

Intermediate OpenCL Programming

Pangfeng Liu
National Taiwan University

May 5, 2022

Buffer

There are three exclusive types of buffers.

CL_MEM_USE_HOST_PTR

CL_MEM_ALLOC_HOST_PTR

CL_MEM_COPY_HOST_PTR

CL_MEM_USE_HOST_PTR

- OpenCL uses the host memory as the buffer so the `host_ptr` must not be `NULL`.
- If the device is GPU then the access is slow because the buffer is in the host (CPU).

CL_MEM_ALLOC_HOST_PTR

- OpenCL will allocate a buffer in the device and the device will use it directly.
- This memory is accessible from the host by `clEnqueueWriteBuffer` and `clEnqueueReadBuffer`.
- Since the device will allocate the buffer we cannot provide a non-NULL `host_ptr`.
- The access is fast because the buffer is in the device

CL_MEM_COPY_HOST_PTR

- OpenCL will copy the contents of the host buffer to a buffer on the device
- The device will use the device buffer.
- The access is fast because the buffer is in the device (GPU/CPU).

Comparison

buffer type	host_ptr	device will use	speed from GPU
USE	<code>!= NULL</code>	the host buffer directly	slow
ALLOC	<code>== NULL</code>	the device buffer	fast
COPY	<code>!= NULL</code>	the device buffer copied from the host buffer	fast

Discussion

- Which of the allocation mode provides the slowest memory access from a device GPU?

Need not to Retrieve

- We would like to implement the vector addition program without getting the results back from the device.
- The idea is that we copy the vectors A and B into the device with `CL_MEM_COPY_HOST_PTR` buffer, and put the results in a `CL_MEM_USE_HOST_PTR` buffer.
- When the computation is over then, we can get the results in the host buffer directly.

Example 1: (vectorAdd-nofetchC.c)

```
70 cl_mem bufferA =  
71     clCreateBuffer(context,  
72                     CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  
73                     N * sizeof(cl_uint), A, &status);  
74 assert(status == CL_SUCCESS);  
75 cl_mem bufferB =  
76     clCreateBuffer(context,  
77                     CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  
78                     N * sizeof(cl_uint), B, &status);  
79 assert(status == CL_SUCCESS);  
80 cl_mem bufferC =  
81     clCreateBuffer(context,  
82                     CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,  
83                     N * sizeof(cl_uint), C, &status);  
84 assert(status == CL_SUCCESS);  
85 printf("Build buffers completes\n");
```

Execution

- After creating buffers, setting the parameter order, and set up the dimension of NDRange, we run the kernel by placing it into the command queue

clFinish

Example 2: (vectorAdd-nofetchC.c)

```
98  size_t globalThreads[] = {(size_t)N};
99  size_t localThreads[] = {1};
100  status =
101      clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
102                             globalThreads, localThreads,
103                             0, NULL, NULL);
104  assert(status == CL_SUCCESS);
105  printf("Kernel execution completes.\n");
106  clFinish(commandQueue);
```

Demonstration

- Run the `vector-nofetchC-cl` program.

Discussion

- Does it produce the correct answer?

Non-blocking

- The previous program does not produce the correct answer because the function call `clEnqueueNDRangeKernel` is *non-blocking*, which means it will not wait for the completion of the kernel.
- Since we check the contents of C before the kernel finishes, the answer is incorrect.
- We need to wait for the commands in the command queue to finish.

Prototype 3: clFinish.h

```
1 cl_int clFinish(cl_command_queue command_queue);
```

Parameters

`command_queue` Wait for the commands in this command queue to finish.

clFinish

Example 4: (vectorAdd-nofetchC-finish.c)

```
98  size_t globalThreads[] = {(size_t)N};
99  size_t localThreads[] = {1};
100  status =
101      clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
102                              globalThreads, localThreads,
103                              0, NULL, NULL);
104  assert(status == CL_SUCCESS);
105  printf("Kernel execution completes.\n");
106  /* getcvector */
107  clFinish(commandQueue);
```

Demonstration

- Run the `vector-nofetchC-finish-cl` program.

