Introduction to OpenCL

David Black-Schaffer
david.black-schaffer@it.uu.se

UPMARC Uppsala Programming for Multicore Architectures Research Center

Disclaimer

- I worked for Apple developing OpenCL
- I'm biased

  (But not in the way you might think…)

What is OpenCL?

Low-level language for high-performance heterogeneous data-parallel computation.

- Access to all compute devices in your system:
  - CPUs
  - GPUs
  - Accelerators (e.g., CELL…but that only exists on PS3 now)
- Based on C99
  - Good (familiar)
  - Bad (ancient, low-level language)
- Portable across devices
  - Vector intrinsics and math libraries
  - Guaranteed precision for operations
- Open standard

Low-level -- doesn't try to do everything for you, but…

High-performance -- you can control all the details to get the maximum performance. This is essential to be successful as a performance-oriented standard. (Things like Java have succeeded here as standards for reasons other than performance.)

Heterogeneous -- runs across all your devices; same code runs on any device.

Data-parallel -- this is the only model that supports good performance today. OpenCL has task-parallelism, but it is largely an after-thought and will not get you good performance on today's hardware.

Vector intrinsics will map to the correct instructions automatically. This means you don't have to write SSE code anymore and you'll still get good performance on scalar devices.

Open Standard - 2008

- Good industry support
- Driving hardware requirements

KHRONOS GROUP

This is a big deal. Note that the big three hardware companies are here (Intel, AMD, and Nvidia), but that there are also a lot of embedded companies (Nokia, Ericsson, ARM, TI). This standard is going to be all over the place in the future. Notably absent is Microsoft with their competing direct compute standard as part of DX11.

Note how this support grew in just one year…

OpenCL vs. CUDA

- CUDA has better tools, language, and features
- OpenCL supports more devices

- But they're basically the same
  - If you can figure out how to make your algorithm run well on one it will work well on the other
  - They both strongly reflect GPU architectures of 2009

OpenCL support will get better, but the language will never evolve as fast as CUDA since it is designed by a committee of competitors.

However, CUDA is very unlikely to start supporting a wide range of devices anytime soon. (At least not for free.)

As OpenCL gets better CUDA's advantage will decrease. The major missing components at the moment are tools and libraries

What is OpenCL Good For?

- Anything that is:
  - Computationally intensive
  - Data-parallel
  - Single-precision*

Note this is because OpenCL was designed for GPUs, and GPUs are good at these things.

*This is changing, the others are not.

These three requirements are important. If your algorithm is not computationally intensive and data-parallel you are going to have a hard time getting a speedup on any 100+ core architecture like a GPU. This is not going to change significantly in the future, although there will be more support for non-data-parallel models. So if you can adjust your algorithm to this model you will be doing yourself a favor for whatever architecture/programming system is popular in the future.

Computational Intensity

- Proportion of **math** ops : **memory** ops
  Remember: memory is slow, math is fast

- Loop body: Low-intensity:

```
A[i] = B[i] + C[i]          1:3
A[i] = B[i] + C[i] * D[i]   2:4
A[i]++                      1:2
```

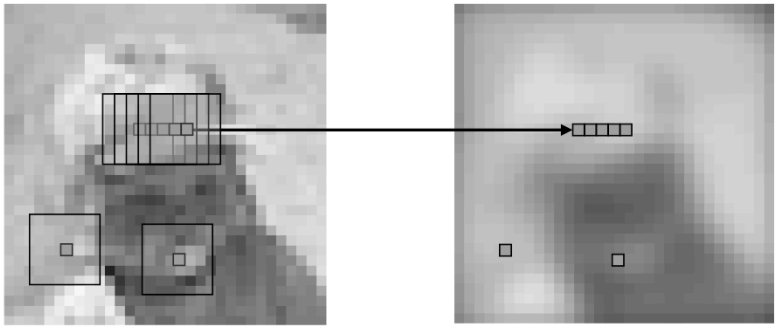
- Loop body: High(er)-intensity:

```
Temp+= A[i]*A[i]            2:1
A[i] = exp(temp)*acos(temp) X:1
```

This is a reminder of how important this is from my previous lecture.

Data-Parallelism

- Same *independent* operations on lots of data*
- Examples:
  - Modify every pixel in an image with *the same* filter
  - Update every point in a grid using *the same* formula

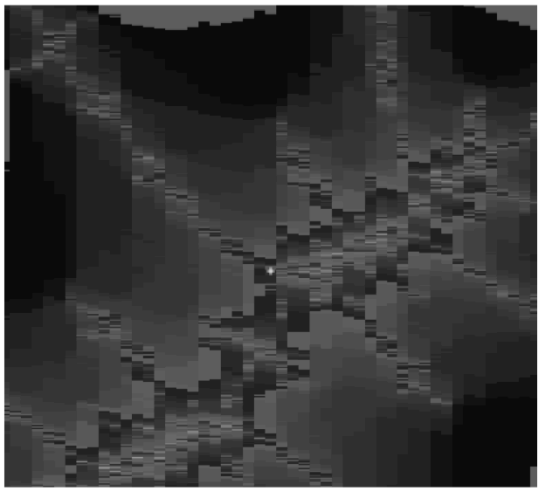


*Performance *may* fall off a cliff if not exactly the same.

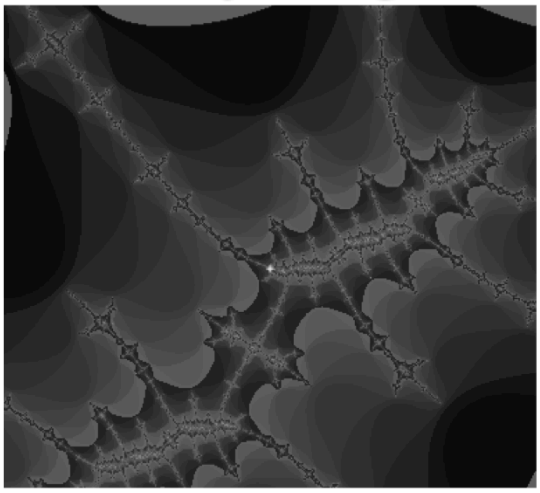
In the image each output pixel is generated by operating on a set of input pixels. Each output result is independent of the other output results, consists of an identical calculation, and therefore can be done in parallel. This algorithm allows OpenCL to run each pixel calculation in parallel, thereby maximizing throughput.

Single Precision

32 bits should be enough for anything…



Single Precision



Double Precision

(Expect double precision everywhere in ~1 year.)
Q: Will double precision be slower? Why?

Double precision on high-end cards (Nvidia Fermi, AMD) is available at approximately half the single-precision performance. More importantly, you only need half the bandwidth to access single-precision data. Try to take advantage of this wherever you can.

Key OpenCL Concepts

- **Global and Local Dimensions**
  - Specify parallelism
- **Compute Kernels**
  - Define computation
- **OpenCL Architecture**
  - Asynchronous command submission
  - Manual data movement

Global and Local Dimensions

How you specify parallelism.

Global Dimensions

- Parallelism is defined by the 1D, 2D, or 3D **global dimensions** for each kernel execution
- A **work-item** (thread) is executed for every point in the global dimensions

- Examples

1k audio:	1024	1024 work-items
HD video:	1920x1080	2M work-items
3D MRI:	256x256x256	16M work-items
HD per line:	1080	1080 work-items
HD per 8x8 block:	240x135	32k work-items

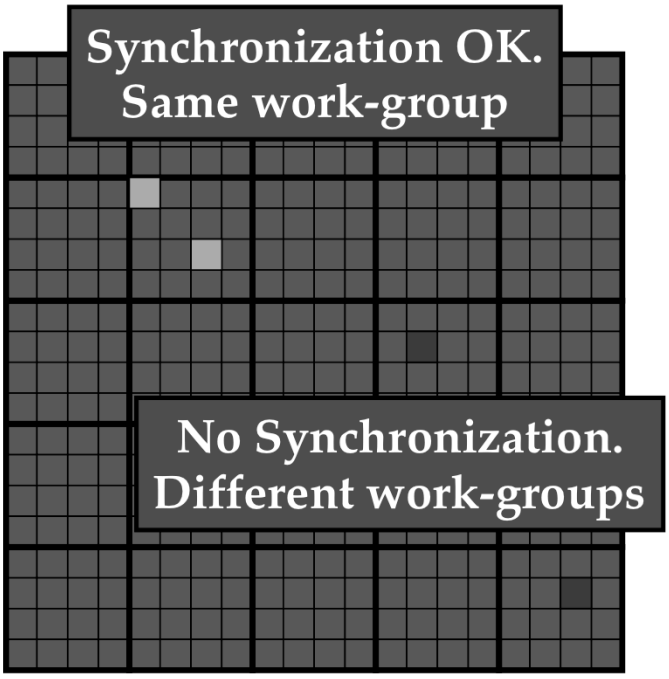
Note that the correct global dimensions for a problem depend on what you want to do. If you want to process each pixel of an HD image in parallel, then 1920x1080 is the right size. If you want to process each line in parallel, then 1080x1x1 would be better, or if you want to process the image in 8x8 blocks, you would use 240x135.

Local Dimensions

- The global dimensions are broken down **evenly** into **local work-groups**
  - Global Dimensions: 100x512
  - Local Dimensions: 10x8, 100x1, 50x2, 2x128
  - Invalid Local Dimensions: 10x10, 16x16
- Each work-group is logically executed together on one **compute unit**
  (Nvidia Streaming Multiprocess)
- Synchronization is **only** allowed between **work-items in the same work-group**

Local dimensions define how work-items are grouped together for execution on the same compute-unit. The local dimensions must divide the global dimensions evenly, and are limited in total size by the hardware resources.

Local Dimensions and Synchronization

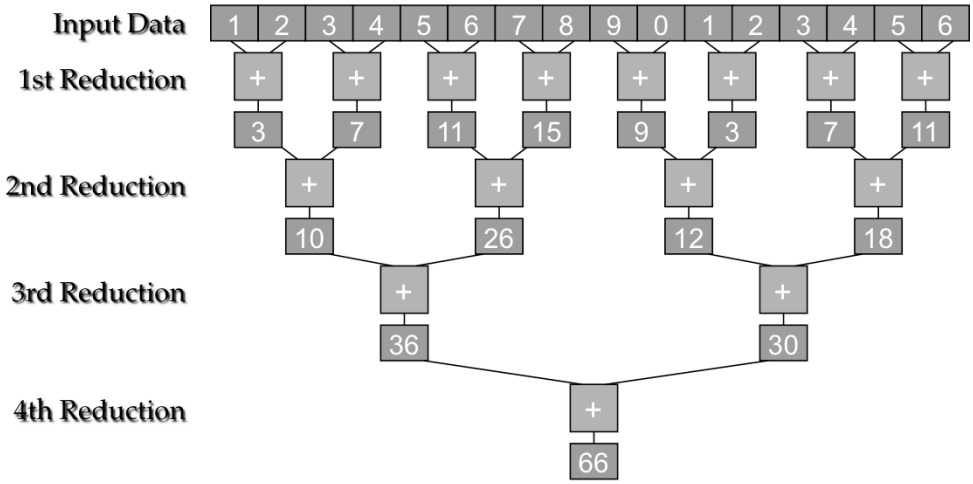


Global domain: 20x20
Work-group size: 4x4

Work-group size limited by hardware. (~512)

Implications for algorithms: e.g., reduction size.

