The OpenCL Programming Book

4.3 Calling the Kernel

| Data Parallelism and Task Parallelism

As stated in the "1-3-3 Types of Parallelism" section, parallelizable code is either "Data Parallel" or "Task Parallel". In OpenCL, the difference between the two is whether the same kernel or different kernels are executed in parallel. The difference becomes obvious in terms of execution time when executed on the GPU.

At present, most GPUs contain multiple processors, but hardware such as instruction fetch and program counters are shared across the processors. For this reason, the GPUs are incapable of running different tasks in parallel.

As shown in Figure 4.2, when multiple processors perform the same task, the number of tasks equal to the number of processors can be performed at once. Figure 4.3 shows the case when multiple tasks are scheduled to be performed in parallel on the GPU. Since the processors can only process the same set of instructions across the cores, the processors scheduled to process Task B must be in idle mode until Task A is finished.

Figure 4.2: Efficient use of the GPU

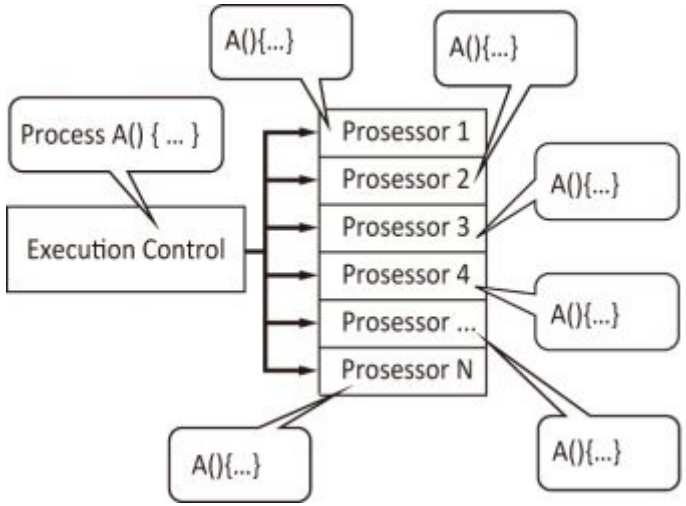
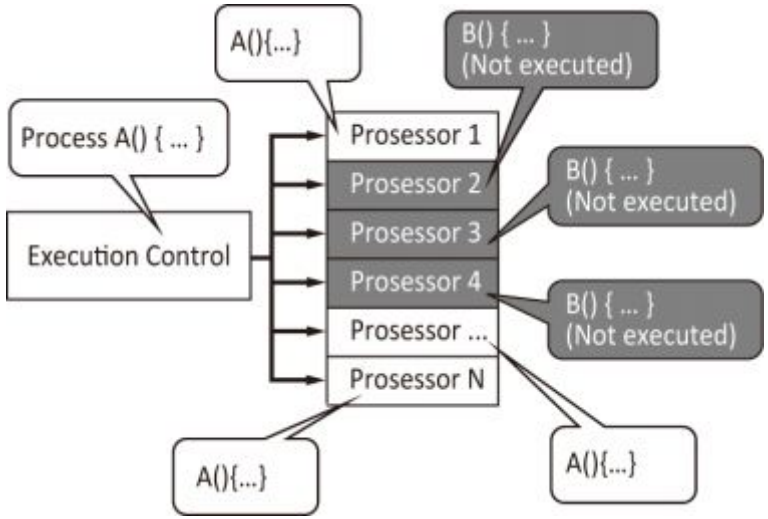


Figure 4.3: Inefficient use of the GPU



For data parallel tasks suited for a device like the GPU, OpenCL provides an API to run the same kernel across multiple processors, called clEnqueueNDRangeKernel(). When developing an application, the task type and the hardware need to be considered wisely, and use the appropriate API function.

