Advanced OpenCL Programming

Pangfeng Liu
National Taiwan University

June 7, 2015

Multiple Devices

- We have been using one device for computation.
- Now we will use multiple device 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)

```c
#define CL_USE_DEPRECATED_OPENCL_2_0_APIS
#include <stdio.h>
#include <assert.h>
#include <CL/cl.h>

#define N 1024
#define Blk 64
#define BSIDE (N / Blk)
#define MAXGPU 10
#define MAXK 1024
#define MAXLOG 4096
#define DEVICENUM 2
#define ITEMPERDEVICE (N * N / DEVICENUM)
#define NANO2SECOND 1000000000.0

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)

```
  cl_device_id GPU[MAXGPU];
  cl_uint GPU_id_got;
  status =
    clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU,
                   MAXGPU, GPU, &GPU_id_got);
  assert(status == CL_SUCCESS &&
         GPU_id_got >= DEVICENUM);
  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)

```
  cl_context context =
    clCreateContext(NULL, DEVICENUM, GPU, NULL, NULL,
                    &status);
  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)

```
  cl_command_queue commandQueue[DEVICENUM];
  for (int device = 0; device < DEVICENUM; device++) {
    commandQueue[device] =
      clCreateCommandQueue(context, GPU[device],
                           CL_QUEUE_PROFILING_ENABLE,
                           &status);
    assert(status == CL_SUCCESS);
  }
```

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

```
cl_int clGetProgramBuildInfo(cl_program program,
                              cl_device_id device,
                              cl_program_build_info param_name,
                              size_t param_value_size,
                              void *param_value,
                              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)

```
status =
  clBuildProgram(program, DEVICENUM, GPU, NULL,
                 NULL, NULL);
if (status != CL_SUCCESS) {
  char log[MAXLOG];
  size_t logLength;
  for (int device = 0; device < DEVICENUM; device++) {
    clGetProgramBuildInfo(program, GPU[device],
                          CL_PROGRAM_BUILD_LOG,
                          MAXLOG, log, &logLength);
    puts(log);
  }
  exit(-1);
}
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 / 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)

```
cl_mem bufferA[DEVICENUM];
for (int device = 0; device < DEVICENUM; device++) {
  bufferA[device] =
    clCreateBuffer(context,
      CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
      ITEMPERDEVICE * sizeof(cl_uint),
      ((cl_uint *)A) + device * ITEMPERDEVICE,
      &status);
  assert(status == CL_SUCCESS);
}
```

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)

```
  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);
```

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)

```
cl_mem bufferC[DEVICENUM];
for (int device = 0; device < DEVICENUM; device++) {
  bufferC[device] =
    clCreateBuffer(context,
      CL_MEM_WRITE_ONLY | CL_MEM_USE_HOST_PTR,
      ITEMPERDEVICE * sizeof(cl_uint),
      ((cl_uint *) C) + device * ITEMPERDEVICE,
      &status);
  assert(status == CL_SUCCESS);
}
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 / 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)

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

Set Argument

- We will launch `DEVICENUM` kernels. Each of them will compute `N / 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)

```
  for (int device = 0; device < DEVICENUM; device++) {
    status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
                            (void*)(&bufferA[device]));
    assert(status == CL_SUCCESS);
    status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
                            (void*)&bufferB);
    assert(status == CL_SUCCESS);
    status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
                            (void*)(&bufferC[device]));
    assert(status == CL_SUCCESS);
    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)

```
    status =
      clEnqueueNDRangeKernel(commandQueue[device],
                             kernel, 2, NULL,
                             globalThreads, localThreads,
                             0, NULL, &(events[device]));
    assert(status == CL_SUCCESS);
  }
```

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)

```
  clWaitForEvents(DEVICENUM, events);
  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)

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

Kernel

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

```
__kernel void mul(__global int A[N/DEVICENUM][N],
                  __global int B[N][N],
                  __global int C[N/DEVICENUM][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];
```

NDRange

- We declare A and C as `N / 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 runs *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)

```
  cl_ulong base;
  status =
    clGetEventProfilingInfo(events[0],
      CL_PROFILING_COMMAND_QUEUED,
      sizeof(cl_ulong), &base, NULL);
  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)

