INTRODUCTION TO OPENCL AND DEVICE FISSION

Arnaboldi Gabriele
Matr. 765568, (arnaboldi@libero.it)

Report for the master course of Embedded Systems
Reviser: PhD. Patrick Bellasi (bellasi@elet.polimi.it)

Received: January, 11 2014

Abstract

Over the past years, parallel computation has emerged as one of the most promising paradigms for achieving higher performances and, thanks to their architecture, modern graphics adapters revealed to be the best choice to implement such applications. As a token of this claim, various GPGPU APIs and programming framework has been publicly released in the last years, and OpenCL is one of the most interesting and versatile of them. The goal of this report is to first provide a general idea on what GPU computing is and then to analyze how the OpenCL model works and how a basic OpenCL application should be implemented. After the basic concepts have been sorted out, we'll move on a more advanced concept, that is Device Fission. Device Fission allows the programmer to create many different subdevices and to have total control over hardware resources, this can be very useful when resources are limited, for example when working on embedded environments. To better describe how Device Fission works, we'll present five different scenarios in which it can be applied. To conclude this report we'll present some interesting examples of real world OpenCL application and studies.

1 GPGPU Introduction

GPGPU (*General Purpose computation on Graphics Processor Unit*), also known as *GPU Computing*, is the utilization of the graphic adapter of a personal computer to perform generic and non-graphic related computation that is usually handled by the CPU.
Once designed and optimized specifically to perform only fixed graphic calculations, modern GPUs are evolving into *high-performances many-core* processors that can be virtually used to perform any task, and developers who port their applications to GPUs are able to achieve speedups of orders of magnitude vs. optimized CPU implementations of the same code.[1] The reason why GPUs are so fast has to be searched in the nature of what graphic rendering is about, that is an *intensive parallel computation*: GPUs are engineered to work specifically on *parallel data processing*, rather than data caching, flow control and pipelined execution like CPUs. (**Figure 1**)

Figure 1: GPUs are more focused to computation rather than data caching.

Moreover, computational units of GPUs are smaller and more numerous (typical GPUs can host hundreds of cores, while CPUs usually contains only 2 or 4) and are highly specialized to perform simple operation *in parallel* over a huge amount of data at the same time.
GPUs have been built to exploit application parallelism, and their particular architecture allow them to execute up to a couple TeraFlops of operations per second, against to the few GigaFlops that a single CPU can handle alone. (**Figure 2**)

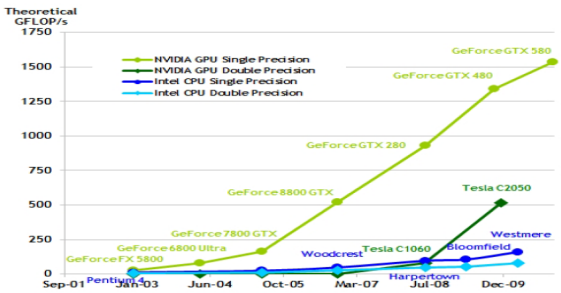

Figure 2: CPU vs GPU growth rate in terms of FLoating point Operations Per Second.

Due to its parallel nature, GPU computing can be very effective for applications involving huge amount of data, especially if the data can be represented in sequential structures like vectors and arrays.

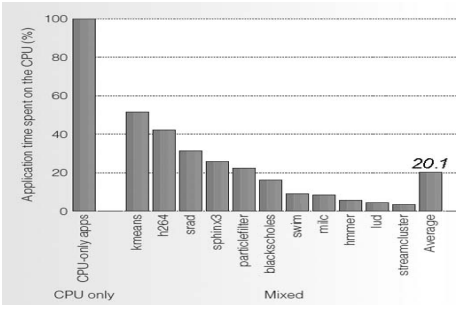

Figure 3: Time spent on the CPU on various benchmarks using mixed CPU-GPU computation. [2]

As you can see in **Figure 3**, GPU can offload the CPU from most of its work, but this doesn't mean that CPU performance is no longer critical. Many applications don't (and can't) map all their code to the GPU, and in certain cases the CPU can run some part of the code in a more effective way than the GPU. Furthermore, not all the code can be mapped easily and clearly on the GPU, as programming GPGPU application can be way more difficult and cryptic than programming in the classical way.

1.1 Basic Principles of GPGPU programming

In this section we'll introduce some of the basic principles behind GPGPU programming. Some of these concepts are directly related to the "graphic-wise" manner in which the GPU executes its operations, and having a little knowledge of some of the basic concepts of computer graphics can help to understand how GPU computing work and why it is so fast. For instance, GPUs can handle bidimensional matrices natively because that's the natural way to represent textures in memory. On the other hand CPUs designed to to work on single dimension vectors, because that's how the standard RAM is structured.
A GPU is basically a *stream processor*[1]: a single **kernel** is executed over a stream of data in a monolitic fashion, at a given moment, the same instruction is executed by every computing unit, each one working on a different piece of data.

1.1.1 Textures = Arrays

Due to the linear structure of memory, traditional CPUs cannot *physically* create multi-dimensional array, and accessing single rows and columns of a matrix is achieved by offsetting memory addresses of a large linear array. Each one of these "jump" in memory translates directly in performance loss.
On the other hand, GPUs are architectured to work natively with textures, that are naturally represented in memory by two-dimensional arrays.
The ability to work directly over bidimensional (and three-dimensional) memory structures is one of the main reason why GPUs are much more faster than CPUs when it comes to elaborate data.
On the other hand, CPUs can handle memory structures much more bigger (and virtually infinite) compared to the those available for the GPU, as many graphic adapters can only work on textures limited in size. The usual maximum size of a texture is usually 2048*2048, or 4096*4096, but modern graphic cards can handle textures of 8192*8192 in size. The maximum texture size available for a certain graphic adapter can usually be easily retrieved by simple queries made available by the programming API you are using.

> **OpenCL Reference 1**
> For Example, with OpenCL, you can obtain the maximum supported texture size with the *clGetDeviceInfo()* function, passing the *CL_DEVICE_IMAGE2D_MAX_WIDTH* and *CL_DEVICE_IMAGE2D_MAX_HEIGHT* as parameters.

We will discuss how textures (and *memory objects* in general) are created and accessed in Section 2.1, when we will introduce the OpenCL API.

1.1.2 Kernels

If you are familiar with graphic pipeline programming, kernels are the GPGPU equivalent of **shaders**.
Kernel programming is the core concept behind GPU computation and forces the developer to think in a different way as he is used to, as kernels are oriented toward *parallel execution over stream of data*, while standard CPU programming is oriented toward a classical *loop-iterated* implementation.
Since the data of our application is stored into multi-dimensional memory objects (the equivalent of a graphical texture), GPU computing basically consists in feeding this memory structures to a kernel that will execute its code over different data elements simultaneously, in the same way that a graphic shader apply the same transformation over multiple pixels to obtain the final image.
Once the kernel has finished its computation, the output will be a new memory object that contains the result of the calculation.
If we keep in mind that GPUs were born to elaborate graphical data, it will be easier to understand how kernels work and what is the best approach to be taken when it comes to write a GPGPU application. To understand how this mechanism works, here's an example of a simple graphical shader written in HLSL language that basically scans a texture to find black pixels and turn them to white:

[1] Stream Processing refers to a SIMD (Single Instruction Multiple Data) paradigm that allows application to exploit (limited) parallel processing over data.

```
float3 BlackToWhite(PixelShaderInput input)
{
  if(input.Color.r == 0 &&
     input.Color.g == 0 &&
     input.Color.b == 0)
     return float3(1,1,1);
  else
     return input.Color;
}
```

The same code in a traditional loop-oriented implementation will be something like:

```
void BlackToWhite(float input[4096][4096][3],
                  float output* [4096][4096][3])
{
  for (int y=0,y<4096,y++)
    for(int x=0;x<4096;x++)
    {
      if(input[x][y][0] == 0 &&
         input[x][y][1] == 0 &&
         input[x][y][2] == 0)
      {
         *output[x][y][0] = 1;
         *output[x][y][1] = 1;
         *output[x][y][2] = 1;
      }
      else
      {
         *output[x][y][0] = input[x][y][0];
         *output[x][y][1] = input[x][y][1];
         *output[x][y][2] = input[x][y][2];
      }
    }
}
```

By comparing the two examples, we can note two fundamental things:

1. In the shader we do not implement any cycle, the code is iterated *automatically* on every element of the input structure. Also there is no mapping between the input and the output, but only a **single return**, because the output element is automatically mapped to the same texture coordinate of the input.

2. Since GPU are meant to work with colors, shaders can natively work on 4 different channels at a time (RGBA), making GPU computation even more versatile and powerful.

While shaders have to be written in low-level specific languages like HLSL or GLSL, kernels take advantage of APIs that allow the programmers to implement and execute them like normal functions. We'll show some examples on how to implement kernels in Section 2.1.4.

OpenCL Reference 2
For Example, using the OpenCL API, you can easily create objects of type **cl_kernel** and initialize them with the **clCreateKernel()** function. On page 5 you can see an example of a kernel function implemented using OpenCL.

1.1.3 Computation and feedback

Since GPU's final purpose is to draw something on screen, GPGPU application cannot be simply "executed" like traditional ones, and kernels (although they are basically functions) cannot be simply "called". To execute a kernel application over the GPU we have to make it think that it is actually drawing something. In GPU computing, "to execute something" translates to "*to draw something*". The operations needed for a kernel call (and their shader execution equivalent) are summarized in **Table 1**.
After the computation has been performed, the result is stored in the target surface.

	Drawing perspective
1)	Assign the input texture to a texture channel of the graphic adapter
2)	Set the drawing surface
3)	Define the area to be drawn.
4)	Load the shader
5)	Render the image
	Computation perspective
1)	Initialize the input data structure and feed it to the kernel
2)	Define in which memory object the output data will be stored
3)	Initialize the indices and set the bounds of the loop
4)	
5)	Iterate through data and execute the kernel.

Table 1: Drawing and Kernel Execution comparison

1.2 GPGPU APIs and Languages

In this section we will briefly introduce the most common languages and APIs used to develop GPGPU applications.