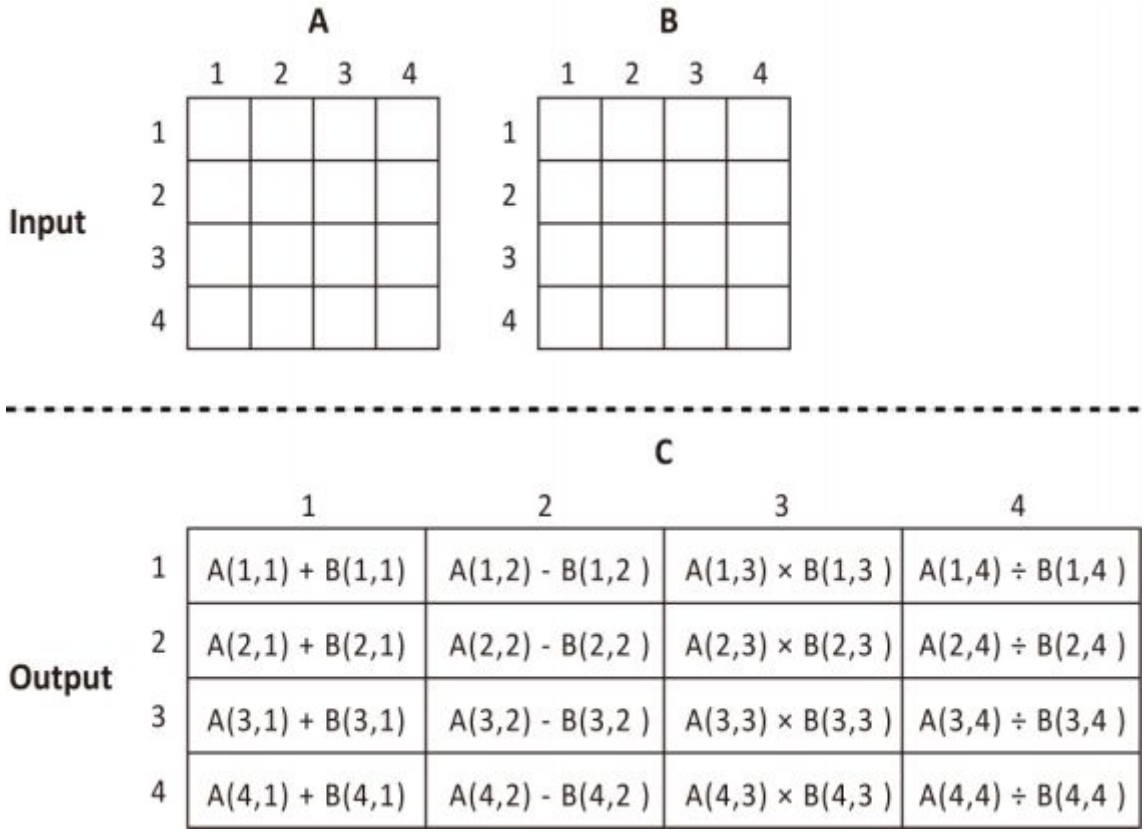
» Mersenne Twister

› Notes

This section will use vector-ized arithmetic operation to explain the basic method of implementations for data parallel and task parallel commands. The provided sample code is meant to illustrate the parallelization concepts.

The sample code performs the basic arithmetic operations, which are addition, subtraction, multiplication and division, between float values. The overview is shown in Figure 4.4.

Figure 4.4: Basic arithmetic operations between floats



As the figure shows, the input data consists of 2 sets of 4x4 matrices A and B. The output data is a 4x4 matrix C.

We will first show the data-parallel implementation (List 4.8, List 4.9). This program treats each row of data as one group in order to perform the computation.

List 4.8: Data parallel model - kernel dataParallel.cl

```
__kernel void dataParallel(__global float* A, __global float* B, __global float* C)
{
        int base = 4*get_global_id(0);
        C[base+0] = A[base+0] + B[base+0];
        C[base+1] = A[base+1] - B[base+1];
        C[base+2] = A[base+2] * B[base+2];
        C[base+3] = A[base+3] / B[base+3];
}
```