```
  for (int device = 0; device < DEVICENUM; device++) {
    cl_ulong timeEnterQueue, timeSubmit, timeStart,
      timeEnd;
    status =
      clGetEventProfilingInfo(events[device],
        CL_PROFILING_COMMAND_QUEUED,
        sizeof(cl_ulong), &timeEnterQueue, NULL);
    assert(status == CL_SUCCESS);
    status =
      clGetEventProfilingInfo(events[device],
        CL_PROFILING_COMMAND_SUBMIT,
        sizeof(cl_ulong), &timeSubmit, NULL);
    assert(status == CL_SUCCESS);
```

Get Time

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

```
    status =
      clGetEventProfilingInfo(events[device],
        CL_PROFILING_COMMAND_START,
        sizeof(cl_ulong), &timeStart, NULL);
    assert(status == CL_SUCCESS);
    status =
      clGetEventProfilingInfo(events[device],
        CL_PROFILING_COMMAND_END,
         sizeof(cl_ulong), &timeEnd, NULL);
    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)

```
    printf("\nkernel entered queue at %f\n",
           (timeEnterQueue - base) / NANO2SECOND);
    printf("kernel submitted to device at %f\n",
           (timeSubmit - base) / NANO2SECOND);
    printf("kernel started at %f\n",
           (timeStart - base) / NANO2SECOND);
    printf("kernel ended  at %f\n",
           (timeEnd - base) / NANO2SECOND);
    printf("kernel queued time %f seconds\n",
           (timeSubmit - timeEnterQueue) / NANO2SECOND);
    printf("kernel submission time %f seconds\n",
           (timeStart - timeSubmit) / NANO2SECOND);
    printf("kernel execution time %f seconds\n",
         (timeEnd - timeStart) / NANO2SECOND);
  }
```

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)

```
#define CL_USE_DEPRECATED_OPENCL_2_0_APIS
#include <stdio.h>
#include <assert.h>
#include <CL/cl.h>
#define N (65536 * 4)
#define MAXGPU 10
#define MAXK 1024
#define MAXLOG 4096
#define DEVICENUM 3
#define NANO2SECOND 1000000000.0

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)

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

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)

```
  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_READ_WRITE,
                   N * sizeof(cl_uint), NULL, &status);
  assert(status == CL_SUCCESS);
```

Buffers

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

Example 23: (vectorAdd-dependency.c)

```
  cl_mem bufferD =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                   N * sizeof(cl_uint), D, &status);
  assert(status == CL_SUCCESS);
  cl_mem bufferE =
    clCreateBuffer(context,
                   CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR,
                   N * sizeof(cl_uint), E, &status);
  assert(status == CL_SUCCESS);
  cl_mem bufferF =
    clCreateBuffer(context,
                   CL_MEM_READ_WRITE,
                   N * sizeof(cl_uint), NULL, &status);
  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)

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

Discussion

- What are the characteristic 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)

```
  size_t globalThreads[] = {(size_t)N};
  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)

```
  cl_event events[3];
  status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
                          (void*)&bufferA);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
                          (void*)&bufferB);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
                          (void*)&bufferC);
  assert(status == CL_SUCCESS);

  status =
    clEnqueueNDRangeKernel(commandQueue[0], kernel, 1, NULL,
                           globalThreads, localThreads,
                           0, NULL, &(events[0]));
  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)

```
  status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
                          (void*)&bufferD);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
                          (void*)&bufferE);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
                          (void*)&bufferF);
  assert(status == CL_SUCCESS);
  status =
    clEnqueueNDRangeKernel(commandQueue[1], kernel, 1, NULL,
                           globalThreads, localThreads,
                           0, NULL, &(events[1]));
  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)

```
  status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
                          (void*)&bufferC);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
                          (void*)&bufferF);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
                          (void*)&bufferG);
  assert(status == CL_SUCCESS);
  status =
    clEnqueueNDRangeKernel(commandQueue[2], kernel, 1, NULL,
                           globalThreads, localThreads,
                           0, NULL, &(events[2]));
  assert(status == CL_SUCCESS);
```

Wait for Events

- We wait for all three kernel to finish.

Example 29: (vectorAdd-dependency.c)

```
  status = clWaitForEvents(DEVICENUM, events);
  assert(status == CL_SUCCESS);
  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

```
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);
```

$G = C + F$

- 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)

```
  status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
                          (void*)&bufferC);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
                          (void*)&bufferF);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
                          (void*)&bufferG);
  assert(status == CL_SUCCESS);
  status =
    clEnqueueNDRangeKernel(commandQueue[2], kernel, 1, NULL,
                           globalThreads, localThreads,
                           2, events, &(events[2]));
  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.

flFinish

- 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)

