GPU Architecture and Programming with OpenCL

David Black-Schaffer
david.black-schaffer@it.uu.se
Room 1221

Today’s Topic

- GPU architecture
  - What and why
  - The good
  - The bad
- Compute Models for GPUs
  - Data-parallel
- OpenCL
  - Programming model
  - Memory model
  - Hello World
- Ideas for Ph.D. student projects

GPU Architecture: Why?

- Answer: Triangles
- Real Answer: Games
- Really Real Answer: Money

GPUs: Architectures for Drawing Triangles Fast

- Basic processing:
  - Project triangles into 2D
  - Find the pixels for each triangle
  - Determine color for each pixel
- Where is most of the work?
  - 10k triangles (30k vertices)
  - Project, clip, calculate lighting
  - 1920x1200 = 2.3M pixels
  - 8x oversampling = 18.4M pixels
  - 7 texture lookups
  - 43 shader ops
  - @ 60fps
    - Compute: 47.5 GOPs
    - Memory: 123GB/s
    - Intel Nehalem: 106 GFLOPs, 32GB/s
Example Shader: Water

```
float4 WaterShader(float2 TexCoord : UVS) : SV_Target

// Preprocess the lighting. Uses DirectX's lighting model.
float4 Lighting = LightingModel(TexCoord, 0.0f, 1.0f);

// Preprocess the water reflection. Uses DirectX's lighting model.
float4 Reflection = ReflectModel(TexCoord, 0.0f, 1.0f);

// Water shading
float4 WaterShading = WaterModel(TexCoord, 0.0f, 1.0f);

// Combine lighting and water shading
float4 CombinedShading = Lighting + Reflection + WaterShading;

// Add some additional effects
float4 FinalShading = FinalModel(TexCoord, 0.0f, 1.0f);

// Output the final shading
SV_Target = FinalShading;
```

GPGPU: General Purpose GPUs

- **Question:** Can we use GPUs for non-graphics tasks?
- **Answer:** Yes!
  - They’re incredibly fast and awesome
- **Answer:** Maybe
  - They’re fast, but hard to program
- **Answer:** Not really
  - My algorithm runs slower on the GPU than on the CPU
- **Answer:** No
  - I need more precision/memory/synchronization/other

Why Should You Care?

**Intel Nehalem 4-core**
- 130W, 263mm²
- 32 GB/s BW, 106 GFLOPs (SP)
- Big caches (8MB)
- Out-of-order
- 0.8 GFLOPs/W

**AMD Radeon 5870**
- 188W, 334mm²
- 154 GB/s BW, 2720 GFLOPs (SP)
- Small caches (<1MB)
- Hardware thread scheduling
- 14.5 GFLOPs/W

1) Process pixels in parallel

- **Data-parallel:**
  - 2.3M pixels per frame
    - => lots of work
  - All pixels are independent
    - => no synchronization
  - Lots of spatial locality
    - => regular memory access
- **Great speedups**
  - Limited only by the amount of hardware

GPU Design
**GPU Design**

2) Focus on throughput, not latency
- Each pixel can take a long time…
  …as long as we process many at the same time.
- Great scalability
  - Lots of simple parallel processors
  - Low clock speed

<table>
<thead>
<tr>
<th>Latency-optimized (fast, serial)</th>
<th>Throughput-optimized (slow, parallel)</th>
</tr>
</thead>
</table>

**CPU vs. GPU Philosophy: Performance**

4 Massive CPU Cores: Big caches, branch predictors, out-of-order, multiple-issue, speculative execution, double-precision…
About 2 IPC per core, 8 IPC total @3GHz

8* Wimpy GPU Cores: No caches, in-order, single-issue, single-precision…
About 1 IPC per core, 64 IPC total @1.5GHz

**Example GPUs**

Nvidia G80
AMD 5870

**CPU Memory Philosophy**

Instructions:
- \( g = f + 1 \)
- \( f = \text{ld}(e) \)
- \( d = d + 1 \)
- \( e = \text{ld}(d) \)
- \( c = b + a \)
- \( b = a + 1 \)
CPU Memory Philosophy

Instructions

\[
\begin{align*}
g &= f + 1 \\
+ &= ld(st) \\
d &= d + 1 \\
c &= b + a \\
b &= a + 1
\end{align*}
\]

Cycle 0

Memory access will take \(~100\) cycles...

