

Advanced OpenCL Programming

Pangfeng Liu
National Taiwan University

May 31, 2020

Multiple Devices

- We have been using one device for computation.
- Now we will use multiple devices to solve a problem.
- We will use matrix multiplication as an example.

Partition

- Since we are very poor and cannot afford four GPUs, we will partition the data by rows (not by block) among devices.
- We will partition matrix by rows, so that the memory assigned to a device is contiguous.
- Each kernel will run on a device, and compute part of the answers.

Partition

- We will partition A and C by rows, and do *not* partition B .
- For example, if we use two devices, then the first kernel will compute the top half of C , and another kernel will compute the bottom half of C .
- The reason for not partitioning B is that all kernels need the entire B .

Constants

- MAXLOG is the maximum number of bytes for storing compilation log. More on this later.
- The number of device is DEVICENUM. We will use 2 in our humble installation.
- ITEMPERDEVICE is the number of work item in NDRange of a kernel.

Constants

Example 1: (matrixMul-time-copy-local-multidevice.c)

```
2
3 #include <stdio.h>
4 #include <assert.h>
5 #include <CL/cl.h>
6
7 #define N 1024
8 #define Blk 64
9 #define BSIDE (N / Blk)
10 #define MAXGPU 10
11 #define MAXK 1024
12 #define MAXLOG 4096
13 #define DEVICENUM 2
14 #define ITEMPERDEVICE (N * N / DEVICENUM)
15 #define NANO2SECOND 1000000000.0
16
17 cl_uint A[N][N], B[N][N], C[N][N];
```

Discussion

- If N is 1024, and the number of device is 2, then how many works items are there in a kernel?

Select Devices

- We will select the first DEVICENUM GPU device from our humble installation.
- The code is similar to those earlier version, except that now we need to make sure that the number of GPUs found is at least DEVICENUM.

Devices

Example 2: (matrixMul-time-copy-local-multidevice.c)

```
30  cl_device_id GPU[MAXGPU];  
31  cl_uint GPU_id_got;  
32  status =  
33      clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU,  
34                      MAXGPU, GPU, &GPU_id_got);  
35  assert(status == CL_SUCCESS &&  
36          GPU_id_got >= DEVICENUM);  
37  printf("There are %d GPU devices\n", GPU_id_got);
```

Context

- Since a context can consist of multiple devices, we need only one context.
- Here we include the first `DEVICENUM` GPUs in the context.

Context

Example 3: (matrixMul-time-copy-local-multidevice.c)

```
39  cl_context context =  
40      clCreateContext(NULL, DEVICENUM, GPU, NULL, NULL,  
41                      &status);  
42  assert(status == CL_SUCCESS);
```

Multiple Queue

- Recall that a command queue connects to a device.
- Since we have multiple devices, we need multiple command queues.
- We put the command queues in the `commandQueue` array.
- Note that we set the `CL_QUEUE_PROFILING_ENABLE` to enable profiling.

Command Queue

Example 4: (matrixMul-time-copy-local-multidevice.c)

```
44  const cl_queue_properties properties[] =  
45      {CL_QUEUE_PROPERTIES, CL_QUEUE_PROFILING_ENABLE, 0};  
46  cl_command_queue commandQueue[DEVICENUM];  
47  for (int device = 0; device < DEVICENUM; device++) {  
48      commandQueue[device] =  
49          clCreateCommandQueueWithProperties(context, GPU[device],  
50                                             properties, &status);  
51      assert(status == CL_SUCCESS);  
52  }
```

One Context, Multiple Devices

- Up to this point, we can image there is only one context, which has multiple devices.
- Within this context we will have one command queue to connect to each device.
- Later we will send commands to these devices. The command sent into a command queue will run on the device this command queue connects to.

Discussion

- What will be sent into these command queues as commands?

Program

- The kernel will be sent into the command queue as commands.
- Before this can happen we need to compile the kernel.
- Recall that the “kernel” at this point is only a set of strings, which need to be compiled.

Build Program

- We call `clBuildProgram` to build executable.
- Note that we pass *all* devices we want to use as parameters, so OpenCL will build executable for all our `DEVICENUM` devices.

