International Journal of Networking and Computing – www.ijnc.org
ISSN 2185-2839 (print) ISSN 2185-2847 (online)
Volume 2, Number 2, pages 188–205, July 2012

Invitation to a Standard Programming Interface for Massively Parallel Computing Environment: OpenCL

Shinichi Yamagiwa
Faculty of Engineering, Information and Systems,
University of Tsukuba / JST PRESTO,
Tsukuba, Ibaraki, 305-8573, Japan

Received: April 18, 2012
Revised: June 11, 2012
Accepted: June 29, 2012
Communicated by Koji Nakano

Abstract

Multicore/manycore architecture accelerates demand for a new programming environment to utilize the massive processors integrated in an LSI. GPU (Graphics Processing Unit) is one of the typical hardware environments. The programming environments on GPU are traditionally vendor-/hardware-specific, where complicate the management of uniform programs that access computing resources of the massively parallel platform. The recently released OpenCL is expected to become a standard for providing a uniform programming environment for the heterogeneous processors from different vendors. This tutorial paper introduces the overview of the OpenCL that motivates the programmers who are going to program the massively parallel hardware or who migrates the programming method from another vendor specific programming interface to the OpenCL. This paper explains the characteristics of the OpenCL interface with describing in detail the basic structures used in the program. Moreover, this paper discusses performance aspects to evaluate advanced programming techniques that improve the performance of the OpenCL applications.

Keywords: GPGPU, Stream Computing, OpenCL

1 Introduction

Parallel programming [14] has needed special consideration for the programmer to implement concurrent execution of multiple processes with communication or virtual shared memory concept in a very rich hardware environment [6] such as cluster computers or supercomputers. In this decade, multiprocessor architecture shifts to *multicore* technique that integrates multiple CPU cores into an LSI. This technology shift provides a new environment for an instant parallel processing environment in a personal desktop machine. Especially, GPU (Graphics Processing Unit) provides originally a fast graphics process to draw complicate images. However, the recent advancement in GPU technology integrates many small processors in an LSI [7]. This *manycore* architecture attracts programmers who need intensive computing to the GPU-based computing (GPGPU: General Purpose computing on Graphics Processing Units) field (http://gpgpu.org/) due to its highly parallel processing potential. However, such massively parallel environment is completely new for current

[0]This is an abstract footnote

parallel programming methods. Therefore defining a new programming method on the platforms was indispensable for utilizing the hardware resources.

From the aspect of performance, a single GPU IC has become to achieve 1 TFLOPS such as the NVIDIA Tesla 2050. Various fields [13] such as condensed matter physics [18] have already taken advantage of the power of the GPU. However GPU programming style follows the *Stream Computing* programming paradigm, which is significantly different than the traditional CPU programming paradigm.

The Stream Computing [4] is a programming paradigm in which the data is processed in a continuous manner, as a unique data-flow of information. This data-flow is called *stream*. These streams are usually composed by a set of massive data. Each data unit is potential availability to be processed in parallel because dependency among the data units does not exist in a stream. The operations applied to each element in a stream are packed into a function called *kernel*. The benefit of this paradigm provides a high parallelism on massively parallel processor architecture. Thus, the kernel is applied to computations for units in input data streams to generate ones of output data streams in parallel. The kernel is assigned to each processor in a manycore LSI and finally multiple kernels works concurrently to implement highly intensive computation.

The different programming style from the traditional CPU-based programming method has caused the development of various GPGPU environments [16] [2] that eliminate the mismatches of programming concepts and facilitate to implement the task for the stream-based applications. Nowadays, the most mature stream-based programming environment is the NVIDIA-specific CUDA (Compute Unified Device Architecture) [11], which is one of the most popular GPGPU tool for highly parallel programming. However, CUDA is a platform-dependent interface, where it is dedicated to the NVIDIA GPUs. For this reason, a new GPGPU environment called *OpenCL* [12] has been developed to provide a standard interface for stream-based programming. This tutorial paper presents introduction to the OpenCL and the programming techniques to help OpenCL starters to enroll in the OpenCL environment.

This paper will show the overview of the OpenCL in section 2 that explains basic programming concepts and data structures used in the programming environment. Section 3 will introduce how to code a small kernel example to implement a concurrent invocation of kernel. To improve performance using additional features in executing hardware, section 4 proposes advanced techniques using special interfaces of the OepnCL. Section 5 will discuss the performance differences among different coding methods using the advanced features. Finally, section 6 concludes this tutorial paper and introduces some advanced research projects with the future direction.

2 OpenCL Overview

The OpenCL is the first open royalty-free standard for general purpose parallel programming across heterogeneous processors. It was initially proposed by Apple and finally developed by the Kronos Group, which released the OpenCL 1.0 version in 2008. The OpenCL supports various platforms that include multiple processors applying the stream-based computing manner. The supported system implies to accept one or more data streams, to calculate it using the kernel and to generate one or more output data streams. In this tutorial paper, we assume an environment where a personal computer has a GPU. Therefore, the programming environment is configured on the PC.

2.1 OpenCL Software Development Kits

We have to download the drivers for our GPU in order to give OpenCL support. We highly recommend to install one of the available Software Development Kits (SDK) for OpenCL programming that will help us to start programming OpenCL applications. Here, we briefly describe the characteristics of the SDKs released from NVIDIA, ATI and Intel respectively.

Depending on the vendor of our GPU, we can install the NVIDIA SDK or the ATI stream SDK. If your GPU is not compatible to the OpenCL, we still can develop OpenCL programs on Intel CPUs using the Intel OpenCL SDK. Moreover, the ATI stream SDK provides OpenCL support to

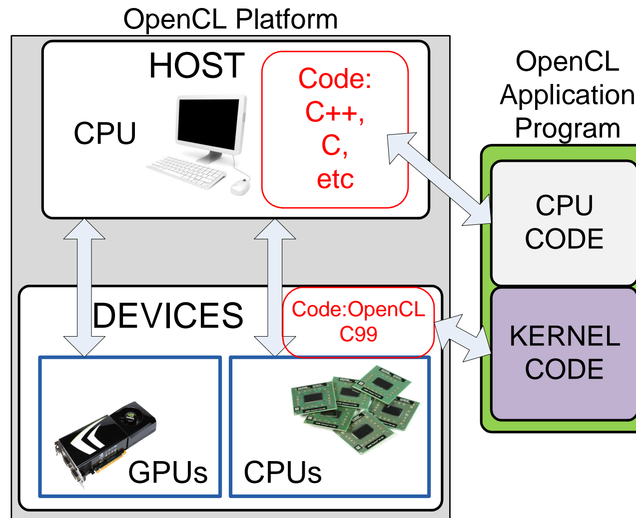


Figure 1: OpenCL Simplified Platform Model

any CPU even without ATI GPUs. All of these SDKs are available for both Windows and Linux OS.

