Karlsruhe Institute of Technology
Computer Graphics Group

M.Sc. Anton Kaplanyan* M.Sc. Jan Novák† Prof. Dr.-Ing. Carsten Dachsbacher‡

GPU Computing: Introduction

Abstract

Exploiting the vast horse power of contemporary GPUs for general purpose applications has become a must for any real time or interactive application nowadays. Current computer games use the GPUs not only for rendering graphics, but also for collision detection, physics, or artificial intelligence. General purpose computing on GPUs (GPGPU) has also penetrated the field of scientific computing enabling real time experience of large scale fluid simulations, medical visualization, signal processing, etc. This lecture introduces the concepts of programming graphics cards for non-graphical applications, such as data sorting, image filtering (e.g. denoising, sharpening), or physically based simulations.

In this very first assignment you will briefly hear about the history of GPU computing and the motivations that drive us to harness contemporary GPUs for general purpose computation. We will introduce the architectural considerations and constraints of contemporary GPUs that have to be reflected in the algorithm, shall it be efficient. You will also learn the basic concepts of the OpenCL programming language wrapped in a simple framework that we will be using during the course. All the knowledge you gain by reading this paper will be then applied in two simple starting assignments.

1 Parallel Programming

In few cases, the transition between sequential and parallel environment can be trivial. Consider for example a simple particle simulation, where particles are affected only by a gravity field. Sequential algorithm would iterate over all particles at each time step and perform some kind of integration technique to compute the new position (and velocity) of each particle. On a parallel architecture, we can achieve the same by creating a number of threads, each handling exactly one particle. If the parallel hardware contains enough of processing units, all particles can be processed in a single parallel step speeding up the simulation by a factor of N.

Unfortunately, not all problems can be parallelized in such a trivial manner; even worse, the number of such problems is quite low. In practical applications, we are facing much harder tasks that often require adjusting the algorithm or even reformulating of the problem. Even though each problem can be essentially unique, there is a set of well working paradigms that significantly increase the chances of successfully parallelizing an algorithm without too much of frustration and a risk of being fired. The road to victory consists of the following steps:

1. **Decompose the problem** into a set of smaller tasks and identify whether each of them can be (easily) parallelized or not. Whereas some parts can always be identified as embarrassingly parallel, others may require serialization of processing units and/or inter-thread communication. Finding the right granularity already at the beginning can save a lot of effort in the later development.

2. **Estimate the trade-offs** of parallelizing the inherently serial parts of the algorithm. If such parts can be efficiently processed on the CPU and the transfer between CPU and GPU is only marginal (compared to the rest of the algorithm), there is no need to parallelize these parts at any cost, as it may result in overall slow-down. On the other hand, even a slightly slower GPU implementation can be preferable if transferring data between RAM and GPU memory is costly.

3. **Reformulate the problem** if the solution does not seem to fit the architecture well. This is of course not possible in all cases, but using a different data layout, order of access, or some preprocessing can tackle the problem more efficiently. An example from the early GPGPU: architectures at the time did not allow for efficient scatter operations, the key to success at that time was to use a gather operation instead: collecting the data from neighbors instead of distributing it to them.

4. **Pick the right algorithm** for your application. Given a problem we can typically find several algorithms accomplishing the same using different approaches. It is necessary to compare them in terms of storage and bandwidth requirements, arithmetic intensity, and cost and step efficiency.

5. **Profile and analyze** your implementation. There are often several options how we can optimize an initial implementation leading to significant speed up. Make sure that the accesses to the global device memory are aligned, kernels do not waste registers, the transfer between CPU and GPU is minimized, and the number of threads enables high occupancy. These are only some basic concepts that we will (and some others) introduce during individual assignments. When optimizing you should always obey *Amdahl's Law* and focus on parts that consume most of the execution time, rather than those that can be easily optimized.

Parallel computing is a comprehensive and complex area of computer science that is hard to master without practice and experience. During this course we will try teach you a little bit of computational thinking through a number of assignments with emphasis on the right choice of algorithms and optimizations. Nevertheless, as the scope of the assignments must be limited (unfortunately), we point you to some useful literature and on-line seminars that can serve as supplementary material:

- *Programming Massively Parallel Processors, A Hands-on Approach*, David B. Kirk, Wen-mei W. Hwu., Morgan Kaufmann, 2010
- *NVIDIA OpenCL Programming Guide*

*e-mail: anton.kaplanyan@kit.edu
†e-mail: jan.novak@kit.edu
‡e-mail: dachsbacher@kit.edu

http://developer.nvidia.com/object/gpucomputing.html

- *GPU Computing Online Seminars*
  http://developer.nvidia.com/object/gpu_computing_online.html

1.1 History of Graphics Accelerators

Historical beginnings of modern graphics processing units date back to mid eighties, when the Amiga Corporation released their first computer featuring a device that would be nowadays recognized as a full graphics accelerator. Prior to this turning point, all the computers generated the graphics content on central processing unit (CPU). Offloading the computation of graphics to a dedicated device allowed higher specialization of the hardware and relieved the computational requirements on the CPU. By 1995, replaceable graphics cards with fixed-function accelerators surpassed expensive general-purpose coprocessors, which have completely faded away from the market in next few years (note the historical trend that was completely opposite to contemporary evolution of GPUs).

Large number of manufacturers and increasing demand on hardware-accelerated 3D graphics led to an establishment of two application programming interface (API) standards named OpenGL and DirectX. Whereas the first did not restrict its usage to a particular hardware and benefited from cutting-edge technologies of individual card series, the latter was usually one step behind due to its strict marketing policy targeting only a subset of vendors. Nevertheless, the difference quickly disappeared as Microsoft started working closely with GPU developers reaching a widespread adoption of its DirectX 5.0 in gaming market.

After 2001, GPU manufacturers enhanced the accelerators by adding support for programmable shading, allowing game developers and designers to adjust rendering algorithms to produce customized results. GPUs were equipped with conditional statements, loops, unordered accesses to the memory (gather and later scatter operations), and became moderately programmable devices. Such features enabled first attempts to exploit the graphics dedicated hardware for computing non-graphical tasks. The true revolution in the design of graphics cards came in 2007, when both market leading vendors, NVIDIA and ATI, dismissed the idea of separate specialized (vertex and fragment) shaders and replaced them with a single set of unified processing units. Instead of processing vertices and fragments at different units, the computation is nowadays performed on one set of unified processors only. Furthermore, the simplified architecture allows less complicated hardware design, which can be manufactured with shorter and faster silicon technology. Rendering of graphics is carried out with respect to the traditional graphics pipeline, where the GPU consecutively utilizes the set of processing units for vertex operations, geometry processing, fragment shading, and possibly some others. Thanks to the unified design, porting general tasks is nowadays less restrictive. Note the historical back-evolution: though highly advanced, powerful, and much more mature, modern GPUs are in some sense conceptually very similar to graphics accelerators manufactured before 1995.