List 4.9: Data parallel model - host dataParallel.c

```
#include <stdio.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

#define MAX_SOURCE_SIZE (0x100000)

int main()
{
        cl_platform_id platform_id = NULL;
        cl_device_id device_id = NULL;
        cl_context context = NULL;
        cl_command_queue command_queue = NULL;
        cl_mem Amobj = NULL;
        cl_mem Bmobj = NULL;
        cl_mem Cmobj = NULL;
        cl_program program = NULL;
        cl_kernel kernel = NULL;
        cl_uint ret_num_devices;
        cl_uint ret_num_platforms;
```

```
25.         cl_int ret;
26.
27.         int i, j;
28.         float *A;
29.         float *B;
30.         float *C;
31.
32.         A = (float *)malloc(4*4*sizeof(float));
33.         B = (float *)malloc(4*4*sizeof(float));
34.         C = (float *)malloc(4*4*sizeof(float));
35.
36.         FILE *fp;
37.         const char fileName[] = "./dataParallel.cl";
38.         size_t source_size;
39.         char *source_str;
40.
41.         /* Load kernel source file */
42.         fp = fopen(fileName, "r");
43.         if (!fp) {
44.                 fprintf(stderr, "Failed to load kernel.\n");
45.                 exit(1);
46.         }
47.         source_str = (char *)malloc(MAX_SOURCE_SIZE);
48.         source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
49.         fclose(fp);
50.
51.         /* Initialize input data */
52.         for (i=0; i < 4; i++) {
53.                 for (j=0; j < 4; j++) {
54.                         A[i*4+j] = i*4+j+1;
55.                         B[i*4+j] = j*4+i+1;
56.                 }
57.         }
58.
59.         /* Get Platform/Device Information
60.         ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
61.         ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &r
    et_num_devices);
62.
```

```
63.         /* Create OpenCL Context */
64.         context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
65.
66.         * Create command queue */
67.         command_queue = clCreateCommandQeue(context, device_id, 0, &ret);
68.
69.         * Create Buffer Object */
70.         Amobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 4*4*sizeof(float), NULL,
     &ret);
71.         Bmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 4*4*sizeof(float), NULL,
     &ret);
72.         Cmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 4*4*sizeof(float), NULL,
     &ret);
73.
74.         /* Copy input data to the memory buffer */
75.         ret = clEnqueueWriteBuffer(command_queue, Amobj, CL_TRUE, 0, 4*4*sizeof(flo
    at), A, 0, NULL, NULL);
76.         ret = clEnqueueWriteBuffer(command_queue, Bmobj, CL_TRUE, 0, 4*4*sizeof(flo
    at), B, 0, NULL, NULL);
77.
78.         /* Create kernel program from source file*/
79.         program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
     (const size_t *)&source_size, &ret);
80.         ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
81.
82.         /* Create data parallel OpenCL kernel */
83.         kernel = clCreateKernel(program, "dataParallel", &ret);
84.
85.         /* Set OpenCL kernel arguments */
86.         ret = clSetKernelArg(kernel, 0, sizeof(cl_mem), (void *)&Amobj);
87.         ret = clSetKernelArg(kernel, 1, sizeof(cl_mem), (void *)&Bmobj);
88.         ret = clSetKernelArg(kernel, 2, sizeof(cl_mem), (void *)&Cmobj);
89.
90.         size_t global_item_size = 4;
91.         size_t local_item_size = 1;
92.
93.         /* Execute OpenCL kernel as data parallel */
94.         ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
95.         &global_item_size, &local_item_size, 0, NULL, NULL);
```

```
96.
97.         /* Transfer result to host */
98.         ret = clEnqueueReadBuffer(command_queue, Cmobj, CL_TRUE, 0, 4*4*sizeof(floa
    t), C, 0, NULL, NULL);
99.
100.        /* Display Results */
101.        for (i=0; i < 4; i++) {
102.                for (j=0; j < 4; j++) {
103.                        printf("%7.2f ", C[i*4+j]);
104.                }
105.                printf("\n");
106.        }
107.
108.
109.        /* Finalization */
110.        ret = clFlush(command_queue);
111.        ret = clFinish(command_queue);
112.        ret = clReleaseKernel(kernel);
113.        ret = clReleaseProgram(program);
114.        ret = clReleaseMemObject(Amobj);
115.        ret = clReleaseMemObject(Bmobj);
116.        ret = clReleaseMemObject(Cmobj);
117.        ret = clReleaseCommandQueue(command_queue);
118.        ret = clReleaseContext(context);
119.
120.        free(source_str);
121.
122.        free(A);
123.        free(B);
124.        free(C);
125.
126.        return 0;
127. }
```

Next, we will show the task parallel version of the same thing (List 4.10, List 4.11). In this sample, the tasks are grouped according to the type of arithmetic operation being performed.

List 4.10: Task parallel model - kernel taskParallel.cl

```
1.  __kernel void taskParallelAdd(__global float* A, __global float* B, __global float*
     C)
2.  {
3.          int base = 0;
4.
5.          C[base+0] = A[base+0] + B[base+0];
6.          C[base+4] = A[base+4] + B[base+4];
7.          C[base+8] = A[base+8] + B[base+8];
8.          C[base+12] = A[base+12] + B[base+12];
9.  }
10.
11. __kernel void taskParallelSub(__global float* A, __global float* B, __global float*
     C)
12. {
13.         int base = 1;
14.
15.         C[base+0] = A[base+0] - B[base+0];
16.         C[base+4] = A[base+4] - B[base+4];
17.         C[base+8] = A[base+8] - B[base+8];
18.         C[base+12] = A[base+12] - B[base+12];
19. }
20.
21. __kernel void taskParallelMul(__global float* A, __global float* B, __global float*
     C)
22. {
23.         int base = 2;
24.
25.         C[base+0] = A[base+0] * B[base+0];
26.         C[base+4] = A[base+4] * B[base+4];
27.         C[base+8] = A[base+8] * B[base+8];
28.         C[base+12] = A[base+12] * B[base+12];
29. }
30.
31. __kernel void taskParallelDiv(__global float* A, __global float* B, __global float*
     C)
32. {
33.         int base = 3;
34.
35.         C[base+0] = A[base+0] / B[base+0];
```

```
        C[base+4] = A[base+4] / B[base+4];
        C[base+8] = A[base+8] / B[base+8];
        C[base+12] = A[base+12] / B[base+12];
}
```

