Forward Progress on GPU Concurrency

Alastair Donaldson, Imperial College London

Slides and accompanying code prepared jointly with Tyler Sorensen

Accompanying paper written jointly with Jeroen Ketema, Tyler Sorensen and John Wickerson
Agenda

• Overview of key concepts in OpenCL – a major GPU programming model
• Implementing a reduction in OpenCL, showcasing pitfalls
• Demo of GPUVerify – static data race detection for OpenCL
• Discussion of how to achieve global synchronization in OpenCL
• Problems with global synchronization due to unfair scheduling
• Discovery protocol to enable portable global synchronization
Threads and blocks

An OpenCL kernel is executed by a set of *threads*

The threads are sub-divided into *workgroups* of equal size
Example: 3 blocks, 4 threads per block

```
<table>
<thead>
<tr>
<th>global id:</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
</tr>
</thead>
<tbody>
<tr>
<td>local id:</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>group id:</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
```

```
global id = group size \times group id + local id
```
Memory

Each thread has access to its own *private* memory

Threads in a workgroup share *local* memory

All threads share *global* memory
Barrier synchronization

Workgroup 0

\[ T_0, T_1, T_2, T_3 \]

Workgroup 1

\[ T_4, T_5, T_6, T_7 \]
Barrier synchronization

Workgroup 0

\[ T_0 \quad T_1 \quad T_2 \quad T_3 \]

Barrier

Workgroup 1

\[ T_4 \quad T_5 \quad T_6 \quad T_7 \]
Barrier synchronization

Workgroup 0

\( T_0 \quad T_1 \quad T_2 \quad T_3 \)

Barrier

Safe to read data written before the barrier

Workgroup 1

\( T_4 \quad T_5 \quad T_6 \quad T_7 \)
Global barrier?

Workgroup 0

\[ T_0 \quad T_1 \quad T_2 \quad T_3 \]

Global barrier

Workgroup 1

\[ T_4 \quad T_5 \quad T_6 \quad T_7 \]
Global barrier?

<table>
<thead>
<tr>
<th>Workgroup 0</th>
<th>Workgroup 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_0$</td>
<td>$T_4$</td>
</tr>
<tr>
<td>$T_1$</td>
<td>$T_5$</td>
</tr>
<tr>
<td>$T_2$</td>
<td>$T_6$</td>
</tr>
<tr>
<td>$T_3$</td>
<td>$T_7$</td>
</tr>
</tbody>
</table>

Global barrier
Global barrier?

Useful construct, but not provided as a primitive in OpenCL – more later!

Safe to read data written before the barrier
Atomics

OpenCL 2.0 provides atomic operations for fine-grained communication between threads.

The OpenCL memory model is relaxed: non sequentially consistent behaviours are allowed.
Data race

Two memory accesses *race* if and only if:
- They are issued by different threads
- They access a common memory location
- At least one access is non-atomic
- At least one access modifies the memory location
- No synchronization operation separates the accesses

Synchronization can be via a *barrier*, or through *synchronizing atomics*
Let’s program a reduction in OpenCL

\[
\text{reduce}([x_1, x_2, ..., x_n]) = x_1 + x_2 + ... + x_n
\]
Reduction example with a single workgroup

threads:

partial sums:

data:

result:
GPUVerify: static data race analysis for GPU kernels

- OpenCL kernel
- CUDA kernel

Front-end: build on CLANG/LLVM

Kernel transformation engine

Sequential Boogie program

Candidate loop invariants

Widely used, very robust

Z3 SMT solver

CVC4 SMT solver

Boogie verification engine

Reusing existing infrastructures makes soundness easier to argue

Our innovations are here
To learn more about GPUVerify:

The original paper:

• Adam Betts, Nathan Chong, Alastair F. Donaldson, Shaz Qadeer, Paul Thomson: *GPUVerify: a verifier for GPU kernels*. OOPSLA 2012: 113-132

Extended journal version:


Tool paper on engineering details, including optimizations:

Global barrier motivation: graph traversal

• Frontier based graph traversal framework
Global barrier motivation: graph traversal

• Frontier based graph traversal framework
Global barrier motivation: graph traversal

- Frontier based graph traversal framework
Global barrier motivation: graph traversal

- Frontier based graph traversal framework
Global barrier motivation: graph traversal

- Frontier based graph traversal framework

```
  Dependencies!!

  base

  frontier 1

  frontier 2

  ...

  frontier n
```
Global barrier motivation: graph traversal

• Frontier based graph traversal framework

Dependencies!!