1.2 Programming Languages and Environments

In order to write a GPU program, we first need to choose a suitable programming language. Besides others, there are three mainstream shading languages (GLSL, HLSL, and Cg) enabling general computing via mapping the algorithm to the traditional graphics pipeline. Since the primary target is processing of graphics, aka shading, programs written in these languages tightly follow the graphics pipeline and require the programmer to handle the data as vertices and fragments and store resources and results in buffers and textures. To hide the architecture of the underlying hardware, various research groups created languages for general computation on GPUs and multi-core CPUs. Among the most popular belong the Sh, Brook, and RapidMind, which yielded success mostly in other areas than computer graphics. Nevertheless, none of them brought a major breakthrough, mostly due to the inherent limitations imposed from hardware restrictions.

The situation improved with the unified architecture of GPUs. Contemporary NVIDIA graphics cards automatically support CUDA programming language, and the platform-independent OpenCL closes the gap for the remaining vendors, supporting even heterogenous computation on multiple central and graphics processing units at the same time. Unlike the shading languages, both CUDA and OpenCL enable true general purpose computation without resorting to the traditional graphics pipeline. The syntax and semantic rules are inherited from C with a few additional keywords to specify different types of execution units and memories.

2 Computing Architecture of Modern GPUs

Before we introduce the OpenCL programming language that will be used throughout this course, we will outline the architecture of modern GPUs. You should then clearly understand how the abstract language maps to the actual hardware. As most of the computers in the lab are equipped with NVIDIA graphics cards, and also because the NVIDIA's Compute Unified Device Architecture (CUDA) is more open to general computing, we will describe the individual parts of the compute architecture in the context of contemporary NVIDIA GPUs.

2.1 Streaming Design

Modern many-core GPUs (those developed by ATI and NVIDIA, not Intel's Larrabee, which was supposed to be conceptually different) consist of a several *streaming multiprocessors* (SM) that operate on large sets of streaming data. Each multiprocessor contains a number of *streaming processors* (SP). To perform the actual computation, individual SPs are equipped with several arithmethic (ALU), and floating point (FPU) units. The streaming multiprocessor is further provided with several load, store, and special function units, which are used for loading and storing data, and transcendental functions (e.g. sine, cosine, etc.) respectively. In order to execute the code, each SM uses an instruction cache, from which the warp scheduler and dispatch units fetch instructions and match them with GPU threads to be executed on the SM. Data can be stored either in registers, or in one of the very fast on-chip memories, depending on the access and privacy restrictions. As the capacity of the on-chip memory is highly limited, contemporary GPUs have gigabytes of device memory. We will provide some more detail about different storage options in Section 2.2. Figure 1 illustrates the architecture of a single streaming multiprocessor.

2.2 Memory Model

Memory model of contemporary NVIDIA graphics cards is shown on Figure 2. Different memory spaces can be classified regarding their degree of privacy. Each thread has a private *local* memory that cannot be shared. For cooperation of threads within a block *shared memory* can be used. Finally, an arbitrary exchange of data between all threads can only be achieved via a transfer through the *global memory*. Since different memory spaces have different parameters, such as latency and capacity, we provide a brief description of each in the following sections.

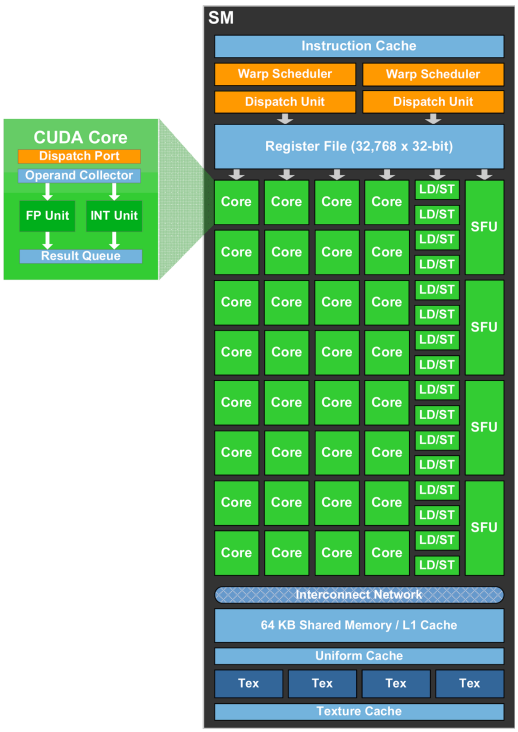

Figure 1: *Architecture of the GF100 (Fermi) streaming multiprocessor. Image courtesy of NVIDIA.*

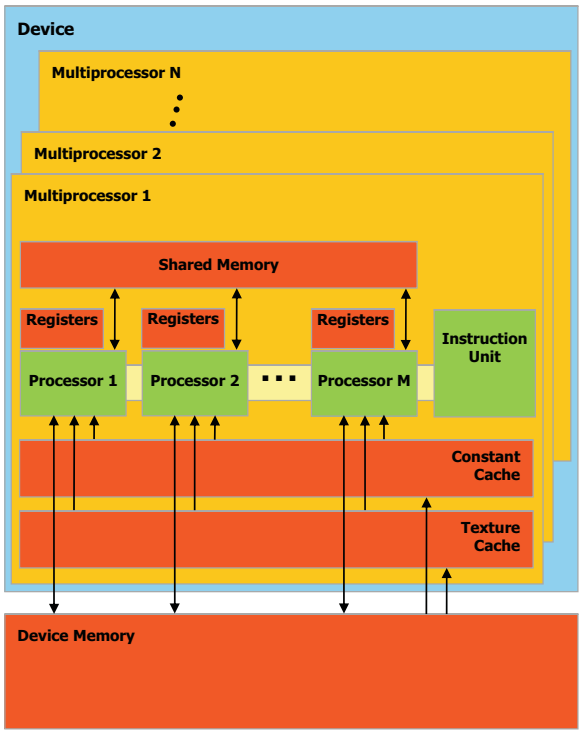

Figure 2: *Memory hierarchy of CUDA GPUs. Image courtesy of NVIDIA.*

2.2.1 Device Memory

The most prominent feature of the device memory is its high capacity, which in case of the newest GPUs reaches up to 4 GB. On the other hand, all memory spaces reserved in the device memory exhibit very high latency (400 to 600 clock cycles) prohibiting an extensive usage, when high performance is requested. Individual spaces are listed below.

- **Global Memory** is the most general space allowing both reading and writing data. Accesses to the global memory were prior to Fermi GPUs not cached. Despite the automatic caching we should try to use well-defined addressing to coalesce the accesses into a single transaction and minimize the overall latency.
- **Texture Memory**, as its name suggests, is optimized for storing textures. This type of storage is a read-only memory capable of automatically performing bilinear and trilinear interpolation of neighboring values (when floating point coordinates are used for addressing). Data fetches from the memory are cached, efficiently hiding the latency when multiple threads access the same item.
- **Constant Memory** represents a specific part of the device memory, which allows to store limited amount (64 KB) of constant data (on CUDA called symbols). Similarly to the texture memory, the accesses are cached but only reading is allowed. Constant memory should be used for *small* variables that are shared among all threads and do no require interpolation.
- **Local Memory** space is automatically allocated during the execution of kernels to provide the threads with storage for local variables that do not fit into the registers. Since local memory is not cached, the accesses are as expensive as accessing the global memory; however, the latency is partially hidden by automatic coalescing.

