Lecture 7: GPUs: From Program to Hardware

G63.2011.002/G22.2945.001 · October 19, 2010
Outline

Execution: From Grid to Core

Dealing with Space: GPU Memory

Dealing with Time: Synchronization
Outline

Execution: From Grid to Core

Dealing with Space: GPU Memory

Dealing with Time: Synchronization
Connection: Hardware ↔ Programming Model

- Fetch/Decode
- 32 kiB Ctx Private ("Registers")
- 16 kiB Ctx Shared

Axis 0
Axis 1
Hardware

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

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

GPU Ideas:

1. Slim down each core → more of them
2. Less Decode, more ALU
3. Waiting for memory? Just work on something else. → extra context storage
Connection: Hardware ↔ Programming Model

Fetch/Decode

32 kiB Ctx Private ("Registers")

16 kiB Ctx Shared

Axis 0

Axis 1

Hardware

Really: Group provides pool of parallelism to draw from.

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

Fetch/Decode

32 kiB Ctx Private ("Registers")

16 kiB Ctx Shared

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

- **Fetch/Decode**
- **32 kiB Ctx Private** (“Registers”)
- **16 kiB Ctx Shared**

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Who cares how many cores?

Execution: From Grid to Core  Memory  Synchronization
Who cares how many cores?

Idea:

- Program as if there were “infinitely” many cores
- Program as if there were “infinitely” many ALUs per core

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
or
• Sequential program → parallel hardware?
Execution: From Grid to Core Memory Synchronization
Connection: Hardware ↔ Programming Model

- Axis 0
- Axis 1

Execution: From Grid to Core Memory Synchronization
Connection: Hardware ↔ Programming Model

Axis 0

Axis 1

Hardware

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Software representation

Hardware

Axis 0

Axis 1

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Grid (Kernel: Function on Grid)

Axis 0 →

Axis 1 ↓

Software representation

Hardware

Execution: From Grid to Core

Memory

Synchronization
Connection: Hardware ↔ Programming Model

Execution: From Grid to Core
Connection: Hardware ↔ Programming Model

Grid (Kernel: Function on Grid)

Axis 0

Axis 1

(Work) Group

(Work) Item

Software representation

Hardware

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Software representation

Hardware

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Software representation

Hardware

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Software representation

Hardware

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Axis 0

Axis 1

Software representation

Hardware

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Axis 0

Axis 1

Software representation

Hardware

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Axis 0

Axis 1

Software representation

Hardware

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Axis 0

Axis 1

Software representation

Hardware

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Software representation

Hardware

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Axis 0

Axis 1

Software representation

Hardware

Execution: From Grid to Core  Memory  Synchronization
Really: Group provides pool of parallelism to draw from.

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

Software representation

Hardware

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Axis 0

Axis 1

Software representation

Hardware

Execution: From Grid to Core  Memory  Synchronization
Connection: Hardware ↔ Programming Model

Software:

- `get_local_id(axis)`?/`size(axis)`?
- `get_group_id(axis)`?/`num_groups(axis)`?
- `get_global_id(axis)`?/`size(axis)`?

axis = 0, 1, 2, ...

Execution: From Grid to Core  Memory  Synchronization
Grids can be 1,2,3-dimensional.

Software:

- `get_local_id(axis)`?/`size(axis)`?
- `get_group_id(axis)`?/`num_groups(axis)`?
- `get_global_id(axis)`?/`size(axis)`?

Axis 0, 1, 2, ...

Execution: From Grid to Core Memory Synchronization
#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

#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;
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;
}
```

cl-helper.h:

```c
void print_platforms_devices();
```

prints:

plat 0: vendor 'NVIDIA Corporation'
   dev 0 'GeForce GTX 285'
plat 1: vendor 'Advanced Micro Devices, Inc.'
   dev 0 'Intel (R) Xeon(R) CPU E5405 @ 2.00GHz'

Choosing a device:

```c
void create_context_on(
    const char *plat_name, const char*dev_name, cl_uint idx ,
    cl_context *ctx, cl_command_queue *queue, int enable_profiling);
```
Dive into OpenCL: Memory

// 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));

Execution: From Grid to Core  Memory  Synchronization
More cl-helper.h:

