Lecture 6: GPU Programming

G63.2011.002/G22.2945.001 · October 12, 2010
Admin bits

- Start thinking about final projects
  - Find Teams
- About HW4
- Legislative Day (Dec 14)
- HW3 posted
Outline

GPU Architecture (recap)

Programming GPUs
Outline

GPU Architecture (recap)

Programming GPUs
“CPU-style” Cores

Credit: Kayvon Fatahalian (Stanford)
Slimming down

Idea #1:

Remove components that help a single instruction stream run fast

Credit: Kayvon Fatahalian (Stanford)
More Space: Double the Number of Cores

Credit: Kayvon Fatahalian (Stanford)
... again

Credit: Kayvon Fatahalian (Stanford)
... and again

16 cores = 16 simultaneous instruction streams

→ 16 independent instruction streams

Reality: instruction streams not actually very different/independent

Credit: Kayvon Fatahalian (Stanford)
Idea #2
Amortize cost/complexity of managing an instruction stream across many ALUs
→ SIMD

Credit: Kayvon Fatahalian (Stanford)
Idea #2

Amortize cost/complexity of managing an instruction stream across many ALUs

→ SIMD

Credit: Kayvon Fatahalian (Stanford)
Idea #2
Amortize cost/complexity of managing an instruction stream across many ALUs
→ SIMD
Saving Yet More Space

Idea #2

Amortize cost/complexity of managing an instruction stream across many ALUs

→ SIMD

Credit: Kayvon Fatahalian (Stanford)
Gratuitous Amounts of Parallelism!

128 fragments in parallel = 16 simultaneous instruction streams
16 cores = 128 ALUs

Credit: Kayvon Fatahalian (Stanford)
Gratuitous Amounts of Parallelism!

Example:

128 instruction streams in parallel
16 independent groups of 8 synchronized streams
Example:

128 instruction streams in parallel
16 independent groups of 8 synchronized streams

Great if everybody in a group does the same thing.

But what if not?

What leads to divergent instruction streams?
But what about branches?

```cpp
if (x > 0) {
  y = pow(x, exp);
  y *= Ks;
  refl = y + Ka;
} else {
  x = 0;
  refl = Ka;
}
```

`<unconditional shader code>`

`<resume unconditional shader code>`

Credit: Kayvon Fatahalian (Stanford)
But what about branches?

```
if (x > 0) {
    y = pow(x, exp);
    y *= Ks;
    refl = y + Ka;
} else {
    x = 0;
    refl = Ka;
}
```
But what about branches?

Not all ALUs do useful work!
Worst case: 1/8 performance

Credit: Kayvon Fatahalian (Stanford)
But what about branches?

```
if (x > 0) {
  y = pow(x, exp);
  y *= Ks;
  refl = y + Ka;
} else {
  x = 0;
  refl = Ka;
}
```

Credit: Kayvon Fatahalian (Stanford)
Problem
Memory still has very high latency... but we’ve removed most of the hardware that helps us deal with that.

We’ve removed
- caches
- branch prediction
- out-of-order execution

So what now?
Remaining Problem: Slow Memory

Problem
Memory still has very high latency... but we’ve removed most of the hardware that helps us deal with that.

We’ve removed
- caches
- branch prediction
- out-of-order execution

So what now?

Idea #3
Even more parallelism
+ Some extra memory
= A solution!
Remaining Problem: Slow Memory

Problem Memory still has very high latency. . .

. . . but we've removed most of the hardware that helps us deal with that. We've removed

- caches
- branch prediction
- out-of-order execution

So what now?

Idea #3

Even more parallelism

+ Some extra memory

= A solution!

SIGGRAPH 2009: Beyond Programmable Shading: http://s09.idav.ucdavis.edu/
Remaining Problem: Slow Memory

Memory still has very high latency... but we've removed most of the hardware that helps us deal with that. We've removed:

- caches
- branch prediction
- out-of-order execution

So what now?

Idea #3

Even more parallelism

+ Some extra memory

= A solution!
Hiding Memory Latency

Credit: Kayvon Fatahalian (Stanford)
Hiding Memory Latency

Time (clocks)

Frag 1 … 8
Frag 9… 16
Frag 17 … 24
Frag 25 … 32

Credit: Kayvon Fatahalian (Stanford)
Hiding Memory Latency

Credit: Kayvon Fatahalian (Stanford)
Hiding Memory Latency

Time (clocks)

Frag 1 ... 8
Frag 9 ... 16
Frag 17 ... 24
Frag 25 ... 32

Runnable

Stall

Credit: Kayvon Fatahalian (Stanford)
Hiding Memory Latency

Time (clocks)