Compilation Log

- The kernel source is compiled when an OpenCL program runs.
- It is very inconvenient because when we run an OpenCL program the execution will not display compilation errors of the kernel.
- We will use `clGetProgramBuildInfo` to show the error message if the compilation of the kernel fails.

Prototype 5: clGetProgramBuildInfo.h

```
1  cl_int  clGetProgramBuildInfo(cl_program program,  
2                                cl_device_id device,  
3                                cl_program_build_info param_name,  
4                                size_t param_value_size,  
5                                void *param_value,  
6                                size_t *param_value_size_ret);
```

Parameters

`program` The compiled program.

`device` The device you want to query.

`param_name` The information you want to query.

`param_value_size` The length of `param_value` in bytes.

`param_value` The location for the query answer.

`param_value_size_ret` The number of bytes returned from the query.

Error

- When there is an error in `clBuildProgram`, we will call `clGetProgramBuildInfo` to find out.
- Since we are not sure about which device causes the compilation error, we list the information from all of them.
- We query with `CL_PROGRAM_BUILD_LOG` for the compilation log, and place it into the log buffer we prepared.

Build Program

Example 6: (matrixMul-time-copy-local-multidevice.c)

```
66  status =
67      clBuildProgram(program, DEVICENUM, GPU, NULL,
68                      NULL, NULL);
69  if (status != CL_SUCCESS) {
70      char log[MAXLOG];
71      size_t logLength;
72      for (int device = 0; device < DEVICENUM; device++) {
73          clGetProgramBuildInfo(program, GPU[device],
74                                CL_PROGRAM_BUILD_LOG,
75                                MAXLOG, log, &logLength);
76          puts(log);
77      }
78      exit(-1);
79  }
80  printf("Build program completes\n");
```

Discussion

- What is the type of the compilation log?
- What does puts do?

Partition

- Now we are ready to partition host buffers A , B and C .
- We will create `DEVICENUM` buffers for A , and each of which will be given to a kernel as a parameter for computations.
- We partition A into sub-matrix of $N / \text{DEVICENUM}$ rows of A each, and assign each partition to an OpenCL buffer.

Buffer A

- ITEMPERDEVICE means the number of items per device.
- The index device determines the starting position of a buffer, i.e., where the matrix A will be partitioned.

Buffer for A

Example 7: (matrixMul-time-copy-local-multidevice.c)

```
92  cl_mem bufferA[DEVICENUM];
93  for (int device = 0; device < DEVICENUM; device++) {
94      bufferA[device] =
95          clCreateBuffer(context,
96              CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
97              ITEMPERDEVICE * sizeof(cl_uint),
98              ((cl_uint *)A) + device * ITEMPERDEVICE,
99              &status);
100  assert(status == CL_SUCCESS);
101  }
```

Buffer B

- We do not need to partition B because it will be used by all kernels.

Buffer for B

Example 8: (matrixMul-time-copy-local-multidevice.c)

```
103  cl_mem bufferB =  
104      clCreateBuffer(context,  
105          CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,  
106          N * N * sizeof(cl_uint), B, &status);  
107  assert(status == CL_SUCCESS);
```

Buffer C

- Buffers for C are similarly built as for A .
- Unlike A and B that will copy host buffers to device buffers, C will use host memory directly.

Buffer for C

Example 9: (matrixMul-time-copy-local-multidevice.c)

```
109  cl_mem bufferC[DEVICENUM];
110  for (int device = 0; device < DEVICENUM; device++) {
111      bufferC[device] =
112          clCreateBuffer(context,
113              CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
114              ITEMPERDEVICE * sizeof(cl_uint),
115              ((cl_uint *) C) + device * ITEMPERDEVICE,
116              &status);
117      assert(status == CL_SUCCESS);
118  }
119  printf("Build buffers completes\n");
```

Discussion

- Make sure that you understand the offset calculation in A and C .

NDRange

- According to our partition we declare `NDRange` as a two dimension array with $N / \text{DEVICENUM}$ rows and N columns.
- This is consistent with C , where each work item will compute an element of C .
- The work group is still `BSIDE` by `BSIDE` since we will use the previous local memory algorithm.
- We will have `DEVICENUM` kernels, so we will need `DEVICENUM` events for synchronization and profiling.

