support for OpenCL and OpenGL (a graphics rendering library) over the next few years.

But hang on.. Did you say OpenCL is a language?

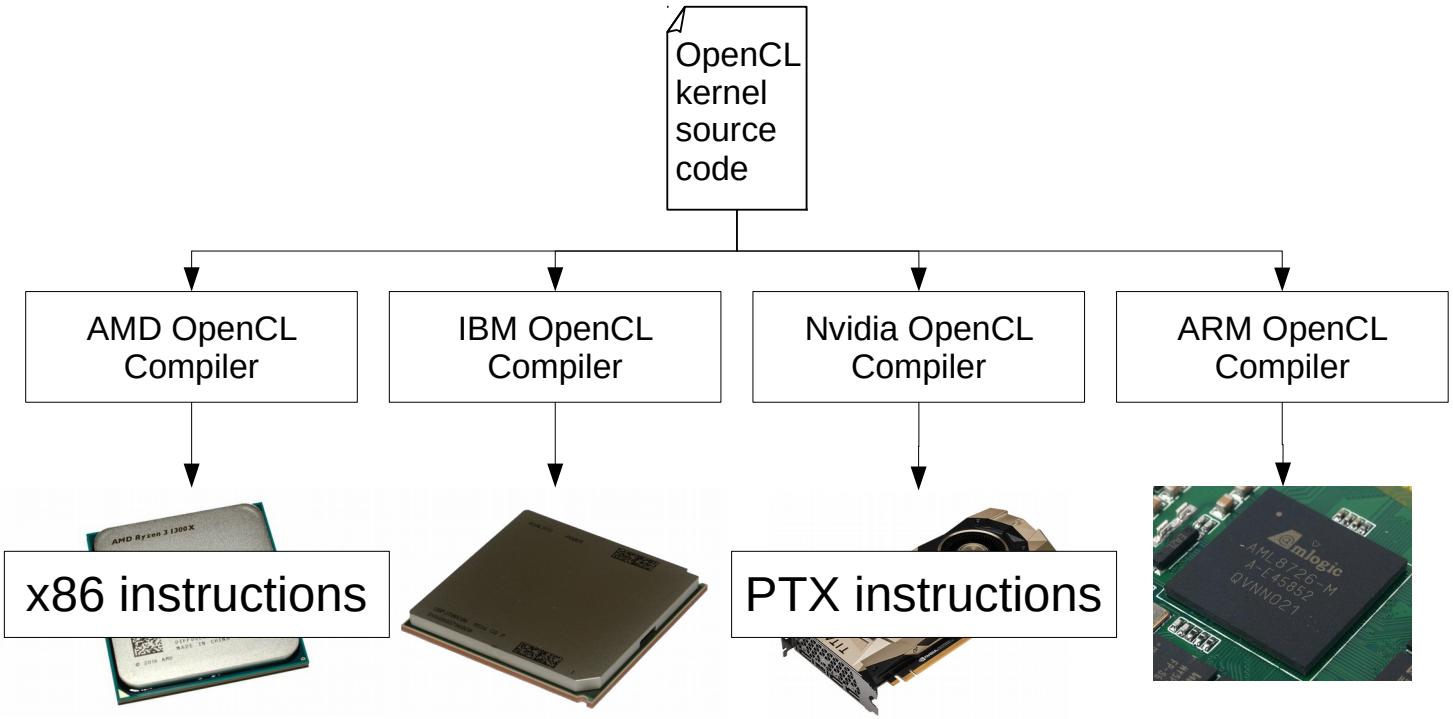
OpenCL is a language



Well yes. You see, the upside of having a cross-platform API is that your code is portable. The downside is that even between generations of the same vendor's chips there can be significant differences in what code is optimal.

OpenCL therefore ships source code with the program, and the runtime on the machine executing the kernels compiles it for whatever platform you want to run it on.

OpenCL is a language



This means that the same code tends to result in widely varying binaries, due to different compilers and instruction sets.

+ Lots of hardware is supported

- Need to generalise the API

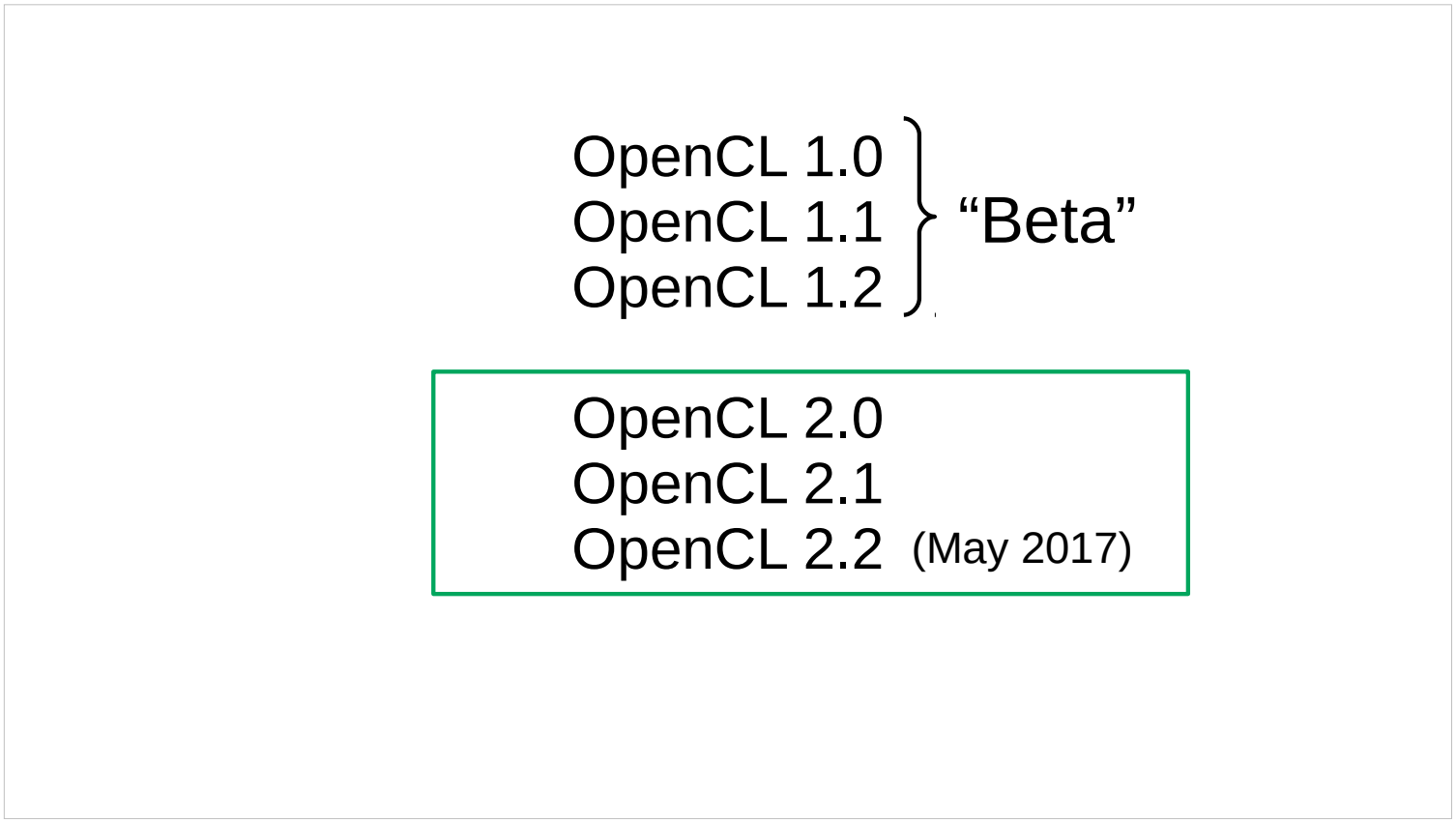
- May sacrifice some performance
(although not necessarily)

Another cost of having the multi platform flexibility is that the API needs to be able to deal with the possible presence of hardware from more than one vendor that’s capable of executing kernels.