1.2.1 CUDA (www.nvidia.com)

CUDA is the parallel programming platform introduced by Nvidia in 2006 and it has currently reached version 5. The main focus of CUDA is on parallelism and automatic scalability: the program model forces the programmer to partition the main problem into coarse sub-problems that can be solved independently and in parallel. The various threads are automatically scaled in runtime over the different cores of the GPU, and the developer doesn't have to know in advance the architecture of the graphic adapter. (**Figure 4**).

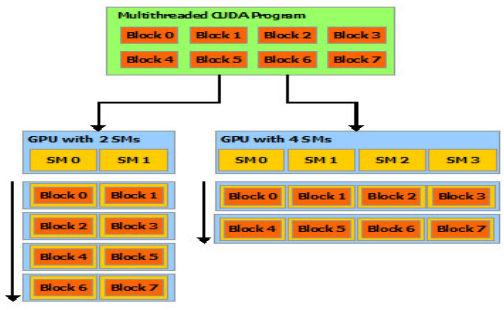

Figure 4: CUDA executables are automatically scaled over the various SMs (Stream Multiprocessors): a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

CUDA is mainly based on C-language and follow the basic principles introduced in Section 1.1, as it uses *Kernels* and texture-like memory structures. It also introduces new concepts like **thread hierarchy** (allowing to create *multi-dimensional thread blocks* that can be executed in parallel) and **memory hierarchy** (for example each thread has its own local memory and can share memory with thread in their own block).
On the downside, since it is a proprietary framework, CUDA executables will only run on Nvidia graphic cards, and the code cannot fall back on the CPU in the case a CUDA accelerated hardware is not available on the system.[7]

1.2.2 OpenCL (www.khronos.org/opencl/)

OpenCL (Open Computing Language) is the parallel programming model developed by the Khronos group that focuses on cross-platforming: differently from CUDA, OpenCL is supported on a wide range of devices and can also be executed on embedded systems and mobile devices. OpenCL applications can also be executed on standard CPUs and don't strictly need to have a graphic adapter installed on the system, this guarantees compatibility with many different devices, especially in the field of embedded systems or non conventional computing. The current version of OpenCL is 1.2 and we will discuss this parallel programming framework more in deep in Section 2.

1.2.3 DirectCompute

Also known as Computer Shader (for more info refer to the msdn library: http://msdn.microsoft.com/), DirectCompute is the GPU computing API developed by Microsoft and it is part of the DirectX APIs collection starting from version 11. Since it is part of the DirectX package, DirectCompute uses HLSL shading language and integrates well with applications already written with the DirectX API, however, its compatibility is limited to desktop graphic adapters that supports DX10 or 11 and only for Windows (Vista or later) operating systems.

2 OpenCL and Device Fission

OpenCL consists of an API for coordinating *parallel computation across heterogeneous processors* (CPU, GPU and potentialy any other processor) and it is supported by a wide range of systems and platforms, making it the perfect choice for parallel computation not only on traditional desktop CPU-GPU configuration, but also on embedded systems. OpenCL was initially developed by Apple, but is now maintained by the Khronos Group (http://www.khronos.org) that, in 2011, released the 1.2 version of the API, introducing new features such as enanched image support, built-in kernels and **device partitioning** (also known as Device Fission).
Since the strength of OpenCL is its heterogeneity, to compile and to execute and OpenCL application, one must install on the system the libraries specific to the target platform, and usually each hardware manufacturer provide its own SDK to develop and run OpenCL applications.

2.1 OpenCL Architecture

To define the structure of an OpenCL application, first we will introduce its core components. Some of these components (Host and Kernels) are "software components" that, together, form the core of an OpenCL executable, while other components (Compute Devices and Units) refer to the underlying hardware that has to run our application.
The architecture of an OpenCL application is summarized in **Figure 5**, and the main components that needs further investigation are the **Host** (executed on the CPU), the **Kernels** (executed on the processig elements of the OpenCL device) and the **device**. We'll now introduce the concepts behind the host and the device; kernels and their implementation will be discussed later in Section 2.1.4

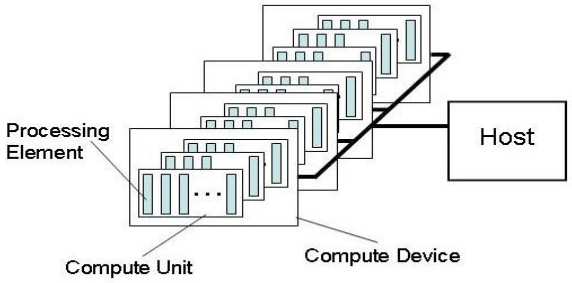

Figure 5: OpenCL Architecture

2.1.1 The Host

The Host can be viewed as the **controller** of the application. It usually doesn't perform any task specific to the domain of the application and its main function is only to setup and configure the environment, to issue commands to the compute units and to coordinate the kernels. Since the host is basically a "standard" executable that executes "normal" code, *it is always executed on the CPU and never on the GPU*.
Here's list of the main tasks that every host should do:

1. Declare the necessary structures (Kernels, devices, memory objects...)

2. Query for hardware capability (Number of devices available, number of processing units, buffers size...)

3. Obtain and lock hardware resources

4. Create and initialize memory objects (see Section 2.1.6)

5. Create kernels

6. Issue orders to kernels (Launch, synchronization, memory management)

7. Retrieve output data from kernels

8. Manage errors

9. Release of hardware resources, deallocate memory, and final cleanup.

OpenCL Reference 3
In OpenCL the host is a standard C/C++ application implemented inside the classical 'main' function. Every host must always implement and initialize these five data structures in order to properly setup an OpenCL environment:
cl_device_id to obtain an handle to the hardware device,
cl_kernel to have a reference to each kernel implemented,
cl_program to define the main program that contains all the kernels,
a **cl_command_queue** for each kernel, used to issue commands to them ,
a **cl_context** that wraps all programs, kernels, devices and memory object of an OpenCL application.
In Appendix **??** you can see an example of OpenCL host implementation.

2.1.2 The Device(s)

One or more OpenCL Devices can be connected to the host. These devices can be physical (e.g. the graphic adapter installed on the system or the CPU if no GPU is available) or even virtual (e.g. remote GPUs in a cluster configuration or sub-devices using device partitioning) Each device hosts several **Compute Units** that are basically the cores of a CPU or the Stream Multiprocessors of a GPU and each Compute Unit is further divided into different **Processing Elements** that can work in both **SIMD** mode (Single Instruction Multiple Data), and **SPMD** mode (Single Program Multiple Data). Each Processing Element has it own program counter and run independetly from the others.

OpenCL Reference 4
Devices are stored into a **cl_device_id** structure that is basically an array that can be filled with all the available devices on a system by calling the **clGetDeviceIDs()** function, specifying the type of device required (CL_DEVICE_TYPE_CPU or CL_DEVICE_TYPE_GPU). Once we have filled the cl_device_id structure, it can be passed to the **clGetDeviceInfo()** function with **CL_DEVICE_MAX_COMPUTE_UNITS** as parameter to query the maximum number of Compute Units available.

2.1.3 Application Execution, Work Items and Index Space

From a very coarse point of view, we can describe the execution of an OpenCL application as a three-step process:

1. when the application is launched, the **host** is executed on the CPU to define the **program context** (Section 2.1.5) for the kernels and to dispatch them for execution.

2. the **kernels** are launched by the host and executed on the OpenCL device(s). They compute their code in parallel over the input stream of data.

3. the output data is returned to the host.

In OpenCL, an *instance* of a kernel is called **work-item** and it executes over an **index-space** that contains the data. Multiple instances of a kernel can run simultaneously on the Coumpute Units of a Device (See **Figure 5**).
Index spaces are basically the memory objects over which the work-items compute their istructions. In OpenCL index-spaces are also called **NDRange** (N-dimensional index space, where N can be a value of one, two or three, since GPUs can work on 1,2 or 3-dimensional textures).
In **Figure 6** you can see how work-items and work-groups are organized, and in the next section there's a little example of how a kernel can obtain its own global ID over the NDRange.

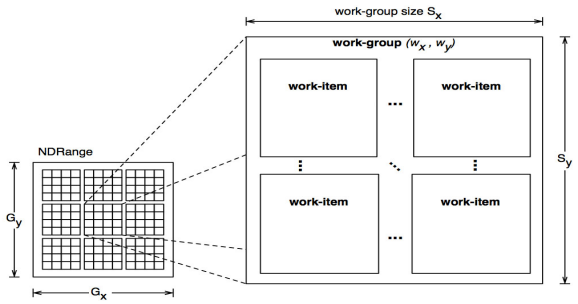

Figure 6: Work-items mapped over a two-dimensional NDRange. As you can see, work-items can be organized in Work-Groups and every work-item has both a global and a local ID inside its work-group.

OpenCL Reference 5
For example, to execute a kernel the host must call the **clEnqueueNDRangeKernel()** function. As you can see from the function name, kernels are not simply 'executed' but they must be added to a specific command-queue that will later be launched.

2.1.4 Kernel Implementation

Below is an example of a simple kernel that calculates the square of the input NDRange:

```
__kernel void myKernel(__global int* input,
                       __global int* output)
{
    int workID = get_global_id(0);
    output[workID] = input[workID] * input[workID];
}
```

As you can see, a kernel is nothing different from a normal C/C++ function. The interesting thing of a kernel, is that it executes over *each element* of the input memory object, even if no iteration is specified. This is due to the fact that when a kernel is dispatched for execution, it automatically creates multiple "copies" of the function (the work-item), each one with its own **workID** that maps directly to the NDRange and that will execute on a different Processing Element. Each work-item has both a **global environment** shared with all other work-items and its own **local environment** that can be accessed using the specific function that the OpenCL API offers:

get_work_dim	Number of dimensions in use
get_global_size	Number of global work items
get_global_id	Global work item ID value
get_local_size	Number of local work items
get_local_id	Local work item ID
get_num_groups	Number of work groups
get_group_id	Work group ID

Table 2: Work-Item Built-In Functions

2.1.5 Program Context and Command-Queue

As we alreaty mentioned, one of the host functions is to define the **context** for the application. Contexts are used by the OpenCL runtime for managing objects such as command-queues, memory, program and kernel objects and for executing kernels on one or more devices specified in the context.
A context basically consists in:

1. a collection of **OpenCL devices** to be used
2. a collection of the functions that will be executed on the devices (the kernels)
3. **program objects**, that are simply the source files and executables that implements the kernels
4. **memory objects**: the NDRange objects to be elaborated

After the context has been created, the host initialize a structure called **command queue** that is used to schedule and dispatch commands to the devices within the context. The command structure is very simple and the commands that the host may issue fall only into three categories:

1. **Kernel execution commands**: Execute a kernel on a Processing Elements of a specific device.
2. **Memory commands**: Transfer data to, from, or between memory objects, or map and unmap memory objects from the host address space.
3. **Synchronization commands**: Used to specify the order of execution of commands and to synchronize the kernels.