All previously mentioned device spaces, except for the local memory, are allocated and initialized by the host. Threads can only output the results of computation into the global memory; hence, it is used for exchanging data between successive kernels. Texture memory should be used for read-only data with spatial locality, whereas constant memory is suitable for common parameters and static variables.

2.2.2 On-chip Memory

The counterpart of the device memory is the on-chip memory, which manifests very low latency. Since it is placed directly on the multiprocessor, its capacity is very low allowing only a limited and specific usage, mostly caching and fast inter-thread communication.

- **Registers** are one of the most important features of the GPU when it comes to complex algorithms. If your program requires to many registers, it will hurt the performance since the warp scheduler cannot schedule enough threads on the SM. Multiprocessors on contemporary CUDA GPUs are equipped with 16384 (32768 on Fermi) registers with zero latency.
- **Shared Memory**, sometimes also called parallel data cache, or group-shared memory (in the context of DirectX), or local memory (in OpenCL), serves as a low latency storage for cooperation between threads. Its capacity of 16 KB (up to 48 KB on Fermi) is split between all blocks running on the multiprocessor in pseudo-parallel. The memory is composed of 16 (32 on Fermi) banks that can be accessed simultaneously; therefore, the threads must coordinate its accesses to avoid conflicts and subsequent serialization. The lifetime of

variables in shared memory equal to the lifetime of the block, so any variables left in the memory after the block has been processed are automatically discarded.

- **Texture Cache** hides the latency of accessing the texture memory. The cache is optimized for 2D spatial locality, so the highest performance is achieved when threads of the same warp access neighboring addresses. The capacity of the cache varies between 6 and 8 KB per multiprocessor, depending on the graphics card.
- **Constant Cache** is similar to texture cache: it caches the data read from the constant memory. The cache is shared by all processing units within the SM and its capacity on CUDA cards is 8 KB.

The only on-chip memory available to the programmer is the shared memory. Usage of both caches and registers is managed automatically by the memory manager, hiding any implementation details from the programmer.

3 The OpenCL Platform

Programmers have often been challenged by the task of solving the same problem on different architectures. Classical language standards, like ANSI C or C++, made life a lot easier: instead of using assembly instructions, the same high-level code could be compiled to any specific CPU ISA (Instruction Set Architecture). As the hardware generations evolved and took different directions, the goal to have "one code to rule them all" became more and more difficult to reach. The growth of clock rates of CPUs is slowing down, so the only way to continue the trend of Moore's law remained to increase the number of cores on the chip, thus making the execution parallel. A classical single-threaded program uses only small part of the available resources, forcing programmers to adopt new algorithms from the world of distributed systems. Contemporary GPUs are offering an efficient alternative for wide range of programming problems via languages very similar to C and C++. As the different platforms have different features for optimization, (out-of-order execution, specialized memory for textures), the programmer needs more and more specific knowledge about the hardware than before for low-level optimization.

Open Computing Language (OpenCL) is a new standard in parallel computing that targets simultaneous computation on heterogeneous platforms. It has been proposed by Apple Inc. and developed in joint cooperation with other leading companies in the field (Intel, NVIDIA, AMD, IBM, Motorola, and many others). Since it is an open standard (from 2008 maintained by the Khronos Group, which also takes care of OpenGL and OpenAL) it promises cross-platform applicability and support of many hardware vendors. By employing abstraction layers, an OpenCL application can be mapped to various hardware and can take different execution paths based on the available device capabilities. The programmer should still be highly familiar with parallelization, but can exploit the features of the underlying architecture by only knowing the fact that it implements some parts of a standardized model. From now on, we shall exclusively focus on GPGPU programming using OpenCL, but we should always keep in mind that using the same abstraction, unified parallelization is possible for heterogeneous, multi-CPU-GPU systems as well.

Even though OpenCL strives to provide an abstract foundation for general purpose computing, it still requires the programmer to follow a set of paradigms, arising from the common features of various architectures. These are described within the context of four abstract models (**Platform**, **Execution**, **Memory**, and **Programming** models) that OpenCL uses to hide hardware complexity.

3.1 Platform Model - Host and Devices

In the context of parallel programming, we need to distinguish between the hardware that performs the actual computation (*device*) and the one that controls the data and execution flow (*host*). The host is responsible for issuing routines and commands specified by the programmer that are then outsourced to devices connected to the host. A device may consist of multiple *compute units* that execute instructions on several *processing elements* in parallel.

In our case, the host is always a CPU and the device a CUDA GPU with compute units mapped to SMs, and processing elements represented by the individual scalar processors. OpenCL assumes that both, the host and the device, are equipped with their own memory, referring to them as the *host memory* and *device memory*. The responsibility for organizing data transfers between host and device memory is left on the host, which also calls appropriate functions for allocating and deallocating device memory. In general, you should always enumerate the available OpenCL devices to see, whether there are any to perform the actual computation.

3.2 Execution Model

As outlined in the platform model, OpenCL programs consist of two separate parts: *kernels* that execute on connected devices, and *host program* that executes on the host and initiates the individual kernels and memory transfers. A kernel is a single function that is invoked on the device and executes on a set of data in parallel. Think of a kernel as a block of code that simultaneously operates on the streaming data. In order to build more complex algorithms, we will often need multiple kernels and several other objects, such as memory arrays. OpenCL encapsulates all such information that is specific to the program by a *context*. The context includes the following resources:

- **Devices**
- **Kernels**
- **Program objects**
- **Memory objects**

The context is created on the host using an OpenCL API call. In order to manipulate resources (e.g. textures), we always have to access them through the context. In addition to devices and kernels, each parallel application also requires a *program object* that contains the source code and the compiled binary of the kernels. In order to execute the binary on the device, we need to add an execution event into a *command queue*. Adding individual commands into a queue (instead of executing them right away) allows for a non-blocking execution: after placing the command to the end of the queue, the host program can continue running without waiting for the command to be finished. Wenever the device is ready, it checks the command queue and takes the next pending item for execution. These items can be kernel execution commands, memory commands, or synchronization commands.

A single instance of a kernel (i.e. kernel applied to a single item in the data stream) is called *work-item* in OpenCL. All instances then form a global space of work-items that is called *NDRange* (n-dimensional range, where n is one, two, or three). Consider for example a problem of simulating a set of particles in a force field: the work-item in such case is simply an application of the force to a single particle. In this case, the NDRange will be a one-dimensional space containing applications of the force to all particles. We can further say that the problem has high granularity: we can decompose it into many small tasks that can be processed simultaneously. To distinguish between individual work-items, each work-item in the NDRange is assigned a global unique ID.