Discussion

- Does it produce the correct answer?

Host Buffer?

- In this NVIDIA implementation it seems that the buffer C is not on host.
- We need to copy C back to the host.

Read the buffer C back

Example 5: (vectorAdd-fetchC.c)

```
98  size_t globalThreads[] = {(size_t)N};
99  size_t localThreads[] = {1};
100  status =
101      clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
102                              globalThreads, localThreads,
103                              0, NULL, NULL);
104  assert(status == CL_SUCCESS);
105  printf("Kernel execution completes.\n");
106  #ifdef USEclFINISH
107      clFinish(commandQueue);
108  #else
109      clEnqueueReadBuffer(commandQueue, bufferC, CL_TRUE,
110                          0, N * sizeof(cl_uint), C,
111                          0, NULL, NULL);
112  #endif
```

Demonstration

- Run the `vector-fetchC-cl` program.

Discussion

- Does it produce the correct answer?

NDRange

- We can set the dimension of NDRange in the `clEnqueueNDRangeKernel` call.
- In the previous example we set the dimension to one, now we want to set it to two.
- Now a work item in NDRange will have two indices for two dimensions.

Index

- We can call `get_global_id(i)` to know its index in the i -th dimension.
- The parameter `i` must be within the dimension of `NDRange`

Prototype 6: getGlobalId.h

```
1  size_t get_global_id(uint dimindx);
```

Parameters

`dimindx` The dimension in which we want to know the index.

Domain

- We will use an array to keep track of the global indices a kernel function can see from a work item.
- We declare this array as `int globalId[2][N][N]`, where N is 16.
- The first 2 is for two dimensions.

Kernel

Example 7: (get-global-id.cl)

```
1 #define N 16
2
3 __kernel void getGlobalId(__global int globalId[2][N][N])
4 {
5     int id0 = get_global_id(0);
6     int id1 = get_global_id(1);
7     globalId[0][id0][id1] = id0;
8     globalId[1][id0][id1] = id1;
9 }
```

Kernel

- We first call `get_global_id(0)` and `get_global_id(1)` to know the indices of this work item on the two dimensions, and put them into `id0` and `id1`.
- Then we place `id0` and `id1` into corresponding cells of `globalId`.

Discussion

- Google `__global` to and find out what it means.

Buffer

- In the main program we declare a host buffer `bufferGlobalId` to hold the global indices.
- This buffer will link to the `globalId[2][N][N]` parameter in the kernel function.

Buffers

Example 8: (get-global-id.c)

```
73  cl_mem bufferGlobalId =  
74      clCreateBuffer(context,  
75                      CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,  
76                      2 * N * N * sizeof(cl_uint), globalId, &status)  
77  assert(status == CL_SUCCESS);  
78  printf("Build buffers completes\n");  
79  /* setarg */  
80  status = clSetKernelArg(kernel, 0, sizeof(cl_mem),  
81                          (void*)&bufferGlobalId);  
82  assert(status == CL_SUCCESS);  
83  printf("Set kernel arguments completes\n");
```

Dimension

- Now we set the dimension of `NDRange` to 2, and set the size of each dimension to `N`, as in `globalDim`.
- Since `NDRange` has two dimensions, a work group will also have two dimension. For simplicity we set it to one by one, as in `localDim`.
- We place the kernel into the command queue, then call `clFinish` to wait for the completion.

Buffers

Example 9: (get-global-id.c)

```
85  size_t globalDim[] = {(size_t)N, (size_t)N};
86  size_t localDim[] = {1, 1};
87  status =
88      clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL,
89                              globalDim, localDim,
90                              0, NULL, NULL);
91  assert(status == CL_SUCCESS);
92  printf("Specify the shape of the domain completes.\n");
93  /* getresult */
94  #ifdef USEclFINSIH
95      clFinish(commandQueue);
96  #else
97      clEnqueueReadBuffer(commandQueue, bufferGlobalId, CL_TRUE,
98                          0, 2 * N * N * sizeof(cl_uint), globalId,
99                          0, NULL, NULL);
100 #endif
101  printf("Kernel execution completes.\n");
102
103  printId("get_global_id(0)", globalId[0]);
```

printId

- We implement a `printId` function to print the contents in any array.
- The first parameter is a string for identification purpose, and the second parameter is the array of indices to print.