2.1.1 NVIDIA CUDA Toolkit 4.0

The NVIDIA CUDA Toolkit 4.0 and the NVIDIA drivers for OpenCL support can both be found at [10]. After downloading and installing the GPU drivers, it is recommended to download and read all the documentation provided by the CUDA Toolkit, and also to download the *GPU Computing SDK code samples*. This package includes many OpenCL program examples that we can just compile and run, or even used them as templates to code our own OpenCL programs. Moreover, the CUDA Toolkit also includes an OpenCL visual profiler that dynamically measures the utilization of the GPU resources of OpenCL program, providing us useful information to optimize the program.

2.1.2 ATI stream SDK

The ATI stream SDK with OpenCL support can be found at [1]. As mentioned before, this SDK provides OpenCL support not only on ATI GPUs but also on CPUs. This kit includes OpenCL code samples and documentation to let us start it on the environment. In addition the package includes a GPU visual profiler and another useful tool called *KernelAnalizer*. This software can be used to check syntax and semantics errors in an OpenCL kernel program. Moreover, the KernelAnalizer also can translate an OpenCL kernel program to available assembly languages that can be executed on various ATI GPUs.

2.1.3 Intel OpenCL SDK

The Intel OpenCL SDK, which can be found at [5], allows us to create OpenCL applications and run them on Intel processors. The SDK is available on both Windows and Linux and includes useful documentation to let us start programming optimizing OpenCL programs on Intel processors. The package also includes a few OpenCL samples and an offline compiler that is similar to the KernelAnalazer included in the ATI SDK. This tool can not only compile an OpenCL program to check errors but also can translate an OpenCL program to an assembly code or LLVM (Low Level Virtual Machine) code. The LLVM is a compiler infrastructure used to improve the optimization of programs written in various programming languages such as C, C++, Fortran, etc.

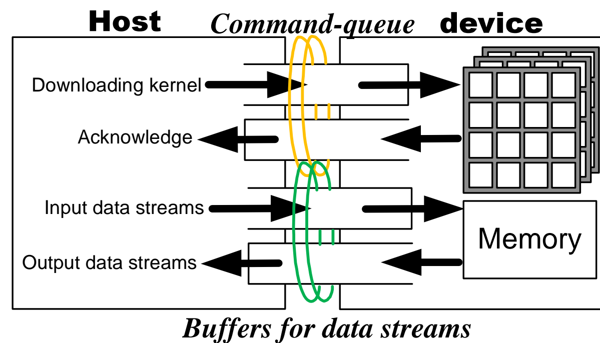


Figure 2: Command queue and buffers to exchange commands and data streams between a host and a device.

2.2 OpenCL Runtime

In order to explain in detail what is behind of the OpenCL, let us start by roughly explaining its platform model shown in Figure 1. In this model, a *host* is connected to one or more OpenCL *devices*. For example, the host is the PC and the device is a GPU or a CPU. The host must control the execution timing of kernel using dedicated functions to send the kernel to the device.

When we say the "OpenCL program", it is the kernel program downloaded to the device and executed on it. The syntax is compatible to the C99 standard. Some language directives and variable types are added to the syntax. Therefore, the host needs a separate program that controls the timing to execute the kernel program. To implement this mechanism, the host calls runtime functions with linking to the OpenCL library using C/C++ interface. The kernel is compiled by calling a function from the library during the host program execution. Comparing to the CUDA environment, the CUDA provides a compiler for the entire program including the kernel. Therefore, although the CUDA environment is a compiler-oriented programming interface, the OpenCL one is a library-based environment. Thus, the OpenCL application can be invoked on any platform that supports C/C++ liking to the functions for kernel execution. If the host controls different devices combining with multiple OpenCL libraries, an OpenCL application can control single or multiple kernels on different devices such as on a GPU and on a CPU. Thus, any platform vendor is able to provide an OpenCL environment if a library package for the kernel execution is provided and the heterogeneous device environment is seamlessly gathered in a single interface.

Now let us consider how to manage the device from the host. The host program performs the flow defining resources included in the device. First, the host program needs to defines a structure called *context*. The context defines the runtime environment including kernels, devices, memory management and various execution parameters. The context also includes an object called *command-queue*, which is the communication path between the host and the device to exchange the commands against the downloaded kernel as shown in Fig 2. Using a command-queue, the host submits commands to the device. For example, the commands include a request to execute a kernel in a specific device. In summary, a host OpenCL program defines a context with one or more devices, and each device has a command-queue associated to send commands to the device. Now in the next section let us see how OpenCL manages the execution of a kernel in a device.

2.3 Taking a glance at an OpenCL program

Here, we see a vector summation kernel program as shown in Figure 3. This program performs summations of units of `a` and `b` and generates a stream `c`. For example, when the stream `a` consists of 1.2, 3.4 and 5.3, and the stream `b` consists of 1.5, 5.2 and 6.1, the output stream `c` becomes a sequence of 2.7, 8.6 and 11.4. In this case `iNumElements` is 3. The kernel program is assigned to a device in parallel. This parallelization mechanism will be explained in the next section. This code can be saved in a file with the extension `.cl` or to a string variable in the host program.

```
__kernel void VectorAdd(__global const float* a,
                        __global const float* b,
                        __global float* c,
                        int iNumElements)
{
      int id = get_global_id(0);
      if  (id >= iNumElements)
      {
            return;
      }
      c[id] = a[id] + b[id];
}
```

Figure 3: VectorAdd Kernel Code.

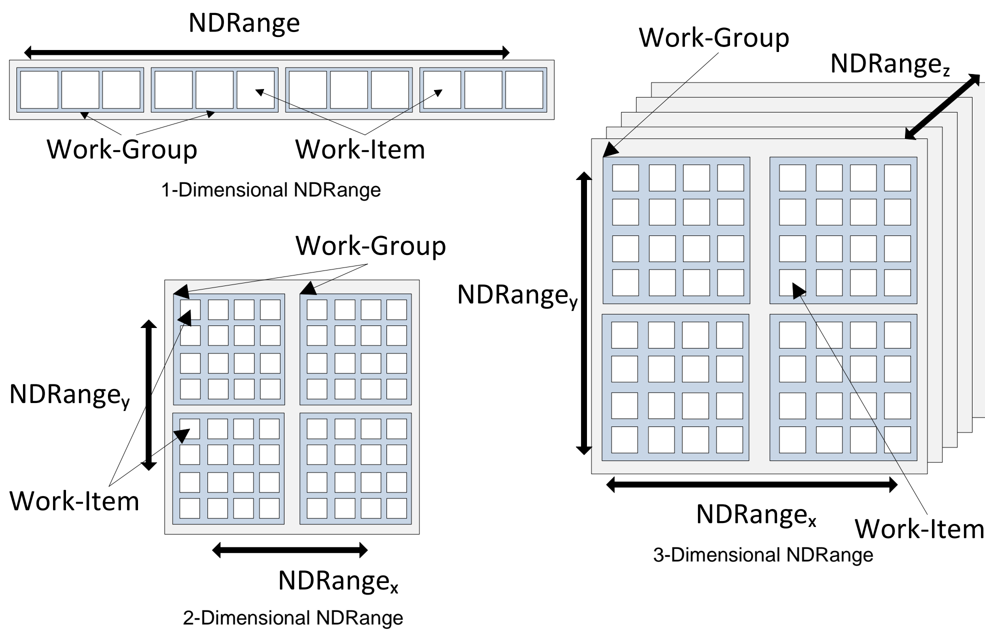


Figure 4: N-Dimensional NDRange.