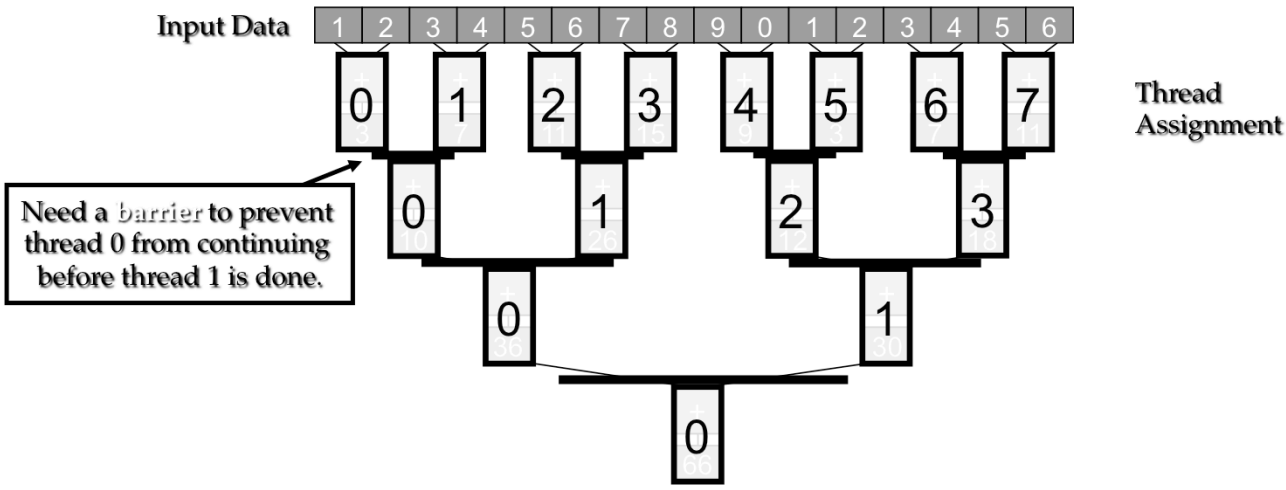
Synchronization Example: Reduction



Note: Reductions can include sums, min, max, product, etc.

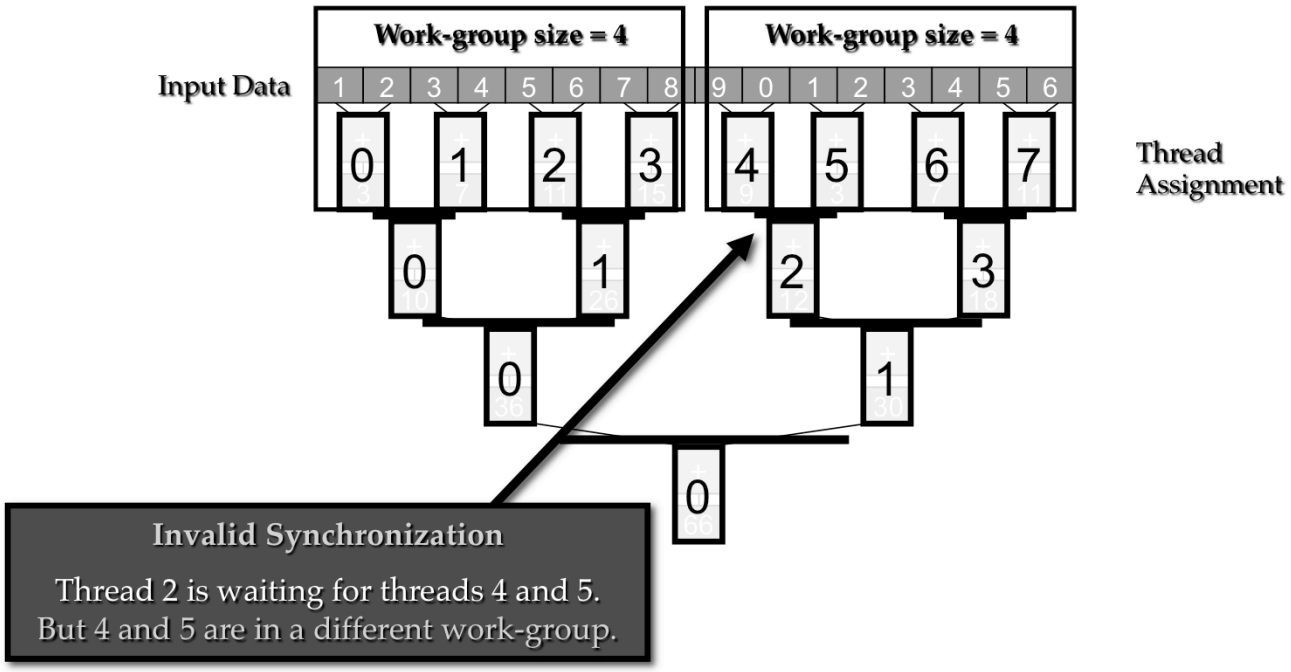
Parallel reduction does the reduction on sets of data at each step, thereby reducing the amount of data at each step.

Synchronization Example: Reduction



When assigning threads to do the reduction in parallel, each step needs to wait for the threads in the previous step to finish so it can be sure the results are valid before it continues. In this case, thread 0 needs to wait for thread 1 at each step.

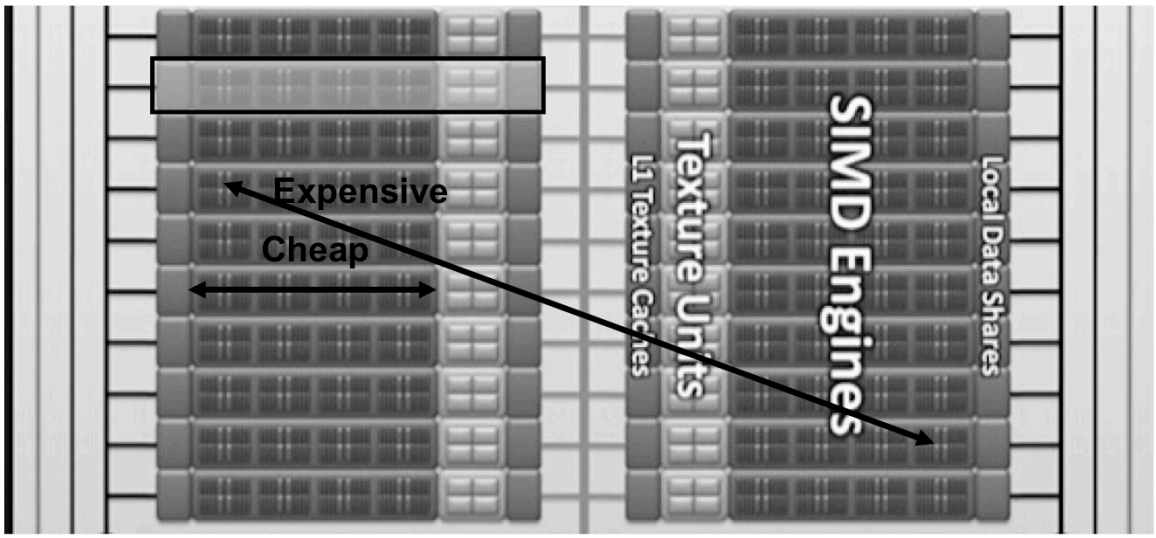
Synchronization Example: Reduction



In OpenCL, the work-group size can play an important role here. If the work-group size is too small, the reduction may need to synchronize across work-groups which is not supported in OpenCL. Here thread 2 on the second reduction step is trying to wait for the results of threads 4 and 5, which are in a different work-group. Since this type of synchronization is not supported, the results will be undefined. To handle this in OpenCL you need to restructure your algorithm.

Why Limited Synchronization?

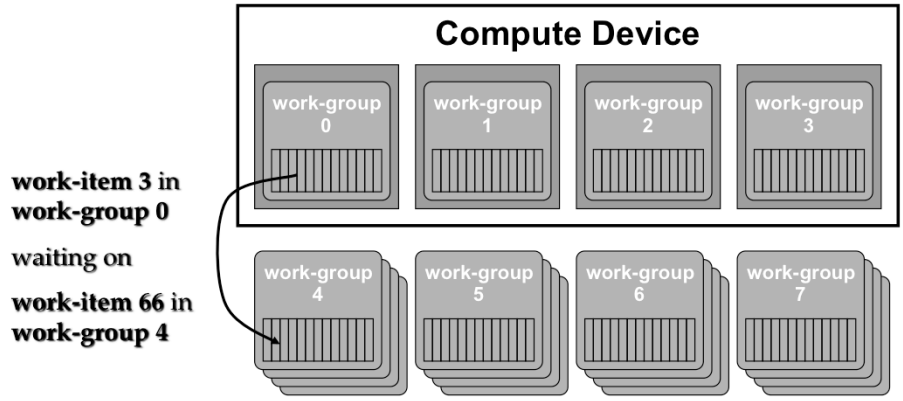
- Scales well in hardware
  - Only work-items within a work-group need to communicate
  - GPUs run 32-128 work-groups in parallel



This type of scaling is going to be the case for all architectures. If you can keep your synchronization local (even if the hardware supports global) you will get better performance.

What About Spinlocks in OpenCL?

```
while (!lock[n]) {}
```



Problem: no guarantee that work-group 4 will get to run until work-group 0 finishes: no forward progress.

Spinlocks are explicitly not allowed between work-groups in OpenCL because there is no guarantee that the scheduler on the device will make forward progress.

E.g., in this example, the scheduler may have decided that work-group 4 will run on the same compute unit as work-group 0, and it may well wait for work-group 0 to finish before running 4.

This would mean that work-group 4 would never run (because work-group 0 is waiting for work-group 4 before it will finish) and the kernel will hang.

Until there are guarantees about the thread schedulers, this type of synchronization is not permitted in OpenCL.

With that said, on Nvidia hardware, at least, if you have no more work-groups than streaming multiprocessors, you can get away with this.

Global Synchronization

- OpenCL only supports global synchronization at the end of a kernel execution.
- Very expensive.

(Newer hardware will support this more flexibly, but slowly.)

Each stage of your reduction needs to be a separate kernel to ensure that all the work-items from the previous stage are synchronized. This is painful.

Choosing Dimensions

- **Global dimensions**
  - Natural division for the problem
  - Too few: no latency hiding (GPU; SMT CPU)
  - Too many: too much overhead (CPU)
  - In general:
    - **GPU: >2000**
      (multiple of 16 or 64)
    - **CPU: ~2*# CPU cores**
      (Intel does some cool stuff here…)
- **Local dimensions**
  - May be determined by the algorithm
  - Optimize for best processor utilization
    (hardware-specific)

Picking the best local dimension size is very hardware dependent. Most GPUs operate on chunks of 16 or 64 work-items at a time, so you want to make sure your local dimensions are an even multiple of that value. (If not, some portion of the hardware will be unutilized.) Unfortunately the fine-tuning of this parameter is algorithm and hardware dependent, so there is no way to know the optimal number without testing.