Buffers

Example 10: (get-global-id.c)

```
11 void printId(char *title, cl_uint id[N][N])
12 {
13     puts(title);
14     for (int i = 0; i < N; i++) {
15         for (int j = 0; j < N; j++)
16             printf("%2d ", id[i][j]);
17         printf("\n");
18     }
19 }
```

Demonstration

- Run the `get-global-id-cl` program.

Discussion

- Describe the printed indices.
- Will the output be affected if we set the work group size differently?

NDRange

- Local index is the index within a work group.
- We can specify the size of each dimension for a work group in a `clEnqueueNDRangeKernel` call.
- In this example we will set the size of local dimension and observe the results.

Local Index

- We call `get_local_id(i)` to know its index in the i -th dimension.
- The parameter `i` must be within the dimension of `NDRange` because the local and the global index have the same number of dimensions.

Prototype 11: getLocalId.h

```
1 size_t get_local_id(uint dimindx);
```

Parameters

`dimindx` The dimension in which we want to know the index.

Domain

- We will use two arrays to keep track of global and local index a kernel function can see from a work item respectively.
- We declare two arrays as `int globalId[2][N][N]` and `int localId[2][N][N]`.

Kernel

Example 12: (get-global-local-id.cl)

```
1  #define N 16
2
3  __kernel void getGlobalId(__global int globalId[2][N][N],
4  __global int localId[2][N][N])
5  {
6      int id0 = get_global_id(0);
7      int id1 = get_global_id(1);
8      globalId[0][id0][id1] = get_global_id(0);
9      globalId[1][id0][id1] = get_global_id(1);
10     localId[0][id0][id1] = get_local_id(0);
11     localId[1][id0][id1] = get_local_id(1);
12 }
```

Kernel

- We first call `get_global_id(0)` and `get_global_id(1)` to know the indices of this work item in `NDRange`, and put them into `id0` and `id1`.
- Then we call `get_global_id` and `get_local_id` to get the indices.

Buffers

- In the main program we declare two host buffers `bufferGlobalId` and `bufferLocalId` to hold the global and local indices.
- This buffer will link to parameters `globalId[2][N][N]` and `localId[2][N][N]` in the kernel function.

Buffers

Example 13: (get-global-local-id.c)

```
74  cl_mem bufferGlobalId =
75      clCreateBuffer(context,
76                      CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
77                      2 * N * N * sizeof(cl_uint), globalId, &status)
78  assert(status == CL_SUCCESS);
79  cl_mem bufferLocalId =
80      clCreateBuffer(context,
81                      CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
82                      2 * N * N * sizeof(cl_uint), localId, &status);
83  assert(status == CL_SUCCESS);
84  printf("Build buffers completes\n");
85  /* setarg */
86  status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
87                          (void*)&bufferGlobalId);
88  assert(status == CL_SUCCESS);
89  status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
90                          (void*)&bufferLocalId);
91  assert(status == CL_SUCCESS);
92  printf("Set kernel arguments completes\n");
```


Dimension

- Now we set the dimension of `NDRange` to 2, and set the size of each dimension to `N`, as in `globalDim`.
- We set the size of dimension of a work group from the command line arguments, and place the sizes into `localDim`.
- We place the kernel into the command queue, then call `clFinish` to wait for the completion.

Buffers

Example 14: (get-global-local-id.c)

```
94  size_t globalDim[] = {(size_t)N, (size_t)N};
95  int groupRow = atoi(argv[2]);
96  int groupCol = atoi(argv[3]);
97  size_t localDim[] = {groupRow, groupCol};
98  status =
99      clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL,
100                             globalDim, localDim,
101                             0, NULL, NULL);
102  assert(status == CL_SUCCESS);
103  printf("Specify the shape of the domain completes.\n");
104  /* getresult */
105  #ifdef USEclFINSIH
106      clFinish(commandQueue);
107  #else
108      clEnqueueReadBuffer(commandQueue, bufferGlobalId, CL_TRUE,
109                          0, 2 * N * N * sizeof(cl_uint), globalId,
110                          0, NULL, NULL);
111      clEnqueueReadBuffer(commandQueue, bufferLocalId, CL_TRUE,
112                          0, 2 * N * N * sizeof(cl_uint), localId,
```

Demonstration

- Run the `get-global-id-cl` program by setting the work group size to be 4 by 4.
- Run the `get-global-id-cl` program by setting the work group size to be 2 by 4.

