# 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
- 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.

The precision is important as historically GPUs have not cared about accuracy as long as the images looked "good". These requirements are forcing them to take accuracy seriously.

## Open Standard - 2008

- Good industry support
- Driving hardware requirements

© Copyright Khronos Group, 2008

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.

## Huge Industry Support - 2010

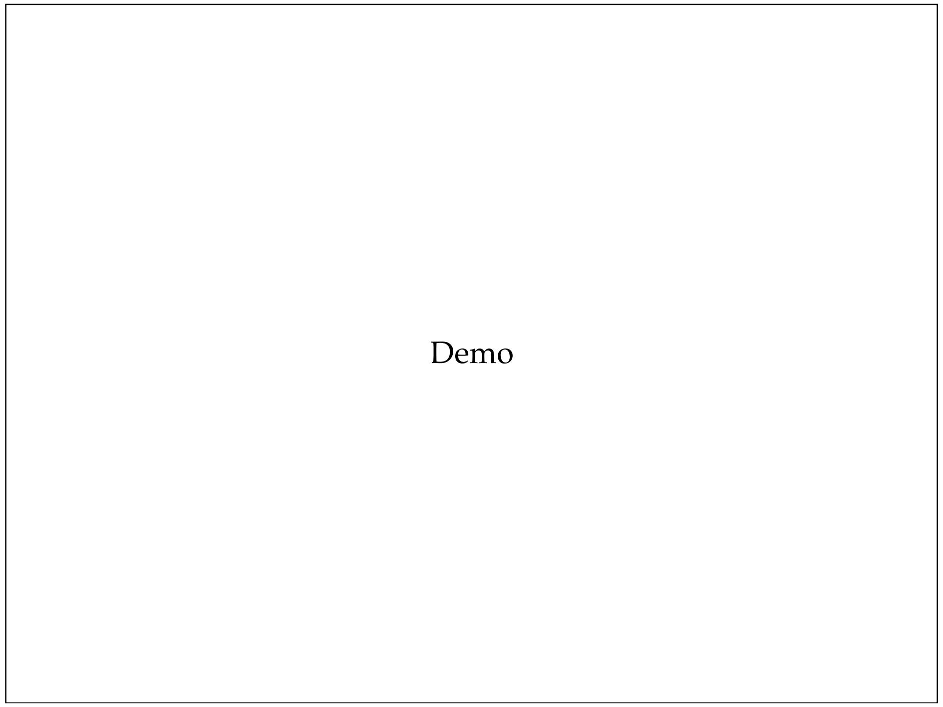
KHRONOS GROUP

Over 100 companies creating visual computing standards

Board of Promoters

© Copyright Khronos Group, 2010 - Page 2

Note how this support grew in just one year…



## Demo

The demo is a Mandelbrot fractal generator where you can see the performance difference between straight C code and OpenCL on the CPU, GPU, and combined CPU+GPU.

## 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

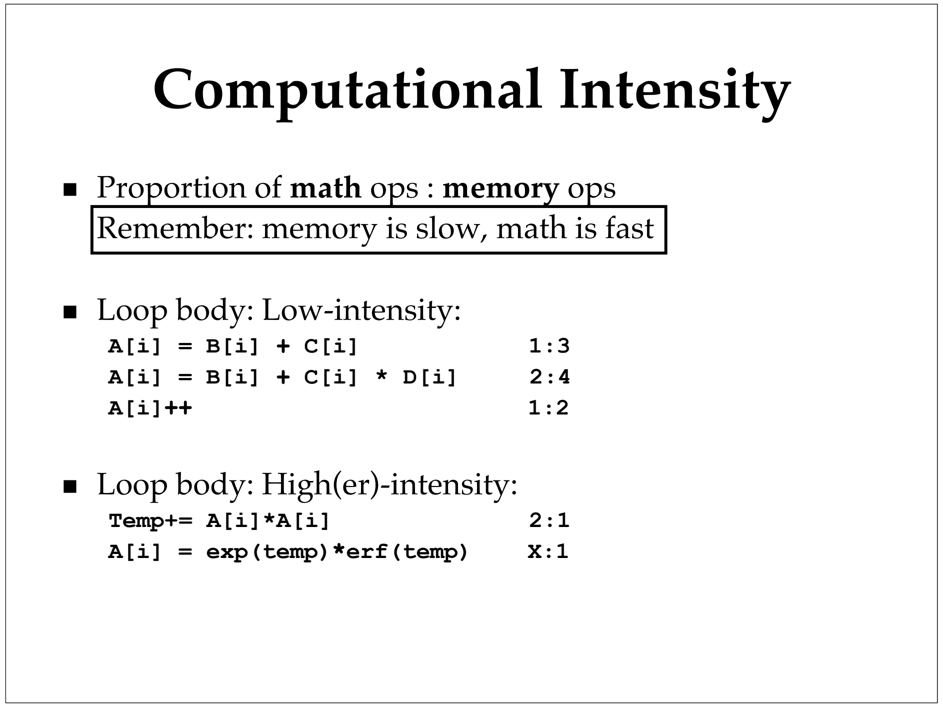
## What is OpenCL Good For?

