High-Performance Scientific Computing

Lecture 4: OpenCL

MATH-GA 2011 / CSCI-GA 2945 · September 26, 2012
Today

Tool of the day: Make

Chips for Throughput

OpenCL: Overview

OpenCL: Between host and device

OpenCL: Device Language

OpenCL: Synchronization
Bits and pieces

• HW1 graded before weekend
• HW2 due
• HW3 out
• Sign up for HPC account
• Any more OMP questions?
• OMP anecdote
Final project

Examples from two years ago:
- GPU-parallel finite difference solver in flexible geometries
- GPU-parallel password cracking
- MPI-parallel CFD via the vortex method
- GPU-parallel ruling extraction (geometry)

Remarks:
- Group projects encouraged!
- Use the mailing list to find buddies
- Non-numerical algorithms ok
Outline

Tool of the day: Make

Chips for Throughput

OpenCL: Overview

OpenCL: Between host and device

OpenCL: Device Language

OpenCL: Synchronization
Demo time
Tool of the day: Make

Chips for Throughput

OpenCL: Overview

OpenCL: Between host and device

OpenCL: Device Language

OpenCL: Synchronization
CPU Chip Real Estate

65 nm, 4 SP ops at a time, 1 MiB L2.
“CPU-style” Cores

Credit: Kayvon Fatahalian (Stanford)
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 independent instruction streams

Reality: instruction streams not actually very different/independent

Credit: Kayvon Fatahalian (Stanford)
Saving Yet More Space

Recall: simple processing core
Fetch/Decode
ALU (Execute)
Execution Context

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
Idea #2

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

→ SIMD
Idea #2
Amortize cost/complexity of managing an instruction stream across many ALUs
→ SIMD
Gratuitous Amounts of Parallelism!

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?

Credit: Kayvon Fatahalian (Stanford)
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?

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

\[
\begin{align*}
\text{if } (x > 0) \{ \\
\quad y &= \text{pow}(x, \exp); \\
\quad y &= K_s; \\
\quad \text{refl} &= y + K_a; \\
\} \text{ else } \{ \\
\quad x &= 0; \\
\quad \text{refl} &= K_a; \\
\}
\end{align*}
\]

<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;
}
```

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;
}
```
Recent Processor Architecture

- Commodity chips
- “Infinitely” many cores
- “Infinite” vector width
- Must hide memory latency (→ ILP, SMT)

- Compute bandwidth ≫ Memory bandwidth
- Bandwidth only achievable by *homogeneity*

Timeline:
- NV GT200 (2008)
- NV Fermi (2010)
- Intel IVB (2012)
- AMD Tahiti (2012)
- NV GK110 (2012?)
Outline

Tool of the day: Make

Chips for Throughput

OpenCL: Overview

OpenCL: Between host and device

OpenCL: Device Language

OpenCL: Synchronization
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 ‘JIT’ compilation

Defines:

- Host-side programming interface (library)
- Device-side programming language (!)
OpenCL: Vocabulary

Host (CPU)
OpenCL: Vocabulary

- **Host (CPU)**
- **Memory**
- **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

Tool of the day: Make Chips for Throughput OpenCL: Overview OpenCL: Between host and device OpenCL: Device Language OpenCL: Synchronization
OpenCL: Vocabulary
OpenCL: Vocabulary

- Host (CPU)
- Memory
- Compute Device 0 (Platform 0)
- Memory
- Compute Device 1 (Platform 0)
- Memory
- Compute Device 0 (Platform 1)
- Memory
- Compute Device 1 (Platform 1)
- Memory
OpenCL: Vocabulary

Host (CPU)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)
OpenCL: Vocabulary

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: Vocabulary

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: Vocabulary

Host (CPU)

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

Compute Device 0 (Platform 1)
Compute Device 1 (Platform 1)
OpenCL: Vocabulary

(host “chip”, has memory interface)

Host (CPU)

Compute Device 0 (Platform 0)

Compute Device 1 (Platform 0)

Compute Device 0 (Platform 1)

Compute Device 1 (Platform 1)
OpenCL: Vocabulary

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)

Platform 0 (e.g. CPUs)

Platform 1 (e.g. GPUs)

(think “chip”, has memory interface)
OpenCL: Vocabulary

Host (CPU)

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)

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

Processing Element (think “SIMD lane”)