Frag 1 … 8
Frag 9… 16
Frag 17 … 24
Frag 25 … 32

Runnable
Runnable
Runnable
Runnable

Stall
Stall
Stall
Stall

Credit: Kayvon Fatahalian (Stanford)
Hiding Memory Latency

Increase run time of one group
To maximum throughput of many groups

Credit: Kayvon Fatahalian (Stanford)
GPU Architecture Summary

Core Ideas:

1. Many slimmed down cores → lots of parallelism
2. More ALUs, Fewer Control Units
3. Avoid memory stalls by interleaving execution of SIMD groups

Credit: Kayvon Fatahalian (Stanford)
Outline

GPU Architecture (recap)

Programming GPUs
  Intro to OpenCL: The five W’s
Outline

GPU Architecture (recap)

Programming GPUs
  Intro to OpenCL: The five W’s
## GPU Programming: Gains and Losses

<table>
<thead>
<tr>
<th>Gains</th>
<th>Losses</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory Bandwidth (140 GB/s vs. 12 GB/s)</td>
<td>No performance portability</td>
</tr>
<tr>
<td>Compute Bandwidth (Peak: 1 TF/s vs. 50 GF/s, Real: 200 GF/s vs. 10 GF/s)</td>
<td>Data size ↔ Algorithm design</td>
</tr>
<tr>
<td>Data-parallel programming</td>
<td>Cheap branches (i.e. if statements)</td>
</tr>
<tr>
<td>Functional portability between devices (via OpenCL)</td>
<td>Fine-grained malloc/free (*)</td>
</tr>
<tr>
<td></td>
<td>Recursion (*)</td>
</tr>
<tr>
<td></td>
<td>Function pointers (*)</td>
</tr>
<tr>
<td></td>
<td>IEEE 754 FP compliance (*)</td>
</tr>
</tbody>
</table>

*) Less problematic with new hardware. (Nvidia “Fermi”)
### GPU Programming: Gains and Losses

<table>
<thead>
<tr>
<th>Gains</th>
<th>Losses</th>
</tr>
</thead>
</table>
| ✅ Memory Bandwidth  
(140 GB/s vs. 12 GB/s)  
✅ Compute Bandwidth  
(Peak: 1 TF/s vs. 50 GF/s,  
Real: 200 GF/s vs. 10 GF/s)  
✅ Data-parallel programming  
✅ Functional portability between devices (via OpenCL) | 🕵️‍♀️ No performance portability  
🚫 Data size ⇔ Alg. design  
🚫 Cheap branches (i.e. ifs)  
🚫 Fine-grained malloc/free*)  
🚫 Recursion *)  
🚫 Function pointers *)  
🚫 IEEE 754 FP compliance *) |

*) Less problematic with new hardware. (Nvidia “Fermi”)
What is OpenCL?

OpenCL (Open Computing Language) is an open, royalty-free standard for general purpose parallel programming across CPUs, GPUs and other processors. [OpenCL 1.1 spec]

- Device-neutral (Nv GPU, AMD GPU, Intel/AMD CPU)
- Vendor-neutral
- Comes with RTCG

Defines:
- Host-side programming interface (library)
- Device-side programming language (!)
• **Diverse industry participation**  
  - Processor vendors, system OEMs, middleware vendors, application developers

• **Many industry-leading experts involved in OpenCL’s design**  
  - A healthy diversity of industry perspectives

• **Apple made initial proposal and is very active in the working group**  
  - Serving as specification editor
When?

- **Six months from proposal to released OpenCL 1.0 specification**
  - Due to a strong initial proposal and a shared commercial incentive

- **Multiple conformant implementations shipping**
  - Apple’s Mac OS X Snow Leopard now ships with OpenCL

- **18 month cadence between OpenCL 1.0 and OpenCL 1.1**
  - Backwards compatibility protect software investment

---

**OpenCL Timeline**

- **Jun08**: Apple proposes OpenCL working group and contributes draft specification to Khronos
- **Dec08**: Khronos publicly releases OpenCL 1.0 as royalty-free specification
- **May09**: Khronos releases OpenCL 1.0 conformance tests to ensure high-quality implementations
- **2H09**: Multiple conformant implementations ship across diverse OS and platforms
- **Jun10**: OpenCL 1.1 Specification released and first implementations ship

---

Credit: Khronos Group
Processor Parallelism

CPUs
Multiple cores driving performance increases

GPU Architecture (recap)

Graphics APIs and Shading Languages

Multi-processor programming – e.g. OpenMP

Emerging Intersection

Heterogeneous Computing

OpenCL

OpenCL is a programming framework for heterogeneous compute resources

© Copyright Khronos Group, 2010 - Page 3

Credit: Khronos Group
OpenCL: Computing as a Service

Host (CPU)
OpenCL: Computing as a Service

Host (CPU)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)
OpenCL: Computing as a Service