On the other hand, the host code is also prepared to control the execution timing of the kernel. To give the input streams and to receive the output data streams form the kernel, the host program needs to add the buffers to exchange data between the host and the device. Moreover, the parallelization granularity is also defined by the host program. These controls will be explained from the next step.

The full code of this example is included in the *VectorAdd* folder of the NVIDIA SDK, and part of the code is illustrated in Figure 7. Before we see the host code, let us understand the execution mechanism of the kernel.

2.4 OpenCL Execution Model

The OpenCL defines an index space called *NDRange* that is created when the host program submits a kernel for execution. For each point of the NDRange an instance of the kernel is executed. This kernel instance is called *work-item* which is identified by a global ID in the index space. All the work-items execute the same kernel program. However the execution pathway through the code and the data operated by the work-item may vary per work-item. For example, depending on the global ID value of a work-item, the execution pathway of the work-item may enter or not into a conditional clause.

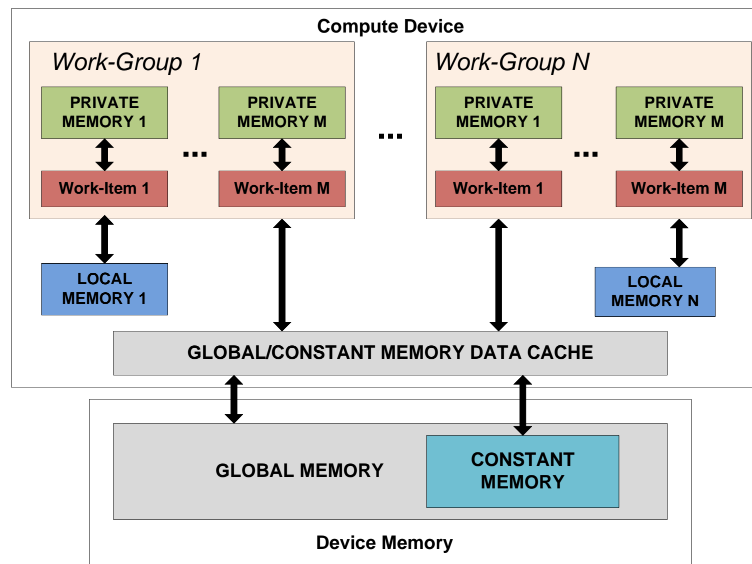


Figure 5: OpenCL Memory Regions.

Moreover, work-items can be organized into *work-groups*. A unique work-group ID is assigned to each work-group. In addition a local ID is assigned to each work-item within a work-group. Therefore, work-items can be identified by its global ID or by a combination of its local ID and work-group ID. In summary, each work-item is assigned two IDs (*global* and *local*) and each work-group is assigned an ID.

The size of the index space, i.e. the total number of work-items, is defined by the NDRange. Because the NDRange can be one, two or three dimensional, each work-item ID will be a N-dimensional tuple, where N is the dimension of the NDRange. Figure4 shows examples of NDRange.

In this example each small square represents a work-item and each of the four large squares that include the work-items represents a work-group. The NDRange is used to identify and to control all the work-items that are executed. For instance, in a program that performs the matrix operation $A+B=C$, each work-item of the NDRange performs a sum operation to calculate an element of the matrix C. To explain with actual indices, the work-item with the ID (0,0) will perform the operation $A[0][0]+B[0][0]=C[0][0]$. Because we have control of all the work-items using the NDRange and the IDs, the sum operation inside the kernel function is just $A[x][y]+B[x][y]=C[x][y]$ where x and y are the IDs of each work-item specified by the NDRange.

2.5 OpenCL Memory Model

OpenCL divides the memory domains used by each work-item into four distinct regions as shown in Figure 5:

- **Global memory**: Any work-item can read/write to this region.
- **Constant memory**: Region of the global memory that obtains constant data during the execution of the work-items. This region is read only.
- **Local memory**: Region that is placed locally in a work-group, which means that is shared only by all the work-items in an specific work-group.
- **Private memory**: Region that is placed privately in a work-item. Any variable defined in this region is accessible by a specific work-item.

The host program uses the OpenCL API to create memory objects in global memory and then enqueue memory commands to transfer data between the host memory and the device memory. For

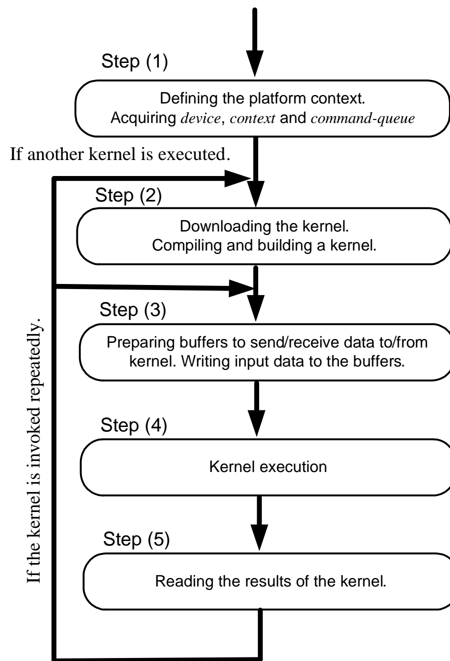


Figure 6: Host program flow.

this data transfer, there are two ways using the API in the host program: explicitly copying data from host to device, or mapping and unmapping region that corresponds to a memory object. In the former case, the host has to submit commands to transfer data between a device memory object and host memory, which can be performed by blocking or by non-blocking operations. In the latter case, the host can map a region from this memory object into its address space. Once a region from the memory object has been mapped, the host is allowed to write/read to/from this region. When the host finishes all the accesses, the region is unmapped.

According to the usage of functionalities and resources defined in the OpenCL device explained above, the host program configures the command-queue and the buffers to send/receive the kernel controls and the data streams against the device. After invoking the kernel program by the host, it is parallelized automatically into the work-items with the NDRange definition. The parallelism can be increased easily by assigning a large number of NDRange. For example, because NVIDIA Tesla C2050 has 480 stream processors (small processors that execute kernels), the kernel is assigned to the processors. Thus, the very high parallelism is able to be implemented on a device.

Now we have understood how the OpenCL program works roughly. Let us get into the core of the world with an experimental environment.

