GPGPU Training Supporting Document

Jan G. Cornelis

May 19, 2014

Contents

1 A View on GPUs

1.1 GPGPU

A GPU is a graphics processing unit. Its main purpose is to show images on the screen and to carry out the necessary computations to do so. During the last 20 years the computational power of GPUs has increased enormously and a number of people started to look at the possibility to use the GPU for general purpose computations, hence the term GPGPU. Initially it was necessary to cast these general purpose problems into a graphic framework i.e. it was necessary to write the code using for example OpenGL or DirectX, APIs used to program computer graphics. This approach was very cumbersome and efforts were made to design and implement ways to ease the creation of general purpose programs for the GPU.

In November 2006 the first commercially available framework for general purpose computing appeared: CUDA from NVIDIA. CUDA stands for "Compute Unified Device Architecture" and presents itself as a set of extensions to the C programming language and a runtime library. CUDA is proprietary and can only be used with NVIDIA GPUs.

OpenCL is an open standard that provides a framework for writing programs that execute across heterogeneous platforms. A heterogeneous platform is basically defined as a platform that contains more than one computing device, for example a CPU and a GPU. OpenCL was initially developed by Apple, but it has become an open standard that is maintained by the non-profit technology consortium Khronos Group, that also maintains OpenGL. Version 1.0 of OpenCL was released in 2008. At the moment of writing, OpenCL is at version 1.2 and there are several commercial implementations available from for example NVIDIA, AMD, Intel and IBM.

1.2 Black Box View

If you have a program that takes too long because it has to carry out a lot of computations on a lot of data, you can accelerate it by letting the GPU carry out these computations. You will have to write GPU-specific code that instructs the GPU how to do this and you will have to modify your original program such that instead of carrying out the computation itself, it lets the GPU do this. This incurs the following steps:

- Initialization: before you can use the GPU to let it carry out computations it is necessary to initialize a number of things.
- Data Transfer: before you can request the GPU to compute something, it is necessary to send it the data on which it should carry out the computations. When the GPU has finished computing, you need to retrieve the results.

- Execution on the GPU: you need to send the code to the GPU and specify its execution configuration. The execution configuration will be discussed in the next subsection.

Transferring data from the CPU to the GPU - or rather from the CPU-RAM to the GPU-RAM - and back takes place over the PCIe bus and is the main bottleneck in GPU computing. You should keep this in mind and make sure that the time gained by computing on the GPU is large enough to compensate for this overhead.

1.3 The Execution Model

A GPU is a collection of multiprocessors and a large amount of global memory that is connected to the rest of the computer via the PCIe bus. A multiprocessor is a collection of scalar processors together with an amount of local memory and registers.

When you send code to the GPU for execution you need to specify its execution configuration i.e. you need to specify how many threads or work items should execute this code and how they are structured. Work items may be organized in a 1-, 2- or 3-dimensional space. Furthermore, work items are part of work groups. We will call the organization of all work items the global range and the organization of the work items in a work group the local range. For example, Figure 1 shows 144 work items organized in a 2-dimensional global range of 12 by 12 work items and in 9 work groups of 4 by 4 work items. Note that such low numbers would never be used in practice, we chose them for the clarity of the image. Within the global range a work item is uniquely defined by its global id. Similarly within a local range a work item is uniquely defined by its local id.

When code is sent to the GPU for execution in a given configuration, the GPU will create a queue with work groups that need to be executed. Work groups are assigned to multiprocessors where they run to completion. When work groups have finished executing, new work groups will be scheduled and so on until all work groups have completed. It follows that each work group executes independent from the others and that there is no way to synchronize work groups during the execution of the code. It is however possible to synchronize work items of the same work group.

Figure 2 shows the memory model used by OpenCL - see Section 1.4. The global memory is accessible to all work items and may be cached; but it is up to the programmer to ensure consistency. Work items of the same work group can use local memory as a sort of explicit cache. Finally, registers are used to store a work item's private variables.

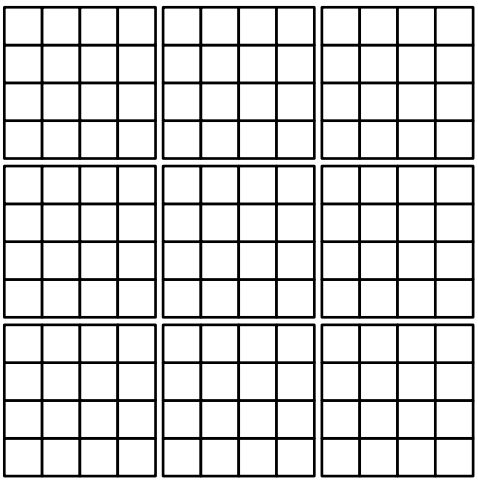

Figure 1: Sample Work item configuration

1.4 OpenCL

In the previous subsections we discussed how the GPU can be used for general purpose computation and how code is executed on the GPU - the execution model. In this subsection we will discuss OpenCL. OpenCL defines both an API and a language. The API specifies the objects and functions that should be used in your C or C++ code, while the language specifies the language used to write the code that will be executed on the GPU. In OpenCL terminology, the code that is executed on the GPU is called the kernel.