List 4.11: Task parallel model - host taskParallel.c

```
#include <stdio.h>
#include <stdlib.h>

#ifdef __APPLE__
#include <OpenCL/opencl.h>
#else
#include <CL/cl.h>
#endif

#define MAX_SOURCE_SIZE (0x100000)

int main()
{
        cl_platform_id platform_id = NULL;
        cl_device_id device_id = NULL;
        cl_context context = NULL;
        cl_command_queue command_queue = NULL;
        cl_mem Amobj = NULL;
        cl_mem Bmobj = NULL;
        cl_mem Cmobj = NULL;
        cl_program program = NULL;
        cl_kernel kernel[4] = {NULL, NULL, NULL, NULL};
        cl_uint ret_num_devices;
        cl_uint ret_num_platforms;
        cl_int ret;

        int i, j;
        float* A;
        float* B;
        float* C;

```

```
32.         A = (float*)malloc(4*4*sizeof(float));
33.         B = (float*)malloc(4*4*sizeof(float));
34.         C = (float*)malloc(4*4*sizeof(float));
35.
36.
37.         FILE *fp;
38.         const char fileName[] = "./taskParallel.cl";
39.         size_t source_size;
40.         char *source_str;
41.
42.         /* Load kernel source file */
43.         fp = fopen(fileName, "rb");
44.         if (!fp) {
45.                 fprintf(stderr, "Failed to load kernel.\n");
46.                 exit(1);
47.         }
48.         source_str = (char *)malloc(MAX_SOURCE_SIZE);
49.         source_size = fread(source_str, 1, MAX_SOURCE_SIZE, fp);
50.         fclose(fp);
51.
52.         /* Initialize input data */
53.         for (i=0; i < 4; i++) {
54.                 for (j=0; j < 4; j++) {
55.                         A[i*4+j] = i*4+j+1;
56.                         B[i*4+j] = j*4+i+1;
57.                 }
58.         }
59.
60.         /* Get platform/device information */
61.         ret = clGetPlatformIDs(1, &platform_id, &ret_num_platforms);
62.         ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_DEFAULT, 1, &device_id, &r
    et_num_devices);
63.
64.         /* Create OpenCL Context */
65.         context = clCreateContext(NULL, 1, &device_id, NULL, NULL, &ret);
66.
67.         /* Create command queue */
```

```
68.         command_queue = clCreateCommandQueue(context, device_id, CL_QUEUE_OUT_OF_OR
    DER_EXEC_MODE_ENABLE, &ret);
69.
70.         /* Create buffer object */
71.         Amobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 4*4*sizeof(float), NULL,
     &ret);
72.         Bmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 4*4*sizeof(float), NULL,
     &ret);
73.         Cmobj = clCreateBuffer(context, CL_MEM_READ_WRITE, 4*4*sizeof(float), NULL,
     &ret);
74.
75.         /* Copy input data to memory buffer */
76.         ret = clEnqueueWriteBuffer(command_queue, Amobj, CL_TRUE, 0, 4*4*sizeof(flo
    at), A, 0, NULL, NULL);
77.         ret = clEnqueueWriteBuffer(command_queue, Bmobj, CL_TRUE, 0, 4*4*sizeof(flo
    at), B, 0, NULL, NULL);
78.
79.         /* Create kernel from source */
80.         program = clCreateProgramWithSource(context, 1, (const char **)&source_str,
     (const size_t *)&source_size, &ret);
81.         ret = clBuildProgram(program, 1, &device_id, NULL, NULL, NULL);
82.
83.         /* Create task parallel OpenCL kernel */
84.         kernel[0] = clCreateKernel(program, "taskParallelAdd", &ret);
85.         kernel[1] = clCreateKernel(program, "taskParallelSub", &ret);
86.         kernel[2] = clCreateKernel(program, "taskParallelMul", &ret);
87.         kernel[3] = clCreateKernel(program, "taskParallelDiv", &ret);
88.
89.         /* Set OpenCL kernel arguments */
90.         for (i=0; i < 4; i++) {
91.                 ret = clSetKernelArg(kernel[i], 0, sizeof(cl_mem), (void *)&Amobj);
92.                 ret = clSetKernelArg(kernel[i], 1, sizeof(cl_mem), (void *)&Bmobj);
93.                 ret = clSetKernelArg(kernel[i], 2, sizeof(cl_mem), (void *)&Cmobj);
94.         }
95.
96.         /* Execute OpenCL kernel as task parallel */
```

```
 97.         for (i=0; i < 4; i++) {
 98.                 ret = clEnqueueTask(command_queue, kernel[i], 0, NULL, NULL);
 99.         }
100.
101.         /* Copy result to host */
102.         ret = clEnqueueReadBuffer(command_queue, Cmobj, CL_TRUE, 0, 4*4*sizeof(floa
     t), C, 0, NULL, NULL);
103.
104.         /* Display result */
105.         for (i=0; i < 4; i++) {
106.                 for (j=0; j < 4; j++) {
107.                         printf("%7.2f ", C[i*4+j]);
108.                 }
109.                 printf("\n");
110.         }
111.
112.         /* Finalization */
113.         ret = clFlush(command_queue);
114.         ret = clFinish(command_queue);
115.         ret = clReleaseKernel(kernel[0]);
116.         ret = clReleaseKernel(kernel[1]);
117.         ret = clReleaseKernel(kernel[2]);
118.         ret = clReleaseKernel(kernel[3]);
119.         ret = clReleaseProgram(program);
120.         ret = clReleaseMemObject(Amobj);
121.         ret = clReleaseMemObject(Bmobj);
122.         ret = clReleaseMemObject(Cmobj);
123.         ret = clReleaseCommandQueue(command_queue);
124.         ret = clReleaseContext(context);
125.
126.         free(source_str);
127.
128.         free(A);
129.         free(B);
130.         free(C);
131.
132.         return 0;
133. }
```

