TITLE TO BE DEFINED

The report subtitle

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

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

Received: April, 01 2011

Abstract

This is a normal text in 10pt type size and 12pt line spacing. Place here a summary of your report. Focus on what is the problem and how you propose to tackle it. Briefly resume achieved results.
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Cras dapibus. Donec fermentum mauris non lectus. Nulla porttitor pede id ante. Donec in erat pellentesque erat ultrices rutrum. Pellentesque id tellus. Donec vel mi non dui adipiscing tempor. Nunc laoreet pede ut lectus. Aenean aliquam quam a sem. Duis at elit? Ut dolor massa, dictum vel; convallis quis, ullamcorper id, urna.

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 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. Developers who port their applications to GPUs are able to achieve speedups of orders of magnitude vs. optimized CPU implementations.[1]
The reason why GPUs are so fast has to be searched in the nature of what graphic rendering is about: an *intensive parallel computetion* and , for this reason, GPUs are engineered to work specifically on *data processing*, rather than data caching and flow control like CPUs. (**Figure 1**)

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

Moreover, computational units of GPUs are specialized to perform simple operation but *in parallel*.

GPUs have been built to exploit application parallelism, and a single graphic adapter can host hundreds, if not thousands, of cores; this translates in TeraFlops of operations per second compared to the "'few"' GigaFlops a CPU can handle alone. (**Figure 2**)

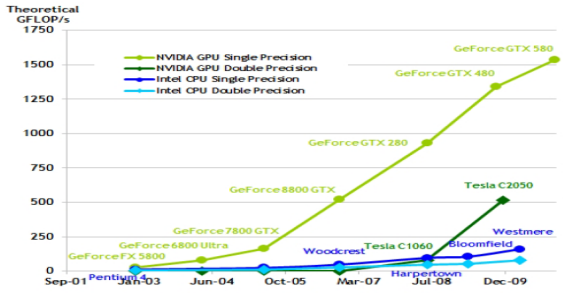

Figure 2: CPU vs GPU growth rate.

Due to its parallel nature, GPU computing can be very effective for applications involving huge amount of data, especially if the data can be structured in structures like vectors and arrays. Parallel GPU computation can be applied in various applications and research-field, from computer vision, to mathematical simulations, to bio-informatics, and it often allows to achieve in a few months the same results that would have required years to achieve with a standard CPU-only approach (with speedups of the order of 250x,350x).

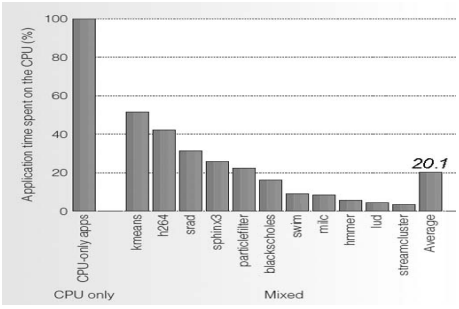

Figure 3: Time spent on the CPU on various benchmarks using mixed CPU-GPU computation. [2]

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 more effectively than the GPU. Furthermore, not all the code can be mapped easily and clearly on the GPU, as programming in this way can be way more difficult than programming in a standard way, it is not possible to simply "'port"' the code from the CPU to the GPU.

1.1 Basic Principles

In this section we'll introduce some of the basic principles behind GPGPU programming. Some of these concepts are directly related to the "'graphical-oriented"' way in which the GPU operates, and having a little kwowledge of basic concepts of computer gprahics can help to understand why GPU computing is so fast. For instance, GPUs can handle bidimensional matrices natively because that's the natural way to represent images in memory, while CPUs are limited to always work on single dimension arrays.
A GPU is basically a *stream processor*[1]: a single **kernel** is executed over a stream of data in a monolitic fashion. Thanks to the GPU architecture, the data can be elaborated in parallel, and more than one kernel can be executed at the same time.

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.
Besides, 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 can reach sizes of 8192*8192. 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 of the 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 applicationis 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, like a GPU shader apply the same transformation on multiple pixels at the same time to obtain a new image.
The output of kernel computation will be a new memory object that contains the result of the calculation. If we keep in mind that GPUs were born to elaborate graphics data, it will be easier to understand how kernels work and 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:

```
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;
}
```

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

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 discuss how to implement some kernel application in Section 2.1.2.

> **OpenCL Reference (2)**
> For Example, using the OpenCL API, you can create objects of type **cl_kernel** and initialize them with the **clCreateKernel()** function.

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**.

	Drawing perspective
1)	Assign the input texture to a texture channel of the graphic adapter
2)	Setting the drawing surface
3)	Define the quad to be drawn. The quad must fit the entire viewport and the texture must be mapped to it.
4)	Load the shader
5)	Render the quad
	Computation perspective
1)	Define the input data and feed it to the kernel
2)	Define the memory object where the output data will be stored
3)	Initialize the indices and setting the bounds of the loop
4)	
5)	Iterate through data and execute the kernel.