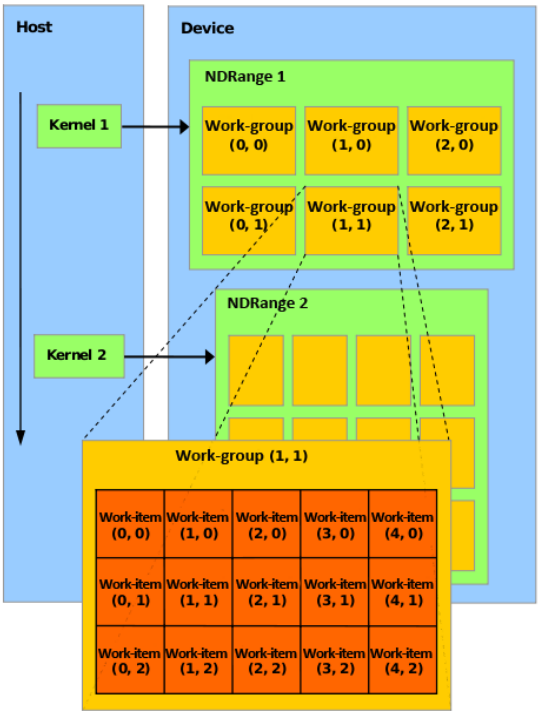

Figure 3: *Hierarchical execution model of OpenCL applications. Image courtesy of NVIDIA.*

In more complicated situations, when we, for instance, need to account for forces between particles, it is beneficial to somehow cluster them and process them in groups, solving the interactions locally first. To partition the computation into groups, OpenCL organizes work-items into *work-groups* that provide more coarse-grained decomposition of the problem. Work-items that belong to one work-group execute on processing units of one SM and are identified locally by a unique ID within the work-group. Furthermore, work-items in a work-group can communicate and share data through the local memory (shared memory in CUDA naming conventions). The hierarchical execution model is shown in Figure 3.

3.3 Memory Model

The *device memory* available to OpenCL programs can be categorized into four main memory regions. If you refer back to Section 2.2, you will see that the OpenCL device memory maps to the memory model of a CUDA with just a few minor differences. Please note that the device memory in context of OpenCL refers to all memory on the device, whereas CUDA programming guide uses the term to distinguish the global device memory only (i.e. all memory except for the on-chip storage).

- **Global memory** is accessible form host as well as from kernels. Work-items can read from / write to any element of a memory object in the global memory, but the order of executing two operations on a single memory element within one kernel is undefined. Synchronization on global memory can be achieved by splitting the computation into multiple kernels. (Or using atomic instruction extensions, but that is out of the scope of this course).
- **Constant memory** is a special region of global memory that remains constant during the execution of a kernel. This memory region can only be written by the host prior to the kernel execution. Recognizing that some part of your data is constant can greatly improve the performance of your code, since it allows the device to cache data in the constant memory cache.
- **Local memory** is a special region that is specific to a work-group. This region can be used to declare variables that are shared between work-items of the same group. Besides this, accessing local memory should be orders of magnitude faster than accessing global memory. On the CUDA architecture, local memory maps to *shared memory* of the SM.
- **Private memory** is represented by registers that hold the private and temporary variables of a work-item. The content of this region is only valid within the lifetime of the work-item.

As the device memory is independent of the host, the host program can manipulate the device memory objects through the OpenCL command queue. One way of interaction is to explicitly copy data between device memory objects or between the device and host memory. The host is also allowed to *map* regions of device memory to its own address space. Such memory access can be blocking or non-blocking. A non-blocking command returns immediately after a command was successfully enqueued, while the blocking version stops the execution of host code until the copying operation is performed and the host memory involved in the operation can be reused.

3.4 Programming Model

OpenCL recognizes two different models of parallelism: *data parallel* and *task parallel*. In the scope of this course, we will always talk about data parallel problems and algorithms. For details on the task parallel programming model, please refer to the OpenCL Specification.

3.4.1 Explicit Hierarchical Data Parallel Programming Model

In context of GPGPU development we will always talk about an *explicit hierarchical data parallel programming model*. Data parallel means that the same instructions are applied to all data in the stream. The programming model is hierarchical, since the work is organized hierarchically: kernels are executed in work-items, which are structured into work-groups, and finally into an NDRange. Finally, the model is also explicit, because we define the total number of work-items that execute in parallel, and also how these work-items are organized into work-groups.

3.4.2 Synchronization

In order to enable cooperation between work-items, we need synchronization primitives. Synchronization within a work-group can be achieved by placing a barrier. Until all work-items reach the barrier, none of them is allowed to continue executing subsequent instructions. This means that barriers should never be placed in conditional branches, unless all work-items evaluate the condition with the same boolean result. Violating this rule may result in locking the execution on the device, which in case the of operating systems without GPU driver timeout results in complete freeze of the system.

Synchronization between work-items in different work-groups is not possible. Nevertheless, as kernels are guaranteed to finish in the order they were added into the command queue, we can achieve synchronization on a global level by splitting the computation into several kernels.

4 Assignment 1: First OpenCL Program

4.1 Skills You Learn

In the very first assignment you will learn the basics of OpenCL programming. After completing the exercise, you will be able to:

- Set up a working OpenCL context
- Launch kernels on the GPU for a given number of threads
- Allocate memory on the GPU and transfer data between the host and the device

4.2 Problem

Let us begin with a really simple problem. Suppose we are given two large vectors of integers, A and B. Our task is to simply add these two vectors together, using the elements of the second vector in reverse order, see Figure 4. This problem is trivial to parallelize, since the elements of the vectors can be added independently. We can simply create as many threads as elements the vectors have, then each thread will be responsible for adding two components, one from each vector. If our machine would be able to run infinite number of threads in parallel, our program would execute in one step for any vector size.

In our exercises, we will use the Nvidia OpenCL SDK, however, the code you write will be fully portable to any compatible OpenCL platform. The SDK is already installed on capable computers in the lab. For your personal use, you can download it from NVIDIA's website (`http://developer.nvidia.com/object/gpucomputing.html`). We have also prepared a startup solution that contains the skeleton of the code you will have to complete during the exercise. Before we start, let us quickly examine the contents of the project file. Open the solution *Assignment1_Startup.sln* (or *Assignment1_Startup08.sln* in case you are using VS 2008).

Your solution already contains the following files:

- `main.cpp` The entry point of your application. This is where you will create and set up an OpenCL context, execute one or more kernels, and release the context and all the other resources.
- `IAssignment.h` Most of our assignments follow the same execution pattern, therefore we prepared a base interface, which you should implement for each specific task. This would allow you to easily test all your results from the `main.cpp` code, without significant changes to the code.
- `CTimer.h`, `CTimer.cpp` A simple timer class we will use to measure performance. The class is already implemented and ready for use.
- `Common.h`, `Common.cpp` These files contain utility code for frequently needed tasks, such as loading a kernel code from a file or printing out error messages.
- `CSimpleArrays.h`, `CSimpleArrays.cpp` Chunks of the first assignment.

4.3 Setting up the Device Context

The first step of every OpenCL application is setting up a working context using one of the available OpenCL devices. In the `main.cpp` you will find two empty functions where you should put the global initialization and cleanup:

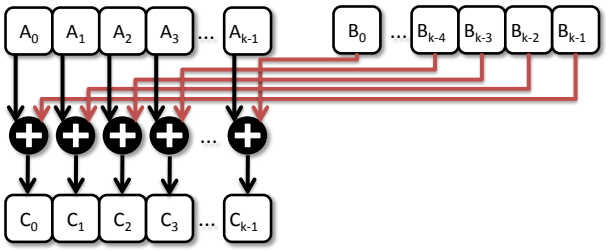

Figure 4: *Element-wise addition of two vectors, where we use the element of the second vector in reverse order.*

```
bool InitContextResources()
{
  return false;
}

void CleanupContextResources()
{
}
```

`InitContextResources()` will be called once when your application starts up. Once all computations are finished, the framework will call the `CleanupContextResources()`, where you should release all allocated resources.