We will first briefly discuss the OpenCL API. Because our main interest are the performance characteristics of kernels on GPUs, our focus will lie on the OpenCL language. For a detailed overview of the OpenCL API we refer you to the reference pages:

`http://www.khronos.org/registry/cl/sdk/1.2/docs/man/xhtml`.

1.4.1 API

The OpenCL API corresponds to the subsection in the reference pages called "OpenCL Runtime". The API defines a number of objects that are central to its usage. We will list the most important ones together with the most important related functions. More detailed information for the mentioned functions can be found in the reference pages.

Platform An object of type `platform_id` identifies an OpenCL implementation. It is chosen from a list returned by `clGetPlatformIDs`. Specific information is queried by `clGetPlatformInfo`.

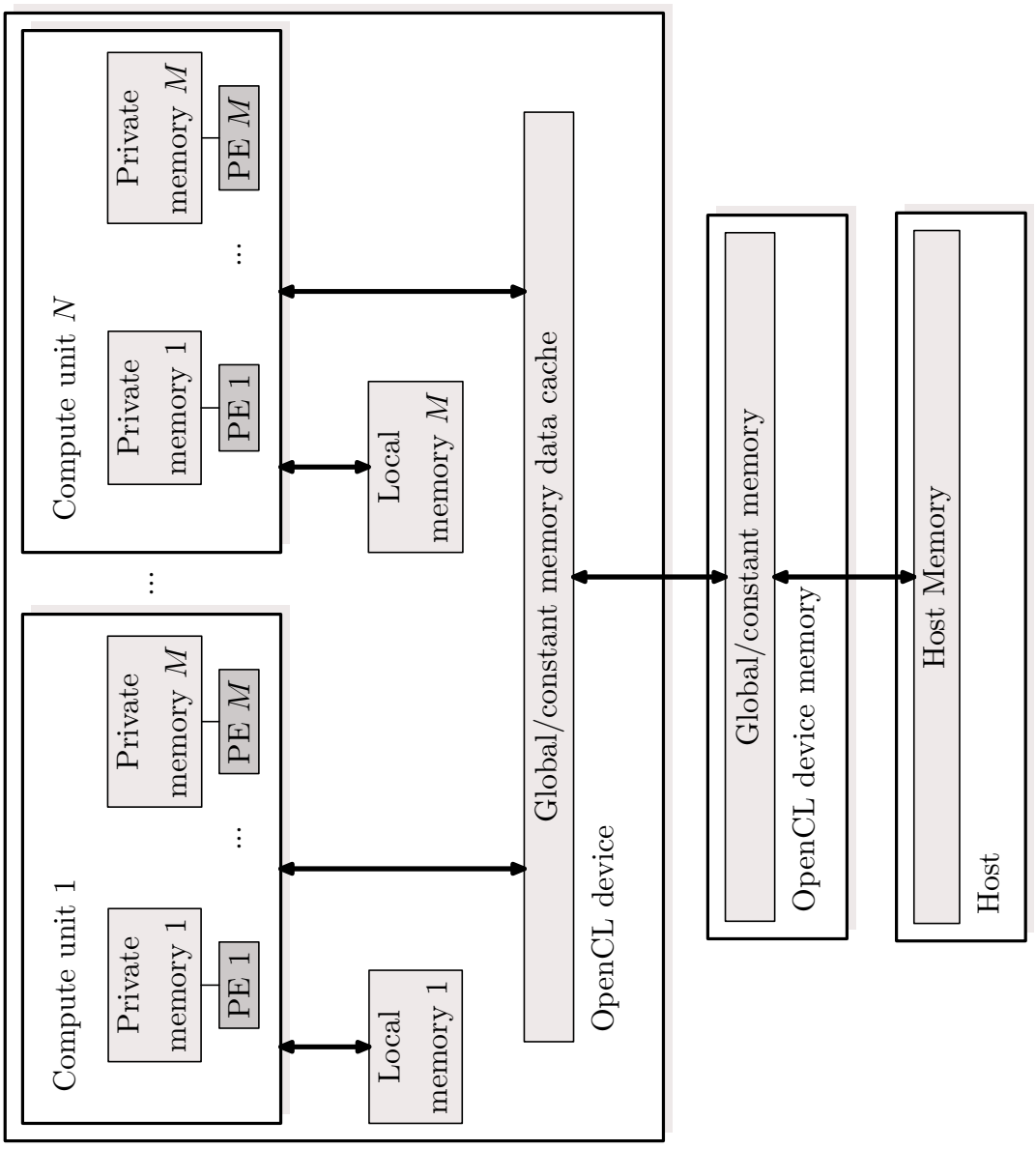

Figure 2: The Memory Model

Device An object of type `cl_device_id` identifies a compute device. It is chosen from a list returned by `clGetDeviceIDs`. Specific information is queried by `clGetDeviceInfo`.