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

I am going to focus on the GPU but OpenCL can run on the CPU as well.

*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

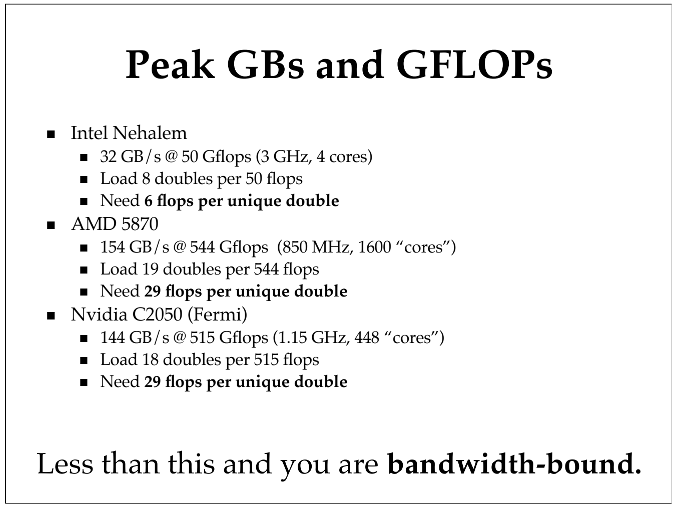
- Proportion of **math** ops : **memory** ops
  Remember: memory is slow, math is fast
- Loop body: Low-intensity:

```
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)*erf(temp)    X:1
```

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



## Peak GBs and GFLOPs

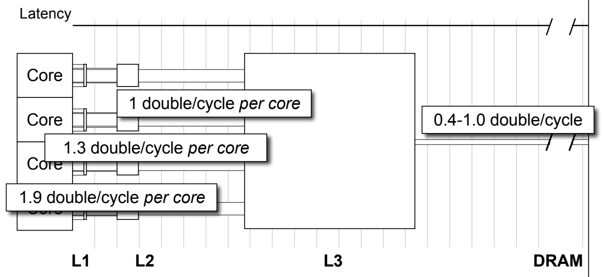
- Intel Nehalem
  - 32 GB/s @ 50 Gflops (3 GHz, 4 cores)
  - Load 8 doubles per 50 flops
  - Need **6 flops per unique double**
- AMD 5870
  - 154 GB/s @ 544 Gflops (850 MHz, 1600 “cores”)
  - Load 19 doubles per 544 flops
  - Need **29 flops per unique double**
- Nvidia C2050 (Fermi)
  - 144 GB/s @ 515 Gflops (1.15 GHz, 448 “cores”)
  - Load 18 doubles per 515 flops
  - Need **29 flops per unique double**

Less than this and you are **bandwidth-bound.**

Important! You should always have a feeling for your storage and bandwidth requirements when trying to estimate what performance to expect.

## Real World

Intel Nehalem 3GHz (2009)

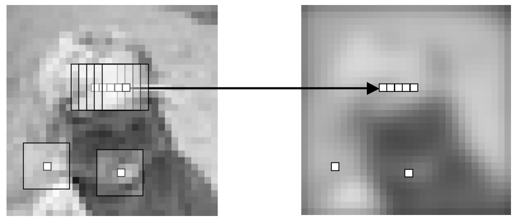

Latency
Core
Core
Core
1 double/cycle *per core*
1.3 double/cycle *per core*
1.9 double/cycle *per core*
0.4-1.0 double/cycle
L1
L2
L3
DRAM

D. Molka, et. al., *Memory Performance and Cache Coherency Effects on an Intel Nehalem Multiprocessor System*, PACT 2009.

These numbers are important. They say a lot about the performance you can expect.

## 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

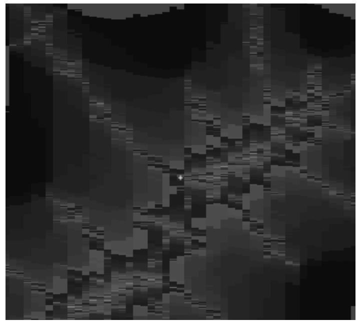


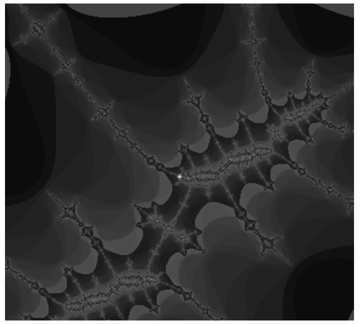
*Performance *may* fall off a cliff if not exactly the same.

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.

