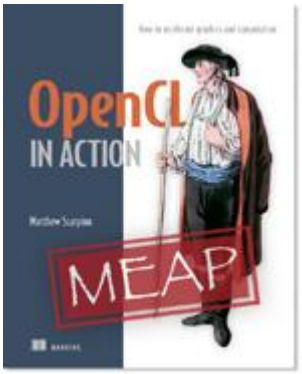
MANNING PUBLICATIONS

Data partitioning

An article from



OpenCL in Action
EARLY ACCESS EDITION

Accelerating Graphics and Computation
Matthew Scarpino
MEAP Began: December 2010
Softbound print: Early 2012 | 475 pages
ISBN: 9781617290176

This article is taken from the book OpenCL in Action. *The author discusses data partitioning, an important element in implementing algorithms with OpenCL due to the size of data that has to be processed.*

If you're implementing an algorithm with OpenCL, you probably have a great deal of data to process. This makes data partitioning an important priority—the better you distribute the processing load, the sooner your computational tasks will be finished.

You already know how to divide data among multiple devices, but you can partition your data even further. Most OpenCL devices contain several processing elements, and with the right code, you can control how much data each processing element receives.

There's only one function to know: `clEnqueueNDRangeKernel`. This is one of the most important functions in the OpenCL API, and it's also one of the most complex. Like `clEnqueueTask`, this places a kernel in a command queue for execution. But unlike `clEnqueueTask`, `clEnqueueNDRangeKernel` allows you to control how the kernel execution is distributed among the device's processing elements. This is shown by its signature, which is as follows:

```
clEnqueueNDRangeKernel(cl_command_queue queue, cl_kernel kernel,
     cl_uint work_dims, const size_t *global_work_offset,
     const size_t *global_work_size, const size_t *local_work_size,
     cl_uint num_events, const cl_event *wait_list, cl_event *event)
```

This is considerably more involved than `clEnqueueTask`. The difference between the two functions is that `clEnqueueNDRangeKernel` accepts four additional arguments:

- `work_dims`—The number of dimensions in the data.
- `global_work_offset`—The global ID offsets in each dimension.
- `global_work_size`—The number of work-items in each dimension.
- `local_work_size`—The number of work-items in a work-group in each dimension.

Don't be concerned if these terms don't make sense just yet. The goal of this article is to explain what they mean and how to configure them so that you can take the best advantage of your hardware.

Loops and work-items

When you have a great deal of data, it's common to iterate through the data using loops. If you're dealing with multidimensional data in regular C/C++, you might use a nested loop such as the following:

```
for(i=0; i<Z; i++) {
   for(j=0; j<Y; j++) {
      for(k=0; k<X; k++) {
         process(point[i][j][k]);
      }
   }
}
```

Loops like this are common but inefficient. The inefficiency arises because each iteration requires a separate comparison and addition. Comparisons are time-consuming on the best of processors, but they're especially slow on dedicated number-crunchers like graphic processor units (GPUs). GPUs excel at performing the same operations over and over again, but they're not efficient when it comes to making decisions. If a GPU has to check a condition and branch, it may take hundreds of cycles before it can get back to crunching numbers at full speed.

One fascinating aspect of OpenCL is that you don't have to configure these loops in your kernel. Instead, your kernel only executes code that would lie inside the innermost loop. We call this individual kernel execution a *work-item*. In the example loop above, the work-item consists of the single function call: `process(point[i][j][k])`.

It's important to understand the difference between kernels and work-items. A kernel identifies a set of tasks to be performed on data. A work-item is a single implementation of the kernel on a specific set of data. For every kernel, there are multiple work-items. In the example above, a kernel might be represented by `process(point[i][j][k])`. A specific implementation of this kernel, such as `process(point[1][2][3])`, would be a work-item.

The array `{i, j, k}` is called the work-item's *global ID*. It uniquely identifies the work-item and allows it to access the data that it's supposed to process. As an example, the following kernel code accesses the elements of the work-item's ID and uses them to process a point.

```
int i = get_global_id(0);
int j = get_global_id(1);
int k = get_global_id(2);
process(point[i][j][k]);
```

Once this work item has executed, a new work item will execute with a different global ID.

The number of elements in a global ID is referred to as the data's *dimensionality*. You configure this by setting the `work_dims` argument of `clEnqueueNDRangeKernel`. The minimum number of dimensions is one and the maximum number depends on the device. To find the maximum number of dimensions, call `clGetDeviceInfo` with the `CL_DEVICE_MAX_WORK_ITEM_DIMENSIONS` parameter. In the `i-j-k` loop above, you would set `work_dims` equal to 3.