OpenCL Reference 6
Context and Command Queues are created by simply calling the **clCreateContext()** and **clCreateCommandQueue()** functions in the host.

2.1.6 The Memory

There are four types of memory regions in OpenCL: **Global Memory**, **Constant Memory**, **Local Memory** and **Private Memory**.

- Global memory grants read/write privileges to **all** the work-items in every work-group. Basically, every work-item (*that are part of the same context*) can access it.
- Constant memory is only used to store constants. Only the host has write privileges over it, while kernels can only read from it.
- Local memory is shared among all the work-items that form a group. The host has no access to this memory area.
- Private memory is allocated directly by the work-item and can be used only by itself. The host has no access to this part of memory.

Since computation is carried on parallely, one of the major issues about memory is **consistency**. OpenCL uses a relaxed consistency memory model: the state of memory visible to a work-item is not guaranteed to be consistent across the collection of work-items at all times. **Table 3** summarizes memory consistency for the various regions of memory available.

Memory	**Consistency**
Private	memory is not shared, read/write consistency is always guaranteed
Local	consistency is guaranteed between work-items of the same work-group
Global	consistency is guaranteed between work-items of the same work-group, but not between multiple work-groups in the case they are assigned to execute the same kernel

Table 3: Memory consistency

In OpenCL computation is performed over **memory objects**. There are two distinct memory objects: **buffers** and **images**.

- Buffers are used to store a one-dimensional collection of elements (like an array), and those elements can be scalar values (int, float, etc.), vectors or user defined structures. Buffers are stored sequentially and *can be accessed using pointers*; elements of a buffer are stored in memory in the *same format* as they are used by kernels (i.e. if the kernel works on single integers, these elements are stored in memory as integers, this is not true for image objects)

- Images are used to store bi- or three-dimensional structures and their elements can only be selected from a list of predefined image formats (i.e. you cannot simply declare int or floats in an image object). Differently from buffers, elements of images *cannot be accessed directly with a pointer* and they are *always stored in memory as 4-dimensinal vectors* (since graphic shaders work on RGB and Alpha components of the pixels of an image)

OpenCL Reference 7
In OpenCL memory objects are stored into **cl_mem** structs, that can be easily initializated with the **clCreateBuffer()** and **clCreateImage()** functions.
Image format availables may vary from one graphic adapter to another, a list of supported image formats can be obtained using the **clGetSupportedImageFormats()** query.

2.2 Device Fission Introduction

Device Fission is an extension of the OpenCL specification (fully defined in OpenCL 1.2, although it was already available in OpenCL 1.1 as well as an optional extension) that adds a whole new level of control over parallel computation and hardware management.
As the term 'fission' implies (dividing or splitting something into two or more parts), device fission allows the subdividing of a device into one or more virtual sub-devices. This practice, when used carefully, can provide a huge performance boost, especially when executing parallel code on the CPU instead of the GPU [9] (At present day device fission is suported only on CPUs and not on GPUs [6]) Implementing applications that use device fission *does* require some knowledge of the underlying target hardware and, if not used properly, it can lead to worst performances and it may impact code portability.

2.2.1 Device Fission as a better way to manage Embedded Systems

Device fissioning can be very useful in embedded environments and, in general, in any situation where the system resources are limited:

- Device fission allows to use only a *portion* of a device, this means that the OpenCL runtime will not take the entire device for itself and other non-OpenCL application can work on it at the same time. It can also help to reduce power consumption since high-task parallelism at low core frequencies can give better power perfomaces [11]

- Device fission allows to implement specialized memory sharing models that can be useful when it comes to manage the limited memory of an embedded system. (For example see the partitioning by affinity domain described in Section 2.3.2)

- Since embedded systems may have limited graphic capabilities (or no graphic adapter at all), Device Fission may help to better exploit parallel computation where the only option is to use a CPU and not a GPU

2.2.2 Sub Devices

Each subdevice can have its own **program context** and **command-queue** (See Section 2.1.5), this means that each subdevice can have its own private area of memory and that kernels can be dispatched independently to one subdevice or another.

OpenCL Reference 8
In OpenCL you can create new sub-devices using the **clCreateSubDevices()** function. The first parameter to pass is of type **cl_device_id** and it is the ID of the 'parent' device to be partitioned.
You can query for a list of available devices on the system by using the **clGetDeviceIDs()** function. Each sub-device can have a maximum number of compute units specified by the **CL_DEVICE_PARTITION_MAX_COMPUTE_UNITS** property. Since sub-devices are treated in the same exact way as 'normal' devices, different contexts and command-queue can be created by simply passing the ID of the subdevice to the **clCreateContext()** and **clCreateCommandQueue()** functions .

There are three main way to partition a device: **equally**, **by counts** and **by affinity domain**:

- partitioning a device **equally** means that the application will try to split the device into as many sub-devices as possible, each containting a number of Compute Units specified by the programmer. If that number does not divide evenly into the maximum available compute units, the remaining are not used.
- **by counts** means that the device is not divided automatically (and equally), but accordinlgy to a list provided by the programmer (e.g. given the list (4, 8, 16) the device will be divided into three sub-devices containing respectively 4, 8 and 16 CUs)
- partitioning by **affinity domain** is an automatic process that will create sub-devices composed of CUs that share similar levels of cache-hierarchy (specified by the programmer, e.g. L1, L2, L3 caches or NUMA nodes). This can be very useful when micro-management of memory is useful or for system where memory is a critical factor (for example in embedded systems)

A list of some examples on how these partitioning parameters can be used to create different configurations on the same hardware can be found in the **Appendix A**.
Another interesting feature that allow even more flexibility and control over the hardware is that device fission allows sub-devices to be further partitioned and therefore create a tree-like structure like the one shown in **Figure 7**:

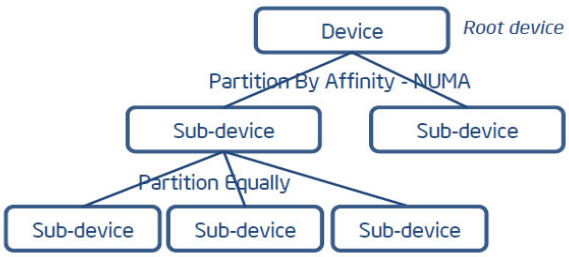

Figure 7: Sub-devices can be further partitioned to create an hierarchy of devices. Each node of the tree can be partitioned using different partition parameters.

2.3 Device Fission Strategies

Device fission can be used in several way to increase performance of OpenCL applications and to manage the (limited) resources of a system in a more efficient way. There are several standard approaches in which device fission can make the difference if used correctly and in the next sections we'll present some examples.

2.3.1 High-Priority dedicated sub-device

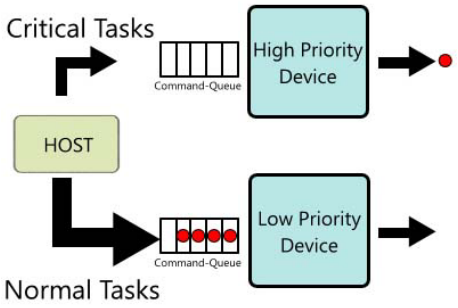

Figure 8: Scenario 1: a sub device is created to compute high-priority tasks

In this scenario, a sub-device is created to deal with critical tasks, while the remaining Compute Units of the device will be used for standard computation. Low priority tasks will be dispatched to the low priority queue, in this way the high priority device will will always have its command queue free and will be able to execute its tasks as soon as they are issued. This configuration can be simply achieved by partitioning the device *by counts*, reserving a few cores (1 or 2) for high-priority computation and leaving the rest for normal computation.
The following code demostrates how to partition the device properly, the full code for this scenario can be found in **Appendix ??**.

```
// Get Device ID from platform [0]
// a list of available platforms can be obtained
// using clGetPlatformIDs() function

clGetDeviceIDs( platforms[0], CL_DEVICE_TYPE_CPU,
                1, &device_id, NULL);

// Create two sub-devices, the parameters for the
// partitioning are storend into a
// cl_device_partition_property array

cl_device_partition_property param[5];
param[0] = CL_DEVICE_PARTITION_BY_COUNTS;
param[1] = 2;   // 1st device: 2 compute units
param[2] = 4;  // 2nd device: 4 compute units
param[3] = CL_DEVICE_PARTITION_BY_COUNTS_LIST_END;
```

```
param[4] = 0; //end of list

//now we can create the sub-devices
cl_device_id output_IDs[2];
clCreateSubDevices(device_id, param, 2,
                   output_IDs, NULL);

//our 'High Priority' device:
cl_device_id *hpDevice = &output_IDs[0];
//our 'normal' device:
cl_device_id *normalDevice = &output_IDs[1];
```

Test Results

We partitioned a CPU device with 4 cores in total and created two sub-devices, one with 3 Compute Units (the low-priority one) and one with just one Compute Unit (the high-priority one). Then we tried to dispatch different sets of kernels marked 'normal' and 'critical'. First we dispatched all of them only to the low-priority device, and the result was that the tasks were completed in the same exact order they were dispatched, so the critical tasks had to wait for other tasks to finish. Then we routed the high priority tasks to the dedicated subdevice, with the result that those tasks were completed as soon as they were dispatched.
In another test we decided to block the low-priority subdevice by letting it execute an infinite while loop, the result was that the normal tasks were not able to continue their execution, but the the critical ones were still executed normally as soon as they were dispatched since they were routed to a different device.