3 Getting started with OpenCL

In this section we will explain in detail a simple OpenCL program example for the host side to understand the usual OpenCL program structure and some of the most used OpenCL API functions. Let us divide the host program lines into the different steps that almost every OpenCL program must follow. Because the structure of a host program is very repetitive, it can be used for other OpenCL programs. Figure 6 shows the steps to execute the kernel using the OpenCL library. In the following sections, let us see the details of the steps. The main routine of the steps is to download the kernel and to execute it configuring the input data streams, and finally to receive the generated results from the kernel. The detailed functions and the categories are listed in Figure 7.

```
STEP(1)//Get an OpenCL platform
ciErr1 = clGetPlatformIDs(1, &cpPlatform, NULL);

//Get the devices
ciErr1 = clGetDeviceIDs(cpPlatform, CL_DEVICE_TYPE_GPU,
                        1, &cdDevice, NULL);

//Create the context
cxGPUContext = clCreateContext(0, 1, &cdDevice, NULL,
                               NULL, &ciErr1);

// Create a command-queue
cqCommandQueue = clCreateCommandQueue(cxGPUContext,
                                      cdDevice, 0, &ciErr1);

STEP(2) // Create the program
cpProgram = clCreateProgramWithSource(cxGPUContext, 1,
                               (const char **)&cSourceCL,
                               &szKernelLength, &ciErr1);

// Build the program
ciErr1 = clBuildProgram(cpProgram, 0, NULL, NULL,
                         NULL, NULL);

// Create the kernel
ckKernel = clCreateKernel(cpProgram, "VectorAdd", &ciErr1);

STEP(3)// Allocate the OpenCL buffer memory objects for
    source and result on the device GLOBAL MEM
cmDevSrcA = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY,
                           sizeof(cl_float)*szGlobalWorkSize,
                            NULL, &ciErr1);
cmDevSrcB = clCreateBuffer(cxGPUContext, CL_MEM_READ_ONLY,
                           sizeof(cl_float)*szGlobalWorkSize,
                            NULL, &ciErr2);
cmDevDst = clCreateBuffer(cxGPUContext, CL_MEM_WRITE_ONLY,
                           sizeof(cl_float)*szGlobalWorkSize,
                           NULL, &ciErr3);

// Write of data to GPU device
ciErr1 = clEnqueueWriteBuffer(cqCommandQueue, cmDevSrcA,
                              CL_FALSE, 0,
                          sizeof(cl_float)*szGlobalWorkSize,
                              srcA, 0, NULL, NULL);
ciErr1 |= clEnqueueWriteBuffer(cqCommandQueue, cmDevSrcB,
                               CL_FALSE, 0,
                          sizeof(cl_float)*szGlobalWorkSize,
                               srcB, 0, NULL, NULL);

STEP(4) // Set the Argument values
ciErr1 = clSetKernelArg(ckKernel, 0, sizeof(cl_mem),
                        (void*)&cmDevSrcA);
ciErr1 |= clSetKernelArg(ckKernel, 1, sizeof(cl_mem),
                        (void*)&cmDevSrcB);
ciErr1 |= clSetKernelArg(ckKernel, 2, sizeof(cl_mem),
                        (void*)&cmDevDst);
ciErr1 |= clSetKernelArg(ckKernel, 3, sizeof(cl_int),
                        (void*)&iNumElements);

// Launch kernel
ciErr1 = clEnqueueNDRangeKernel(cqCommandQueue, ckKernel,
                               1, NULL, &szGlobalWorkSize,
                          &szLocalWorkSize, 0, NULL, NULL);

STEP(5)// Read of results
ciErr1 = clEnqueueReadBuffer(cqCommandQueue, cmDevDst,
                         CL_TRUE, 0,
                         sizeof(cl_float)*szGlobalWorkSize,
                          dst, 0, NULL, NULL);
```

Figure 7: VectorAdd Host Code.

3.1 OpenCL Initialization

First of all we need to find an available device in our system and create the necessary OpenCL structures (context, command-queue). This can be done with the functions below:

- `clGetPlatformIDs`: This function returns a list with the available OpenCL platform (NVIDIA, ATI, Intel, etc.) in the system. In this case, because the first argument is `1`, we only get one platform in the list.
- `clGetDeviceIDs`: This function returns a list with the available devices on the selected platform. In this case, the function returns only one device of type GPU.
- `clCreateContext`: This function creates a new context for the target device. This structure stores all the OpenCL resources (buffers, command-queues, kernel program, etc.) used by the device.
- `clCreateCommandQueue`: This function creates a command-queue for the selected context for the target device. This structure is used to send commands (a kernel execution, data transfers, etc.) from the host to the device.

Notice that if an OpenCL API function fails, we will not receive any information about the error. However, all the functions would return an error number, some of them as the function returning value and others as a parameter. If the function was executed successfully, a `CL_SUCCESS` value is returned. Otherwise, one of the possible errors listed in the OpenCL header is returned.

3.2 Compiling Kernel Program

The next step is to create and compile the *kernel*, which is the function executed on the device by all the work-items. OpenCL can read the code of the kernel program either from a file with `.cl` extension or simply from a string in the host code. These are the functions used for the operations:

- `clCreateProgramWithSource`: This function creates a program object for the selected context, loading the source code stored in the string `cSourceCL` into the program object.
- `clBuildProgram`: This function compiles and links the program object created by the previous function. In this function the syntax and the semantic of the kernel program is checked.
- `clCreateKernel`: This function creates a kernel object from a successful built program object. We must specify in the second argument the name of the kernel function, in the example, `VectorAdd`.

3.3 Creation and Initialization of Buffers

We need to create, to initialize and to send the data from the host to the device. The conventional C function `malloc` is used to create the buffers in the host side. We also need to create the buffers that will allocate the data on the device side. These are the functions used in this step:

- `clCreateBuffer`: This function creates a buffer object on the global memory of the device.
- `clEnqueueWriteBuffer`: This function performs a data transfer from a buffer allocated on the host side to a buffer allocated on the device side.

3.4 Kernel Execution

After all the previous steps, the kernel program is almost ready to be executed. The last step before the execution is to set the kernel arguments to link each of the kernel function parameters to either buffers on the device side or single scalar variables. When all the parameters are set, we can proceed to invoke the kernel. These functions are used for the operations:

- `clSetKernelArg`: This function sets the values of the kernel arguments to the data pointed by the last argument of this function.
- `clEnqueueNDRangeKernel`: This function sends a command to execute the kernel on the target device. Here we specify the dimension and the size of the NDRange.

3.5 Reading Back Output Data

The last step after the kernel execution finishes is to send back the data from the device side to the host side. This is done by the following function:

- `clEnqueueReadBuffer`: This function performs a data transfer from a buffer allocated on the device side to a buffer allocated on the host side.

All the previous functions are written in the host program of an OpenCL application.

