Intermediate OpenCL Programming

Pangfeng Liu
National Taiwan University

May 25, 2020

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	!= NULL	the host buffer directly	slow
ALLOC	== NULL	the device buffer	fast
COPY	!= NULL	the device buffer copied from the host buffer	fast

Discussion

- Which of the allcoation 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 vector A and B into 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)

```
  cl_mem bufferA =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                   N * sizeof(cl_uint), A, &status);
  assert(status == CL_SUCCESS);
  cl_mem bufferB =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                   N * sizeof(cl_uint), B, &status);
  assert(status == CL_SUCCESS);
  cl_mem bufferC =
    clCreateBuffer(context,
                   CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
                   N * sizeof(cl_uint), C, &status);
  assert(status == CL_SUCCESS);
  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)

```
  size_t globalThreads[] = {(size_t)N};
  size_t localThreads[] = {1};
  status =
    clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
                           globalThreads, localThreads,
                           0, NULL, NULL);
  assert(status == CL_SUCCESS);
  printf("Kernel execution completes.\n");
```

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

```
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)

```
  size_t globalThreads[] = {(size_t)N};
  size_t localThreads[] = {1};
  status =
    clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
                           globalThreads, localThreads,
                           0, NULL, NULL);
  assert(status == CL_SUCCESS);
  printf("Kernel execution completes.\n");
  /* getcvector */
#ifdef USEclFINSIH
  clFinish(commandQueue);
#else
  clEnqueueReadBuffer(commandQueue, bufferC, CL_TRUE,
                      0, N * sizeof(cl_uint), globalId,
                      0, NULL, NULL);
#endif
```

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)

```
  size_t globalThreads[] = {(size_t)N};
  size_t localThreads[] = {1};
  status =
    clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
                           globalThreads, localThreads,
                           0, NULL, NULL);
  assert(status == CL_SUCCESS);
  printf("Kernel execution completes.\n");
#ifdef USEclFINSIH
  clFinish(commandQueue);
#else
  clEnqueueReadBuffer(commandQueue, bufferC, CL_TRUE,
                      0, N * sizeof(cl_uint), C,
                      0, NULL, NULL);
#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

```
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)

```
#define N 16

__kernel void getGlobalId(__global int globalId[2][N][N])
{
  int id0 = get_global_id(0);
  int id1 = get_global_id(1);
  globalId[0][id0][id1] = id0;
  globalId[1][id0][id1] = id1;
}
```

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)

```
cl_mem bufferGlobalId =
  clCreateBuffer(context,
                 CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
                 2 * N * N * sizeof(cl_uint), globalId, &status);
assert(status == CL_SUCCESS);
printf("Build buffers completes\n");
/* setarg */
status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
                        (void*)&bufferGlobalId);
assert(status == CL_SUCCESS);
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)

```
  size_t globalDim[] = {(size_t)N, (size_t)N};
  size_t localDim[] = {1, 1};
  status =
    clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL,
                           globalDim, localDim,
                           0, NULL, NULL);
  assert(status == CL_SUCCESS);
  printf("Specify the shape of the domain completes.\n");
  /* getresult */
#ifdef USEclFINSIH
  clFinish(commandQueue);
#else
  clEnqueueReadBuffer(commandQueue, bufferGlobalId, CL_TRUE,
                      0, 2 * N * N * sizeof(cl_uint), globalId,
                      0, NULL, NULL);
#endif
  printf("Kernel execution completes.\n");

  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)

```c
void printId(char *title, cl_uint id[N][N])
{
  puts(title);
  for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++)
      printf("%2d ", id[i][j]);
    printf("\n");
  }
}
```

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

```
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)

```
#define N 16

__kernel void getGlobalId(__global int globalId[2][N][N],
          __global int localId[2][N][N])
{
        int id0 = get_global_id(0);
        int id1 = get_global_id(1);
        globalId[0][id0][id1] = get_global_id(0);
        globalId[1][id0][id1] = get_global_id(1);
        localId[0][id0][id1] = get_local_id(0);
        localId[1][id0][id1] = get_local_id(1);
}
```

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)