As you can see, the source codes are very similar. The only differences are in the kernels themselves, and the way to execute these kernels. In the data parallel model, the 4 arithmetic operations are grouped as one set of commands in a kernel, while in the task parallel model, 4 different kernels are implemented for each type of arithmetic operation.

At a glance, it may seem that since the task parallel model requires more code, that it also must perform more operations. However, regardless of which model is used for this problem, the number of operations being performed by the device is actually the same. Despite this fact, some problems are easier, and performance can vary by choosing one over the other, so the parallelization model must be considered wisely in the planning stage of the application.

We will now walkthrough the source code for the data parallel model.

```
1.  __kernel void dataParallel(__global float * A, __global float * B, __global float *
    C)
```

When the data parallel task is queued, work-items are created. Each of these work-items executes the same kernel in parallel.

```
1.          int base = 4*get_global_id(0);
```

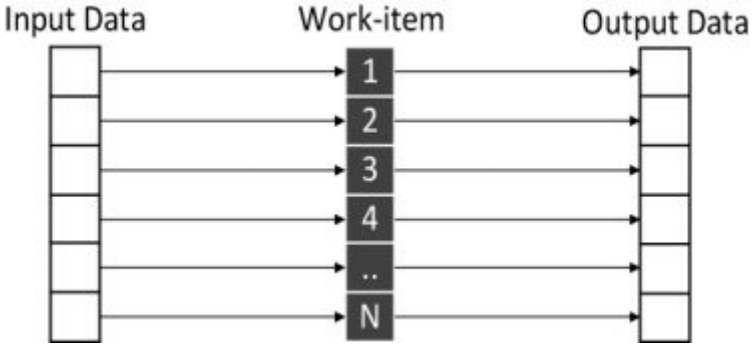
The get_global_id(0) gets the global work-item ID, which is used to decide the data to process, so that each work-items can process different sets of data in parallel. In general, data parallel processing is done using the following steps.

1. Get work-item ID

2. Process the subset of data corresponding to the work-item ID

A block diagram of the process is shown in Figure 4.5.

Figure 4.5: Block diagram of the data-parallel model in relation to work-items



In this example, the global work-item is multiplied by 4 and stored in the variable "base". This value is used to decide which element of the array A and B gets processed.

```
        C[base+0] = A[base+0] + B[base+0];
        C[base+1] = A[base+1] - B[base+1];
        C[base+2] = A[base+2] * B[base+2];
        C[base+3] = A[base+3] / B[base+3];

```

Since each work-item have different IDs, the variable "base" also have a different value for each work-item, which keeps the work-items from processing the same data. In this way, large amount of data can be processed concurrently.

We have discussed that numerous work items get created, but we have not touched upon how to decide the number of work-items to create. This is done in the following code segment from the host code.

```
        size_t global_item_size = 4;
        size_t local_item_size = 1;

        /* Execute OpenCL kernel as data parallel */
        ret = clEnqueueNDRangeKernel(command_queue, kernel, 1, NULL,
                &global_item_size, &local_item_size, 0, NULL, NULL);
```

The clEnqueueNDRangeKernel() is an OpenCL API command used to queue data parallel tasks. The 5th and 6th arguments determine the work-item size. In this case, the global_item_size is set to 4, and the local_item_size is set to 1. The overall steps are summarized as follows.