Now we will take a look at the kernel program that will be executed on the target device. The full code of the kernel for the VectorAdd example is shown in Figure 3. Every kernel function in OpenCL must start with the directive `__kernel`. Then, each argument declared in the function is linked to a buffer allocated in the device memory. For each of these arguments we must specify the memory region where the buffer will be allocated. In this example, we store the input arrays `a` and `b` in the constant region using the directive `__global const`. The output array `c` will be allocated in the global region. For a scalar variable like `iNumElements` we do not need to explicitly specify the memory region because it is not a pointer to buffer.

In the kernel function the sum operation is executed for each element of the arrays. Before the operation, we use the embedded function (a function provided by the OpenCL C programming language) `get_global_id` to get the work-item ID inside the NDRange. Then each work-item performs a sum operation. Because the number of work-items may be greater than the size of the arrays, we use the conditional structure to assure that we will not try to access an invalid position in the array.

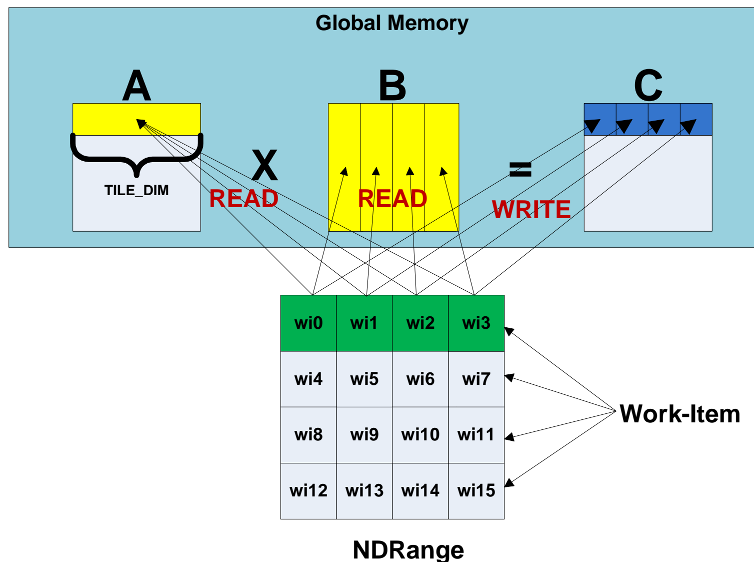
In this section we have explained how to set up the environment to program in OpenCL and we have described a simple OpenCL program to understand the basics of this language. However, OpenCL offers much more features, especially to improve the performance of our programs. The next section explains some of these characteristics.

4 Advanced Features

The basic structure of an OpenCL program shown in the previous section only uses a small number of the OpenCL features. Although just by using this limited number of functionalities correct OpenCL applications can be programmed, these applications will not reach the desirable performance level. In order to achieve the best performance available in our programs, we must deeply study all the features that OpenCL offers us to control issues that bring a large impact in the performance, such as memory accesses, data transfers between the host and the device, etc. In this section we will describe two techniques related to these issues: the use of local memory and a method to eliminate redundant data transfers [8].

4.1 Using Local Memory

As described in section 2.5, the GPU memory is divided to four regions: global, constant, local and private. In the CPU case the memory layers are abstracted away from the programmer as the multiple memory layers are virtually combined by a high level language compiler. However on the GPU we must know the details of the memory hierarchy in order to achieve the best performance in our applications. In section 2.5 we explained the differences regarding access visibility of the different memory domains. However they differ also in access times and sizes. Global memory is the largest and the slowest memory. Normally a GPU has GBytes of global memory. On the other hand, the local memory and the private memory are the fastest but are available only in KBytes. Therefore,



(a) Calculation process of Matrix Multiply using only global memory

```
__kernel void GlobalMatrixMultiply(
                                    __global float* a,
                                    __global float* b,
                                    __global float* c,
                                    int N)
{
      int row = get_global_id(1);
      int col = get_global_id(0);
      float sum = 0.0f;
      for(int i=0; i<TILE_DIM; i++){
            sum += a[row*TILE_DIM+i] * b[i*N+col];
      }
      c[row*N+col] = sum;
}
```

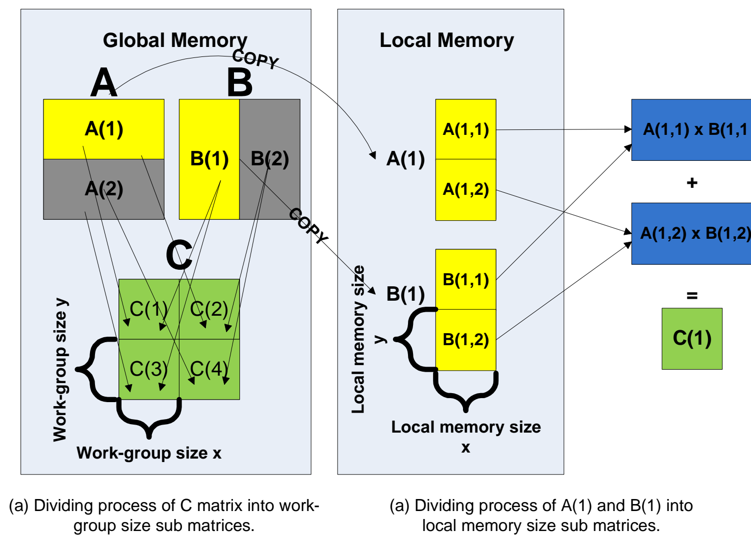
(b) Kernel of Matrix Multiply using only global memory

Figure 8: MatrixMultiply problem with global memory only.

as a legend rule in our kernel functions, we must minimize the read/write accesses to a large region allocated in the global memory by copying a small block of the region to the local memory and then processing the data in the local memory intensively. Using a *Matrix Multiplication* example let us see the detail of the optimization technique regarding the data access management.

The matrix multiplication problem without any optimization (i.e. using global memory only) is shown in Figure 8(a) and its OpenCL kernel code is listed in Figure 8(b). In this problem each work-item in the 2-dimensional NDRange calculates one element of the matrix C by reading one row of matrix A and one column of matrix B. The constant shown in Figure 8(a) called *TILE_DIM* is defined at the beginning of the kernel using `#define` statement. It figures the size of the tile in the problem (i.e. the size of a row of A and the size of a column of B). As shown in Figure 8(a), each row in A is read by a work-item for TILE_DIM times (in this example 4 times). The same issue happens with each column of B. Therefore, each element of the matrices A and B is read for TILE_DIM times from global memory which is the memory region with the slowest access time in the GPU memory hierarchy.

In order to optimize this program, we should decrease the number of accesses to global memory. This is done by copying the matrices A and B from the global memory to the local memory once, and then processing the data from the local memory. Therefore, the elements of the matrices allocated in the global memory are only read once from it and instead are read for TILE_DIM times from local memory.



(a) Dividing process of C matrix into work-group size sub matrices.

