

OpenCL exercise 4: Matrix Multiplication

Kaicong Sun

Local Memory

- ▶ Local memory: Shared by all work items of one work group
 - ▶ Two work items in the same work group will see the same data
 - ▶ Two work items in the different work groups will see different data
- ▶ Size: 16kB or 48kB
- ▶ Significantly higher memory bandwidth compared to global memory
- ▶ Significantly lower latency compared to global memory
 - ▶ Random accesses patterns are fast
- ▶ Used for:
 - ▶ Manually caching data from global memory
 - ▶ Storage for data being worked on
- ▶ In Cuda: “Shared memory”
 - ▶ Has nothing to do with Cuda “Local memory”

Local Memory: Syntax

- ▶ Declare “i” as 32-bit integer in local memory

```
__local int i;
```

- ▶ Declare “a” as 2D array of unsigned 32-bit integers with a size of 10x15

```
__local uint a[10][15];
```

- ▶ “10” and “15” must be compile-time constants
- ▶ Wait until all threads have reached this point and prevent any local memory accesses from being moved accros this line

```
barrier(CLK_LOCAL_MEM_FENCE);
```

Local Memory: Syntax

Dynamically sized local memory area:

- ▶ Kernel definition:

```
__kernel void kernel1(__local float* localMem) { ... }
```

- ▶ Use inside kernel

```
localMem[i] = 10;  
foo = localMem[j];
```

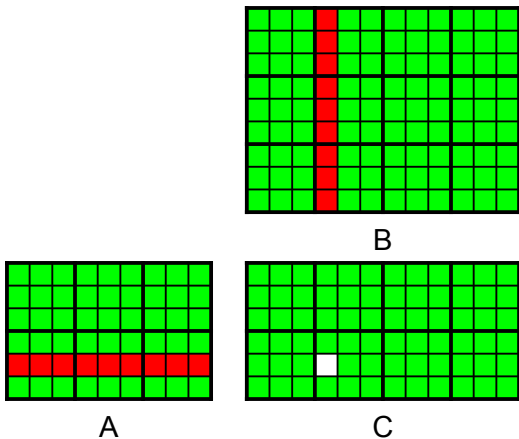
- ▶ Calling the kernel

```
kernel1.setArg(0, cl::Local(wgX * wgY * sizeof(float)));  
queue.enqueueNDRangeKernel(kernel1, ...);
```

- ▶ `localMem[]` will contain space for `wgX * wgY` floats

Matrix Multiplication

- ▶ 3 matrices: **A**, **B**, **C**, calculate $C = AB$
- ▶ $C_{i,j} = \sum_{k=1}^n A_{i,k} B_{k,j}$

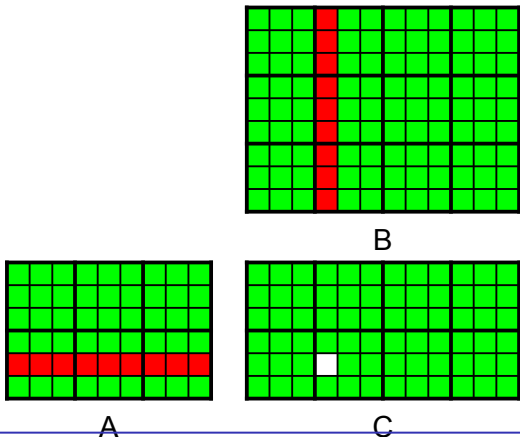


Host code

```
1  for (std::size_t j = 0; j < countAY; j++) {
2      for (std::size_t i = 0; i < countBX; i++) {
3          float sum = 0;
4          for (std::size_t k = 0; k < countAX_BY; k++) {
5              float a = h_inputA[k + j * countAX_BY];
6              float b = h_inputB[i + k * countBX];
7              sum += a * b;
8          }
9          h_outputC[i + j * countBX] = sum;
10     }
11 }
```

Task 1: Simple GPU implementation

Implement host code on GPU, use one work item for each element in C (the result matrix).