You should take the following steps to prepare everything to run your first kernel:

1. Query the installed (supported) OpenCL platforms in your system.
2. Enumarate the installed OpenCL devices supporting the selected platform, and get a reference to one of them. You can execute OpenCL kernels on this device using the context.
3. Create a new context on the device. This abstract object will hold all the resources your application will create on the device, including kernels and memory objects.
4. Finally, create at least one command queue. This will allow the host to asynchronously communicate with selected devices and issue commands to it.

All the initialization functions follow a similar pattern. If an error occurs, an error code is returned (data type `cl_int`). You should always examine the returned value before proceeding, to make sure that the previous call finished successfully. In order to perform this check with minimum overhead, `Common.h` contains helper macros for checking the errors and printing out error messages to the console. This can be very helpful when debugging your code. For example, the first two steps of the list above should look like this:

```
cl_int clError;

//get platform ID
V_RETURN_FALSE_CL( clGetPlatformIDs(1, &g_CLPlatform,
    NULL), "Failed to get CL platform ID" );

//get a reference to the first available GPU device
V_RETURN_FALSE_CL( clGetDeviceIDs(g_CLPlatform,
    CL_DEVICE_TYPE_GPU, 1, &g_CLDevice, NULL), "No GPU
    device found." );
```

Copy these function calls to your `InitContextResources()` function. Half of your initialization is already done. There are other

types of OpenCL calls, where the return value is a new resource, for instance. The error code in such cases will be returned in an output function parameter. This is also the case when creating a new context:

```
//Create a new OpenCL context on the selected device
g_CLContext = clCreateContext(0, 1, &g_CLDevice, NULL,
    NULL, &clError);
V_RETURN_FALSE_CL(clError, "Failed to create OpenCL
    context.");
```

The only thing remaining is to create a command queue using the context you have created and the device you selected. Use the `clCreateCommandQueue` function, which returns a reference to the new command queue. The OpenCL specification of the header of this function is:

```
cl_command_queue clCreateCommandQueue (cl_context
  context, cl_device_id device,
  cl_command_queue_properties properties,
  cl_int *errcode_ret)
```

As the `properties` parameter, simply pass 0. If everything went well, return `true` to let the `main` function know that your OpenCL device is ready to use.

Now, when you run your application, the output will be something like this:

```
OpenCL context initialized.
Running vector addition example...

Computing CPU reference result...DONE
Computing GPU result...DONE
INVALID RESULTS!

Press any key...
```

The main function already tries to test your vector addition example, but of course it fails since we have not implemented it yet. This is our next step, but first we should make sure that the code will be neat and clean, thus, when allocating some resources, we also have to release them. You can release the command queue and the context in the `CleanupContextResources()` function:

```
if(g_CLCommandQueue)
  clReleaseCommandQueue(g_CLCommandQueue);
if(g_CLContext)     clReleaseContext(g_CLContext);
```

4.4 Kernel Implementation

In C, the algorithm performing reversed addition would look like this:

```
for(unsigned int i = 0; i < m_ArraySize; i++)
{
  m_hC[i] = m_hA[i] + m_hB[m_ArraySize - i - 1];
}
```

Please note the `m_h` prefix before the array names. Since we have objects in the host but also in the device memory, it is a good programming practice to denote pointers to these separate memory spaces differently. Therefore, we recommend to use the following naming convention: prefix `m_` is for members, prefix `g_` is for global variables, `h` stands for host, and `d` for device.

```
//integer arrays in the host memory
int *m_hA, *m_hB, *m_hC;

//integer arrays in the device memory
cl_mem m_dA, m_dB, m_dC;
```

In order to implement the kernel, you should create a new file named `VectorAdd.cl` (in the root directory of the project) that will contain the code of the kernel. This file will be loaded and compiled at run time.

The entry point of the kernel will be very similar to a C function, using some special OpenCL keywords. As an input parameter, the kernel expects two integer arrays, and the number of elements in the arrays. The result of the reversed addition will be written to the third array.

```
__kernel void VecAdd(__global const int* a, __global
    const int* b, __global int* c, int numElements)
{}
```

The `__kernel` prefix tells the OpenCL compiler that this is an entry function of a kernel that can be invoked from the host, using the command queue. The `__global` prefix is also important: it signals that pointers `a`, `b`, and `c` refer to *global memory*.

After reading the introduction materials it should be clear that *the kernel will be executed in parallel for each item/thread*. So how will a given thread know, which elements of the arrays to add? For this we have use the built-in indexing mechanism. The `get_global_id(i)` built-in function returns the index of the thread within the given NDRange dimension. Since our arrays are one-dimensional, we only need to know the index in the first dimension.

```
int GID = get_global_id(0);
//an example of using the index to access an item:
int myElement = a[GID];
```

Use the global work-item index to implement the addition of two elements in the same way as shown in the C code at the beginning of this section. Make sure that one thread only produces one component of the resulting vector. As a programmer of the kernel, we do not have any information about the actual number of threads that will be executed, but we can assume that the number of threads is at least equal to the number of elements. Therefore, you should also check if the thread index is less than the total number of elements in the array to avoid accessing data beyond the extent of the array.

4.5 Managing Data on the Device

If you examine the implementation of the overloaded function `InitResources()` in the `CSimpleArrays` class, you will see that it already allocates three arrays in the host memory and fills two of them with random integers. Your task in this step is to create their device counterparts, on which your kernel can perform the parallel addition. You should allocate three memory objects, then copy the input data from the host arrays to the device memory objects.

The `clCreateBuffer` function can be used to allocate buffer objects of a given size on the device. For optimization reasons, we should also inform the device how do we want to use the allocated memory later on. For example, memory which we only read can be cached efficiently. You can allocate a buffer storing one of your integer arrays using the following syntax:

```
cl_int clError;
m_dPtr = clCreateBuffer(<Context>, [CL_MEM_READ_ONLY |
    CL_MEM_WRITE_ONLY], sizeof(cl_int) * <ArraySize>,
    NULL, &clError);
```

Extend the `InitResources()` function to allocate the two input arrays `m_dA`, `m_dB` and the output array `m_dC` on the device, using the optimal memory access flags. `CL_MEM_READ_ONLY` means that the given buffer is a readable constant from the kernel; `CL_MEM_WRITE_ONLY` means that the kernel can write data to the buffer, but cannot read from it. You should check for errors the same way as before.

Having the buffers on the device created, the next step is to copy the input data into them. Once the kernel is finished, you will also need to copy the results from the output buffer. These memory operations you are performed using the *command queue* object.

