POLITECNICO DI TORINO

Corso di Laurea Magistrale

in Ingegneria Elettronica

Tesi di Laurea Magistrale

# Differences between CUDA and OpenCL through a SAR focusing system

Relatore

Prof. Claudio Passerone

Candidato

Ten. Ing. Alberto Matta

Anno Accademico 2019/2020

*A mia moglie,*

*per il continuo e fondamentale sostegno che mi ha dato, senza la quale non avrei raggiunto questo risultato. Lei è la ragione che mi spinge sempre a migliorarmi.*

# Index

Introduction 1

Chapter one: GPUs

1.1 Introduction 2

1.2 GPU Architecture 3

1.3 CPU vs GPU 5

1.4 GPGPU 7

1.5 Stream Processing 10

1.6 The future and challenges 14

Chapter two: CUDA and OpenCL

2.1 Introduction 17

2.2 CUDA 17

2.2.1 CUDA development model 22

2.2.2 CUDA toolchain 22

2.2.3 C for CUDA kernel programming 23

2.3 OpenCL 24

2.3.1 OpenCL development model 30

2.3.2 OpenCL toolchain 30

2.3.3 C for OpenCL kernel programming 30

Chapter three: CUDA vs OpenCL

3.1 Introduction 32

3.2 Work environment 34

Chapter four: SAR focusing system

4.1 Introduction 54

4.2 SAR ( Synthetic Aperture Radar) 54

4.3 History 54

4.4 The SAR system analyzed 56

4.5 How it works 57

4.6 How SAR pictures the world 62

Chapter five: porting CUDA to OpenCL in SAR focusing system

5.1 Introduction 66

5.2 The initial work 66

5.3 Achievable improvements 78

5.3.1 Porting CUDA to OpenCL 78

5.3.2 Modifying the code 93

Conclusion 96

References 97

# Introduction

The aim of this work is to highlight the differences and the similarities between two different platforms for GPUs' coding, CUDA (Compute Unified Device Architecture) and OpenCL (Open Computing Language). The work will be explained further through a practical example, exposing how the code of an actual SAR (Synthetic Aperture Radar) focusing system is organized and showing the highlights and weaknesses of such system in comparison to a different coding of the same system. In conclusion, the main key points of the two different approaches applied to the SAR focusing system will be analyzed.

Currently, both platforms are largely used and each of them has its highlights and weaknesses, so they are both used but for different aims and targets. The SAR focusing system which will be analyzed is written in CUDA, and an OpenCL code for it has not been written yet.

In this work I will show why and when it is better to choose one platform or the other, and I will show which solutions have to be made to porting the CUDA code to OpenCL code for the SAR focusing system.

Firstly, I will talk in a general way about GPUs, to better frame the topic of interest.

Secondly, I will present in a deeper way the two different platforms, CUDA and OpenCL.

Thirdly, I will compare the two platforms, showing their highlight and weaknesses, and why and when it is better choosing one or the other.

Fourthly, I will talk in a general way about the SAR focusing system, to better understand the example and in such a way to better explain the following step.

Fifthly, I will talk about how to porting the SAR focusing system's CUDA code in OpenCL code and why doing it could be useful, and why not.

# Chapter one: GPUs

## 1.1 Introduction

A GPU (Graphics Processing Unit) is a specialized electronic circuit designed to rapidly calculate high intensive tasks (as arithmetic ones) in parallel. It is present in very different scenarios, such as embedded systems, personal computers, smartphones, game consoles and workstations. The main reason to the rising presence and importance of GPUs is that they are very efficient in doing high stressful parallel computations; a CPU (Central Processing Unit) doing the same task will take a lot of more time to achieve the desired results.

The first GPU models were introduced in the seventies, mainly for gaming purposes, in a form of specialized graphic circuits. The term GPU was introduced by Sony in 1994, in reference to their 32 bit graphic unit (designed by Toshiba) which is part of the PlayStation gaming console. The GPUs obtained a boost in their performance and utilization starting from the new century. The main actor of this rising in importance is Nvidia, which in 2007 released the CUDA platform, the very first widely adopted programming model for GPU computing. The problem with CUDA was that Nvidia GPUs were the only ones that could run on CUDA, so Nvidia had the monopoly of GPUs. Nowadays, OpenCL, released in 2009 from Khronos Group, is broadly supported. OpenCL is an open standard which runs on both CPUs and GPUs of different vendors, so its main feature is the portability. Intel, AMD, Nvidia and ARM cards are supported, in such a way that OpenCL became the first competitor of CUDA in GPUs programming.

At the very beginning, GPUs job was intended for graphic processing purposes only. Through the years, CPUs became more powerful and with a lot of more transistors, more or less following the Moore's Law, which says that every two years the number of transistors in a dense IC (integrated circuit) double up. This is an empirical law, based on observation of an historical trend. So, the transistors became smaller and smaller (nanometric order of magnitude), to be able to have more and more of them on the same IC (billions of transistors are present in modern ICs). In addition to that, the traditional way of improving performance by increasing the microprocessor core clock frequency leaded to an increase of the energy consumption. The greatest problem with the above solution is the dissipation of heat: such a concentration of transistors and the increased clock rates lead to a very hot environment, which is difficult to cool down and dissipate, especially in consumer products with very tight cost constraints. The heat leads to other problems, such as anomalous behaviors from the IC or anomalous power consumptions. For these reasons, the market trend has changed and multicore processors ICs became the most popular solution. Instead of adding more transistors on the same IC, this solution aims to rising the number of CPUs, called cores, in such a way to have an increase in speed mostly for multithreading and parallel computing techniques. In this scenario, GPU become more popular not only for graphic purposes, but also for high intensive computational ones. In fact, GPUs have a lot of space dedicated to arithmetical computations respect to CPUs, and they show a very big speed up in operations that requires such abilities. For this reasons, GPU which are used not only for graphic processing tasks are called GPGPU, which stands for General Purpose GPU. In the environment in which we work there are one or more CPUs, which have to communicate and collaborate among them, and one or more GPUs, which have to do the hard work to speed up the required operations. CPUs maintain the control of the environment, while the GPUs are directed on high parallel computational problems. So, the CPU is called *host*, while the GPU is called *device*.

## 1.2 GPU Architecture

Nowadays, GPUs communicate with the CPU through PCI-Express. Earlier generations used AGP (Accelerated Graphics Port), which is an extended version of the original PCI I/O bus. Graphics applications use OpenGL (Segal and Akeley, 2006) or Direct3D (Microsoft DirectX Specification) API (Application Programming Interface) functions, which is a computing interface that defines interactions between multiple software, that use the GPU as a coprocessor. The APIs send a variety of commands, instructions, and data to the GPU through a graphic device driver optimized for the particular GPU which is used. The graphics logical pipeline is described in the following image. It consists of different stages, which will be described below.

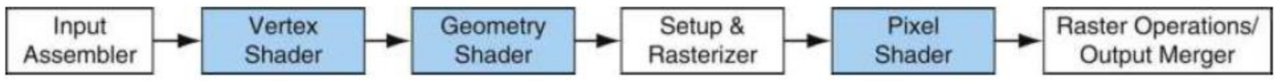


Figure 1.1 Graphics logical pipeline

An artificial image is synthesized from a model consisting of:

- Geometric shapes and appearance descriptions (color, surface, texture, …) for each object in the scene
- Environment descriptions, such as lighting, atmospheric properties, …

The result of the synthesis is a 2D representation.

Vertex shader consists in transforming a 3D polygonal (triangle) representation of the object's surface to a 2D projection of triangles (translations, rotations and scaling are used). Output vertices also include attributes, like color and surface orientation. So, we have a stream of vertices in 3D space as input and vertices positioned on screen as output.

Geometry shader consists in assembling vertices into primitives. Additional operations, such as clipping primitives, are performed to avoid downstream processing that will not contribute to image. So, we have independent vertices in 2D as input and vertices grouped into primitives as output.

Setup & rasterizer consists in converting each 2D triangle to a collection of pixel fragments, corresponding to a discrete sampling of the triangle over a uniform grid. Other operations are performed, such as compute covered pixels, sample vertex attributes and generate a parametric description of the triangle. So, we have triangles positioned on the screen as input and pixel fragments as output.

Pixel shader consists in processing each pixel fragment to compute a final color value (shading). This also include geometric or appearance descriptions related to both object and environment (e.g. texture mapping to determine material properties or lighting). So, we have fragments as input and shaded fragments as output.

Finally, resulting shaded pixel fragments are written to a buffer (frame buffer), where depth buffering is used to determine whether one fragment is closer to the viewer than another at a specific pixel location (occlusion).

Unified GPU architectures, like the one described above, are based on a parallel array of many programmable processors, unlike their predecessors which had separate processors, each one dedicated to different processes. They unify all the shader processes described before and parallel computing on the same processors. The programmable processor array is firmly integrated with fixed function processors for texture filtering, rasterization, raster operations, anti-aliasing, compression, decompression, display, video decoding, and high-definition video processing. These fixed-function processors significantly outperform more general programmable processors in terms of absolute performance constrained by area, cost, power budget. In contrast to multicore CPUs, manycore GPUs are construct in a different architectural way, almost focused on executing many parallel threads efficiently on many processor cores. For that reason, in GPUs are present many simpler cores, optimized for data parallel computing among groups of threads. So, while CPUs transistor budget is balanced among computation, on-chip caches and overhead, GPUs transistor budget is mainly devoted to computation. Such a multitude of cores are typically organized into multithreaded multiprocessors, so many processors which each one able to provide multiple threads of execution concurrently.

## 1.3 CPU vs GPU

Let’s take some initial considerations about these two different processors from one of the major vendors in that field, Intel[1]. Directly from its website we can make ourself a general idea about the main differences between CPUs and GPUs.

Constructed from millions of transistors, the CPU can have multiple processing cores and is commonly referred to as the brain of the computer. It is essential to all modern computing systems as it executes the commands and processes needed for your computer and operating system. The CPU is also important in determining how fast programs can run, from surfing the web to building spreadsheets.

The GPU is a processor that is made up of many smaller and more specialized cores. By working together, the cores deliver massive performance when a processing task can be divided up and processed across many cores.

These two processors have a lot in common: both are critical computing engines, both are silicon-based microprocessors, both handle data. But CPUs and GPUs have different architecture and are built for different purposes. The CPU is suited to a wide variety of workloads, especially those for which latency or per-core performance are important. A powerful execution engine, the CPU focuses its smaller number of cores on individual tasks and on getting things done quickly. This makes it uniquely well equipped for jobs ranging from serial computing to running databases. GPU began as specialized ASICs developed to accelerate specific 3D rendering tasks. Over time, these fixed function engines became more programmable and more flexible. While graphics and the increasingly lifelike visuals of today’s top games remain their principal function, GPUs have evolved to become more general purpose parallel processors as well, handling a growing range of applications.

To better understand how CPUs and GPUs are actually made we can have a look to both their architectures in the following figure, where there are the highlighted main differences between the two processors.

[1] © Intel Corporation, “CPU vs. GPU: Making the Most of Both”, “CPU vs. GPU: What’s the Difference?”, https://www.intel.com/content/www/us/en/products/docs/processors/cpu-vs-gpu.html

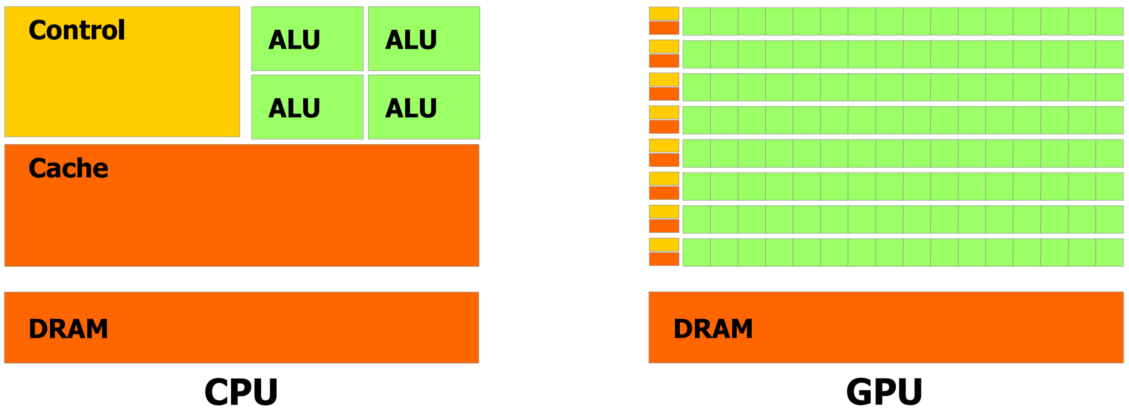


Figure 1.2 CPU and GPU architectural comparison

As we can see, the control and cache parts in the GPUs are undersized to exploit the computational parts, so the ALUs, due to the fact that GPUs and CPUs tasks are different and they give their best combined together. We can analyze in a deeper way the difference between the two electronic circuitries to have a better general overview and understanding:

- Latency intolerance vs latency tolerance
    - CPUs are low latency low throughput processors
    - GPUs are high latency high throughput processors
    - This differences better highlight the two tasks of CPUs and GPUs; the first have to be very responsive, so they have large caches to minimize the latency, while the latter have to compute a lot of parallel data concurrently, so they have a lot of ALUs to maximize the throughput

- Multithreaded cores vs SIMT (Single Instruction Multiple Threads) cores
    - In CPUs there are multiple tasks for multiple threads, while GPUs are based on SIMD (Single Instruction Multiple Data), so the same instruction operates on many different data
    - In CPUs there is a few number of heavy threads running on cores, while in GPUs there is an high number of lightweight threads running on cores
    - For the same chip size, GPUs can have a lot of additional ALUs thanks to the absence of branch prediction units, speculative units, out-of-order units and smaller cache sizes. GPUs do not need those units, but they have to be efficient in what they do: computation.

- Task parallelism vs data parallelism
    - In CPUs, different instructions operate on different tasks
    - In GPUs, the same instruction operates on different data

## 1.4 GPGPU

The term GPGPU (General Purpose computing on GPU) is the use of a GPU for high parallel tasks computation. So it can be considered as the exploit of the characteristics of a GPU, which traditionally was used to handle computations for only computer graphic applications, to perform computations which were usually handled by the CPU. In this way, the CPU is able to have less heavy work which takes a lot of time to be completed, giving it to the GPU which employs a lot less time to complete the same work. The combined use of a CPU with a GPU (or more than one GPU) has been proved to provide a lot of advantages in terms of performance and throughput. The use of GPUs as GPGPUs became a reality around 2001, with the advent of programmable shaders and floating point support on graphic processors. Problems which involved matrices and vectors (of different dimensions) were easy to translate to a GPU, and the scientific computing community decided to use GPUs to boost some operations which were computed by the CPUs. This change in analyzing the computational problems required to reformulate them in terms of graphic primitives, in such a way to allow the GPUs to handle them. This operation was required due to the fact that, at the beginning, the two major APIs for graphic processors were OpenGL and DirectX, which were not able to handle codes in different ways than graphic ones. These APIs were followed by CUDA, which allowed the programmers to ignore the underlying graphical concept to adopt an high performance computing concept, in such a way that modern GPGPUs can make use of the speed of a GPU without needing full and explicit conversion of the data in graphical form. GPGPUs are written through a framework, such as CUDA or OpenCL, which allows to the code running on a CPU to poll a GPU shader for return values.

GPUs aim is oriented for graphic purposes, so they are very limited in operations and programming; for such a reason, they are only effective for problems which can be solved using stream processes and the hardware has to be set and utilized in specific ways. This stream processing nature of GPUs was valid in the past, with older APIs (OpenGL and DirectX), as much as it is today with modern APIs (CUDA and OpenCL), despite the change in programming, i.e. it is no longer needed to map the computation into graphic primitives. GPUs use stream processing (so they are called also stream processors, which are processors that can operate in parallel by running one kernel on may records in a stream at once) because they can only process independent vertices and fragments, but they can do it with many of them in parallel, which become a winning way of work when is needed to process many vertices or fragments in the same way. A stream is a set of records which require similar computation processes, and, for their nature, they provide data parallelism. A kernel is a function which is applied to each element in the stream. To be better understandable, in GPUs vertices and fragments are the elements in streams and vertex and fragment shaders are the kernel which run on them. So, in GPGPUs is important to have an high arithmetic intensity (defined as the number of operations performed per word of memory transferred) to not limit the computational speedup due to the memory access latency. Ideal characteristics are then large data sets, high parallelism, minimal dependency between data elements.