## OpenCL Compute Model

- 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

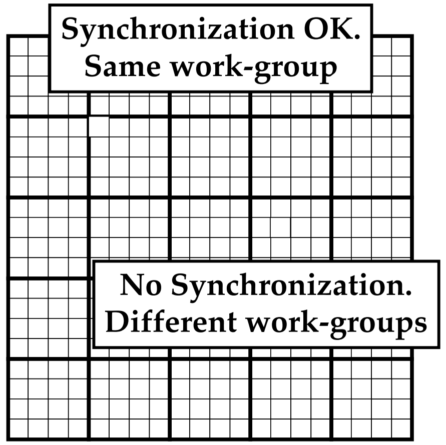
| | | |
|---|---|---|
| 1k audio: | 1024 | 1024 work-items |
| HD video: | 1920x1080 | 2M work-items |
| 3D MRI: | 256x256x256 | 16M work-items |
| HD per line: | 1080 | 1080 work-items |
| HD per 8x8 block: | 240x135 | 32k work-items |

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 into **local work-groups**
- Each work-group is logically executed together on one compute unit
- Synchronization is **only** allowed between **work-items in the same work-group**
  This is important.

## Local Dimensions and Synchronization

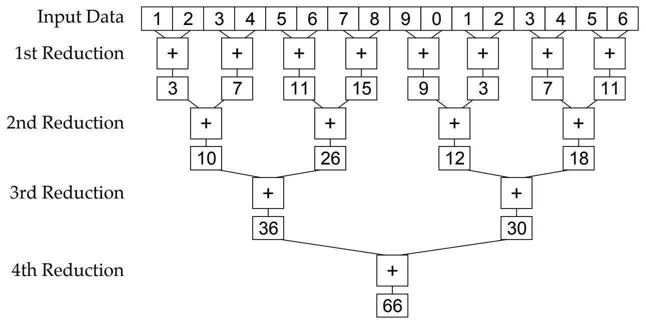


| | |
|---|---|
| Global domain: | 20x20 |
| Work-group size: | 4x4 |

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

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

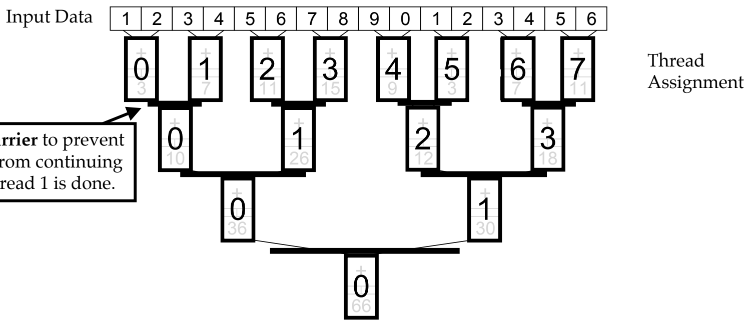
## Synchronization Example: Reduction



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

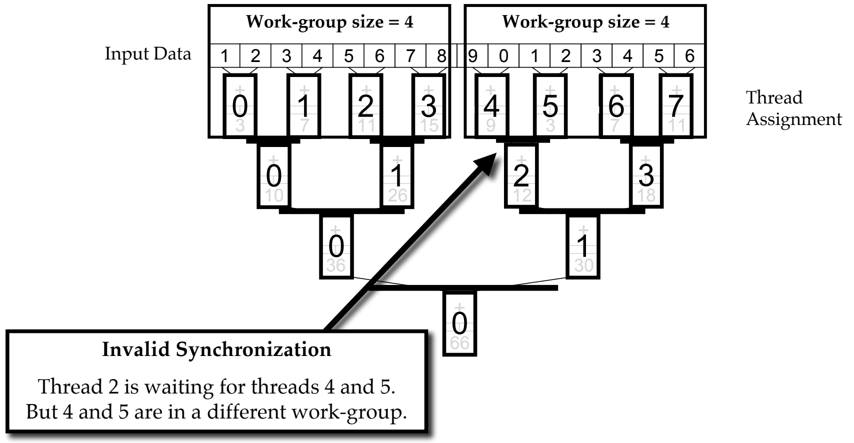
## Synchronization Example: Reduction

Need a **barrier** to prevent thread 0 from continuing before thread 1 is done.



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



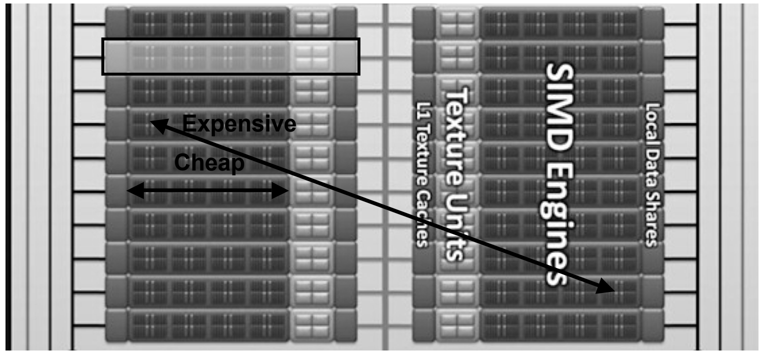
**Invalid Synchronization**

Thread 2 is waiting for threads 4 and 5.
But 4 and 5 are in a different work-group.