Discussion

- Describe the printed indices under different group sizes.

Matrix Multiplication

- We now use matrix multiplication as an example of using global index.
- We will multiply two N by N matrices using GPU.

Kernel

- The multiplication kernel has three parameters – A, B and C.
- We will multiply A with B, and place the results in C.
- We first get the row and column number of this work item, then perform an inner product.

Kernel

Example 15: (mul-kernel.cl)

```
1  #define N 1024
2
3  __kernel void mul(__global int matrixA[N][N],
4                    __global int matrixB[N][N],
5                    __global int matrixC[N][N])
6  {
7      int row = get_global_id(0);
8      int col = get_global_id(1);
9      int sum = 0;
10     for (int i = 0; i < N; i++)
11         sum += matrixA[row][i] * matrixB[i][col];
12     matrixC[row][col] = sum;
13 }
```

Matrices

- The main program first prepares host memory matrices A and B.

Buffers

Example 16: (matrixMul.c)

```
61  for (int i = 0; i < N; i++)  
62      for (int j = 0; j < N; j++) {  
63          A[i][j] = i + j;  
64          B[i][j] = i - j;  
65      }
```

Buffers

- The main program then creates OpenCL buffers for A, B and C.
- The contents of A and B will be copied into devices (`CL_MEM_COPY_HOST_PTR`), and the GPU will use host memory buffer directly (`CL_MEM_USE_HOST_PTR`).

Buffers

Example 17: (matrixMul.c)

```
67 cl_mem bufferA =  
68     clCreateBuffer(context,  
69                     CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  
70                     N * N * sizeof(cl_uint), A, &status);  
71 assert(status == CL_SUCCESS);  
72 cl_mem bufferB =  
73     clCreateBuffer(context,  
74                     CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  
75                     N * N * sizeof(cl_uint), B, &status);  
76 assert(status == CL_SUCCESS);  
77 cl_mem bufferC =  
78     clCreateBuffer(context,  
79                     CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,  
80                     N * N * sizeof(cl_uint), C, &status);  
81 assert(status == CL_SUCCESS);  
82 printf("Build buffers completes\n");
```

Run Kernel

- We set the size of both dimensions to N , as in `globalDim`.
- We set the sizes of dimensions of a work group as 1 by 1 and place the sizes into `localDim`.
- We place the kernel into the command queue, then call `clFinish` to wait for the completion.

Buffers

Example 18: (matrixMul.c)

```
95  size_t globalThreads[] = {(size_t)N, (size_t)N};
96  size_t localThreads[] = {1, 1};
97  status =
98      clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL,
99                              globalThreads, localThreads,
100                              0, NULL, NULL);
101  assert(status == CL_SUCCESS);
102  printf("Specify the shape of the domain completes.\n");
103  /* getcvector */
104  #ifdef USEclFINSIH
105      clFinish(commandQueue);
106  #else
107      clEnqueueReadBuffer(commandQueue, bufferC, CL_TRUE,
108                          0, N * N * sizeof(cl_uint), C,
109                          0, NULL, NULL);
110  #endif
111  printf("Kernel execution completes.\n");
```

Demonstration

- Run the `matrixMul-cl` program.

Discussion

- Did you notice the significant speed difference in the computation and verification parts?

Time Measurement

- We would like to measure the kernel execution time.
- We send the kernel to a device through a *command queue*, so we need to enable the command queue for profiling.
- We associate an event with the end of a kernel execution, then wait for the event.
- We retrieve the timing information from the event.