1. Create work-items on the host

2. Process data corresponding to the global work item ID on the kernel

We will now walkthrough the source code for the task parallel model. In this model, different kernels are allowed to be executed in parallel. Note that different kernels are implemented for each of the 4 arithmetic operations.

```
1.          /* Execute OpenCL kernel as task parallel */
2.          for (i=0; i < 4; i++) {
3.                  ret = clEnqueueTask(command_queue, kernel[i], 0, NULL, NULL);
4.           }
```

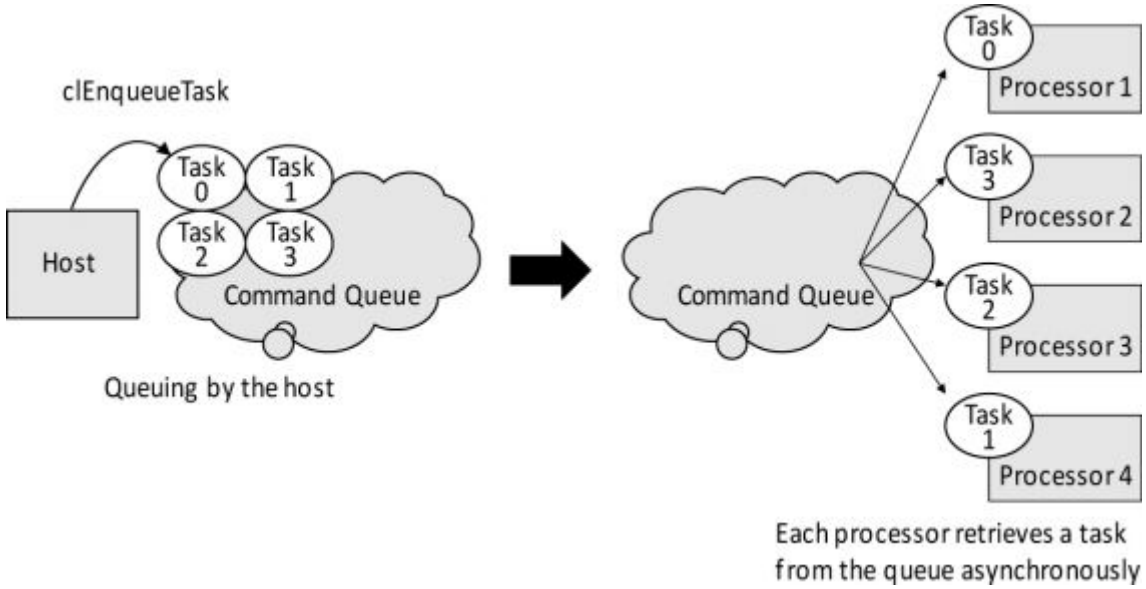
The above code segment queues the 4 kernels.

In OpenCL, in order to execute a task parallel process, the out-of-order mode must be enabled when the command queue is created. Using this mode, the queued task does not wait until the previous task is finished if there are idle compute units available that can be executing that task.

```
1.  /* Create command queue */
2.  command_queue = clCreateCommandQueue(context, device_id, CL_QUEUE_OUT_OF_ORDER_EXE
C_MODE_ENABLE, &ret);
```

The block diagram of the nature of command queues and parallel execution are shown in Figure 4.6.

Figure 4.6: Command queues and parallel execution



The clEnqueueTask() is used as an example in the above figure, but a similar parallel processing could take place for other combinations of enqueue-functions, such as clEnqueueNDRangeKernel(), clEnqueueReadBuffer(), and clEnqueueWriteBuffer(). For example, since PCI Express supports simultaneous bi-directional memory transfers, queuing the clEnqueueReadBuffer() and clEnqueueWriteBuffer() can execute read and write commands simultaneously, provided that the commands are being performed by different processors. In the above diagram, we can expect the 4 tasks to be executed in parallel, since they are being queued in a command queue that has out-of-execution enabled.

Work Group

The last section discussed the concept of work-items. This section will introduce the concept of work-groups.

Work-items are grouped together into work-groups. A work-group must consist of at least 1 work-item, and the maximum is dependent on the platform. The work-items within a work-group can synchronize, as well as share local memory with each other.

In order to implement a data parallel kernel, the number of work-groups must be specified in addition to the number of work-items. This is why 2 different parameters had to be sent to the clEnqueueNDRangeKernel() function.

```
1.          size_t global_item_size = 4;
2.          size_t local_item_size = 1;
```

The above code means that each work-group is made up of 1 work-item, and that there are 4 work-groups to be processed.

The number of work-items per work-group is consistent throughout every work-group. If the number of work-items cannot be divided evenly among the work-groups, clEnqueueNDRangeKernel() fails, returning the error value CL_INVALID_WORK_GROUP_SIZE.

The code List 4.9 only used the global work-item ID to process the kernel, but it is also possible to retrieve the local work-item ID corresponding to the work-group ID. The relationship between the global work-item ID, local work-item ID, and the work-group ID are shown below in Figure 4.7. The function used to retrieve these ID's from within the kernel are shown in Table 4.1.