```c
#define CHECK_CL_ERROR(STATUS_CODE, WHAT) \ 
   if ((STATUS_CODE) != CL_SUCCESS) \ 
   { \ 
      printf ( stderr, \
         "*** '%s' in '%s' on line %d failed with error '%s'.\n", \ 
            WHAT, __FILE__, __LINE__, \ 
            cl_error_to_str (STATUS_CODE)); \ 
      abort (); \ 
   } 

#define CALL_CL_GUARDEDED(NAME, ARGLIST) \ 
   { \ 
      cl_int status_code; \ 
      status_code = NAME ARGLIST; \ 
      CHECK_CL_ERROR(status_code, #NAME); \ 
   }
```

Execution: From Grid to Core  Memory  Synchronization
// 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; }

Execution: From Grid to Core Memory Synchronization
Common OpenCL calling sequences

Create

```c
cl_int status;
cl_something obj = clCreateSomething (... , &status)
CHECK_CL_ERROR(status, "create something")
```

Do

```c
cl_int status = clDoSomething(ctx, ...);
->
CALL_CL_GUARDED(clDoSomething, (ctx, ...));
```
Common OpenCL calling sequences

**Create**

```c
cl_int status;
cl_something obj = clCreateSomething(..., &status);
CHECK_CL_ERROR(status, "create something")
```

**Do**

```c
cl_int status = clDoSomething(ctx, ...);
->
CALL_CL_GUARDED(clDoSomething, (ctx, ...));
```

**Query size:**

```c
size_t size_in_bytes;
CALL_CL_GUARDED(clGetDeviceInfo, (dev, CL_DEVICE_NAME, 0, NULL, &size_in_bytes);
```

```c
size_t size_in_bytes;
CALL_CL_GUARDED(clGetDeviceInfo, (dev, CL_DEVICE_NAME, 0, NULL, &size_in_bytes);
```
HW3 Part 1 Demo
Outline

Execution: From Grid to Core

Dealing with Space: GPU Memory

Dealing with Time: Synchronization
Example: Matrix Transpose
Transpose? Simple Enough!

```c
__kernel
void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
{
    int read_idx = get_global_id(0) + get_global_id(1) * a_width;
    int write_idx = get_global_id(1) + get_global_id(0) * a_height;

