Lesson 2: Programming GPUs

To manage the complexity of GPU Computing, we have divided the computing aspects in 4 levels.

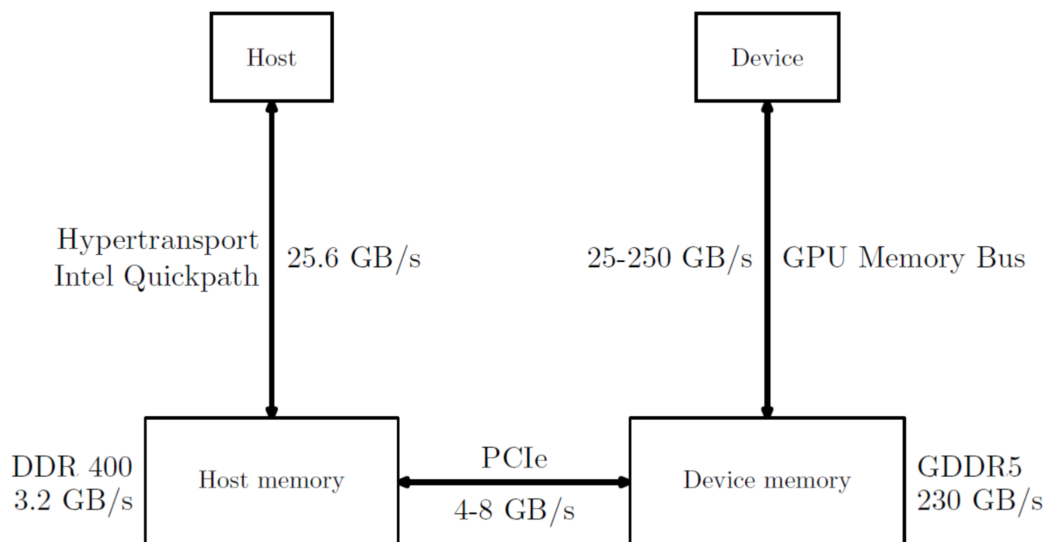
Level 0: Host code

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. This happens through the PCIe bus. 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 section.

A GPU is a collection of cores (multiprocessors) and a large amount of global memory that is connected to the rest of the computer via the PCIe bus, as shown in the following figure:



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 often an important 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.

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, called the **kernel**.

We will first briefly discuss the OpenCL API. Because for simple applications, basic code is sufficient which is always the same, 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

The OpenCL API

The OpenCL API corresponds to the section 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`.

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` or by `clCreateProgramWithBinary`. 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. The following code 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.

OpenCL host code using the C++ API

```
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);
}
```

Level 1: Parallel execution on the device

Now that we know how to start a GPU program from the host, we will look at how a GPU program, the **kernel**, should be built.

The OpenCL language

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

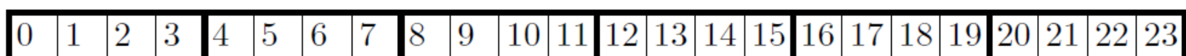
Work items are laid out in a **global index space**. This space may be 1-, 2- or 3-dimensional. In each dimension, threads are numbered as 0, 1, …, (size of dimension - 1). A thread can find out its position in the collection of all threads with the `get_global_id(int dimension)` function. It retrieves the thread's index in a given dimension of the global space.. Function `get_global_size(int dimension)` returns the size in the given dimension.

Mapping work items on the data

An important task is to decide what each work item will do. For most algorithms, this is done by assigning different parts of the data to each thread, i.e. defining a mapping of the work items on the data.

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 arrays and 2-dimensional data structures as matrices. Most often it is best to use the same structure for data as for the work items. We will consider 1-dimensional structures here.