Device Utilization

- Work-groups run together on compute units
- If the size of the work-group is not matched to the size of the compute unit you waste cores
- Example:
  - Global size 1300x2000, local size 13x4
  - Each work-group size: 13*4=52
  - If the hardware has 16 cores per compute unit:
    - We use them 100% of the cores for the first 16*3 work-items
    - But we then have 4 left over, so we waste 12 for the last threads
    - Utilization: 52/(16*4)=81%

Each compute unit runs one work-group:

Work-items in the compute-units are executed in a time-multiplexed manner in some size group. (Nvidia has a group size of 32; AMD 64.) If your work-group size is not an even multiple of this magic number then some of your hardware resources will be wasted. In general it is better to pad your input data size up to some nice multiple to allow your work-groups to be a nice multiple of this size. This ensures you don't waste resources on every work-group due to underutilization.

Questions so far?

computational intensity
data parallelism
global and local dimensions
synchronization
reductions
device utilization…

Compute Kernels

How you do your computations.

OpenCL Kernels

- A unit of code that is executed in parallel
- C99 syntax (no recursion or function ptrs)
- Think of the kernel as the "inner loop"

```
Regular C:

void calcSin(float *data) {
  for (int id=0; id<1024; id++)
    data[id] = sin(data[id]);
}
```

```
OpenCL Kernel:

void kernel calcSin(global float *data) {
  int id = get_global_id(0);
  data[id] = sin(data[id]);
}
```

The C code is run in parallel by having OpenCL split up the outer loop in parallel. Each compute kernel then determines which work it should do by calling get_global_id(), and then doing the work for that iteration.

OpenCL Kernel C

- Basic C99
  - With all the bad things that entails
- Plus…
  - Vectors (portability)
  - Rounding and conversions (performance)
  - Intrinsic functions (accuracy)

(Pointers to more information at the end of the slides.)

C99 is not really a language anyone wants to write in. It is, however, decent for performance because it doesn't provide any high-level features that get in your way. OpenCL adds some very nice features, whose guaranteed availability and precision enhance portability, performance, and accuracy.

OpenCL C - Intrinsics

gentype **exp** (gentype *x*)
gentype **exp2** (gentype)
gentype **exp10** (gentype)

gentype **expm1** (gentype *x*)
gentype **fabs** (gentype)
gentype **fdim** (gentype *x*, gentype *y*)
gentype **floor** (gentype)

gentype **fma** (gentype *a*,
gentype *b*, gentype *c*)

gentype **fmax** (gentype *x*, gentype *y*)

gentype **fmax** (gentype *x*, float *y*)

gentype **fmin**[42] (gentype *x*, gentype *y*)

gentype **fmin** (gentype *x*, float *y*)

gentype **fmod** (gentype *x*, gentype *y*)
gentype **fract** (gentype *x*,
__global gentype **iptr*)[43]
gentype **fract** (gentype *x*,
__local gentype **iptr*)
gentype **fract** (gentype *x*,
__private gentype **iptr*)

gentype **frexp** (gentype *x*,
__global int*n* *exp)
gentype **frexp** (gentype *x*,
__local int*n* *exp)
gentype **frexp** (gentype *x*,
__private int*n* *exp)

gentype **lgamma** (gentype *x*)
gentype **lgamma_r** (gentype *x*,
__global int*n* **signp*)
gentype **lgamma_r** (gentype *x*,
__local int*n* **signp*)
gentype **lgamma_r** (gentype *x*,
__private int*n* **signp*)

gentype **log** (gentype)
gentype **log2** (gentype)
gentype **log10** (gentype)
gentype **log1p** (gentype *x*)
gentype **logb** (gentype *x*)

gentype **mad** (gentype *a*,
gentype *b*, gentype *c*)

gentype **maxmag** (gentype *x*, gentype *y*)

gentype **minmag** (gentype *x*, gentype *y*)

gentype **modf** (gentype *x*,
__global gentype **iptr*)
gentype **modf** (gentype *x*,
__local gentype **iptr*)
gentype **modf** (gentype *x*,
private gentype **iptr*)

float*n* **nan** (uint*n* *nancode*)

gentype **nextafter** (gentype *x*,
gentype *y*)

gentype **hypot** (gentype *x*, gentype *y*)

int*n* **ilogb** (gentype *x*)

gentype **ldexp** (gentype *x*, int*n* *n*)

gentype **ldexp** (gentype *x*, int *n*)

gentype **acos** (gentype)
gentype **acosh** (gentype)
gentype **acospi** (gentype *x*)
gentype **asin** (gentype)
gentype **asinh** (gentype)
gentype **asinpi** (gentype *x*)
gentype **atan** (gentype *y_over_x*)
gentype **atan2** (gentype *y*, gentype *x*)
gentype **atanh** (gentype)
gentype **atanpi** (gentype *x*)
gentype **atan2pi** (gentype *y*, gentype *x*)
gentype **cbrt** (gentype)
gentype **ceil** (gentype)

gentype **copysign** (gentype *x*, gentype *y*)

gentype **cos** (gentype)
gentype **cosh** (gentype)
gentype **cospi** (gentype *x*)
gentype **erfc** (gentype)
gentype **erf** (gentype)
gentype **pow** (gentype *x*, gentype *y*)
gentype **pown** (gentype *x*, int*n* *y*)
gentype **powr** (gentype *x*, gentype *y*)
gentype **remainder** (gentype *x*,
gentype *y*)

gentype **remquo** (gentype *x*,

gentype **rint** (gentype)

gentype **rootn** (gentype *x*, int*n* *y*)
gentype **round** (gentype *x*)

gentype **rsqrt** (gentype)
gentype **sin** (gentype)
gentype **sincos** (gentype *x*,
__global gentype **cosval*)
gentype **sincos** (gentype *x*,
__local gentype **cosval*)
gentype **sincos** (gentype *x*,
__private gentype **cosval*)
gentype **sinh** (gentype)
gentype **sinpi** (gentype *x*)
gentype **sqrt** (gentype)
gentype **tan** (gentype)
gentype **tanh** (gentype)
gentype **tanpi** (gentype *x*)
gentype **tgamma** (gentype)
gentype **trunc** (gentype)

OpenCL C - (faster) Intrinsics

- Explicitly trade-off precision and performance
  - `navtive_` - fastest; no accuracy guarantee
  - `half_` - faster; less accuracy

gentype **native_log2** (gentype *x*)
gentype **native_log10** (gentype *x*)
gentype **native_powr** (gentype *x*, gentype *y*)
gentype **native_recip** (gentype *x*)
gentype **native_rsqrt** (gentype *x*)
gentype **native_sin** (gentype *x*)
gentype **native_sqrt** (gentype *x*)
gentype **native_tan** (gentype *x*)

gentype **half_cos** (gentype *x*)
gentype **half_divide** (gentype *x*, gentype *y*)
gentype **half_exp** (gentype *x*)
gentype **half_exp2** (gentype *x*)
gentype **half_exp10** (gentype *x*)
gentype **half_log** (gentype *x*)
gentype **half_log2** (gentype *x*)
gentype **half_log10** (gentype *x*)
gentype **half_powr** (gentype *x*, gentype *y*)
gentype **half_recip** (gentype *x*)
gentype **half_rsqrt** (gentype *x*)
gentype **half_sin** (gentype *x*)
gentype **half_sqrt** (gentype *x*)
gentype **half_tan** (gentype *x*)

gentype **native_cos** (gentype *x*)
gentype **native_divide** (gentype *x*, gentype *y*)
gentype **native_exp** (gentype *x*)
gentype **native_exp2** (gentype *x*)
gentype **native_exp10** (gentype *x*)
gentype **native_log** (gentype *x*)

native_ calls have no accuracy guarantee and vary from device to device.

Utility Functions

- Information about each work-item
  - **get_global_id(dim)**
    current work-item's ID in a particular dimension

```
int id = get_global_id(0);
data[id] = sin(data[id]);
```

  - **get_work_dim()**
    number of global dimensions in use
  - **get_global_size(dim)**
    number of global work-items in a particular dimension
  - **get_local_size(), get_local_id(), get_num_groups(), get_group_id()**
    information about the local dimensions
- Determine what work each work-item does.

These utility functions are used to figure out what part of the work each work-item should be doing. In some sense this is inefficient because each thread has to re-compute this information, but that's the tradeoff for "easy" data parallelism.

OpenCL Intrinsics

- Guaranteed availability in OpenCL
- Guaranteed precision in OpenCL

(These are explicitly tested for all OpenCL conformant devices.)

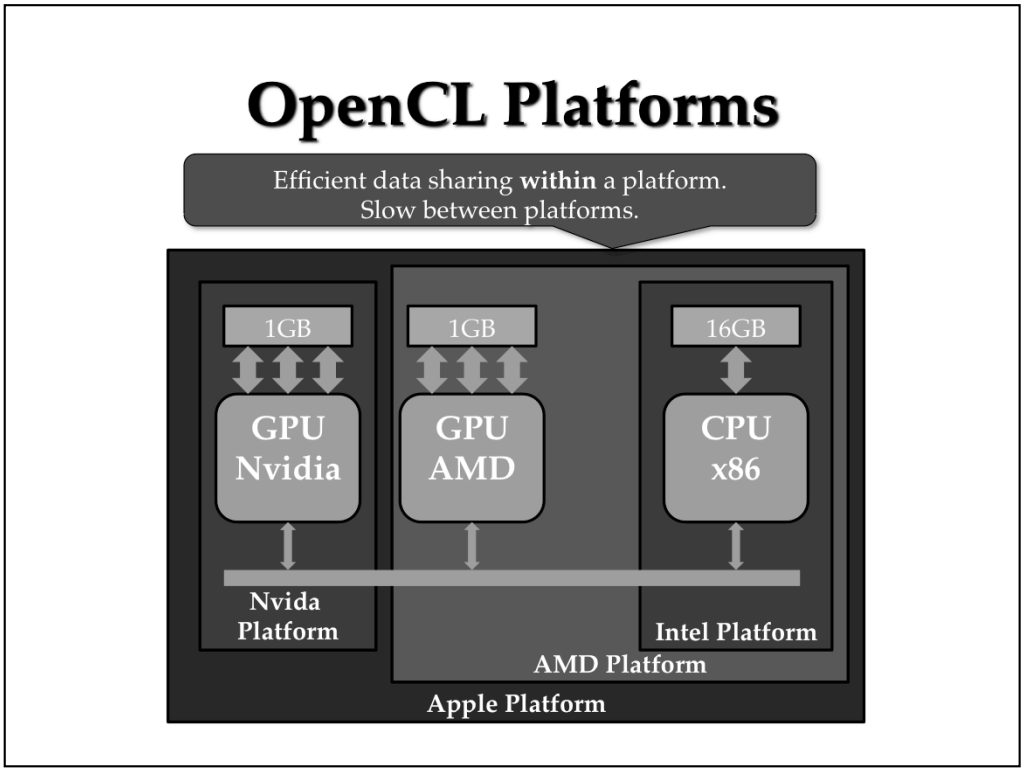
- Enhances portability and performance
- Control of performance/precision tradeoff

Questions so far?