Table 1: Drawing and Kernel Execution comparison

After the computation has been performed, the result is stored in the target surface.

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**).

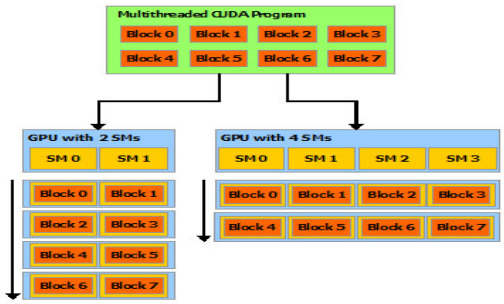

Figure 4: CUDA executables are automatically scaled over the various SMs (Stream Multiprocessors): a GPU with more multiprocessors will automatically execute the program in less time than a GPU with fewer multiprocessors.

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 used on some embedded or mobile devices like Android phones and iPhones.
OpenCL applications can also be executed on standard CPUs and don't need to have a graphic adapter installed on the system. The current version of OpenCL is 1.2 and it was released in 2011. We will discuss OpenCL 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/), DitectCompute is the GPU computing API developed by Microsoft which 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**.
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 (like Compute Units) refer to the underlying hardware that has to run our application.
The architecture of an OpenCL application is summarized in **Figure** 5

The Host

The Host can be viewed as the **controller** of the application. It usually doesn't perform tasks specific to the domain of the application and its main function is 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.

OpenCL Reference (3)
The host is a standard C/C++ application implemented inside a 'main' function. Every host always need these five data structures to properly setup an OpenCL environment: **cl_device_id**, **cl_kernel**, **cl_program**, **cl_command_queue**, and **cl_context**.

The Kernels

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 Multiprocessor 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.

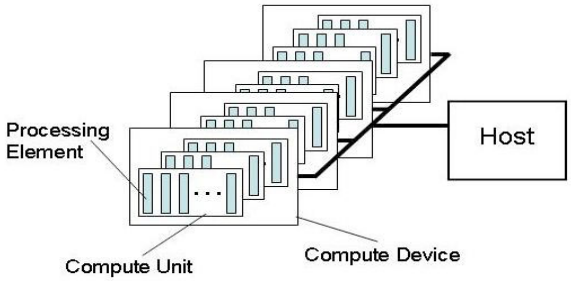

Figure 5: OpenCL Architecture

2.1.1 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 2.1.3) for the kernels and to dispatch them for execution.

2. the **kernels** are executed on the OpenCL device(s) and compute their code over a stream of data.

3. the resulting 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).

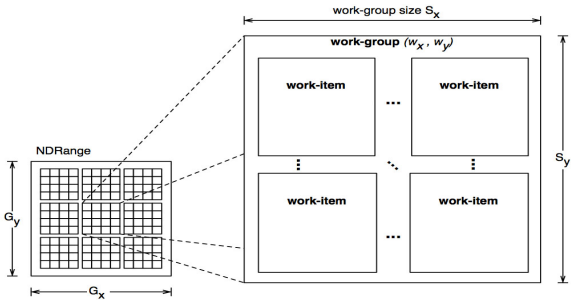

Figure 6: Work-items mapped over a two-dimensional NDRange. As you can see, work-items can be organized in Work-Groups and every work-item has both a global and a local ID inside its work-group.

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 enqueued in a specific command-queue, and they work over a NDRange memory object.

2.1.2 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. Each work-item has its own local environment that can be accessed using the work-item related 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

2.1.3 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.
It 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 Processing Elements of a 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.

2.1.4 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 is 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 [8] (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 [**?**]

- 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 2.1.3), 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 subdevice 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 subdevices 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**:

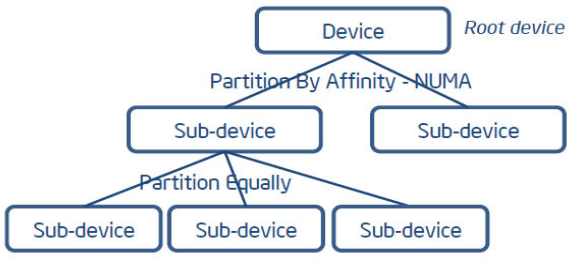

Figure 7: Sub-devices can be further partitioned to create an hierarchy of devices. Each node of the tree can be partitioned using different partition parameters.

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.