Steps

- We set the property of the command queue to allow profiling.
- We then declare a variable of type `cl_event` for the event.
- We then supply the event as the last parameter when calling `clEnqueueNDRangeKernel`.
- We then wait for this event by calling a function `clWaitForEvents`.
- Finally we extract the timing information from the event by calling `clGetEventProfilingInfo`.

Command Queue

- The kernel is sent to a device through a *command queue*.
- If we want to time (profile) the kernel execution, we need to explicitly set the property of the command queue to `CL_PROFILING_COMMAND_QUEUED`.

Prototype 19: clCreateCommandQueue.h

```
1 cl_command_queue  
2 clCreateCommandQueueWithProperties(  
3     cl_context context,  
4     cl_device_id device,  
5     const cl_queue_properties  
6     *properties,  
7     cl_int *errcode_ret);
```

Property

- We explicitly set the `CL_PROFILING_COMMAND_QUEUED` property of the command queue when we created it.
- Note that the API needs a list of property/value pairs, which must end in 0.
- This *list* is actually an array.

Example 20: (matrixMul-time.c)

```
35  const cl_queue_properties properties[] =  
36      {CL_QUEUE_PROPERTIES, CL_QUEUE_PROFILING_ENABLE, 0};  
37  cl_command_queue commandQueue =  
38      clCreateCommandQueueWithProperties(context, GPU[0],  
39                                          properties, &status);  
40  assert(status == CL_SUCCESS);
```

Discussion

- Google to find out other properties that can be set for a command queue.

Kernel Execution

- After setting the command queue for profiling, we can start the kernel.
- We supply event as the last parameter while calling `clEnqueueNDRangeKernel`, so now we have associated the event with the kernel execution.

Prototype 21: clEnqueueNDRangeKernel.h

```
1  cl_int  
2  clEnqueueNDRangeKernel (cl_command_queue command_queue,  
3                          cl_kernel kernel,  
4                          cl_uint work_dim,  
5                          const size_t *global_work_offset,  
6                          const size_t *global_work_size,  
7                          const size_t *local_work_size,  
8                          cl_uint num_events_in_wait_list,  
9                          const cl_event *event_wait_list,  
10                         cl_event *event);
```


Example 22: (matrixMul-time.c)

```
98  size_t globalThreads[] = {(size_t)N, (size_t)N};
99  size_t localThreads[] = {1, 1};
100  cl_event event;
101  status =
102      clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL,
103                             globalThreads, localThreads,
104                             0, NULL, &event);
105  assert(status == CL_SUCCESS);
```

Wait for Event

- Now we have submitted the kernel to execution, we just wait for the event to happen.
- We call `clWaitForEvents` to wait for event(s).

clWaitForEvents

Prototype 23: clWaitForEvents.h

```
1 cl_int clWaitForEvents(cl_uint num_events,  
2                        const cl_event *event_list);
```

Parameters

`num_events` The number of events to wait for.

`event_list` The array of events to wait for.

Example 24: (matrixMul-time.c)

```
107 status = clWaitForEvents(1, &event);  
108 assert(status == CL_SUCCESS);  
109 clEnqueueReadBuffer(commandQueue, bufferC, CL_TRUE,  
110                     0, N * N * sizeof(cl_uint), C,  
111                     0, NULL, NULL);  
112 printf("Kernel execution completes.\n");
```

Discussion

- Trace the `matrixMul-time.c` to understand the flow so far.

Three Stages

A kernel execution will go through three stages.

- First, the kernel joins the command queue for execution.
- When it is the turn of this kernel, it starts running on the device.
- When the kernel finishes execution, it will trigger the event you associated with it.

Get Information

The event can provide the following four times.

- The time when the kernel joined the command queue.
- The time when the kernel went to the device for execution.
- The time when the kernel started execution.
- The time when the kernel finished execution.

Get Information

- We call `clGetEventProfilingInfo` to know the four times.
- The time-stamp is of type `cl_ulong`.

clGetEventProfilingInfo

Prototype 25: clGetEventProfilingInfo.h

```
1  cl_int  
2  clGetEventProfilingInfo(cl_event event,  
3                          cl_profiling_info param_name,  
4                          size_t param_value_size,  
5                          void *param_value,  
6                          size_t *param_value_size_ret);
```

Parameters

`event` The event that we want to query.