    a_t[write_idx] = a[read_idx];
}
```
Measuring Performance

Writing high-performance Codes

Mindset: What is going to be the limiting factor?

- Floating point throughput?
- Memory bandwidth?
  - Cache sizes?

Benchmark the assumed limiting factor right away.

Evaluate

- Know your peak throughputs (roughly)
- Are you getting close?
- Are you tracking the right limiting factor?

Execution: From Grid to Core Memory Synchronization
Measuring Performance

Writing high-performance Codes

Mindset: What is going to be the limiting factor?

- Floating point throughput?
- Memory bandwidth?
  - Cache sizes?

Benchmark the assumed limiting factor right away.
Measuring Performance

Writing high-performance Codes
Mindset: What is going to be the limiting factor?

- Floating point throughput?
- Memory bandwidth?
  - Cache sizes?

Benchmark the assumed limiting factor right away.

Evaluate

- Know your peak throughputs (roughly)
- Are you getting close?
- Are you tracking the right limiting factor?
Performance: Matrix transpose

Very likely: Bound by memory bandwidth.
Performance: Matrix transpose

Very likely: Bound by memory bandwidth.

**Fantastic!** Far slower than CPU. Why?
Intra-device Work Distribution

Why was that so slow?
Why was that so slow?

- Launch with $1 \times 1$ workgroups
Intra-device Work Distribution

Why was that so slow?

❌ Launch with $1 \times 1$ workgroups

✅ Launch with $16 \times 16$ workgroups
Intra-device Work Distribution

Why was that so slow?

✘ Launch with $1 \times 1$ workgroups

✔ Launch with $16 \times 16$ workgroups

Again: Work Groups

- Work group size matters. A lot.
- Determines work distribution among processors
- Optimal size? Up to experimentation
Intra-device Work Distribution

Why was that so slow?

✗ Launch with $1 \times 1$ workgroups

✓ Launch with $16 \times 16$ workgroups

Again: Work Groups

- Work group size matters. A lot.
- Determines work distribution among processors
- Optimal size? Up to experimentation

Actually: Bound by control overhead (group launches)
Better. 1.5× faster than CPU—not great. Why?
How does computer memory work?

One (reading) memory transaction (simplified):

Processor

<table>
<thead>
<tr>
<th>D0..15</th>
</tr>
</thead>
<tbody>
<tr>
<td>A0..15</td>
</tr>
<tr>
<td>R/\W</td>
</tr>
<tr>
<td>CLK</td>
</tr>
</tbody>
</table>

Memory
How does computer memory work?

One (reading) memory transaction (simplified):

![Diagram of a computer memory transaction]

Observation: Access (and addressing) happens in bus-width-size "chunks".

Execution: From Grid to Core Memory Synchronization
How does computer memory work?

One (reading) memory transaction (simplified):

- Processor
- Memory

- D0..15
- A0..15
- R/\textit{W}
- CLK

Observation: Access (and addressing) happens in bus-width-size “chunks”.

Execution: From Grid to Core Memory Synchronization
How does computer memory work?

One (reading) memory transaction (simplified):

Processor

Memory

CLK

R/\bar{W}

A0..15

D0..15

Observation: Access (and addressing) happens in bus-width-size “chunks.”
How does computer memory work?

One (reading) memory transaction (simplified):
How does computer memory work?

One (reading) memory transaction (simplified):
How does computer memory work?

One (reading) memory transaction (simplified):

Observation: Access (and addressing) happens in bus-width-size “chunks”.
Problem
Memory chips have only one data bus.
So how can multiple threads read multiple data items from memory simultaneously?
Memory for Parallel Machines

Problem
Memory chips have only one data bus.
So how can multiple threads read multiple data items from memory simultaneously?

Solutions: Parallel Access to Memory

- Split a really wide data bus, but have only one address bus ("Global" memory: off-chip)
- Have many “small memories” ("banks") with separate data and address busses, select by address LSB. ("Local" memory: on-chip)
Naive: Using Global Memory

```c
__kernel
void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
{
    int read_idx = get_global_id (0) + get_global_id (1) * a_width;
    int write_idx = get_global_id (1) + get_global_id (0) * a_height;

    a_t[ write_idx ] = a[read_idx];
}
```
Naive: Using Global Memory

```c
__kernel
void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
{
    int read_idx = get_global_id(0) + get_global_id(1) * a_width;
    int write_idx = get_global_id(1) + get_global_id(0) * a_height;

    a_t[write_idx] = a[read_idx];
}
```

Reading from global mem:

```
    . . .
```

stride: 1

Execution: From Grid to Core  Memory  Synchronization
Naive: Using Global Memory

```c
__kernel
void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
{
    int read_idx = get_global_id(0) + get_global_id(1) * a_width;
    int write_idx = get_global_id(1) + get_global_id(0) * a_height;

    a_t[write_idx] = a[read_idx];
}
```

Reading from global mem:

```
|   |   |   | . . |   |   |
```

~

stride: 1 → one mem.trans.
Naive: Using Global Memory

```c
__kernel
void transpose(
   __global float *a_t, __global float *a,
   unsigned a_width, unsigned a_height)
{
    int read_idx = get_global_id(0) + get_global_id(1) * a_width;
    int write_idx = get_global_id(1) + get_global_id(0) * a_height;

    a_t[write_idx] = a[read_idx];
}
```

Reading from global mem: 

```
. . .
```

stride: 1 → one mem.trans.

Writing to global mem: 

```
. . .
```

stride: 16
Naive: Using Global Memory

```c
__kernel
void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
{
    int read_idx = get_global_id(0) + get_global_id(1) * a_width;
    int write_idx = get_global_id(1) + get_global_id(0) * a_height;

    a_t[write_idx] = a[read_idx];
}
```

Reading from global mem:

```
[ ] [ ] [ ] [ ] [ ] [ ] ...
```

stride: 1 → one mem.trans.

Writing to global mem:

```
[ ] [ ] [ ] [ ] [ ] [ ] ...
```

stride: 16 → 16 mem.trans.!
Transpose: Idea

- Global memory dislikes non-unit strides.
- Local memory doesn’t mind.
Transpose: Idea

- Global memory dislikes non-unit strides.
- Local memory doesn’t mind.

Idea

- Don’t transpose element-by-element.
- Transpose block-by-block instead.
Transpose: Idea

- Global memory dislikes non-unit strides.
- Local memory doesn’t mind.

Idea

- Don’t transpose element-by-element.
- Transpose block-by-block instead.

1. Read untransposed block from global and write to local
2. Read block transposed from local and write to global
Execution: From Grid to Core Memory Synchronization
Part 1/3:

```c
#define BLOCK_SIZE 16
#define A_BLOCK_STRIDE (BLOCK_SIZE * a_width)
#define A_T_BLOCK_STRIDE (BLOCK_SIZE * a_height)

__kernel void transpose(
    __global float *a_t, __global float *a,
    unsigned a_width, unsigned a_height)
```
Part 2/3:

```c
{  
  __local float a_local [BLOCK_SIZE][BLOCK_SIZE];
  int  base_idx_a =
     get_group_id (0) * BLOCK_SIZE +
     get_group_id (1) * A_BLOCK_STRIDE;
  int  base_idx_a_t =
     get_group_id (1) * BLOCK_SIZE +
     get_group_id (0) * A_T_BLOCK_STRIDE;

  int  glob_idx_a =
       base_idx_a + get_local_id (0)
       + a_width * get_local_id (1);
  int  glob_idx_a_t =
       base_idx_a_t + get_local_id (0)
       + a_height * get_local_id (1);
}
```
Improved: With Local Memory

Part 3/3:

```c
a_local [get_local_id (1)][get_local_id (0)] = a[glob_idx_a];
barrier (CLK_LOCAL_MEM_FENCE);

a_t[glob_idx_a_t] = a_local[get_local_id (0)][get_local_id (1)];
```
**Performance: Matrix transpose**

![Graph showing memory bandwidth vs. matrix width/height for different implementations: Silly, Naive, WithLocal.](image)

*Much better.* Not peak, but pretty good.
CL Memory Model: Summary

- **Private Memory**
  - Per work-item
- **Local Memory**
  - Shared within a workgroup
- **Global/Constant Memory**
  - Visible to all workgroups
- **Host Memory**
  - On the CPU

Memory management is Explicit
You must move data from host -> global -> local ... and back

Credit: Khronos Group
Outline

Execution: From Grid to Core

Dealing with Space: GPU Memory

Dealing with Time: Synchronization
Synchronization

What is a Barrier?
Synchronization

What is a Barrier?

---
Synchronization

What is a Barrier?
What is a Barrier?
What is a Barrier?
What is a Barrier?
Synchronization

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

write 18

17
What is a Memory Fence?
Synchronization

What is a Memory Fence?

write 18

read 17

17
What is a Memory Fence?

write 18

17
What is a Memory Fence?

```
write 18
```

18
What is a Memory Fence?
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.

write 18

17
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.
Intra-group synchronization

Spelling in device language:

```c
flags =
    CLK_LOCAL_MEMORY_FENCE
    CLK_GLOBAL_MEMORY_FENCE

barrier ( flags );
mem_fence(flags);
read_mem_fence(flags);
write_mem_fence(flags);
```

⚠️ Only within one workgroup!
OpenCL: Command Queues

- Host and Device run asynchronously
- Host submits to queue:
  - Computations
  - Memory Transfers
  - Sync primitives
  - ...
- Host can wait for drained queue
- Multiple Queues:
  Can overlap
  Compute + Transfer
Recap: Concurrency and Synchronization

GPUs have layers of concurrency.

Each layer has its synchronization primitives.
Recap: Concurrency and Synchronization

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

- **Intra-block:**
  - `barrier(...)`, `mem_fence(...)`
  
- **Inter-block:**
  - Kernel launch

- **CPU-GPU:**
  - Command queues, Events
Image Credits

- sxc.hu/cema
- Onions: flickr.com/darwinbell [cc]