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?

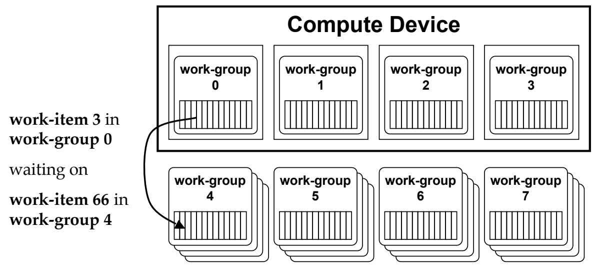
- Scales well in hardware
  - Only work-items within a work-group need to communicate
  - GPUs run 32-128 work-groups in parallel



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.

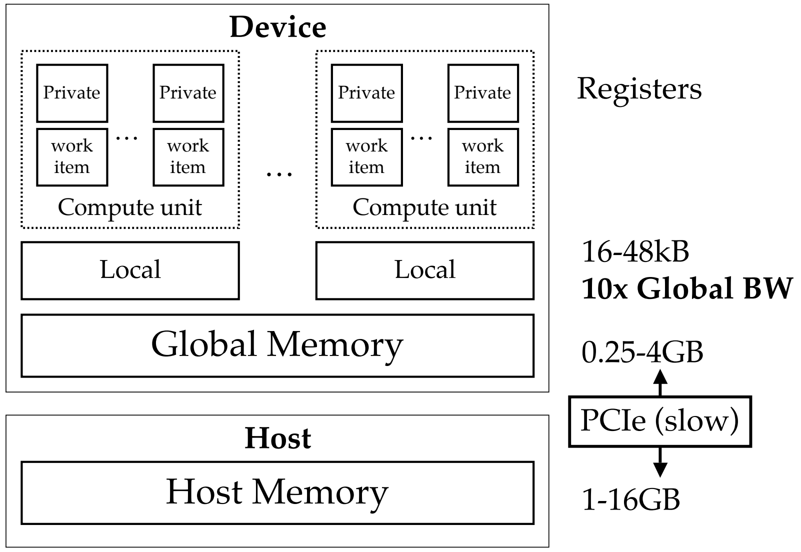
(Nvidia is starting to ship hardware that support this more flexibly.)

## Choosing Local and Global Dimensions

- **Global dimensions**
  - Natural division for the problem
  - Too few: no latency hiding (GPU; SMT CPU)
  - Too many: too much overhead (particularly 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.

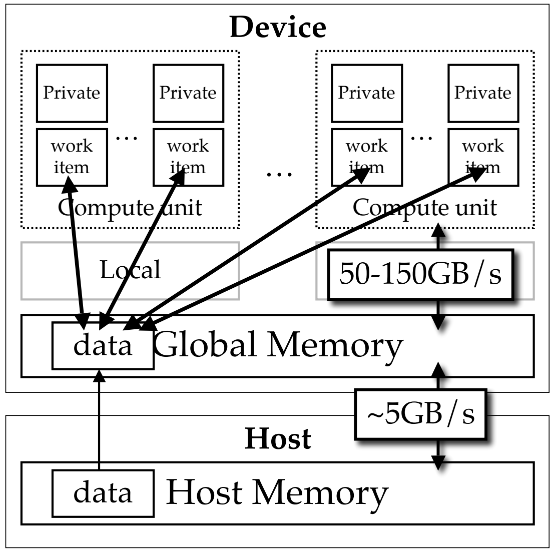
## OpenCL Memory Model



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.

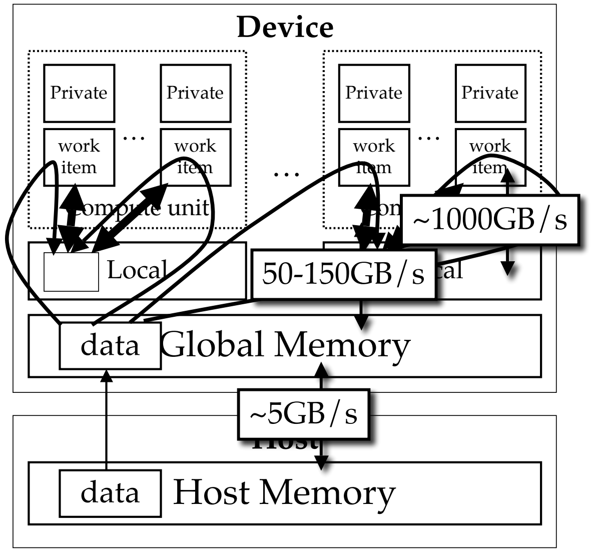
Note how this memory model looks a lot like a GPU…