Platform 0 (e.g. CPUs)

Platform 1 (e.g. GPUs)

(think “chip”, has memory interface)
OpenCL: Vocabulary

- Host (CPU)
- Compute Device 0 (Platform 0)
  - ... (Processing Elements)
- Compute Device 1 (Platform 0)
  - ... (Processing Elements)
- Compute Device 0 (Platform 1)
  - ... (Processing Elements)
- Compute Device 1 (Platform 1)
  - ... (Processing Elements)

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

Tool of the day: Make Chips for Throughput  OpenCL: Overview  OpenCL: Between host and device  OpenCL: Device Language
OpenCL: Vocabulary

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: Vocabulary

- **Host (CPU)**
- **Compute Device 0 (Platform 0)**
  - Compute Device 1 (Platform 0)
- **Compute Device 0 (Platform 1)**
  - Compute Device 1 (Platform 1)

**Device Language:** \( \sim \text{C99} \)
Who cares how many cores? 

Idea:
• Program as if there were “infinitely” many cores
• Program as if there were “infinitely” many ALUs per core

Consider: Which is easy to do automatically?
• Parallel program → sequential hardware
• Sequential program → parallel hardware?

Really: Group provides pool of parallelism to draw from.

X, Y, Z order within group matters. (Not among groups, though.)

Grids can be 1, 2, 3-dimensional.

Tool of the day: Make Chips for Throughput 

OpenCL: Overview
OpenCL: Between host and device
OpenCL: Device Language
### Connection: Hardware ↔ Programming Model

**Who cares how many cores?**

<table>
<thead>
<tr>
<th>Fetch/Decode</th>
<th>Fetch/Decode</th>
<th>Fetch/Decode</th>
</tr>
</thead>
<tbody>
<tr>
<td>32 KiB Ctx Private (&quot;Registers&quot;)</td>
<td>32 KiB Ctx Private (&quot;Registers&quot;)</td>
<td>32 KiB Ctx Private (&quot;Registers&quot;)</td>
</tr>
<tr>
<td>16 KiB Ctx Shared</td>
<td>16 KiB Ctx Shared</td>
<td>16 KiB Ctx Shared</td>
</tr>
</tbody>
</table>

### Idea:
- Program as if there were "infinitely" many cores
- Program as if there were "infinitely" many ALUs per core

**Consider:** Which is easy to do automatically?
- Parallel program → sequential hardware
- Sequential program → parallel hardware?

**Axis 0**
**Axis 1**
**Hardware**
**Software representation**

**Really:** Group provides pool of parallelism to draw from. X, Y, Z order within group matters. (Not among groups, though.)

**Grids can be 1, 2, 3-dimensional.**

---

**Tool of the day:** Make Chips for Throughput

**OpenCL: Overview**
**OpenCL: Between host and device**
**OpenCL: Device Language**
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Idea:

- Program as if there were “infinitely” many cores
- Program as if there were “infinitely” many ALUs per core
Who cares how many cores?

Consider: Which is easy to do automatically?

- **Parallel program** → sequential hardware

  or

- **Sequential program** → parallel hardware?

---

Axes 0

Axis 1

Hardware

Software representation

Really: Group provides pool of parallelism to draw from.

X, Y, Z order within group matters. (Not among groups, though.)

Grids can be 1, 2, 3-dimensional.

Tool of the day: Make Chips for Throughput

OpenCL: Overview

OpenCL: Between host and device

OpenCL: Device Language
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Idea:
• Program as if there were "infinitely" many cores
• Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
• Parallel program → sequential hardware or
• Sequential program → parallel hardware?

Really: Group provides pool of parallelism to draw from.

Axis 0
Axis 1
Hardware
Software representation

Tool of the day: Make Chips for Throughput
OpenCL: Overview
OpenCL: Between host and device
OpenCL: Device Language
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Idea:
• Program as if there were “infinitely” many cores
• Program as if there were “infinitely” many ALUs per core

Consider: Which is easy to do automatically?
• Parallel program → sequential hardware
• Sequential program → parallel hardware?

Axis 0

Axis 1

Really: Group provides pool of parallelism to draw from.
X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.

Tool of the day: Make Chips for Throughput
OpenCL: Overview
OpenCL: Between host and device
OpenCL: Device Language
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Idea:
• Program as if there were "infinitely" many cores
• Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
• Parallel program → sequential hardware
• Sequential program → parallel hardware?