Image objects can be two or three dimensional, while buffer objects are accessed in one dimension only. `clEnqueueNDRangeKernel` doesn't care about this distinction. If you're dealing with image objects, you should probably set `work_dims` equal to two or three. But for buffer objects, you can set whatever dimensionality you think best. For a buffer object containing a two-dimensional matrix, such as that shown in figure 1, you might set `work_dims` equal to two.

Figure 1 Transferring buffer object data in rectangles

Work sizes and offsets

The left side of figure 2 depicts a processing loop. The right side presents the *index space* corresponding to the loop. The index space contains all the possible combinations of indices. If there are N indices in a loop, the corresponding index space has N dimensions.

```
for(i=0; i<4; i++) {
   for(j=2; j<8; j++) {
      for(k=3; k<12; k++) {
         process(point[i][j][k]);
      }
   }
}
```

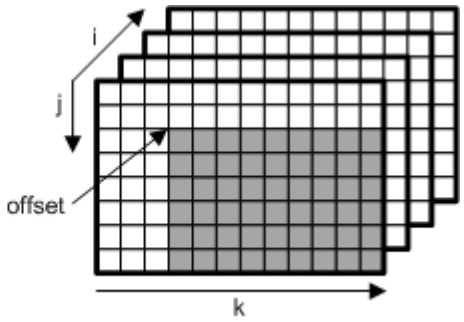


Figure 2 A processing loop and its index space

The `global_work_sizes` argument of `clEnqueueNDRangeKernel` identifies how many work-items need to be processed for each dimension. The inner loop starts at k=3 and proceeds to k=11, so there are 9 work-items to be processed in the k-direction. Similarly, there are 6 work-items to be processed in the j-direction and 4 work-items to be processed in the i-direction. Therefore, you'd set `global_work_sizes` to `{4, 6, 9}`.

When the first work-item starts its execution, we want it to access data corresponding to the index triple `(0, 2, 3)` because these are the initial values of i, j, and k. In other words, we want the first work-item's global ID to equal `{0, 2, 3}`. We specify this in code by setting `global_work_offset` in `clEnqueueNDRangeKernel` to `{0, 2, 3}`.

A simple one-dimensional example

A good way to understand `clEnqueueNDRangeKernel` is to see how it's used in code. Figure 3 presents the multiplication of a vector by a matrix. It also depicts the buffer object containing the matrix data and the manner in the matrix data is partitioned among four work-items.

$$\begin{vmatrix} 0.0 & 2.0 & 4.0 & 6.0 \\ 8.0 & 10.0 & 12.0 & 14.0 \\ 16.0 & 18.0 & 20.0 & 22.0 \\ 24.0 & 26.0 & 28.0 & 30.0 \end{vmatrix} \begin{vmatrix} 0.0 \\ 3.0 \\ 6.0 \\ 9.0 \end{vmatrix} = \begin{vmatrix} 84.0 \\ 228.0 \\ 372.0 \\ 516.0 \end{vmatrix}$$

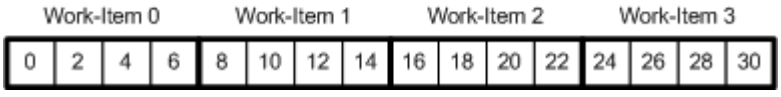


Figure 3 Partitioning data in a matrix-vector multiplication

The matrix-vector multiplication consists of four dot-products, and I've chosen to perform the multiplication using four work-items. This is accomplished with the following code:

```
work_items_per_kernel = 4;

clEnqueueNDRangeKernel(queue, kernel, 1, NULL, &work_items_per_kernel,
      NULL, 0, NULL, NULL);
```

This tells OpenCL that the data to be partitioned has a single dimension and that there are four work-items in the kernel. It sets the global offset to `0` and doesn't create any work-groups.

On the kernel side, each work-item checks its global ID and accesses one row of the matrix. It multiplies this row (1×4) by the vector (4×1) using the `dot` function, and places the result (1×1) in an array position determined by its ID. This is shown in the following code:

```
int i = get_global_id(0);
result[i] = dot(matrix[i], vector[0]);
```

See? No loops. Once one work-item finishes executing, another can proceed without any of the delay associated with `for` statements or similar constructs.

Work-groups and compute units

Work-items may require synchronization to process data effectively. To make this possible, we combine work-items into *work-groups*. Work-items inside a work-group can synchronize their computation using fences and barriers. OpenCL does not support synchronization between work-items in different work-groups.