(a) Dividing process of A(1) and B(1) into local memory size sub matrices.

Figure 9: Process to divide the problem into sub-problems to use local memory.

However, this optimization implies a relevant problem: the size of local memory is significantly smaller than the size of global memory. Hence, if the size of the local memory is smaller than the one of the data allocated on the global memory, all the data cannot be stored to local memory. To solve this problem we use a technique that divides the whole problem into sub-problems that are small enough to fit into local memory. To understand this technique, let us take a look at Figure 9. The problem is divided into blocks and each block is read into the local memory. First of all we must remember that local memory is only shared by all the work-items in a work-group, and also that the size of a work-group is limited (256 or 512 work-items). Therefore, as shown in Figure 9(a), we must divide first the result matrix C into sub-matrices that fit into a single work-group and also break both A and B into sub-matrices. In this example, we can divide C into 4 sub-matrices (C(1), C(2), C(3) and C(4)). Therefore, A and B are divided into 2 sub-matrices (A(1), A(2), B(1) and B(2)). The values of matrix C are processed as shown in Figure 9(a) ($A(1) * B(1) = C(1), A(1) * B(2) = C(2)$, etc.). These four sub-problems are solved in parallel on different GPU cores.

If we focus on the first sub-problem, we observe that the sizes of the sub-matrices A and B are still greater than the space available in the local memory. Therefore, assuming that the local memory size is enough to fit half of matrices A and B, we divide this sub-problem into two sub-problems that fit into shared memory as shown in Figure 9(b). Thus, the elements of the sub-matrix C(1) are calculated in two steps.

In Figure 10 we can see the optimized kernel version of the matrix multiplication using the method above. First of all, we need to determine the start and the end of the two sub-matrices A and B. We use `get_group_id` and `get_local_id` functions to get the work-group ID and the work-item ID inside of the work-group respectively (Figure 10, Step 1). Then we calculate the limits of the sub-matrices by using the IDs and the witdh of the matrices *(wA,wB)* (Figure 10, Step 2). The value of the constant BLOCK_SIZE is the size in one dimension of the work-group. For instance, if the maximum size of a work-group is 256 work-items and we are using that size, BLOCK_SIZE would be 16 because the NDRange is 2-dimensional.

After the limits of the sub-matrices A and B are calculated, the first loop iterates through the sub-matrices. Inside the loop, each work-item copies an element of the matrices A and B from global memory to local memory (Figure 10, Step 3). Then, just right after this operation there is a *barrier* that synchronizes all the work-items in a work-group, in order to make sure that the sub-matrices are loaded before performing any calculation. Next, the work-item calculates the intermediate value of the matrix C (Figure 10, Step 4) that will become the final value after the last iteration of the main loop. After this operation we put another barrier to assure that the computation in the Step 4 is completed before loading two new sub-matrices of A and B in local memory.

```
__kernel void LocalMatrixMultiply(__global float* C,
                                  __global float* A,
                                  __global float* B,
                                  int wA, int wB){
    int bx = get_group_id(0);
    int by = get_group_id(1);
    int tx = get_local_id(0);
    int ty = get_local_id(1);

    int aBegin = wA * BLOCK_SIZE * by;
    int aEnd   = aBegin + wA - 1;
    int aStep  = BLOCK_SIZE;
    int bBegin = BLOCK_SIZE * bx;
    int bStep  = BLOCK_SIZE * wB;

    for (int a = aBegin, b = bBegin;
             a <= aEnd;
             a += aStep, b += bStep){
        __local float As[BLOCK_SIZE][BLOCK_SIZE];
        __local float Bs[BLOCK_SIZE][BLOCK_SIZE];
        As[ty][tx] = A[a + wA * ty + tx];
        Bs[ty][tx] = B[b + wB * ty + tx];
        barrier(CLK_LOCAL_MEM_FENCE);
        for (int k = 0; k < BLOCK_SIZE; ++k)
            Csub += As[ty][k] * Bs[k][tx];
        barrier(CLK_LOCAL_MEM_FENCE);}
    int c = wB * BLOCK_SIZE * by + BLOCK_SIZE * bx;
    C[c + wB * ty + tx] = Csub;}
```

STEP 1
STEP 2
Limits of the sub matrices of A and B
Copy to local memory
STEP 3
Synchronization points
STEP 4
STEP 5

Figure 10: Matrix multiply kernel function using local memory.

Finally the last operation to be done is to write the final value of the matrix C into the global memory where C is stored (Figure 10, Step 5).

The most difficult part of the technique above is to divide the problem into smaller parts that can fit into a work-group considering the limited size of the local memory. In some algorithms with a strong data dependency the programmer has to pay attention to the process of using the local memory because only the work-items in the same work-group can be synchronized. That is, work-items in different work-groups cannot be synchronized by using barriers. Therefore, the data accessed by work-items in different work-groups must be carefully updated in order to avoid the synchronization problem such as a data race condition.

4.2 Reducing Host-Device Data Transfers

Because the host is connected to the device via a peripheral bus (ex. PCI express) the data for the input streams of a kernel must be transferred from the host to the device before executing the kernel function. Then the one for the output streams must be send back to the host after the kernel execution.

These data transfers affect negatively the performance of our program, especially in the case of a recursive application (i.e. a kernel function is executed for more than one time and the output data of each iteration is used as the input data of the next iteration). An example of these types of algorithms is the IIR (Infinite Impulse Response) filter that can be applied to image processing for emphasizing the edges of graphical objects. Using 16 coefficients, the filter is expressed by the equation as follows:

$$y_n = \sum_{i=0}^{15} a_i x_{n-1} - \sum_{j=0}^{15} b_j y_{n-j-1}$$

Here, because the y outputted once is used again in the right side of the equation, the algorithm after every kernel execution exchanges the input data stream of y and the output data stream of y. If we use the methods explained in the previous sections that simply copy data from the output to the input buffers in the host side (we call this *Copy Host*), the overhead caused by the I/O exchanges will become very large because they use the peripheral bus to copy the data between the host and the device. Therefore, we must find another method to exchange the I/O data.

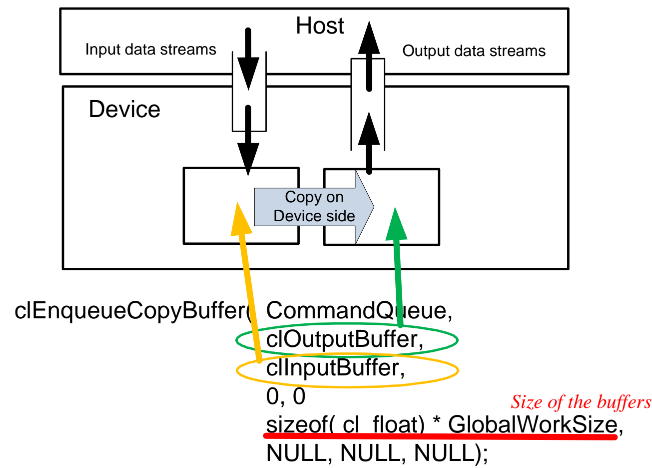


Figure 11: Copy Device method.