2.3.1 High-Priority dedicated sub-device

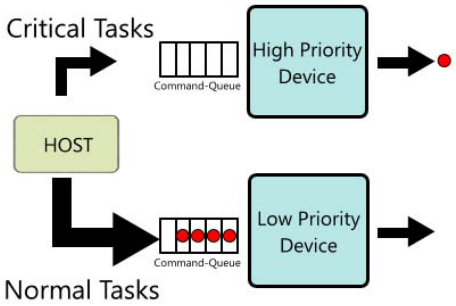

Figure 8: Scenario 1: a sub device is created to compute high-priority tasks

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. This configuration can be simply achieved by partitioning the device *by counts*, reserving a few cores 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 B**.

```
// 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

In this scenario we created two sub-devices, one with 3 Compute Units (the low-priority one) and one with just one CU (the high-priority one) and 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 stall the low-priority subdevice with an infinite while loop: the normal tasks were not able to continue their execution, but the the critical ones were executed normally as soon as they were dispatched thanks to the different routing.

2.3.2 Memory Proximity

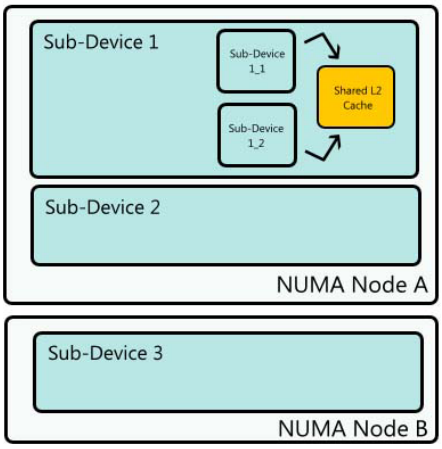

Figure 9: The main device is partitioned into 2 subdevices that have Compute Units that share the same NUMA node. Subdevice 1 is further partitioned to obtain two devices that shares the same L2 Cache.

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 icrease 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-dependant, and it is not guaranteed that a particular partitionig scheme will work on different

platforms.

> **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_BT_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);
```

2.3.3 Warm Cores Exploitation

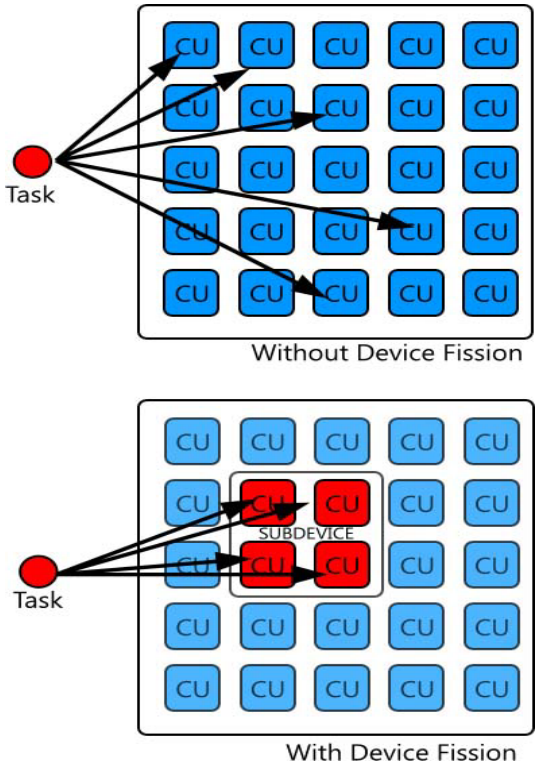

Figure 10: Without device fission there is no real control on which cores will be used, and tasks may be dispatched to 'cold' cores. With device fission we can redirect tasks to a specific (small) subdevice and exploit 'warm' cores.

Without device fission, when a new command is submitted to the command-queue it may happen that it will br 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 behaviour 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

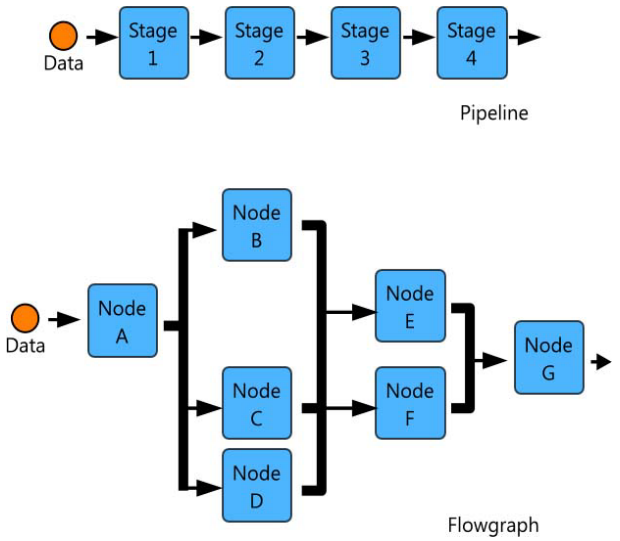

Figure 11: Device partitioning allows to create 'virtual' pipelines

Since device fission gives total control over task dispatching, we can use it to create complex structures like

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 no device can "'overlap"' resources with another.
SISTEMARE.....Use Partition By Affinity to create N sub-devices one sub-device for each NUMA node. The sub-devices can then use all NUMA node's resources including all of the available cache.