NDRange

Example 10: (matrixMul-time-copy-local-multidevice.c)

```
121 size_t globalThreads[] =  
122     {(size_t)(N / DEVICENUM), (size_t)N};  
123 size_t localThreads[] = {BSIDE, BSIDE};  
124 cl_event events[DEVICENUM];
```

Set Argument

- We will launch `DEVICENUM` kernels. Each of them will compute $N / \text{DEVICENUM}$ rows and N columns of C .
- We loop through all devices, and assign the argument order as A , B , and C .
- Note that we need to supply the correct A and C since there are `DEVICENUM` of them.

Set Arguments

Example 11: (matrixMul-time-copy-local-multidevice.c)

```
126  for (int device = 0; device < DEVICENUM; device++) {
127      status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
128                              (void*)&bufferA[device]);
129      assert(status == CL_SUCCESS);
130      status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
131                              (void*)&bufferB);
132      assert(status == CL_SUCCESS);
133      status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
134                              (void*)&bufferC[device]);
135      assert(status == CL_SUCCESS);
136      printf("Set kernel arguments completes\n");
```

Kernel

- Now we are ready to launch kernel.
- Note that we need to supply the command queue to the device as a parameters.
- The same kernel is launched again and again for `DEVICENUM` times. The only difference is the buffers *A* and *C* used in launching the kernel.
- We associate an event with each kernel launching.

Start Kernel

Example 12: (matrixMul-time-copy-local-multidevice.c)

```
138     status =  
139         clEnqueueNDRangeKernel(commandQueue[device],  
140                                kernel, 2, NULL,  
141                                globalThreads, localThreads,  
142                                0, NULL, &(events[device]));  
143     assert(status == CL_SUCCESS);  
144 }
```

Wait for Completion

- Since kernel launching is non-blocking. we need to wait for them to complete.
- We simply use `clWaitForEvents`, and wait for all events to complete.

Wait for Kernels

Example 13: (matrixMul-time-copy-local-multidevice.c)

```
146 clWaitForEvents(DEVICENUM, events);
147 for (int device = 0; device < DEVICENUM; device++) {
148     clEnqueueReadBuffer(commandQueue[device], bufferC[device],
149                         0, ITEMPERDEVICE * sizeof(cl_uint),
150                         0, NULL, NULL);
151 }
152 printf("Kernel execution completes.\n");
```

Discussion

- How many events do we need to wait for the whole thing to complete?

Kernel

- The kernel function is very similar to the previous local memory version.
- We also use the constant `DEVICENUM` to denote the number of devices.

Kernel

Example 14: (mul-local-multidevice-kernel.cl)

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

Kernel

Example 15: (mul-local-multidevice-kernel.cl)

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

NDRange

- We declare A and C as $N / \text{DEVICENUM}$ by N arrays, because the domain is partitioned among devices.
- The rest of the code is *not* changed at all!

Reason

- The reason for this “no need to change” is that a kernel function is acting from a local view, i.e., it is a work item within a work group, so all the operations are still the same.
- The kernel still thinks from the point of view of blocks, and the related operations are done in local indices.
- We only need the global index while accessing A , B , and C .
- Since the kernel is given the corresponding parts of A and B , so the operation is correct.

Discussion

- Convince yourself that the code is correct.

Timing

- We would like to know the detailed timing of the kernels.
- In particular we would like to prove the two kernel run *in parallel*.
- To do so we need the absolute time of the events from all devices.

Base

- The time returned by the event is difficult to interpret.
- We would like to establish a relative time for inspection.
- We choose the time the first kernel entering the queue as the base time, and report the relative time to it for ease understanding.

Get Time

Example 16: (matrixMul-time-copy-local-multidevice.c)

```
154 cl_ulong base;  
155 status =  
156     clGetEventProfilingInfo(events[0],  
157     CL_PROFILING_COMMAND_QUEUED,  
158     sizeof(cl_ulong), &base, NULL);  
159 assert(status == CL_SUCCESS);
```

Get Time

- We loop through all devices and get timing information from all of them.
- This part is the same as before.