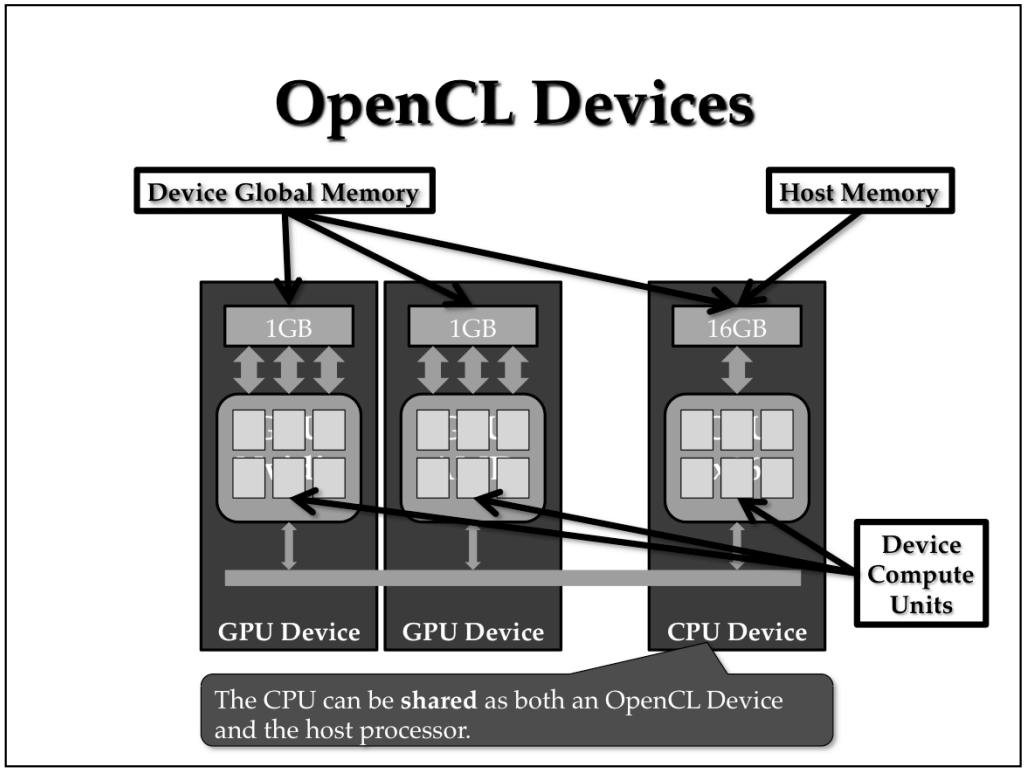
kernels
OpenCL kernel C
intrinsic functions
utility functions…

OpenCL Architecture

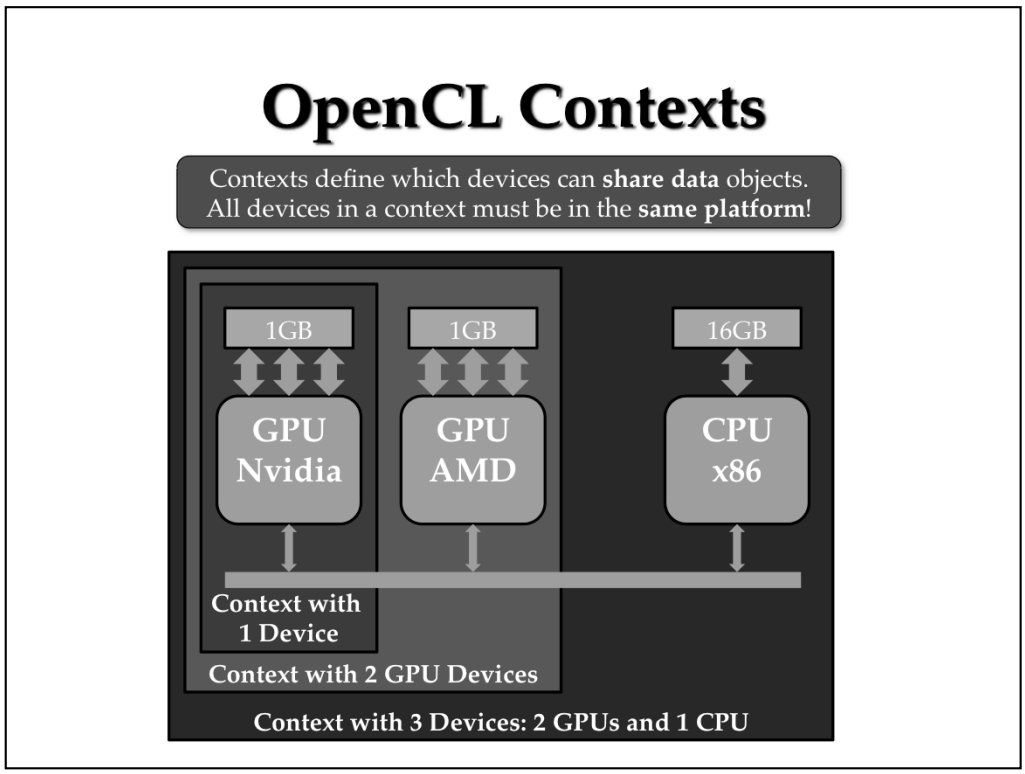
Asynchronous command submission
Manual data movement



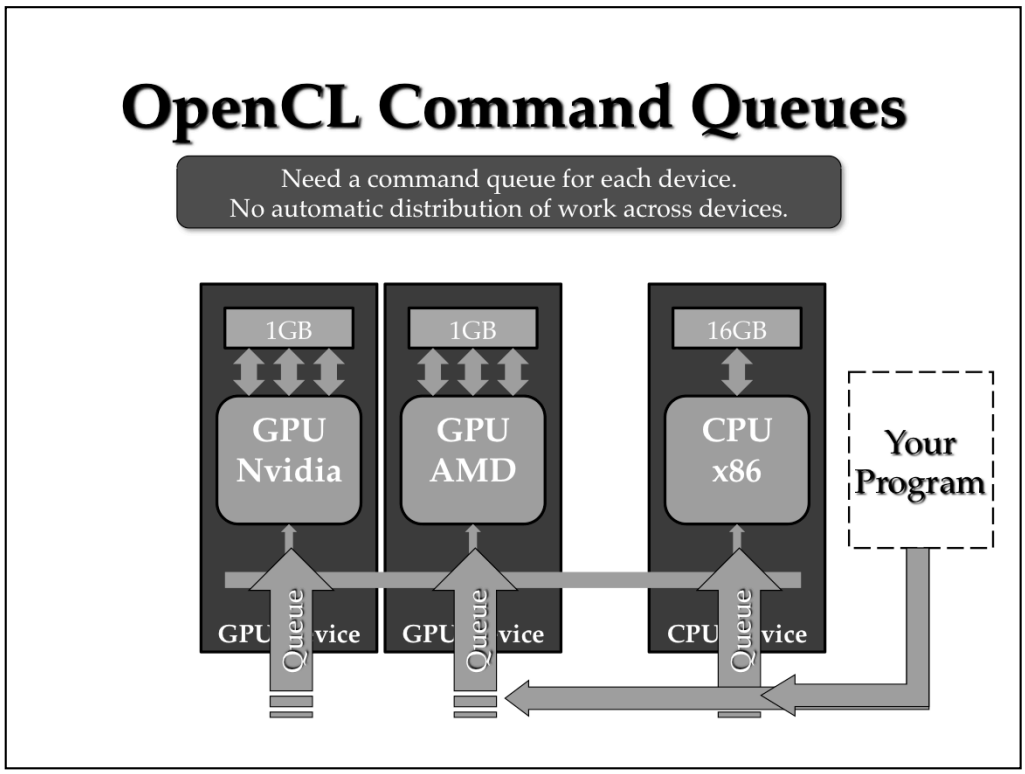
To enable competitor's software to work together, OpenCL uses a level of indirection called the platform. When you select a platform, you basically get all commands for that platform shipped off to that vendor's libraries. This means there is no optimization for moving data between devices on different platforms. So if you have devices from different vendors (and you're not on a Mac) you will have to manually copy data back and forth to the host to copy it between devices.



Each device has a global memory and some number of compute units. The host processor can also be a device, in which case its processor cores and memory are shared between the host execution and OpenCL.



In OpenCL you create contexts that contain multiple devices to enable sharing of data between them. By doing this you enable the OpenCL runtime to move data efficiently between those devices.

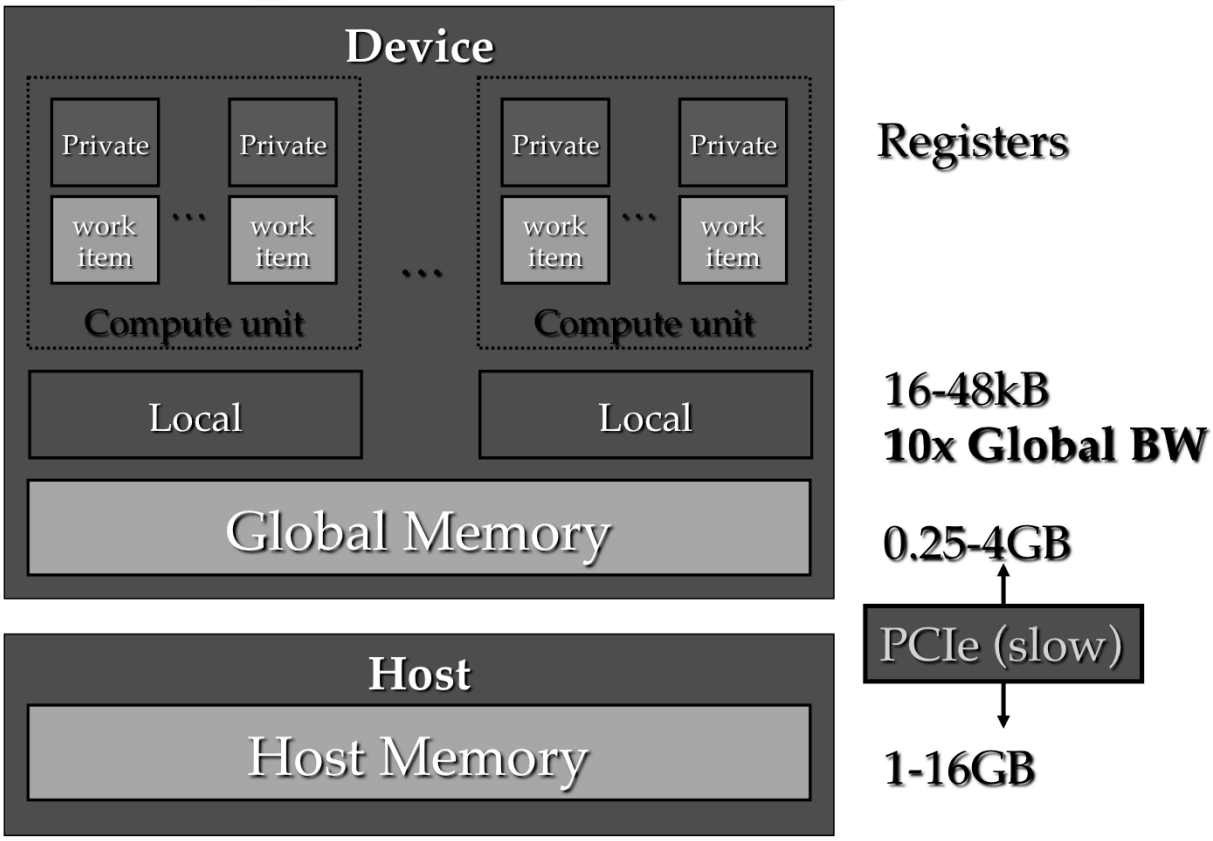


To use the devices you submit work to them (asynchronously) through command queues. You need at least one queue per device, and you must manually divide up work and choose devices.