Host (CPU)

- Memory

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)
OpenCL: Computing as a Service

Host (CPU) | Compute Device 0 (Platform 0) | Compute Device 1 (Platform 0) |...| Compute Device 0 (Platform 1) | Compute Device 1 (Platform 1)

Memory

Platform 0 (e.g. CPUs) Platform 1 (e.g. GPUs) (think "chip", has memory interface)

Compute Unit (think "processor", has insn. fetch)

Processing Element (think "SIMD lane")

C "Runtime"

Device Language: ∼ C99

GPU Architecture (recap) Programming GPUs
OpenCL: Computing as a Service

Host (CPU)

- Compute Device 0 (Platform 0)
  - Compute Device 1 (Platform 0)
  - Compute Device 0 (Platform 1)
  - Compute Device 1 (Platform 1)

Platform 0 (e.g. CPUs)
Platform 1 (e.g. GPUs)

(Think "chip", has memory interface)

Compute Unit (think "processor", has insn. fetch)

Processing Element (think "SIMD lane")

C "Runtime"

Device Language: ~ C99

Programming GPUs
OpenCL: Computing as a Service

Platform 0 (e.g. CPUs)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)

Host (CPU)
OpenCL: Computing as a Service

Host (CPU)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)

Platform 1 (e.g. GPUs)
OpenCL: Computing as a Service

Host (CPU)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)
OpenCL: Computing as a Service

Host (CPU)

(think “chip”, has memory interface)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)
OpenCL: Computing as a Service

Host (CPU)

Compute Unit (think “processor”, has insn. fetch)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)

(think “chip”, has memory interface)
OpenCL: Computing as a Service

Host (CPU)

Compute Unit (think “processor”, has insn. fetch)

Processing Element (think “SIMD lane”)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)

(think “chip”, has memory interface)
OpenCL: Computing as a Service

Host (CPU)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)

Platform 0 (e.g. CPUs)
Platform 1 (e.g. GPUs)
OpenCL: Computing as a Service

Host (CPU) -> C "Runtime" -> Compute Device 0 (Platform 0) -> Compute Device 1 (Platform 0) -> Compute Device 0 (Platform 1) -> Compute Device 1 (Platform 1)
OpenCL: Computing as a Service

Host (CPU)

C "Runtime"

Device Language: ~ C99

Compute Device 0 (Platform 0)
Compute Device 1 (Platform 0)
Compute Device 0 (Platform 1)
Compute Device 1 (Platform 1)
Define the “best” N-dimensioned index space for your algorithm
- Global Dimensions: 1024 x 1024 (whole problem space)
- Local Dimensions: 128 x 128 (work group ... executes together)

Synchronization between work-items possible only within workgroups: barriers and memory fences

Cannot synchronize outside of a workgroup
OpenCL: Execution Model

- Two-tiered Parallelism
  - Grid = $N_x \times N_y \times N_z$ work groups
  - Work group = $S_x \times S_y \times S_z$ work items
  - Total: $\prod_{i \in \{x,y,z\}} S_i N_i$ work items
- Abstraction of core/SIMD lane HW concept
- Comm/Sync only within work group
- Grid/Group $\approx$ outer loops in an algorithm
- Device Language:
  
  \[
  \text{get\{} \{\text{global,group,local}\} \_ \{\text{id, size}\} (axis) \]

\[\text{1D Grid} \]

\[\begin{array}{|c|c|c|}
\hline
\text{Group} & \text{Group} & \text{Group} \\
(0, 0) & (1, 0) & (2, 0) \\
\hline
\text{Group} & \text{Group} & \text{Group} \\
(0, 1) & (1, 1) & (2, 1) \\
\hline
\end{array} \]

\[\text{Work Group} (1, 0)\]

\[\begin{array}{|c|c|c|}
\hline
\text{Item} & \text{Item} & \text{Item} & \text{Item} \\
(0, 0) & (1, 0) & (2, 0) & (3, 0) \\
\hline
\text{Item} & \text{Item} & \text{Item} & \text{Item} \\
(0, 1) & (1, 1) & (2, 1) & (3, 1) \\
\hline
\text{Item} & \text{Item} & \text{Item} & \text{Item} \\
(0, 2) & (1, 2) & (2, 2) & (3, 2) \\
\hline
\text{Item} & \text{Item} & \text{Item} & \text{Item} \\
(0, 3) & (1, 3) & (2, 3) & (3, 3) \\
\hline
\end{array} \]
Workgroups: Hardware to Software

HW reality: SIMD lanes
SW abstraction: $n$-dim. work group

How do the two fit together?
→ Lexicographically!

Remember HW shenanigans:
- Quad-pumped Fetch/Decode
- Extra width for latency hiding
Workgroups: Hardware to Software

HW reality: SIMD lanes

SW abstraction: \( n \)-dim. work group

How do the two fit together?