## 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

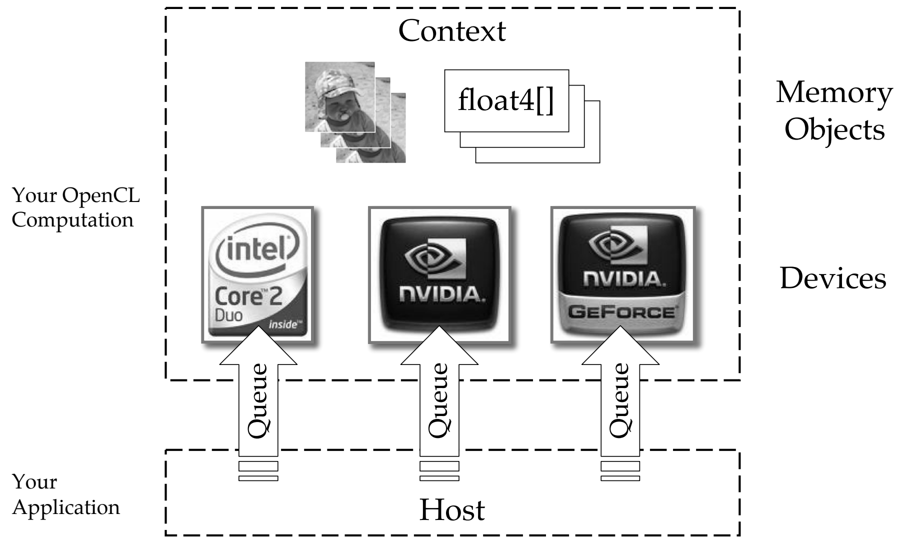


Using local memory is much more complicated. The user must not only allocate the data, but have the kernel code running on each work-item copy the appropriate data from the global memory into the local memory before it can be used. This is quite complicated (as are all software-managed memories) but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

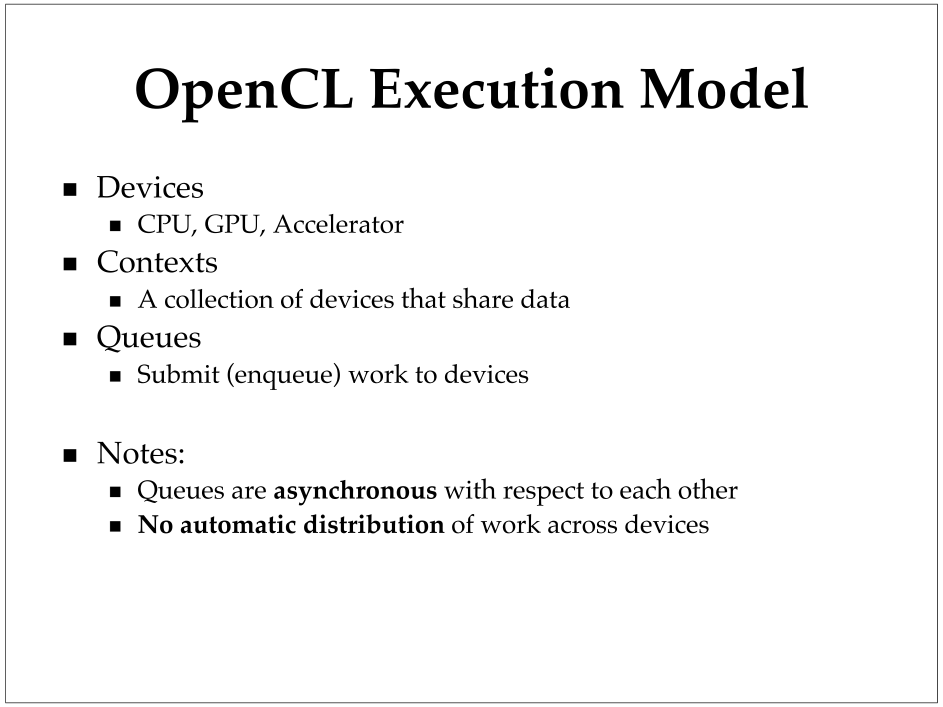
## Moving Data

- 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?)

## OpenCL Execution Model



A compute context indicates which devices share which memory objects. This is necessary so OpenCL can know how to move the data around to the appropriate device when you are ready to use it. Your application interacts with the compute devices by submitting work to command queues for each device.