The `ComputeGPU()` method of your assignment class is still empty. This method is called by the main function when your resources are already allocated. Add some code now to this part to copy data to your buffer locations, using the `EnqueueWriteBuffer()` method. This method can be used in different ways, we will now use a non-blocking version: it means, that the request to copy data from a host pointer to a device pointer is enqueued to the command queue, but the host code continues executing subsequent commands and does not wait for the device to perform the copy operation. We can use the non-blocking call, since we do not use the device buffer from the CPU, and *we do not change or free* data referenced by the host pointer during execution. The code for copying:

```
clErr |= clEnqueueWriteBuffer(CommandQueue, m_dB,
    CL_FALSE, 0, m_ArraySize * sizeof(int), m_hB, 0,
    NULL, NULL);
V_RETURN_CL(clErr, "Error copying data from host to
    device!");
```

Refer to the OpenCL specification for more details on the function call. The second parameter is the pointer to the target device memory object, the sixth parameter is a pointer to the source host memory location. The third parameter determines if the call is blocking or non-blocking. After this the random numbers are already in the GPU memory, ready to be processed.

Do not forget to release allocated device memory when it is no longer needed. For this go to the `ReleaseResources()` method of your class and free all allocated objects. You can release a buffer on the device using the `clReleaseMemObject(<buffer>)` function call. Since many modifications have been made, you should compile and run your code to see if it executes properly.

4.6 Compile and Execute the Kernel

The kernel code is implemented and the necessary memory is copied to the device, hence we can examine the results. Creating executable kernel objects on the device requires three main steps:

1. Create an OpenCL program object from a source file.
2. Compile the program object on the device. This generates a binary program in the memory of the device.
3. Create one or more kernel objects from the program.

First, add a reference to a program object and its single kernel to your class members:

```
cl_program    m_Program;
cl_kernel     m_Kernel;
```

As these objects are also resources on the device, extend your existing code of `InitResources()` to load and compile the program using the following code snippet:

```
//load and compile kernels
char* programCode = NULL;
size_t programSize = 0;

LoadProgram("VectorAdd.cl", &programCode, &programSize);

//create a program object (it can contain multiple kernel
    entry points)
m_Program = clCreateProgramWithSource(Context, 1, (const
    char**) &programCode, &programSize, &clError);
V_RETURN_FALSE_CL(clError, "Failed to create program from
    file.");

//build the program
clError = clBuildProgram(m_Program, 1, &Device, NULL,
    NULL, NULL);

if(clError != CL_SUCCESS)
{
  PrintBuildLog(m_Program, Device);
  return false;
}

//create kernels from the program
m_Kernel = clCreateKernel(m_Program, "VecAdd", &clError);
V_RETURN_FALSE_CL(clError, "Failed to create kernel:
    VecAdd");
```

`LoadProgram()` is a utility function we have provided to load the program code from a given location. As a location, use the path of the kernel you implemented in Section 4.4. Then the above example uses `clCreateProgramWithSource()` to create one program object using the given source. Note that no real compilation took place until now. You will have to use `clBuildProgram()` to actually compile the program code into binary instructions for the selected OpenCL device. Finally, `clCreateKernel()` creates a kernel object from a built program. The second parameter is the name of the entry point of the kernel, this must be the same as the name of your function with the `__kernel` prefix. Note that this way your code can contain multiple kernels, each specified by a different entry point.

Obviously, the compilation can fail due to coding errors. We can of course detect them using the usual error code mechanism, but the programmer would probably like to know the cause of the failure. Since accessing the error message of the OpenCL build process is slightly complicated, we provide you with another utility function. If you detect that the program build failed, call the `PrintBuildLog()` function which lists the related error messages to the console.

Our kernel expects some parameters: the pointers to device memory arrays and a number of elements. To provide them, we have to *bind* values to these attributes before executing the kernel. This following code binds the first input attribute to array `m_dA`:

```
clError = clSetKernelArg(m_Kernel, 0, sizeof(cl_mem), (
    void*)&m_dA);
```

The second parameter is the zero-based index of the attribute. Bind all four attributes using the same function call to the kernel. The third parameter is the size of the passed value in bytes, so always replace it using the appropriate type.

Now the kernel is properly compiled and ready to use. You can use the command queue to run the kernel with a given number of threads (work-items) using the `clEnqueueNDRangeKernel()` function:

```
cl_int clEnqueueNDRangeKernel (
  cl_command_queue command_queue,
  cl_kernel kernel,
  cl_uint work_dim,
  const size_t *global_work_offset,
  const size_t *global_work_size,
  const size_t *local_work_size,
  cl_uint num_events_in_wait_list,
  const cl_event *event_wait_list,
  cl_event *event)
```

The parameters that use should use are:

- **command_queue:** The command to run the kernel will be enqueued into this command queue.
- **kernel:** The kernel object to execute.
- **work_dim:** The number of dimensions of the NDRange (1-3). Currently you should use 1, since you want to add one-dimensional arrays.
- **global_work_size:** The number of *work-items* in *work_dim* dimensions that execute the kernel function. The total number of work-items is computed by multiplying these values: global_work_size[0] * ... * global_work_size[work_dim - 1].
- **local_work_size:** The number of *work-items* in a *work-group*. The numbers specified in the *global_work_size* should always be multiples of the elements of the *local_work_size*.

All the remaining parameters should be set to NULL.

To make the code more flexible, we will let the user to specify the local work size. That means, your assignment class implementation must always calculate the necessary number of *work-groups* to execute based on this user-controlled parameter. Actually, you cannot set directly the number of *work-items*, but you have to set the global work size, which indirectly determines this number.

You will extend the previously modified `ComputeGPU()` method of your class, which already contains the copying of the input data to the device. Prior to calling the `clEnqueueNDRangeKernel` command, you have to determine the number of necessary *work-items* to be launched. Of course this depends of the number of components in the vectors (one *work-item* for each), but also on the local work size, as the global work size should be the multiple of these dimensions. We have provided a simple helper function that computes the necessary dimensions for a given data element number:

```
// CSimpleArrays.ComputeGPU() method...
size_t globalWorkSize = GetGlobalWorkSize(m_ArraySize,
    LocalWorkSize[0]);
size_t nGroups = globalWorkSize / LocalWorkSize[0];
cout<<"Executing "<<globalWorkSize<<" threads in "<<
    nGroups<<" groups of size "<<LocalWorkSize[0]<<endl;

[...]
// Common.cpp
size_t GetGlobalWorkSize(size_t DataSize, size_t
    LocalWorkSize)
{
  size_t r = DataSize % LocalWorkSize;
  if(r == 0)
  {
    return DataSize;
  }
  else
  {
    return DataSize + LocalWorkSize - r;
  }
}
```

Using the resulting global work size, run your kernel on the device:

```
clErr = clEnqueueNDRangeKernel(CommandQueue, m_Kernel, 1,
     NULL, &globalWorkSize, LocalWorkSize, 0, NULL, NULL
    );
V_RETURN_CL(clErr, "Error executing kernel!");
```

After this command is executed, the results should reside in the device memory, on the address referenced by `m_dC`. To read the data back, issue a reading command:

```
clErr = clEnqueueReadBuffer(CommandQueue, m_dC, CL_TRUE,
    0, m_ArraySize * sizeof(int), m_hGPUResult, 0, NULL,
     NULL);
```

Note that this is a blocking call, as opposed to the recent memory writing example. We need to have the valid data in the resulting buffer before proceeding, therefore, the host must wait until the device completes the execution of the command.