Axis 0

Axis 1

Hardware

Tool of the day: Make Chips for Throughput
OpenCL: Overview  OpenCL: Between host and device  OpenCL: Device Language
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Idea:
• Program as if there were "infinitely" many cores
• Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
• Parallel program → sequential hardware
• Sequential program → parallel hardware?

Really: Group provides pool of parallelism to draw from.
X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.

Tool of the day: Make Chips for Throughput
OpenCL: Overview
OpenCL: Between host and device
OpenCL: Device Language
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Idea:

• Program as if there were "infinitely" many cores
• Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?

• Parallel program → sequential hardware
• Sequential program → parallel hardware?

Axis 0

Axis 1

Grid (Kernel: Function on Grid)

Software representation

Hardware

Tool of the day: Make Chips for Throughput  OpenCL: Overview  OpenCL: Between host and device  OpenCL: Device Language
Connection: Hardware ↔ Programming Model

(Work) Group
Grid
(Kernel: Function on Grid)

Axis 0
Axis 1

Software representation

Hardware

Who cares how many cores?

Idea:
• Program as if there were "infinitely" many cores
• Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
• Parallel program → sequential hardware
or
• Sequential program → parallel hardware?

Grids can be 1, 2, 3-dimensional.

Tool of the day: Make Chips for Throughput
OpenCL: Overview
OpenCL: Between host and device
OpenCL: Device Language
Who cares how many cores?

Idea:
- Program as if there were "infinitely" many cores
- Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
- Parallel program → sequential hardware
- Sequential program → parallel hardware?
Connection: Hardware ↔ Programming Model

Software representation

Hardware

Axis 0

Axis 1

Who cares how many cores?

Idea:
• Program as if there were "infinitely" many cores
• Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
• Parallel program → sequential hardware
• Sequential program → parallel hardware?

Really: Group provides pool of parallelism to draw from.

X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.

Tool of the day: Make Chips for Throughput

OpenCL: Overview
OpenCL: Between host and device
OpenCL: Device Language
Who cares how many cores?

Idea:
• Program as if there were "infinitely" many cores
• Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
• Parallel program → sequential hardware or
• Sequential program → parallel hardware?

Really: Group provides pool of parallelism to draw from.

X, Y, Z order within group matters. (Not among groups, though.)

Grids can be 1, 2, 3-dimensional.
Who cares how many cores?

Idea:

• Program as if there were "infinitely" many cores
• Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?

• Parallel program → sequential hardware
• Sequential program → parallel hardware?

Really: Group provides pool of parallelism to draw from.

X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.

Software representation

Hardware

Connection: Hardware ↔ Programming Model
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Idea:
- Program as if there were "infinitely" many cores
- Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
- Parallel program → sequential hardware
- Sequential program → parallel hardware?

Axis 0

Axis 1

Software representation

Hardware

Really: Group provides pool of parallelism to draw from.
X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.

Tool of the day: Make Chips for Throughput

OpenCL: Overview

OpenCL: Between host and device

OpenCL: Device Language
Connection: Hardware ↔ Programming Model

- Axis 0
- Axis 1

Software representation

Hardware

Who cares how many cores?

Idea:
- Program as if there were "infinitely" many cores
- Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
- Parallel program → sequential hardware
- Sequential program → parallel hardware?

Really: Group provides pool of parallelism to draw from.

X, Y, Z order within group matters. (Not among groups, though.)

Grids can be 1, 2, 3-dimensional.
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Idea:
• Program as if there were "infinitely" many cores
• Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
• Parallel program → sequential hardware
  or
• Sequential program → parallel hardware?

Axis 0
Axis 1

Software representation

Hardware

Really: Group provides pool of parallelism to draw from.

X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.

Tool of the day: Make Chips for Throughput

OpenCL: Overview
OpenCL: Between host and device
OpenCL: Device Language
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Idea:
- Program as if there were "infinitely" many cores
- Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
- Parallel program → sequential hardware
- Sequential program → parallel hardware?

Axis 0

Axis 1

Software representation

Hardware

Really: Group provides pool of parallelism to draw from.

X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1, 2, 3-dimensional.

Tool of the day: Make Chips for Throughput

OpenCL: Overview

OpenCL: Between host and device

OpenCL: Device Language
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Idea:
• Program as if there were "infinitely" many cores
• Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
• Parallel program → sequential hardware
• Sequential program → parallel hardware?