Context An object of type `cl_context` groups a number of compute devices. It is created by `clCreateContext`. Specific information is queried by `clGetContextInfo`.

Command Queue An object of type `cl_command_queue` is associated with one device. It is created by `clCreateCommandQueue`. Specific information is queried by `clGetCommandQueueInfo`.

Memory Object An object of type `cl_mem` represents an area in memory. It is created by `clCreateBuffer`. Specific information is queried by `clGetMemObjectInfo`.

Program An object of type `cl_program` represents a bunch of OpenCL code. It is created by `clCreateProgramWithSource`[1]. Specific information is queried by `clGetProgramInfo`.

Kernel An object of type `cl_kernel` represents an OpenCL kernel. It is created by `clCreateKernel`. Specific information is queried by `clGetKernelInfo`.

A typical host program will first choose a platform and device and use these to create a context and a command queue. Given the OpenCL source code it will create an OpenCL program and compile it against the device. It will create memory objects to hold input and output data on the device. The kernel will be created from the OpenCL source code and `clSetKernelArg` will associate memory objects and other values with the kernel arguments. Next, the input data will be sent to the device using `clEnqueueWriteBuffer`. To launch a kernel we will use the function `clEnqueueNDRangeKernel`. The execution configuration is specified by parameters to this function. When the kernel has completed, the results are retrieved from the device using `clEnqueueReadBuffer`. Finally, you are supposed to call a number of `clRelease` functions to avoid memory leaks.

The OpenCL standard also defines a C++ wrapper that must be built on top of the C API. The wrapper defines most of the objects in terms of classes and most of the functions in terms of methods. Obviously, the `clCreate` functions are replaced by constructors and destructors remove the need for manual cleanup. Listing 1 shows a function that uses the OpenCL C++ wrapper to execute the SAXPY operation on the GPU. More information about this operation can be found at `http://en.wikipedia.org/wiki/SAXPY`.

[1] Or by `clCreateProgramWithBinary`.

Listing 1: OpenCL host code

```
void saxpy(float* x, float* y, int n, float a)
{

    std::vector<cl::Platform> platforms;
    cl::Platform::get(&platforms);

    std::vector<cl::Device> devices;
    platforms[0].getDevices(CL_DEVICE_TYPE_GPU, &devices);

    cl::Context context(devices);

    cl::CommandQueue commandQueue(context, devices[0]);

    std::string sourceText = fileToString("saxpy.ocl");
    std::pair<const char*, size_t> source(sourceText.c_str(), sourceText.size());
    cl::Program::Sources sources;
    sources.push_back(source);
    cl::Program program = cl::Program(context, sources);
    program.build(devices);

    cl::Buffer xBuffer(context, CL_MEM_READ_ONLY, sizeof(float)*n);
    cl::Buffer yBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*n);

    cl::Kernel kernel(program, "saxpy");
    kernel.setArg<cl::Buffer>(0, xBuffer);
    kernel.setArg<cl::Buffer>(1, yBuffer);
    kernel.setArg<float>(2, a);

    commandQueue.enqueueWriteBuffer(xBuffer, CL_TRUE, 0, sizeof(float)*n, x);
    commandQueue.enqueueWriteBuffer(yBuffer, CL_TRUE, 0, sizeof(float)*n, y);

    cl::NDRange global(n);
    cl::NDRange local(128);

    commandQueue.enqueueNDRangeKernel(kernel, cl::NullRange, global, local);

    commandQueue.enqueueReadBuffer(yBuffer, CL_TRUE, 0, sizeof(float)*n, y);
}
```

1.4.2 Language

The OpenCL language corresponds to the subsection in the reference pages called "OpenCL Compiler". The OpenCL language is C with some extra keywords and a number of restrictions.

The OpenCL programming language is basically a version of C that is extended with a number of keywords. In the OpenCL programming language you will define kernels and device functions. The difference between a kernel and a device function is simple: a kernel can be launched from the CPU, while a device function can only be launched from the GPU from a kernel or another device function. A kernel function is differentiated from a device function by the keyword `__kernel`. Furthermore, a kernel always has a `void` return type while a device function may have any kind of return type.