Figure 4.7: Work-group ID and Work-item ID

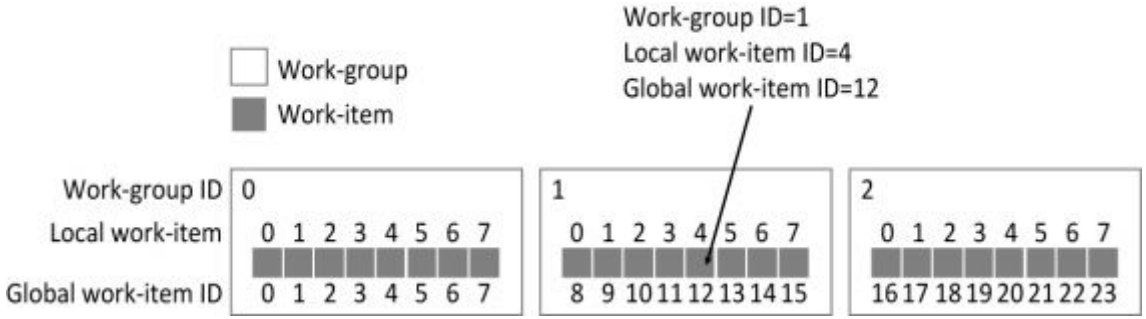
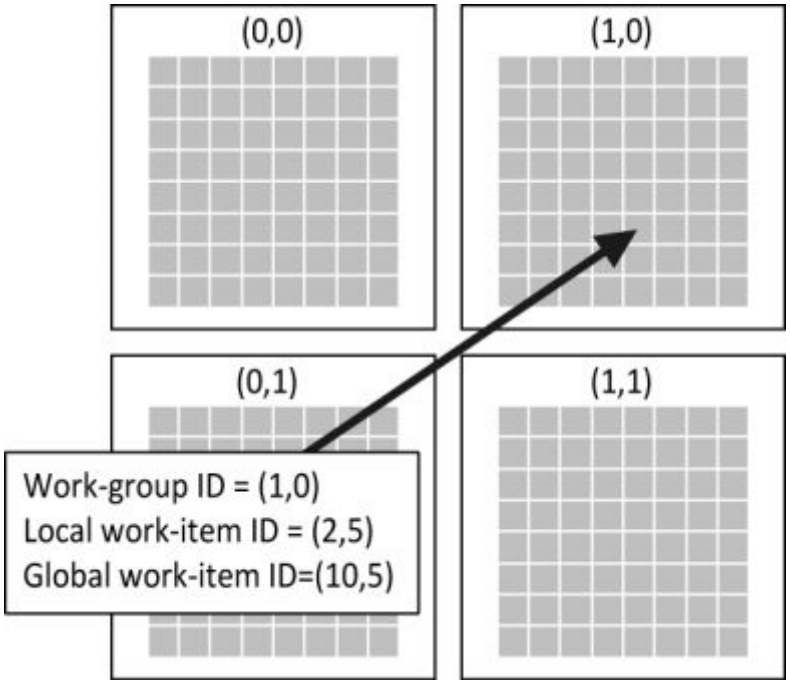


Table 4.1: Functions used to retrieve the ID's

Function	Retrieved value
get_group_id	Work-group ID
get_global_id	Global work-item ID
get_local_id	Local work-item ID

Since 2-D images or 3-D spaces are commonly processed, the work-items and work-groups can be specified in 2 or 3 dimensions. Figure 4.8 shows an example where the work-group and the work-item are defined in 2-D.

Figure 4.8: Work-group and work-item defined in 2-D



Since the work-group and the work-items can have up to 3 dimensions, the ID's that are used to index them also have 3 dimensions. The get_group_id(), get_global_id(), get_local_id() can each take an argument between 0 and 2, each corresponding to the dimension. The ID's for the work-items in Figure 4.8 are shown below in Table 4.2.

Table 4.2: The ID's of the work-item in Figure 4.8

Call	Retrieved ID
get_group_id(0)	1

get_group_id(1)	0
get_global_id(0)	10
get_global_id(1)	5
get_local_id(0)	2
get_local_id(1)	5

Note that the index space dimension and the number of work-items per work-group can vary depending on the device. The maximum index space dimension can be obtained using the clGetDeviceInfo() function to get the value of CL_DEVICE_WORK_ITEM_DIMENSIONS, and the maximum number of work-items per work-group can be obtained by getting the value of CL_DEVICE_IMAGE_SUPPORT. The data-type of CL_DEVICE_MAX_WORK_ITEM is cl_uint, and for CL_DEVICE_IMAGE_SUPPORT, it is an array of type size_t.

Also, as of this writing (12/2009), the OpenCL implementation for the CPU on Mac OS X only allows 1 work-item per work-group.