CPU Memory Philosophy

Instructions

\[
\begin{align*}
g &= f + 1 \\
+ &= ld(st) \\
d &= d + 1 \\
c &= b + a \\
b &= a + 1
\end{align*}
\]

Cycle 0

L1 Cache

Hit!
CPU Memory Philosophy

Cycle 3

++

ld/st

L1 Cache

L2 Cache

d= d+1
e= ld d

Instructions/.null

L1 Cache

L2 Cache

Miss!

g= t+1

Now we stall the processor for 20 cycles waiting on the L2...

Cycle 23

++

ld/st

L1 Cache

L2 Cache

d= d+1
e= ld d

Instructions/.null

L1 Cache

L2 Cache

Hit!

g= t+1

d= d+1
e= ld d

b= a+1

Instructions/.null

L1 Cache

L2 Cache

Miss!

g= t+1

b= a+1

Instructions/.null

L1 Cache

L2 Cache

Hit!

Now we stall the processor for 20 cycles waiting on the L2...

Cycle 24

++

ld/st

L1 Cache

L2 Cache

d= d+1
e= ld d

Instructions/.null

L1 Cache

L2 Cache

d= d+1
e= ld d

b= a+1

Instructions/.null

L1 Cache

L2 Cache

Miss!

g= t+1

Now we stall the processor for 20 cycles waiting on the L2...
CPU Memory Philosophy

Instructions

- Big caches + instruction window + out-of-order + multiple-issue
- Approach
  - Reduce memory latencies with caches
  - Hide memory latencies with other instructions
- As long as you hit in the cache you get good performance

GPU Memory Philosophy

Instructions

- Big caches + instruction window + out-of-order + multiple-issue
- Approach
  - Reduce memory latencies with caches
  - Hide memory latencies with other instructions
- As long as you hit in the cache you get good performance
GPU Memory Philosophy

Instructions

Cycle 0

```
+ b=a+1
ld/st
```

Cycle 1

```
+ c=b+a
ld/st
```

Cycle 2

```
Solution: Give Up
```

No cache ~ 100+ cycles

Cycle 2

```
c=b+a
b=a+1
```

Memory
GPU Memory Philosophy

Instructions

First load ready!

Memory

Cycle 6

Cycle 102

Cycle 103

Cycle 103
GPU Memory Philosophy

- Thousands of hardware threads
- 1 cycle context switching
- Hardware thread scheduling
- As long as there is enough work in other threads to cover latency you get high throughput.

Notes:
- GPUs have caches for textures
- GPUs will soon have data caches

GPU Instruction Bandwidth

- GPU compute units fetch 1 instruction per cycle...
  ...and share it with 8 processor cores.
- What if they don’t all want the same instruction? (divergent execution)

Divergent Execution

Divergent execution can dramatically hurt performance. Avoid it on GPUs today.
Divergent Execution for Real

Per-pixel Mandelbrot calculation:

```c
while (x*x + y*y <= (4.0f) && iteration < max_iterations) {
    float xtemp = x*x - y*y + x0;
    y = 2*y*x + y0;
    x = xtemp;
    iteration++;
}
```

Color determined by iteration count…
...each color took a different number of iterations.

Every different color is a divergent execution of a work-item.

Instruction Divergence

- Some architectures are worse…
  - AMD’s GPUs are 4-way SIMD
    - If you don’t process 4-wide vectors you lose.
  - Intel’s Larabee is (was?) 16-way SIMD
    - Theoretically the compiler can handle this.
- Some architectures are getting better…
  - Nvidia Fermi can fetch 2 instructions per cycle
  - But it has twice as many cores
- In general:
  - Data-parallel will always be fastest
  - Penalty for control-flow varies from none to huge

CPU and GPU Architecture

- GPUs are throughput-optimized
  - Each thread may take a long time, but thousands of threads
- CPUs are latency-optimized
  - Each thread runs as fast as possible, but only a few threads

- GPUs have hundreds of wimpy cores
- CPUs have a few massive cores

- GPUs excel at regular math-intensive work
  - Lots of ALUs for math, little hardware for control
- CPUs excel at irregular control-intensive work
  - Lots of hardware for control, few ALUs

OpenCL
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)
- Based on C99
- Portable across devices
- Vector intrinsics and math libraries
- Guaranteed precision for operations
- Open standard

What is OpenCL Good For?