As in C, data is accessed through pointers. A number of keywords exist to specify the address space to which a pointer belongs: `__global` for global memory, `__local` for local memory and `__private` for private memory (registers). The last one is also the default when no keyword is given. There is also a `__constant` keyword but this one will be discussed in Section 1.5.

As mentioned previously a thread can find out its position in the collection of all threads and in the work group to which it belongs. The two main functions to do so are `get_global_id` and `get_local_id` that are used to retrieve the thread's index in a given dimension of the global or local thread space. Other interesting functions with similar purpose are `get_global_size` and `get_local_size`.

Finally, we mention the existence of the `barrier` function which forces the threads of a work-group to wait at the point where the function is called until all threads of the work-group reach this point.

Example To illustrate the above we demonstrate a kernel that implements the SAXPY operation. This operation takes two vectors X and Y and a scalar value α and carries out the following operation:

$$Y = \alpha \times X + Y$$

Listing 2 contains the code of an OpenCL kernel that implements this operation. This kernel illustrates the simplest way to map threads or work items on data: each thread processes on output data element. It is however possible to use more complicated mappings[2]. Code to launch this kernel was shown earlier in Listing 1.

[2]Given the importance of mapping work items to data items we have dedicated Section 2 to it.

Listing 2: SAXPY kernel

```
__kernel void
saxpy(__global float *X, __global float *Y, float alpha, unsigned int N)
{
    int index = get_global_id(0);
    if (index < N)
        Y[index] = alpha * X[index] + Y[index];
}
```

1.5 The Underlying Hardware

To write code that performs well it is important to have some idea about how the hardware is organized. The discussion in this subsection will mainly apply to NVIDIA GPUs but some of the information may be extrapolated to AMD GPUs.

1.5.1 NVIDIA

We first have a look at how a heterogeneous system looks like. Figure 3 gives an idea of how and how fast the different components of a heterogeneous system are connected. As mentioned earlier one of the main bottlenecks of GPU computing is the transfer of data from the CPU to the GPU and back. To amortize this cost, it is necessary to do as many computation as possible on the GPU once your data has been transferred. In the remainder of this discussion we will ignore the transfer of the data, given that is represents a constant cost to the performance of your code.

An NVIDIA GPU consists of a number of streaming multiprocessors, named SM or SMX in more recent generations. An SM holds 8 or 32 scalar processors, while an SMX has 192 scalar processors. Furthermore an SM has ? special function units (SFU), ? KB of local memory and a ? KB register file. Starting from GPUs of compute capability 2.0 an SM also holds ? of L1 global memory cache, while all SMs share ? of L2 global memory cache. An important concept in NVIDIA GPUs is the so-called warp. This is a group of 32 threads that execute in SIMD fashion.

There exist a number of reasons for a GPU not to process at the best possible performance. The best possible performance might be determined by the memory access subsystem if the code is memory bound or by the computational subsystem if the code is computation bound. We shall discuss performance more in detail in a later chapter, but we will discuss here hardware sources for overhead.

Memory Subsystem Memory requests are serviced per (half-) warp. Therefore, the maximum memory throughput for global memory can only be achieved when all bytes of such a memory request can be retrieved as part

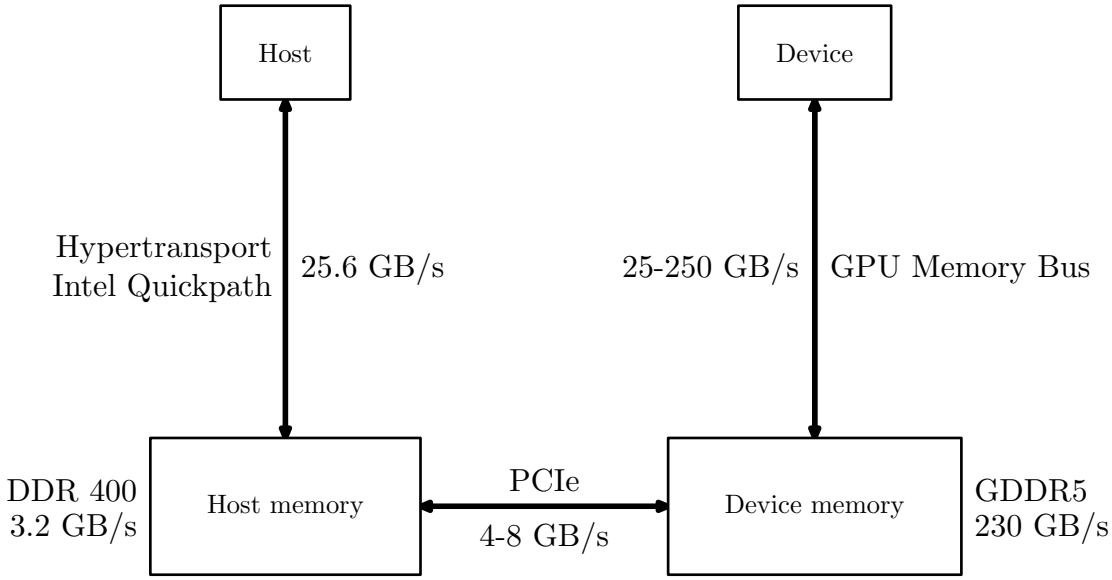

Figure 3: Organization of a Heterogeneous System

of one transaction. When this is not the case, excess bytes will be fetched and furthermore the time between memory transactions will increase further slowing down the memory access. This phenomenon is known under the name *non-coalesced memory access*.