For instance, consider a machine with an Intel APU, an AMD GPU, and an Nvidia GPU. You could run your program on the CPU, as well as the GPU from three different vendors.

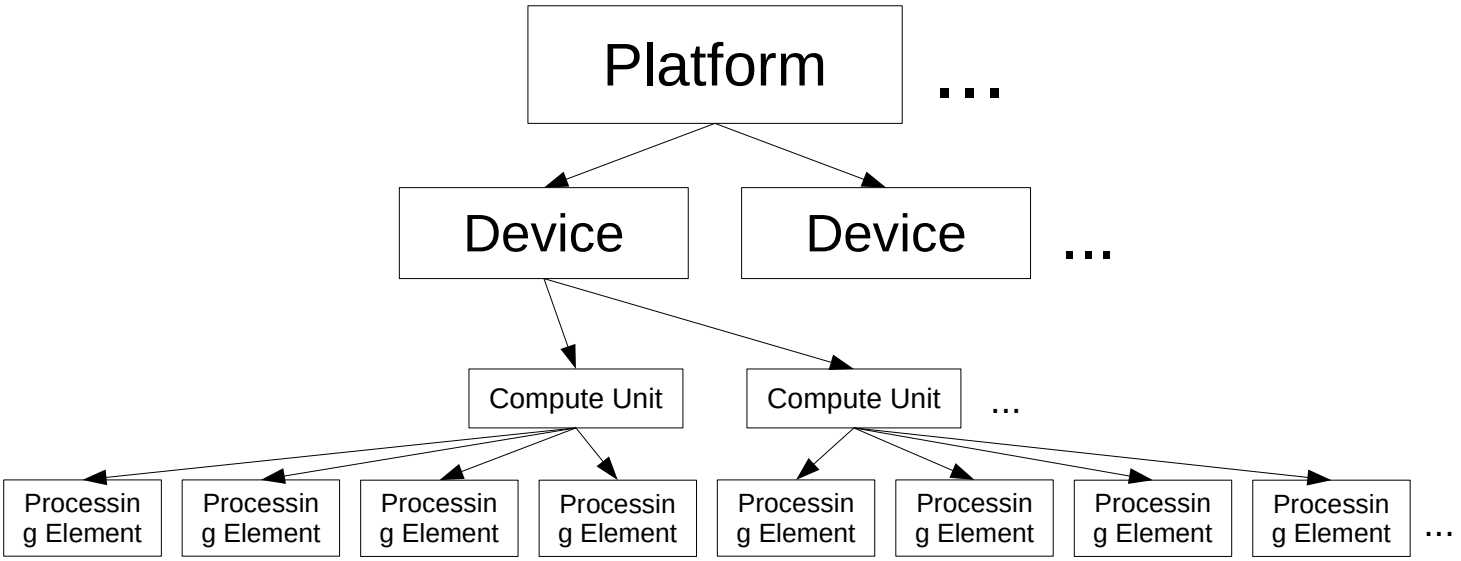
When using CUDA, you can only choose between Nvidia cards.

So how do we write OpenCL programs?



Answering that question to some extent involves asking which version of the standard you use. I recommend you always use 2.0 to 2.2, which is generally better designed.

First, some terminology



In general, a Platform is the platform of a particular hardware vendor. Think of Intel or AMD. Each platform can have one or more devices which are available to run OpenCL code on.

In turn, a device consists of a number of compute units, each of which has a number of processing elements.

To draw a parallel (heh) to CUDA here, a compute unit is roughly an SM, and a processing element a CUDA core.

First, some terminology

CUDA	OpenCL
	Platform
Host	Host
Device	Device
Streaming Multiprocessor	Compute Unit
CUDA core	Processing Element

But CUDA has no notion of a Platform.

How to launch a kernel:

1. Create a context
2. Load the kernel source code
3. Compile the kernel
4. Allocate memory on device
5. Copy memory to device
6. Launch kernel
7. Copy back results

How to launch a kernel:

1. Create a context
2. Load the kernel source code
3. Compile the kernel
4. Allocate memory on device
5. Copy memory to device
6. Launch kernel
7. Copy back results

There are a number of steps involved with writing a kernel in OpenCL. However, as you might see here, the last steps are the same as with CUDA.

Creating a context: Discovering platforms

```
#include <iostream>
#include <CL/cl2.hpp>

int main() {
    cl_platform_id platforms[10];
    cl_uint platformCount;
    clGetPlatformIDs(10, platforms, &platformCount);

    for(int i = 0; i < platformCount; i++) {
        size_t size = 0;
        char* platformName = new char[200];
        clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME, size, platformName, nullptr);
        std::cout << platformName << std::endl;
    }

    return 0;
}
```

Prints “NVIDIA CUDA” on my laptop

The very first step in creating an OpenCL program is discovering which devices we can run it on.

Getting OpenCL working (on Ubuntu):

```
sudo apt install ocl-icd-* opencl-headers clinfo
```

Compiling a program:

```
g++ main.cpp -o clprogram -lOpenCL
```

clinfo is a handy tool for looking at platform/device information and testing whether your OpenCL installation has succeeded.

Creating a context: Discovering devices

```
#include <iostream>
#include <CL/cl2.hpp>

int main() {
    cl_platform_id platforms[10];
    cl_uint platformCount;
    clGetPlatformIDs(10, platforms, &platformCount);

    for(int i = 0; i < platformCount; i++) {
        char* name = new char[200];
        clGetPlatformInfo(platforms[i], CL_PLATFORM_NAME, 200, name, nullptr);
        std::cout << "Platform: " << name << std::endl;

        cl_device_id devices[10];
        cl_uint deviceCount;
        clGetDeviceIDs(platforms[i], CL_DEVICE_TYPE_ALL, 10, devices, &deviceCount);
        for(int j = 0; j < deviceCount; j++) {
            clGetDeviceInfo(devices[i], CL_DEVICE_NAME, 200, name, nullptr);
            std::cout << "  Device: " << name << std::endl;
        }
    }

    return 0;
}
```

CL_TYPE_DEVICE_ALL
CL_TYPE_DEVICE_CPU
CL_TYPE_DEVICE_GPU
CL_TYPE_DEVICE_ACCELERATOR
CL_TYPE_DEVICE_DEFAULT

Output:
Platform: NVIDIA CUDA
  Device: GeForce GTX 1070

Once we have found a platform we like, we can list any devices it has made available. It is also possible to filter this list by a particular device type if you so desire (see list in the top right corner).