How to achieve global synchronization?

...
Global barrier motivation: graph traversal

• Frontier based graph traversal framework

without global barrier, need to call GPU $n$ times
Global barrier motivation: graph traversal

• Frontier based graph traversal framework

How to achieve global synchronization?

*Dependencies!!*

with global barrier, can use blocking synchronization
Barrier implementation

• Start with an implementation from Xiao and Feng (2010)
  • Written in CUDA (ported to OpenCL)
  • No formal memory consistency properties
Slave Workgroup 0
T1 T0

Master Workgroup
T0 T1

Slave Workgroup 1
T0 T1

barrier

spin(x_0 \neq 1) \quad spin(x_1 \neq 1)
Master Workgroup

Slave Workgroup 0
T1    T0

Slave Workgroup 1
T0    T1

barrier

T0

spin(x_0 \neq 1)    spin(x_1 \neq 1)

barrier
Slave Workgroup 0

T1  T0

barrier

\( x_0 = 1 \)

barrier

Master Workgroup

T0  T1

\( \text{spin}(x_0 \neq 1) \quad \text{spin}(x_1 \neq 1) \)

Slave Workgroup 1

T0  T1

barrier

\( x_1 = 1 \)
Master Workgroup

T0  T1

T0

spin(x₀ ≠ 1)  spin(x₁ ≠ 1)

T1

Slave Workgroup 0

T1  T0

barrier

x₀ = 1

spin(x₀ ≠ 0)

Slave Workgroup 1

T0  T1

barrier

x₁ = 1

spin(x₁ ≠ 0)
T1  T0
Slave Workgroup 0

barrier

\[ x_0 = 1 \]

spin(\( x_0 \neq 0 \))

barrier

T1  T0
Slave Workgroup 0

barrier

\[ x_0 = 1 \]

spin(\( x_0 \neq 1 \))

\[ x_0 = 0 \]

barrier

x_1 = 0

Slave Workgroup 1

T0  T1

barrier

\[ x_1 = 1 \]

spin(\( x_1 \neq 0 \))

barrier

Spin Workgroup 1

T0  T1

barrier

\[ x_1 = 0 \]

spin(\( x_1 \neq 1 \))

\[ x_1 = 0 \]

barrier

Spin Workgroup 1
barrier spin(x_0 != 0)

Master Workgroup

spin(x_0 != 1) spin(x_1 != 1)

barrier

x_0 = 0 x_1 = 0

Slave Workgroup 0

T1 T0

barrier

x_0 = 1

spin(x_0 != 0)

Slave Workgroup 1

T0 T1

barrier

x_1 = 1

spin(x_1 != 0)

T0

barrier
\( \text{spin}(x_0 \neq 0) \)

\( \text{spin}(x_0 \neq 1) \)

\( T_1 \rightarrow T_0 \)

\( x_0 = 1 \)

\( \text{spin}(x_0 \neq 0) \)

\( x_0 = 0 \)

\( \text{spin}(x_1 \neq 0) \)

\( x_1 = 1 \)
Let’s implement this
Barrier memory consistency

• Device release-acquire rule

T0 and T1 in different workgroups

\[ W_{rel} y = 1 \quad \rightarrow \quad R_{acq} y = 1 \]
\( \text{spin}(x_0 \neq 0) \)

\( x_0 = 1 \)

\( \text{spin}(x_0 \neq 0) \)

\( \text{spin}(x_1 \neq 1) \)

\( x_1 = 0 \)

\( \text{spin}(x_1 \neq 0) \)

\( x_0 = 0 \)

\( x_1 = 1 \)
\( \text{spin}(R_{\text{acq}}x_0 \neq 0) \)

\( W_{\text{rel}} x_0 = 1 \)

\( \text{spin}(R_{\text{acq}}x_1 \neq 0) \)

\( W_{\text{rel}} x_1 = 1 \)

\( W_{\text{rel}} x_0 = 0 \)