Compile and run your code again. If everything went fine, you should see the following output:

```
[...]
Computing CPU reference result...DONE
Computing GPU result...Executing 1048576 threads in 4096
    groups of size 256
DONE
GOLD TEST PASSED!
Computing CPU reference result...DONE
Computing GPU result...Executing 1048576 threads in 2048
    groups of size 512
DONE
GOLD TEST PASSED!
[...]
```

Your kernel was executed two times, once having 256, then 512 *work-items* in a *work-group*. The results you copied back to the `m_hGPUResult` buffer are automatically compared to the reference solution, computed on the CPU. If your implementation is correct, both tests passed, and your solution for the first assignment is compete.

4.7 Measuring Execution Times

Until now, we did not say anything about optimizing the performance of the application, however, this is one of the most important things you should always spend time on once you have a working, algorithmically correct solution. There can be multiple alternative algorithms to solve a problem, and even a specific algorithm can be implemented in a few different ways. The performance of the application is not only influenced by the theoretical complexity of the algorithm, but also by practical, hardware-dependent factors.

The most basic way of measuring the performance of our implementation is to measure its execution time. We are usually interested in the execution time of a given kernel: we measure the system time before we start the kernel, and measure it after the kernel execution is finished. The difference between the two times will give us the kernel execution time. In the solution, you will find a `CTimer` class to perform this measurement, so the first attempt would result in a similar way:

```
CTimer timer;
timer.Start();
    clErr |= clEnqueueNDRangeKernel(CommandQueue, Kernel,
         Dimensions, NULL, pGlobalWorkSize,
        pLocalWorkSize, 0, NULL, NULL);
timer.Stop();
double ms = 1000 * timer.GetElapsedTime();
```

There are two problems with this code. The main issue is that the kernel will not be executed immediately when we call the `clEnqueueNDRangeKernel()` function, but it only enqueues a command to the command queue. Second, since the call is asynchronous, it immediately returns, so even if the kernel was run immediately, we would stop the timer while the kernel is still running. We should also mention, that sometimes the execution of the kernel is really fast, and the built-in timer of the operating system has a finite (not too high) resolution. In order to measure the execution time more accurately, it is better to execute the examined kernel *N* times, then divide the measured time interval with *N*. The higher N is, the closer we can get to the real execution time.

Using this reasoning, the following code snippet measures the execution time of a given kernel properly:

```
CTimer timer;
cl_int clErr;

//wait until the command queue is empty... iniefficient
    but allows accurate timing
clErr = clFinish(CommandQueue);

timer.Start();

//run the kernel N times
for(unsigned int i = 0; i < NIterations; i++)
{
  clErr |= clEnqueueNDRangeKernel(CommandQueue, Kernel,
      Dimensions, NULL, pGlobalWorkSize, pLocalWorkSize,
       0, NULL, NULL);
}
//wait until the command queue is empty again
clErr |= clFinish(CommandQueue);

timer.Stop();

if(clErr != CL_SUCCESS)
{
  cout<<"kernel execution failure"<<endl;
  return -1;
}

double ms = 1000 * timer.GetElapsedTime() / double(
    NIterations);

return ms;
```

Navigate to the `RunKernelNTimes` function in your solution, and type the above code to the fuction's body. From now you have a function that measures the execution time of any kernel you provide to it. Update your previously written code for the first assignment, to measure the performance of your kernel.

4.8 Evaluation

- Management of the device and context (1 point)
- Management of memory (3 points)
- Kernel implementation and execution (3 points)
- Timing of the program: use 4 different vector sizes and 4 different work-group sizes and compare the relative timings (**in a graph**). You will be asked to provide some explanation during the evaluation. (3 point)

5 Assignment 2: Reorganizing Memory Access Through Local Memory

In this assignment you will learn about utilizing the *on-chip local memory* to improve parallel memory access efficiency of your application. The problem we need to solve sounds trivial to implement on the GPU at the first moment: instead of vectors, we are given a two-dimensional array of floats (e.g. an image) that we need to rotate with 90 degrees. Using a 1D linear array to represent the 2D float-image we can index the elements using a single index of `y * array_width + x`.

A straightforward CPU-implementation of rotating the picture would look like this:

```
//Rotate clockwise
for(unsigned int x = 0; x < m_SizeX; x++)
{
  for(unsigned int y = 0; y < m_SizeY; y++)
  {
    // MR(sY - y - 1, x) = M(x, y)
    m_hMR[ x * m_SizeY + (m_SizeY - y - 1) ] = m_hM[ y *
        m_SizeX + x ];
  }
}

//Rotate counterclockwise
for(unsigned int x = 0; x < m_SizeX; x++)
{
  for(unsigned int y = 0; y < m_SizeY; y++)
  {
    // MR(y, Sx - x - 1) = M(x, y)
    m_hMR[ (m_SizeX - x - 1) * m_SizeY + y ] = m_hM[ y *
        m_SizeX + x ];
  }
}
```

where `m_hMR` is a linear array containing the rotated float-matrix, and `m_hM` is the original matrix. Note, that if the dimensions of the original matrix is $X \times Y$ then the rotated result will be an $Y \times X$ matrix, so you will have to carefully index matrix elements.

During the last meeting you have been assigned to one of two groups, **A** or **B**, that have slightly different task:

- **Group A:** Rotate the given matrix 90 degrees **clockwise** (right).
- **Group B:** Rotate the given matrix 90 degrees **counterclockwise** (left).

Within the same solution as for the previous assignment open the code of the `CMatrixRotate.h`. It follows the same structure as `CSimpleArrays`. The CPU-based solution is already implemented for your reference. As the very first step, go to `ComputeCPU()` method, and *uncomment the section related to your group*. Now you have a reference solution that you will need to make sure your OpenCL implementation is correct.

The code of this assignment class is just a backbone, which you have to extend according to the guidelines here and comments in the code. Proceed as follows:

- **Memory allocation** Create two linear float arrays on the device, which you will use as the input and output of your rotation-kernel. The dimensions of these arrays should be the same as the already existing `m_hM` and `m_hMR` arrays. Implement the allocation of these arrays in the `InitResources()` method, and release them in the `ReleaseResources()` method. The output array should be write-only, the input (original) array should be read-only for the kernels.

- **Copy input data to device** Using the host pointer `m_hM`, copy the original matrix to your device memory, using a *non-blocking* call.

5.1 Naïve Implementation

Our experience form the previous assignment already allows us to quickly implement the first version of the kernel that executes the matrix rotation. In this kernel, each thread would be responsible for loading exactly one matrix element into a local variable, and writing it to the new location (according to the rotated index) directly to the output array in the global memory. Create a new file in the root directory of the project and name it `MatrixRot.cl`. Use the following function header for your kernel:

```
__kernel void MatrixRotNaive(__global const float* M,
    __global float* MR, uint SizeX, uint SizeY)
{}
```

As you can see, the first parameter is an input parameter (as it is constant) and the results should be written to the output array passed in the second attribute. Since the problem is now two-dimensional, you will run this kernel in a two-dimensional *NDRange*, so you will receive two size attributes as well. To access a specific element in the matrix, use the 2D global index of *work-items*.

```
//get the global index of the thread
int2 GID;
GID.x = get_global_id(0);
GID.y = get_global_id(1);
```

Extend the kernel code to rotate the matrix properly.