Global memory is organized in partitions. If there are concurrent requests to the same partition by different work groups, these requests need to be serialized. This phenomenon is known under the name *partition camping*.

Shared memory is organized in banks. In order to maximize the reading and writing of shared memory the threads of a warp should access different banks. If this is not the case, their access will be serialized. This is known as *bank conflicts*.

Computational Subsystem Threads of a warp execute in SIMD fashion. Therefore, if the code contains branches and threads of the same warp follow different branches, then the execution of these branches is serialized. We say that the code suffers *branching*.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

```
// host code
cl::NDRange global(W);

// device code
unsigned int data_index = get_global_id(0);
```

Table 1: One to one mapping

2 Mapping Work Items on Data Items

2.1 Terminology

The work items are laid out in a *global index space*. This space is further subdivided in *local index spaces*. Both spaces may be 1-, 2- or 3-dimensional. The global index space and local index space must have the same number of dimensions. Furthermore, the number of work items along a certain dimension of the local index space must be a whole divisor of the number of work items along the same dimension of the global index space.

Like the work items, the data items are laid out in a 1-, 2- or 3-dimensional structure. 1-dimensional data structures are often referred to as *array* and 2-dimensional data structures as *matrix*.

For the sake of consistency we will refer to the number of data items in each dimension as W, H and D. These abbreviations suggest width, height and depth.

2.2 One-dimensional Data

We consider four different mapping types: *one-to-one*, *contiguous one-to-many*, *globally spaced one-to-many* and *locally spaced one-to-many*. We will explain their characteristics and provide the necessary information to use each kind of mapping in the following subsubsections.

2.2.1 One-to-one

The one-to-one mapping is the most simple mapping: it maps one work item to one data item. Table 1 contains relevant code and illustrates this mapping for $W = 24$ and work items organized in work groups containing 4 work items. Note that the work group size does not really matter for this example, but it will for the last one-dimensional mapping type we will present.

0	0	1	1	2	2	3	3	4	4	5	5	6	6	7	7	8	8	9	9	10	10	11	11

```
// host code
cl::NDRange global(W/N);

// device code
unsigned int data_index[N];
for (int i = 0; i < N; ++i)
    data_index[i] = N*get_global_id(0) + i;
```

Table 2: Contiguous one to many mapping

0	1	2	3	4	5	6	7	8	9	10	11	0	1	2	3	4	5	6	7	8	9	10	11

```
// host code
cl::NDRange global(W/N);

// device code
unsigned int data_index[N];
for (int i = 0; i < N; ++i)
    data_index[i] = get_global_id(0) + i*get_global_size(0);
```

Table 3: Globally spaced one to many mapping

2.2.2 Contiguous One-to-many

The one-to-many mapping maps one work item to many data items. The simplest way to do this is to assign contiguous data items to the same work item. Sample code and the resulting mapping for the case where two data items are assigned to each work item is shown in Table 2.

2.2.3 Globally Spaced One-to-many

Another way to map one work item to many data items is to assign to it data items that are not contiguous. One way to do it is to chop up the data in a number of parts equal to the number of data items assigned to each work (One can easily see that in one such part of the data contains as many data elements as there are data items) and to let each work item process a data item of each part. Sample code and the resulting mapping for the case where two data items are assigned to each work item is shown in Table 3.

2.2.4 Locally Spaced One-to-many

Finally, we can view the data as made up of tiles where one tile contains as many data items as would be processed by the work items of one work group (we note that implicitly this is also the case in the one-to-one and the contiguous one-to-many mapping). In this mapping one work item will process N non-contiguous data elements from the tile that corresponds to

0	1	2	3	0	1	2	3	4	5	6	7	4	5	6	7	8	9	10	11	8	9	10	11

```
// host code
cl::NDRange global(W/N);

// device code
unsigned int data_index[N];
for (int i = 0; i < N; ++i)
    data_index[i] = get_group_id(0)*N*get_local_size(0) + get_local_id(0) +
                    i*get_local_size(0);
```

Table 4: Locally spaced one to many mapping

its work group. Sample code and the resulting mapping for the case where two data items are assigned to each work item is shown in Table 4.