Due to the evolution of GPUs into GPGPUs, so with different aims from graphical ones (GPUs are anyway used also for them), their area of work has expanded in the last years. In particular, GPUs and GPGPUs are used in:

- Automatic parallelization
- Computer clusters or a variant of a parallel computing
    - High performance computing clusters
    - Grid computing
    - Load balancing clusters
- Physical based simulation and physics engines
- Statistical physics
- Segmentation
- Level set method
- Ct reconstruction
- Fast Fourier transform
- GPU learning
    - Machine learning
    - Data mining
- K-nearest neighbor algorithm
- Fuzzy logic
- Tone mapping
- Audio signal processing
    - Audio and sound effects processing to use a GPU for DSP
    - Analog signal processing
    - Speech processing

- Digital image processing
- Video processing
    - Hardware accelerated video encoding and pre processing
    - Hardware accelerated video decoding and post processing
        - Motion compensation
        - Inverse discrete cosine transform
        - Variable length decoding
        - Inverse quantization
        - In-loop deblocking
        - Bitstream processing
        - Deinterlacing
        - Noise reduction
        - Edge enhancement
        - Color correction
- Global illumination
- Geometric computing
- Scientific computing
    - Monte Carlo simulation of light propagation
    - Weather forecasting
    - Climate research
    - Molecular modeling on GPU
    - Quantum mechanical physics
    - Astrophysics
- Bioinformatics
- Computational finance
- Medical imaging
- Clinical decision support system
- Computer vision
- Digital signal processing
- Control engineering
- Operations research
- Neural networks
- Database operations
- Computational Fluid Dynamics

- Cryptography and cryptoanalysis
- Performance modeling
    - Implementation of Message Digest Algorithm (MD6), Advanced Encryption Standard (AES), Data Encryption Standard (DES), Rivest-Shaver-Adleman (RSA), elliptic curve cryptography (ECC)
    - Password cracking
    - Cryptocurrency transactions processing
- Electronic design automation
- Antivirus software
- Intrusion detection
- Increase computing power for distributed computing projects

## 1.5 Stream processing

As mentioned in the previous sub-chapter, the GPUs (and so GPGPUs) use stream processing, which is the reason behind the fact that they are also called stream processors. Stream processing is a computer programming paradigm, which allows some applications to facilitate the exploit a limited form of parallel processing and structures programs in a way that allows high efficiency in computation and communication. These applications can use multiple and different computational units, e.g. the floating point unit on a GPU or on a FPGA (Field Programmable Gate Array), without explicitly managing allocation, synchronization, or communication among those units. The stream processing paradigm simplifies parallel software and hardware by limiting the parallel computation that can be performed. Given a sequence of data (called stream) of the same type, a series of operations (called kernel functions) is applied to each element in the stream. Kernel function are commonly pipelined, and optimal local on-chip memory reusage is attempted, in such a way to minimize the loss in bandwidth, associated with external memory interaction. Uniform streaming, in which one kernel function is applied to all elements in the stream, is commonly used. Since the kernel and stream abstractions are susceptible to data dependencies, compiler tools can fully automate and optimize on-chip management tasks. Stream processing hardware can use scoreboarding (a centralized method for dynamically scheduling a pipeline so that the instructions can execute out of order where there are no conflicts and the hardware is available), for example, to initiate a DMA (Direct Access Memory) when dependencies become known. The elimination of manual DMA management reduces software complexity, and an associated elimination for hardware cached I/O, reduces the data area expanse that has to be involved with service by specialized computational units such as ALUs (Arithmetic Logic Units). Stream processing consists of a tradeoff, driven by a data-centric model that works very well for traditional DSP or GPU-type applications (e.g. image, video, and digital signal processing) but that does not work with the same efficiency for general purpose processing with more randomized data access (e.g. databases). By sacrificing some flexibility in the model, the implications allow easier, faster and more efficient execution. Depending on the context, processor designed may be tuned for maximum efficiency or a tradeoff for flexibility. Stream processing is mostly suitable for applications that exhibit three application characteristics:

- Compute intensity, which is the number of arithmetic operations per I/O or global memory reference. In many signal processing applications today it is well over 50:1 and increasing with algorithmic complexity
- Data parallelism, which exists in a kernel if the same function is applied to all records of an input stream and a number of records can be processed simultaneously without waiting for results from previous records
- Data locality, which is a specific type of temporal locality common in signal and media processing applications where data is produced once, read once or twice later in the application, and never read again. Intermediate streams passed between kernels as well as intermediate data within kernel functions can capture this locality directly using the stream processing programming model

Some examples of records within streams are:

- In graphics, each record might be the vertex, normal, and color information for a triangle
- In image processing, each record might be a single pixel from an image
- In a video encoder, each record might be 256 pixels forming a macroblock of data
- In wireless signal processing, each record could be a sequence of samples received from an antenna

For each record we can only read from the input, perform some operations on it, and write to the output. It is allowed to have multiple inputs and multiple outputs, but it is never allowed to have a piece of memory that is both readable and writable.

In the stream programming model, applications are constructed by chaining multiple kernels together. For instance, implementing the graphics pipeline in the stream programming model involves writing a vertex program kernel, a triangle assembly kernel, a clipping kernel, and so on, and then connecting the output from one kernel into the input of the next kernel. In the following figure it is shown how the entire graphics pipeline maps onto the stream model. This model makes the communication between kernels explicit, taking advantage of the data locality between kernels inherent in the graphics pipeline.

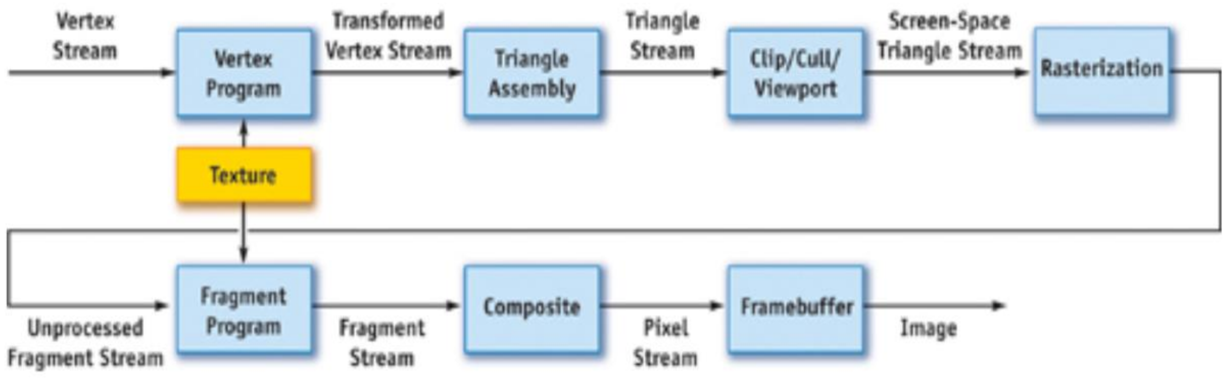


Figure 1.3 Mapping the stream model

The graphics pipeline is a good match for the stream model for several reasons. The graphics pipeline is traditionally structured as stages of computation connected by fixed data flow between the stages. This structure is analogous to the stream and kernel abstractions of the stream programming model. Data flow between stages in the graphics pipeline is highly localized, with data produced by a stage immediately consumed by the next stage; in the stream programming model, streams passed between kernels exhibit similar behavior. And the computation involved in each stage of the pipeline is typically uniform across different primitives, allowing these stages to be easily mapped to kernels. The stream model enables efficient computation in several ways. Most important, streams expose parallelism in the application. Because kernels operate on entire system, stream elements can be processed in parallel using data parallel hardware. Long streams with many elements allow this data level parallelism to be highly efficient. Within the processing of a single element, we can exploit instruction level parallelism. And because applications are constructed from multiple kernels, multiple kernels can be deeply pipelined and processed in parallel, using task-level parallelism. Dividing the application of interest into kernels allows a hardware implementation to specialize hardware for one or more kernels' execution. Special-purpose hardware, with its superior efficiency over programmable hardware, can thus be used appropriately in this programming model. Moreover, allowing only simple control flow in kernel execution (such as the data-parallel evaluation of a function on each input element) permits hardware implementations to devote most of their transistors to datapath hardware rather than control hardware. Efficient communication is also one of the primary goals of the stream programming model. First, off-chip (global) communication is more efficient when entire streams, rather than individual elements, are transferred to or from memory, because the fixed cost of initiating a transfer can be amortized over an entire stream rather than a single element. Next, structuring applications as chains of kernels allows the intermediate results between kernels to be kept on-chip and not transferred to and from memory. Efficient kernels attempts to keep their inputs and their intermediate computed data local within kernel execution units; therefore, data referenced within kernel execution do not go off-chip or across a chip to a data cache, as would typically happen in a CPU. And  finally, deep pipelining of execution allows hardware implementations to continue to do useful work while waiting for data to return from global memories. This high degree of latency tolerance allows hardware implementations to optimize for throughput rather than latency.

The stream programming model structures a program in a way that both exposes parallelism and permits efficient communication. Expressing programs in the stream model is only half the solution, however. High performance graphics hardware must effectively exploit the high arithmetic performance and the efficient computation exposed by the stream model. The first strep to build a high performance GPU is to map kernels in the graphics pipeline to independent functional units on a single chip. Each kernel is thus implemented on a separate area of the chip in an organization known as task parallel, which permits not only task level parallelism (because all kernels can be run simultaneously) but also hardware specialization of each functional unit to the given kernel. The task parallel organization also allows efficient communication between kernels: because the functional units implementing neighboring kernels in the graphics pipeline are adjacent on the chip, they communicate effectively without requiring global memory access. Within each stage of the graphics pipeline that maps to a processing unit on the chip, GPUs exploit the independence of each stream element by processing multiple data elements in parallel. The combination of task level and data level parallelism allows GPUs to profitably use dozens of functional units simultaneously. Inputs to the graphics pipeline must be processed by each kernel in sequence. Consequently, it may take thousands of cycles to complete the processing of a single element. If a high latency memory reference is required in processing any given element, the processing unit can simply work on other elements while the data is being fetched. The deep pipelines of modern GPUs, then, effectively tolerate high latency operations.

For many years, the kernels that make up the graphics pipeline were implemented in graphics hardware as fixed function units that offered little to no user programmability. In 2000, for the first time, GPUs allowed users the opportunity to program individual kernels in the graphics pipeline. Today’s GPUs feature high performance data parallel processors that implement two kernels in the graphics pipeline: a vertex program that allows users to run a program on each vertex that passes through the pipeline, and a fragment program that allows users to run a program on each fragment.

Both of these stages permit single precision floating point computation. Although these additions were primarily intended to provide users with more flexible shading and lighting calculations, their ability to sustain high computation rates in user specified programs with sufficient precision to address general purpose computing has effectively made them programmable stream processors, which is, processors that are attractive for a much wider variety of applications than simply graphics pipeline.

## 1.6 The future and challenges

The migration of GPUs into programmable stream processors reflects the culmination of several historical trends. The first trend is the ability to concentrate large amounts of computation on a single processor die. Equally important has been the ability and talent of GPU designers in effectively using these computation resources. The economies of scale that are associated with building tens of millions of processors per year have allowed the cost of a GPU to fall enough to make a GPU a standard part od today's desktop computer. And the addition of reasonably high-precision programmability to the pipeline has completed the transition from a hardwired, special purpose processor to a powerful programmable processor that can address a wide variety of tasks.

- Technology trends:

  Each new generation of hardware will present a challenge to GPU vendors. New transistors will be devoted to increased performance, in large part through greater amounts of parallelism, and to new functionality in the pipeline. We will also see these architectures evolve with changes in technology. Future architectures will increasingly use transistors to replace the need for communication. We can expect more aggressive caching techniques that not only alleviate off-chip communication but also mitigate the need for some on-chip communication. We will also see computation increasingly replace communication when appropriate. For example, the use of texture memory as a lookup table may be replaced by calculating the values in that lookup table dynamically. And instead of sending data to a distant on-chip computation resource and then sending the result back, we may simply replicate the resource and compute our result locally. In the tradeoff between communicate and recompute/cache, we will increasingly choose the latter. The increasing cost of communication will also influence the microarchitecture of future chips. Designers must now explicitly plan for the time required to send data across a chip; even local communication times are becoming significant in a timing budget.

- Power management:

  Ideas for how to use future GPU transistors must be tempered by the realities of their costs. Power management become a critical piece of today's GPU designs as each generation of hardware has increased its power demand. The future may hold more aggressive dynamic power management targeted at individual stages; increasing amounts of custom or power-aware design for power-hungry parts of the GPU; and more sophisticated cooling management for high-end GPUs. Technology trends indicate that the power demand will only continue to rise with future chip generations, so continued work in this area will remain an important challenge.

- Supporting more programmability and functionality:

  While the current generation of graphics hardware features substantially more programmability than previous generations, the general programmability of GPUs is still far from ideal. One step toward addressing this trend is to improve the functionality and flexibility within the two current programmable units (vertex and fragment). It is likely that we will see their instruction sets converge and add functionalities, and that their control flow capabilities will become more general as well. We may even see programmable hardware shared between these two stages in an effort to better utilize these resources. GPU architects will have to be mindful, however, that such improvements do not affect the GPU's performance in its core tasks. Another option will be expanding programmability to different units. Geometric primitives particularly benefit from programmability, so we may soon see programmable processing on surfaces, triangles, and pixels. As GPU vendors support more general pipelines and more complex and varied shader computation, many researchers have used the GPU to address tasks outside the bounds of the graphics pipeline. The general purpose computation on GPUs (GPGPU) community has successfully addressed problems in visual simulation, image processing, numerical methods, and databases with graphics hardware. We can expect that these efforts will grow in the future as GPUs continue to increase in performance and functionality. Historically, we have seen GPUs subsume functionality previously belonging to the CPU. Early consumer level graphics hardware could not perform geometry processing on the graphics processor; it was only a few years ago that the entire graphics pipeline could be fabricated on a single chip. Although since that time the primary increase in GPU functionality has been directed toward programmability within the graphics pipeline, we should not expect that GPU vendors have halted their efforts to identify more functions to integrate onto a GPU. In particular, today's games often require large amounts of computation in physics and artificial intelligence computations. Such computation may be attractive for future GPUs.

- GPU functionality subsumed by CPU (or vice versa):

  We can be confident that CPU vendors will not stand still as GPUs incorporate more processing power and more capability onto their future chips. The ever increasing number of transistors with each process generation may eventually lead to conflict between CPU and GPU manufacturers. The future may reserve us an environment with the CPU as the core of the newer computer systems, which could be eventually incorporate GPU or stream functionality on the CPU itself; or maybe it may reserve us an environment with a GPU as the core of the newer computer systems, which could be eventually be enhanced with CPU functionalities.

# Chapter two: CUDA and OpenCL

## 2.1 Introduction

CUDA and OpenCL are two different parallel computing platforms and APIs created by two different vendors, Nvidia and Khronos Group. They are the currently most used interfaces for GPUs programming and their main difference are:

- Proprietary vs open source
- Homogeneous vs heterogeneous
- Performance vs portability

We will have a deeper look into each of them.

## 2.2 CUDA

CUDA[2], which stands for Compute Unified Device Architecture, is a parallel computing platform and API developed by Nvidia in 2007. It allows the programmers to use CUDA compatible GPUs for GPGPU tasks. It is designed to work with high level programming languages, such as C, C++ and Python. This peculiarity gives the opportunity to have a larger number of users, due to the fact that the previous APIs such as OpenGL (Open Graphics Library) required a specified language to write the programs to be run on GPUs. In this way CUDA increases its accessibility. The CUDA platform is accessed by programmers via CUDA-accelerated libraries, compiler directives such as OpenACC (Open Accelerators) and extension to the high level programming languages mentioned before, so C, C++, Python and others. In particular, C and C++ programmers can use "CUDA C/C++" compiled to PTX (Parallel Thread Execution) with NVCC (Nvidia CUDA Compiler), an Nvidia's LLVM (Low Level Virtual Machine) compiler, used to develop a front end for the programming language and a back end for the ISA (Instruction Set Architecture). The platform supports also other interfaces, such as OpenCL and OpenGL. It also provides a low level API (called CUDA Driver) and an high level API (called CUDA Runtime). It works on all Nvidia's GPUs (GeForce, Quadro and Tesla, to quote the most famous ones). It runs on all standard OS (Operating Systems) such as Linux, Windows and MacOS. Actually, the last released stable CUDA toolkit is the 11.0.