\( W_{\text{rel}} x_1 = 0 \)

\( \text{Master Workgroup} \)

\( T0 \)

\( T1 \)

\( \text{Slave Workgroup 0} \)

\( T1 \)

\( T0 \)

\( \text{Slave Workgroup 1} \)

\( T0 \)

\( T1 \)

\( \text{W}_{\text{rel}} x_1 = 1 \)

\( \text{spin}(R_{\text{acq}}x_1 \neq 0) \)

\( \text{W}_{\text{rel}} x_0 = 0 \)

\( \text{W}_{\text{rel}} x_1 = 0 \)
Let’s implement this
Problem with global barrier

• Global synchronization leads to **deadlock** if too many workgroups participate in barrier
Occupancy bound execution

Program with 5 workgroups

| w5 | w4 | w2 | w1 | w0 |

workgroup queue

GPU with 3 compute units
Occupancy bound execution
Occupancy bound execution

Program with 5 workgroups

workgroup queue

GPU with 3 compute units
Occupancy bound execution

Program with 5 workgroups

workgroup queue

GPU with 3 compute units

finished workgroups
Occupancy bound execution

Program with 5 workgroups

workgroup queue

w5

w0

w1

w4

CU

CU

CU

GPU with 3 compute units

finished workgroups

w2
Occupancy bound execution

Program with 5 workgroups

workgroup queue

w5

w2 w0

finished workgroups

w1

w4

GPU with 3 compute units
Occupancy bound execution

Program with 5 workgroups

workgroup queue

GPU with 3 compute units

finished workgroups

w0
w2
w5
w1
w4
Occupancy bound execution

Program with 5 workgroups

workgroup queue

Finished!

GPU with 3 compute units

finished workgroups
Occupancy bound execution

Program with 5 workgroups

w5 w4 w2 w1 w0

workgroup queue

GPU with 3 compute units
Occupancy bound execution

Program with 5 workgroups: w5, w4, w2, w1, w0

Workgroup queue

GPU with 3 compute units: CU, CU, CU
Occupancy bound execution

Program with 5 workgroups

**Cannot synchronise with workgroups in queue**

**Barrier gives deadlock!**

GPU with 3 compute units
Occupyancy bound execution

Program with 5 workgroups

workgroup queue

Barrier is possible if we know the occupancy

GPU with 3 compute units
Occupancy bound execution

• Launch as many workgroups as compute units?
Occupancy bound execution

• Launch as many workgroups as compute units?
Occupancy bound execution

- Launch as many workgroups as compute units?

Program with 5 workgroups: w5, w4, w2, w1, w0

Workgroup queue

GPU with 3 compute units: CU, CU, CU
Occupancy bound execution

- Launch as many workgroups as compute units?

Program with 5 workgroups

Depending on resources, multiple wgs can execute on CU

GPU with 3 compute units
## Recall of occupancy discovery

<table>
<thead>
<tr>
<th>Chip</th>
<th>Compute Units</th>
<th>Occupancy Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 980</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>Quadro K500</td>
<td>12</td>
<td></td>
</tr>
<tr>
<td>Iris 6100</td>
<td>47</td>
<td></td>
</tr>
<tr>
<td>HD 5500</td>
<td>24</td>
<td></td>
</tr>
<tr>
<td>Radeon R9</td>
<td>28</td>
<td></td>
</tr>
<tr>
<td>Radeon R7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>T628-4</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>T628-2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
Recall of occupancy discovery

<table>
<thead>
<tr>
<th>Chip</th>
<th>Compute Units</th>
<th>Occupancy Bound</th>
</tr>
</thead>
<tbody>
<tr>
<td>GTX 980</td>
<td>16</td>
<td>32</td>
</tr>
<tr>
<td>Quadro K500</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>Iris 6100</td>
<td>47</td>
<td>6</td>
</tr>
<tr>
<td>HD 5500</td>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td>Radeon R9</td>
<td>28</td>
<td>48</td>
</tr>
<tr>
<td>Radeon R7</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>T628-4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>T628-2</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>
Our approach (scheduling)