2.3.2 Memory Proximity

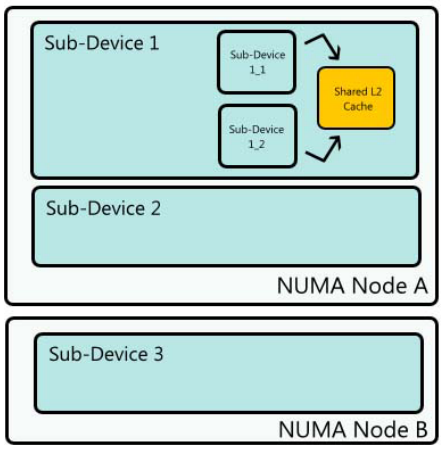

Figure 9: The main device is partitioned into 2 subdevices that have Compute Units that share the same NUMA node. Subdevice 1 is further partitioned to obtain two devices that shares the same L2 Cache.

If the work-items of the application share high amounts of data, it can be useful to create sub-devices that share the same cache or are physically placed on the same NUMA node to increase performance. Without device fission, there is no guarantee that the work items will have these characteristics. This kind of partitioning can be achieved using the *partition by affinity model*, specifying which level of memory needs to be shared.
Obviously, this kind of partition is highly hardware-dependent, and it is not guaranteed that a particular partitioning scheme will work on different platforms. To maintain portability the host should check which partition models are available on that specific device and decide at runtime how to subdivide it.

> **OpenCL Reference 9**
> The memory domains available on a particular device can be obtained by a simple query with the **clGetDeviceInfo()** passing **CL_DEVICE_PARTITION_PROPERTIES** and **CL_DEVICE_PARTITION_AFFINITY_DOMAIN** as parameters.

The following example creates a partitioning model like the one shown in **Figure 9**

```
clGetDeviceIDs( platforms[0], CL_DEVICE_TYPE_CPU,
                1, &device_id, NULL);

// First partitioning
cl_device_partition_property params[3];
params[0] = CL_DEVICE_PARTITION_BT_AFFINITY_DOMAIN;
params[1] = CL_DEVICE_AFFINITY_DOMAIN_NUMA;
params[2] = 0; // End of list

// Create the sub-devices:
cl_device_id subdevices_IDs[2];
clCreateSubDevices(device_id, params, 2,
                   subdevices_IDs, NULL);

// We devide the first subdevice into 2 further
// subdevices
cl_device_partition_property params[3];
params[0] = CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN;
params[1] = CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE;
params[2] = 0; // End of list

cl_device_id subsubdevices_IDs[2];
clCreateSubDevices(subdevices_IDs[0], params, 2,
                   subsubdevices_IDs, NULL);
```

The case study in Section 3.2.3 contains an example on how partition by affinity is used to exploit temporal and spatial cache locality.

2.3.3 Warm Cores Exploitation

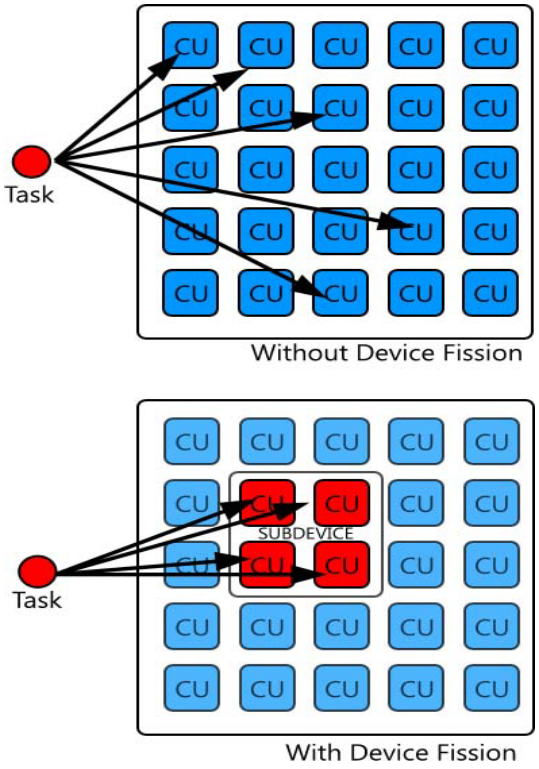

Figure 10: Without device fission there is no real control on which cores will be used, and tasks may be dispatched to 'cold' cores. With device fission we can redirect tasks to a specific (small) subdevice and exploit 'warm' cores.

Without device fission, when a new command is submitted to the command-queue it may happen that it will be dispatched for execution to a 'cold' core. 'Cold' cores are those that have cache memory filled with data that is not relevant to the current OpenCL program or task. With device fission we can force to use already 'warmed up' cores to minimize latency time. This scenario adapts well for short-running programs, where the overhead of 'warming' the core is relevant. For long-running programs, using device fission in this way may have very little effect or even degrade performance. This approach can also be used for thermal management purposes, as it can have effect on which area of the CPU will heat more.
The implementation of this kind of behavior is similar to the one described in Section 2.3.1: we can divide the device by *counts*, reserving a 'warm' area over which we will dispatch fast and short tasks.

2.3.4 Flowgraph and Pipeline Computation

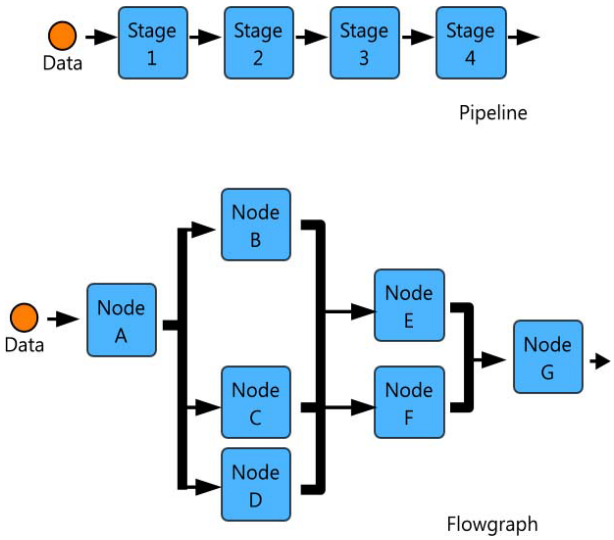

Figure 11: Device partitioning allows to create 'virtual' pipelines

Since device fission gives total control over task dispatching, we can use it to create complex structures like the one shown in **Figure 11**. This technique allows to create high-specialized virtual devices (like virtual hardware) for carrying out specific tasks and a good synchronization from the host must be provided.

2.3.5 Maximize Throughput

This scenario applies when data sharing is very limited or completely absent, and the only important thing is to maintain high levels of throughput. To achieve this, a job must have *all the resources* available for itself (i.e. all the on-chip caches), and no other jobs may interfere, therefore we must create *one and only one* device for each NUMA node, so devices will never try to access the caches of other devices.
To create this kind of partitioning Partition By Affinity must be used to fission the device into N sub-devices, where N is the number of NUMA node available.

3 Real World Examples and Case Studies of OpencCL and Device Fission Applications

3.1 OpenCL for FPGA implementation

The great flexibility of OpenCL allows it to run on a great variety of different hardware and platforms. One interesting application, expecially in the field of embedded systems, is to use it as substitute of traditional HDLs (Hardware Description Languages) like VHDL or Verilog when FPGA programming is required.

3.1.1 OpenCL-FPGA Implementation

As we already saw, OpenCL applications consists of two part: the host and the kernels. When migrating to FPGA, the developer has to opt for two solutions:

1. An "All in One solution", by implementing the CPU that will run the standard C/C++ host code directly on the FPGA. (For example by using a soft macro provided by the microprocessor manufacturer)

2. A separated solution, by using an external microprocessor for the host and by programming the FPGA to execute the kernels only.

Unlike CPUs and GPUs, where parallel threads are executed on the different (and generic) cores available, FPGAs offer a different strategy. Kernel functions can be transformed into dedicated, specialized, and deeply pipelined **hardware circuits** that are inherently multi-threaded thanks to the concept of pipeline parallelism. Each pipeline can then be replicated over the entire FPGA to provide even more parallelism. This two levels of parallelism translates into an immediate boost in performance.
Figure 12 shows a simple example of how kernels are translated into separate, multiple **hardware pipelines**.

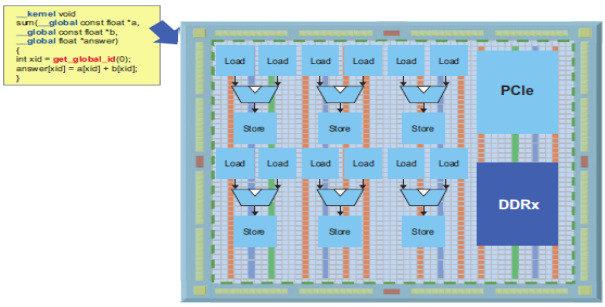

Figure 12: Kernels are translated into multiple hardware pipelines, inherently providing two levels of parallelism: pipeline parallelism and thread level parallelism

The most important concept behind the OpenCL-to-FPGA approach is the notion of **pipeline parallelism**. Basically, an OpenCL-to-FPGA compiler is able to implement the scenario observed in Section 2.3.4 in an automatic and more efficient way. We will describe how pipeline parallelism work by introducing an example, shown in **Figure 13**:

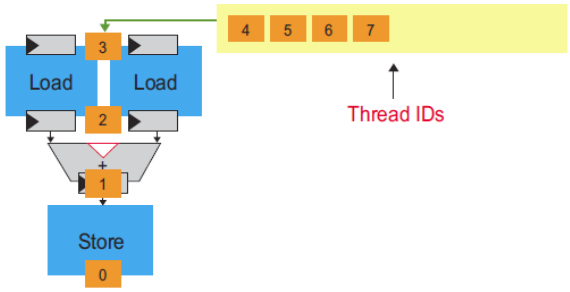

Figure 13: Pipeline parallelism example

On the first clock cycle, thread 0 is clocked into the two load units. This indicates that they should begin fetching the first elements of data from arrays A and B. On the second clock cycle, thread 1 is clocked in at the same time that thread 0 has completed its read from memory and stored the results in the registers following the load units. On cycle 3, thread 2 is clocked in, thread 1 captures its returned data, and thread 0 stores the sum of the two values that it loaded. It is evident that in the steady state, all parts of the pipeline are active, with each stage processing a different thread.
In **Figure 14** you can see a general scheme that summarizes how OpenCL-FPGA applications should be implemented. Since memory is shared between all the components, one crucial point in parallel computation is the memory management, and as we can see from the figure several memory interfaces are needed, but generally OpenCL-FPGA compilers are able to implement such interfaces automatically, exempting the developer from such tedious and delicate task.