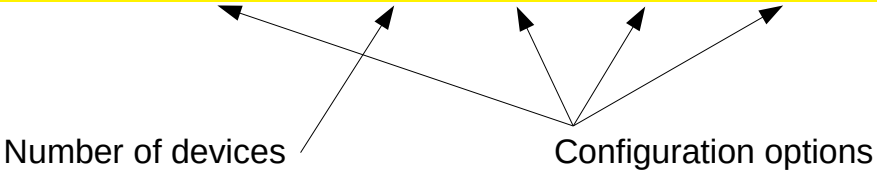
Creating a context: Selecting any available GPU

```
#include <iostream>
#include <CL/cl2.hpp>

int main() {
    cl_platform_id platform;
    clGetPlatformIDs(1, &platform, nullptr);

    cl_device_id device;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, nullptr);

    char* name = new char[200];
    clGetDeviceInfo(device, CL_DEVICE_NAME, 200, name, nullptr);
    std::cout << "Selected device: " << name << std::endl;

    return 0;
}
```

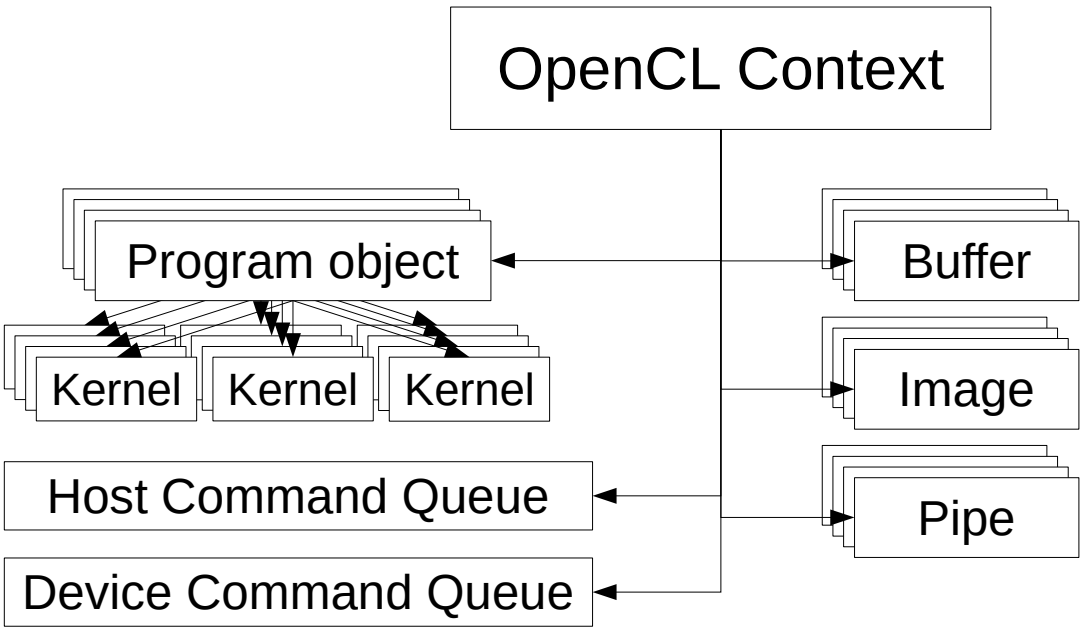
Let’s trim this code down a bit by selecting a default platform and device.

Creating a context: Creating the context itself

```
#include <iostream>
#include <CL/cl2.hpp>

int main() {
    cl_platform_id platform;
    clGetPlatformIDs(1, &platform, nullptr);

    cl_device_id device;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, nullptr);

    char* name = new char[200];
    clGetDeviceInfo(device, CL_DEVICE_NAME, 200, name, nullptr);
    std::cout << "Selected device: " << name << std::endl;

    cl_context context = clCreateContext(nullptr, 1, &device, nullptr, nullptr, nullptr);

    return 0;
}
```



The next thing we need to do is to create a context on the device we picked.



Contexts keep track of everything we're doing with that particular device. It contains program objects (essentially compiled OpenCL source files), memory buffers and resources, as well as a command queue.

Creating a context: Creating a command queue

```
#include <iostream>
#include <CL/cl2.hpp>

int main() {
    cl_platform_id platform;
    clGetPlatformIDs(1, &platform, nullptr);

    cl_device_id device;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, nullptr);

    char* name = new char[200];
    clGetDeviceInfo(device, CL_DEVICE_NAME, 200, name, nullptr);
    std::cout << "Selected device: " << name << std::endl;

    cl_context context = clCreateContext(nullptr, 1, &device, nullptr, nullptr, nullptr);

    cl_command_queue commandQueue = clCreateCommandQueueWithProperties(
                                                        context, device, 0, nullptr);

    return 0;
}
```

This command queue needs to be created explicitly though.

Loading the kernel source code

```
#include <iostream>
#include <CL/cl2.hpp>
#include <string>
#include <fstream>
#include <streambuf>
std::string loadFileFromDisk(std::string src) {
    std::ifstream t(src);
    return std::string((
                std::istreambuf_iterator<char>(t)), std::istreambuf_iterator<char>());
}

int main() {
    cl_platform_id platform;
    clGetPlatformIDs(1, &platform, nullptr);
    cl_device_id device;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, nullptr);
    cl_context context = clCreateContext(nullptr, 1, &device, nullptr, nullptr, nullptr);
    cl_command_queue commandQueue =
                        clCreateCommandQueueWithProperties(context, device, 0, nullptr);

    std::string kernelSourceCode = loadFileFromDisk("kernel.cl");

    return 0;
}
```

We can now load in our kernel source code from a text file of our choosing.

Compiling the OpenCL kernel

```
#include <iostream>
#include <CL/cl2.hpp>
#include <string>
#include <fstream>
#include <streambuf>
std::string loadFileFromDisk(std::string src) {std::ifstream t(src);return
std::string((std::istreambuf_iterator<char>(t)), std::istreambuf_iterator<char>());}
int main() {
    cl_platform_id platform;
    clGetPlatformIDs(1, &platform, nullptr);
    cl_device_id device;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, nullptr);
    cl_context context = clCreateContext(nullptr, 1, &device, nullptr, nullptr, nullptr);
    cl_command_queue commandQueue =
                        clCreateCommandQueueWithProperties(context, device, 0, nullptr);
    std::string kernelSourceCode = loadFileFromDisk("kernel.cl");

    const char* c_kernelSourceCode = kernelSourceCode.c_str();
    const size_t c_kernelSourceSize = kernelSourceCode.size();

    cl_program program = clCreateProgramWithSource(
                        context, 1, &c_kernelSourceCode, &c_kernelSourceSize, nullptr);
    clBuildProgram(program, 0, nullptr, "-cl-std=CL2.0 -cl-mad-enable", nullptr, nullptr);

    return 0;
}
```

We subsequently need to hand it over to OpenCL and compile it. Note I’m not doing any error checking here (one of the reasons there are lots of nullptr parameters in the code).

Time to look at kernels!

Which means we can now look at some kernels.