[2] https://developer.nvidia.com/cuda-zone

Here below is a list of the main CUDA libraries:

- cuBLAS, CUDA basic linear algebra subroutine library
- CUDART, CUDA runtime library
- cuFFT, CUDA fast Fourier transform library
- cuRAND, CUDA random number generation library
- cuSOLVER, CUDA based collection of dense and sparse direct solvers
- cuSPARSE, CUDA sparse matrix library
- NPP, Nvidia performance primitives library
- nvGRAPH, Nvidia graph analytics library
- NVML, Nvidia management library
- NVRTC, Nvidia runtime compilation library

Using CUDA has some advantages:

- Scattered reads, i.e. the code can read from arbitrary addresses in memory
- Unified virtual memory
- Unified memory
- Shared memory, which is shared among threads
- Fast transfers from and to the GPU
- Full support for integer and bitwise operation

Unfortunately, CUDA has also some disadvantages:

- CUDA source code is now processed according to C++ syntax rules, so older versions based on simple C could fail to compile or could have a not wanted behavior
- Interoperability with rendering languages is limited; for example, OpenGL can have access to CUDA memory but not the opposite
- Transfers between CPU (host) and GPU (device) memory could not be so efficient in terms of performance due to system bus bandwidth and latency
- Threads should run in groups of at least 32 of them to better exploit the GPU's performance
- There are nor emulators or fallback functionality for newer revisions
- CUDA-enabled GPUs are available only from Nvidia

The last point is the worst, because it means that Nvidia obliges users to buy their GPUs to be able to use CUDA. Through the years some attempts were made to run CUDA on other GPUs than Nvidia's ones, such as:

- Project Coriander, which converts CUDA C++11 to OpenCL 1.2 C
- CU2CL, which converts CUDA 3.2 C++ to OpenCL
- GPUOpen HIP, which converts CUDA from 4 to 11 C++ to AMD cards format

In the following, a list of some applications that use CUDA:

- Accelerated rendering of 3D graphics
- Accelerated interconversion of video file formats
- Accelerated encryption, decryption and compression
- Bioinformatics
- Distributed calculations
- Medical analysis simulations
- Physical simulations
- Neural network training (machine learning field)
- Face recognition
- Distributed computing
- Molecular dynamics
- Mining (crypto currency field)

A CUDA program consists of one or more phases which are executed on either the host or a device, as it could be the GPU. The phases that exhibit a lot of data parallelism are implemented in the device code, while the phases that exhibit little or no data parallelism are implemented in the host code. A CUDA program is a unified source code encompassing both host and device code. The NVIDIA_C compiler (nvcc) separates the two during the compilation process. The host code is straight ANSI C code; it is further compiled with the host's standard C compilers and runs on an ordinary CPU process. The device code is written using ANSI C extended with keywords for labeling data-parallel functions, so kernels, and their associated data structures. The device code is typically further compiled by the nvcc and executed on a GPU device. In situations where no device is available or the kernel is more appropriately executed on a CPU, one can also choose to execute kernels on a CPU using the emulation features in CUDA SDK (Software Development Kit). The steps in executing CUDA programs are:

- Device initialization
- Device memory allocation
- Copies data to device memory
- Executes kernel (calling _global_function)
- Copies data from device memory (retrieve results)

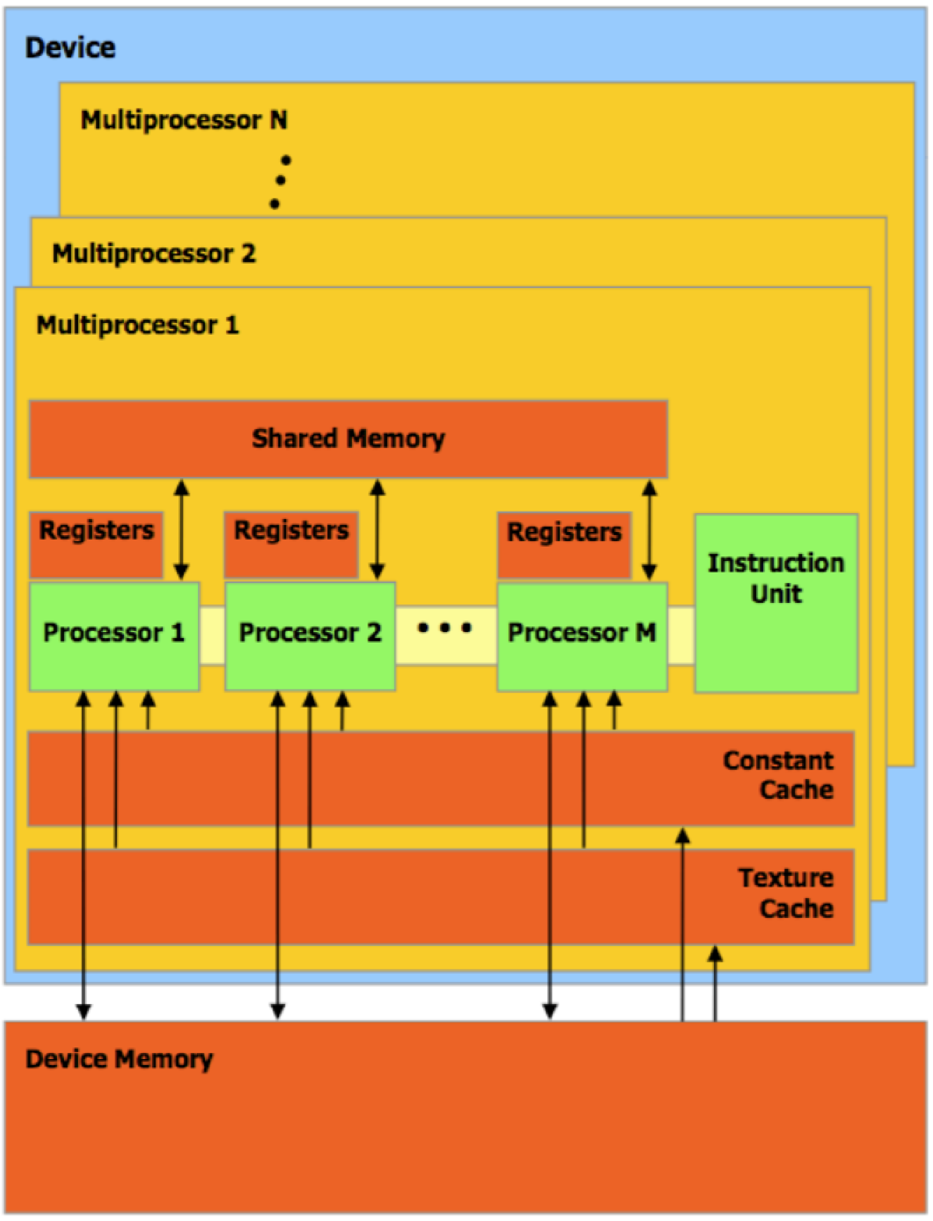


Fig. 2.1 CUDA Device model

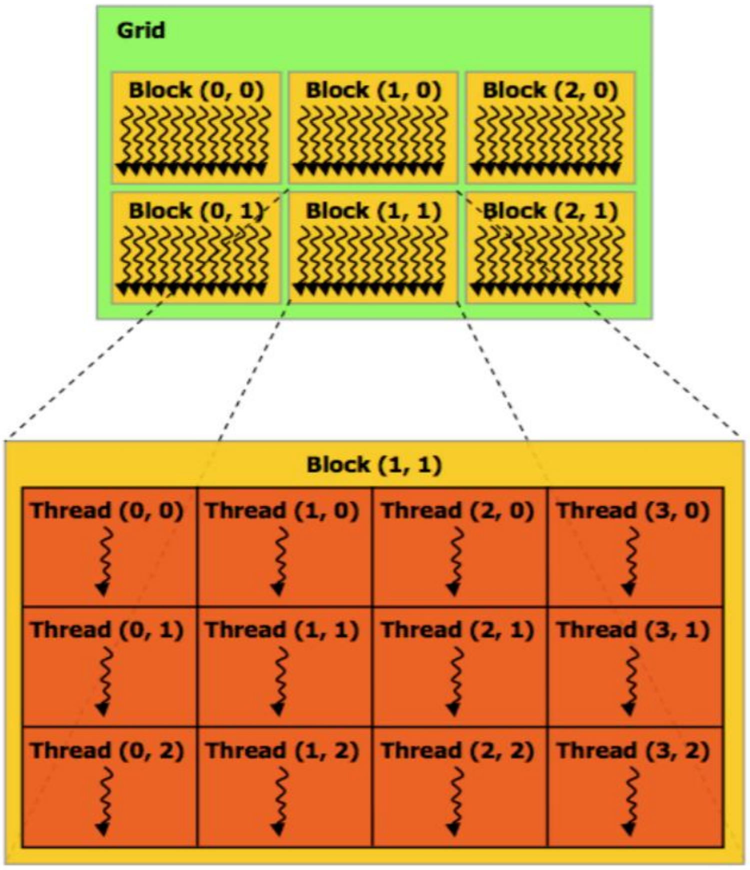


Fig. 2.2 CUDA Execution model

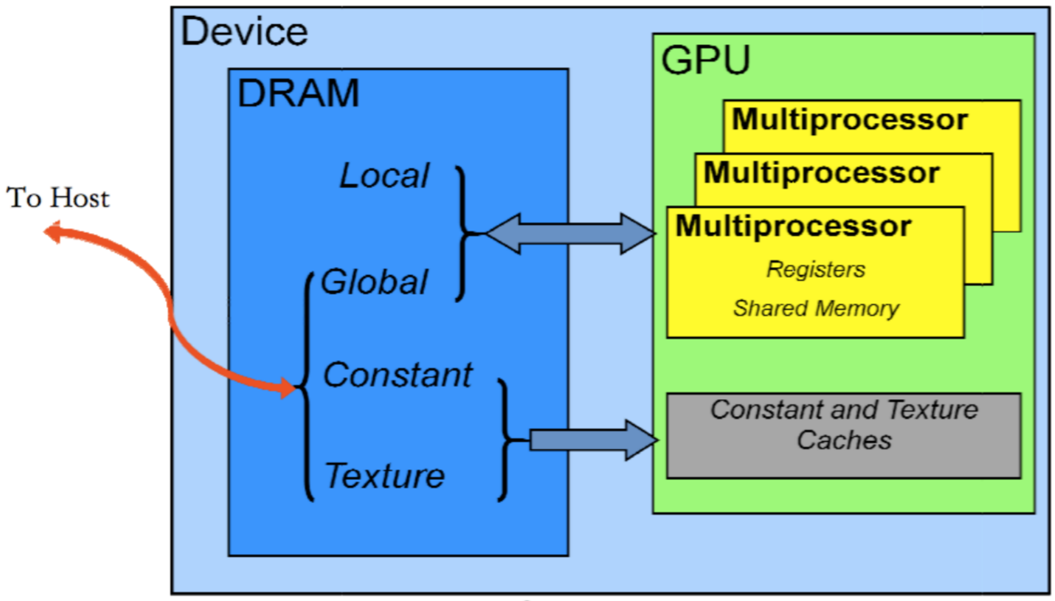


Fig. 2.3 CUDA Memory model

### 2.2.1 CUDA development model

A CUDA application consists of host program and CUDA device program. The host program activates computation kernels, which are data parallel routine programs, in the device program, and they are executed on the device for multiple data items in parallel by device threads. Computation kernels are written in C for CUDA or PTX (a low level parallel thread execution virtual machine and instruction set architecture used in Nvidia's CUDA programming environment); the first adds language extensions and built-in functions for device programming. It is also present some support for other kernel programming languages. Then, the host program accesses the device with either C runtime for CUDA or CUDA Driver API:

- C runtime interface is higher level and less verbose to use than the Driver API
- With C runtime computation kernels can be invoked from the host program with convenient CUDA-specific invocation syntax
- The Driver API provides more finer grained control
- Bindings to other programming languages can be built on top of either API

Finally, we can say that device and host code can be mixed or written on separate source files, that graphics interoperability is provided with OpenGL and Direct3D and that Nvidia provides also OpenCL interface for CUDA.

### 2.2.2 CUDA toolchain

The device program is compiled by the CUDA SDK-provided nvcc compiler, which emits CUDA PTX assembly or device-specific binary code for the device code. PTX is an intermediate code specified in CUDA that is further compiled and translated by the device driver to actual device machine code. The device program files can be compiled separately or mixed with host code if CUDA SDK-provided nvcc compiler is used; moreover, the latter is also required if CUDA custom kernel invocation syntax is used. Finally, separate compilation can output C host code for integrating with the host toolchain. In the following image, we can see a generic scheme of the CUDA system architecture, in which we can recognize some of the features we have talk about previously.

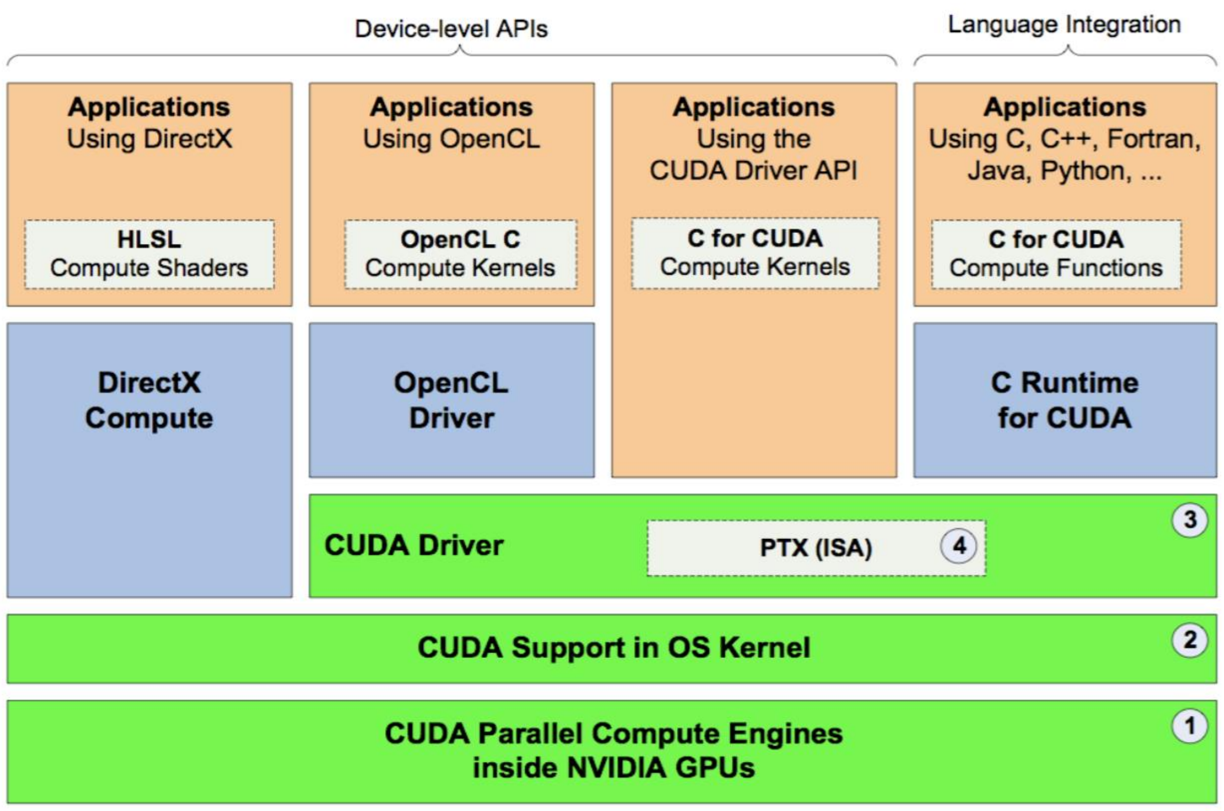

Device-level APIs
Language Integration
Applications Using DirectX
HLSL Compute Shaders
Applications Using OpenCL
OpenCL C Compute Kernels
Applications Using the CUDA Driver API
C for CUDA Compute Kernels
Applications Using C, C++, Fortran, Java, Python, ...
C for CUDA Compute Functions
DirectX Compute
OpenCL Driver
C Runtime for CUDA
CUDA Driver
PTX (ISA)
4
3
CUDA Support in OS Kernel
2
CUDA Parallel Compute Engines inside NVIDIA GPUs
1

Fig 2.4 CUDA System Architecture

### 2.2.3 C for CUDA kernel programming

C for CUDA kernel programming is based on C programming language with extensions and restrictions (to be noted that the C language standard version used as base is not defined). Here below a list of the extensions and restrictions included:

- Extensions:
  - Built-in vector data types, but no built-in operators or math functions for them
  - Function and variable type qualifiers
  - Built-in variables for accessing thread indices
  - Intrinsic floating point, integer and fast math functions
  - Texture functions
  - Memory fence and synchronization functions
  - Voting functions (from CC 1.2)
  - Atomic functions (from CC 1.1)

    - Limited C++ language features support: function and operator overloading, default parameters, namespaces, function templates
- Restrictions:
    - No recursion support, static variables, variable number of arguments or taking pointer of device functions
    - No dynamic memory allocation
    - Access to full set of standard C library (e.g. *stdio*) only in emulation mode

- Numerical accuracy:
    - Accuracy and deviations from IEEE-754[3] are specified
    - For deviating operations compliant, but slower software versions are provided

## 2.3 OpenCL

OpenCL[4], which stands for Open Computing Language, is a framework for writing programs which are executed in an heterogenous environment, e.g. composed by a CPU and one or more GPUs. It specifies both the programming languages to be used to setup the devices and the APIs to have platform control and to execute programs on devices. OpenCL was born in 2009 from the non-profit technology consortium Khronos Group. It was created as the open source opponent of Nvidia's CUDA, and today is its main competitor, not only from a moral point of view (proprietary vs open source software) but also from the market point of view. Its capability to run on different vendors' GPUs, so to not be obliged to use an Nvidia's one, is one of the main reasons of its success. Computing systems are seen by OpenCL as a collection of *devices* (CPUs or GPUs) attached to an *host* (CPU). Programs are written in a C-like language and the functions to be executed are called *kernels*. A single device usually consists of several *compute units* which, in turn, consists of different PEs (*Processing Elements*). So, a single kernel execution can run all the PEs or a substantial number of them. OpenCL also provides an API which allows host's programs to launch kernels on the device and to manage their memory. A key point of OpenCL is its portability not only on devices (GPUs from different vendors) but also on hosts (CPUs from different vendors and with different characteristics), so programs in OpenCL are meant to be compiled in run-time (known also as JIT, or Just-In-Time execution). OpenCL gives also the opportunity to use SPIR (Standard Portable

[3] https://standards.ieee.org/standard/60559-2020.html
[4] https://www.khronos.org/opencl/

Intermediate Representation, but actually named SPIR-V, which is natively incorporated in OpenCL and not more an external extension which used LLVM compiler), useful to programming in other languages and to protect the kernel source; moreover, now OpenCL supports also SYCL, an high level programming model used to improve productivity.