Get Time

Example 17: (matrixMul-time-copy-local-multidevice.c)

```
161 for (int device = 0; device < DEVICENUM; device++) {
162     cl_ulong timeEnterQueue, timeSubmit, timeStart,
163         timeEnd;
164     status =
165         clGetEventProfilingInfo(events[device],
166             CL_PROFILING_COMMAND_QUEUED,
167             sizeof(cl_ulong), &timeEnterQueue, NULL);
168     assert(status == CL_SUCCESS);
169     status =
170         clGetEventProfilingInfo(events[device],
171             CL_PROFILING_COMMAND_SUBMIT,
172             sizeof(cl_ulong), &timeSubmit, NULL);
173     assert(status == CL_SUCCESS);
```

Get Time

Example 18: (matrixMul-time-copy-local-multidevice.c)

```
175     status =  
176         clGetEventProfilingInfo(events[device],  
177             CL_PROFILING_COMMAND_START,  
178             sizeof(cl_ulong), &timeStart, NULL);  
179     assert(status == CL_SUCCESS);  
180     status =  
181         clGetEventProfilingInfo(events[device],  
182             CL_PROFILING_COMMAND_END,  
183             sizeof(cl_ulong), &timeEnd, NULL);  
184     assert(status == CL_SUCCESS);
```

Relative Time

- In addition to queuing time, submission time, and execution, we also report the relative time when the four events happened.
- The relative time is in nanosecond.

Print Time

Example 19: (matrixMul-time-copy-local-multidevice.c)

```
186 printf("\nkernel entered queue at %f\n",
187         (timeEnterQueue - base) / NANO2SECOND);
188 printf("kernel submitted to device at %f\n",
189         (timeSubmit - base) / NANO2SECOND);
190 printf("kernel started at %f\n",
191         (timeStart - base) / NANO2SECOND);
192 printf("kernel ended at %f\n",
193         (timeEnd - base) / NANO2SECOND);
194 printf("kernel queued time %f seconds\n",
195         (timeSubmit - timeEnterQueue) / NANO2SECOND);
196 printf("kernel submission time %f seconds\n",
197         (timeStart - timeSubmit) / NANO2SECOND);
198 printf("kernel execution time %f seconds\n",
199         (timeEnd - timeStart) / NANO2SECOND);
200 }
```

Demonstration

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

Discussion

- Does the kernels run in parallel? Observe the results and give your answer.

Dependency

- We may launch multiple kernels into multiple devices.
- Kernels may have dependency, and we need to ensure dependency among kernels.

Example

- In the following example we compute $G = (A + B) + (D + E)$, where all vector has length N .
- We compute $C = A + B$, then $F = D + E$, then we compute $G = C + F$. The last computation must wait for the first two to complete.
- We will use one device for *each* of the three additions.

Constants

- We first declare the constants and variables.

Example 20: (vectorAdd-dependency.c)

```
2  #define CL_USE_DEPRECATED_OPENCL_2_0_APIS
3  #include <stdio.h>
4  #include <assert.h>
5  #include <CL/cl.h>
6  #define N (65536 * 4)
7  #define MAXGPU 10
8  #define MAXK 1024
9  #define MAXLOG 4096
10 #define DEVICENUM 3
11 #define NANOS2SECOND 1000000000.0
12
13 cl_uint A[N], B[N], C[N], D[N], E[N], F[N], G[N];
```

Initialization

- We initialize the A , B , D and E vectors.

Example 21: (vectorAdd-dependency.c)

```
82  for (int i = 0; i < N; i++) {  
83      A[i] = i;  
84      B[i] = N - i;  
85      D[i] = i;  
86      E[i] = N - i;  
87  }
```

Buffers

- We then create buffers for kernel parameters.
- A and B are read only, and need to be copied into device.
- C is both read and write enable.

Example 22: (vectorAdd-dependency.c)

```
89  cl_mem bufferA =
90      clCreateBuffer(context,
91                      CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
92                      N * sizeof(cl_uint), A, &status);
93  assert(status == CL_SUCCESS);
94  cl_mem bufferB =
95      clCreateBuffer(context,
96                      CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
97                      N * sizeof(cl_uint), B, &status);
98  assert(status == CL_SUCCESS);
99  cl_mem bufferC =
100      clCreateBuffer(context,
101                      CL_MEM_READ_WRITE,
102                      N * sizeof(cl_uint), NULL, &status);
103  assert(status == CL_SUCCESS);
```


Buffers

- Similarly we create buffers for D , E and F .