Aren’t they beautiful?

```
__kernel void addArrays(__global int* out,
                        __global int* array1,
                        __global int* array2) {
   int threadIndex = get_global_id(0);
   out[threadIndex] = array1[threadIndex] + array2[threadIndex];
}
```

This is what a basic kernel looks like in OpenCL. In many ways, this should remind you of CUDA. A function marked as a kernel is executed many times in parallel.

Each instance that's launched first determines which thread it is, and then uses that value to do a particular operation.

Compiling the OpenCL kernel

```
#include <iostream>
#include <CL/cl2.hpp>
#include <string>
#include <fstream>
#include <streambuf>
std::string loadFileFromDisk(std::string src) {std::ifstream t(src);return
std::string((std::istreambuf_iterator<char>(t)), std::istreambuf_iterator<char>());}
int main() {
    cl_platform_id platform;
    clGetPlatformIDs(1, &platform, nullptr);
    cl_device_id device;
    clGetDeviceIDs(platform, CL_DEVICE_TYPE_GPU, 1, &device, nullptr);
    cl_context context = clCreateContext(nullptr, 1, &device, nullptr, nullptr, nullptr);
    cl_command_queue commandQueue =
                        clCreateCommandQueueWithProperties(context, device, 0, nullptr);
    std::string kernelSourceCode = loadFileFromDisk("kernel.cl");

    const char* c_kernelSourceCode = kernelSourceCode.c_str();
    const size_t c_kernelSourceSize = kernelSourceCode.size();
    cl_program program = clCreateProgramWithSource(
                        context, 1, &c_kernelSourceCode, &c_kernelSourceSize, nullptr);
    clBuildProgram(program, 0, nullptr, "-cl-std=CL2.0 -cl-mad-enable", nullptr, nullptr);
    cl_kernel kernel = clCreateKernel(program, "addArrays", nullptr);
    // I will continue from here on the next slides!!
    return 0;
}
```

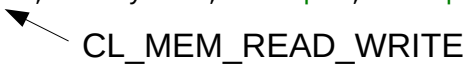
Now that we have a kernel, we can launch it.

We don’t have a fancy bit of syntax like CUDA does, so we’ll go the manual route instead.

The first step in that is getting a handle of our kernel. Since a source file can contain multiple kernels, we need to tell OpenCL explicitly which one to use. Note that one of the parameters is a string with the kernel name from the previous slide.

Allocating Buffers

```
size_t arraySize = 512 * sizeof(int);
cl_mem outBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, arraySize, nullptr, nullptr);
cl_mem array1Buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, arraySize, nullptr, nullptr);
cl_mem array2Buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, arraySize, nullptr, nullptr);
```



Next, we need to allocate our input buffers on the device.

We can use the clCreateBuffer() function for that. Note that unlike CUDA, we also need to specify whether we intend to read, write, or both to this kernel. The OpenCL compiler can use this for specific optimisations.

Copy memory to device

```
size_t arraySize = 512 * sizeof(int);
cl_mem outBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, arraySize, nullptr, nullptr);
cl_mem array1Buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, arraySize, nullptr, nullptr);
cl_mem array2Buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, arraySize, nullptr, nullptr);

int* out = new int[512];
int* array1 = new int[512];
int* array2 = new int[512];

for(int i = 0; i < 512; i++) {out[i] = i; array1[i] = i; array2[i] = i;}
const int zero = 0;
clEnqueueFillBuffer(commandQueue, outBuffer, &zero, sizeof(int),
                            0, arraySize, 0, nullptr, nullptr);

clEnqueueWriteBuffer(commandQueue, array1Buffer, CL_TRUE, 0, arraySize,
                            array1, 0, nullptr, nullptr);
clEnqueueWriteBuffer(commandQueue, array2Buffer, CL_TRUE, 0, arraySize,
                            array2, 0, nullptr, nullptr);
```

clEnqueueFillBuffer(commandQueue, deviceBuffer, pattern, patternSize, startOffsetBytes, bufferSizeBytes, [dontcare], [dontcare], [dontcare]);
clEnqueueWriteBuffer(commandQueue, deviceBuffer, blockingWrite, startOffsetBytes, bufferSizeBytes, hostPointer, [dontcare], [dontcare], [dontcare]);

We can not either initialise the buffers we created with a default value (clEnqueueFillBuffer()), or copy some memory from the host side (clEnqueueWriteBuffer()).

For the latter function, note that this is the OpenCL equivalent of cudaMemcpy(), except there are different function signatures for copying memory to and from the device.

Launch kernel

```
size_t arraySize = 512 * sizeof(int);
cl_mem outBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, arraySize, nullptr, nullptr);
cl_mem array1Buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, arraySize, nullptr, nullptr);
cl_mem array2Buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, arraySize, nullptr, nullptr);
int* out = new int[512];
int* array1 = new int[512];
int* array2 = new int[512];
for(int i = 0; i < 512; i++) {out[i] = i; array1[i] = i; array2[i] = i;}
const int zero = 0;
clEnqueueFillBuffer(commandQueue, outBuffer, &zero, sizeof(int),
                        0, arraySize, 0, nullptr, nullptr);
clEnqueueWriteBuffer(commandQueue, array1Buffer, CL_TRUE, 0, arraySize,
                        array1, 0, nullptr, nullptr);
clEnqueueWriteBuffer(commandQueue, array2Buffer, CL_TRUE, 0, arraySize,
                        array2, 0, nullptr, nullptr);
clSetKernelArg(kernel, 0, sizeof(cl_mem), &outBuffer);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &array1Buffer);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &array2Buffer);
size_t globalDimensions[1];
globalDimensions[0] = 512;
size_t workGroupDimensions[1];
workGroupDimensions[0] = 32;
clEnqueueNDRangeKernel(commandQueue, kernel, 1, nullptr, globalDimensions,
                        workGroupDimensions, 0, nullptr, nullptr);
```

Launching the kernel itself requires setting the parameter values first. Next, we can define Ndrange (grid) and workgroup (block) dimensions.

Finally, at last, we enqueue a kernel launch.

Copy back the results

```
size_t arraySize = 512 * sizeof(int);
cl_mem outBuffer = clCreateBuffer(context, CL_MEM_WRITE_ONLY, arraySize, nullptr, nullptr);
cl_mem array1Buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, arraySize, nullptr, nullptr);
cl_mem array2Buffer = clCreateBuffer(context, CL_MEM_READ_ONLY, arraySize, nullptr, nullptr);
int* out = new int[512];
int* array1 = new int[512];
int* array2 = new int[512];
for(int i = 0; i < 512; i++) {out[i] = i; array1[i] = i; array2[i] = i;}
const int zero = 0;
clEnqueueFillBuffer(commandQueue, outBuffer, &zero, sizeof(int),
                        0, arraySize, 0, nullptr, nullptr);
clEnqueueWriteBuffer(commandQueue, array1Buffer, CL_TRUE, 0, arraySize,
                        array1, 0, nullptr, nullptr);
clEnqueueWriteBuffer(commandQueue, array2Buffer, CL_TRUE, 0, arraySize,
                        array2, 0, nullptr, nullptr);
clSetKernelArg(kernel, 0, sizeof(cl_mem), &outBuffer);
clSetKernelArg(kernel, 1, sizeof(cl_mem), &array1Buffer);
clSetKernelArg(kernel, 2, sizeof(cl_mem), &array2Buffer);
size_t globalDimensions[1];
globalDimensions[0] = 512;
size_t workGroupDimensions[1];
workGroupDimensions[0] = 32;
clEnqueueNDRangeKernel(commandQueue, kernel, 1, nullptr, workGroupDimensions,
                        workGroupDimensions, 0, nullptr, nullptr);
clEnqueueReadBuffer(commandQueue, outBuffer, CL_TRUE, 0, arraySize, out, 0, nullptr, nullptr)
```

Once that has finished executing, we can copy the results back to the CPU.