2.3 Two-dimensional Data

2.3.1 One-to-one

See Table 5.

2.3.2 Contiguous One-to-many

See Table 6.

2.3.3 Horizontal globally spaced one-to-many

See Table 7.

2.3.4 Vertical globally spaced one-to-many

See Table 8.

2.3.5 Horizontal locally spaced one-to-many

See Table 9.

2.3.6 Vertical locally spaced one-to-many

See Table 10.

0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.10	0.11	0.12	0.13	0.14	0.15
1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11	1.12	1.13	1.14	1.15
2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10	2.11	2.12	2.13	2.14	2.15
3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11	3.12	3.13	3.14	3.15
4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9	4.10	4.11	4.12	4.13	4.14	4.15
5.0	5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	5.10	5.11	5.12	5.13	5.14	5.15
6.0	6.1	6.2	6.3	6.4	6.5	6.6	6.7	6.8	6.9	6.10	6.11	6.12	6.13	6.14	6.15
7.0	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.8	7.9	7.10	7.11	7.12	7.13	7.14	7.15
8.0	8.1	8.2	8.3	8.4	8.5	8.6	8.7	8.8	8.9	8.10	8.11	8.12	8.13	8.14	8.15
9.0	9.1	9.2	9.3	9.4	9.5	9.6	9.7	9.8	9.9	9.10	9.11	9.12	9.13	9.14	9.15
10.0	10.1	10.2	10.3	10.4	10.5	10.6	10.7	10.8	10.9	10.10	10.11	10.12	10.13	10.14	10.15
11.0	11.1	11.2	11.3	11.4	11.5	11.6	11.7	11.8	11.9	11.10	11.11	11.12	11.13	11.14	11.15
12.0	12.1	12.2	12.3	12.4	12.5	12.6	12.7	12.8	12.9	12.10	12.11	12.12	12.13	12.14	12.15
13.0	13.1	13.2	13.3	13.4	13.5	13.6	13.7	13.8	13.9	13.10	13.11	13.12	13.13	13.14	13.15
14.0	14.1	14.2	14.3	14.4	14.5	14.6	14.7	14.8	14.9	14.10	14.11	14.12	14.13	14.14	14.15
15.0	15.1	15.2	15.3	15.4	15.5	15.6	15.7	15.8	15.9	15.10	15.11	15.12	15.13	15.14	15.15

```
// host code
cl::NDRange global(W,W);

// device code
unsigned int data_x = get_global_id(0);
unsigned int data_y = get_global_id(1);
```

Table 5: One to one mapping

0.0	0.0	0.1	0.1	0.2	0.2	0.3	0.3	0.4	0.4	0.5	0.5	0.6	0.6	0.7	0.7
1.0	1.0	1.1	1.1	1.2	1.2	1.3	1.3	1.4	1.4	1.5	1.5	1.6	1.6	1.7	1.7
2.0	2.0	2.1	2.1	2.2	2.2	2.3	2.3	2.4	2.4	2.5	2.5	2.6	2.6	2.7	2.7
3.0	3.0	3.1	3.1	3.2	3.2	3.3	3.3	3.4	3.4	3.5	3.5	3.6	3.6	3.7	3.7
4.0	4.0	4.1	4.1	4.2	4.2	4.3	4.3	4.4	4.4	4.5	4.5	4.6	4.6	4.7	4.7
5.0	5.0	5.1	5.1	5.2	5.2	5.3	5.3	5.4	5.4	5.5	5.5	5.6	5.6	5.7	5.7
6.0	6.0	6.1	6.1	6.2	6.2	6.3	6.3	6.4	6.4	6.5	6.5	6.6	6.6	6.7	6.7
7.0	7.0	7.1	7.1	7.2	7.2	7.3	7.3	7.4	7.4	7.5	7.5	7.6	7.6	7.7	7.7
8.0	8.0	8.1	8.1	8.2	8.2	8.3	8.3	8.4	8.4	8.5	8.5	8.6	8.6	8.7	8.7
9.0	9.0	9.1	9.1	9.2	9.2	9.3	9.3	9.4	9.4	9.5	9.5	9.6	9.6	9.7	9.7
10.0	10.0	10.1	10.1	10.2	10.2	10.3	10.3	10.4	10.4	10.5	10.5	10.6	10.6	10.7	10.7
11.0	11.0	11.1	11.1	11.2	11.2	11.3	11.3	11.4	11.4	11.5	11.5	11.6	11.6	11.7	11.7
12.0	12.0	12.1	12.1	12.2	12.2	12.3	12.3	12.4	12.4	12.5	12.5	12.6	12.6	12.7	12.7
13.0	13.0	13.1	13.1	13.2	13.2	13.3	13.3	13.4	13.4	13.5	13.5	13.6	13.6	13.7	13.7
14.0	14.0	14.1	14.1	14.2	14.2	14.3	14.3	14.4	14.4	14.5	14.5	14.6	14.6	14.7	14.7
15.0	15.0	15.1	15.1	15.2	15.2	15.3	15.3	15.4	15.4	15.5	15.5	15.6	15.6	15.7	15.7

```
// host code
cl::NDRange global(W/N,W);

// device code
unsigned int data_x[N];
unsigned int data_y = get_global_id(1);
for (int i = 0; i < N; ++i)
    data_x[i] = N*get_global_id(0) + i;
```

