Introduction to OpenCL

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

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

Disclaimer
- I worked for Apple developing OpenCL
- I’m biased
  (But not in the way you might think…)

Open Standard - 2008
- Good industry support
- Driving hardware requirements

Huge Industry Support - 2010

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-precision¹

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

¹This is changing, the others are not.

Computational Intensity

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

- Loop body: Low-intensity:
  \[ A[i] = B[i] + C[i] \]
  \[ A[i] = B[i] + C[i] \times D[i] \]
  \[ A[i]++ \]

- Loop body: High-intensity:
  \[ \text{Temp} := A[i] \times A[i] \]
  \[ A[i] = \exp(\text{temp}) \times \acos(\text{temp}) \]

Data-Parallelism

- Same independent operations on lots of data²
- Examples:
  - Modify every pixel in an image with the same filter
  - Update every point in a grid using the same formula

²Performance may fall off a cliff if not exactly the same.

Single Precision

32 bits should be enough for anything…

(Expect double precision everywhere in ~1 year.)

Q: Will double precision be slower? Why?

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 work-items
- HD video: 1920x1080 work-items
- 3D MRI: 256x256x256 work-items
- HD per line: 1080 work-items
- HD per 8x8 block: 240x135 work-items

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

Local Dimensions and Synchronization

- Each work-group is logically executed together on one compute unit (Nvidia Streaming Multiprocessor).
- Synchronization is only allowed between work-items in the same work-group.

Synchronization Example: Reduction

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

What About Spinlocks in OpenCL?
```
while (!lock[n]) {}
```

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

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)

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: 13x4-52
  - If the hardware has 16 cores per compute unit:
    - We can share 100% of the cores for the first 160 work-items
    - But we then have 4 left over, so we waste 12 for the last threads
    - Utilization: 52/(16*4)=81%
Compute Kernels

How you do your computations.

OpenCL Kernels

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

Regular C:

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

OpenCL Kernel:

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

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.)

OpenCL C - Intrinsics

- Explicitly trade-off precision and performance
  - `native` - fastest; no accuracy guarantee
  - `half` - faster, less accuracy

Utility Functions

- Information about each work-item
  - `get_global_id(dim)` current work-item’s ID in a particular dimension
  - `get_global_size(dim)` number of global work-items in a particular dimension
  - `get_local_size(dim)` number of global work-items in a particular dimension
  - `get_local_size0, get_local_id0, get_num_groups0, get_group_id()` information about the local dimensions
- Determine what work each work-item does.
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

OpenCL Platforms

Efficient data sharing within a platform. Slow between platforms.

OpenCL Devices

Device Global Memory
Host Memory

OpenCL Contexts

Contexts define which devices can share data objects. All devices in a context must be in the same platform.

The CPU can be shared as both an OpenCL Device and the host processor.
OpenCL Command Queues

Data Movement

Getting your data to the device... and back.

OpenCL Memory Model

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?
  
  \[
  \text{float} \ a = \text{big_array}[n];
  \]

- 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 bytes = 32 bits

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

\[
\text{float} \ a = \text{big_array}[n] \Rightarrow \text{loads 32 floats}!!
\]

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
- No cache (except new cards)
- Other threads need to use them at the same time
  \(\Rightarrow\) Memory Access Coalescing

On a GPU:

- No cache (except new cards)
- Other threads need to use them at the same time
  \(\Rightarrow\) Memory Access Coalescing

GPU Memory Coalescing

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

  Example:
  
  - Thread 0: `float a = big_array[0]` loads `big_array[0]`
  - All threads load same data and use 4 bytes each
    \(\Rightarrow\) share one access, use 100% of data

  But...
  