The C-like programming language used to write kernels is called OpenCL C and is based on C99 (which is the informal name of ISO/IEC 9899:1999, an old version of the C language) with some changes to better run on devices. OpenCL functions are marked *_kernel*, instead of having a main function as in C, to indicate the starting point of the devices which have to be called by the host program. Pointers to the memory hierarchy are marked as *_global, _constant, _local, _private,* as seen before. Procedure pointers, bit fields and variable-sized arrays are omitted; moreover, recursion is not allowed. The C standard library is modified with a custom library to better exploit arithmetical programming. Currently, OpenCL framework is updated at version 3.0, with a propension to C++-like programming language respect to C-like one. OpenCL is composed by a set of headers and a shared library which are loaded at run time. An additional ICD (Installable Client Driver) has to be installed for every different vendor's card which has to be used, So, there is an ICD for Nvidia's cards, an ICD for AMD's cards and so on, in order to support different cards from different vendors. To make it possible, OpenCL headers are used by the consumer application and vendors have to update their drivers to implement OpenCL calls.

Here below, a list of different implementations from different vendors to allow the OpenCL usage on their cards:

- MESA Gallium Compute, also known as MESA Clover, implementation for AMD and Nvidia
- BEIGNET, implementation for Ivy Bridge, Skylake and Android
- NEO, implementation for Intel Ice Lake and Tiger Lake. This replaces the Intel's implementation BEIGNET
- ROCm, implementation for AMDs CPUs and APUs (Accelerated Processing Unit) and Intel CPUs from generation 7th and newer
- POCL, portable implementation for some CPUs and GPUs using CUDA and HSA (Heterogeneous System Architecture, a cross vendor set of specifications for allowing integration of CPUs and GPUs on the same bus). It runs also on Mac OS
- Shamrock, porting implementation of MESA Clover for ARM
- FreeOCL, an implementation with an external compiler for a more stable platform
- MOCL, implementation based on POCL for Intel Xeon Phi accelerators

As mentioned before, the most important feature of OpenCL is its portability, reached through its abstraction of memory and the execution model, so it is possible to run any OpenCL kernel on any implementation which is compliant with the required characteristics. As a trade off, to have such a portability some waivers have to be done. So performance is not comparable with ad hoc solutions, such as CUDA. Acceptable levels of performance across different devices have been reached, otherwise OpenCL would not be taken into account as one of the main platforms for GPU's programming. The main reasons between CUDA and OpenCL performance are due to differences in programming model, different optimizations on native kernels, architecture related differences, and compiler differences.

In a document of the Delft University of Technology of 2011[5], three researchers concluded that "there is no reason for OpenCL to obtain worse performance than CUDA under a fair comparison. Several benchmarks also show the interesting performance gaps. The reasons behind the gaps are analyzed thoroughly and they can all be essentially related to various behaviors of programmers, compilers and users. We also port all the real-world benchmarks to other platforms with minor modifications to show OpenCL's potential for portability. Since it has been shown in this paper that OpenCL is a good alternative to CUDA, we would like to develop an auto-tuner to adapt general-purpose OpenCL programs to all available specific platforms to fully exploit the hardware".

In another document of D-Wave Systems Inc.[6], another team of researchers say that "the changing performance for different problem sizes are due to differences in data structure sizes and their placement in GPU memory. GPU performance is very dependent on these issues. However, these effects are specific to the algorithm used, so here we focus on the performance difference between CUDA and OpenCL. For all problem sizes, both the kernel and the end-to-end times show considerable difference in favor of CUDA. The OpenCL kernel's performance is between about 13% and 63% slower, and the end-to-end time is between about 16% and 67% slower. As expected, the kernel and end-to-end running times approach each other in value with bigger problem sizes, because the kernel time's contribution to the total running time increases."

So it is true that OpenCL has worse performance results compared to CUDA, but it is also true that is a matter of the youthfulness of the framework and the inexperience of the developers in that environment.

An OpenCL program is similar to a dynamic library, and an OpenCL kernel is similar to an exported function from the dynamic library. Applications directly call the functions exported by a dynamic library from their code; however, they can not call an OpenCL kernel directly to a command-queue created for a device. The kernel is executed asynchronously with the application code running on the host CPU. The OpenCL specification is defined in four parts, called models, which are:

[5] Jianbin Fang, Ana Lucia Varbanescu and Henk Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL", Parallel and Distributed Systems Group, Delft University of Technology (Delft, Netherlands), September 2011
[6] K. Karimi, N. G. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," May 2010

- Platform model:
    - It specifies that there is one processor coordinating execution (the host) and one or more processors capable of executing OpenCL code (the devices). It defines an abstract hardware model that is used by programmers when writing OpenCL functions (the kernels) that execute on the devices.

- Execution model:
    - It defines how the OpenCL environment is configured on the host and how kernels (the code for a work-item; basically a C function) are executed on the device. This includes setting up an OpenCL context (the environment within which work-items (the basic unit of work on an OpenCL device) executes, includes devices and their memories and command queues) on the host, providing mechanism for interaction between the host and any device, and defining a concurrency model used for kernel execution on devices. OpenCL application runs on a host which submits work to the compute devices.

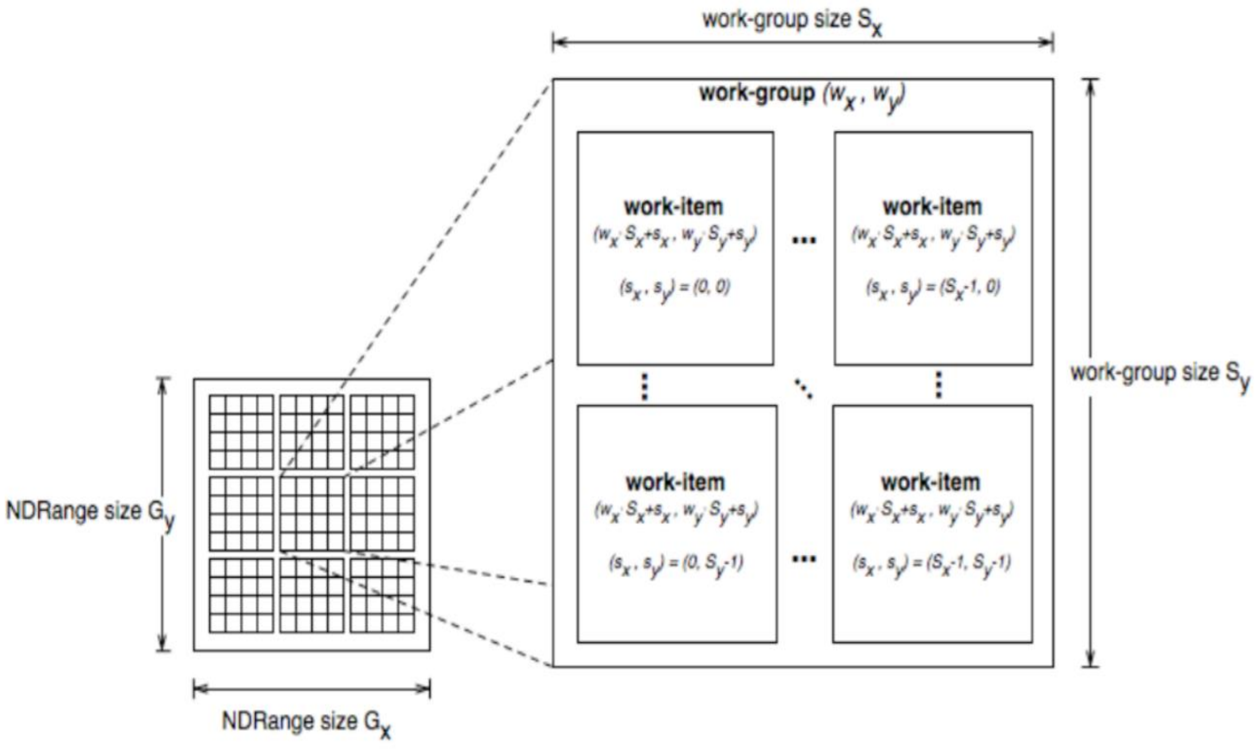

work-group size $S_x$
work-group $(w_x, w_y)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, 0)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, 0)$
work-group size $S_y$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (0, S_y-1)$
work-item
$(w_x \cdot S_x+s_x, w_y \cdot S_y+s_y)$
$(s_x, s_y) = (S_x-1, S_y-1)$
NDRange size $G_y$
NDRange size $G_x$

Fig 2.5 OpenCL Execution model

- Memory model:
    - It defines the abstract memory hierarchy that kernels use, regardless of the actual underlying memory architecture. The memory model closely resembles current GPU memory hierarchies, although this has not limited adoptability by other accelerators.
        - Global memory, shared by all PEs and with an high latency, marked as *_global*
        - Read only memory, writable only by the host and with a low latency, marked as *_constant*
        - Local memory, shared by only a group of PEs, marked as *_local*
        - Per element private memory, the elements' registers, marked as *_private*

      It is not compulsory to implement all the hierarchy for all situations; in fact, it is possible to use only some levels of the hierarchy itself. Moreover, devices are not obliged to share their memory with the host but, in the case in which that happen, the host API provides handles to make the transfers from and to host and devices possible.

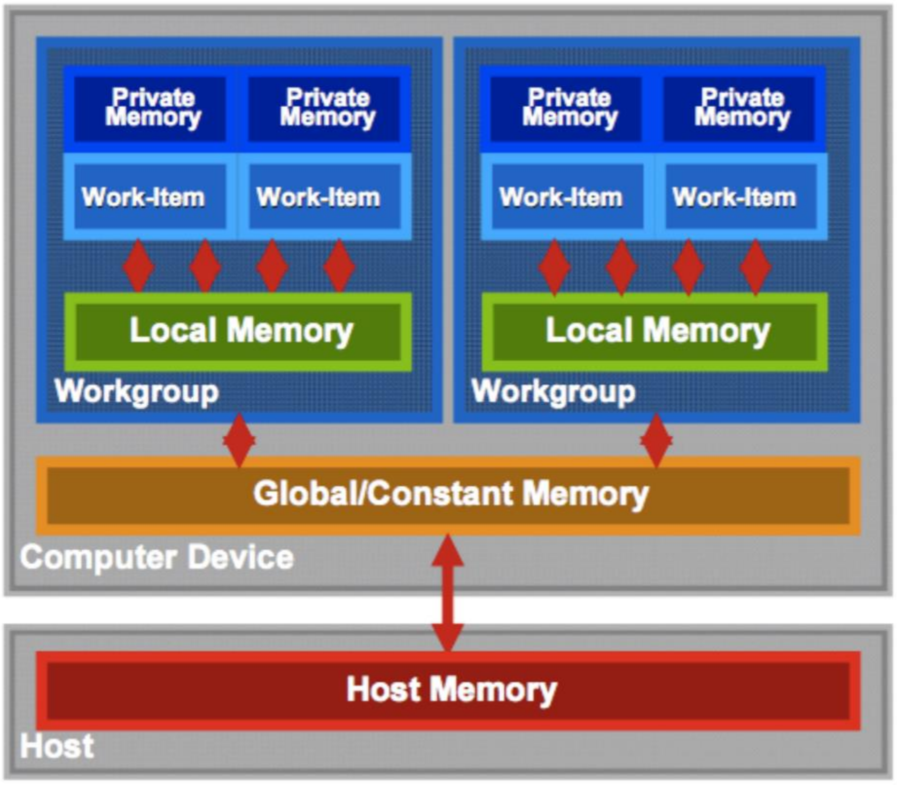


Fig 2.6 OpenCL Memory model

- Programming model:
    - It defines how the concurrency model is mapped to physical hardware. OpenCL uses Dynamic/Runtime compilation model:
        - The code is compiled to an IR (Intermediate Representation), usually an assembler or a virtual machine, known as offline compilation
        - The IR is compiled to a machine code for execution; this step is much shorter and it is known as online compilation

The steps in executing OpenCL programs are listed below:

- Query host for OpenCL devices
- Create a context to associate OpenCL devices
- Create programs for execution on one or more associated devices
- Select kernels to execute from the programs
- Create memory objects accessible from the host and/or the device
- Copy memory data to the device as needed
- Provide kernels to command queue for execution
- Copy results from the device to the host

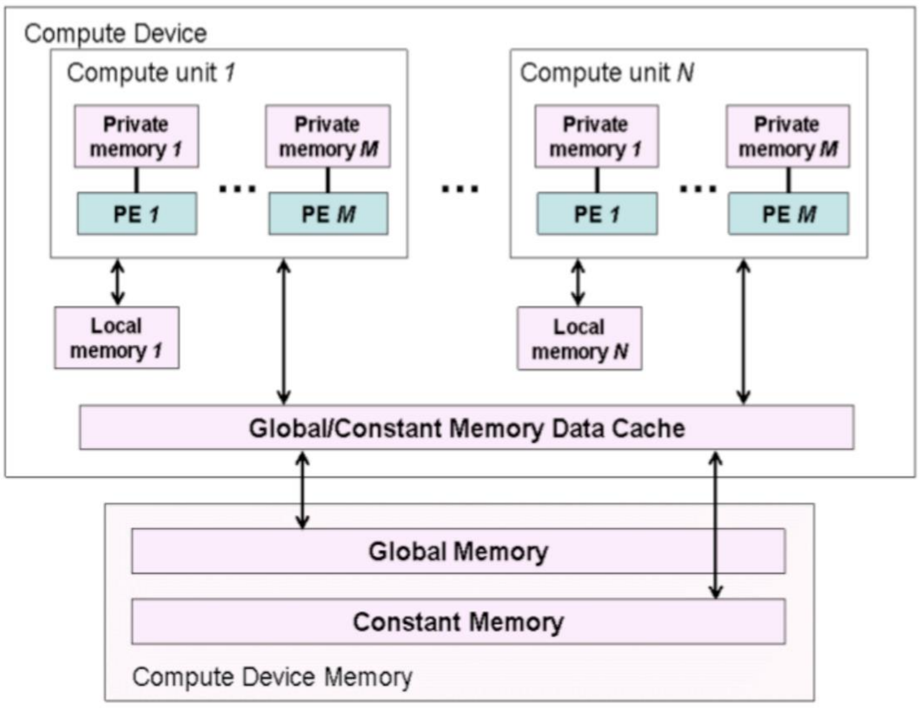


Fig 2.7 OpenCL Device model

### 2.3.1 OpenCL development model

An OpenCL application consists of host program and OpenCL program to be executed on the computation device; the host program activates computation kernels, which are data parallel routine programs, in the device program, and they are executed on the device for multiple data items in parallel by device processing elements (also task parallel and hybrid models are supported). OpenCL kernels are written with the OpenCL C programming language:

- OpenCL C is based on C99 with extensions and limitations
- In addition the language, OpenCL specifies library of built-in functions
- Implementations can also provide other means to write kernels

The host program controls the device by using the OpenCL C API (bindings to other host programming languages can be built on top of the C API) and graphics interoperability is provided with OpenGL.

### 2.3.2 OpenCL toolchain

An OpenCL implementation must provide a compiler from OpenCL C to supported device executable code; that compiler must support standard set of OpenCL defined options. The kernels can be compiled either online (runtime) or offline (build time). In online compilation the host program provides to the OpenCL API a compiled OpenCL C source text. Runtime compilation is more flexible for the final application, but may be problematic in some cases:

- Compilation errors need to be extracted through the OpenCL API at development time
- The kernel source code is included in the application binaries

Finally, the host program is compiled with the default host toolchain and OpenCL is used through its C API.

### 2.3.3 OpenCL C kernel programming

OpenCL C kernel programming is based on the C programming language with extensions and restrictions (it is based on the C99 version of the language standard). Here below a resuming list of the main extensions and restrictions:

- Extensions:
    - Built-in first-class vector data types with literal syntax, operators and functions
    - Explicit data conversions
    - Address space, function and attribute qualifiers
    - OpenCL-specific #*pragma* directives
    - Built-in functions for accessing work item indices
    - Built-in math, integer, relational and vector functions
    - Image read and write functions
    - Memory fence and synchronization functions
    - Asynchronous memory copying and prefetch functions
    - Optional extensions: e.g. atomic operations, etc.

- Restrictions:
    - No recursion, pointer to pointer arguments to kernels, variable number of arguments of pointers to functions
    - No dynamic memory allocation
    - No double-precision floating point support by default
    - Most C99 standard headers and libraries cannot be used
    - *extern*, *static*, *auto* and *register* storage-class specifiers are not supported
    - C99 variable length arrays are not supported
    - Writing to arrays or *struct* members with element size less than 32 bits is not supported by default
    - Many restrictions can be addressed by extensions, e.g. double-precision support, byte addressing, etc.