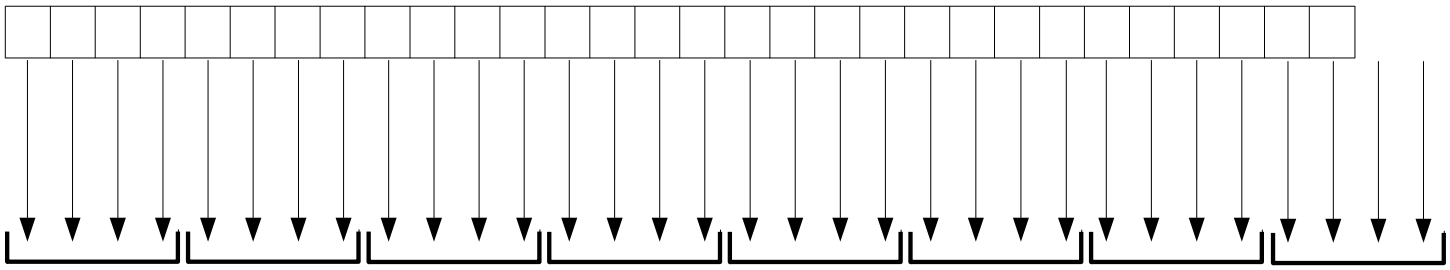
How does OpenCL execute kernels?

Different names, similar concepts

CUDA	OpenCL
Grid	NDRange
Block	Workgroup
Thread	Work-item
Shared Memory	Local Memory

The execution models of CUDA and OpenCL (large grid of threads cut into smaller pieces) are very similar.

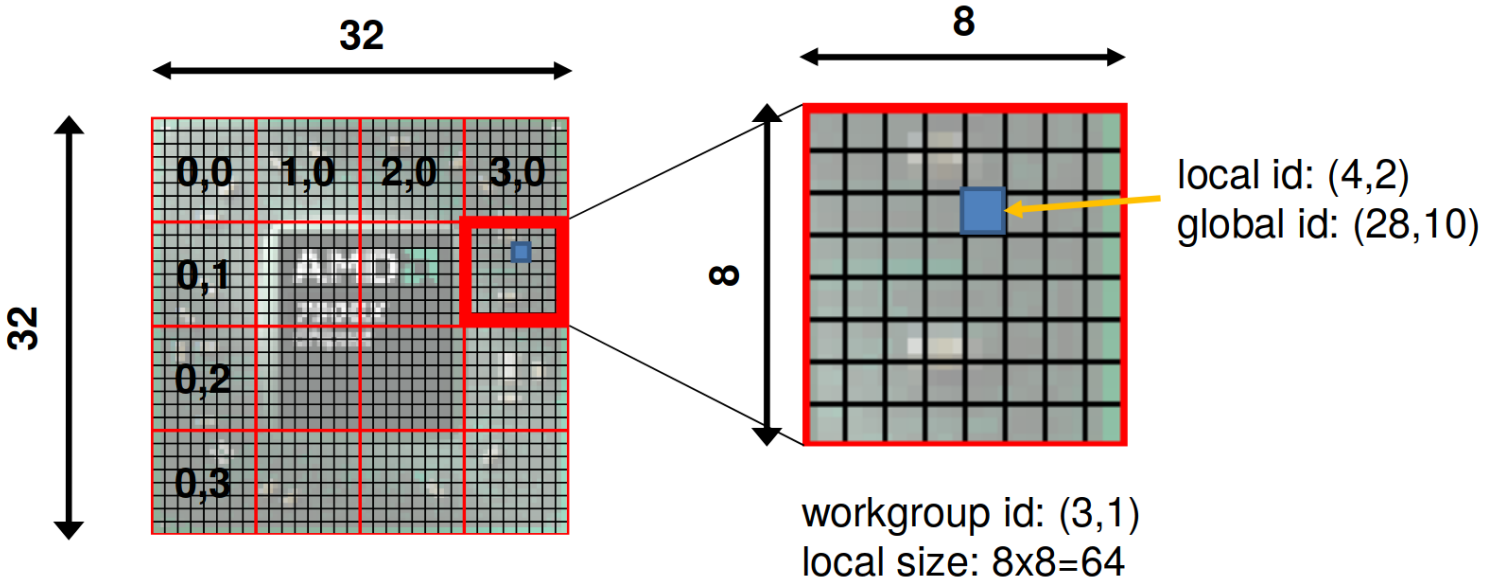
But with one important difference



CUDA: Launch x blocks, each containing y threads to cover z elements.

But there is one important difference. In CUDA, you can only launch a certain number of blocks, each containing a certain number of threads. You’re responsible yourself to figure out how many blocks you need, and ensure no thread reads memory that’s out of bounds.

But with one important difference



OpenCL: Launch enough threads to cover z elements, processed in blocks of y threads.

The difference here is that OpenCL only launches the threads you actually need. It figures out, based on the number of threads you specified in the Ndrange and the size of the work groups how many workgroups it needs to cover the ndrange.

Also, when you request the global ID, it will return the index of the thread in the ndrange (“grid”). In the edge cases where threads do not fill an entire work group, no threads are launched for out of bounds values.

```
// Get the number of dimensions in
// the NDRange and work groups
unsigned int get_work_dim();

// Get the width/height/depth of the NDRange
size_t get_global_size(unsigned int dimension);

// Get the width/height/depth of the work groups
size_t get_local_size(unsigned int dimension);

// Get the number of work groups for each axis
size_t get_num_groups(unsigned int dimension);

//Example: get the NDRange height
size_t imageHeight = get_global_size(1);
```

Here's how you can get hold of the grid and workgroup dimensions at runtime.

```
// Get the x/y/z-coordinate in the NDRange
size_t get_global_id(unsigned int dimension);

// Get the x/y/z-coordinate in the work group
size_t get_local_id(unsigned int dimension);

// Get the x/y/z-index of the work group
size_t get_group_id(unsigned int dimension);

// Example: get the work-item's 2D coordinate
size_t xCoordinate = get_global_id(0);
size_t yCoordinate = get_global_id(1);
```

And your work-item's (thread) particular location inside of the ndrange.