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

Note: I am going to focus on the GPU

*This 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:
  \[ A[i] = B[i] + C[i] \]

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

\[ A[2++] = 1:2 \]
\[ \text{Temp} = A[i] \times A[i] \]
\[ A[2] = \exp(\text{temp}) \times \text{erf}(\text{temp}) \]
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.

Single Precision

- 32 bits should be enough for anything…

This is changing. Expect double precision everywhere in 2 years.

OpenCL Compute Model

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

**Examples**
- 1k audio: 1024
- HD video: 1920x1080
- 3D MRI: 256x256x256
- HD per line: 1080
- HD per 8x8 block: 240x135

- 1024 work-items
- 2M work-items
- 16M work-items
- 1080 work-items
- 32k work-items

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 OK.
Same work-group

No Synchronization.
Different work-groups

Synchronization Example:
Reduction

Input Data
1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6

Thread Assignment
0 1 2 3 4 5 6 7

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

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

Choosing Local and Global Dimensions

- **Global dimensions**
  - Natural division for the problem
  - Too few: no latency hiding
  - Too many: (too little work each) too much overhead
  - In general:
    - GPU: >2000
    - CPU: ~2*#CPU cores

- **Local dimensions**
  - May be determined by the algorithm
  - Optimize for best processor utilization (hardware-specific)

OpenCL Memory Model

- **Device**
  - Local
  - Global Memory
  - Host Memory

- **Registers**
  - 16-32kB
  - 0.25-4GB
  - 1-16GB

- **Global Memory**
  - 10x Global BW
  - 50-200GB/s

- **Host Memory**
  - ~5GB/s

- **PCIe (slow)**
  - ~0.25-4GB
### OpenCL Memory Model

- **Device**
  - Private work-item
  - Global Memory
  - Local
  - Host Memory

- **Host**
  - ~1000GB/s
  - ~5GB/s

### 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

- **Context**
  - float4[]

- **Memory Objects**

- **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
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:
```c
void calcSin(float *data) {
    for (int id=0; id<1023; 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]);
}
```

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

```c
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.

OpenCL Hello World

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

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

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

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

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

- Create and initialize the input

```c
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.

- Set the kernel arguments
- Enqueue the kernel

```c
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...)

- Read back the results

```c
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.

The Demo
More OpenCL

- Querying Devices
- Images
- Events

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
  - `CLDEVICE_IMAGE_SUPPORT`
    Some GPUs don't support images today
  - `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, RG, RGB, RGBA, INTENSITY, LUMINANCE
  - 8/16/32 bit signed/unsigned, float
  - Linear interpolation, edge wrapping and clamping
- Why?
  - Hardware accelerated access on GPUs
  - Want to enable this fast path
  - GPUs cache texture lookups today
- But...
  - Slow on the CPU (which is why Larabee does 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)
Events

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

```c
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

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

Performance Optimizations

- 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
- Memory Accesses (~16x)
  - Ordering matters for coalescing
  - Addresses should be sequential across threads
- Local Memory (~16x)
  - Much larger bandwidth
  - Must manually manage
- 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)

Debugging (Or Not)

- Very little debugging support on GPUs
- Start on the CPU
  - At least you can use printf()...
- 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
GPU Projects

Approaches

- Data-parallel
  - Simplest mapping
  - Just need right compute-to-memory ratio
- Thread-parallel
  - Generally a bad mapping
  - Threads that don’t do the same thing pay a big penalty
  - Only cheap local synchronization
- Reduction
  - Require synchronization between stages
  - Tricky across work-groups
- Scan-based
  - Handles variable length data
  - Brute-force, but fully data-parallel

Scan Algorithms

Simple Scan

- Produces all sums of the elements
- Also works with min, max, or, etc.
- Log scaling with the number of elements
- Data-parallel
- Can do conditional operations too
  - Pass in a second array of flags
  - Conditionally propagate data based on flags
  - Allows for data-parallel execution of variable-length operations (this is awesome)

Images from http://en.wikipedia.org/wiki/Prefix_sum

Project Ideas

- JPEG zero-run encoding performance for varying sizes
  - 64 quantized coefficients; need to count zeros and then Huffman encode
  - Parallel scan vs. serial for RLE
- Variable length processing
  - Serial scan has nearly 2x the data bandwidth
  - But it’s fully parallel
  - At what level does it make sense?
    - Local memory
    - Global memory