- Numerical accuracy:
    - Accuracy and deviations from IEEE-754[7] are specified
    - Some additional requirements specified beyond C99 TC2[8]

[7] https://standards.ieee.org/standard/60559-2020.html

[8] ISO/IEC 9899:TC2, http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf

# Chapter three: CUDA vs OpenCL

## 3.1 Introduction

In this chapter, CUDA and OpenCL platforms will be analyzed to better highlight their differences and to better understand in which case it is better to use one or the other. Before starting the analysis on the main differences between the two APIs, let's see some resuming table to better highlight the key points of the two platforms.

| | CUDA | OpenCL |
|---|---|---|
| **What it is** | HW architecture, ISA, programming language, API, SDK and tools | Open API and language specification |
| **Proprietary or open technology** | Proprietary | Open and royalty-free |
| **When introduced** | Q4 2006 | Q4 2008 |
| **SDK vendor** | Nvidia | Implementation vendors |
| **Free SDK** | Yes | Depends on vendor |
| **Multiple implementation vendors** | No, just Nvidia | Yes: Apple, Nvidia, AMD, IBM |
| **Multiple OS support** | Yes: Windows, Linux, Mac OS X; 32 and 64-bit | Depends on vendor |
| **Heterogeneous device support** | No, just Nvidia GPUs | Yes |
| **Embedded profile available** | No | Yes |

Table 3.1 Main differences between CUDA and OpenCL

| | CUDA | OpenCL |
|---|---|---|
| **Custom toolchain needed for host program** | Yes, if mixed device/host code or custom kernel invocation syntax is used | No |
| **Support for using platform default toolchain for the host program** | Yes | Yes, the only option |
| **Run-time device program compilation support** | Yes, from PTX (only with the Driver API) | Yes, from OpenCL C source text |

Table 3.2 Toolchain comparison between CUDA and OpenCL

| | CUDA | OpenCL |
|---|---|---|
| **Explicit host and device code separation** | Yes *) | Yes |
| **Custom kernel programming language** | Yes | Yes |
| **Multiple computation kernel programming languages** | Yes | Only OpenCL C or vendor-specific language(s) |
| **Data parallel kernels support** | Yes, the default model | Yes |
| **Task parallel kernels support** | No, at least not efficiently | Yes |
| **Device program intermediate language specified** | Yes, PTX | Implementation specific or no intermediate language used |
| **Multiple programming interfaces** | Yes, *including* OpenCL | Only the specified C API with possible vendor extensions |
| **Deep host and device program integration support** | Yes, with very efficient syntax | No, only separate compilation and kernel invocation with API calls |
| **Graphics interoperability support** | Yes, with OpenGL and Direct3D | Yes, with OpenGL |

Table 3.3 Development model comparison between CUDA and OpenCL

| | CUDA | OpenCL |
|---|---|---|
| **Base language version defined** | No, just "based on C" and some C++ features are supported | Yes, C99 |
| **Access to work-item indices** | Through built-in variables | Through built-in functions |
| **Address space qualification needed for kernel pointer arguments** | No, defaults to global memory | Yes |
| **First-class built-in vector types** | Just vector types defined, no operators or functions | Yes: vector types, literals, built-in operators and functions |
| **Voting functions** | Yes (CC 1.2 or greater) | No |
| **Atomic functions** | Yes (CC 1.1 or greater) | Only as extension |
| **Asynchronous memory copying and prefetch functions** | No | Yes |
| **Support for C++ language features** | Yes: limited, but useful set of features supported | No |

Table 3.4 Kernel programming differences between CUDA and OpenCL

## 3.2 Work environment

Currently, CUDA is the standard platform for GPU programming; it is a largely tested and working environment which has been used and tried-out for years. Programmers are used to write through that platform and CUDA was the very first alternative to programming GPU in a more comfortable way. Moreover, it works well. So, it is clear the reason behind its success. The only problem is that it sacrifice portability (CUDA works only on Nvidia GPUs) to better exploit performances (and obviously it is also a matter of money for Nvidia).

OpenCL was released some years after CUDA, and at the time the market was already used to work with the latter to writing GPUs' programs. The main difference is that OpenCL was presented as an open source alternative with a great portability in such a way that every GPU programmer, not only the ones using Nvidia cards, could program in an efficient and more comfortable way. In fact, OpenCL shares the main ways of programming of CUDA, so it keeps its easiness with in addition the major portability. The other side of the coin is the lower performance compared to CUDA; in fact, the portability does not allow to have specific characteristics to have a more precise environment based only on one vendor.

To validate the fact that OpenCL shares many things with CUDA, we can analyze the thread hierarchy of both of them:

- In CUDA, the hierarchy goes in an ascending order through *thread, warp, thread block, grid*
- In OpenCL, the hierarchy goes in an ascending order through *work-item, work-group, NDRange*

The above items have been written in order, to show their correlation. The second item is a collection of a group of the first items, the third item is a collection of a group of the second items and so on, for both CUDA and OpenCL. Grid and NDRange are the macroblocks of all threads or work-items which are launched during a kernel execution. In OpenCL the correspondence of a CUDA warp is absent, due to the fact that it does not run on a specific vendor's card; however, work-items work as a group, in such a way that we can consider that warps are only theoretically absent, but not in practice. It means that they work as they formed a warp without actually doing it. So, all the items work in a very similar way, with only name differences. Considering that, it is important to highlight that it is a priority considering on which card OpenCL has to run to better exploit the performance of the software accordingly to the specific vendor's system (for example, with Nvidia we have warps and with AMD we have wavefronts; both have to be taken into account to have the better possible program).

CUDA can be used in two different ways, via the runtime API, which provides a C-like set of routines and extensions, and via the driver API, which provides lower level control over the hardware but requires more code and programming effort. Both OpenCL and CUDA define kernel as the code which runs on the GPU (or in a more general way on a device, so not on the host). Setting up the device for kernel execution differs substantially between CUDA and OpenCL. Their APIs for context creation and data copying are different, and different conventions are followed for mapping the kernel onto the device's processing elements. These differences could affect the length of time needed to code and debug a device application. OpenCL aims at reaching a portable language for GPU programming, capable of targeting dissimilar parallel processing devices. Unlike a CUDA kernel, an OpenCL one can be compiled at runtime, which would add to an OpenCL's running time. On the other hand, just-in-time compilation may allow the compiler to generate code that makes better use of the target GPU. CUDA, however, is developed by the same company that develops the hardware on which it executes, so it is reasonable to expect it to better match the computing characteristics of the GPU, offering more access to features and better performance.

The OpenCL API is a C with a wrapper API that is defined in terms of the C API. There are third-party bindings for many languages, including Java, Python and .NET. The code that executes on an OpenCL device, which in general is not the same device as the host CPU, is written in the OpenCL C language, which is a restricted version of the C99 language with extensions appropriate for executing data-parallel code on a variety of heterogeneous systems. CUDA encourages the use of scalar code in kernels. While this works in OpenCL as well, depending on the desired target architecture, it may be more efficient to write programs operating on OpenCL's vector types, such as float, as opposed to pure scalar types. This is useful for both, for example, AMD CPUs and GPUs, which can operate efficiently on vector types. OpenCL also provides flexible swizzle/broadcast primitives for efficient creation and rearrangement of vector types. CUDA does not provide rich facilities for task parallelism, and so it may be beneficial to think about how to take advantage of OpenCL's task parallelism as you port your application

OpenCL shares a range of core ideas with CUDA: they have similar platform models, memory models, execution models, and programming models. They share the user's point of view, such as that to a programmer the computing system consists of a host (a traditional CPU) and one or more devices that are massively parallel processors equipped with a large number of arithmetic execution units. There also exists a mapping between CUDA and OpenCL in memory and execution terms. Additionally, their syntax for various keywords and built-in functions are fairly similar to each other.

Considering that CUDA and OpenCL shares many concepts (also some of the fundamental ones), the best way to highlight their differences is to compare their performance. There has been a fair amount of work on performance comparison of programming models for multi-core/many-core processors.

Rick Weber et al.[9] presented a collection of Quantum Monte Carlo algorithms implemented in CUDA, OpenCL, Brook+, C++, and VHDL. They gave systematic comparison of several application accelerators on performance, design methodology, platform, and architectures. Their result show that OpenCL provides application portability between multi-core processors and GPUs, but may incur a loss in performance.

Rob van Nieuwpoort et al.[10] explained how to implement and optimize signal-processing applications on multi-core CPUs and many-core architectures. They used correlation (a streaming, possibly real-time, and I/O intensive application) as a running example, investigating the aspects of performance, power efficiency, and programmability. This study includes an interesting analysis of OpenCL: the problem of performance portability is not fully solved by OpenCL and so programmers have to take more architectural details into consideration.

In another work[11], the authors compared programming features, platform, device portability, and performance of GPU APIs for cloth modeling. Implementations in GLSL (OpenGL Shading Language, an high level shading language with a syntax based on the C programming language), CUDA and OpenCL are given. They conclude that OpenCL and CUDA have more flexible programming options for general computations than GLSL. However, GLSL remains better for interoperability with a graphics API.

In one more work[12], a comparison between two GPGPU programming approaches (CUDA and OpenGL) is given using a weighted Jacobi iterative solver for the bidomain equations. The CUDA approach using texture memory is shown to be faster than the OpenGL version.

Kamran Karimi et al.[13], compared the performance of CUDA and OpenCL using complex, near-identical kernels. They showed that there are minimal modifications involved when converting a CUDA kernel to an OpenCL kernel. Their performance experiments measure and compare data transfer time to and from the GPU, kernel execution time, and end-to-end application execution time for both CUDA and OpenCL. Only one application or algorithm is used in all the work mentioned above.

[9] R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, “Comparing Hardware Accelerators in Scientific Applications: A Case Study,” IEEE Transactions on Parallel and Distributed Systems, vol. 22, pp. 58–68, January 2011

[10] R. van Nieuwpoort and J. Romein, “Correlating radio astronomy signals with Many-Core hardware,” International Journal of Parallel Programming, vol. 39, pp. 88–114, Feb. 2011

[11] T. I. Vassilev, “Comparison of several parallel API for cloth modelling on modern GPUs,” in Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies, CompSysTech ’10, (New York, NY, USA), pp. 131–136, ACM, 2010

[12] R. Amorim, G. Haase, M. Liebmann, and R. Weber dos Santos, “Comparing CUDA and OpenGL implementations for a Jacobi iteration,” pp. 22–32, June 2009

[13] K. Karimi, N. G. Dickson, and F. Hamze, “A Performance Comparison of CUDA and OpenCL,” May 2010

Ping Du et al.[14] evaluated many aspects of adopting OpenCL as a performance-portable method for GPGPU application development. The triangular solver (TRSM) and matrix multiplication (GEMM) have been selected for implementation in OpenCL. Their experimental results show that nearly 50% of peak performance could be obtained in GEMM on both NVIDIA Tesla C2050 and ATI Radeon 5870 in OpenCL. Their results also show that goof performance can be achieved when architectural specifics are taken into account in the algorithm design.

In another work[15], the authors quantitatively evaluated the performance of CUDA and OpenCL programs developed with almost the same computations. The main reason leading to these performance differences are investigated for applications including matrix multiplication from CUDA SDK and CP, MRI-Q, MRI-HD from the Parboil benchmark suite. Their results show that if the kernels are properly optimized, the performance of OpenCL programs is comparable with their CUDA counterparts. They also showed that the compiler options of the OpenCL C compiler and the execution configuration parameters have to be tuned for each GPU to obtain its best performance.

The majority of the above quoted works have used very few applications to compare existing programming models. A different work[16] observed a large set of different applications to show the performance differences of CUDA and OpenCL, giving a detailed analysis of the performance gap from all possible aspects. We will analyze it to better understand the difference in CUDA and OpenCL from a performance point of view.

| App. | Suite | Dwarf/Class* | Performance Metric | Description |
|---|---|---|---|---|
| BFS | Rodinia | Graph Traversal | sec | Graph breadth first search |
| Sobel | SELF | Dense Linear Algebra | sec | Sobel operator on a gray image in X direction |
| TranP | SELF | Dense Linear Algebra | GB/sec | Matrix transposition with shared memory |
| Reduce | SHOC | Reduce* | GB/sec | Calculate a reduction of an array |
| FFT | SHOC | Spectral Methods | GFlops/sec | Fast Fourier Transform |
| MD | SHOC | N-Body Methods | GFlops/sec | Molecular dynamics |
| SPMV | SHOC | Sparse Linear Algebra | GFlops/sec | Multiplication of sparse matrix and vector (CSR) |
| St2D | SHOC | Structured Grids | sec | A two-dimensional nine point stencil calculation |
| DXTC | NSDK | Dense Linear Algebra | MPixels/sec | High quality DXT compression |
| RdxS | NSDK | Sort* | MElements/sec | Radix sort |
| Scan | NSDK | Scan* | MElements/sec | Get prefix sum of an array |
| STNW | NSDK | Sort* | MElements/sec | Use comparator networks to sort an array |
| MxM | NSDK | Dense Linear Algebra | GFlops/sec | Matrix multiplication |
| FDTD | NSDK | Structured Grids | MPoints/sec | Finite-difference time-domain method |

Table 3.5 List of selected benchmarks in the [8] work

[14] P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, "From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming," tech. rep., Department of Computer Science, UTK, Knoxville Tennessee, September 2010

[15] K. Komatsu1, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi1, "Evaluating Performance and Portability of OpenCL Programs," in Proceedings of the Fifth international Workshop on Automatic Performance Tuning(iWAPT2010), (Berkeley, USA), June 2010

[16] Jianbin Fang, Ana Lucia Varbanescu and Henk Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL", Parallel and Distributed Systems Group, Delft University of Technology (Delft, Netherlands), September 2011

In order to compare the performance of CUDA and OpenCL, it is defined a normalized performance metric, called Performance Ratio (*PR*), which can be described as:

$$\text{PR} = \frac{Performance_{OpenCL}}{Performance_{CUDA}}$$

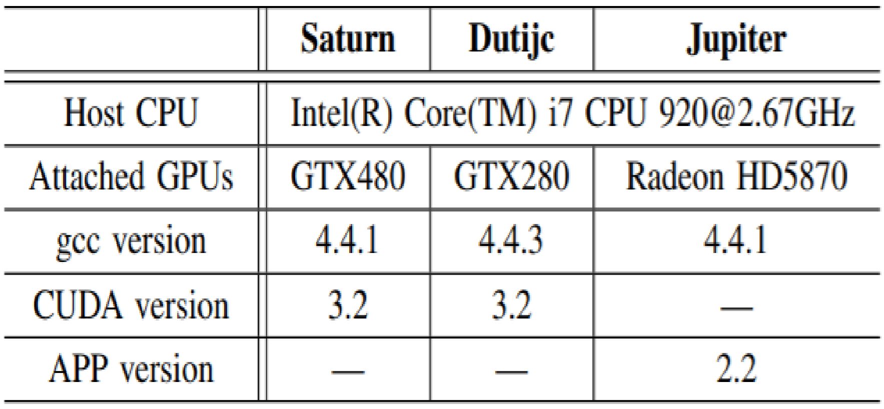
For *PR* < 1, the performance of OpenCL is worse than CUDA; otherwise, OpenCL will give a better or same performance. In an intuitive way, if $|1\text{-}PR| < 0.1$, it can be assumed that CUDA and OpenCL have similar performance. When it comes to different domains, performance metrics have different meanings. In memory systems, the bandwidth of memories can be seen as an important performance metric. The higher the bandwidth is, the better the performance is. For sorting algorithms, performance may refer to the number of elements a processor finishes sorting in unit time. Floating-point operations per second (Flops/sec) is a typical performance metric in scientific computing. Exceptionally, performance is inversely proportional to the time a benchmark that takes from start to end. For the above reasons, in the work [8], which we are now analyzing, the authors decided to select specific performance metrics for different benchmarks, as it can be seen in Table 3.1.

Benchmarks are selected from the SHOC (Scalable HeterOgeneous Computing) benchmark suite, Nvidia's SDK, and the Rodinia benchmark suite. The authors also used some self designed applications. These benchmarks fall into two categories: synthetic applications and real-world applications:

- Synthetic applications:
    - Synthetic applications are those which provide ideal instructions to make full use of the underlying hardware. The authors selected two synthetic applications from the SHOC benchmark suite: MaxFlops and DeviceMemory, which are used to measure peak performance (floating point operations and device-memory bandwidth) of GPUs in GFlops/sec and GB/sec. In their work[8], peak performance includes theoretical peak performance and achieved peak performance. Theoretical peak performance (or theoretical performance) can be calculated using hardware specifications, while achieved peak performance (or achieved performance) is measured by running synthetic applications on real hardware

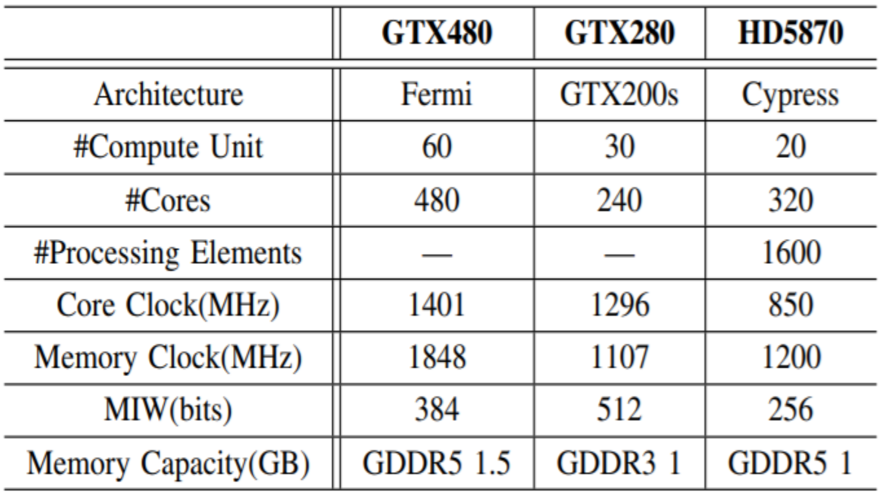
- Real-world applicarions:
    - Real-world applications include algorithms frequently used in real-world domains. The real-world applications selected are listed in Table 3.1. Among them, Sobel, TranP in both CUDA and OpenCL, and BFS in OpenCL are developed by the authors (denoted by “SELF”); others are selected from the SHOC benchmarks suite (“SHOC”), Nvidia’s CUDA SDK (“NSDK”) and the Rodinia benchmark suite (only BFS in CUDA, denoted by “Rodinia”).