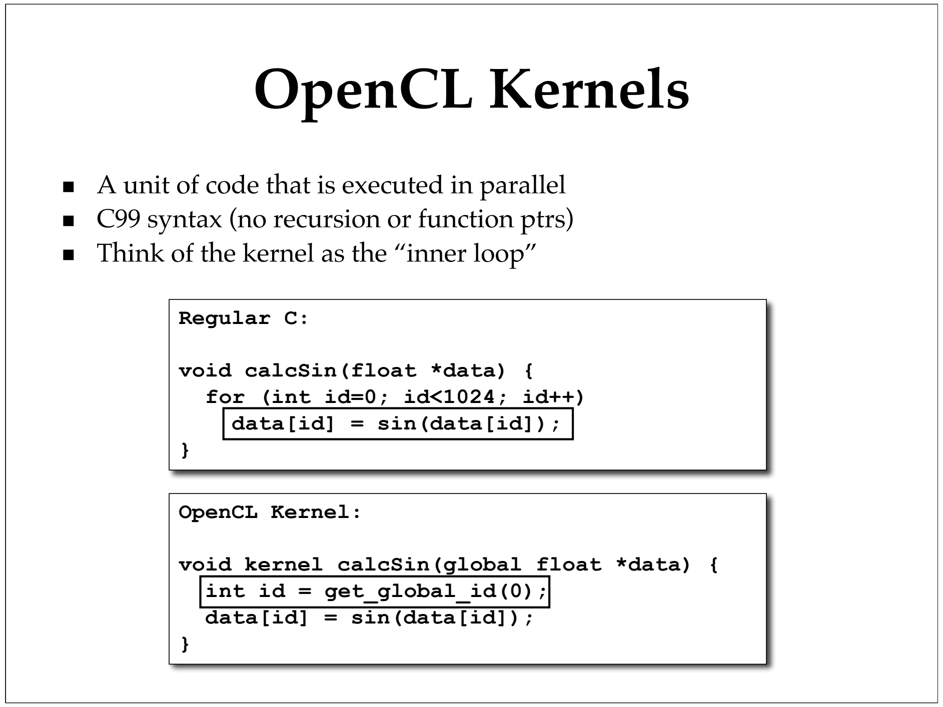


## OpenCL Execution Model

- Devices
  - CPU, GPU, Accelerator
- Contexts
  - A collection of devices that share data
- Queues
  - Submit (enqueue) work to devices

- Notes:
  - Queues are **asynchronous** with respect to each other
  - **No automatic distribution** of work across devices

This last point is annoying. You have to manually split up and send your work to multiple GPUs or across the CPU and GPU. OpenCL 1.1 provides a feature to help with this that lets you provide an offset to the global dimensions for each execution.



## 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 C

- Vectors
- Rounding modes and conversions
- Intrinsic functions

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

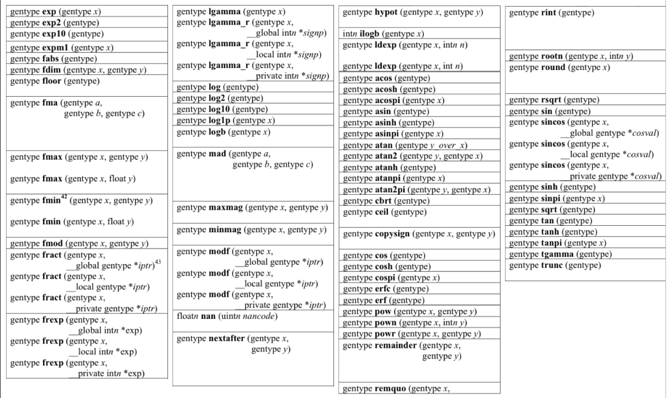
## OpenCL C - Vectors

- 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(`**…**`)`**

## 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*) |

## 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 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?

## An OpenCL Program

1. Get the devices
2. Create contexts and queues
3. Create programs and kernels
4. Create memory objects
5. **Enqueue writes** to initialize memory objects
6. **Enqueue kernel** executions
7. **Wait** for them to finish
8. **Enqueue reads** to get back data
9. Repeat 5-8

## OpenCL Hello World

- Get the device
- Create a context
- Create a command queue

```
              clGetDeviceIDs(NULL, CL_DEVICE_TYPE_DEFAULT,
                             1, &device, NULL);

context =     clCreateContext(NULL, 1, &device,
                             NULL, NULL, NULL);

queue =       clCreateCommandQueue(context, device,
                      (cl_command_queue_properties)0, NULL);
```

This example has no error checking. This is very foolish.

Changing CL_DEVICE_TYPE_DEFAULT to CL_DEVICE_TYPE_GPU will get a GPU device if there is one. Always check error returns, and use a context callback function to get more detailed information.

## OpenCL Hello World

- Create a program with the source
- Build the program and create a kernel

```
char *source = {
"kernel calcSin(global float *data) {   \n"
"  int id = get_global_id(0);           \n"
"  data[id] = sin(data[id]);            \n"
"}                                      \n"};

program =     clCreateProgramWithSource(context, 1,
                    (const char**)&source, NULL, NULL);

              clBuildProgram(program, 0,
                    NULL, NULL, NULL, NULL);

kernel =      clCreateKernel(program, "calcSin", NULL);
```

The source for the kernel here is just a string. You can read this from a file, generate it via sprintf, or include it as a constant in your code. If you want more security, you can build a binary and store that, but the interface for doing so is very primitive today. Unless you are executing on exactly the same hardware that generated the binary there is no guarantee it will work. I would advise avoiding binary kernels for the immediate future.

## OpenCL Hello World

- Create and initialize the input

```
buffer =      clCreateBuffer(context, CL_MEM_COPY_HOST_PTR,
                           sizeof(cl_float)*10240,
                           data, NULL);
```

Note that the buffer specifies the **context** so OpenCL knows which devices may share it.