Program

<table>
<thead>
<tr>
<th>w0</th>
<th>w9</th>
<th>w8</th>
<th>w7</th>
<th>w6</th>
</tr>
</thead>
<tbody>
<tr>
<td>w5</td>
<td>w4</td>
<td>w2</td>
<td>w1</td>
<td>w0</td>
</tr>
</tbody>
</table>

workgroup queue

GPU with 3 compute units
Our approach (scheduling)

Program

workgroup queue

GPU with 3 compute units
Our approach (scheduling)

Program

workgroup queue

Dynamically estimate occupancy

GPU with 3 compute units
Our approach (scheduling)

Program

workgroup queue

Dynamically estimate occupancy

GPU with 3 compute units
Finding occupant workgroups

• Executed by 1 thread per workgroup

• Two phases:
  • Polling
  • Closing
Finding occupant workgroups

• Executed by 1 thread per workgroup

• Three global variables:
  • Mutex: m
  • Bool poll flag: poll_open
  • Integer counter: count
Finding occupant workgroups

- Executed by 1 thread per workgroup

- Three global variables:
  - Mutex: m
  - Bool poll flag: poll_open
  - Integer counter: count

```c
lock(m)
if (poll_open) {
    count++;
    unlock(m);
} else {
    unlock(m);
    return false;
}
```

```c
lock(m)
if (poll_open) {
    poll_open = false
} unlock(m)
return true;
```
Finding occupant workgroups

- Executed by 1 thread per workgroup

- Three global variables:
  - Mutex: \( m \)
  - Bool poll flag: \( \text{poll\_open} \)
  - Integer counter: \( \text{count} \)

```c
lock(m)
if (poll_open) {
    count++;
    unlock(m);
}
else {
    unlock(m);
    return false;
}
```

```c
lock(m)
if (poll_open) {
    poll_open = false
}
unlock(m)
return true;
```
Finding occupant workgroups

- Executed by 1 thread per workgroup

- Three global variables:
  - Mutex: \texttt{m}
  - Bool poll flag: \texttt{poll\_open}
  - Integer counter: \texttt{count}

```
lock(m)
if (poll_open) {
    count++;
    unlock(m);
}
else {
    unlock(m);
    return false;
}
```

```
lock(m)
if (poll_open) {
    poll_open = false
}
unlock(m)
return true;
```
Finding occupant workgroups

• Executed by 1 thread per workgroup

• Three global variables:
  • Mutex: m
  • Bool poll flag: poll_open
  • Integer counter: count

```plaintext
lock(m)
if (poll_open) {
    count++;
    unlock(m);
}
else {
    unlock(m);
    return false;
}
```

```plaintext
lock(m)
if (poll_open) {
    poll_open = false
}
unlock(m)
return true;
```
Finding occupant workgroups

• Executed by 1 thread per workgroup

• Three global variables:
  
  • Mutex: m
  
  • Bool poll flag: poll_open
  
  • Integer counter: count

```c
lock(m)
if (poll_open) {
    count++;
    unlock(m);
}
else {
    unlock(m);
    return false;
}

lock(m)
if (poll_open) {
    poll_open = false
}
unlock(m)
return true;
```
Let’s implement the discovery protocol
Let’s implement a portable barrier using the discovery protocol
Further reading

Our proposal for a portable inter-workgroup barrier:

• Tyler Sorensen, Alastair F. Donaldson, Mark Batty, Ganesh Gopalakrishnan, Zvonimir Rakamaric: Portable inter-workgroup barrier synchronisation for GPUs. OOPSLA 2016: 39-58

Our proposal for enabling fair scheduling on GPUs:

• Tyler Sorensen, Hugues Evrard, Alastair F. Donaldson: Cooperative kernels: GPU multitasking for blocking algorithms. ESEC/SIGSOFT FSE 2017: 431-441
Summary

• OpenCL provides low-level control over GPU architectural features
• Traditional hierarchical execution model uses barriers to synchronize inside workgroups, with no communication between workgroups
• OpenCL 2.0 provides atomic operations and memory model to facilitate inter workgroup communication
• But the OpenCL execution model provides few guarantees – makes it hard to build reliable concurrency primitives such as barriers
Current research directions

• Theoretical study of execution model hierarchy
• Empirical study of execution model characteristics provided by current GPUs
• Cooperative kernels for GPU multi-tasking (presented at ESEC/FSE tomorrow)