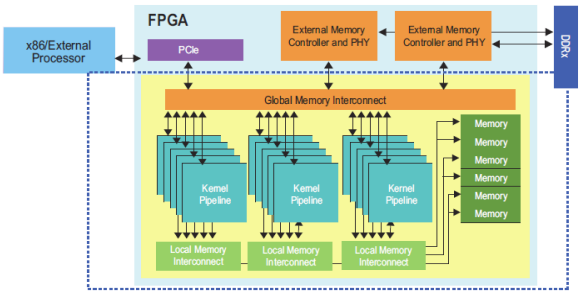

Figure 14: OpenCL-FPGA Implementation scheme

3.1.2 Benefits

Using OpenCL as a substitute of traditional HDLs can provide various benefits:

- **Improved Time To Market:** OpenCL offers a quicker and simpler way to implement parallel al-

ghorithms compared to traditional FPGA development using lower level hardware description language (HDLs) such as Verilog or VHDL [12]. This because OpenCL inherently offers the ability to describe parallel computation, while the main challenge in HDLs languages was exactly to extract thread-level parallelism from a sequential program. OpenCL offers instead the ability to the programmer to specify and control parallelism in an explicit and clear way.

- **Better Performance:** Dedicated ad-hoc hardware structures allow faster computation than using generic CPUs. Furthermore OpenCL-FPGA compilers automatically exploit pipeline parallelism to make computation even faster.

- **Less Power Consumption:** Benchmarks show that FPGA applications consumes lot less power to execute the same OpenCL code in comparison to CPU or GPU solutions.

3.1.3 Case Study: Monte Carlo Black-Scholes Method

In this experiment an economic model was used to benchmark an OpenCL-FPGA unit. The model is based on the Monte Carlo Black-Scholes method and it is used to compute the expected payoff of stock prices over millions of different paths. The entire algorithm used for this benchmark can be implemented in approximately 300 lines of OpenCL code that is portable from FPGA to CPU and GPU. The comparison in performance between these 3 different platforms are shown in **Table 4**.

Platform	**Power** [Watts]	**Performance** [Billions of simulations per seconds]	**Efficiency** [Millions of simulations per second per watt]
CPU	130	0.032	0.0025
GPU	212	10.1	48
FPGA	45	12.0	266

Table 4: Monte Carlo Black-Scholes benchmark results

As we can see, not only the OpenCL framework targeting a FPGA board exceeds the throughput of both a CPU and a GPU, but it also consumes one-fifth the power of comparable GPUs when executing the same code.

3.1.4 Case Study: Document Filtering

In this benchmark the focus is set on a more practical problem than the previous, as it will consider an algorithm used in modern data centers. A recent report [14] from International Data Corporation (IDC) examined the requirements of high performance computing data centers, and conducted a survey of the top constraints in expanding current data center capabilities. From the results, it was obvious that power and cooling costs are the key impediments of compute capability expansion, and as we have seen in the previous benchmark, FPGA computation offer an huge improvement in power consumption.
In this experiment, a document filtering algorithm was implemented using OpenCL to program an FPGA. The algorithm basically consists in analyzing an incoming stream of documents and find the ones that best match a user's interest. The results are shown in **Table 5**:

Platform	**Power** [Watts]	**Performance** [Million of Terms per seconds]	**Efficiency** [Millions of Terms per Joule]
CPU	130**	2070	15.9
GPU	215	3240	15.1
FPGA	21	1755	83.6

Table 5: Document Filtering benchmark results
**Does not include memory consumption.

As you can see, although CPUs and GPUs can perform better in terms on throughput, the power efficiency of these two platform can be five time lower than the efficiency of FPGAs. It is interesting to note that in this case the performance of the FPGA was limited by the external memory bandwidth, and not by the FPGA itself. With a proper setup, the authors of this test extimate an increase in performance up to 2925 MT/s maintaining the same power consumption level, that would raise the power efficiency value from 83.6 MT/J to 139,3 MT/J.
These results demonstrate that introducing an FPGA OpenCL implementation of algorithms, could bring a dramatic decrease of power consumption and thus cooling costs in modern data centers.

3.2 Investigating performance portability of OpenCL

In the next example [10] OpenCL was used to re-implement an existing benchmark algorithm to see how well it performs against the native implementation of the code. The results are very interesting and surprising, and they pave the way for a complete new point of reflection.
The second part of the experiment shows instead how the introduction of Device Fission to better use the memory allow to obtain considerable speedups over the unmanaged version of the same code.

3.2.1 Background: the LU algorithm

LU is an application level benchmark part of the NPB Suite (NAS Parallel Benchmark) that consists in a series of parallel aerodynamic simulations designed by NASA. LU is short for Lower-Upper Gauss-Seidel solver and the algorithm is basically a simplified Navier-Stokes equation solver that uses three-dimensional data grid for its calculations. The size of these data cubes is always N^3,

and for the purpose of this example we'll focus only on three specific classes of problems: Class A problems (size 64^3), Class B problems (size 102^3) and Class C problems (size 162^3). To perform the calculations, each cube of size $n \times n \times n$ is divided into 2D "slices" of size $n \times n \times 1$, and each "slice" is assigned to a different processor for computation. This algorithm is further optimized used a technique called *k-blocking*; **Figure 15** shows a visual representation of how data is explored, as you can see each block must wait for the adjacent block to be completely computed (the black ones) before computation can start.

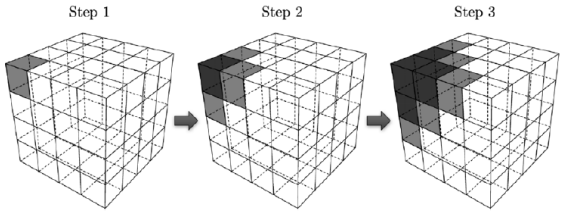

Figure 15: Data cubes used in the LU algorithm

3.2.2 The Experiment

Table 6 shows a list of the devices used in the experiment. One important aspect of OpenCL to note is its **code portability**: the same source code can be used on different devices with different numbers of cores and processing units, as OpenCL is able to scale over them automatically. This property of OpenCl is not to be underevaluated and it already offers many benefits:

- it is easier to maintain a single code that targets all platforms, as opposed to separate hand-tuned versions of the same code for each alternative platform.
- it reduces the risk of being locked into a single vendor solution.
- benchmarking is simplified, as the results can be compared from a single code source.
- it represents a "safer" investment for computing sites, as new codes (and ported legacy codes) will run on both existing and future architectures.

Platform	Compute Units	Processing Elements
Intel X5550, 2.66 GHz (x86 CPU)	4	4
Intel X5660, 2.80 GHz (x86 CPU)	12	12
NVIDIA Tesla C1060 (GPU)	30	240
NVIDIA Tesla C2050 (GPU)	14	448
AMD/ATI FirePro V7800 (GPU)	18	1440

Table 6: Platforms used in the experiment.

The goal of the first tests was to analyze the performances of OpenCL implementation of LU against native FORTRAN77 (for CPUs) and CUDA (for GPUs) implementations.
Figure 16 shows the results of the comparison between OpenCL and FORTRAN77. Each implementation was compiled using two different compilers for each approach.

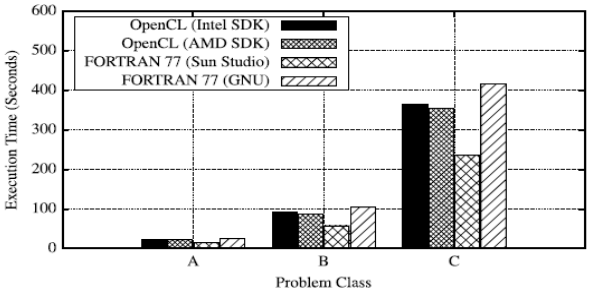

Figure 16: OpenCL vs Fortran Comparison

As we can see from the graph the advantage of OpenCL (if any) is very marginal, and in the case of the Sun Studio compiler, the FORTRAN77 implementation performs way better. This result is very interesting because it demonstrates two things:

1. that *OpenCL is not always the best option in term of performance*
2. that the compiler used can play a very important role

The reason of this OpenCL "failure" could be indeed attributed to the fact that the FORTRAN compiler is very mature and has a long story of optimizations and fine-tuning behind it, while OpenCL standard is quite young; and another interesting thing that can be seen from the graph is that there is practically no difference in performance between the two versions of the OpenCL implementation, and this means that no compiler is better optimized that the other.
Let's now see how OpenCL perform against the CUDA implementation (GPU), the results are shown in **Figure 17**:

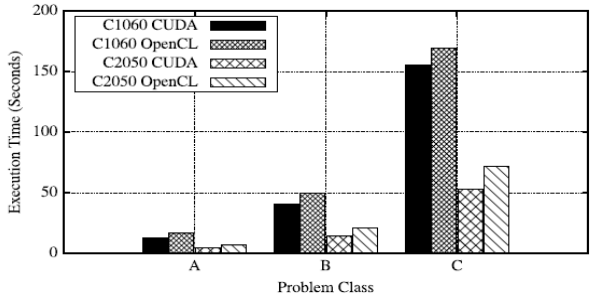

Figure 17: OpenCL vs CUDA Comparison

As we can see even in this case the CUDA implementation performs a little better, and the reason in still to be searched into specific optimizations offered by the CUDA compiler for NVIDIA boards.