  - Thread 0: `float a = big_array[0]` loads `big_array[0]`
  - Thread 1: `float a = big_array[1024]` loads `big_array[1024]`
  - Thread 2: `float a = big_array[2048]` loads `big_array[2048]`
  - All threads load different data and use only 4 bytes each
    \(\Rightarrow\) waste 96% of loaded data

Questions so far?

platforms/devices/contexts/queues
asynchronous execution/global/local memory
data movement/coalescing...
### An OpenCL Program

1. **Setup**
   1. Get the device (and platform)
   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
   4. Enqueue write 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

---

### OpenCL Hello World: Calculate \( \sin(x) \) in parallel.

#### Setup:
1. Call `clGetDeviceIDs` and specify the number and type of device(s) you want.
2. Create a context and specify the device(s) you want to have in the context.
3. Create a command queue for each device.

#### Compilation:
1. Call `clCreateProgramWithSource` and specify the device(s) you want.
2. Compile the program (and platform), specify the number of device(s) you want.
3. Create a command queue for each device.

#### Create Memory Objects:
1. Enqueue reads/Writes to initialize memory objects
2. Set the kernel arguments
3. Enqueue kernel executions
4. Enqueue reads to get back data
5. Wait for your commands to finish

---

```c
# OpenCL Hello World: Calculate \( \sin(x) \) in parallel.

OpenCL Hello World: 

Your first OpenCL program...
2. Compile: Create a program from the text source.

2. Compile: Compile (build) the program. This is slow…but it's usually lazy, so you may not notice it until later.

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

2. Compile: Create a memory object (buffer or image) and specify the size.

4. Write to the memory object by enqueuing a write to the queue. (Remember: commands are asynchronous!)
6. Enqueue the asynchronous kernel execution, specifying the global dimensions. (And optionally the local dimensions.)

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

8. Wait for everything to finish asynchronously.

Questions so far?

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

More OpenCL

- Querying Devices
- Images
- Events

David Black-Schaffer
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
    - (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 (older AMD GPUs, Cell)
  - 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.*

Images

- 2D and 3D Native Image Types
  - R, RC, 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

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)

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
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)

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

Achilles Heel!
- Producer-consumer kernel chains (keep the data on the device)

OpenCL GPU Performance Optimizations (Kernel)

- Memory Accesses (~10x)
  - Coloring matters for slicing
  - Addresses should be sequential across threads
  - Newer hardware is more forgiving

- Local Memory (~10x)
  - Much larger bandwidth
  - Manually manage
  - Look out for bank conflicts

- Divergent execution (up to 8x)
  - Vectors (2x 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 assembly...
- 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

OpenCL? (Honestly)

Low-level language for high-performance heterogeneous data-parallel 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/novations require different optimizations
5. Hardware and software only support data-parallel
   - There is task-parallel support, but not on today’s GPUs

Is Your Application a 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.

Getting Started

CPU+GPU:
- AMD (linux/windows) or Nvidia on Mac
- Intel has a CPU-only implementation with interesting auto-SIMDization, but it’s not as mature

GPU:
- Nvidia (avoid AMD SIMDness)
- Strongly recommend Fermi (caches)
- Nvidia’s Parallel Nsight for Visual Studio (Windows)

Debugging:
- Nvidia is the only player today
References

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

Questions?

OpenCL C - More Intrinsics

- Many more…
  - Integer (mad24, abs, clamp, clz, …)
  - Common (clamp, degrees, max, step, sign…) (somewhat Nvidia-specific)
  - Geometric (cross, dot, distance, length, normalize)
  - Relational (isequal, isless, any, isnan, select…)
  - Vector load/store (vload_type, vstore_type, …)
  - Synchronization (barrier, mem_fence, …)
  - Asynchronous 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
- Advise: use the natural vector size
  - Don’t try to make everything 32-wide
  - Graphics: RGBA, use 4-wide
  - Positions: 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, pos);
- 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: intfloor(1.5f);
- Examples:
  - int3 = convert_int_rte(float f);
  - uchar8 c8 = convert_uchar_rtz(double8 d);
  - Supports: rte, rtz, rtp, rln (default rtz)
  - Supports saturation: convert_int_sat_rtz(…)
  - Examples:
    - float pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
    - pos = (float4)(1.0f, 2.0f, 3.0f, 4.0f);
    - float16 big.s01ef = pos;
    - float16 big.s2301 = (float4)(pos, pos);