Data Movement

Getting your data to the device…
and back.

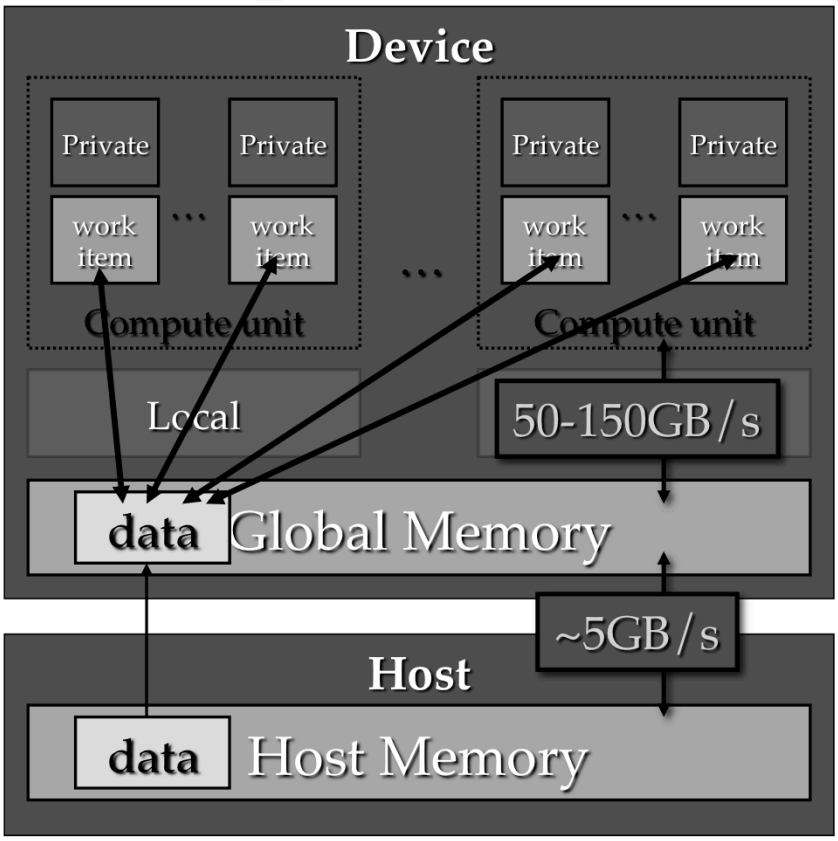
OpenCL Memory Model



Each compute-unit on the compute-device (e.g., CPU or GPU) processes a number of work-items in parallel. In the architecture example shown earlier, the GPU compute-unit had 8 processor cores and could execute 8 work-items in parallel. CPUs typically execute 1 work-item per compute-unit in parallel. Note that this is a physical mapping, whereas the logical mapping may be different. In practice GPUs may run many more work-items per compute-unit by time multiplexing them. The only requirement of OpenCL is that every work-group be run on one physical compute-unit so all work-items in the work-group can synchronize.

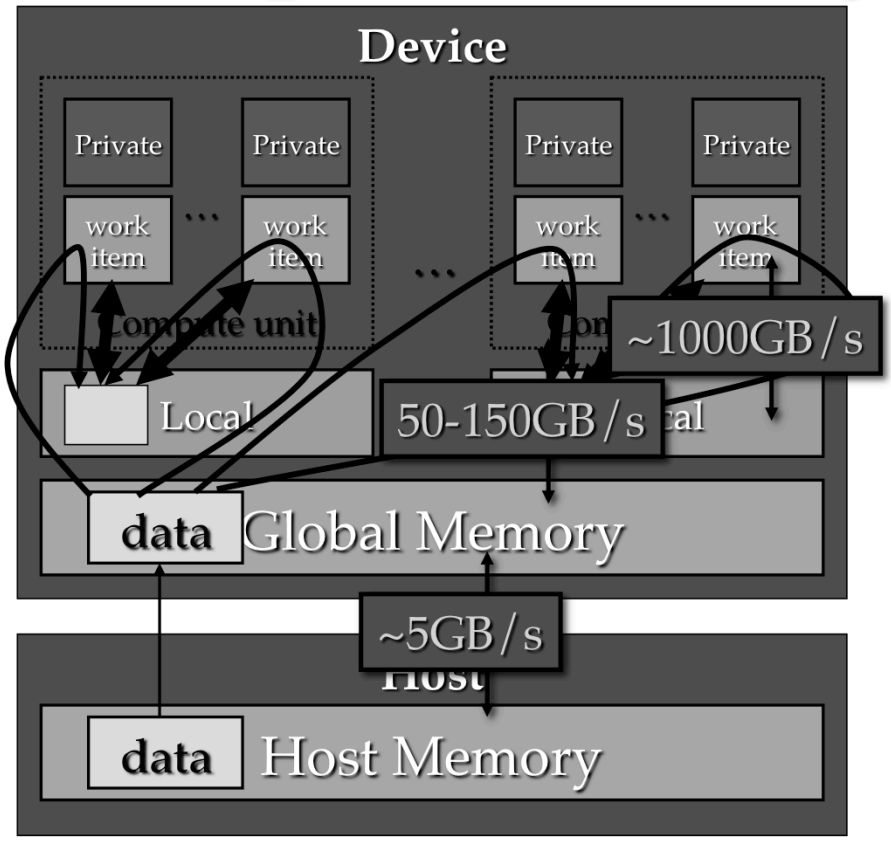
Note how this memory model looks a lot like a GPU…

OpenCL Memory Model



The user must manually allocate and move data to the global memory. From there all work-items can access it.

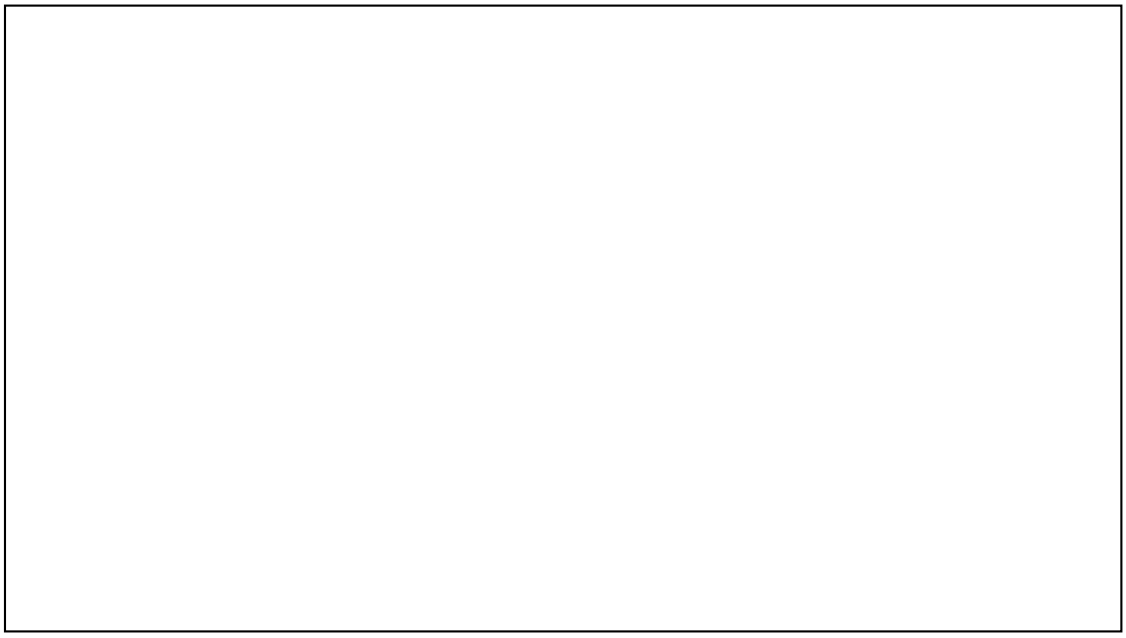
OpenCL Memory Model



Using local memory is much more complicated. The user must not only allocate the data, but have the kernel code running on each work-item copy the appropriate data from the global memory into the local memory before it can be used. This is quite complicated (as are all software-managed memories) but the advantage is a tremendous amount of bandwidth.

Moving Data

- No automatic data movement
- You must explicitly:
  - **Allocate** global data
  - **Write** to it from the host
  - **Allocate** local data
  - **Copy** data from global to local (and back)
- But…
  - You get full control for performance! (Isn't this great?)



Accessing Memory: Question

- How much data does this access from main memory?

```
float a = big_array[n];
```

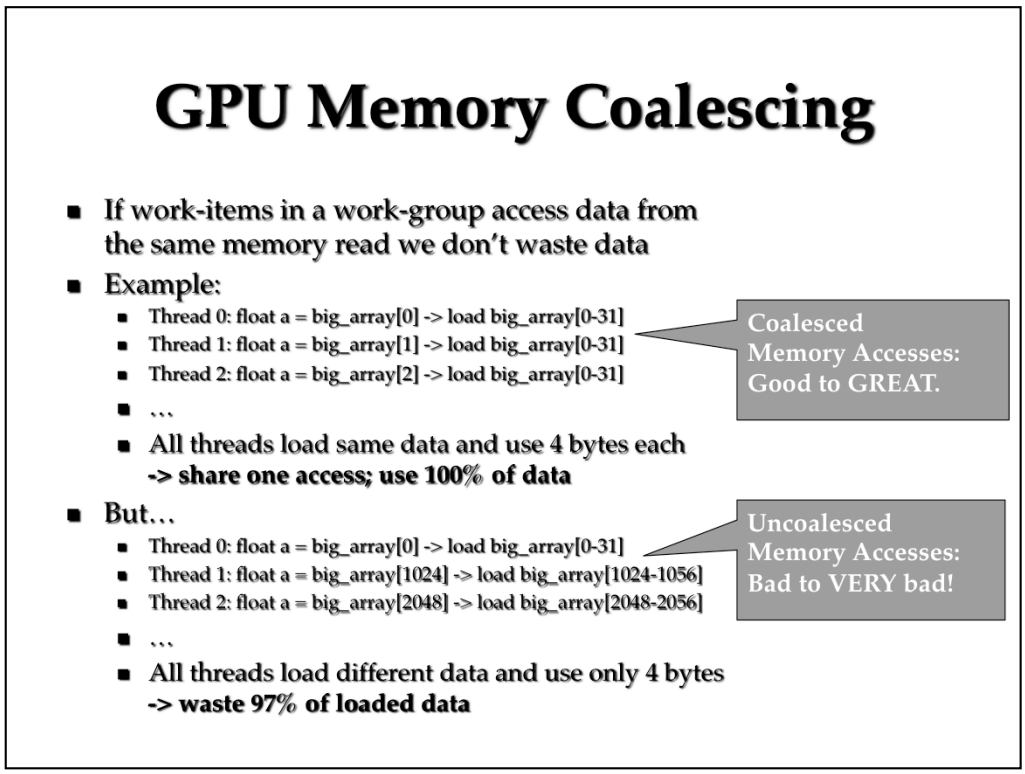
- Pre-question: How much if big_array[n] is:
  - In the cache…?
  - Not in the cache…?
- Pre-pre question: A float is how many bytes?
  - 4 = 32bits