3 Parallel Computation and Device Fission on Embedded Systems

If we want to discuss about parallel computation on embedded systems, first we have to analyze which hardware is currently available and what their capabilities are. In this section we will briefly analyze processors (CPUs and GPUs) that allow some extent of parallelism (4+ cores) and are suitable for an embedded environment.

3.1 Multicore CPUs for embedded systems

AMD Opteron

Opteron CPUs are multi-core high-performance processors targeted for high-end embedded systems and networking and telecommunications infrastructures. Other applications may include storage systems and medical equipment. Opteron processors can host up to 16 cores (6000 series) and even the base models come with 8 cores (4000 series), and allow scalability up to 4 sockets for a total of 64 compute units.[**?**]

Architecture	x86 64bit
Applications	High-end embedded systems
Cores	8,16
Frequency range	1,8 - 1,3Ghz
Parallelism	Up to 64 compute units (4 CPUs x 16 cores)

OpenCL and Device Fission - Since Opteron CPUs are based on x86 architecture, they natively support OpenCL runtime and drivers, and developing OpenCL application for these CPUs is relatively simple. Applications must be compiled and executed using AMD specific drivers that are part of the AMD APP SDK, obtainable for their website (http://developer.amd.com). A video example demonstrating how an application can scale over 24 opteron (desktop) cores can be found on AMD's youtube channel at http://www.youtube.com/user/AMDUnprocessed/

3.1.1 ARM Cortex-A and -R Family

Cortex-A (Application) and Cortex-R (RealTime) are two processors family developed by ARM and the most widely used processors in embedded and mobile devices. Their range of applications vary from mobile devices (smartphones, tablets, digital camera) to automotive systems and medical equipment. Cortex-A processors host 4 cores while Cortex-R are usually dual core CPU , but more than one ARM CPU can easily be interconnected using AMBA techonlogy to create SoCs (System on Chip) with up to 16 cores.
OpenCL and Device Fission - armv7/8 architecture, compatibility Device fission -> big.little

3.1.2 Intel Embedded Platforms

(http://edc.intel.com)

3.2 GPUs for embedded systems

arm mali

3.3 ARM GPUs: Mali Processors Overview

Cortex processors are divided into three families:

- **Application Processors [Cortex-A]**: high-performance processors targeted to mobile and embedded devices such as smartphones, tablets and digital TVs. This architecture comes in both single-core (A?) and multi-core mode, so it is possible to perform parallel computation on devices that implements such processors. Cortex-A processors may include a dedicated NEON processing unit for graphics and multimedia.

- **Realtime Embedded Processors [Cortex-R]**
- **Microcontrollers [Cortex-M]**

3.4 Tre.punto.due

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Mauris eget mauris. Nulla facilisi. Ut condimentum tempor eros? Integer metus mauris, consectetur sit amet, tempor a, facilisis eu, nisl. Vestibulum at turpis. Ut vitae tortor pretium nisl vestibulum blandit. Nulla nibh urna, semper et, elementum at, mattis ut, nisi! Cum sociis natoque penatibus et magnis dis parturient montes, nascetur ridiculus mus. Morbi vel ligula eget lacus convallis venenatis. Aliquam lacinia tincidunt felis. Ut dui.

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 7) The architecture used in the example is a 2 processor (4 cores each) with L3 chache shared among the cores. (**Figure** A)

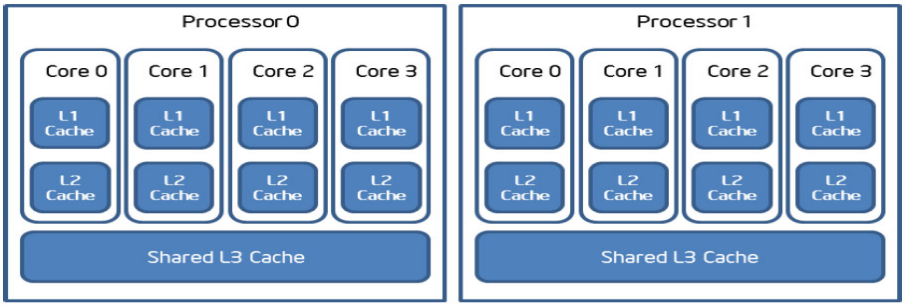

Figure 12: Architecure used in the example

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

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-opencl-c (2011)

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