Table 6: Contiguous one to many mapping

0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7
1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7
2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7
3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7
4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7
5.0	5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.0	5.1	5.2	5.3	5.4	5.5	5.6	5.7
6.0	6.1	6.2	6.3	6.4	6.5	6.6	6.7	6.0	6.1	6.2	6.3	6.4	6.5	6.6	6.7
7.0	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.0	7.1	7.2	7.3	7.4	7.5	7.6	7.7
8.0	8.1	8.2	8.3	8.4	8.5	8.6	8.7	8.0	8.1	8.2	8.3	8.4	8.5	8.6	8.7
9.0	9.1	9.2	9.3	9.4	9.5	9.6	9.7	9.0	9.1	9.2	9.3	9.4	9.5	9.6	9.7
10.0	10.1	10.2	10.3	10.4	10.5	10.6	10.7	10.0	10.1	10.2	10.3	10.4	10.5	10.6	10.7
11.0	11.1	11.2	11.3	11.4	11.5	11.6	11.7	11.0	11.1	11.2	11.3	11.4	11.5	11.6	11.7
12.0	12.1	12.2	12.3	12.4	12.5	12.6	12.7	12.0	12.1	12.2	12.3	12.4	12.5	12.6	12.7
13.0	13.1	13.2	13.3	13.4	13.5	13.6	13.7	13.0	13.1	13.2	13.3	13.4	13.5	13.6	13.7
14.0	14.1	14.2	14.3	14.4	14.5	14.6	14.7	14.0	14.1	14.2	14.3	14.4	14.5	14.6	14.7
15.0	15.1	15.2	15.3	15.4	15.5	15.6	15.7	15.0	15.1	15.2	15.3	15.4	15.5	15.6	15.7

```
// host code
cl::NDRange global(W/N,W);

// device code
unsigned int data_x[N];
unsigned int data_y = get_global_id(1);
for (int i = 0; i < N; ++i)
    data_x[i] = get_global_id(0) + i*get_global_size(0);
```

Table 7: Global horizontally spaced one to many mapping

0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.10	0.11	0.12	0.13	0.14	0.15
1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11	1.12	1.13	1.14	1.15
2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10	2.11	2.12	2.13	2.14	2.15
3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11	3.12	3.13	3.14	3.15
4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9	4.10	4.11	4.12	4.13	4.14	4.15
5.0	5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	5.10	5.11	5.12	5.13	5.14	5.15
6.0	6.1	6.2	6.3	6.4	6.5	6.6	6.7	6.8	6.9	6.10	6.11	6.12	6.13	6.14	6.15
7.0	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.8	7.9	7.10	7.11	7.12	7.13	7.14	7.15
0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.10	0.11	0.12	0.13	0.14	0.15
1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11	1.12	1.13	1.14	1.15
2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10	2.11	2.12	2.13	2.14	2.15
3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11	3.12	3.13	3.14	3.15
4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9	4.10	4.11	4.12	4.13	4.14	4.15
5.0	5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	5.10	5.11	5.12	5.13	5.14	5.15
6.0	6.1	6.2	6.3	6.4	6.5	6.6	6.7	6.8	6.9	6.10	6.11	6.12	6.13	6.14	6.15
7.0	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.8	7.9	7.10	7.11	7.12	7.13	7.14	7.15

```
// host code
cl::NDRange global(W,W/N);

// device code
unsigned int data_x = get_global_id(0);
unsigned int data_y[2];
for (int i = 0; i < N; ++i)
    data_y[i] = get_global_id(1) + i*get_global_size(0);
```