3.2.3 Introducing Device Fission

The same benchmark will now be ran using device fissioning to exploit either temporal or spatial cache locality and therefore use the shared memory in a more efficient way. Previous executions of the benchmark were *unlikely to exhibit good memory behaviour*, because basic OpenCL implementation has no control over which processing unit the work items will be allocated.
In the new test setup, the LU benchmark will be executed over 2 different configurations created with device fissioning (**Figure 18**): two subdevices with 6 compute units each and 4 devices with 3 compute units each.

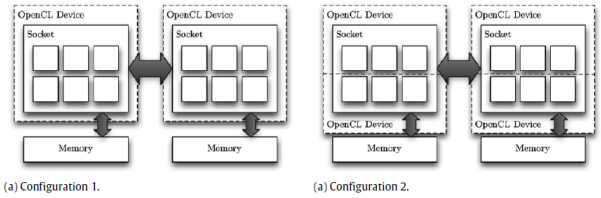

Figure 18: The two configurations used in the device fission tests

The results of the test are listed in **Table 7**, and they clearly suggest that device fission can provide significant performance improvements when used and configured properly, with an average speedup of 2.56x.

Num of Cores	With Device Fission	Witout Device Fission	Speedup
Configuration 1			
24	568.56	1158.60	2.03x
48	397.41	760.73	1.92x
96	276.68	503.25	1.81x
384	123.75	n.a.	n.a.
Configuration 2			
12	622.27	1884.87	3.02x
24	394.63	1158.60	2.93x
48	250.84	760.73	3.03x
192	100.71	320.53	3.18x

Table 7: Runtimes (in seconds) for two device fission configurations.

3.2.4 Conclusions

Even if at a first glance the results of this experiments may seem disappointing (OpenCL failed against FORTRAN and CUDA), it brought to light one interesting point about OpenCL implementations: compiler optimization. The main problem when using OpenCL is that calls to its functions are made by way of a library, and therefore the operation of these functions may not be necessarily defined until runtime, and this restricts the optimisation freedom of the host compiler. One solution to this problem may be a compiler that analyzes both the host *and* the OpenCL code at the same time, producing an optimized but *platform specific* code. Thus there is a conflict between potential compiler optimization and the vast code portability already provided by OpenCL: optimizing the compiler would make the code less portable.
The second part of the experiment shown instead a practical example on the strenght of device fission, as it could indeed offer the possibility to better manage the resources available and produce more performant code.

3.3 OpenCL remote clustering

In this section we'll present an interesting example [15] on how OpenCL can be used in a *distributed* and *heterogeneous* computer environment that can provide an opportunity to increase the performance of parallel and High-Performance Computing applications on clusters, by combining traditional multi-core CPUs, general-purpose GPUs and Accelerator devices.
One limitation of parallel computing using OpenCL is that most applications run their device-specific code (the kernels) only locally on the same computer were the hosts application runs.
Previous studies in this sector have already been made, and it has been demonstrated [16] that OpenMP (one of the main paradigms for HPC parallel programming) can be extended to use a heterogeneous cluster environment by letting the CPU portion of the application run on one local node, and the GPU kernels on cluster-wide remote devices. In this example we will present a first attempt to extend OpenCL in a similar way.

3.3.1 The VirtualCL Cluster Platform

VirtualCL (VCL) is an OpenCL wrapper that allows most applications to transparently utilize remote OpenCL devices as if they all resided on the same local computer.
The basic idea behind this platform is to provide to the application the impression of having of a single host with many local devices, no matter where they phisically are. Users can launch the OpenCL executable on the host computer, then VCL manages and transparently runs the kernels on the different nodes available. VCL is composed of three main components: the VCL library, the broker and the back-end daemon.
The **VCL library** is a cluster-wide front-end for OpenCL that gives applications transparent access to openCL devices in the cluster, hiding their actual location from the calling executable.
The **broker** is a daemon-process connected with the library via a UNIX socket that runs on every host computer. Its main tasks are to monitor for existence and availability of OpenCL devices in the cluster, to allocate them, and to authenticate and route messages between the applications and the back-ends.

The **back-end daemon** runs on each cluster node (and not the on the host, like the broker) and its goal is to execute kernels on behalf of the client application.

3.3.2 VCL Performances

One of the main factors that have to be taken into account when using a wrapper model like VCL is the **overhead** introduced by it. To evaluate it it is sufficient to measure the time taken by the application to run a sequence of identical kernels using the native OpenCL library locally and then the time to run the same kernels with VCL both on local and remote devices. The results for overhead testing are shown in **Table 8** and it is clear that the difference between local and remote use of VCL is very small (about 80 milliseconds) and almost independent from the size of the buffer used.

Buffer Size	**Native OpenCL Time** [ms]	**VCL Overhead (Local)** [ms]	**VCL Overhead (Remote)** [ms]
4 KB	96	35	113
16 KB	100	35	111
64 KB	105	35	106
256 KB	113	36	105
1 MB	111	34	114
4 MB	171	36	114
16 MB	400	36	113
64 MB	1354	33	112
256 MB	4993	37	111

Table 8: VCL overhead results.

The next step is to measure how VCL actually performs using benchmark applications. To run the test, executables from the SHOC (Scalable Heterogeneous Computing) benchmark suite were used, and once again the execution time was measured first for native OpenCL implementation, and then for local and remote VCL implementation. The results are presented in **Table 9**.

Application	**Native Time** [sec]	**VCL Time (Local)** [sec]	**VCL Time (Remote)** [sec]
BusSpeedDownload	0.89	0.88	0.88
BusSpeedReadback	0.91	0.89	0.89
DeviceMemory	31.44	56.78	243.81
KernelCompile	5.91	5.93	5.94
MaxFlops	186.98	156.74	211.20
QueueDelay	0.88	0.93	1.22
FFT	7.29	7.15	7.33
MD	14.08	13.66	13.80
Reduction	1.60	1.58	2.88
SGEMM	2.11	2.13	2.43
Scan	2.53	2.54	6.57
Sort	0.98	1.04	1.53
Spmv	3.25	3.30	5.91
Stencil2D	11.65	12.48	18.94
Triad	6.01	11.83	53.37
S3D	32.39	32.68	33.17

Table 9: VCL benchmark results.

By looking at the results it may seem that using VCL does not provide much advantage over native and local OpenCL implementation, but it is not to be forgotten that the goal was not to achieve better performances and faster execution, but rather to have remote and heterogeneous computation. Using cluster systems, much more computation power is available at the cost of network bandwidth and delay, so VCL has proven to perform better on applications with relatively long kernels and infrequent buffer-I/O operations, while those with many short kernels or with frequent or large I/O operations can't perform too well. The opinion of the authors of this study is that running parallel kernels efficiently on remote devices in a cluster is quite feasible and that VCL should be able to support largescale high-end parallel computing applications. An ideal cluster for running parallel HPC applications with the VirtualCL platform would be a collection of low-cost servers, each with several OpenCL devices, connected by a low-latency, high-bandwidth network to high-end hosting nodes with many cores and large memories.

4 Conclusions

Parallel computing is a very effective mean for achieving higher performances in modern computing and, thanks to their architecture, modern graphic adapters are the best choice for the implementation of parallel applications. Since GPU programming is quite different from the standard programming approach, dedicated APIs and framework such as CUDA and OpenCL are required. In the first chapter we presented an introduction to the general concepts behind the GPGPU program model, and then we focused on the OpenCL API. We introduced concepts like host, kernels, memory objects and we described how they should be implemented using some examples. In the second part of the second chapter, we talked about an important feature introduced by the last implementation of the OpenCL API: Device Fission.
Device Fissioning gives to the programmer total control over its application and allow him to better manage resources and power consumption, and this would be extremely useful in those scenarios where resources are very limited and power is crucial, such as in an embedded environment. To conclude the Device Fission introduction, we provided some useful example scenarios in which it can be applied.
In the third chapter we finally analized some real world applications as well as some research studies that involved OpenCL and device fissioning, and we saw that such framework can be very useful in a wide area of applications, especially in commercial situations where time-to-market is crucial or where power consumption and cooling costs are a limitation for expansion.

A Device Partitioning Example

This example illustrates how the device will be partitioned accordingly to the parameters passed to the **clCreateSubDevices()** function. (See **OpenCL (8)** on page 8) The architecture used in the example is a 2 processor (4 cores each) with L3 chache shared among the cores.

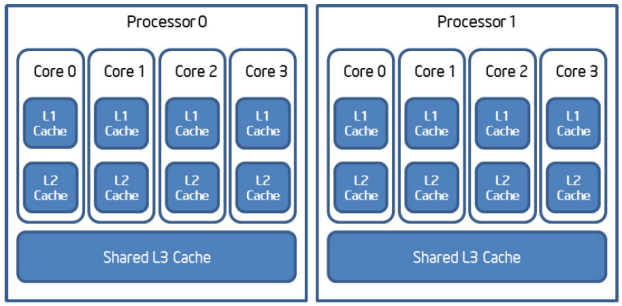

Figure 19: Architecure used in the example

Property List	Description	Result on the Example Target Machine
CL_DEVICE_PARTITION_EQUALLY, 8, 0	Partition the device into as many sub-devices as possible, each with 8 compute units.	2 sub-devices, each with 8 threads.
CL_DEVICE_PARTITION_EQUALLY, 4, 0	Partition the device into as many sub-devices as possible, each with 4 compute units.	4 sub-devices, each with 4 threads.
CL_DEVICE_PARTITION_EQUALLY, 32, 0	Partition the device into as many sub-devices as possible, each with 32 compute units.	Error! 32 exceeds the CL_DEVICE_PARTITION_MAX_ COMPUTE_UNITS.
CL_DEVICE_PARTITION_BY_COUNTS, 3, 1, CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0	Partition the device into two sub-devices, one with 3 compute units and one with 1 compute unit.	1 sub-device with 3 threads and 1 sub-device with 1 thread.
CL_DEVICE_PARTITION_BY_COUNTS, 2, 2, 2, 2 CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0	Partition the device into four sub-devices, each with 2 compute units.	4 sub-devices, each with 2 threads.
CL_DEVICE_PARTITION_BY_COUNTS, 3, 1, CL_DEVICE_PARTITION_BY_COUNTS_LIST_END, 0	Partition the device into two sub-devices, one with 3 compute units and one with 1 compute unit.	1 sub-device with 3 threads and 1 sub-device with 1 threads.
CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_NUMA, 0	Partition the device into sub-devices that share a NUMA node.	2 sub-devices with 8 threads each. Each sub-device is located on its own NUMA node.
CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_L1_CACHE, 0	Partition the device into sub-devices that share an L1 cache.	8 sub-devices with 2 threads each. The L1 cache is not shared in our example machine.
CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_L2_CACHE, 0	Partition the device into sub-devices that share an L2 cache.	8 sub-devices with 2 thread each. The L2 cache is not shared in our example machine.
CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_L3_CACHE, 0	Partition the device into sub-devices that share an L3 cache.	2 sub-devices with 8 threads each. The L3 cache is shared among all 8 threads within each processor.
CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_L4_CACHE, 0	Partition the device into sub-devices that share an L4 cache.	Error! There is no L4 cache.
CL_DEVICE_PARTITION_BY_AFFINITY_DOMAIN, CL_DEVICE_AFFINITY_DOMAIN_NEXT_PARTITIONABLE, 0	Partition the device based on the next partitionable domain. In this case, it is NUMA.	2 sub-devices with 8 threads each. Each sub-device is located on its own NUMA node.