By specifying CL_MEM_COPY_HOST_PTR the data from the pointer “data” will be copied into the memory buffer when it is created.

## OpenCL Hello World

- Set the kernel arguments
- Enqueue 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);
```

Local dimensions are NULL. OpenCL will pick reasonable ones automatically. (Or so you hope…)

Note that you've just enqueued the kernel. It may or may not execute depending on what else is going on and the whims of the runtime.

## OpenCL Hello World

- Read back the results

```
      clEnqueueReadBuffer(queue, buffer,
                          CL_TRUE,
                          0, sizeof(cl_float)*LENGTH,
                          data, 0, NULL, NULL);
```

The **CL_TRUE** argument specifies that the call should **block** until the read is complete. Otherwise you would have to explicitly wait for it to finish.

Specifying CL_TRUE for blocking here is the same as executing clEnqueueReadBuffer non-blocking and then calling clWait().

## OpenCL Hello World

The Demo

Optimization: slide 50

## 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***
  - **CL_DEVICE_GLOBAL_MEM_SIZE***
    Total global memory available on the device
  - **CL_DEVICE_IMAGE_SUPPORT**
    Some GPUs don't support images today (shocking, I know…)
  - **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) and some just don't have support yet (AMD GPUs as of Fall 2009).

## Events

- Subtle point made earlier:

  Queues for **different devices** are **asynchronous with respect to each other**
- Implication:
  - You must **explicitly synchronize** operations **between devices**

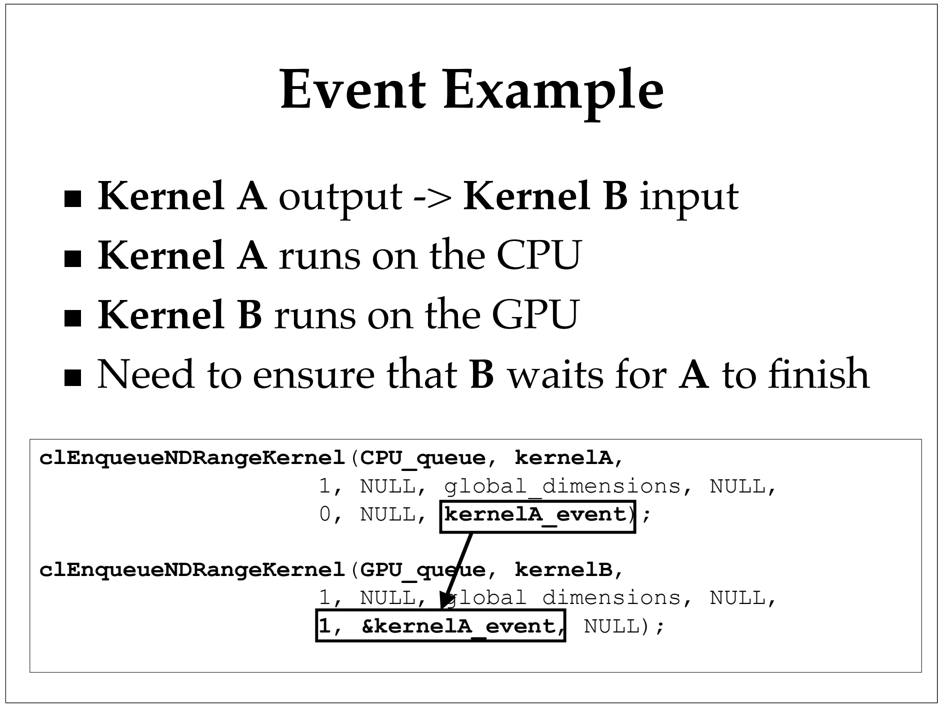
(Also applies to out-of-order queues)

## Events

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

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

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

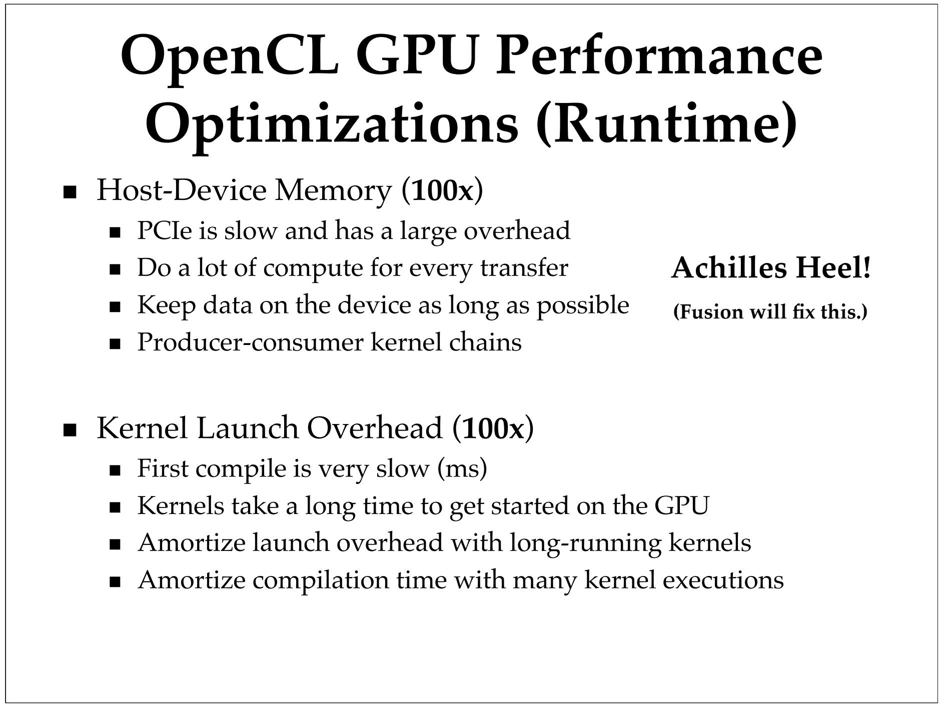