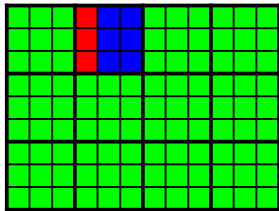


Task 2: Local memory

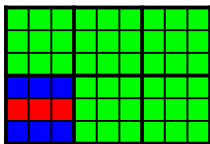
Take advantage of local memory to speed up the calculation

- ▶ Input Matrices are split into quadratic blocks, size of blocks = size of work group
- ▶ Each work group will:
 - ▶ For each input block needed:
 - ▶ Load the block from A into local memory
 - ▶ Load the block from B into local memory
 - ▶ Barrier
 - ▶ Use the data from local memory to calculate the result
 - ▶ Barrier
 - ▶ Store the result

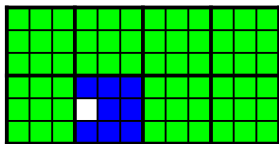
Step 1



B

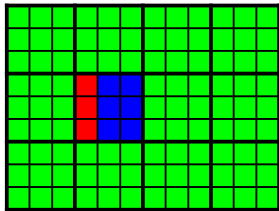


A

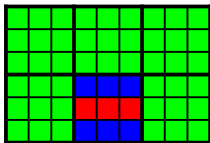


C

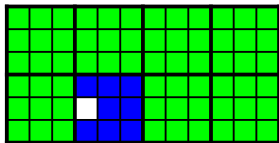
Step 2



B

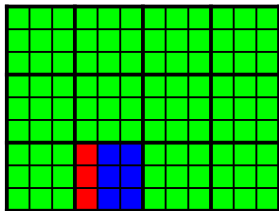


A

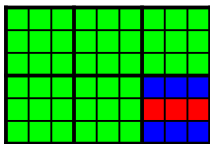


C

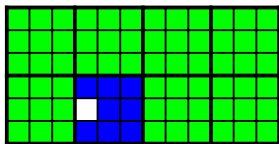
Step 3



B



A



C

Pseudocode

```
1  for (uint j = 0; j < countAY; j++) {
2      for (uint i = 0; i < countBX; i++) {
3          float sum = 0;
4          int k = get_local_id(0);
5          int g = get_local_id(1);
6          __local float l_A[WG_SIZE][WG_SIZE];
7          __local float l_B[WG_SIZE][WG_SIZE];
8          // loop over the submatrices
9          for (uint bs = 0; bs < countAX_BY; bs += WG_SIZE) {
10             //Copy blocks of d_inputA, d_inputB to local memory
11             l_A[...][...] = d_inputA[(k+bs) + j * countAX_BY];
12             l_B[...][...] = d_inputB[i + (g+bs) * countBX];
13
14             barrier(CLK_LOCAL_MEM_FENCE);
15             for (uint k = 0; k < WG_SIZE; k++)
16                 sum += l_A[...][...] * l_B[...][...];
17             barrier(CLK_LOCAL_MEM_FENCE);
18         }
19         d_outputC[i + j * countBX] = sum;
20     }
21 }
```

Hints

- ▶ For copying a block of data (with size of block = size of work group) into local memory, each work item has to copy one value
 - ▶ Two consecutive work items in X-direction should read two consecutive values from global memory (for performance reasons, allows global memory accesses to be “coalesced”)
 - ▶ `localMem[get_local_id(1)][get_local_id(0)] =
globalMem[x + width * y];`
- ▶ To make sure a kernel is only called with a certain work group size:

```
__attribute__((reqd_work_group_size(10, 20, 1)))  
__kernel void kernel1() { ... }
```

Kernel can only be called with a 2D work group with 10x20 elements.

Task 3/4

Two (optional) tasks:

- ▶ Task 3: Make a copy of the kernel of Task 2 and modify it so that it doesn't need any compile-time knowledge of the work group size (i.e. `WG_SIZE`)
 - ▶ Will need a dynamically sized local memory area
 - ▶ Will be slower than kernel 2

```
matrixMulKernel.setArg(4, cl::Local(2 * wgSize
* wgSize * sizeof(float)));
__local float* l_A = localMem;
__local float* l_B = localMem + get_local_size(0)
* get_local_size(1);
```

- ▶ Task 4: Create a kernel similar to kernel 1, but use OpenCL images for *A* and *B*
 - ▶ Will be slower than kernel 2 and kernel 3
 - ▶ Might be slower or faster than kernel 1