The authors obtained all their measurement results on real hardware using three platforms, called Dutijc, Saturn, and Jupiter. Each platform consists of two parts: the host machine (one CPU) and its device part (one or more GPUs). The two following tables shows the detailed configurations of these platforms and of the attached GPUs.



| | **Saturn** | **Dutijc** | **Jupiter** |
|---|---|---|---|
| Host CPU | Intel(R) Core(TM) i7 CPU 920@2.67GHz | | |
| Attached GPUs | GTX480 | GTX280 | Radeon HD5870 |
| gcc version | 4.4.1 | 4.4.3 | 4.4.1 |
| CUDA version | 3.2 | 3.2 | — |
| APP version | — | — | 2.2 |

Table 3.6 Platforms detailed specifications



| | **GTX480** | **GTX280** | **HD5870** |
|---|---|---|---|
| Architecture | Fermi | GTX200s | Cypress |
| #Compute Unit | 60 | 30 | 20 |
| #Cores | 480 | 240 | 320 |
| #Processing Elements | — | — | 1600 |
| Core Clock(MHz) | 1401 | 1296 | 850 |
| Memory Clock(MHz) | 1848 | 1107 | 1200 |
| MIW(bits) | 384 | 512 | 256 |
| Memory Capacity(GB) | GDDR5 1.5 | GDDR3 1 | GDDR5 1 |

Table 3.7 GPUs detailed specifications

Three different GPUs have been used, which are Nvidia GTX280, Nvidia GTX480, and ATI Radeon HD5870. Intel® Core™ i7 CPU 920@2.67 GHz (or Intel 920) and Cell Broadband Engine (or Cell/BE) are also used as OpenCL devices. For the Cell/BE, the authors used the OpenCL implementation for IBM. Fort the Intel920, they used the implementation from AMD (APP v2.2), because when they write their paper Intel's implementation on Linux was still unavailable. In the following, the performance comparisons and analyses.

- Comparing peak performance

  - Bandwidth of device memory:

    $TP_{BW}$ (Theoretical Peak Bandwidth) is defined as:

$$TP_{BW} = MC * \left(\frac{MIW}{8}\right) * 2 * 10^{-9}$$

where MC is the abbreviation for Memory Clock. Using the above equation the authors have calculated the $TP_{BW}$ of GTX280 and GTX480 to be 141.7 GB/sec and 177.4 GB/sec, respectively. $AP_{BW}$ (Achieved Peak Bandwidth) is then measured by reading global memory in a coalesced manner. Moreover, the experimental results show that $AP_{BW}$ depends on work-group-size (or block-size), which has been set to 256. The result of the experiments with DeviceMemory in Saturn (GTX480) and Dutijc (GTX280) are shown in the following figure.

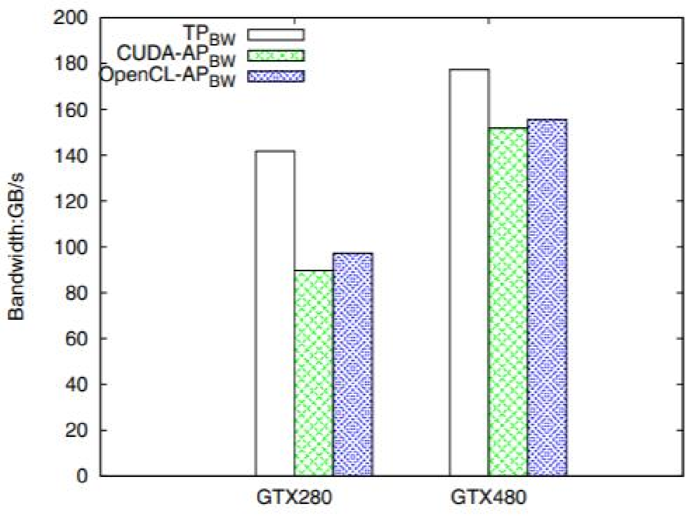


Fig 3.1 Peak bandwidth comparison between GTX280 and GTX480

It can be seen that OpenCL outperforms CUDA in $AP_{BW}$ by 8.5% on GTX280 and 2.4% on GTX480, respectively.

- Floating point performance:

  $TP_{FLOPS}$ (Theoretical Peak Floating Point Operations per Second) is calculated as:

$$TP_{FLOPS} = CC * \#Cores * R * 10^{-9}$$

where CC is short for Core Clock and R stands for maximum operations finished by a scalar core in one cycle. R differs depending on the platforms: it is 3 for GTX280 and 2 for GTX480, due to the dual-issue design of the GT200 architecture. As a result, $TP_{FLOPS}$ is equal to 933.12 GFlops/sec and 1344.96 GFlops/sec for the two GPUs, respectively. $AP_{FLOPS}$ (Achieved Peak FLOPS) in MaxFlops is measured in different ways on GTX280 and GTX480. In the first, a *mul* instruction and a *mad* instruction appear in an interleaved way (in theory they can run on one scalar core simultaneously), while only *mad* instructions are issued for GTX480. The experimental results are compared in the following figure.

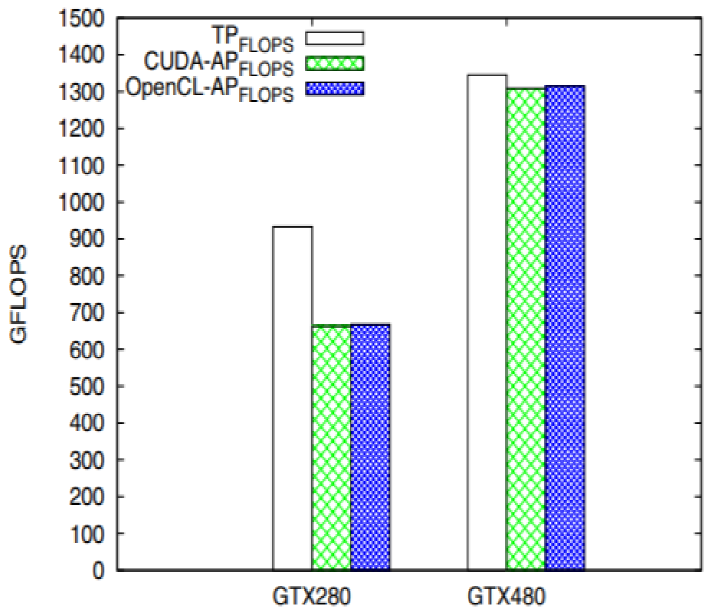


Fig 3.2 Peak FLOPS comparison between GTX280 and GTX480

It can be seen that OpenCL obtains almost the same $AP_{FLOPS}$ as CUDA for GTX280 and GTX480, accounting for approximately 71.5% and 97.7% of the corresponding $TP_{FLOPS}$. Thus, CUDA and OpenCL are able to achieve similar peak performance (to be precise, OpenCL even performs slightly better), which shows that OpenCL has the same potential to use the underlying hardware as CUDA.

- Performance comparison of real-world applications

The real-world applications already mentioned above are selected to compare the performance of CUDA and OpenCL. The PR of all the real-world applications without any modification is shown in the next figure. As it can be seen, PR varies a lot when using different benchmarks and underlying GPUs. Those performance will be analyzed using the following criteria:

- Programming model differences
- Different optimizations on native kernels
- Architecture-related differences
- Compiler and run-time differences

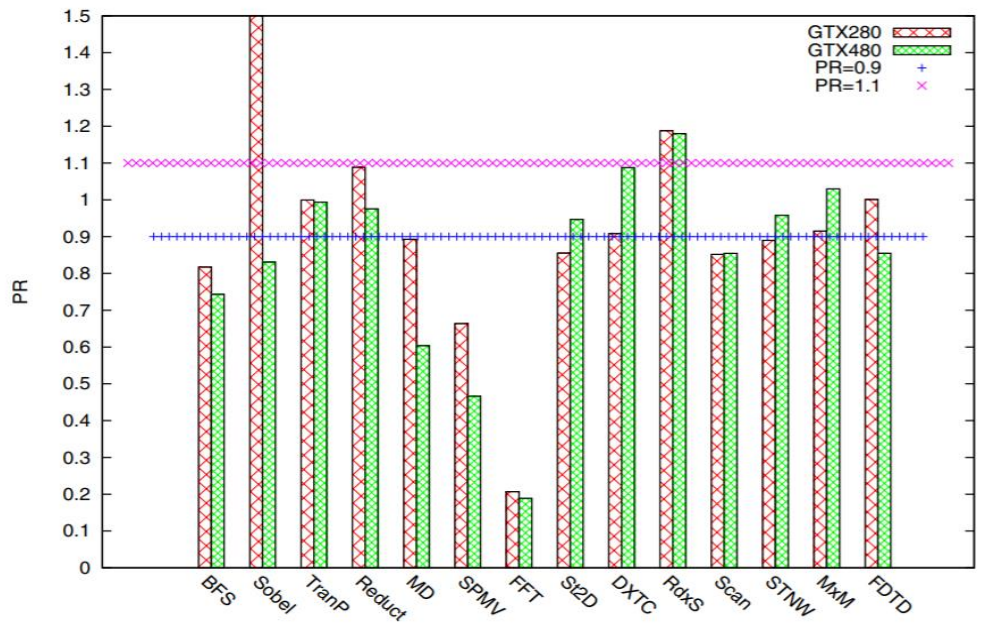
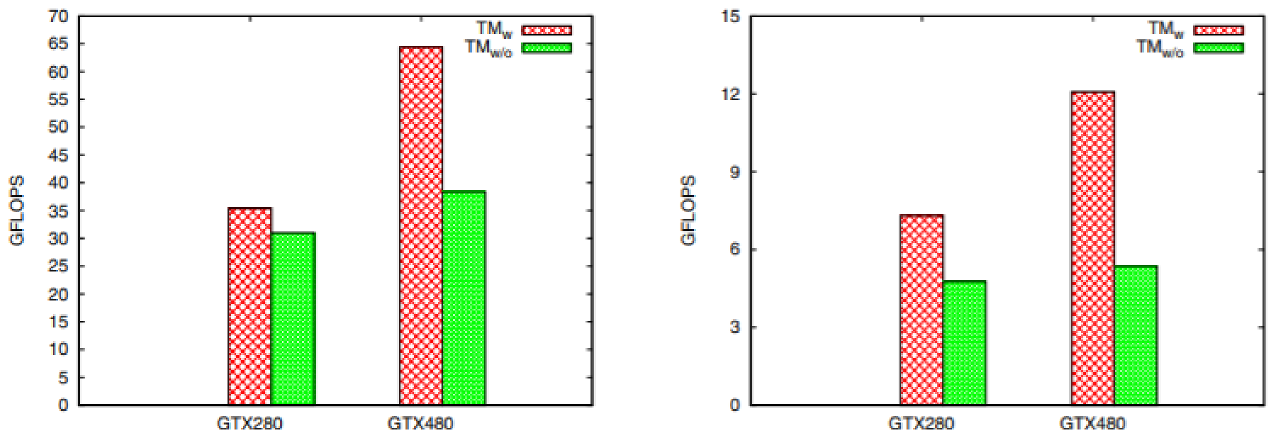


Fig. 3.3 A performance comparison of selected benchmarks. When the top border of a rectangle lies in the area between Line {PR = 0.9} and Line {PR = 1.1}, it is assumed that CUDA and OpenCL have similar performance. (On GTX280 the PR for Sobel is 3.2)

- Programming model differences:

  CUDA and OpenCL have many conceptual similarities. However, there are also several differences in programming models between them. For Example, NDRange in OpenCL represents the number of work-items in the whole problem domain, while GridDim in CUDA is the number of blocks. Additionally, they have different abstractions of device memory hierarchy, where CUDA explicitly supports specific hardware features which OpenCL avoids for portability reasons. Through analyzing kernel codes, the authors find that texture memory is used in the CUDA implementations of MD and SPMV. Both benchmarks have intensive and irregular access to a read-only global vector, which is stored in the texture memory space. The following figure shows the performance of the two applications when running with and without the usage of texture memory.



(a) MD Benchmark (b) SPMV Benchmark

Fig. 3.4 Performance impact of texture memory

As it can be seen, after the removal of the texture memory, the performance drops to about 87.6%, 65.1% on GTX280 and 59.6%, 44.3% on GTX480 of the performance with texture memory for MD and SPMV, respectively. The authors compared the performance of OpenCL and CUDA after removing the usage of texture memory, which results can be seen in the following figure, showing similar performance between CUDA and OpenCL. It is the special support of texture cache that makes the irregular access look more regular.

Then, texture memory plays an important role in performance improvement of kernel programs.

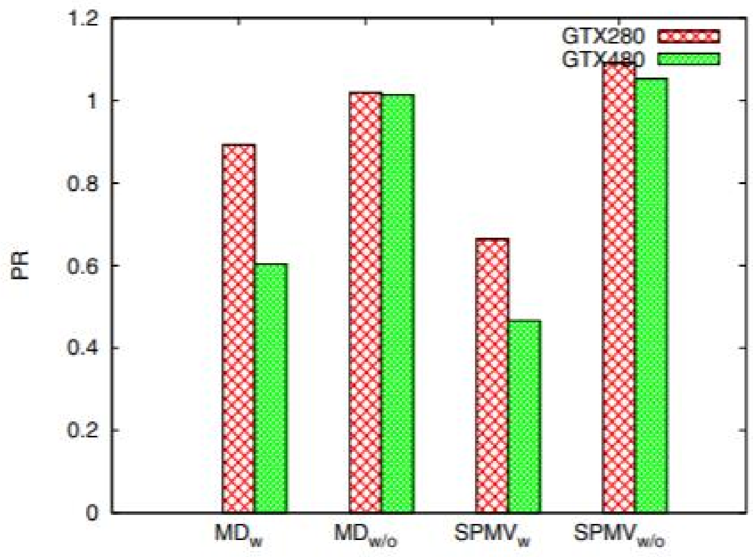


Fig. 3.5 Performance ratio before and after removing texture memory

- Different optimizations on native kernels:

  In a document written by Nvidia[17], many optimization strategies are listed, such as:

  - Ensure global memory accesses are coalesced whenever possible
  - Prefer shared memory access wherever possible
  - Use shift operations to avoid expensive division and modulo calculations
  - Make it easy for the compiler to use branch prediction instead of loops
  - Other strategies

  One key optimization to be performed in kernel codes is to reduce the number of dynamic instructions in the run-time execution. Loop unrolling is one of the techniques that reduces loop overhead and increases the computation per loop iteration[18]. Nvidia's CUDA provides an interface to unroll a loop fully or partially using the pragma *unroll* (a compiler optimization which replaces a piece of code into an unrolled one). When analyzing the native kernel codes of

[17] Nvidia Inc., OpenCL Best Practices Guide, May 2010

[18] M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, "A compiler framework for optimization of affine loop nests for gpgpus," in Proceedings of the 22nd annual international conference on Supercomputing, ICS '08, (New York, NY, USA), pp. 225–234, ACM, 2008

FDTD, the authors found out that the two codes are the same except that the CUDA code uses the pragma *unroll* at both unroll points a and b, while the OpenCL code unrolls the loop only at point b.

```
// Code segment of FDTD kernel
// Step through the xy-planes
#pragma unroll 9  //unroll point: a
for (int iz=0; iz<dimz; iz++){
  // some work here
#pragma unroll RADIUS  //unroll point: b
  for (int i=1; i<=RADIUS; i++){
    // some work here
  }
  // some work here
}
```

Fig. 3.6 Loop unrolling in a native kernel code of FDTD

The performance of the application (in CUDA only) with and without the pragma *unroll* can be seen in Fig. 3.7. As can be seen, the performance without the pragma *unroll* drops to 85.1% and 82.6% of the performance with it for GTX280 and GTX480. The authors then have removed the pragma at point a from the CUDA version and have found out the performance comparison between CUDA and OpenCL, here showed in Fig. 3.8. It can be seen that they achieve similar performance on GTX480, while OpenCL outperforms CUDA by 15.1% on GTX280. Moreover, it can be observed that when adding the pragma *unroll* at unroll point a of the OpenCL implementation, the performance degrades sharply to 48.3% and 66.1% of that of the CUDA implementation for GTX280 and GTX480, once again shown in Fig. 3.8.