Accessing Memory: Answer

- Kind of a trick question…
  - You don't know the details of the machine
- Assume a modern Intel processor
  - Each cache entry (line) is 64 bytes = 16 floats
  - Each DRAM access fetches 2 cache lines, 128 bytes = 32 floats
- So, if I access 1 float from DRAM…
  - Hardware will load 128 bytes = 32 floats
- `float a = big_array[n]` => **loads 32 floats !!**

Coalescing Memory

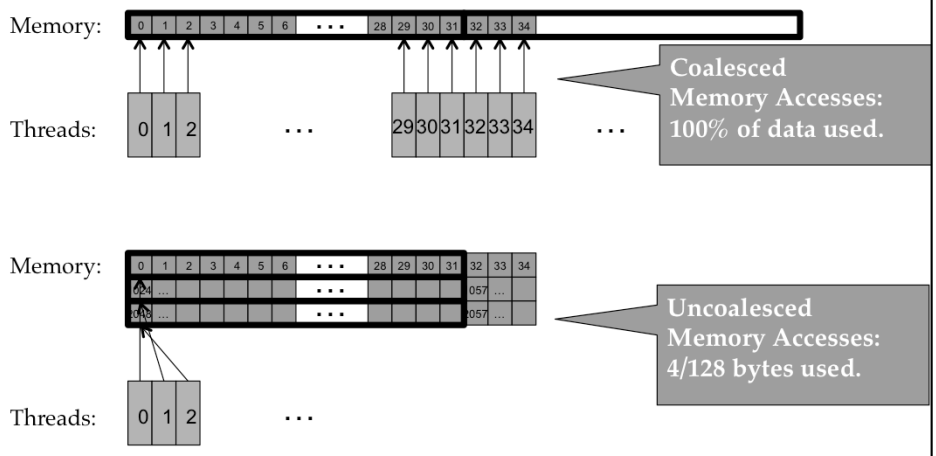
- `float a = big_array[n]` => **loads 32 floats**
- Is this a problem?
  - **No**, if I keep the other 31 floats around and use them soon (100% bandwidth)
  - **Yes**, if I don't keep them around or don't use them soon (3% bandwidth)
- On a CPU:
  - Keep around and use soon through the **cache**
- On a GPU:
  - **No cache** (except new cards!)
  - Other threads need to use them at the same time
    => Memory Access Coalescing



If each thread (work-item) in a group accesses the next element in order, the hardware can coalesce these memory accesses into one access and more efficiently utilize the bandwidth. Newer hardware has better support for this (e.g., they can be out-of-order). This is a first-order optimization for performance.

GPU Memory Coalescing

- If work-items in a work-group access data from the same memory read we don't waste data



Same as previous slide, but graphically.

Questions so far?

platforms/devices/contexts/queues
asynchronous execution
global/local memory
data movement
coalescing…

OpenCL Hello World

Your first OpenCL program…

An OpenCL Program

1. Setup
    1. Get the devices
    2. Create a context (for sharing between devices)
    3. Create command queues (for submitting work)
2. Compilation
    1. Create a program
    2. Build the program (compile)
    3. Create kernels
3. Create memory objects
4. **Enqueue writes** to initialize memory objects
5. Set the kernel arguments
6. **Enqueue kernel** executions
7. **Enqueue reads** to get back data
8. **Wait** for your commands to finish

```
// Setup OpenCL
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, (cl_command_queue_properties)0, NULL);

// Define our kernel. It just calculates the sin of the input data.
char *source = {
    "kernel void calcSin(global float *data) {\n"
    "  int id = get_global_id(0);\n"
    "  data[id] = sin(data[id]);\n"
    "}\n"
};

// Compile the kernel
program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "calcSin", NULL);

// Create the memory object
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, NULL);

// Copy the data to the input
clEnqueueWriteBuffer(queue, buffer, CL_FALSE, 0, DATA_SIZE, data, 0, NULL, NULL);

// Execute the kernel
clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
size_t global_dimensions[] = {LENGTH,0,0};
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
clEnqueueReadBuffer(queue, buffer, CL_FALSE, 0, sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Wait for everything to finish
clFinish(queue);
```

OpenCL Hello World: Calculate sine(x) in parallel.

```
// Setup OpenCL
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, (cl_command_queue_properties)0, NULL);

// Define our kernel. It just calculates the sin of the input data.
char *source = {
    "kernel void calcSin(global float *data) {\n"
    "  int id = get_global_id(0);\n"
    "  data[id] = sin(data[id]);\n"
    "}\n"
};

// Compile the kernel
program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "calcSin", NULL);

// Create the memory object
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, NULL);

// Copy the data to the input
clEnqueueWriteBuffer(queue, buffer, CL_FALSE, 0, DATA_SIZE, data, 0, NULL, NULL);

// Execute the kernel
clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
size_t global_dimensions[] = {LENGTH,0,0};
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
clEnqueueReadBuffer(queue, buffer, CL_FALSE, 0, sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Wait for everything to finish
clFinish(queue);
```

1. Setup: Call clGetDeviceIDs and specify the number and type of device(s) you want.

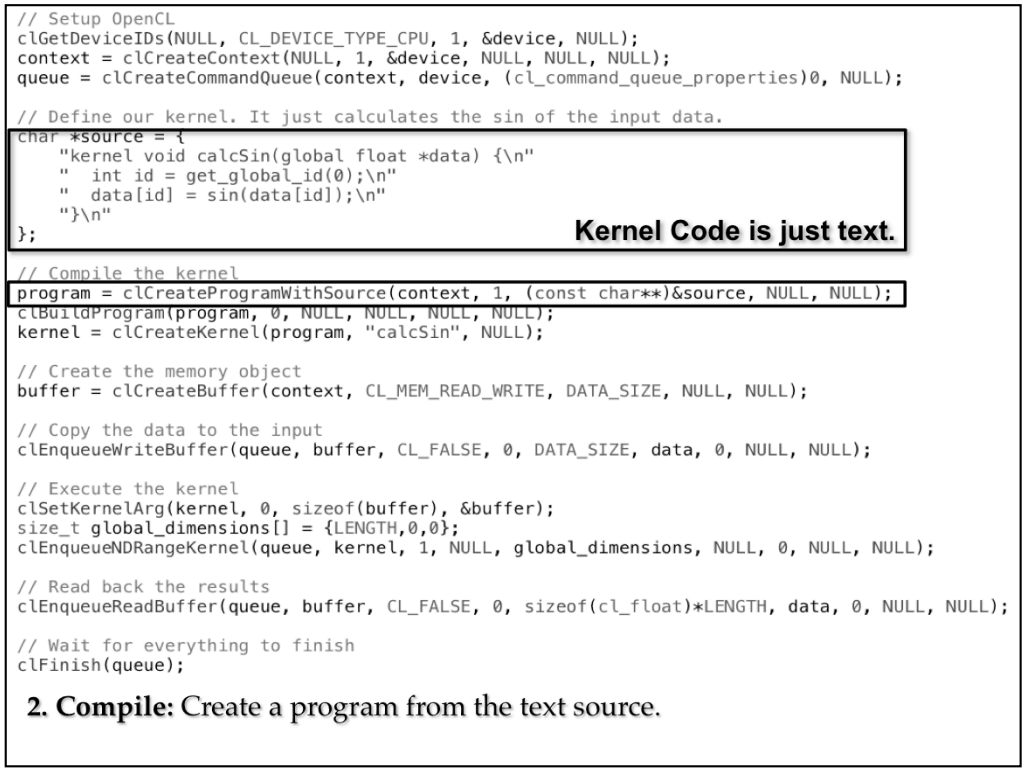
You can call clGetDeviceIDs with a null device list and a pointer to an int to get back the number of devices available. Note that this code does not look for a platform first.

```
// Setup OpenCL
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, (cl_command_queue_properties)0, NULL);

// Define our kernel. It just calculates the sin of the input data.
char *source = {
    "kernel void calcSin(global float *data) {\n"
    "  int id = get_global_id(0);\n"
    "  data[id] = sin(data[id]);\n"
    "}\n"
};

// Compile the kernel
program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "calcSin", NULL);

// Create the memory object
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, NULL);

// Copy the data to the input
clEnqueueWriteBuffer(queue, buffer, CL_FALSE, 0, DATA_SIZE, data, 0, NULL, NULL);

// Execute the kernel
clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
size_t global_dimensions[] = {LENGTH,0,0};
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
clEnqueueReadBuffer(queue, buffer, CL_FALSE, 0, sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Wait for everything to finish
clFinish(queue);
```

1. Setup: Create a context and specify the device(s) you want to have in the context.

```
// Setup OpenCL
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, (cl_command_queue_properties)0, NULL);

// Define our kernel. It just calculates the sin of the input data.
char *source = {
    "kernel void calcSin(global float *data) {\n"
    "  int id = get_global_id(0);\n"
    "  data[id] = sin(data[id]);\n"
    "}\n"
};

// Compile the kernel
program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "calcSin", NULL);

// Create the memory object
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, NULL);

// Copy the data to the input
clEnqueueWriteBuffer(queue, buffer, CL_FALSE, 0, DATA_SIZE, data, 0, NULL, NULL);

// Execute the kernel
clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
size_t global_dimensions[] = {LENGTH,0,0};
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
clEnqueueReadBuffer(queue, buffer, CL_FALSE, 0, sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Wait for everything to finish
clFinish(queue);
```

1. Setup: Create a command queue for each device.



Loading kernel code from a text file is a better idea so you don't have to get the quotes right in the C code for the string.

```
// Setup OpenCL
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, (cl_command_queue_properties)0, NULL);

// Define our kernel. It just calculates the sin of the input data.
char *source = {
    "kernel void calcSin(global float *data) {\n"
    "  int id = get_global_id(0);\n"
    "  data[id] = sin(data[id]);\n"
    "}\n"
};

// Compile the kernel
program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "calcSin", NULL);

// Create the memory object
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, NULL);

// Copy the data to the input
clEnqueueWriteBuffer(queue, buffer, CL_FALSE, 0, DATA_SIZE, data, 0, NULL, NULL);

// Execute the kernel
clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
size_t global_dimensions[] = {LENGTH,0,0};
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
clEnqueueReadBuffer(queue, buffer, CL_FALSE, 0, sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Wait for everything to finish
clFinish(queue);
```

2. Compile: Compile (build) the program. This is slow…

```
// Setup OpenCL
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, (cl_command_queue_properties)0, NULL);

// Define our kernel. It just calculates the sin of the input data.
char *source = {
    "kernel void calcSin(global float *data) {\n"
    "  int id = get_global_id(0);\n"
    "  data[id] = sin(data[id]);\n"
    "}\n"
};

// Compile the kernel
program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "calcSin", NULL);

// Create the memory object
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, NULL);

// Copy the data to the input
clEnqueueWriteBuffer(queue, buffer, CL_FALSE, 0, DATA_SIZE, data, 0, NULL, NULL);

// Execute the kernel
clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
size_t global_dimensions[] = {LENGTH,0,0};
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
clEnqueueReadBuffer(queue, buffer, CL_FALSE, 0, sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Wait for everything to finish
clFinish(queue);
```