Table 10: Device Partitioning Examples (Source: *OpenCL Device Fission for CPU Performance, Intel Corporation*)

B Code Examples of Device Fission implementation

B.1 Scenario 1: High Priority Subdevice

This code recreates the scenario described in Section 2.3.1 at page 8

```
//Host.cpp
int main()
{
#define MEM_SIZE 30 //work-size for each kernel
#define KERNEL_NAME "myKernel.cl"
#define MAX_SOURCE_SIZE 10000    //maximum length of the kernel source file
#define SUB_DEVICES_NUMBER 2

//OpenCL elements (device, platform, memory-objects....)
cl_device_id device_id;
cl_device_id sub_device_ids[SUB_DEVICES_NUMBER];
cl_device_id device_highPriority = NULL;
cl_device_id device_normalPriority = NULL;
cl_context context = NULL;
cl_command_queue command_queue_LO = NULL; //queue for low priority commands
cl_command_queue command_queue_HI = NULL; //queue for high priority commands
cl_mem inputData_true = NULL,inputData_false = NULL;
cl_program program = NULL;
cl_kernel kernel_A = NULL;
cl_kernel kernel_B = NULL;
cl_kernel kernel_C = NULL;
cl_platform_id platform_id = NULL;

cl_int ret;     //used to store return messages from OpenCL function calls

//Obtain platform and device (CPU)
ret = clGetPlatformIDs(1, &platform_id, NULL);
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);

//----> DEVICE FISSION
//query the device for the max number of subdevices & computeUnits
cl_uint maxSubdevices;
cl_uint maxComputeUnits;
ret = clGetDeviceInfo(device_id,CL_DEVICE_PARTITION_MAX_SUB_DEVICES,sizeof(cl_uint),&maxSubdevices,NULL);
ret = clGetDeviceInfo(device_id,CL_DEVICE_MAX_COMPUTE_UNITS,sizeof(cl_uint),&maxComputeUnits,NULL);

//if at least 2 subdevices can be created...
if(maxSubdevices >= 2)
{

cl_device_partition_property params[5];
params[0] = CL_DEVICE_PARTITION_BY_COUNTS;
params[1] = 1; // 1 CU reserved for high-priority computation
params[2] = maxComputeUnits - 1; // remaining CUs used for normal computation
params[3] = CL_DEVICE_PARTITION_BY_COUNTS_LIST_END; // End Count list
params[4] = 0; // End of the property list


// Create the sub-devices:
cl_uint dump;
ret = clCreateSubDevices(device_id, params, SUB_DEVICES_NUMBER, sub_device_ids, &dump);
device_highPriority = sub_device_ids[0];
    device_normalPriority =  sub_device_ids[1];
printf("\n(%d)Subdevices created correctly!",dump);
}
else
{
printf("\nError! Device cannot be partitioned");
exit(-1);
}

//Create context for the device
context = clCreateContext(NULL, 2, sub_device_ids, NULL, NULL, &ret);

//Create Command Queue and Memory Object (monodimensional buffer)
command_queue_LO = clCreateCommandQueue(context, device_normalPriority, 0, &ret);
command_queue_HI = clCreateCommandQueue(context, device_highPriority, 0, &ret);
int arg_false[MEM_SIZE];
int arg_true[MEM_SIZE];
for(int k=0;k<MEM_SIZE;k++)
{
arg_false[k] = 0;
arg_true[k] = 1;
}
inputData_true = clCreateBuffer(context, CL_MEM_READ_WRITE |
CL_MEM_COPY_HOST_PTR ,MEM_SIZE * sizeof(int), arg_true, &ret);
inputData_false = clCreateBuffer(context, CL_MEM_READ_WRITE |
```

```
CL_MEM_COPY_HOST_PTR ,MEM_SIZE * sizeof(int), arg_false, &ret);


/*Create the Kernel program and assign it to the context.
  We will use the clCreateProgramWithSource to build a kernel from an external .cl source file*/

//open the .cl file
FILE *fp;
fp = fopen(KERNEL_NAME, "r");
if (fp != NULL)
{
char * sourceCode = (char*)malloc(MAX_SOURCE_SIZE);
int source_size = fread(sourceCode, 1, MAX_SOURCE_SIZE, fp);
program = clCreateProgramWithSource(context, 1, (const char **)&sourceCode,(const size_t *)&source_size, &ret);
fclose(fp);

//Now we can build and execute the Kernel
ret = clBuildProgram(program, 2, sub_device_ids, NULL, NULL, NULL);
kernel_A = clCreateKernel(program, "lowPriorityTask", &ret);  //create the kernel called "lowPriorityTask"
kernel_B = clCreateKernel(program, "highPriorityTask", &ret); //create the kernel called "highPriorityTask"
kernel_C = clCreateKernel(program, "highPriorityTask", &ret); //create the kernel called "highPriorityTask"
//ret = clSetKernelArg(kernel_A, 0, sizeof(cl_mem), (void *)&inputData_true); //parameters passing to the kernel function
ret = clSetKernelArg(kernel_B, 0, sizeof(cl_mem), (void *)&inputData_true); //parameters passing to the kernel function
ret = clSetKernelArg(kernel_C, 0, sizeof(cl_mem), (void *)&inputData_false); //parameters passing to the kernel function

size_t globalWorkSize[1] = { MEM_SIZE };
size_t localWorkSize[1] = { 1 };
cl_event finished_C,finished_B;

/* First run: we'll execute low priority calls over the LO-device. Half of the higher priority
   tasks (Kernel_B) will be dispatched to the HI command-queue and will be executed immediately,
 while others will be dispatched the LO-device and will have to wait until Kernel_A has finished.*/
printf("\n\n ---With Device Fission: ---");
ret = clEnqueueNDRangeKernel(command_queue_LO, kernel_A, 1, NULL, globalWorkSize, localWorkSize,0, NULL, NULL);
    ret = clEnqueueNDRangeKernel(command_queue_HI, kernel_B, 1, NULL, globalWorkSize, localWorkSize,0, NULL, &finished_B);
    ret = clEnqueueNDRangeKernel(command_queue_LO, kernel_C, 1, NULL, globalWorkSize, localWorkSize,0, NULL, &finished_C);

//wait until completion
clWaitForEvents(1, &finished_C);
clWaitForEvents(1, &finished_B);

/* Now we will run the same program using only one device. The hi-priority task will have to wait
until the low priority command-queue has finished */
printf("\n\n ---Without Device Fission: ---");
ret = clEnqueueNDRangeKernel(command_queue_LO, kernel_A, 1, NULL, globalWorkSize, localWorkSize,0, NULL, NULL);
    ret = clEnqueueNDRangeKernel(command_queue_LO, kernel_B, 1, NULL, globalWorkSize, localWorkSize,0, NULL, NULL);
    ret = clEnqueueNDRangeKernel(command_queue_LO, kernel_C, 1, NULL, globalWorkSize, localWorkSize,0, NULL, NULL);
}
else
{
//Failed to open the kernel file
printf("\nFailed to load kernel.");
}

//Cleaning
ret = clFlush(command_queue_LO);
ret = clFinish(command_queue_LO);
ret = clFinish(command_queue_HI);
ret = clReleaseKernel(kernel_A);
ret = clReleaseKernel(kernel_B);
ret = clReleaseKernel(kernel_C);
ret = clReleaseProgram(program);
ret = clReleaseMemObject(inputData_true);
ret = clReleaseMemObject(inputData_false);
ret = clReleaseCommandQueue(command_queue_LO);
ret = clReleaseCommandQueue(command_queue_HI);
ret = clReleaseContext(context);

scanf("%c",NULL);
}


//myKernel.cl
__kernel void lowPriorityTask()
{
int workID = get_global_id(0);

int n = INT_MAX, first = 0, second = 1, next, c;

printf("\nI'm a [low ] priority task and i'm occupying the LO-Device\t (LO-LO)") ;
    for ( c = 0 ; c < n ; c++ )
   {
      if ( c <= 1 )
         next = c;
      else
```

```
      {
         next = first + second;
         first = second;
         second = next;
      }
    }

}

__kernel void highPriorityTask(__global int* data)
{
int workID = get_global_id(0);
printf("\nI'm a [high] priority task ");
data[workID] == 1 ? printf("and I'm running on the HI-device\t (HI-HI)") :
printf("but I'm running on the LO-device\t (LO-LO)");
}


}
```