`param_name` The information to query.

`param_value_size` The length of buffer (`param_value`) to store the answer.

`param_value` The buffer to store the answer.

`param_value_size_ret` The actual length of the returned answer.

Four Time Points

- We put the four times in four variables – `timeEnterQueue`, `timeSubmit`, `timeStart`, and `timeEnd`.

Time Calculation

Example 26: (matrixMul-time.c)

```
115  cl_ulong timeEnterQueue, timeSubmit, timeStart, timeEnd;
116  status =
117      clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_QUEUED,
118                              sizeof(cl_ulong), &timeEnterQueue, NULL);
119  assert(status == CL_SUCCESS);
120  status =
121      clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_SUBMIT,
122                              sizeof(cl_ulong), &timeSubmit, NULL);
123  assert(status == CL_SUCCESS);
124  status =
125      clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
126                              sizeof(cl_ulong), &timeStart, NULL);
127  assert(status == CL_SUCCESS);
128  status =
129      clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
130                              sizeof(cl_ulong), &timeEnd, NULL);
131  assert(status == CL_SUCCESS);
```

Time Calculation

- We calculate the duration of each stage by the difference of the beginning and the ending time-stamps.
- The unit of time is nano (10^{-9}) second.

Example 27: (matrixMul-time.c)

```
133 printf("kernel queued time %f seconds\n",  
134         (timeSubmit - timeEnterQueue) / 1000000000.0);  
135 printf("kernel submission time %f seconds\n",  
136         (timeStart - timeSubmit) / 1000000000.0);  
137 printf("kernel execution time %f seconds\n",  
138         (timeEnd - timeStart) / 1000000000.0);
```

Demonstration

- Run the `matrixMul-time-cl` program.

Discussion

- Observe the times of the three stages.

Buffer Comparison

- Now we know how to measure kernel execution time, we can compare the effects of different communication buffers on the execution time.
- We will compare the executive time of using `CL_MEM_COPY_HOST_PTR` and `CL_MEM_USE_HOST_PTR` for creating A and B buffers.
- We will fix the allocation method of C to make a meaningful comparison.

Difference

- The only difference between the following two programs is how they allocate A and B matrices.

Example 28: (matrixMul-time-copy.c)

```
70 cl_mem bufferA =
71     clCreateBuffer(context,
72                     CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
73                     N * N * sizeof(cl_uint), A, &status);
74 assert(status == CL_SUCCESS);
75 cl_mem bufferB =
76     clCreateBuffer(context,
77                     CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
78                     N * N * sizeof(cl_uint), B, &status);
79 assert(status == CL_SUCCESS);
80 cl_mem bufferC =
81     clCreateBuffer(context,
82                     CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
83                     N * N * sizeof(cl_uint), C, &status);
84 assert(status == CL_SUCCESS);
85 printf("Build buffers completes\n");
```

Example 29: (matrixMul-time-use.c)

```
70 cl_mem bufferA =
71     clCreateBuffer(context,
72                     CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
73                     N * N * sizeof(cl_uint), A, &status);
74 assert(status == CL_SUCCESS);
75 cl_mem bufferB =
76     clCreateBuffer(context,
77                     CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
78                     N * N * sizeof(cl_uint), B, &status);
79 assert(status == CL_SUCCESS);
80 cl_mem bufferC =
81     clCreateBuffer(context,
82                     CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
83                     N * N * sizeof(cl_uint), C, &status);
84 assert(status == CL_SUCCESS);
85 printf("Build buffers completes\n");
```

Demonstration

- Run the `matrixMul-time-copy-cl` and `matrixMul-time-use-cl` programs.

Discussion

- Compare the execution times of the two programs.
- What is the reason for this difference in kernel execution time?

Local Memory

- Local memory is shared by processing units in the same work group.
- Local memory is fast but small.
- We will use local memory to speed up matrix multiplication.