2. Compile: Create a kernel object for each kernel in your program by name.

```
// Setup OpenCL
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, (cl_command_queue_properties)0, NULL);

// Define our kernel. It just calculates the sin of the input data.
char *source = {
    "kernel void calcSin(global float *data) {\n"
    "  int id = get_global_id(0);\n"
    "  data[id] = sin(data[id]);\n"
    "}\n"
};

// Compile the kernel
program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "calcSin", NULL);

// Create the memory object
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, NULL);

// Copy the data to the input
clEnqueueWriteBuffer(queue, buffer, CL_FALSE, 0, DATA_SIZE, data, 0, NULL, NULL);

// Execute the kernel
clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
size_t global_dimensions[] = {LENGTH,0,0};
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
clEnqueueReadBuffer(queue, buffer, CL_FALSE, 0, sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Wait for everything to finish
clFinish(queue);
```

3. Create a memory object (buffer or image) and specify the size.

```
// Setup OpenCL
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, (cl_command_queue_properties)0, NULL);

// Define our kernel. It just calculates the sin of the input data.
char *source = {
    "kernel void calcSin(global float *data) {\n"
    "  int id = get_global_id(0);\n"
    "  data[id] = sin(data[id]);\n"
    "}\n"
};

// Compile the kernel
program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "calcSin", NULL);

// Create the memory object
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, NULL);

// Copy the data to the input
clEnqueueWriteBuffer(queue, buffer, CL_FALSE, 0, DATA_SIZE, data, 0, NULL, NULL);

// Execute the kernel
clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
size_t global_dimensions[] = {LENGTH,0,0};
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
clEnqueueReadBuffer(queue, buffer, CL_FALSE, 0, sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Wait for everything to finish
clFinish(queue);
```

4. Write to the memory object by **enqueuing** a write to the queue.
(Remember: commands are *asynchronous*!)

```
// Setup OpenCL
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, (cl_command_queue_properties)0, NULL);

// Define our kernel. It just calculates the sin of the input data.
char *source = {
    "kernel void calcSin(global float *data) {\n"
    "  int id = get_global_id(0);\n"
    "  data[id] = sin(data[id]);\n"
    "}\n"
};

// Compile the kernel
program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "calcSin", NULL);

// Create the memory object
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, NULL);

// Copy the data to the input
clEnqueueWriteBuffer(queue, buffer, CL_FALSE, 0, DATA_SIZE, data, 0, NULL, NULL);

// Execute the kernel
clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
size_t global_dimensions[] = {LENGTH,0,0};
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
clEnqueueReadBuffer(queue, buffer, CL_FALSE, 0, sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Wait for everything to finish
clFinish(queue);
```

5. Set the kernel arguments.

```
// Setup OpenCL
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, (cl_command_queue_properties)0, NULL);

// Define our kernel. It just calculates the sin of the input data.
char *source = {
    "kernel void calcSin(global float *data) {\n"
    "  int id = get_global_id(0);\n"
    "  data[id] = sin(data[id]);\n"
    "}\n"
};

// Compile the kernel
program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "calcSin", NULL);

// Create the memory object
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, NULL);

// Copy the data to the input
clEnqueueWriteBuffer(queue, buffer, CL_FALSE, 0, DATA_SIZE, data, 0, NULL, NULL);

// Execute the kernel
clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
size_t global_dimensions[] = {LENGTH,0,0};
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
clEnqueueReadBuffer(queue, buffer, CL_FALSE, 0, sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Wait for everything to finish
clFinish(queue);
```

6. Enqueue the *asynchronous* kernel execution, specifying the global dimensions. (And optionally the local dimensions.)

```
// Setup OpenCL
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, (cl_command_queue_properties)0, NULL);

// Define our kernel. It just calculates the sin of the input data.
char *source = {
    "kernel void calcSin(global float *data) {\n"
    "  int id = get_global_id(0);\n"
    "  data[id] = sin(data[id]);\n"
    "}\n"
};

// Compile the kernel
program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "calcSin", NULL);

// Create the memory object
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, NULL);

// Copy the data to the input
clEnqueueWriteBuffer(queue, buffer, CL_FALSE, 0, DATA_SIZE, data, 0, NULL, NULL);

// Execute the kernel
clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
size_t global_dimensions[] = {LENGTH,0,0};
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
clEnqueueReadBuffer(queue, buffer, CL_FALSE, 0, sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Wait for everything to finish
clFinish(queue);
```

7. Enqueue a read to the memory object to *asynchronously* read back the results.

```
// Setup OpenCL
clGetDeviceIDs(NULL, CL_DEVICE_TYPE_CPU, 1, &device, NULL);
context = clCreateContext(NULL, 1, &device, NULL, NULL, NULL);
queue = clCreateCommandQueue(context, device, (cl_command_queue_properties)0, NULL);

// Define our kernel. It just calculates the sin of the input data.
char *source = {
    "kernel void calcSin(global float *data) {\n"
    "  int id = get_global_id(0);\n"
    "  data[id] = sin(data[id]);\n"
    "}\n"
};

// Compile the kernel
program = clCreateProgramWithSource(context, 1, (const char**)&source, NULL, NULL);
clBuildProgram(program, 0, NULL, NULL, NULL, NULL);
kernel = clCreateKernel(program, "calcSin", NULL);

// Create the memory object
buffer = clCreateBuffer(context, CL_MEM_READ_WRITE, DATA_SIZE, NULL, NULL);

// Copy the data to the input
clEnqueueWriteBuffer(queue, buffer, CL_FALSE, 0, DATA_SIZE, data, 0, NULL, NULL);

// Execute the kernel
clSetKernelArg(kernel, 0, sizeof(buffer), &buffer);
size_t global_dimensions[] = {LENGTH,0,0};
clEnqueueNDRangeKernel(queue, kernel, 1, NULL, global_dimensions, NULL, 0, NULL, NULL);

// Read back the results
clEnqueueReadBuffer(queue, buffer, CL_FALSE, 0, sizeof(cl_float)*LENGTH, data, 0, NULL, NULL);

// Wait for everything to finish
clFinish(queue);
```

7. Wait for everything to finish. Commands are enqueued *asynchronously* so your host program will finish before your commands. You must wait!

OpenCL Hello World

- Setup
  - Get the device/context/queues
  - Compile your program
  - Create and initialize your memory objects
- Execution
  - **Enqueue *asynchronous* commands** (read/write/execute)
  - **Wait** for them to finish
  - Repeat until done

Questions so far?

Hello World
asynchronous enqueues
asynchronous execution
waiting for *asynchronous* execution
asynchrounous…

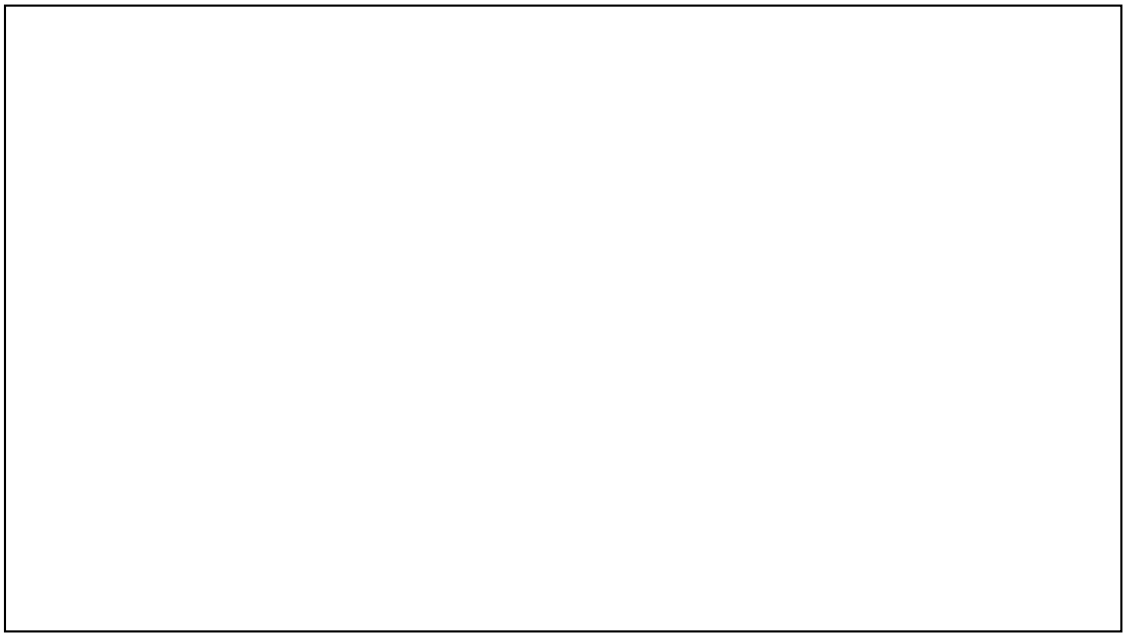
More OpenCL

Querying Devices

Images

Events

Optimization: slide 50



Querying Devices

- Lots of information via clGetDeviceInfo()
  - **CL_DEVICE_MAX_COMPUTE_UNITS***
    Number of compute units that can run work-groups in parallel
  - **CL_DEVICE_MAX_CLOCK_FREQUENCY***
    (When would it ever not be the max frequency?)
  - **CL_DEVICE_GLOBAL_MEM_SIZE***
    Total global memory available on the device
  - **CL_DEVICE_IMAGE_SUPPORT**
    Some devices don't support images
  - **CL_DEVICE_EXTENSIONS**
    double precision, atomic operations, OpenGL integration

*Unfortunately this doesn't tell you how much memory is available right now or which device will run your kernel fastest.

Use the * data carefully. They are all maximums and don't tell you how much you will actually get. Your best bet is to iteratively time your kernel execution and adjust parameters for best performance as you run.

Images

- **2D and 3D Native Image Types**
  - R, RG, RGB, RGBA, INTENSITY, LUMINANCE
  - 8/16/32 bit signed/unsigned, float
  - Linear interpolation, edge wrapping and clamping
- **Why?**
  - Hardware accelerated access (linear interpolation) on GPUs
  - Want to enable this fast path
  - GPUs cache texture lookups today
- **But…**
  - Slow on the CPU (which is why Larabee did this in HW)
  - Not all formats supported on all devices (check first)
  - Writing to images is not fast, and can be *very* slow

Not all devices support images. Some don't have the hardware and have to emulate them (CPUs, CELL).