```
  clFinish(commandQueue[0]);
  clFinish(commandQueue[1]);
  status = clSetKernelArg(kernel, 0, sizeof(cl_mem),
                          (void*)&bufferC);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 1, sizeof(cl_mem),
                          (void*)&bufferF);
  assert(status == CL_SUCCESS);
  status = clSetKernelArg(kernel, 2, sizeof(cl_mem),
                          (void*)&bufferG);
  assert(status == CL_SUCCESS);
  status =
    clEnqueueNDRangeKernel(commandQueue[2], kernel, 1, NULL,
                           globalThreads, localThreads,
                           0, NULL, &(events[2]));
  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)

```
  cl_device_id GPU[MAXGPU];
  cl_uint GPU_id_got;
  status =
    clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_GPU,
                   MAXGPU, GPU, &GPU_id_got);
  assert(status == CL_SUCCESS && GPU_id_got >= 1);
  printf("There are %d GPU devices\n", GPU_id_got);
  cl_uint unit;
  status =
    clGetDeviceInfo(GPU[0], CL_DEVICE_MAX_COMPUTE_UNITS,
                    sizeof(cl_uint), &unit, NULL);
  assert(status == CL_SUCCESS);
  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-grouopsize.dat`.

Example 34: (vectorAdd-groupsize.c)

```c
  size_t workItem[] = {(size_t)N};
  FILE *timefp = fopen("vectorAdd-groupsize.dat", "w");
  assert(timefp != NULL);
  for (int groupSize = 1; groupSize <= 256; groupSize *= 2) {
    cl_event event;
    size_t localSize[1];
    localSize[0] = groupSize;
    status =
      clEnqueueNDRangeKernel(commandQueue, kernel, 1, NULL,
                             workItem, localSize,
                             0, NULL, &event);
    assert(status == CL_SUCCESS);
    /* waitforevent */
    status = clWaitForEvents(1, &event);
    assert(status == CL_SUCCESS);
    printf("The kernel with group size %d completes.\n",
           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`.

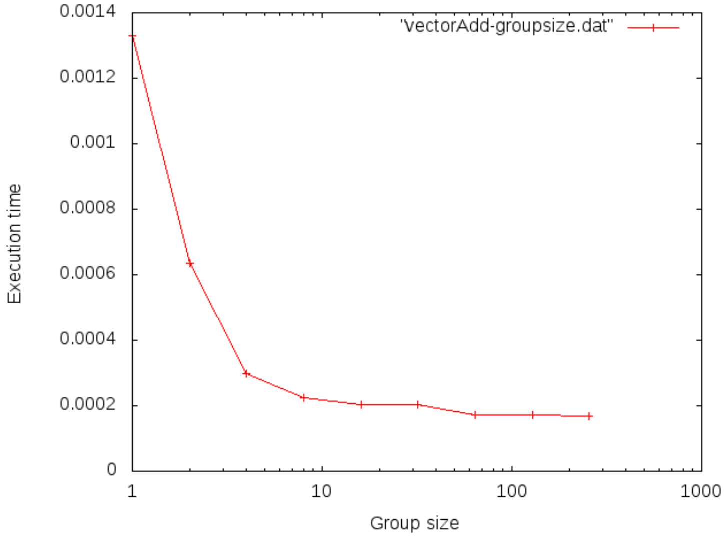
Example 35: (vectorAdd-groupsize.c)

```
    printf("\nkernel entered queue at %f\n",
           (timeEnterQueue - base) / NANO2SECOND);
    printf("kernel submitted to device at %f\n",
           (timeSubmit - base) / NANO2SECOND);
    printf("kernel started at %f\n",
           (timeStart - base) / NANO2SECOND);
    printf("kernel ended  at %f\n",
           (timeEnd - base) / NANO2SECOND);
    printf("kernel queued time %f seconds\n",
           (timeSubmit - timeEnterQueue) / NANO2SECOND);
    printf("kernel submission time %f seconds\n",
           (timeStart - timeSubmit) / NANO2SECOND);
    printf("kernel execution time %f seconds\n",
           (timeEnd - timeStart) / NANO2SECOND);
    fprintf(timefp, "%d %f\n", groupSize,
            (timeEnd - timeStart) / NANO2SECOND);
  }
```

Demonstration

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

Execution Time



Discussion

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