Idea

- Both matrices A and B are N by N .
- We will partition the matrices into Blk by Blk blocks, so each block has N/Blk rows and columns.
- For ease of notation we will use $Block(A, i, j)$ to denote the block in i -th row of blocks and j -th column of blocks in A .

Work Group

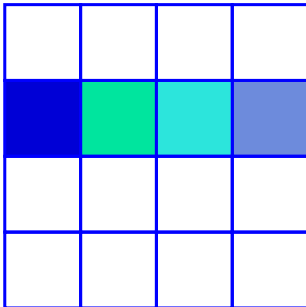
- Each work group will compute a block in C .
- Each thread will compute an element of C .

Divide and Conquer

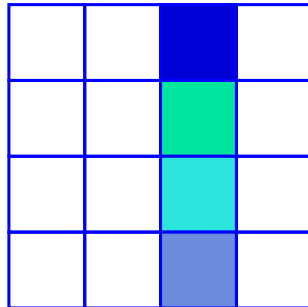
- Now consider the work group that is responsible for computing the $Block(C, i, j)$.
- This work group will first multiply $Block(A, i, 1)$ by $Block(B, 1, j)$.
- This work group will then multiply $Block(A, i, 2)$ by $Block(B, 2, j)$.
- ...
- This work group will then multiply $Block(A, i, N)$ by $Block(B, N, j)$.
- Then $Block(C, i, j)$ is the sum of all these products.

Divide and Conquer

A



B



Discussion

- Make sure that you understand this algorithm.

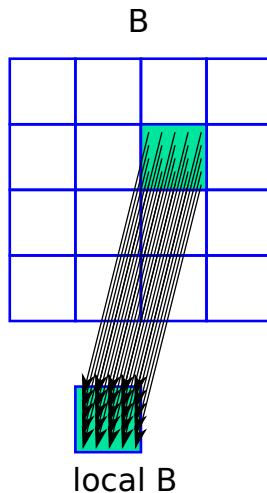
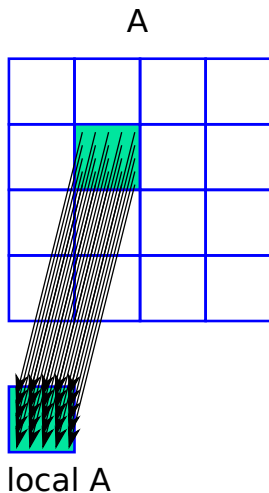
Fast multiplication

- If we know how to do matrix multiplication on two blocks, we know how to solve the whole problem.
- The problem is that if we do this on global memory, it will be slow.
- The idea is to bring $Block(A, i, 1)$ and $Block(B, 1, j)$ into local memory, then we can multiply them fast.
- How??

Memory Movement

- There are $(N/Blk)^2$ elements in $Block(A, i, 1)$.
- There are $(N/Blk)^2$ threads in this work group.
- We make each thread to move an element from both $Block(A, i, 1)$ $Block(B, 1, j)$ into local memory, then from both $Block(A, i, 2)$ and $Block(B, 2, j)$, and so on.
- After each movement each thread computes an element in $Block(C, i, j)$ using the data in local memory.

Move to Local Memory



Profitable

- Why is this profitable?
- Each thread only moves two data.
- Each thread will do a vector inner product on two vector of length (N/Blk) .
- That is, every data moved into local memory is shared by (N/Blk) other threads.

Discussion

- Make sure that you understand this profitable theory.

Steps

Now from a thread, or a kernel point of view, it will go through the following steps.

- Get the global and local indices of this work item.
- Go through Blk iterations, where each of these iterations multiplies two blocks.
 - Move a data from A in global memory into local memory.
 - Move a data from B in global memory into local memory.
 - After both steps are done compute the inner product and add it to a variable `sum`
- Put `sum` into the corresponding C element.

Constants

- Since the kernel function file cannot include other source files, we can only define these constants here. A more consistent method should be used in the future.
- We define a symbol `BSIDE` for the size of a side of the a block.

Example 30: (mul-local-kernel.cl)

```
2 #define N 1024
3 #define Blk 64
4 #define BSIDE (N / Blk)
```

Interface

- We use global memory as the interface between the kernel and the host, and the interface is the same as before.
- We declare two local matrices in local memory, using the `__local` keyword.