```
  cl_mem bufferGlobalId =
    clCreateBuffer(context,
                   CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
                   2 * N * N * sizeof(cl_uint), globalId, &status)
  assert(status == CL_SUCCESS);
  cl_mem bufferLocalId =
    clCreateBuffer(context,
                   CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
                   2 * N * N * sizeof(cl_uint), localId, &status);
  assert(status == CL_SUCCESS);
  printf("Build buffers completes\n");
  /* setarg */
  status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
                          (void*)&bufferGlobalId);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
                          (void*)&bufferLocalId);
  assert(status == CL_SUCCESS);
  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)

```
  size_t globalDim[] = {(size_t)N, (size_t)N};
  int groupRow = atoi(argv[2]);
  int groupCol = atoi(argv[3]);
  size_t localDim[] = {groupRow, groupCol};
  status =
    clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL,
                           globalDim, localDim,
                           0, NULL, NULL);
  assert(status == CL_SUCCESS);
  printf("Specify the shape of the domain completes.\n");
  /* getresult */
#ifdef USEclFINSIH
  clFinish(commandQueue);
#else
  clEnqueueReadBuffer(commandQueue, bufferGlobalId, CL_TRUE,
                      0, 2 * N * N * sizeof(cl_uint), globalId,
                      0, NULL, NULL);
  clEnqueueReadBuffer(commandQueue, bufferLocalId, CL_TRUE,
                      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 preform an inner product.

Kernel

Example 15: (mul-kernel.cl)

```
#define N 1024

__kernel void mul(__global int matrixA[N][N],
                  __global int matrixB[N][N],
                  __global int matrixC[N][N])
{
  int row = get_global_id(0);
  int col = get_global_id(1);
  int sum = 0;
  for (int i = 0; i < N; i++)
    sum += matrixA[row][i] * matrixB[i][col];
  matrixC[row][col] = sum;
}
```

Matrices

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

Buffers

Example 16: (matrixMul.c)

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

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)

```
  cl_mem bufferA =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                   N * N * sizeof(cl_uint), A, &status);
  assert(status == CL_SUCCESS);
  cl_mem bufferB =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                   N * N * sizeof(cl_uint), B, &status);
  assert(status == CL_SUCCESS);
  cl_mem bufferC =
    clCreateBuffer(context,
                   CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
                   N * N * sizeof(cl_uint), C, &status);
  assert(status == CL_SUCCESS);
  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)

```
  size_t globalThreads[] = {(size_t)N, (size_t)N};
  size_t localThreads[] = {1, 1};
  status =
    clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL,
                           globalThreads, localThreads,
                           0, NULL, NULL);
  assert(status == CL_SUCCESS);
  printf("Specify the shape of the domain completes.\n");
  /* getcvector */
#ifdef USEclFINSIH
  clFinish(commandQueue);
#else
  clEnqueueReadBuffer(commandQueue, bufferC, CL_TRUE,
                      0, N * N * sizeof(cl_uint), C,
                      0, NULL, NULL);
#endif
  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.
- The kernel is sent 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

```
cl_command_queue
clCreateCommandQueueWithProperties(
          cl_context context,
          cl_device_id device,
          const cl_queue_properties
          *properties,
          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)

```
const cl_queue_properties properties[] =
  {CL_QUEUE_PROPERTIES, CL_QUEUE_PROFILING_ENABLE, 0};
cl_command_queue commandQueue =
  clCreateCommandQueueWithProperties(context, GPU[0],
                                     properties, &status);
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

```
cl_int
clEnqueueNDRangeKernel (cl_command_queue command_queue,
                        cl_kernel kernel,
                        cl_uint work_dim,
                        const size_t *global_work_offset,
                        const size_t *global_work_size,
                        const size_t *local_work_size,
                        cl_uint num_events_in_wait_list,
                        const cl_event *event_wait_list,
                        cl_event *event);
```

Example 22: (matrixMul-time.c)