Example 23: (vectorAdd-dependency.c)

```
105 cl_mem bufferD =
106     clCreateBuffer(context,
107                    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
108                    N * sizeof(cl_uint), D, &status);
109 assert(status == CL_SUCCESS);
110 cl_mem bufferE =
111     clCreateBuffer(context,
112                    CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
113                    N * sizeof(cl_uint), E, &status);
114 assert(status == CL_SUCCESS);
115 cl_mem bufferF =
116     clCreateBuffer(context,
117                    CL_MEM_READ_WRITE,
118                    N * sizeof(cl_uint), NULL, &status);
119 assert(status == CL_SUCCESS);
```

Buffers

- We create buffers for G .
- This buffer is write only and use the host memory directly, so the host can print it directly.

Example 24: (vectorAdd-dependency.c)

```
121 cl_mem bufferG =  
122     clCreateBuffer(context,  
123                   CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,  
124                   N * sizeof(cl_uint), G, &status);  
125 assert(status == CL_SUCCESS);  
126 printf("Build buffers completes\n");
```

Discussion

- What are the characteristics for read only, read and write, and write only buffers respectively?

Buffers

- Both `NDRange` and the work group are one dimensional.
- The size of `NDRange` is N .
- The size of a work group is 256.

Example 25: (vectorAdd-dependency.c)

```
128 size_t globalThreads[] = {(size_t)N};  
129 size_t localThreads[] = {256};
```

$$C = A + B$$

- Now we launch the first kernel for $C = A + B$.
- We have already built three command queues for three devices.
- We use `commandQueue[0]` for the first addition.
- We also declared three events to denote the completion of the three kernels that we will launch.

Example 26: (vectorAdd-dependency.c)

```
131 cl_event events[3];
132 status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
133                          (void*)&bufferA);
134 assert(status == CL_SUCCESS);
135 status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
136                          (void*)&bufferB);
137 assert(status == CL_SUCCESS);
138 status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
139                          (void*)&bufferC);
140 assert(status == CL_SUCCESS);
141
142 status =
143     clEnqueueNDRangeKernel(commandQueue[0], kernel, 1, NULL,
144                             globalThreads, localThreads,
145                             0, NULL, &(events[0]));
146 assert(status == CL_SUCCESS);
```

$$F = D + E$$

- Similarly we launch the second kernel for $F = D + E$.
- Note that we used `commandQueue[1]` and `events[1]`.

Example 27: (vectorAdd-dependency.c)

```
148 status = clSetKernelArg(kernel, 0, sizeof(cl_mem),  
149                          (void*)&bufferD);  
150 assert(status == CL_SUCCESS);  
151 status = clSetKernelArg(kernel, 1, sizeof(cl_mem),  
152                          (void*)&bufferE);  
153 assert(status == CL_SUCCESS);  
154 status = clSetKernelArg(kernel, 2, sizeof(cl_mem),  
155                          (void*)&bufferF);  
156 assert(status == CL_SUCCESS);  
157 status =  
158     clEnqueueNDRangeKernel(commandQueue[1], kernel, 1, NULL,  
159                             globalThreads, localThreads,  
160                             0, NULL, &(events[1]));  
161 assert(status == CL_SUCCESS);
```

$$G = C + F$$

- Finally we launch the third kernel for $G = C + F$.
- Note that we used `commandQueue[2]` and `events[2]`.

Example 28: (vectorAdd-dependency.c)

```
163 status = clSetKernelArg(kernel, 0, sizeof(cl_mem),  
164                          (void*)&bufferC);  
165 assert(status == CL_SUCCESS);  
166 status = clSetKernelArg(kernel, 1, sizeof(cl_mem),  
167                          (void*)&bufferF);  
168 assert(status == CL_SUCCESS);  
169 status = clSetKernelArg(kernel, 2, sizeof(cl_mem),  
170                          (void*)&bufferG);  
171 assert(status == CL_SUCCESS);  
172 status =  
173     clEnqueueNDRangeKernel(commandQueue[2], kernel, 1, NULL,  
174                             globalThreads, localThreads,  
175                             0, NULL, &(events[2]));  
176 assert(status == CL_SUCCESS);
```

Wait for Events

- We wait for all three kernel to finish.