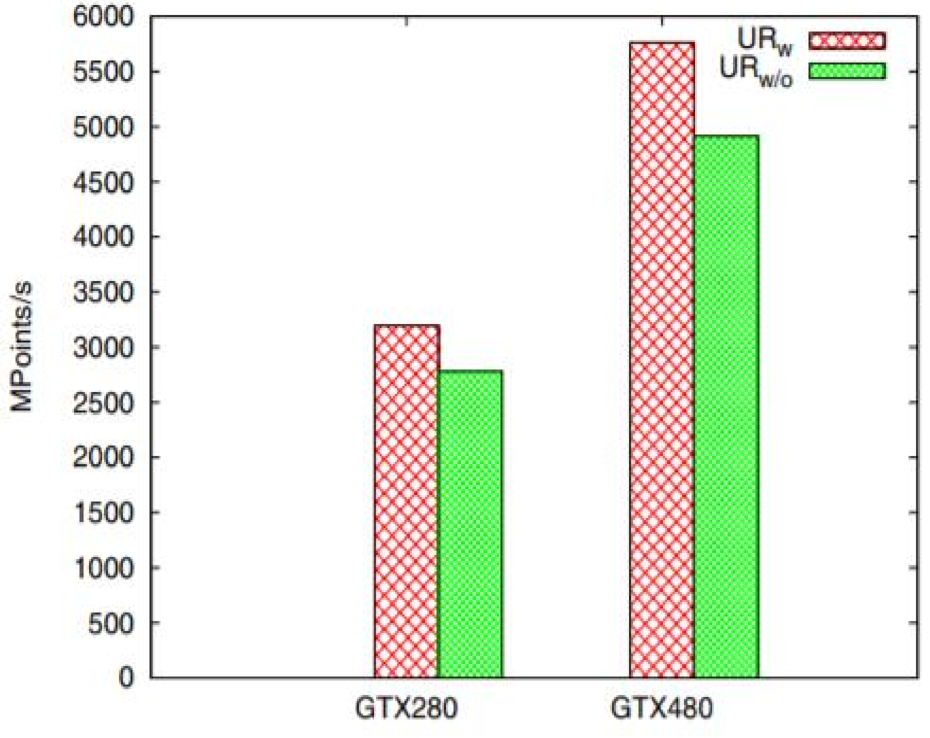


Fig. 3.7 Performance effects of loop unrolling (CUDA only)

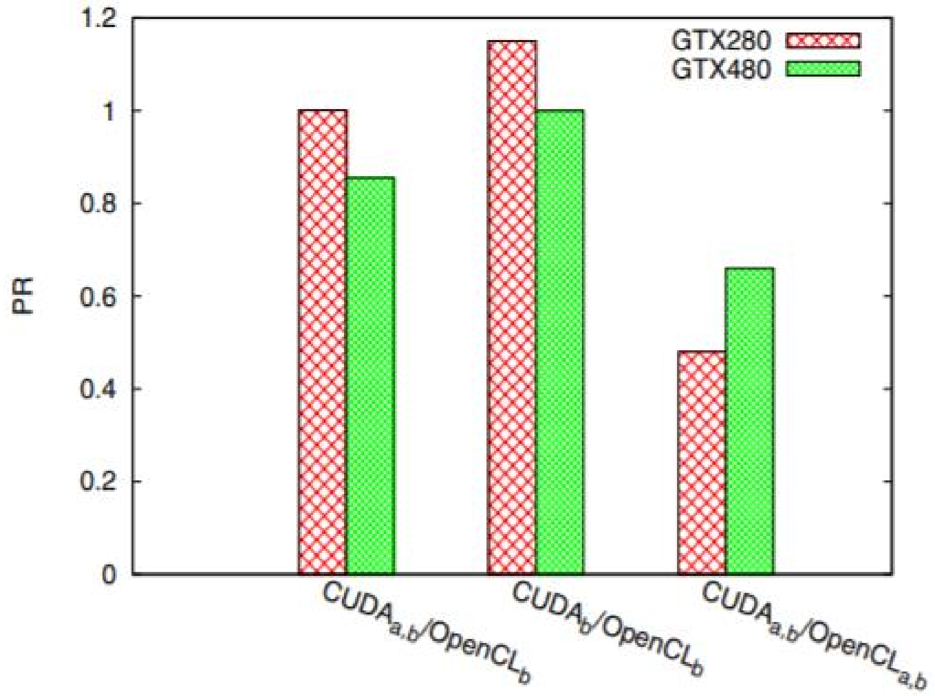


Fig. 3.8 Performance comparison of FDTD with and without loop unrolling at different points ($CUDA_x$ represents we execute loop unrolling at point x, and it is the same for OpenCL)

- Architecture-related differences:

  Since the birth of the original G80, the Fermi architecture can be seen as the most remarkable leap forward for GPGPU computing. It differs from the previous generations by, for example[19]:

  - Improved double precision performance
  - ECC (Error Correcting Code) support
  - True cache hierarchy
  - Faster context switching

  The introduction of the cache hierarchy has a significant impact on Fermi's performance. Looking at Fig. 3.3, it can be seen that values diverge remarkably for Sobel on GTX280 and GTX480. On GTX280, the OpenCL version runs three times faster than the CUDA one, but it only obtains 83% of CUDA's performance when the benchmark runs on GTX480. These differences are caused by the constant memory and the cache. In the implementation with OpenCL, constant memory is employed to store the "filter" in Sobel, while it is not in the CUDA version. After removing the usage of constant memory, the authors repeated the same experiments on the two GPUs. The execution time is showed in Fig. 3.9. On the one side, it can be seen that the kernel execution time drops to one quarter of that without using constant memory on GTX280. On the other side, there are few changes on GTX480 due to the availability of the global memory cache in the Fermi architecture. Overall, CUDA and OpenCL achieve similar performance with and without constant memory on GTX480.

[19] NVIDIA Inc., NVIDIAs Next Generation CUDA Compute Architecture: Fermi, 2009

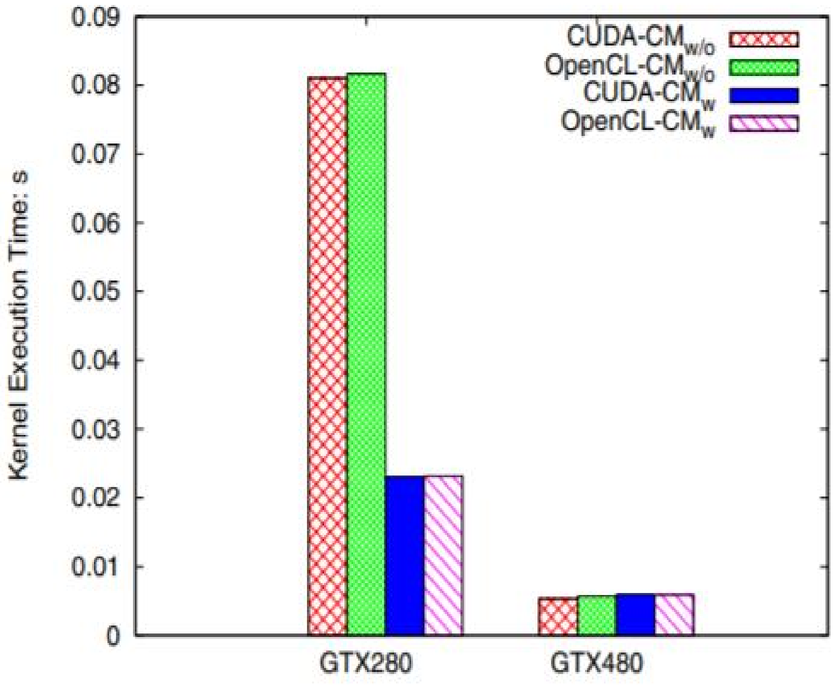
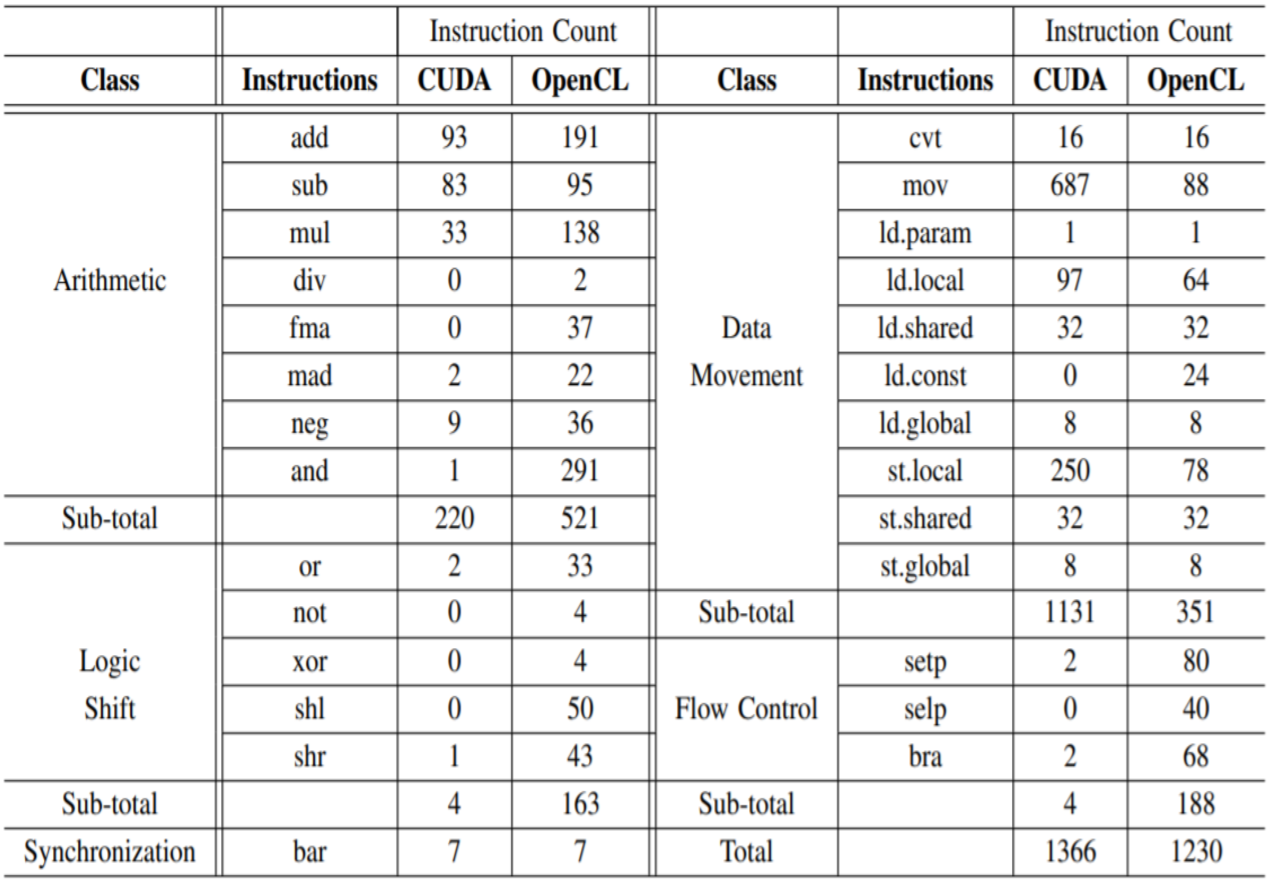


Fig. 3.9 Performance comparison for Sobel with and without constant memory on GTX280 and GTX480

- Compiler and run-time differences:

  Among all the benchmarks, the performance gap between OpenCL and CUDA is the biggest for the FFT. Their native kernel codes are exactly the same. However, when looking into their PTX codes, can be found notable differences between them. A quantitative comparison of these two PTX kernels is presented in the following table. The statistics are gathered for the “forward” kernel of the FFT implementation. Again from the table, the differences between the two PTX codes become visible. The OpenCL front-end compiler generates two times more arithmetic instructions than its CUDA counterpart. There are rarely any logic shift instructions in CUDA, while there are 163 such instructions in the OpenCL kernel. A similar situation happens with the flow-control instructions: there are many more for OpenCL than for CUDA. Although there are many more data movement instructions for CUDA, most of them are *mov*, simply moving data to or from registers or local memories. Finally, it can be noted that all time-consuming instructions such as *ld.global* and *st.global* are exactly the same. This situation can be explained by assuming that the front-end compiler for CUDA has been used and optimized more heavily, thus is more mature, than that of OpenCL. As a result, when it comes to some kernels like “forward” in FFT, OpenCL performs worse than CUDA.



| | | Instruction Count | | | | Instruction Count | |
|---|---|---|---|---|---|---|---|
| **Class** | **Instructions** | **CUDA** | **OpenCL** | **Class** | **Instructions** | **CUDA** | **OpenCL** |
| Arithmetic | add | 93 | 191 | Data Movement | cvt | 16 | 16 |
| | sub | 83 | 95 | | mov | 687 | 88 |
| | mul | 33 | 138 | | ld.param | 1 | 1 |
| | div | 0 | 2 | | ld.local | 97 | 64 |
| | fma | 0 | 37 | | ld.shared | 32 | 32 |
| | mad | 2 | 22 | | ld.const | 0 | 24 |
| | neg | 9 | 36 | | ld.global | 8 | 8 |
| | and | 1 | 291 | | st.local | 250 | 78 |
| Sub-total | | 220 | 521 | | st.shared | 32 | 32 |
| Logic Shift | or | 2 | 33 | | st.global | 8 | 8 |
| | not | 0 | 4 | Sub-total | | 1131 | 351 |
| | xor | 0 | 4 | Flow Control | setp | 2 | 80 |
| | shl | 0 | 50 | | selp | 0 | 40 |
| | shr | 1 | 43 | | bra | 2 | 68 |
| Sub-total | | 4 | 163 | Sub-total | | 4 | 188 |
| Synchronization | bar | 7 | 7 | Total | | 1366 | 1230 |

Table 3.8 Statistics for PTX instructions

BFS is also an interesting example here. It has to invoke the kernel functions several times to solve the whole problem. Thus, the kernel launch time (the time that a kernel takes from entering the command-queue until starting its execution) plays a significant role in the overall performance. The authors’ experimental results shows that the kernel launch time of OpenCL is longer than that of CUDA (the gap size depends on the problem size), due to the differences in the run-time environment. The longer kernel launch time may also explain why OpenCL performs worse than CUDA for applications like BFS.

- A fair comparison

In the above descriptions, it has been shown that the performance gaps between OpenCL and CUDA are due to programming model differences, different optimizations on native kernels, architecture-related differences, and compiler differences. It has been shown that performance can be equalized by systematic code changes. Therefore, in the following it will be presented an eight step fair comparison approach for CUDA and OpenCL applications from the original problem to its final solution, which provides guidelines for investigating the performance gap between CUDA and OpenCL, if any. In the following figure a resuming flow is presented.

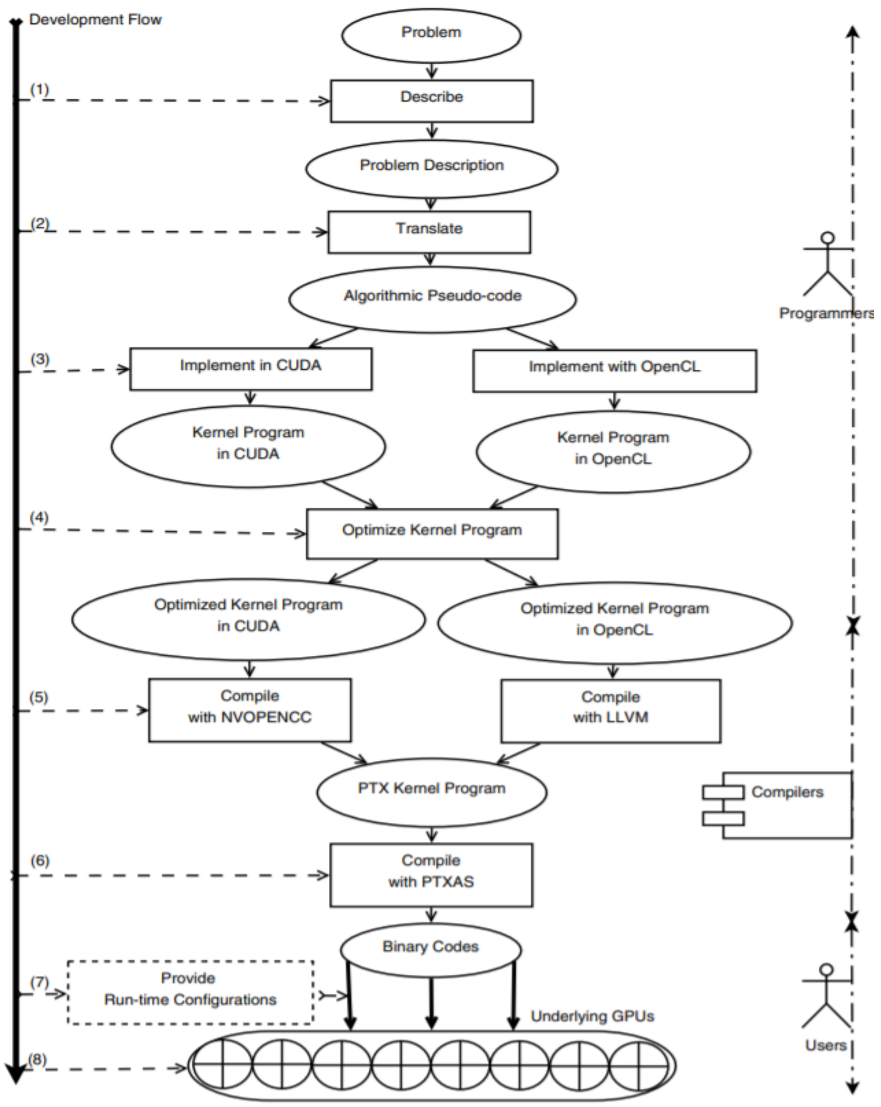


Fig. 3.10 Development flow of GPU kernel programs (the ellipses are the entities such as a program or a description and the rectangles represent actions on the entities. There are three types of roles participating the whole process: programmers, compilers, and users

1) Problem description:

This step describes what the problem is and what form the solution could be.

2) Algorithm translation:

How to address the problem which is given using certain algorithms. The algorithms can be described in pseudo-code which is environment-independent and easier for humans to understand.

3) Implementation:

In this step, the algorithms mentioned above are implemented with different programming models or languages. As for GPU programs, there are two parts: one is the host program and the other is the kernel code running on GPUs. On Nvidia GPUs, CUDA+C and OpenCL+C are usually adopted to implement GPU programs. If two implementations use similar APIs to access the same type of hardware resources, then the two implementations can be considered to be the same. Note that two implementations also have to use the same type of timers to measure performance.