B.2 Scenario 2: Virtual Pipeline Structure

This code recreates the scenario described in Section 2.3.4 at page 10

```
int main()
{
#define MEM_SIZE 40 //we will create a 2-dimensional NDrange
#define KERNEL_NAME "myKernel.cl"
#define MAX_SOURCE_SIZE 10000    //maximum length of the kernel source file
#define SUB_DEVICES_NUMBER 4

//OpenCL elements (device, platform, memory-objects....)
cl_device_id device_id;
cl_device_id sub_device_ids[SUB_DEVICES_NUMBER];
cl_device_id node_A,node_B,node_C,node_D;
cl_context context = NULL;
cl_command_queue command_queue[SUB_DEVICES_NUMBER];
cl_mem inputData_A = NULL;
cl_mem outputData_A = NULL;
cl_mem inputData_D_fromB = NULL;
cl_mem inputData_D_fromC = NULL;
cl_mem outputData = NULL;
cl_program program = NULL;
cl_kernel kernel[SUB_DEVICES_NUMBER];
cl_platform_id platform_id = NULL;

cl_int ret;     //used to store return messages from OpenCL function calls

//Obtain platform and device (CPU)
ret = clGetPlatformIDs(1, &platform_id, NULL);
ret = clGetDeviceIDs(platform_id, CL_DEVICE_TYPE_CPU, 1, &device_id, NULL);

//----> DEVICE FISSION
//query the device for the max number of subdevices & computeUnits
cl_uint maxSubdevices;
cl_uint maxComputeUnits;

ret = clGetDeviceInfo(device_id,CL_DEVICE_PARTITION_MAX_SUB_DEVICES,sizeof(cl_uint),&maxSubdevices,NULL);
ret = clGetDeviceInfo(device_id,CL_DEVICE_MAX_COMPUTE_UNITS,sizeof(cl_uint),&maxComputeUnits,NULL);

//if at least 4 subdevices can be created...
if(maxSubdevices >= SUB_DEVICES_NUMBER)
{
cl_device_partition_property params[7];
params[0] = CL_DEVICE_PARTITION_BY_COUNTS;
params[1] = 1;
params[2] = 1;
params[3] = 1;
params[4] = 1;
params[5] = CL_DEVICE_PARTITION_BY_COUNTS_LIST_END;
params[6] = 0; //end of list

// Create the sub-devices:
cl_uint dump;
ret = clCreateSubDevices(device_id, params, 4, sub_device_ids, &dump);
node_A = sub_device_ids[0];
node_B = sub_device_ids[1];
node_C = sub_device_ids[2];
node_D = sub_device_ids[3];
printf("\n(%d)Subdevices created correctly!",dump);
}
else
{
```

```
printf("\nError! Device cannot be partitioned");
system("pause");
exit(-1);
}

//Create context for the device
context = clCreateContext(NULL, 4, sub_device_ids, NULL, NULL, &ret);

//Create Command Queue and Memory Object (monodimensional buffer)
for(int k=0;k<SUB_DEVICES_NUMBER;k++)
command_queue[k] = clCreateCommandQueue(context, sub_device_ids[k], 0, &ret);


//open the .cl file
FILE *fp;
fp = fopen(KERNEL_NAME, "r");
if (fp != NULL)
{
char * sourceCode = (char*)malloc(MAX_SOURCE_SIZE);
int source_size = fread(sourceCode, 1, MAX_SOURCE_SIZE, fp);
program = clCreateProgramWithSource(context, 1, (const char **)&sourceCode,(const size_t *)&source_size, &ret);
fclose(fp);

//Now we can build and execute the Kernel
ret = clBuildProgram(program, SUB_DEVICES_NUMBER, sub_device_ids, NULL, NULL, NULL);
kernel[0] = clCreateKernel(program, "Kernel_A", &ret);
kernel[1] = clCreateKernel(program, "Kernel_B", &ret);
kernel[2] = clCreateKernel(program, "Kernel_C", &ret);
kernel[3] = clCreateKernel(program, "Kernel_D", &ret);

outputData_A = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR ,MEM_SIZE * sizeof(int), NULL, &ret);
inputData_D_fromB = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR ,MEM_SIZE * sizeof(int), NULL, &ret);
inputData_D_fromC = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR ,MEM_SIZE * sizeof(int), NULL, &ret);
outputData = clCreateBuffer(context, CL_MEM_READ_WRITE | CL_MEM_COPY_HOST_PTR ,MEM_SIZE * sizeof(int), NULL, &ret);
//Initializing Input Data
size_t globalWorkSize[1] = { MEM_SIZE };
size_t localWorkSize[1] = { 1 };
cl_event finished_A,finished_B,finished_C,finished_D;
int data[MEM_SIZE];
inputData_A = clCreateBuffer(context, CL_MEM_READ_ONLY | CL_MEM_COPY_HOST_PTR ,MEM_SIZE * sizeof(int), data, &ret);
ret = clSetKernelArg(kernel[0], 0, sizeof(cl_mem), (void *)&inputData_A); //parameters passing to the kernel function
ret = clSetKernelArg(kernel[0], 1, sizeof(cl_mem), (void *)&outputData_A); //parameters passing to the kernel function

//executing NODE A
ret = clEnqueueNDRangeKernel(command_queue[0], kernel[0], 1, NULL, globalWorkSize, localWorkSize,0, NULL, &finished_A);
clWaitForEvents(1, &finished_A);

//executing NODE B and C simultaneously
ret = clSetKernelArg(kernel[1], 0, sizeof(cl_mem), (void *)&outputData_A); //parameters passing to the kernel function
ret = clSetKernelArg(kernel[1], 1, sizeof(cl_mem), (void *)&inputData_D_fromB); //parameters passing to the kernel function
ret = clSetKernelArg(kernel[2], 0, sizeof(cl_mem), (void *)&outputData_A); //parameters passing to the kernel function
ret = clSetKernelArg(kernel[2], 1, sizeof(cl_mem), (void *)&inputData_D_fromC); //parameters passing to the kernel function
ret = clEnqueueNDRangeKernel(command_queue[1], kernel[1], 1, NULL, globalWorkSize, localWorkSize,0, NULL, &finished_B);
ret = clEnqueueNDRangeKernel(command_queue[2], kernel[2], 1, NULL, globalWorkSize, localWorkSize,0, NULL, &finished_C);
clWaitForEvents(1, &finished_B);
clWaitForEvents(1, &finished_C);

//executing NODE D


ret = clSetKernelArg(kernel[3], 0, sizeof(cl_mem), (void *)&inputData_D_fromB); //parameters passing to the kernel function
ret = clSetKernelArg(kernel[3], 1, sizeof(cl_mem), (void *)&inputData_D_fromC); //parameters passing to the kernel function
ret = clSetKernelArg(kernel[3], 2, sizeof(cl_mem), (void *)&outputData); //parameters passing to the kernel function

ret = clEnqueueNDRangeKernel(command_queue[3], kernel[3], 1, NULL, globalWorkSize, localWorkSize,0, NULL, NULL);

}
else
{
//Failed to open the kernel file
printf("\nFailed to load kernel.");
}

//Cleaning
for(int k=0;k<SUB_DEVICES_NUMBER;k++)
{
ret = clFlush(command_queue[k]);
ret = clFinish(command_queue[k]);
ret = clReleaseKernel(kernel[k]);
ret = clReleaseCommandQueue(command_queue[k]);
}
ret = clReleaseProgram(program);
ret = clReleaseMemObject(inputData_A);
ret = clReleaseMemObject(outputData_A);
ret = clReleaseMemObject(inputData_D_fromB);
```

```
ret = clReleaseMemObject(inputData_D_fromC);
ret = clReleaseMemObject(outputData);


ret = clReleaseContext(context);

scanf("%c",NULL);
}

//myKernel.cl
__kernel void Kernel_A(__global int* input, __global int* output)
{
printf("\nExecuting NODE A..") ;
}

__kernel void Kernel_B(__global int* input, __global int* output)
{
printf("\nExecuting NODE B..") ;
}


__kernel void Kernel_C(__global int* input, __global int* output)
{
printf("\nExecuting NODE C..") ;
}

__kernel void Kernel_D(__global int* input1,__global int* input2, __global int* output)
{
printf("\nExecuting NODE D..") ;
}

}
```

References

[1] : Gpgpu.org: About gpgpu.org. http://gpgpu.org/about

[2] Arora, M., Nath, S., Mazumdar, S., Baden, S.B., Tullsen, D.M.: Redefining the role of the cpu in the era of cpu-gpu integration. Micro, IEEE **32**(6) (Nov.-Dec.) 4–16

[3] Goddeke, D.: GPGPU Basic Math Tutorial. Universitat Dortmund, Dortmund, Germany

[4] Group, K.O.W.: The OpenCL Specification. (2012)

[5] Corporation, N.: Cuda c programming guide. http://docs.nvidia.com/cuda/cuda-c-programming-guide/ (2013)

[6] Gaster, B.R.: Opencl device fission. http://www.khronos.org/assets/uploads/developers/library/2011_GDC_OpenCL/AMD-OpenCL-Device-Fission_GDC-Mar11.pdf (2011)

[7] : Gpgpu apis compared. http://siroro.co.uk/2011/08/02/gpgpu-programming-languages-compared (2011)

[8] Scarpino, M.: A gentle introduction to opencl. http://www.drdobbs.com/parallel/a-gentle-introduction-to-opencl/231002854 (2011)

[9] Corporation, I.: Opencl device fission for cpu performance. (2012)

[10] Pennycook, S., Hammond, S., Wright, S., Herdman, J., Miller, I., Jarvis, S.: An investigation of the performance portability of opencl. Journal of Parallel and Distributed Computing (0) (2012) –

[11] Leskela, J.: Gpgpu on mobile devices. http://www.cvg.ethz.ch/teaching/2011spring/gpgpu/mobile_gpgpu.pdf (2009)

[12] Corporation, A.: Implementing fpga design with the opencl standard. http://www.altera.com/literature/wp/Fwp-01173-opencl.pdf (2013)

[13] Doris Chen, D.S.: Using opencl to evaluate the efficiency of cpus, gpus and fpgas for information filtering. http://www.altera.com/products/software/opencl/ieee-cp-information-filtering.pdf (2012)

[14] J.W. R.Walsh, S.C., Joseph, E.: Special study of power and cooling practices and planning at hpc data centers (2009)

[15] A. Barak, A.S.: The virtualcl (vcl) cluster platform

[16] A. Barak, T. Ben-Nun, E.L., Shiloh, A.: A package for opencl based heterogeneous computing on clusters with many gpu devices. (2010)