In this section we explain two methods that can be used in OpenCL to solve this issue proposing two techniques: *Copy Device* and *Swap*.

4.2.1 Copy Device

The point of this method is to use the internal bus of the GPU to copy data between two buffers on the device side. The available memory bandwidth of this bus in a GPU is much greater than the bandwidth on the peripheral bus. For example, the bandwidth of the bus of an NVIDIA GTX285 is 158GB/s, compared to the PCI Express x16 Gen2 bus that has a bandwidth of 8 GB/s. Therefore, the overhead caused by the copy operations is reduced significantly by using the GPU internal bus.

In OpenCL this method can be used just by invoking the API function `clEnqueueCopyBuffer` right after each execution of the kernel. This function copies the data from a buffer allocated on the device side to another buffer allocated on the device side. For example, if we have two buffers called `clInputBuffer` and `clOutputBuffer`, supposing that the data type they store is float, after each kernel execution we should call the function as shown in Figure 11. Because this function will copy the data from `clOutputBuffer` to `clInputBuffer`, the next iteration the kernel function will get the data from `clInputBuffer` that obtains the previous output data.

4.2.2 Swap Method

The Swap method uses the function `clSetKernelArg` to perform an I/O buffer exchange. As explained in section 3.4, the `clSetKernelArg` function links a device side buffer to an argument in the kernel function in order to set the value of that argument.

An example of the technique to be used in the swap method is shown in Figure 12. In this example we have two buffers allocated on the device side: `srcBuff` and `dstBuff`, that will be exchanged in every iteration. The swap operation is performed just before the kernel execution, using the `clSetKernelArg` function. First of all we must check the current iteration number (Figure 12, 1). If the current iteration number is even, the `srcBuff` is passed to the first parameter of the kernel function (Figure 12, (2.1)), which becomes the input data stream, and `dstBuff` to the second parameter, which does the output one. If the current iteration number is odd, we link the buffers vice versa (Figure 12, (2.2)). Then the kernel function is executed (Figure 12, (3)). Thus, the kernel function will get the input data from the buffer that contains the output data of the previous iteration. These operations are repeated until the application reaches the number of iterations desired (Figure 12, 4).

Finally, according to the fact that the number of iterations performed is even or odd, the final output data will be placed either in the `dstBuff` or in the `srcBuff`. Therefore, we must check the

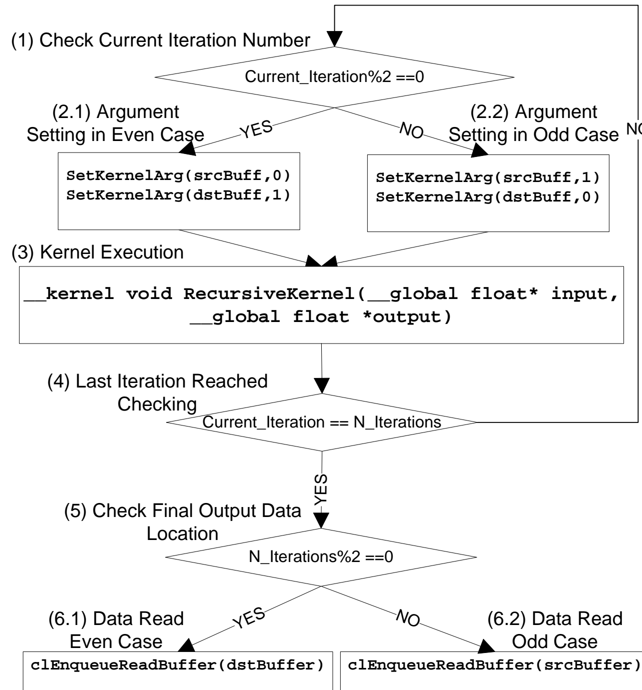


Figure 12: Swap method scheme.

Table 1: Experimental environment

CPU	Core i7-2620M (2.70GHz) with 8GB DDR3
GPU	Nvidia NVS 4200M (core:1480MHz / DDR3 VRAM:1GB)
OS	Windows 7 Home Premium SP1
OpenCL	Version 1.0

number of iterations performed (Figure 12, 5) and then need to decide the buffer to be read for the final result.

Because the swap method does not perform any kind of copy operation between buffers, the overhead caused by these data transfers is eliminated from recursive kernel execution. Therefore, this method should be used in recursive applications in order to achieve the best performance.

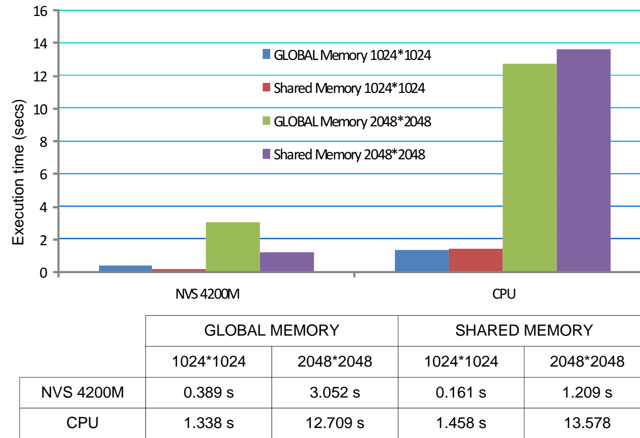
5 Performance Evaluation

This section evaluates performance differences of OpenCL programs between different platforms specifically of a CPU and of a GPU, and also evaluates the performance of the advanced techniques explained in the previous section.

We used two applications for the evaluations: a matrix multiply, to analyze the impact of the use of local memory, and an infinite impulse response (IIR) filter to compare the performances of the different data transfer methods between the host and the device. The environment for the evaluation is listed in Table 1.

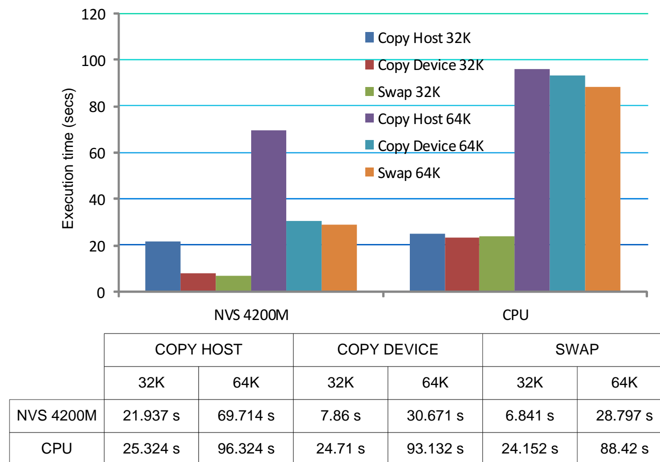
5.1 Matrix Multiplication

The matrix multiplication processes $A * B = C$ of $N \times N$ matrices. We measured the execution times on both GPU and CPU with varying the matrix size from 1024×1024 to 2048×2048. For



	GLOBAL MEMORY		SHARED MEMORY	
	1024*1024	2048*2048	1024*1024	2048*2048
NVS 4200M	0.389 s	3.052 s	0.161 s	1.209 s
CPU	1.338 s	12.709 s	1.458 s	13.578

Figure 13: Performance comparison using the matrix multiplication.