→ Lexicographically!

Remember HW shenanigans:

- Quad-pumped Fetch/Decode
- Extra width for latency hiding
Workgroups: Hardware to Software

HW reality: SIMD lanes
SW abstraction: $n$-dim. work group

How do the two fit together?
→ Lexicographically!

Remember HW shenanigans:
• Quad-pumped Fetch/Decode
• Extra width for latency hiding
Workgroups: Hardware to Software

HW reality: SIMD lanes
SW abstraction: $n$-dim. work group

How do the two fit together?
→ Lexicographically!

Remember HW shenanigans:
- Quad-pumped Fetch/Decode
- Extra width for latency hiding
Workgroups: Hardware to Software

HW reality: SIMD lanes
SW abstraction: $n$-dim. work group

How do the two fit together?
→ Lexicographically!

Remember HW shenanigans:
- Quad-pumped Fetch/Decode
- Extra width for latency hiding
The OpenCL C Language

- **A subset of ISO C99**
  - But without some C99 features such as standard C99 headers, function pointers, recursion, variable length arrays, and bit fields

- **A superset of ISO C99 with additions for:**
  - Work-items and workgroups
  - Vector types
  - Synchronization
  - Address space qualifiers

- **Also includes a large set of built-in functions**
  - Image manipulation
  - Work-item manipulation,
  - Specialized math routines, etc.
Dive into OpenCL: Preparation

```c
#include "cl-helper.h"

int main()
{
    // init
    cl_context ctx; cl_command_queue queue;
    create_context_on("NVIDIA", NULL, 0, &ctx, &queue, 0);

    // allocate and initialize CPU memory
    const size_t sz = 10000;
    float a[sz];
    for (size_t i = 0; i < sz; ++i) a[i] = i;
}
```
// allocate GPU memory, transfer to GPU

cl_int status;
cl_mem buf_a = clCreateBuffer(ctx, CL_MEM_READ_WRITE,
    sizeof(float) * sz, 0, &status);
CHECK_CL_ERROR(status, "clCreateBuffer");

CALL_CL_GUARDED(clEnqueueWriteBuffer, (queue, buf_a, /*blocking*/ CL_TRUE, /*offset*/ 0,
    sz * sizeof(float), a,
    0, NULL, NULL));
Dive into OpenCL: Running

```c
// load kernels
char *knl_text = read_file("twice.cl");
cl_kernel knl = kernel_from_string(ctx, knl_text, "twice", NULL);
free(knl_text);

// run code on GPU
SET_1_KERNEL_ARG(knl, buf_a);
size_t gdim[] = {sz};
size_t ldim[] = {1};
CALL_CL_GUARDED(clEnqueueNDRangeKernel,
    (queue, knl,
     /*dimensions*/ 1, NULL, gdim, ldim,
     0, NULL, NULL));

__kernel void twice(__global float *a)
{ a[get_global_id(0)] *= 2; }
```
Dive into OpenCL: Clean-up

```
43    // clean up...
44    CALL_CL_GUARDED(clReleaseMemObject, (buf_a));
45    CALL_CL_GUARDED(clReleaseKernel, (knl));
46    CALL_CL_GUARDED(clReleaseCommandQueue, (queue));
47    CALL_CL_GUARDED(clReleaseContext, (ctx));
48    }
```

Why check for errors?

- GPUs have (some) memory protection
- Invalid sizes (block/grid/...)
- Out of memory, access restriction, hardware limitations, etc.
Dive into OpenCL: Clean-up

```c
// clean up...
CALL_CL_GUARDED(clReleaseMemObject, (buf_a));
CALL_CL_GUARDED(clReleaseKernel, (knl));
CALL_CL_GUARDED(clReleaseCommandQueue, (queue));
CALL_CL_GUARDED(clReleaseContext, (ctx));
}
```

Why check for errors?

- GPUs have (some) memory protection
- Invalid sizes (block/grid/...)
- Out of memory, access restriction, hardware limitations, etc.

Does this code use the hardware well?
Getting your feet wet

Thinking about GPU programming
How would we modify the program to...
  • ...print the contents of the result?
Thinking about GPU programming

How would we modify the program to...

• ... print the contents of the result?
• ... compute $c_i = a_i b_i$?
Getting your feet wet

Thinking about GPU programming

How would we modify the program to...

- ...print the contents of the result?
- ...compute \( c_i = a_i b_i \) ?
- ...use groups of 256 work items each?
Getting your feet wet

Thinking about GPU programming
How would we modify the program to...

• ...print the contents of the result?
• ...compute $c_i = a_i b_i$?
• ...use groups of 256 work items each?
• ...use groups of $16 \times 16$ work items?