```
  size_t globalThreads[] = {(size_t)N, (size_t)N};
  size_t localThreads[] = {1, 1};
  cl_event event;
  status =
    clEnqueueNDRangeKernel(commandQueue, kernel, 2, NULL,
                           globalThreads, localThreads,
                           0, NULL, &event);
  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

```
cl_int clWaitForEvents(cl_uint num_events,
                       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)

```
status = clWaitForEvents(1, &event);
assert(status == CL_SUCCESS);
clEnqueueReadBuffer(commandQueue, bufferC, CL_TRUE,
                    0, N * N * sizeof(cl_uint), C,
                    0, NULL, NULL);
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 joined the command queue for execution.
- When it is the turn of the kernel, it is submitted to the device for execution.
- When the kernel finished 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 was sent 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

```
cl_int
clGetEventProfilingInfo(cl_event event,
                        cl_profiling_info param_name,
                        size_t param_value_size,
                        void *param_value,
                        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)

```
  cl_ulong timeEnterQueue, timeSubmit, timeStart, timeEnd;
  status =
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_QUEUED,
                            sizeof(cl_ulong), &timeEnterQueue, NUL
  assert(status == CL_SUCCESS);
  status =
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_SUBMIT,
                            sizeof(cl_ulong), &timeSubmit, NULL);
  assert(status == CL_SUCCESS);
  status =
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_START,
                            sizeof(cl_ulong), &timeStart, NULL);
  assert(status == CL_SUCCESS);
  status =
    clGetEventProfilingInfo(event, CL_PROFILING_COMMAND_END,
                            sizeof(cl_ulong), &timeEnd, NULL);
  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)

```
  printf("kernel queued time %f seconds\n",
         (timeSubmit - timeEnterQueue) / 1000000000.0);
  printf("kernel submission time %f seconds\n",
         (timeStart - timeSubmit) / 1000000000.0);
  printf("kernel execution time %f seconds\n",
         (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)

```
  cl_mem bufferA =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                   N * N * sizeof(cl_uint), A, &status);
  assert(status == CL_SUCCESS);
  cl_mem bufferB =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                   N * N * sizeof(cl_uint), B, &status);
  assert(status == CL_SUCCESS);
  cl_mem bufferC =
    clCreateBuffer(context,
                   CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
                   N * N * sizeof(cl_uint), C, &status);
  assert(status == CL_SUCCESS);
  printf("Build buffers completes\n");
```

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

```
  cl_mem bufferA =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                   N * N * sizeof(cl_uint), A, &status);
  assert(status == CL_SUCCESS);
  cl_mem bufferB =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_USE_HOST_PTR,
                   N * N * sizeof(cl_uint), B, &status);
  assert(status == CL_SUCCESS);
  cl_mem bufferC =
    clCreateBuffer(context,
                   CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
                   N * N * sizeof(cl_uint), C, &status);
  assert(status == CL_SUCCESS);
  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.

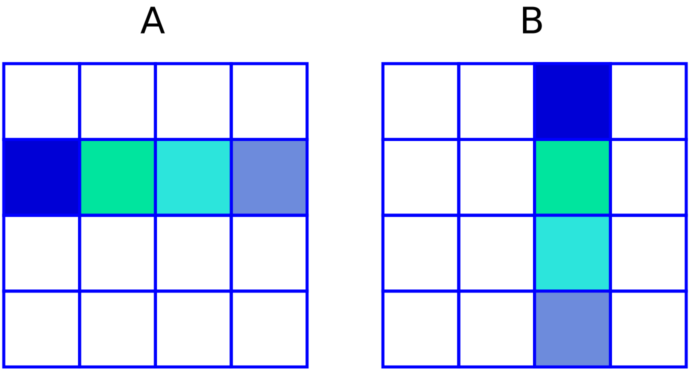
Work Group

- 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



Discussion

- Make sure that you understand this algorithm.