## Event Example

- **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 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

**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 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*

## Why Use OpenCL?

- Industry standard
  - Here to stay
  - Vendor neutral
- Optimized for GPUs
  - 10-100x the performance of CPUs
  - 10-100x the efficiency of CPUs
- Driving future hardware
  - CPUs and GPUs are converging
  - Running fast on OpenCL now is a good bet for the future

## Why Not Use OpenCL?

- Immature
  - ~~Limited heterogeneous support (S)~~
  - Limited profiling tools
  - Limited debugging tools (S+H)
  - Reduced performance (**S**)
    (compared to previous solutions, e.g. CUDA)

GPU: Nvidia & AMD
CPU: Intel & AMD

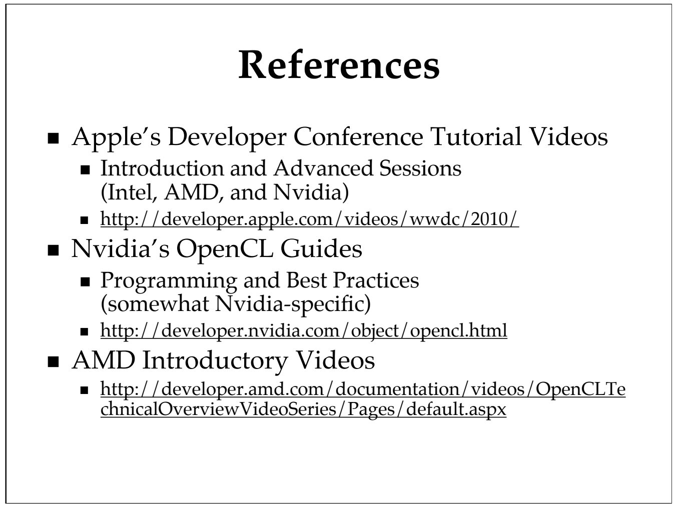
Ironically Nvidia is the only player here.

Nvidia doesn't really like OpenCL

(**S**) - software development: changing rapidly
(**H**) - hardware development needed (>1 year)

## Cynicism Aside…

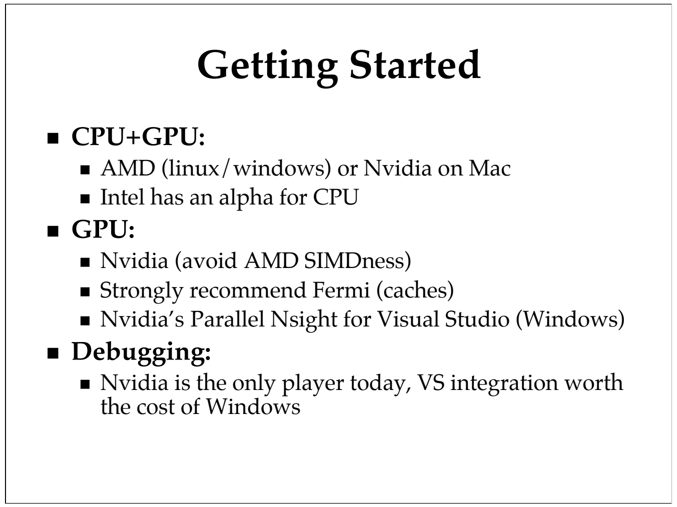
- OpenCL is the best step yet towards platform-independent massively-parallel computing
- You can see order-of-magnitude speedups on real code on shipping hardware
- Just don't expect it to solve your problems automagically, and be prepared for a few bumps on the way



## 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 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.



## Getting Started

- **CPU+GPU:**
  - AMD (linux/windows) or Nvidia on Mac
  - Intel has an alpha for CPU
- **GPU:**
  - Nvidia (avoid AMD SIMDness)
  - Strongly recommend Fermi (caches)
  - Nvidia's Parallel Nsight for Visual Studio (Windows)
- **Debugging:**
  - Nvidia is the only player today, VS integration worth the cost of Windows

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.

## Good Match for a GPU?

- 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.

## Questions?