Example 29: (vectorAdd-dependency.c)

```
178     status = clWaitForEvents(DEVICENUM, events);
179     assert(status == CL_SUCCESS);
180     #ifdef COPYG
181         clEnqueueReadBuffer(commandQueue[0], bufferG, CL_TRUE,
182                             0, N * sizeof(cl_uint), G,
183                             0, NULL, NULL);
184     #endif
185     printf("All three kernels complete.\n");
```

Wait for Events

- The rest of the code just prints the timing information, checks for correctness, and releases resources.
- Please refer to previous discussion.

Demonstration

- Run the `vectorAdd-dependency-cl`.

Discussion

- Does the program produce the correct answer?
- If not, what is the possible reason?

Problem

- The previous program waits for all three kernels to complete, but the third kernel did *not* wait for the first two.
- We will use the *wait for events* mechanism to launch the third kernel.

Prototype 30: 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);
```

$$G = C + F$$

- We launch the third kernel and wait for the first two events in the events array.
- Set `num_events_in_wait_list` to 2 and `event_wait_list` to events.

Example 31: (vectorAdd-dependency-correct.c)

```
163 status = clSetKernelArg(kernel, 0, sizeof(cl_mem),  
164                          (void*)&bufferC);  
165 assert(status == CL_SUCCESS);  
166 status = clSetKernelArg(kernel, 1, sizeof(cl_mem),  
167                          (void*)&bufferF);  
168 assert(status == CL_SUCCESS);  
169 status = clSetKernelArg(kernel, 2, sizeof(cl_mem),  
170                          (void*)&bufferG);  
171 assert(status == CL_SUCCESS);  
172 status =  
173     clEnqueueNDRangeKernel(commandQueue[2], kernel, 1, NULL,  
174                             globalThreads, localThreads,  
175                             2, events, &(events[2]));  
176 assert(status == CL_SUCCESS);
```

Demonstration

- Run the `vectorAdd-dependency-correct-cl`.

Discussion

- Does the program produce the correct answer?
- Observe the timing and make sure the first two kernels run in parallel, and the third kernel will wait for the first two.

`clFinish`

- The host can also explicitly wait for the first two kernels to finish before launching the third kernel by `clFinish`.
- We just need to wait for the command queues to the first two devices to become empty.

Example 32: (vectorAdd-dependency-clfinish.c)

```
167 clFinish(commandQueue[0]);
168 clFinish(commandQueue[1]);
169 status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
170                          (void*)&bufferC);
171 assert(status == CL_SUCCESS);
172 status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
173                          (void*)&bufferF);
174 assert(status == CL_SUCCESS);
175 status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
176                          (void*)&bufferG);
177 assert(status == CL_SUCCESS);
178 status =
179     clEnqueueNDRangeKernel(commandQueue[0], kernel, 1, NULL,
180                             globalThreads, localThreads,
181                             0, NULL, &(events[2]));
182 assert(status == CL_SUCCESS);
```

Demonstration

- Run the `vectorAdd-dependency-clfinish-cl` program.

Discussion

- Does the program produce the correct answer?
- Observe the timing and make sure the first two kernels run in parallel, and the third kernel will wait for the first two.
- Compare the timing results between `vectorAdd-dependency-correct-cl` and `vectorAdd-dependency-clfinish-cl`, especially in when the last kernel joined the command queue.

Group Size

- The previous matrix multiplication program has extremely good performance due to two reasons.
 - It uses local memory to speed up data access.
 - It has a larger group size (256).
- Now we would like to study the effects of group size on performance.

Compute Units

- A device has only a limited number of compute units.
- The work items of a work group will occupy a compute unit.
- For a given number of work items, if the work group size is small, then the number of work groups becomes large, and some work groups will wait for compute unit.
- In other words, a small work group size limits the the number of work items that we can process in parallel.

Classroom

- A school has only a limited number of classrooms.
- The students are divided into group, and one group will occupy a classroom.
- For a given number of students, if the group size is small, then the number of groups becomes large, and some groups will wait for classroom.
- In other words, a small group size limits the the number of students that can study in parallel.

Discussion

- Give your own example of this phenomenon.

Compute Unit Number

- We need only one GPU device.
- We call `clGetDeviceInfo` with `CL_DEVICE_MAX_COMPUTE_UNITS` to get the number of compute units on GPU.
- We need the number of compute units to determine the work group size.

Example 33: (vectorAdd-groupsize.c)