Create a program object using this new source code, build it, and create a kernel object for the `MatrixRotNaive` function. Now you can run the kernel and read back its results from the output array. Proceed in the same manner as shown in the previous example:

- **Compile kernel** Create a program object using this new source code; build it and create a kernel object from the `MatrixRotNaive` function.
- **Set kernel args** The kernel has four arguments, so bind the appropriate values to these arguments.
- **Run the kernel** First you will need to determine the necessary work size to set up the *NDRange*, this time in two dimensions. Use the helper function for both dimensions in the following way:

```
//detemine the necessary number of global work items
size_t globalWorkSize[2];
size_t nGroups[2];

globalWorkSize[0] = GetGlobalWorkSize(m_SizeX,
    LocalWorkSize[0]);
globalWorkSize[1] = GetGlobalWorkSize(m_SizeY,
    LocalWorkSize[1]);

nGroups[0] = globalWorkSize[0] / LocalWorkSize[0];
nGroups[1] = globalWorkSize[1] / LocalWorkSize[1];
cout<<"Executing ("<<globalWorkSize[0]<<" x "<<
    globalWorkSize[1]<<") threads in ("<<
        nGroups[0]<<" x "<<nGroups[1]<<") groups
            of size ("<<
        LocalWorkSize[0]<<" x "<<LocalWorkSize
            [1]<<")."<<endl;
```

Using the resulting dimensions and execute the kernel with the `clEnqueueNDRange` command.

- **Read back results** Create a pointer on the host, called `m_hGPUResultNaive`. Using `clEnqueueReadBuffer`, read back the output of the naive kernel using a blocking call.

Finally, uncomment the code section in the `main()` function so that is also creates an instance of the `CMatrixRotate` class:

```
int main(int argc, char** argv)
{
  if(InitContextResources())
  {
  [...]
    //task 2: matrix rotation
    cout<<"Running matrix rotation example..."<<endl<<
        endl;
    {
      size_t LocalWorkSize[3] = {32, 32, 1};
      RunAssigment(CMatrixRotate(2048, 2048),
          LocalWorkSize);
    }
  }
  CleanupContextResources();
}
```

If implemented correctly, the CPU-based evaluation of your first kernel results should succeed.

5.2 Optimizing the Kernel Using Local Memory

You might wonder why did we always refer to the recent kernel as "naïve". In fact, we can only answer this question if we have a deeper understanding of how the memory controller on GPUs works. As each thread executes the same piece of code on different elements of data in parallel, the threads obviously need to load and store data elements from different locations. As a streaming multiprocessor has a single interface to the global memory, each per-thread memory operation is executed through this shared "gate". Instead of executing a load / store operation for a single thread at once, the memory interface operates on words of 32-, 64- or 128-byte length. It means that when you have e.g. 32 threads accessing float values aligned in a continuous segment of 128 bytes, accessing them can be performed within a single memory transaction. If the addresses are not properly aligned, the memory manager will issue 32 memory instructions, one for each thread.

This feature is called memory coalescing, and can have a significant impact on application's performance, as the delay of a single load / store operation can take 400 to 600 cycles. In fact, coalescing has more constraints than described here, for details please refer to documentation of the NVIDIA CUDA architecture.

Obviously, the naïveimplementation does not coalesce all the memory accesses, and the implementation is *memory bandwidth limited*. Unfortunately, there is a fundamental change in the memory access pattern of the work-items when loading values from the original matrix, and storing values to the rotated matrix. Those threads, which accessed the neighboring values during loading - thus, they could be coalesced by the hardware - will not address continuous words during storing, since horizontal lines in the original matrix become vertical in the result.

The problem is impossible to overcome without using some kind of intermediate storage, allowing the threads to reorganize memory accesses, so that both loading / storing can be horizontally aligned. The key idea is to split the whole rotation into two steps. First, we load a small tile, for example 32×32 elements, into some intermediate storage. After *rotating the tile locally* we will write the elements row-by-row again. As this intermediate storage is small, we can keep it in a fast on-chip memory, the *OpenCL local memory*,

where the latency is comparable to that of the registers. This way the input and output is guaranteed to be aligned, so our performance is significantly increased compared to implementations without the local memory.

After understanding the theory let us put it into practice, and evaluate through measurements whether it really works. Create a new kernel in the code of you program source, using the following header:

```
__kernel void MatrixRotOptimized(__global const float* M,
      __global float* MR, uint SizeX, uint SizeY,
              __local float* block)
```

The only difference compared to the naïve kernel is that a fifth parameter appeared, with a new prefix, `__local`. This prefix tells the compiler that the given pointer refers to the local, on-chip memory. The kernel can both read and write this memory using random access patterns, and it is not accessible from the host. Prior to running the kernel, we will have to allocate local memory for it.

Now perform the coalesced load of data into this local block of memory. Each *work-group* is assigned to one tile of the large matrix and each *work-item* in the *work-group* will load exactly one element from this tile into the temporary local buffer. In the implementation it should look like this:

```
//get global index of the thread (GID)
[...]

//get local index of the thread (LID)
int2 LID;
LID.x = get_local_id(0);
LID.y = get_local_id(1);

block[ LID.y * get_local_size(0) + LID.x ] = M[ GID.y *
     SizeX + GID.x ];

//we need to wait for other local threads to finish
     writing this shared array
barrier(CLK_LOCAL_MEM_FENCE);
```

The code snippet uses the local id on the left side of the equation while the global id on the right, so it maps one unique tile of the input matrix to the local buffer.

Now comes the challenging part of the assignment. You have to write back your results to the large result array in a way that it is correctly rotated, and *the horizontally neighboring work-items write horizontally neighboring addresses.* So we can say that now the output addresses for each thread are fixed, and depending on your rotation direction, you will have to find out how to index the local memory. Implement the remaining part of the code to finish the rotation and produce equivalent results to the naïve kernel.

Create a kernel object for the `MatrixRotOptimized` function as well, and bind its parameters as usual. As this is the first time using the local memory, we show you how to bind the fifth parameter of the kernel, which performs the local memory allocation in the same time:

```
//allocate shared (local) memory for the kernel
clErr = clSetKernelArg(m_OptimizedKernel, 4,
     LocalWorkSize[0] * LocalWorkSize[1] * sizeof(float),
      NULL);
V_RETURN_CL(clErr, "Error allocating shared memory!");
```

As the size of the allocated local memory depends on the local work size, you should add this code snippet to your `ComputeGPU()` method, before the kernel execution command.

You are now able to finish the assignment by reading back the result of the kernel to the `m_hGPUResultOpt` host array. If your implementation is correct, the validation will find the result of this second kernel equivalent to the naïve one, but you should experience relevant speedup when measuring the execution time.

5.3 Evaluation

- Naive implementation (4 points)
- Allocation of the local memory (1 point)
- Loading into local memory using tiles (2 points)
- Tile-wise implementation of the matrix rotation (3 points)