The **one-to-one mapping** is the most simple mapping: it maps one work item onto one data item:



The following code illustrates this mapping for W = 24. W defines the array size and the number of work items. `data_index` is an array of size W and will contain the work item indices. The array is passed as a parameter to the kernel. The resulting array is shown in the figure.

```
// host code
unsigned int data_index[W];
cl::NDRange global(W);

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

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. This results in the following mapping:

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

Work item 0 is mapped onto elements 0 and 1, work item 1 onto 2 and 3, …

The following code maps each work item onto N elements. N should be a whole divisor of W.

```
// host code
unsigned int data_index[W];
cl::NDRange global(W/N);

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

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. The resulting mapping for the case where two data items are assigned to each work item:

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

Mapping of work item onto N elements.

```
// host code
unsigned int data_index[W];
cl::NDRange global(W/N);

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

Example of a one-on-one mapping

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

$$Y = \alpha \,.\, \mathbf{X} + \mathbf{Y}$$

Below 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 one output data element. The code to launch this kernel was shown earlier when explaining level 0.

```
__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];
}
```

If you are certain that the number of work items equals the array sizes, the `if (index < N)` is not necessary.

Measuring performance of instructions with microbenchmarks

A microbenchmark is a program designed to measure one specific characteristic of the hardware. Let's empirically measure the GPU performance. The computational performance is expressed as **operations per second** (Ops or Flops of floating point operations): the number of executed instructions divided by the runtime.

Assume we want to estimate the maximum throughput of floating-point multiplications, i.e. the following instruction:

a = b * c

with a, b and c floating point variables.

A kernel with just this operation won't work. Two issues should be solved:

1. Minimize all overhead so that only the execution time of the instruction is measured.
2. Fool the compiler: make sure the instruction is really executed.

The following benchmark kernel solves both issues and measures performance for a multiplication Instruction:

```
#define N 1000
__kernel void fmul (__global float* destination, float factor, int flag)
{
      float product = 1;
      #pragma unroll N
      for (int i = 0; i < N; ++i)
           product = product * factor;
      if (flag * product)
           destination[ get_global_id(0) ] = product;
}
```

(1) The overhead is minimized in two ways. First, the data on which the instruction operates is passed as a simple argument (`factor`). Secondly, by doing 1000 multiplications, the extra time for launching the kernel is divided by 1000. The relative contribution of other instructions of the kernel can then be neglected. By letting the compiler unroll the loop, there is no loop-overhead. Note that it is important that the iterations of the loop are dependent (result of one iteration is needed in the next iteration), otherwise the different instructions can be executed simultaneously. This is called Instruction Level Parallelism (ILP) and will be discussed later.

(2) Writing the result to global memory is done in the branch of a conditional statement whose condition depends on the result of the computation multiplied by a flag that is passed as an argument. Running the kernel with the flag equal to 0 will make the condition always fail. In this way, the compiler will not optimize the instructions away because the result of the computation is not returned.

The performance is then (W: number of work items):

Flops = W * N / runtime

For many instructions this is enough to measure the performance with sufficient accuracy.

In the above kernel the loop is completely unrolled. For some instructions and machines, however, completely unrolling a loop may cause a performance degradation because the code size becomes too big and does not fit in the instruction cache and/or instruction buffers. This is especially the case for the more recent GPUs which added an instruction buffer between the instruction cache and the schedulers. An alternative is to partially unroll the loop. This can be done in 2 ways:

```
#define N 1024
__kernel void fmul (__global float* destination, float factor, int flag)
{
      float product = 1;
      #pragma unroll 64
      for (int i = 0; i < N; ++i)
           product = product * factor;
      if (flag * product )
           destination[ get_global_id(0) ] = product;
}
```

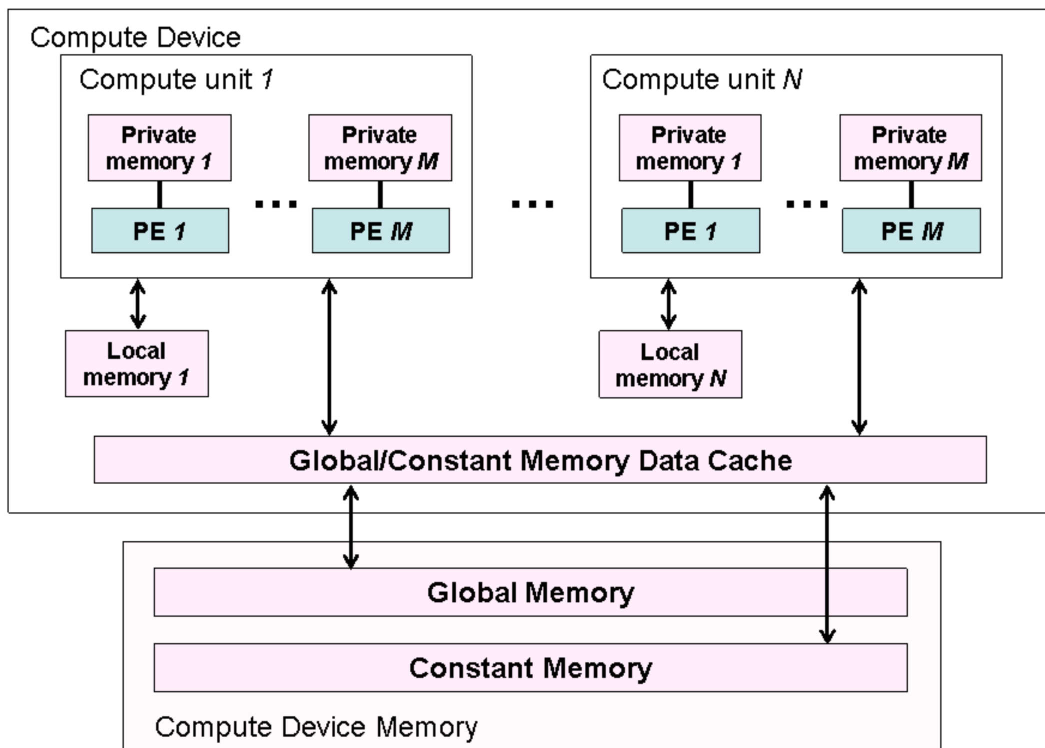
Make sure that 64 is a whole divisor of 1024.

If you don't trust the compiler, the following results in the same code:

```
#define N 1024
__kernel void fmul (__global float* destination, float factor, int flag)
{
      float product = 1;
      #pragma unroll 1 // prevent unrolling
      for (int i = 0; i < N/64; ++i) {
          #pragma unroll
          for (int j = 0; j < 64; ++j)
              product = product * factor;
      }
      if (flag * product )
           destination[ get_global_id(0) ] = product;
}
```

Level 2: Device model and work groups

The following diagram specifies the GPU model that needs to be understood to program GPUs:



The intermediate layer: Compute units and work groups

A GPU consists of several **Compute units**, called **cores** or by Nvidia Streaming Multiprocessors (SM or SMX in more recent generations). Each compute unit contains several Processing Elements (PEs) that will do the effective calculations. The memory will be discussed next. First, we discuss how work items are distributed among the compute units.

The work items are laid out in a 1-, 2- or 3-dimensional **global index space**. This space is subdivided into **work groups**. Within a work group the work items are mapped on to the **local index space**, which must have the same number of dimensions as the global index space. You specify it as the **local range** and pass it together with the global range. If you don't specify it, the compiler will choose an appropriate size. 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.

A work item 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(int dimension)` and `get_local_id(int dimension)` 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(int dimension)` and `get_local_size(int dimension)`.

A work group is executed on a single compute unit.

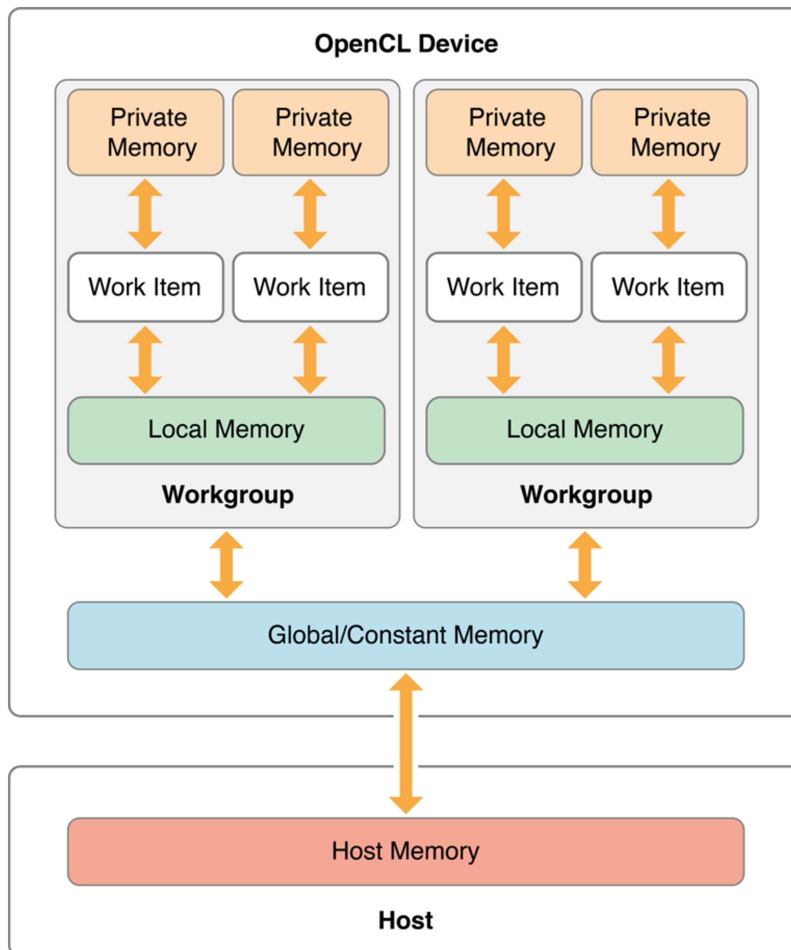
Work items of the same work group:

1. Share local memory which is fast. It acts a ‘cache’ to hold data that must be shared by all work items of a work group.
2. Can be synchronized with a barrier statement. The barrier function forces the work items of a work group to wait at the point where the function is called until all work items of the work group reach this point.

Work groups cannot be synchronized unless calling different kernels.

GPU Memory

The GPU memory hierarchy is as follows:



Global memory (GPU RAM) is fed with the data from host memory (CPU RAM). Constant memory is special hardware for fast retrieval of constants. **Local memory** resides within a compute unit and is shared by all work items of the same work group. It is fast. **Private memory** contains the data of a single work items. In practice these are the registers.

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

Local memory is mainly used for caching the data that will be needed for the whole work group. This happens as follows:

1. First, each work item copies a part of the data from global to local.

2. A barrier ensures that all work items have finished the copying.
3. The work items do their computations using the data from local memory.

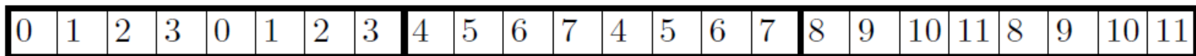
An implementation using local memory of matrix multiplication demonstrates this. Each work group computes a block of the result matrix C. The above 3 steps are performed iteratively. Each iteration a block of the A matrix and B matrix is copied to local and used to partially compute the result of the block of C.

It must be noted that modern GPUs offer automatic caching of global memory.

Allocating local memory is defined for a work group. It can be programmed in two ways: in the kernel body or as kernel argument.

Locally Spaced One-to-many

With local memory, another mapping is possible. Here, each work group contains 4 work items and each work item is mapped onto 2 data elements:



Mapping a work item onto N elements and L (=get_local_size(0)) work items within each work group:

```
// host code
unsigned int data_index[W];
cl::NDRange global(W/N);
cl::NDRange local(L);


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

Level 3 will be discussed in the next chapter