CUDA	OpenCL
`gridDim`	`get_num_groups()`
`blockIdx`	`get_group_id()`
`blockDim`	`get_local_size()`
`threadIdx`	`get_local_id()`

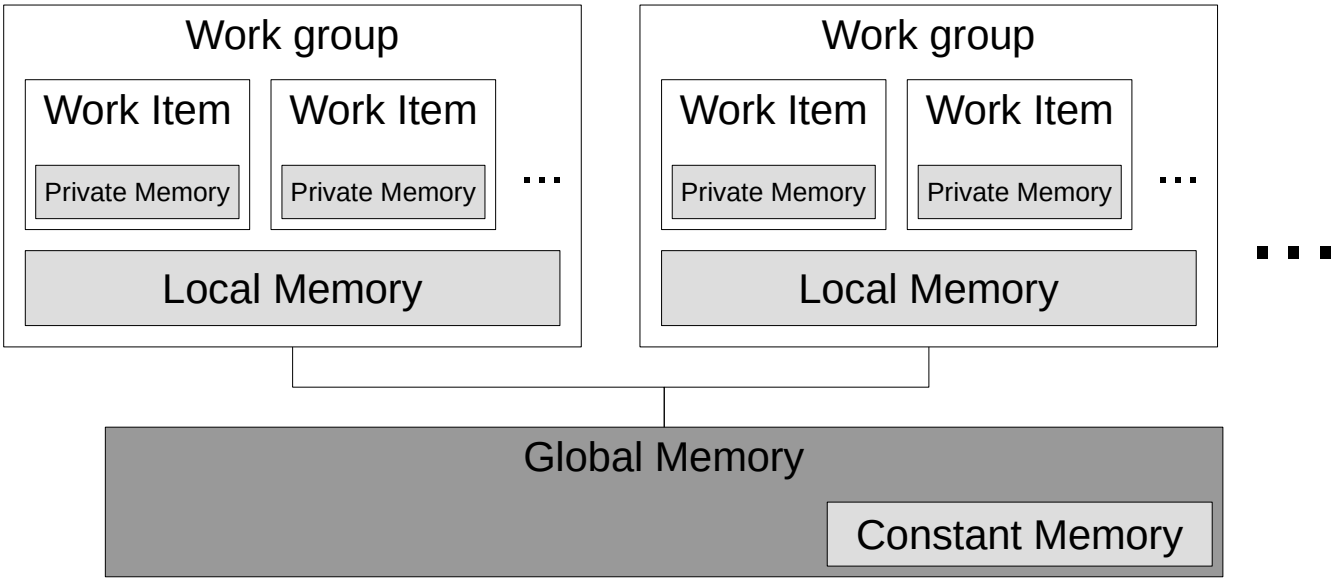
Here’s how to get the equivalent of the block and grid dimensions in OpenCL.

```
__kernel void addArrays(__global int* out,
                        __global int* array1,
                        __global int* array2) {
   int threadIndex = get_global_id(0);
   out[threadIndex] = array1[threadIndex] + array2[threadIndex];
}
```

```
__kernel void addArrays(__global int* out,
                        __global int* array1,
                        __global int* array2) {
   int threadIndex = get_global_id(0);
   out[threadIndex] = array1[threadIndex] + array2[threadIndex];
}
```

Back to the original kernel. What do these __global prefixes actually mean?

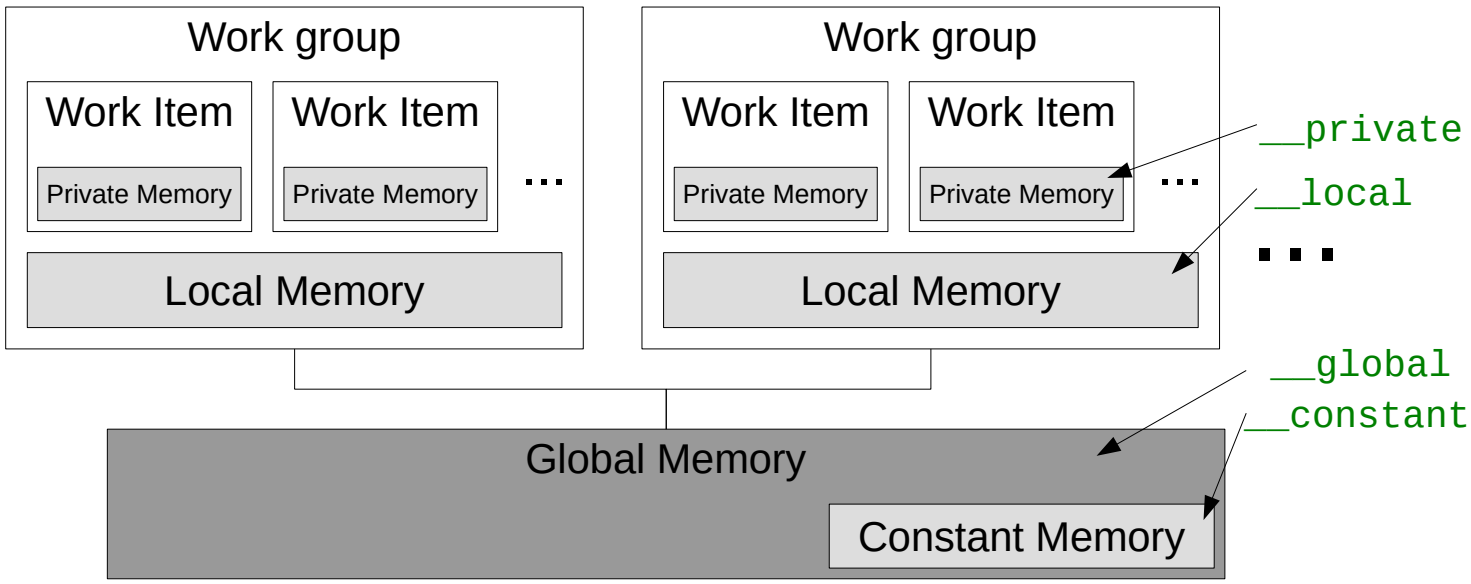
```
__kernel void addArrays(__global int* out,
                        __global int* array1,
                        __global int* array2) {
}
```



There are different types of memory defined in OpenCL. They are mainly tailored to the GPU, as that one has the most complex memory system.

When passing in bits of memory into a kernel, you need to prefix which type of memory the particular value should reside in.

```
__kernel void addArrays(__global int* out,
                        __global int* array1,
                        __global int* array2) {
}
```



__private memory is typically in registers

__constant memory has a special memory path on the GPU

Using Local Memory

CUDA:

```
__shared__ someArray[15];
```

OpenCL:

```
__kernel void addArrays(__local int* array) {
   __local int array2[15];
}

clSetKernelArg(kernel, 0, sizeof(int) * 15, nullptr);
```

Local (=shared) memory can be defined in two ways. Either it is passed in as a parameter into the kernel, where its size has to be set using a setKernelArg call prior to launch, or declared as a constant size array inside the kernel itself.

Various OpenCL features:

Synchronisation
Atomics
Images
Pipes
Work group broadcast
Reduction functions

For the remainder of the lecture, I’d like to go through some specific features.

Synchronisation

Barrier:

The thread waits until all threads have completed executing all instructions up to the barrier

Fence:

Guarantees all the thread's memory modifications are visible to all other threads after the fence

Barriers are synchronisation structures we've seen before. They make sure all threads reach the exact same line of code (in practice: same program counter) before continuing.

Barriers are particularly useful when dealing with situations where one thread can perform memory writes that are seen by another one. This ensures all memory write of the first thread are complete before the second one can see them.