	COPY HOST		COPY DEVICE		SWAP	
	32K	64K	32K	64K	32K	64K
NVS 4200M	21.937 s	69.714 s	7.86 s	30.671 s	6.841 s	28.797 s
CPU	25.324 s	96.324 s	24.71 s	93.132 s	24.152 s	88.42 s

Figure 14: Performance comparison using IIR filter.

the GPU case we also measured the execution times using only global memory and using the local memory. The performance results are shown in Figure 13.

According to the results, we confirm that the use of the GPU local memory improves significantly the performance. However, on the CPU case, because all OpenCL memory objects are already cached by Hardware, we do not gain anything by explicitly caching again via the local memory. We can conclude that the GPU achieves about four times higher performance than the CPU.

5.2 IIR filter

Because the IIR filter is a typical recursive kernel, we use it to compare the performance of the different methods to exchange the I/O streams.

We measured the execution time of the kernel on both GPU and CPU, varying the number of samples from 32K to 64K. The three methods used to exchange the I/O buffers are the ones explained in the section 4.2(*Copy Device*, *Swap* and *Copy Host* method), which uses the OpenCL API functions `clEnqueueReadBuffer` and `clEnqueueWriteBuffer`. The results are shown in Figure 14.

In the GPU case, we confirm that the swap method achieves the best performance among all three methods. In the Copy Host method, the OpenCL API functions transfer the data using the peripheral bus between the host and the device. Therefore this method presents a large overhead

compared to the other two methods. Because the GPU is provided with a high speed bandwidth bus, the performance between the Copy Device and the Swap methods are very similar. The Swap method achieves the best performance because no copy operation is performed.

On the other hand, in the CPU case the difference of performances among the three methods is very small. Because we are using the CPU as the OpenCL device, the buffers on the host side and the buffers on the device side are allocated in the same memory region. Therefore the Copy Host and the Copy Device methods use the same bus to transfer the data between buffers. Therefore the performance difference between the three methods becomes small. All in all, the Swap method achieves the best performance because no copy operation is performed.

6 Summary

This tutorial paper presented an introduction to the OpenCL programming, explaining its characteristics and describing the structure of the OpenCL programs, using an example with a vector summation kernel. In addition, common techniques to improve the performance of the programs have been presented and evaluated in the aspects of memory I/O using the local memory and the recursive execution of kernel program applying the *Copy Device* and *Swap* methods. The evaluation also shows the performance characteristics among GPU and CPU platforms using the same program code.

OpenCL opens a new interesting investigation path in the field of stream computing. Due to its portability, OpenCL can be used in any environment that matches to the OpenCL platform model, also on flexible hardware devices such as FPGA (Field-Programmable Gate Array).

Moreover, OpenCL also allows multiple heterogeneous devices to work together in parallel on the same application. This characteristics could be very useful in a cluster scenario, where CPUs and GPUs are cooperating. Following this idea, we are working on an interface [17] for cluster programming, combining the NVIDIA SDK and the Intel SDK in order to use OpenCL among the CPUs and the GPUs of the cluster.

In summary, OpenCL is a very recently born environment that is expected to become the center of many research fields in the future. This tutorial paper has presented very basic characteristics of what OpenCL is capable to, but there are much more functionalities explained in books such as [9] [3] [15], and research opportunities to be discovered.

Acknowledgment

This work is partially supported by the Japan Science Technology Agency (JST) PRESTO program. And also this work is partially supported by KAKENHI (24300020) Grant-in-Aid for Scientific Research (B). Finally, I appreciate Mr. Pablo Lamilla Alvarez has helped to prepare the fundamental data and the analysis for the results described in this paper.

References

[1] ATI Stream SDK. http://developer.amd.com/sdks/amdappsdk/ downloads/pages/default.aspx.

[2] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. *ACM Trans. Graph.*, 23(3):777–786, 2004.

[3] Benedict R. Gasteri, Lee Howes, David R. Kaeli, and Perhaad Mistry. *Heterogeneous Computing with OpenCL*. ELSEVIER, 2011.

[4] J. Gummaraju and M. Rosenbluml. Stream programming on general-purpose processors. In *In 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 343–354, 2005.

[5] Intel OpenCL SDK. http://software.intel.com/en-us/articles/download-intel-opencl-sdk/.

[6] K Supercomputer. http://www.fujitsu.com/global/about/tech/k/.

[7] David B. Kirk and Wen mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufman, 2010.

[8] Pablo Lamilla, Shinichi Yamagiwa, Masahiro Arai, and Koichi Wada. Elimination Techniques of Redundant Data Transfers among GPUs and CPU on Recursive Stream-Based Applications. In *IPDPS/APDCM11 Anchorage USA*, May 2011.

[9] Aaftab Munshi, Benedict R. Gaster, Timothy G. Mattson, James Fung, and Dan Ginsburg. *OpenCL Programming Guide*. Addison Wesley, 2011.

[10] NVIDIA CUDA Toolkit 4.0. http://developer.nvidia.com/cuda-toolkit-40.

[11] NVIDIA CUDA Zone. http://www.nvidia.com/cuda.

[12] OpenCL. http://www.khronos.org/opencl/.

[13] John D. Owens, David Luebke, Naga Govindaraju, Mark Harris, Jens Kruger, Aaron E. Lefohn, and Timothy J. Purcell. A survey of general-purpose computation on graphics hardware. In *Eurographics 2005, State of the Art Reports*, pages 21–51, August 2005.

[14] Peter Pacheco. *An Introduction to Parallel Programming*. Morgan Kaufman, 2011.

[15] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Satoshi Miki, and Satoru Tagawa. *The OpenCL Programming Book*. 2010.

[16] Shinichi Yamagiwa and Leonel Sousa. Caravela: A novel stream-based distributed computing environment. *Computer*, 40(5):70–77, 2007.

[17] Shinichi Yamagiwa and Leonel Sousa. CaravelaMPI: Message passing interface for parallel gpu-based applications. In *In 8th International Symposium on Parallel and Distributed Computing (ISPDC 09)*, 2009.

[18] Shixun Zhang, Shinichi Yamagiwa, Masahiko Okumura, and Seiji Yunoki. Performance Acceleration of Kernel Polynomial Method Applying Graphics Processing Units. In *IPDPS/APDCM11 Anchorage USA*, May 2011.