Example 31: (mul-local-kernel.cl)

```
6  __kernel void mul(__global int A[N][N],
7                      __global int B[N][N],
8                      __global int C[N][N])
9  {
10     int globalRow = get_global_id(0);
11     int globalCol = get_global_id(1);
12     int localRow = get_local_id(0);
13     int localCol = get_local_id(1);
14
15     __local int ALocal[BSIDE][BSIDE];
16     __local int BLocal[BSIDE][BSIDE];
```

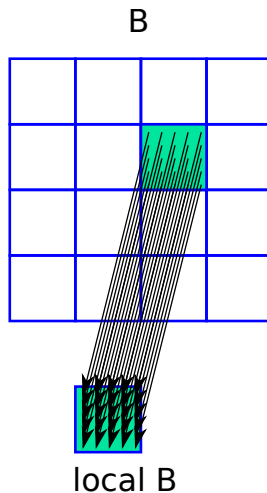
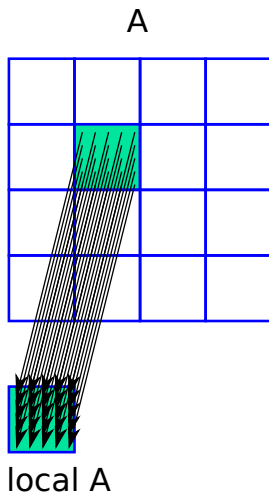
Data Movement

- Each thread (kernel function) moves one data into local A and B .
- However, we need to make sure that when we start the inner product, *all* the data are there.
- Remember that a thread only moves two data, and other threads move other data – we do not know if they have finished.
- We need to synchronize with all other threads in the workgroup.

Example 32: (mul-local-kernel.cl)

```
18  int sum = 0;
19  for (int block = 0; block < Blk; block++) {
20      ALocal[localRow][localCol] =
21          A[globalRow][block * BSIDE + localCol];
22      BLocal[localRow][localCol] =
23          B[block * BSIDE + localRow][globalCol];
24      barrier(CLK_LOCAL_MEM_FENCE);
```

Move to Local Memory



Discussion

- Do we need to synchronize with threads in other work groups?
- Convince yourself that the index calculation is correct.

Synchronization

- We use barrier to synchronize threads.
- All threads in the same group will synchronize.

barrier

Prototype 33: barrier.h

```
1 void barrier(cl_mem_fence_flags flags);
```

Parameters

flags The memory level this synchronization guarantees.

CLK_LOCAL_MEM_FENCE Guarantees that all local memory operations will finish.

CLK_GLOBAL_MEM_FENCE Guarantees that all global memory operations will finish.

Example 34: (mul-local-kernel.cl)

```
18  int sum = 0;
19  for (int block = 0; block < Blk; block++) {
20      ALocal[localRow][localCol] =
21          A[globalRow][block * BSIDE + localCol];
22      BLocal[localRow][localCol] =
23          B[block * BSIDE + localRow][globalCol];
24      barrier(CLK_LOCAL_MEM_FENCE);
25      /* inner */
26      for (int k = 0; k < BSIDE; k++)
27          sum += ALocal[localRow][k] * BLocal[k][localCol];
28      barrier(CLK_LOCAL_MEM_FENCE);
29  }
30  C[globalRow][globalCol] = sum;
31 }
```

C

- The answer in `C` is accumulated throughout the iterations in variable `sum`.
- We place another barrier after the inner product.
- We only update local memory so we use `CLK_LOCAL_MEM_FENCE`.

C

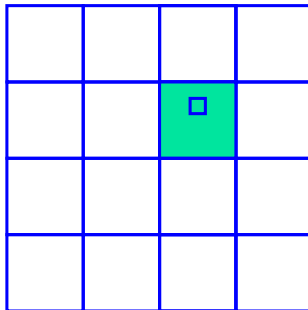


local A



local B

C



Demonstration

- Run the `matrixMul-time-copy-local-cl` program.

Discussion

- Do we need both synchronizations?
- Observe the timing.