However, sometimes all you need is the guarantee that the memory writes are visible to all other threads. In those cases you can instead use a so-called fence. A fence will not stop a thread, but just guarantees that all memory reads and writes up to that point have completed, and are thus visible to the other threads. However, threads do not wait for each other here.

Synchronisation

Barrier:

The thread waits until all threads have completed executing all instructions up to the barrier

→ Synchronises threads at runtime

Fence:

Guarantees all the thread's memory modifications are visible to all other threads after the fence

→ May only be a compiler hint

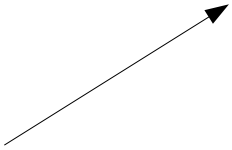
Moreover, fences are not necessarily an instruction. Depending on the architecture your program executes on, they may be nothing more than a compiler hint not to, when rearranging memory reads and writes for performance reasons, move them to the other side of the fence.

On processors capable of executing instructions out of order (such as many CPUs), an actual instruction may need to be emitted to ensure one particular memory write is not performed before another one. If you're curious, look into the "volatile" keyword for C++.

In practice, fences are only useful in extremely rare circumstances. You typically only need atomics, locks, and barriers for any synchronisation you need to do.

Synchronisation

```
void work_group_barrier(flags);
```



CLK_GLOBAL_MEM_FENCE

CLK_LOCAL_MEM_FENCE

CLK_IMAGE_MEM_FENCE

OpenCL is only able to synchronise work-items within a work-group. The barrier also requires you to specify which type of memory commits should be completed before the barrier is broken.

Atomics

Integer only!

- `atomic_add`
- `atomic_sub`
- `atomic_inc`
- `atomic_dec`

- `atomic_xchg`
- `atomic_cmpxchg`

Integer only!

- `atomic_min`
- `atomic_max`
- `atomic_and`
- `atomic_or`
- `atomic_xor`

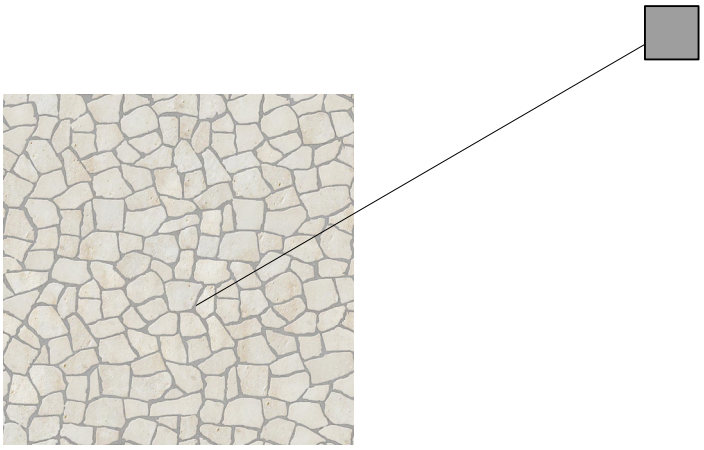
OpenCL of course also has atomics. But only the exchange type can be used on floating point numbers. The other ones only support integers.

Atomics

```
atomic_add(&counter, 42);
```

Using an atomic operation is very similar to CUDA, though.

Images

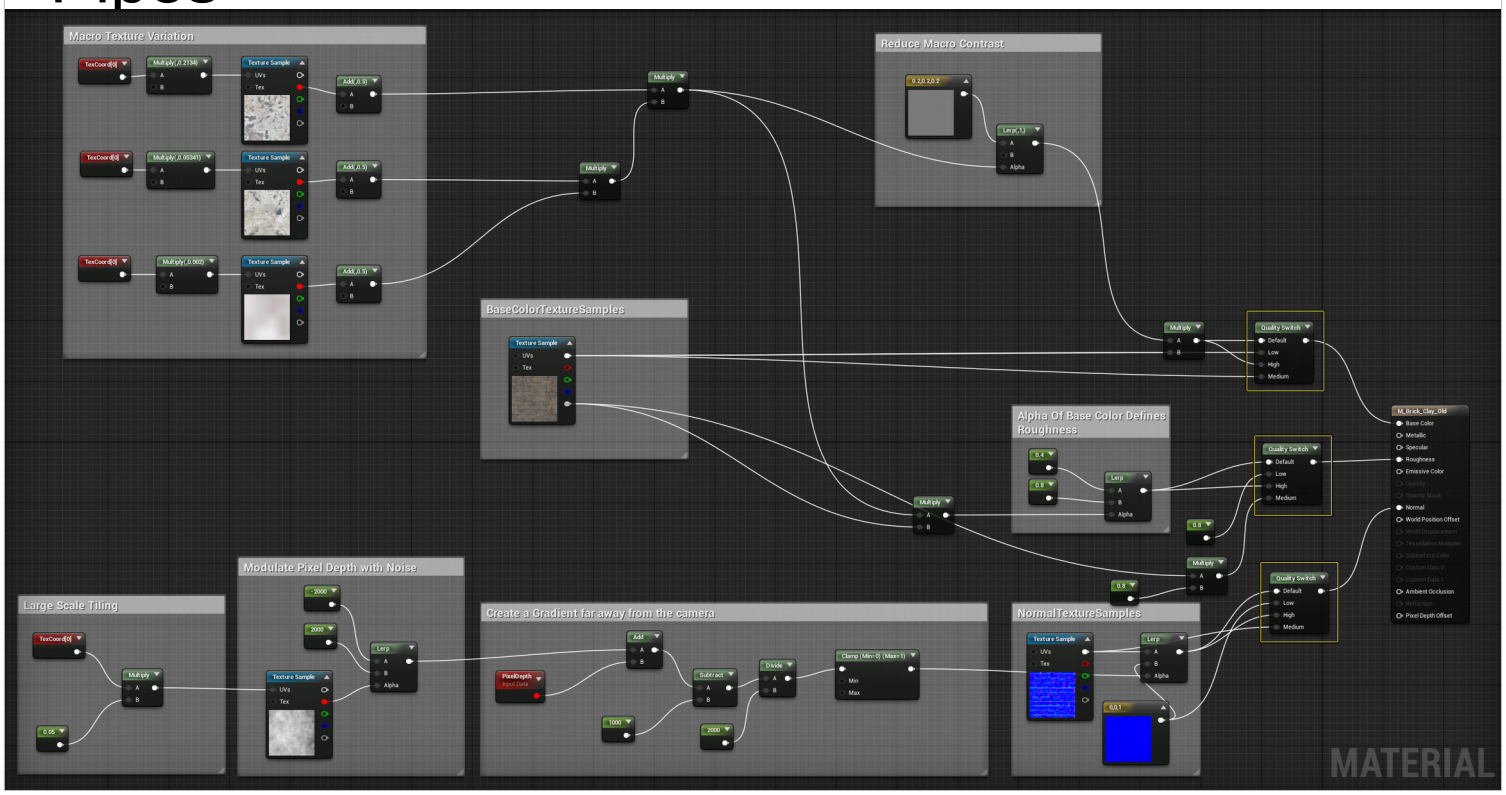


Images allow easier handling of 2D data.

Intended for use on GPU's, where images make use of hardware implementations of image operations in texture units.