Really: Group provides pool of parallelism to draw from.
X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.

Tool of the day: Make Chips for Throughput

OpenCL: Overview
OpenCL: Between host and device
OpenCL: Device Language
Who cares how many cores?

Idea:
• Program as if there were "infinitely" many cores
• Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
• Parallel program → sequential hardware
• Sequential program → parallel hardware?

Really: Group provides pool of parallelism to draw from.
X, Y, Z order within group matters. (Not among groups, though.)

Grids can be 1, 2, 3-dimensional.
Really: Group provides pool of parallelism to draw from.

X,Y,Z order within group matters. (Not among groups, though.)
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Idea:
• Program as if there were "infinitely" many cores
• Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
• Parallel program → sequential hardware
• Sequential program → parallel hardware?

Axis 0
Axis 1

Really: Group provides pool of parallelism to draw from.
X,Y,Z order within group matters. (Not among groups, though.)

Grids can be 1,2,3-dimensional.

Tool of the day: Make Chips for Throughput
OpenCL: Overview
OpenCL: Between host and device
OpenCL: Device Language
Who cares how many cores?

Idea:
- Program as if there were "infinitely" many cores
- Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?

- Parallel program → sequential hardware
- Sequential program → parallel hardware?

Really: Group provides pool of parallelism to draw from.

X, Y, Z order within group matters. (Not among groups, though.)

Grids can be 1, 2, 3-dimensional.
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Idea:
- Program as if there were "infinitely" many cores
- Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
- Parallel program → sequential hardware
- Sequential program → parallel hardware?

Really: Group provides pool of parallelism to draw from.

X, Y, Z order within group matters. (Not among groups, though.)

- get_local_id(axis):size(axis)?
- get_group_id(axis):num_groups(axis)?
- get_global_id(axis):size(axis)?
Who cares how many cores?

Idea:
- Program as if there were "infinitely" many cores
- Program as if there were "infinitely" many ALUs per core

Consider: Which is easy to do automatically?
- Parallel program → sequential hardware
- Sequential program → parallel hardware?

Axis 0
Axis 1

Grids can be 1,2,3-dimensional.

"get_local_id(axis)?/size(axis)?"
"get_group_id(axis)?/num_groups(axis)?"
"get_global_id(axis)?/size(axis)?"

Tools of the day:
- Make Chips for Throughput
- OpenCL: Overview
- OpenCL: Between host and device
- OpenCL: Device Language
Demo time
Tool of the day: Make

Chips for Throughput

OpenCL: Overview

OpenCL: Between host and device

OpenCL: Device Language

OpenCL: Synchronization
OpenCL: Command Queues

- Host and Device run asynchronously
- Host submits to queue:
  - Computations
  - Memory Transfers
  - Sync primitives
  - ...
- Host can wait for drained queue
- Profiling
Outline

Tool of the day: Make

Chips for Throughput

OpenCL: Overview

OpenCL: Between host and device

OpenCL: Device Language

OpenCL: Synchronization
OpenCL device language is C99, with these differences:

+ Index getters
+ Memory space qualifiers
+ Vector data types
+ Many generic (‘overloaded’) math functions
+ Synchronization
- Recursion
- Fine-grained malloc()
- Function pointers
## Address Space Qualifiers

<table>
<thead>
<tr>
<th>Type</th>
<th>Per</th>
<th>“Speed”</th>
</tr>
</thead>
<tbody>
<tr>
<td>private*)</td>
<td>work item</td>
<td>super-fast</td>
</tr>
<tr>
<td>local</td>
<td>group</td>
<td>fast</td>
</tr>
<tr>
<td>global</td>
<td>grid</td>
<td>kinda slow</td>
</tr>
</tbody>
</table>

*) default, so optional

Should really discuss “speed” in terms of latency/bandwidth. Both decrease with distance from the point of execution.
### Address Space Qualifiers

<table>
<thead>
<tr>
<th>Type</th>
<th>Per</th>
<th>“Speed”</th>
</tr>
</thead>
<tbody>
<tr>
<td>private*</td>
<td>work item</td>
<td>super-fast</td>
</tr>
<tr>
<td>local</td>
<td>group</td>
<td>fast</td>
</tr>
<tr>
<td>global</td>
<td>grid</td>
<td>kinda slow</td>
</tr>
</tbody>
</table>

*) default, so optional

Should really discuss “speed” in terms of latency/bandwidth.