Task Parallelism and Event Object

The tasks placed in the command-queue are executed in parallel, but in cases where different tasks have data dependencies, they need to be executed sequentially. In OpenCL, the execution order can be set using an event object.

An event object contains information about the execution status of queued commands. This object is returned on all commands that start with "clEnqueue". In order to make sure task A executes before task B, the event object returned when task A is queued can be one of the inputs to task B, which keeps task B from executing until task A is completed.

For example, the function prototype for clPEnqueueTask() is shown below.

```
1. cl_int clEnqueueTask (cl_command_queue command_queue,
2. cl_kernel kernel,
3. cl_uint num_events_in_wait_list,
4. const cl_event *event_wait_list,
```

```
5.  cl_event *event)
```

The 4th argument is a list of events to be processed before this task can be run, and the 3rd argument is the number of events on the list. The 5th parameter is the event object returned by this task when it is placed in the queue.

List 4.12 shows an example of how the event objects can be used. In this example, kernel_A, kernel_B, kernel_C, kernel_D can all be executed in any order, but these must be completed before kernel_E is executed.

List 4.12: Event object usage example

```
1.  clEnqueue events[4];
2.  clEnqueueTask(command_queue, kernel_A, 0, NULL, &events[0]);
3.  clEnqueueTask(command_queue, kernel_B, 0, NULL, &events[1]);
4.  clEnqueueTask(command_queue, kernel_C, 0, NULL, &events[2]);
5.  clEnqueueTask(command_queue, kernel_D, 0, NULL, &events[3]);
6.  clEnqueueTask(command_queue, kernel_E, 4, events, NULL);
```

---COLUMN: Relationship between execution order and parallel execution---

In OpenCL, the tasks placed in the command queue are executed regardless of the order in which it is placed in the queue. In the specification sheet for parallel processors, there is usually a section on "order". When working on parallel processing algorithms, the concept of "order" and "parallel" need to be fully understood.

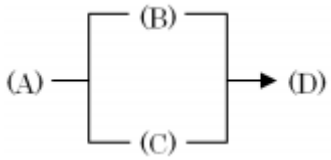
First, we will discuss the concept of "order". As an example, let's assume the following 4 tasks are performed sequentially in the order shown below.

(A) → (B) → (C) → (D)

Now, we will switch the order in which task B and C are performed.

(A) → (C) → (B) → (D)

If tasks B and C are not dependent on each other, the two sets of tasks will result in the same output. If they are not, the two sets of tasks will not yield the correct result. This problem of whether a certain set of tasks can be performed in different order is a process-dependent problem.



Parallel processing is a type of optimization that deals with changing the order of tasks. For example, let's assume tasks B and C are processed in parallel.

In this case, task B may finish before task C, but task D waits for the results of both tasks B and C. This processing is only allowed in the case where the tasks B and C do not depend on each other. Thus, if task C must be performed after task B, then the 2 tasks cannot be processed in parallel.

On the other hand, it may make more sense to implement all the tasks in a single thread. Also, if the tasks are executed on a single-core processor, all the tasks must be performed sequentially. These need to be considered when implementing an algorithm.

In essence, two problems must be solved. One is whether some tasks can be executed out of order, and the other is whether to process the tasks in parallel.

- Tasks must be executed in a specific order = Cannot be parallelized
- Tasks can be executed out of order = Can be parallelized

Specifications are written to be a general reference of the capabilities, and do not deal with the actual implementation. Decision to execute certain tasks in parallel is thus not discussed inside the specification. Instead, it contains information such as how certain tasks are guaranteed to be processed in order.

Explanations on "ordering" is often times difficult to follow (Example: PowerPC's Storage Access Ordering), but often times, treating "order" as the basis when implementing parallel algorithms can clear up existing bottlenecks in the program.

For example, OpenMP's "parallel" construct specifies the code segment to be executed in parallel, but placing this construct in a single-core processor will not process the code segment in parallel. In this case, one may wonder if it meets OpenMP's specification. If you think about the "parallel" construct as something that tells the compiler and the processors that the loops can be executed out-of-order, it clears up the definition, since it means that the ordering can be changed or the tasks can be run parallel, but that they do not have to be.

Previous | Next

About Fixstars

Fixstars Solutions is an innovator in flash storage solutions devoted to “Speed up your Business”. Combining expertise in multi-core processors programming and the use of next generation memory technology, Fixstars provides the best performance and the highest capacity storage solutions to deliver high speed IO as well as power savings to accelerate customers' business in various fields.

Tweets about Fixstars

Tweets about Fixstars

Contact Fixstars

Fixstars Solutions Inc.

9205 Research Drive, 1st Floor, Irvine,
California 92618

View Map

Contact form

Follow us on Twitter