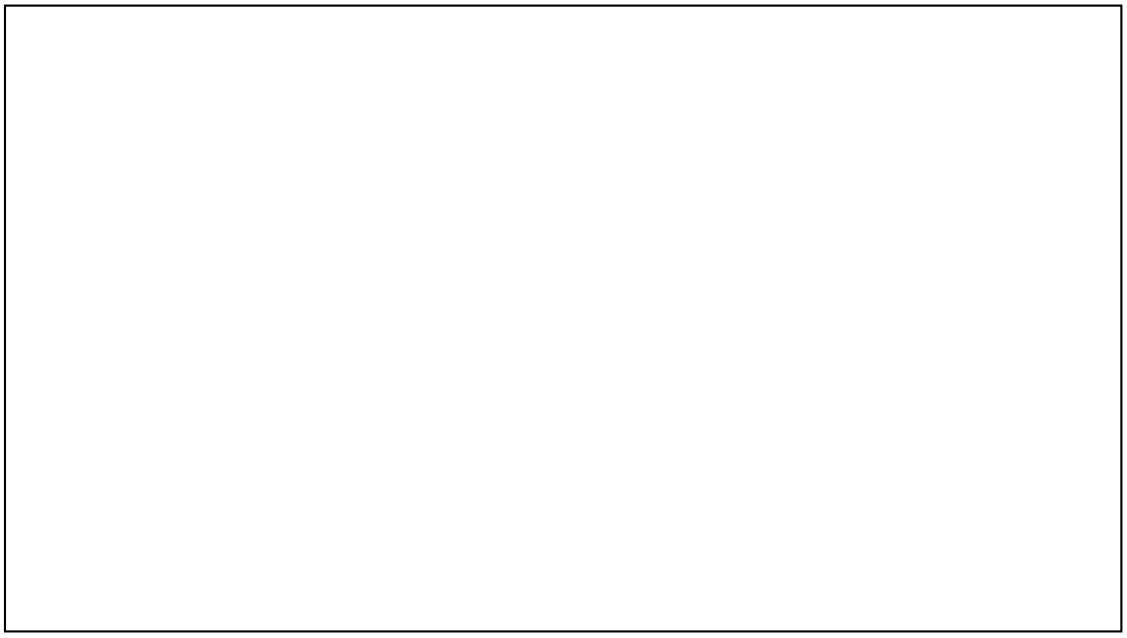
Events

- Subtle point made earlier:

  Queues for **different devices** are ***asynchronous*** **with respect to each other**

- Implication:
  - You must **explicitly synchronize** operations **between devices**

(Also applies to out-of-order queues)



Events

- Every clEnqueue() command can:
  - Return an **event** to track it
  - Accept a **list of events to wait for**

```
clEnqueueNDRangeKernel(queue, kernel,
                       1, NULL, global dimensions, NULL,
                       numberOfEventsInList, &waitList,
                       eventReturned);
```

- Events can also report profiling information
  - Enqueue->Submit->Start->End

Requesting events from OpenCL operations gives you a lot of flexibility to queue up multiple commands and then only wait for them all to finish. This can greatly increase performance. With OpenCL 1.1 you can even get callbacks when commands finish and insert user events to enable you to queue up commands that are waiting on the host program.

Event Example

- **Kernel A** output -> **Kernel B** input
- **Kernel A** runs on the CPU
- **Kernel B** runs on the GPU
- Need to ensure that **B** waits for **A** to finish

```
clEnqueueNDRangeKernel(CPU_queue, kernelA,
                       1, NULL, global_dimensions, NULL,
                       0, NULL, kernelA_event);

clEnqueueNDRangeKernel(GPU_queue, kernelB,
                       1, NULL, global_dimensions, NULL,
                       1, &kernelA_event, NULL);
```

OpenCL Performance

Optimization: slide 50

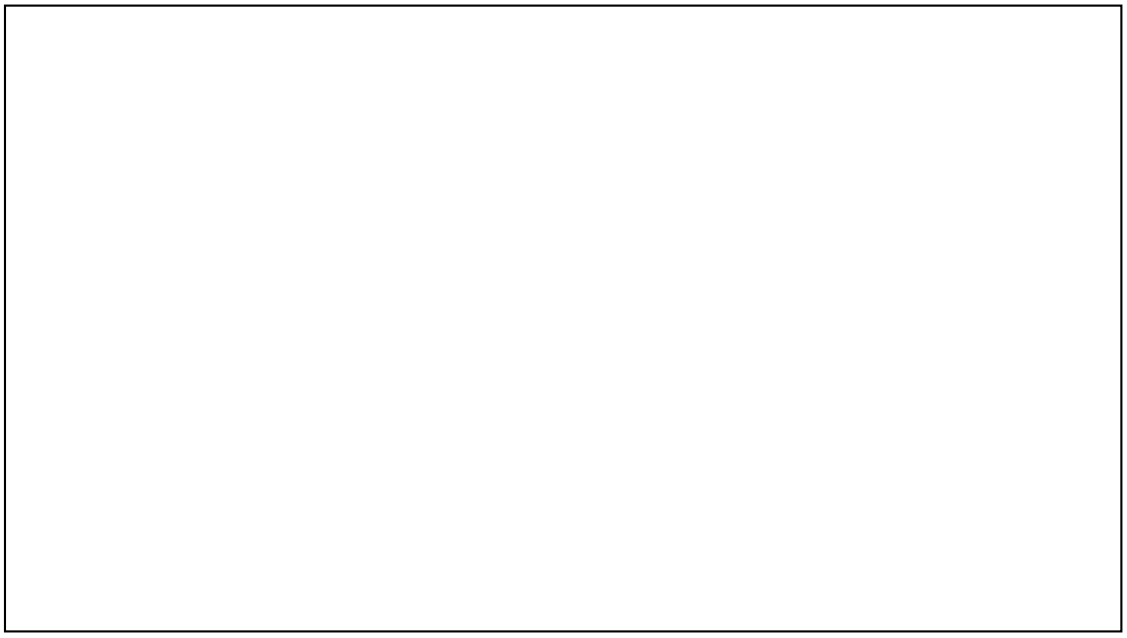
OpenCL GPU Performance Optimizations (Runtime)

- Host-Device Memory (**100x**)
  - PCIe is slow and has a large overhead
  - Do a lot of compute for every transfer
  - Keep data on the device as long as possible
  - Producer-consumer kernel chains (keep the data on the device)

Achilles Heel!

(Fusion will fix this.)

- Kernel Launch Overhead (**100x**)
  - First compile is very slow (ms)
  - Kernels take a long time to get started on the GPU
  - Amortize launch overhead with long-running kernels
  - Amortize compilation time with many kernel executions

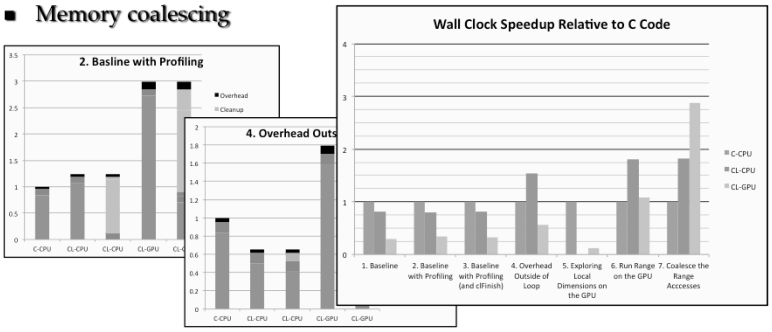


OpenCL GPU Performance Optimizations (Kernel)

- Memory Accesses (**~10x**)
  - Ordering matters for coalescing
  - Addresses should be sequential across threads
  - Newer hardware is more forgiving
- Local Memory (**~10x**)
  - Much larger bandwidth
  - Must manually manage
  - Look out for bank conflicts
- Divergent execution (up to **8x**)
- Vectors (**2-4x** on today's hardware)
  - On vector HW this is critical (AMD GPUs, CPUs)
  - OpenCL will scalarize automatically if needed
- Math (**2x** on intensive workloads)
  - fast_ and native_ variants may be faster (at reduced precision)

OpenCL Lab Session

- Performance optimization for a PDE solver
  - Writing host and kernel code
  - Profiling OpenCL
  - Understanding data transfer overheads
  - Understanding CPU/GPU differences
  - Memory coalescing



These are actual graphs you will produce during the lab session to understand the performance of your code.

OpenCL Debugging (Or Not)

- Poor debugging support on GPUs
  - Except for Nvidia (best on Windows)
- Advice:
  - Start on the CPU
  - At least you can use printf() and look at asembly…
- Watch out for system watchdog timers
  - Long-running kernels will lock the screen
  - Your kernel will be killed after a few seconds
  - Your app will crash
  - Your users will be sad

What is OpenCL? (Honestly)

Low-level[1] language[2] for high-performance[3] heterogeneous[4] data-parallel[5] computation.

1. Manual memory management and parallelization
   You choose the global dimensions and allocate data
2. A framework with C-like computation kernels
   Not really a language
3. If your algorithm is a good fit for the hardware
   E.g., data-parallel
4. Code is portable, but performance is not
   Different vendors/versions require different optimizations
5. Hardware and software only support data-parallel
   There is task-parallel support, but not on today's GPUs

Good Match for a GPU?

- Application Checklist:
  - Data-parallel?
  - Computationally intensive?
  - Avoid global synchronization?
  - Need lots of bandwidth?
  - Use single-precision?
  - Small caches okay?
- If yes, then you're all set.
- If not, consider changing algorithm.

Getting Started

- **CPU+GPU:**
  - AMD (linux/windows) or Nvidia on Mac
  - Intel has a CPU-only implementation
- **GPU:**
  - Nvidia (avoid AMD SIMDness)
  - Strongly recommend Fermi (caches)
  - Nvidia's Parallel Nsight for Visual Studio (Windows)
- **Debugging:**
  - Nvidia is the only player today

For CPU+GPU your best bet today is all AMD on linux windows or Nvidia+Intel on a Mac.

In general, Nvidia's GPU hardware and software on windows with visual studio is the best bet today.

References

- Apple's Developer Conference Tutorial Videos
  - Introduction and Advanced Sessions (Intel, AMD, and Nvidia)
  - http://developer.apple.com/videos/wwdc/2010/
- Nvidia's OpenCL Guides
  - Programming and Best Practices (somewhat Nvidia-specific)
  - http://developer.nvidia.com/object/opencl.html
- AMD's Introductory Videos
  - http://developer.amd.com/documentation/videos/OpenCLTechnicalOverviewVideoSeries/Pages/default.aspx

The Apple sessions are very good and the Nvidia documentation is excellent, but very Nvidia-centric.

Questions?

OpenCL C - More Intrinsics

- Many more…
  - Integer (mad24, abs, clamp, clz, …)
  - Common (clamp, degrees, max, step, sign…)
  - Geometric (cross, dot, distance, length, normalize)
  - Relational (isequal, isless, any, isnan, select…)
  - Vector load/store (vload_*type*, vstore_*type*, …)
  - Synchronization (barrier, mem_fence, …)
  - Async Local Memory Copies
  - Atomic (atomic_add, atomic_xchg, …)
  - Image Read/Write
  - …

OpenCL C - Vectors

- Automatically mapped to HW
  - AMD GPUs, Intel SSE
  - Intel tries to automatically run work-items across SSE!
- Lengths: 2, 4, 8, 16
  - Length 3 in OpenCL 1.1
- **Advice: use the natural vector size**
  - **Don't try to make everything 16-wide**
  - Graphics: RGBA, use 4-wide
  - Position: XYZ or XYZW, use 3- or 4-wide
- Examples
  - `float4 pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);`
  - `pos.xw = (float2)(5.0f, 6.0f);`
  - `float16 big.s01ef = pos;`
  - `float16 big.s2301 = (float4)(pos.hi, pos.even);`
  - Math and logical operations supported as expected
  - Conversions are explicit: `int4 f = convert_int4(pos);`

OpenCL C - Rounding Modes

- Explicit rounding modes for conversions
  - E.g., "convert float to int, rounding to nearest even"
  - Leverages HW support
  - **Avoid:** (int)floor(f + 0.5);
- Examples:
  - `int i = convert_int_rte(float f);`
  - `uchar8 c8 = convert_uchar8_rtz(double8 d);`
  - Supports: rte, rtz, rtp, rtn (default rtz)
  - Supports saturation: `convert_int_sat_rtz(…)`