Like CUDA, there is support for 2D data, called "Images". Using these allows you to make use of the dedicated GPU texture hardware.

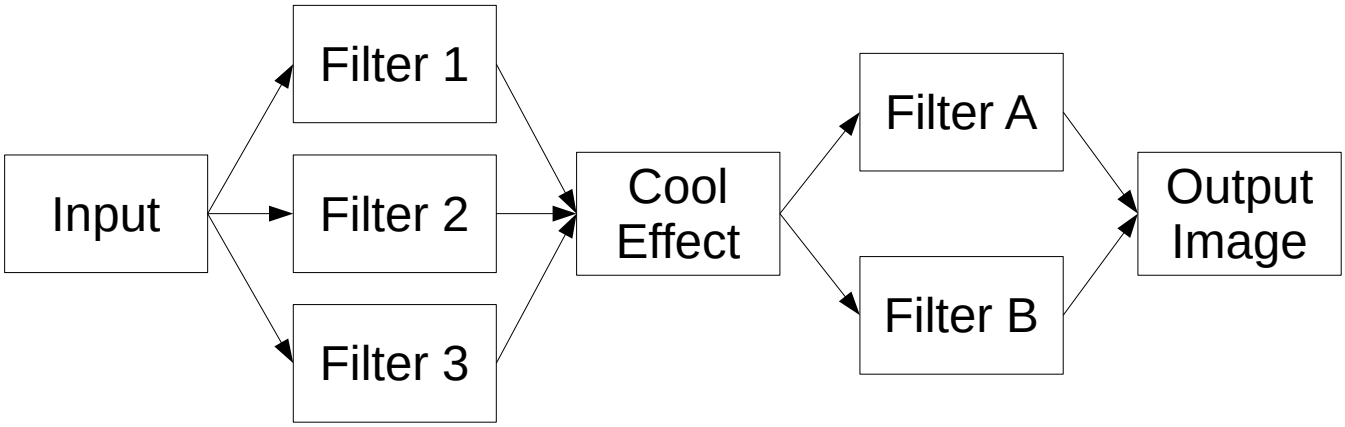
Pipes



Pipes are a concept that allows you to connect the output of one kernel to the input of another, “producer consumer” style.

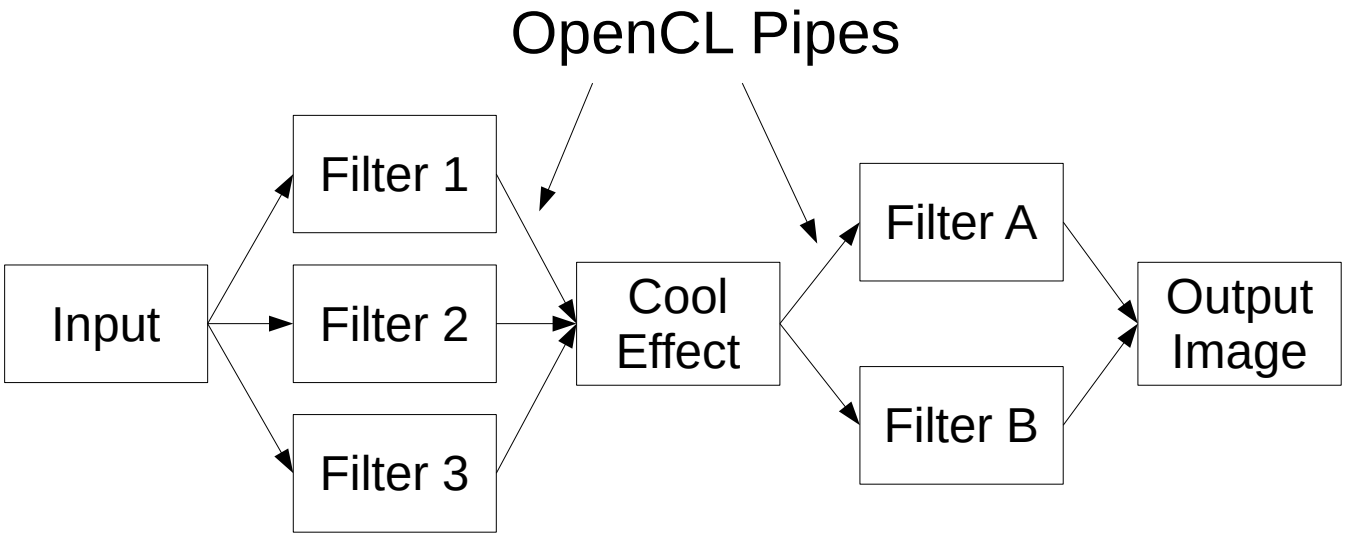
Pipes are essentially queues that are allocated in global memory, and you can put arbitrary data structures into them.

Pipes



This allows you to construct artbitrary pipelines that process data in multiple stages.

Pipes



Do note, however, that the order in which items are inserted and taken out of the pipes are not specified.

Pipes

Pipes are FIFO queues

Independent kernels can atomically read and write to them.

Pipes store a user-defined struct for communication

Allows simplified producer-consumer kernel interaction

```
read_pipe(pipe, &nextItemInPipe);
write_pipe(pipe, &itemToStoreInPipe);
```

Work group broadcast functions

```
// Test whether predicate is true for any or all
// work items in the work group
bool work_group_any(bool predicate);
bool work_group_all(bool predicate);


// Broadcast any value from one work item to all other
// work items in the work group
[any type] work_group_broadcast([any type], src_local_id);
```

Functions for 2D and 3D also exist

Since OpenCL is platform agnostic, there's no warp-level instrinsics like CUDA has. However, there are still some functions available for exchanging data between work-items.

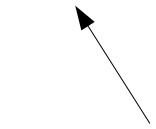
First, there are two "ballot" like functions, except you can only perform an AND and OR operation (all and any, respectively).

Second, you can broadcast arbitrary data from one work-item to all others inside the work-group. Note that src_local_id must be the same value for all work-items. So you can't use it like a shuffle instruction of sorts.

Work group broadcast functions

OpenCL supports reduce and scan operations

```
[any type] work_group_reduce_<op>([any type]);
[any type] work_group_scan_inclusive_<op>([any type]);
[any type] work_group_scan_exclusive_<op>([any type]);
```



Operator to use.
For example, add or max

A reduction allows you to reduce a value over all work-items in the work.group. Here <op> is replaced with a the operator of your choice, which can either be add, min, and max.

Scan returns the reduced value up to the index of the work item's index within the work-group. In the inclusive case, it includes the value of the work-item itself, and it does not for the exclusive variety.

Other features:

Event system

Multiple command queues

Out of order command queues

Precompiled kernels

There are other features in OpenCL, but I won’t go into those apart from mentioning they exist.

OpenCL’s Future

OpenCL is still in development, but currently it’s moving towards integration with the Vulkan API. Over time, the two are going to be merged.

Next week: The FINAL Lecture!

Send me specific things you'd like me to explain!

Have a look through the previous slide sets before the lecture!

Link:

https://docs.google.com/forms/d/e/1FAIpQLSfEOE4N6FD_q0ujHKvXEP3SupPcHHoYWf21J0WIcfzorer-5A/viewform