Table 8: Global vertically spaced one to many mapping

0.0	0.1	0.2	0.3	0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.4	0.5	0.6	0.7
1.0	1.1	1.2	1.3	1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.4	1.5	1.6	1.7
2.0	2.1	2.2	2.3	2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.4	2.5	2.6	2.7
3.0	3.1	3.2	3.3	3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.4	3.5	3.6	3.7
4.0	4.1	4.2	4.3	4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.4	4.5	4.6	4.7
5.0	5.1	5.2	5.3	5.0	5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.4	5.5	5.6	5.7
6.0	6.1	6.2	6.3	6.0	6.1	6.2	6.3	6.4	6.5	6.6	6.7	6.4	6.5	6.6	6.7
7.0	7.1	7.2	7.3	7.0	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.4	7.5	7.6	7.7
8.0	8.1	8.2	8.3	8.0	8.1	8.2	8.3	8.4	8.5	8.6	8.7	8.4	8.5	8.6	8.7
9.0	9.1	9.2	9.3	9.0	9.1	9.2	9.3	9.4	9.5	9.6	9.7	9.4	9.5	9.6	9.7
10.0	10.1	10.2	10.3	10.0	10.1	10.2	10.3	10.4	10.5	10.6	10.7	10.4	10.5	10.6	10.7
11.0	11.1	11.2	11.3	11.0	11.1	11.2	11.3	11.4	11.5	11.6	11.7	11.4	11.5	11.6	11.7
12.0	12.1	12.2	12.3	12.0	12.1	12.2	12.3	12.4	12.5	12.6	12.7	12.4	12.5	12.6	12.7
13.0	13.1	13.2	13.3	13.0	13.1	13.2	13.3	13.4	13.5	13.6	13.7	13.4	13.5	13.6	13.7
14.0	14.1	14.2	14.3	14.0	14.1	14.2	14.3	14.4	14.5	14.6	14.7	14.4	14.5	14.6	14.7
15.0	15.1	15.2	15.3	15.0	15.1	15.2	15.3	15.4	15.5	15.6	15.7	15.4	15.5	15.6	15.7

```
// host code
cl::NDRange global(W/N,W);

// device code
unsigned int data_x[N];
unsigned int data_y = get_global_id(1);
for (int i = 0; i < N; ++i)
    data_x[i] = get_group_id(0)*get_local_size(0)*N + get_local_id(0) +
                i * get_local_size(0);
```

Table 9: Local horizontally spaced one to many mapping

0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.10	0.11	0.12	0.13	0.14	0.15
1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11	1.12	1.13	1.14	1.15
2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10	2.11	2.12	2.13	2.14	2.15
3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11	3.12	3.13	3.14	3.15
0.0	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9	0.10	0.11	0.12	0.13	0.14	0.15
1.0	1.1	1.2	1.3	1.4	1.5	1.6	1.7	1.8	1.9	1.10	1.11	1.12	1.13	1.14	1.15
2.0	2.1	2.2	2.3	2.4	2.5	2.6	2.7	2.8	2.9	2.10	2.11	2.12	2.13	2.14	2.15
3.0	3.1	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10	3.11	3.12	3.13	3.14	3.15
4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9	4.10	4.11	4.12	4.13	4.14	4.15
5.0	5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	5.10	5.11	5.12	5.13	5.14	5.15
6.0	6.1	6.2	6.3	6.4	6.5	6.6	6.7	6.8	6.9	6.10	6.11	6.12	6.13	6.14	6.15
7.0	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.8	7.9	7.10	7.11	7.12	7.13	7.14	7.15
4.0	4.1	4.2	4.3	4.4	4.5	4.6	4.7	4.8	4.9	4.10	4.11	4.12	4.13	4.14	4.15
5.0	5.1	5.2	5.3	5.4	5.5	5.6	5.7	5.8	5.9	5.10	5.11	5.12	5.13	5.14	5.15
6.0	6.1	6.2	6.3	6.4	6.5	6.6	6.7	6.8	6.9	6.10	6.11	6.12	6.13	6.14	6.15
7.0	7.1	7.2	7.3	7.4	7.5	7.6	7.7	7.8	7.9	7.10	7.11	7.12	7.13	7.14	7.15

```
// host code
cl::NDRange global(W,W/N);

// device code
unsigned int data_x = get_global_id(0);
unsigned int data_y[N];
for (int i = 0; i < N; ++i)
    data_y[i] = get_group_id(1)*get_local_size(1)*N + get_local_id(1) +
                i * get_local_size(1);
```

Table 10: Local vertically spaced one to many mapping

3 Miscellaneous stuff

3.1 Higher dimensional data

- Find elements (x,y) of a 2-dimensional data structure with width W stored as a 1-dimensional list *data*:

  ```
  data[W*y + x]
  ```

- Find elements (x,y,z) of a 3-dimensional data structure with width W and height H stored as a 1-dimensional list *data*:

  ```
  data[W*H*z + W*y + x]
  ```

3.2 Commonly made mistakes

Here are a few commonly made mistakes:

- Storing very small reference data in global memory.
- Misunderstanding the use of the work-item query functions.
- Misunderstanding that one kernel specifies the code for 1 (one) work item.
- Placing a barrier statement within a conditional statement.
- Placing a barrier statement after a return statement.
- . . . add your own.