```
24  cl_device_id GPU[MAXGPU];
25  cl_uint GPU_id_got;
26  status =
27      clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU,
28                      MAXGPU, GPU, &GPU_id_got);
29  assert(status == CL_SUCCESS && GPU_id_got >= 1);
30  printf("There are %d GPU devices\n", GPU_id_got);
31  cl_uint unit;
32  status =
33      clGetDeviceInfo(GPU[0], CL_DEVICE_MAX_COMPUTE_UNITS,
34                      sizeof(cl_uint), &unit, NULL);
35  assert(status == CL_SUCCESS);
36  printf("# of compute units is %d\n", unit);
```

Group Size

- We fix the number of work items to N , and vary the size of the work group from 1 to 256.
- We wait for the event before launching the next kernel.
- We will record the timing information into a file `vectorAdd-groupsize.dat`.

Example 34: (vectorAdd-groupsize.c)

```
110 size_t workItem[] = {(size_t)N};
111 FILE *timefp = fopen("vectorAdd-groupsize.dat", "w");
112 assert(timefp != NULL);
113 for (int groupSize = 1; groupSize <= 256; groupSize *= 2) {
114     cl_event event;
115     size_t localSize[1];
116     localSize[0] = groupSize;
117     status =
118         clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
119                                workItem, localSize,
120                                0, NULL, &event);
121     assert(status == CL_SUCCESS);
122     /* waitforevent */
123     status = clWaitForEvents(1, &event);
124     assert(status == CL_SUCCESS);
125     printf("The kernel with group size %d completes.\n",
126            groupSize);
```

Get Timing Information

- The rest of the code will retrieve timing information from the event.
- We output the group size and the execution time into a file `vectorAdd-grouopsize.dat`.

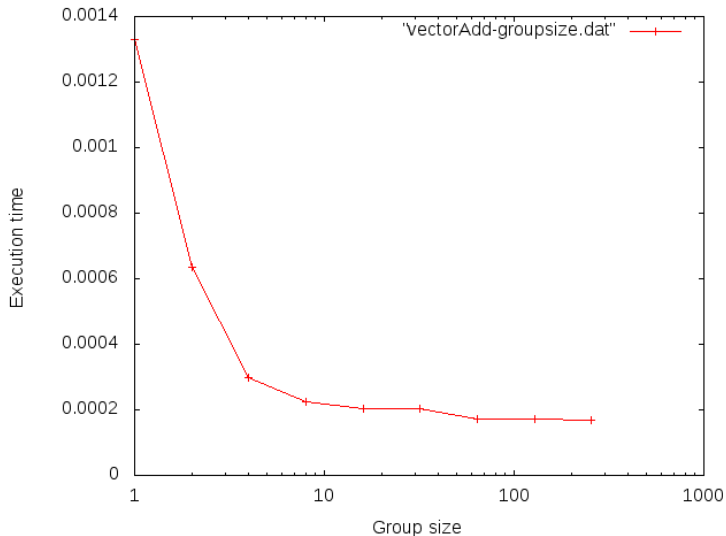
Example 35: (vectorAdd-groupsize.c)

```
159     printf("\nkernel entered queue at %f\n",
160           (timeEnterQueue - base) / NANO2SECOND);
161     printf("kernel submitted to device at %f\n",
162           (timeSubmit - base) / NANO2SECOND);
163     printf("kernel started at %f\n",
164           (timeStart - base) / NANO2SECOND);
165     printf("kernel ended at %f\n",
166           (timeEnd - base) / NANO2SECOND);
167     printf("kernel queued time %f seconds\n",
168           (timeSubmit - timeEnterQueue) / NANO2SECOND);
169     printf("kernel submission time %f seconds\n",
170           (timeStart - timeSubmit) / NANO2SECOND);
171     printf("kernel execution time %f seconds\n",
172           (timeEnd - timeStart) / NANO2SECOND);
173     fprintf(timefp, "%d %f\n", groupSize,
174           (timeEnd - timeStart) / NANO2SECOND);
175 }
```

Demonstration

- We plot the kernel execution time for different group sizes.
- The x-axis is in log scale.

Execution Time



Discussion

- Observe the execution time and draw your conclusion about the effects of group size on performance.