Like a work-item, each work-group has a unique ID that represents its location in the overall index space. The values used to compute group IDs are determined by the size of the work-group. You configure this size through the `local_work_size` argument of `clEnqueueNDRangeKernel`. The elements in this array identify how many work-items can fit in the work-group in each dimension.

For example, let's make work-groups out of the two-dimensional slices in the index space depicted in figure 2. There are four slices, so we'll have four work-groups. Each group contains six work-items in the j-direction and nine work-items in the k-direction. Therefore, we would set `local_work_size` in `clEnqueueNDRangeKernel` equal to `{0, 6, 9}`.

For a work-group to use barriers and fences, its work-items must run on processing elements that share memory. In OpenCL, a group of processing elements with shared memory is called a *compute unit*. Each work-group executes on a single compute unit, and each compute unit executes only one work-group. Figure 4 shows this relationship graphically.

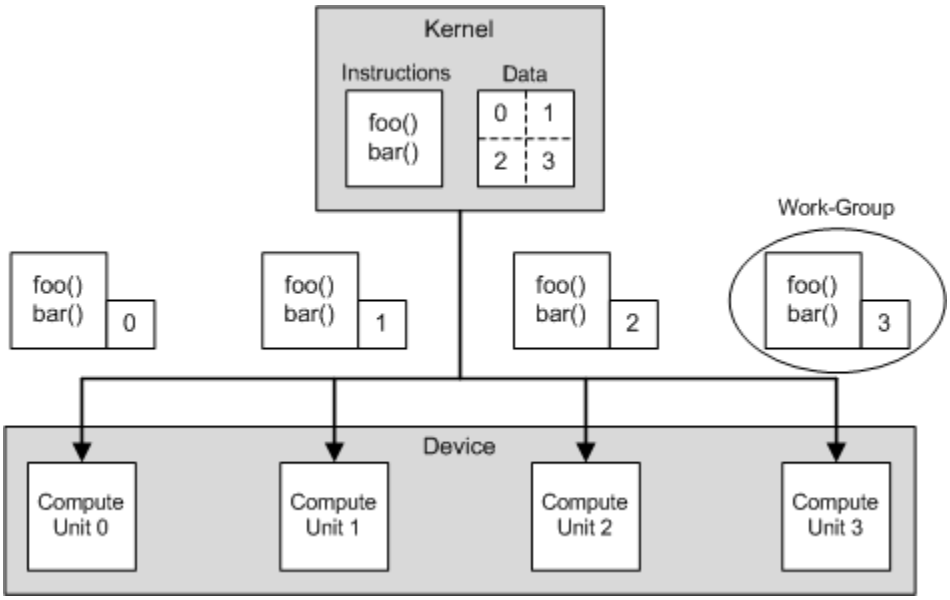


Figure 4 Sending work-groups to compute units

In addition to its global ID, each work-item in a work-group also has a local ID that identifies its position in the group. By accessing this, work-items can coordinate their processing with other work-items in a work-group.

You don't have to create work-groups. If you set `local_work_size` equal to `NULL`, OpenCL will decide how to distribute work-items among a device's processing elements.

Work-items, work-groups, and hardware

This discussion has explained how to configure work-items and work-groups in code, but a question arises: How many work-items should you create? The answer depends partly on your algorithm and partly on the target hardware. If the processing elements in a device are identical, it's a good idea to make your number of work-items a multiple of the number of elements. In general, however, it's a good idea to run experiments and profile the time taken for each.

But if you're releasing a product for customers, experimentation isn't an option. In this case, you should call `clGetDeviceInfo` to access information about the architecture of your target devices. To determine how many work groups are needed, I recommend calling `clGetDeviceInfo` with any or all of the following:

- `CL_DEVICE_MAX_COMPUTE_UNITS`—The maximum number of available compute units.
- `CL_DEVICE_MAX_MEM_ALLOC_SIZE`—The maximum size that can be allocated for a memory unit.
- `CL_DEVICE_MAX_WORK_GROUP_SIZE`—The maximum size allowed for work-groups.
- `CL_DEVICE_MAX_WORK_ITEM_SIZES`—The maximum size allowed for work-items.

It takes time and practice to determine how best to partition your computation using work-items.

Summary

We discussed data partitioning, which is crucial for any OpenCL application that demands high performance. The basic unit of work is the work-item, which corresponds to the code executed within a traditional C/C++ loop. Each work-item receives a global ID that allows it to access data specifically intended for it. If work-items require synchronization, they can be placed into work-groups. Each work-group executes on a single compute unit on the device.