*Both* decrease with distance from the point of execution.
Outline

Tool of the day: Make

Chips for Throughput

OpenCL: Overview

OpenCL: Between host and device

OpenCL: Device Language

OpenCL: Synchronization
Concurrency and Synchronization

GPUs have layers of concurrency.
Each layer has its synchronization primitives.
GPUs have layers of concurrency.
Each layer has its synchronization primitives.

- **Intra-group:**
  - `barrier(...)`,
  - `mem_fence(...)`
  - `... = CK_{LOCAL,GLOBAL}_MEM_FENCE`

- **Inter-group:**
  - Kernel launch

- **CPU-GPU:**
  - Command queues, Events
Synchronization

What is a Barrier?
What is a Barrier?
Synchronization

What is a Barrier?
What is a Barrier?
Synchronization

What is a Barrier?
Synchronization

What is a Barrier?
Synchronization

What is a Barrier?
What is a Memory Fence?
Synchronization

What is a Memory Fence?

write 18

17
What is a Memory Fence?

```
write 18

17

read
```
What is a Memory Fence?
What is a Memory Fence?
Synchronization

What is a Memory Fence?

write 18

18
What is a Memory Fence?
What is a Memory Fence? An ordering restriction for memory access.
Synchronization

What is a Memory Fence? An ordering restriction for memory access.
What is a Memory Fence? An ordering restriction for memory access.
Synchronization

What is a Memory Fence? An ordering restriction for memory access.
What is a Memory Fence? An ordering restriction for memory access.
What is a Memory Fence? An ordering restriction for memory access.
What is a Memory Fence? An ordering restriction for memory access.
What is a Memory Fence? An ordering restriction for memory access.
Golden Rule:
Results of the algorithm must be independent of the order in which work groups are executed.
Golden Rule:
Results of the algorithm must be independent of the order in which work groups are executed.

Consequences:

- Work groups may read the same information from global memory.
- But: Two work groups may not validly write different things to the same global memory.
- Kernel launch serves as
  - Global barrier
  - Global memory fence
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

How?

`atomic`

```c
{add, inc, cmpxchg, ...}(int *global, int value);
```
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!
Atomic Operations

Collaborative (inter-block) Global Memory Update:

[Diagram showing the process of Read, Increment, and Write with Interruptible annotations]

How?

atomic {
  add, inc, cmpxchg, ...
}(int *global, int value);

Tool of the day: Make Chips for Throughput
OpenCL: Overview
OpenCL: Between host and device
OpenCL: Device Language
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Atomic Global Memory Update:
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Atomic Global Memory Update:

Tool of the day: Make Chips for Throughput OpenCL: Overview OpenCL: Between host and device OpenCL: Device Language
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!

Atomic Global Memory Update:

Read → Increment → Write

Protected

Protected

How?

atomic {
add, inc, cmpxchg, . . .
} (int *global, int value);
Atomic Operations

Collaborative (inter-block) Global Memory Update:

Read → Increment → Write

Interruptible!

Atomic Global Memory Update:

Read → Increment → Write

Protected

How?
atomic_{add,inc,cmpxchg, ...}(int *global, int value);
**Atomic: Compare-and-swap**

```c
int atomic_cmpxchg (__global int *p, int cmp, int val)
int atomic_cmpxchg (__local int *p, int cmp, int val)
```

Does:

- Read the 32-bit value (referred to as old) stored at location pointed by p.
- Compute `(old == cmp) ? val : old`.
- Store result at location pointed by p.
- Returns `old`.

---

Tool of the day: Make  Chips for Throughput  OpenCL: Overview  OpenCL: Between host and device  OpenCL: Device Language
Atomic: Compare-and-swap

```c
int atomic_cmpxchg(__global int *p, int cmp, int val)
int atomic_cmpxchg(__local int *p, int cmp, int val)
```

Does:

- Read the 32-bit value (referred to as old) stored at location pointed by p.
- Compute \((\text{old} == \text{cmp}) \ ? \ \text{val} : \ \text{old})\.
- Store result at location pointed by p.
- Returns old.

Implement atomic float add?
Questions?
Image Credits

- Isaiah die shot: VIA Technologies
- Onions: flickr.com/darwinbell (cc)