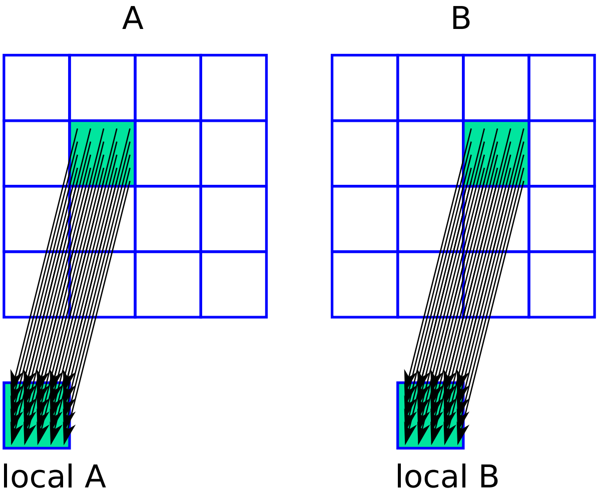
Fast multiplication

- 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)

```
#define N 1024
#define Blk 64
#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)

```
__kernel void mul(__global int A[N][N],
                  __global int B[N][N],
                  __global int C[N][N])
{
  int globalRow = get_global_id(0);
  int globalCol = get_global_id(1);
  int localRow = get_local_id(0);
  int localCol = get_local_id(1);

  __local int ALocal[BSIDE][BSIDE];
  __local int BLocal[BSIDE][BSIDE];
```

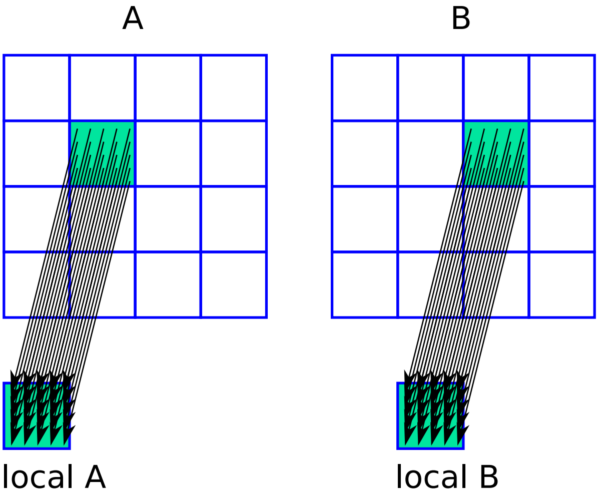
Data Movement

- 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 a thread only moves two data, other data are moved by other threads – we do not know if they have finished or not.
- We need to synchronize with all other threads in the work group.

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

```
  int sum = 0;
  for (int block = 0; block < Blk; block++) {
    ALocal[localRow][localCol] =
      A[globalRow][block * BSIDE + localCol];
    BLocal[localRow][localCol] =
      B[block * BSIDE + localRow][globalCol];
    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

```
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.

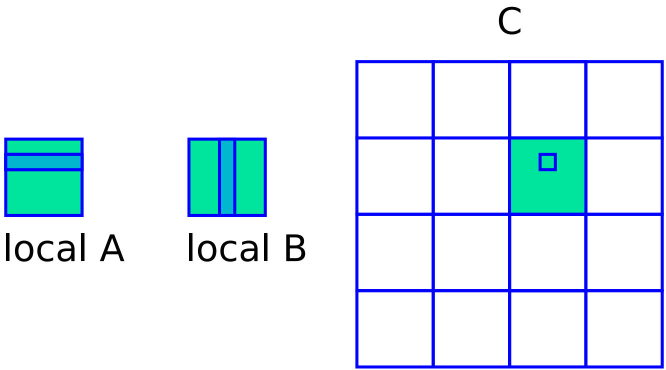
Example 34: (mul-local-kernel.cl)

```
  int sum = 0;
  for (int block = 0; block < Blk; block++) {
    ALocal[localRow][localCol] =
      A[globalRow][block * BSIDE + localCol];
    BLocal[localRow][localCol] =
      B[block * BSIDE + localRow][globalCol];
    barrier(CLK_LOCAL_MEM_FENCE);
    /* inner */
    for (int k = 0; k < BSIDE; k++)
      sum += ALocal[localRow][k] * BLocal[k][localCol];
    barrier(CLK_LOCAL_MEM_FENCE);
  }
  C[globalRow][globalCol] = sum;
}
```

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



Demonstration

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

Discussion

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