4) Native kernel optimization:

After implementation, architecture-dependent optimizations on kernel programs are executed. For example, whether to use the shared memory (or local memory in OpenCL), whether to employ vectorization, whether to unroll loops, whether to reduce bank-conflicts, whether to use texture memory in CUDA, and whether to access global memory in a coalesced, way are decisions that should be taken into account. On the one side, optimizations on native kernels is a time-consuming and error-prone job; on the other side, it can contribute to performance improvement significantly.

5) First-stage compilation and optimization:

The first-stage compiler adopted in CUDA is called NVOPENCC. There is a similar front-end compiler for OpenCL in this stage. This stage compiles kernel codes into PTX codes, a low-level parallel thread execution virtual machine and instruction set architecture developed by Nvidia[20]. Some advanced optimizations are also executed in this stage.

6) Second-stage compilation and optimization:

PTXAS (the back-end compiler) translates PTX codes into binary format in this step and it may execute some additional optimizations.

[20] NVIDIA Inc., PTX: Parallel Thread Execution ISA Version 2.2, October 2010

7) Program configuration and start-up:

Before executing the program prepared so far, we need to configure two kinds of parameters:

- Problem parameters (the parameters of the problem to be solved such as the size of the matrix)
- Algorithmic parameters (for example, block-size or work-group size)

Although these parameters do not change the correctness of final results, they can have a significant impact on the performance of the application.

8) Running on GPUs:

With the help of drivers, the binary codes are finally scheduled to run on the GPUs.

These eight steps make up the application development flow from an original problem to its final solution. Based on this, it can be defined that a comparison between CUDA and OpenCL is considered "fair" when configurations in all the eight steps of the comparison are the same. According to the analysis in the previous pages, OpenCL can obtain similar performance to CUDA in the case of "a fair comparison". In real-world, programmers are responsible for steps from (1) to (4) and compilers take charge of steps (5) and (6). Finally, users will employ the application through steps (7) and (8), as illustrated in Fig. 3.10. Each of the eight steps is probably executed by different programmers (they have different programming habits, abilities and choices) or different compilers (they have different optimizations) or different users (they have different requirements and investments). All those lead to the difficulty of making sure that a performance comparison is fair between CUDA and OpenCL.

To final summarize this chapter, which the aim is to highlight the differences between CUDA and OpenCL, we have to clarify some points:

- CUDA and OpenCL shares many concepts, but they are different APIs/platforms for GPU/GPGPU programming
- Both of them have weak points and highlights, so not always one is better than the other
- OpenCL strength is the portability
- CUDA strength is performance on Nvidia's cards

- They do not differ a lot under a performance point of view, if they are analyzed in a very similar environment
- The best choice is to choose one platform or the other depending on the requirements of the work to be done

# Chapter four: SAR focusing system

## 4.1 Introduction

In this section I will talk in a general way about the SAR (Synthetic Aperture Radar) focusing system. The system taken into account has been developed with CUDA platform, and in this chapter we will discuss about the environment in which our example is placed.

## 4.2 SAR (Synthetic Aperture Radar)

SAR is a typology of radar that allows creating two or three dimensional images of an object, such as Earth's topography. It uses the motion of the radar antenna over a landscape to get the spatial data needed to create the two dimensional images of the landscape itself. For that reason, SAR is usually installed on a moving platform, e.g. a satellite, and its name derives from the fact that the radar pulses to return to the antenna the wanted data while the device is in movement over the target area for a certain amount of time. This is the difference between synthetic and physical aperture: the fact that the antenna itself is in movement. To create an image, many consecutive pulses are transmitted over the target area and the return pulses are received and recorded to be processed. While pulses are received and processed the antenna is in movement and other pulses from different points are continuously recorded and processed, to finally be merged all together to obtain an high resolution image of the target zone.

## 4.3 History

SAR was invented by Carl A. Wiley in 1951 while it was working for the Atlas ICBM program (ICBM stands for Intercontinental Ballistic Missile); the following year he constructed with some colleagues a concept validation system for that radar and in that decade the company in which he worked, the Goodyear Aircraft (than changed in Goodyear Aerospace) gave multiple contributions to SAR technology. At the same time and with no correlation with Wiley's work, some trials were executed at the University of Illinois' Control System Laboratory on a system very similar to the SAR one. Both works did the processing of the radar return's data with electrical circuit filtering methods, using Doppler frequency theory, but the results were not very good due to the limitations of the technology available in those years. The project was carried on by the Department of Defense, which was searching new techniques for military reconnaissance and collection of data for the intelligence. The University of Michigan and University of Illinois gave their contribution, showing new implementations as a Doppler assisted sub beamwidth resolution and a produced stripmap image. Later contributions were carried on again by the Defense system, in particular by the Army, Navy and Air Force combined together, which assigned a project to the University of Michigan. Here a group known as Radar and Optics Laboratory proposed to take into account not only some particular Doppler shifts but the entire history of all shifts; that would yield a better resolution quality. It was proposed to record the received signals useful to the final result in a cathode-ray tube, a vacuum tube used to display images. Those recording would be after processed in laboratory with a dedicated equipment. It was chosen a 35 mm film to do the recording, which, without premeditation, showed a final result with a great diffraction effect, not compatible with a sharp final focusing. A physicist recognize the opportunity to exploit that problem as an advantage and the recording, until that moment considered a scalar one, was changed into pairs of opposite sign vector ones of many spatial frequencies with the addition of a zero frequency bias quantity. This trick, in addition to some changes in the architecture of the focusing system to allow that, was the key to obtaining only the wanted beam to pass through a selected frequency band aperture. Moreover, to be coherent with that change, the light used to illuminate the area to be recoded had to be monochromatic and coherent, so a mercury vapor lamp passed through a color filter was chosen, the best then available technology; this resulted in a necessary long exposure to obtain a decent result, due to the weakness of the amount of light. The following problem was not how to create the radar, an already known topic at that time, but rather it was how to obtain signal linearity and frequency stability. Different approaches were found by the above quoted Universities, again with the help of the Department of Defense. Both of them installed the radar on a C-46 aircraft and they had the possibility to do many flights to continually test and debug the system. Some results were obtained (15m resolution), such as that an initial conclusion of the work by the Department of Defense was canceled due to the good news. In the following years other improvements were applied and in seventies a result of a 30,48 cm resolution was achieved. At the same time, technology increase in computer processing allowed to achieve finer results, considering also the possibility to illuminate better, in terms of time and power, the target area under different degrees, which was not possible before due to antenna limitations. This was called the spotlight mode. From here in after, it was clear that the SAR was very suitable for spacecraft systems; in fact, not only the Doppler effect was accurate also at such a distance, but the typical movement of

a satellite around an orbit was exactly what the researchers were searching to have specific results. In the years, technology  has scaled up and so has done SAR
construction, whit the possibility to have now great results in term of focusing a certain target zone.

## 4.4 The SAR system analyzed

The SAR focusing system we are taking into account belongs to the COSMO-SkyMed (Constellation of small Satellites for Mediterranean basin Observation) Earth-observation satellite space-based radar system, which is commissioned and funded by the Italian Ministry of Research and the MoD (Ministry of Defense) and directed by the ASI (Italian Space Agency) for both civilian and military purposes. It aims to provide data and services for different reasons, such as defense purposes, environmental monitoring and surveillance applications for the management of exogeneous, endogenous and anthropogenic risks, provision of commercial products and services.

The system is composed by four equal satellites which are named COSMO-SkyMed 1, 2, 3 and 4 respectively and, obviously, they are all equipped with a microwave high-resolution SAR operating in X-band, and they operate at more or less 620 km of height over the Earth surface, repeating their ground track every sixteen days. The operating life of each satellite is estimated to be five years.

From the COSMO-SkyMed official website are reported the driving mission requirements for the constellation development, which are:

- Capability to serve at the same time both civil and military users through an integrated approach (dual use system)
- Large amount of daily acquired images
- Satellites worldwide accessibility
- All weather and day/night acquisition capabilities
- Very short interval between the acceptance of the user request acquisition and the release of the remote sensing product (system response time)
- High image quality (e.g. spatial and radiometric resolution)
- Intrinsic capability to be a cooperating, interoperable, expandable to the other EO missions, multimission-borne element providing EO integrated services to large user communities on a worldwide scale (concepts of expandability, interoperability and multisensoriality)

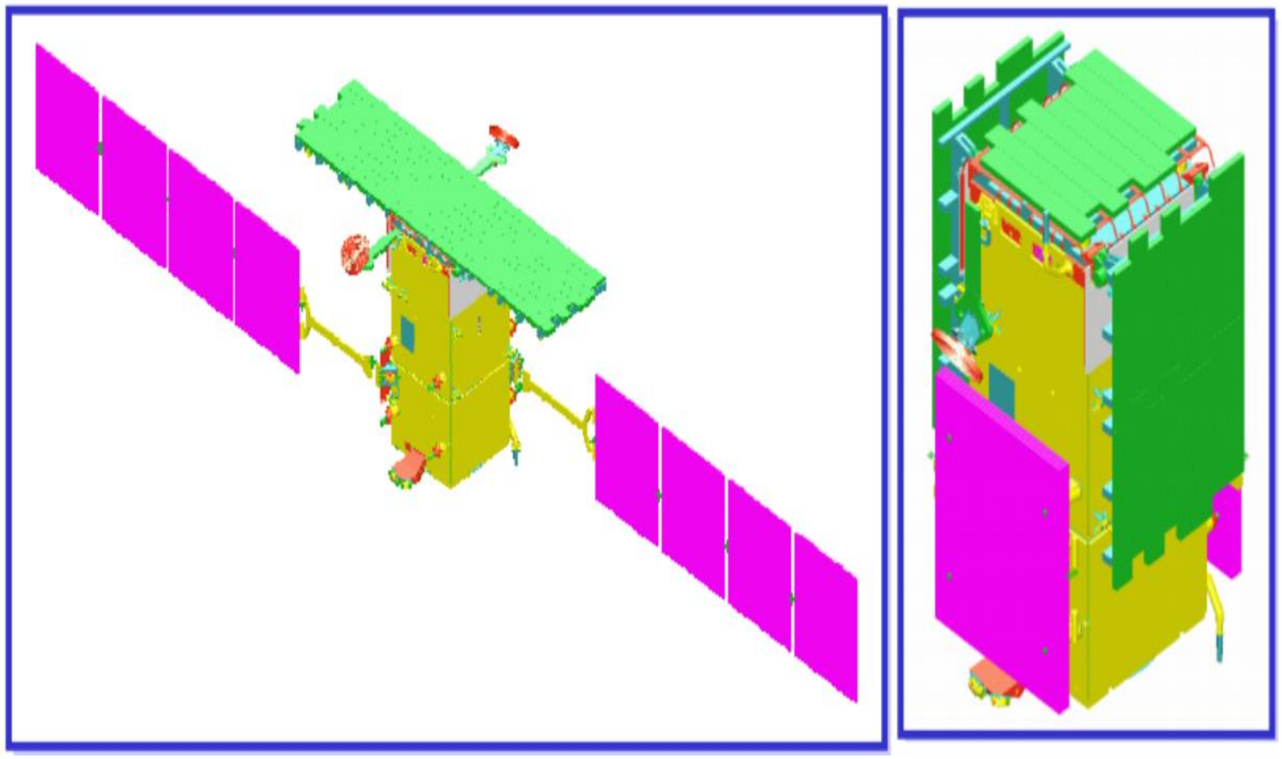


Figure 4.1 SAR satellite (deployed and stowed configuration)

## 4.5 How it works

A SAR is a radar for imaging purposes installed on a moving system, which is able to provide a remote sensing in many situations (all-weather, day and night times) throughout the illumination of radar beam. Electromagnetic pulses are transmitted and the return pulses are recorded and collected for the following processing; in fact, unlike optical sensors, a post processing procedure is required to achieve the desired image from the data obtained. For achieving those purposes, COSMO-SkyMed satellites uses three different sensor imaging operating modes:

- Two stripmap modes, for metric resolutions over tenth of km images; one mode is polarimetric with images acquired in two polarizations
- A spotlight mode, for metric resolutions over small images
- Two ScanSAR for medium to coarse (100 m) resolution over large swath

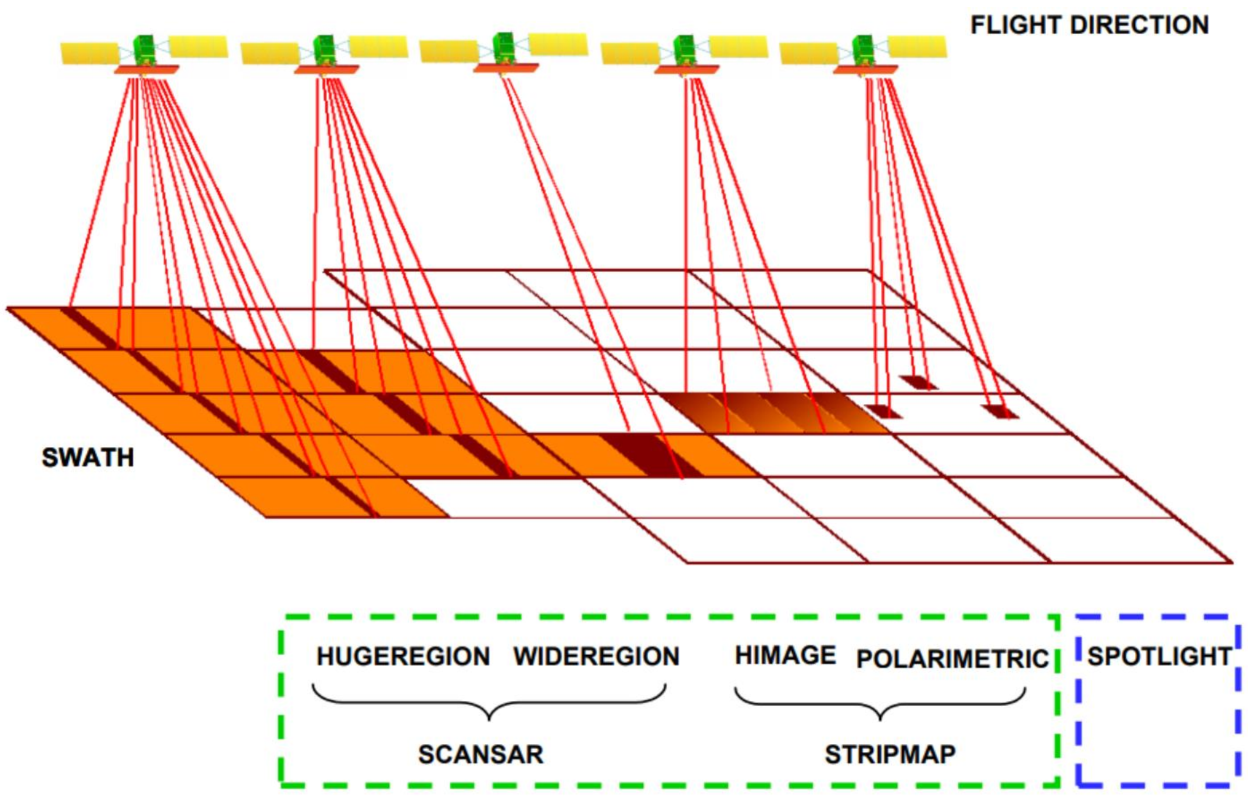


Figure 4.2 The three different imaging operating modes

We will analyze all of them for a better understanding but, in the next chapter, only the stripmap mode and the spotlight will be present in the software analysis.

The **stripmap mode** is the most common imaging mode and it is obtained by pointing the antenna towards a fixed direction with reference to the system flight path. The ground is illuminated continuously by the antenna as the system moves and operates. The acquisition is theoretically unlimited in the azimuth direction, but the reality is different due to duty cycle limitations of the SAR system (about 600s, allowing a strip length of 4500 km and more). Two different implementation of stripmap mode are provided: the Himage and the PingPong. In the first one, the radar transmitting and receiving configurations are time invariant, allowing to receive from each ground scatterer the full Doppler bandwidth allowed by the azimuth aperture of the antenna beamwidth. The main characteristics are a swath width of about 40 km, an azimuth extension for the standard product of about 40 km (6.5 seconds acquisition), PRF (Pulse Recurrence Frequency) values ranging from a minimum of 2905.9 Hz to a peak of 3874.5 Hz, a chirp duration in a range between 35 and 40 microseconds, a chirp bandwidth accommodated along range on the basis of the required ground resolution, spanning from 65.64 MHz at the far range (with a sampling rate of 82.50 MHz) to 138.60 MHz at the far range (with a sampling rate of 176.25 MHz). In the second one, a strip acquisition is implemented by alternating a pair of transmitting and receiving polarization across bursts (cross-polarization) obtained by mean of the antenna (which may be adjusted to be different on transmit and on receive). The acquisition is therefore performed in strip mode alternating the signal polarization between two of possible ones, so the combinations could be VV (vertical transmit and receive), HH (horizontal transmit and receive), HV (horizontal transmit and vertical receive) and VH (vertical transmit and horizontal receive). In this parametric burst mode only a part of the SAR length is available in azimuth and for that reason the azimuth resolution is limited. This mode provides a swath width value of about 30 km, an azimuth extension for the standard product of about 30 km (5.0 seconds acquisition), PRF values ranging from a minimum of 2905.9 Hz to a peak of 3632.4 Hz, a chirp duration fixed at 30 microseconds, a chirp bandwidth accommodated along range on the basis of the required ground resolution, spacing from 14.77 MHz at the far range (with a sampling rate of 18.75 MHz) to 38.37 MHz at the far range (with a sampling rate of 48.75 MHz.

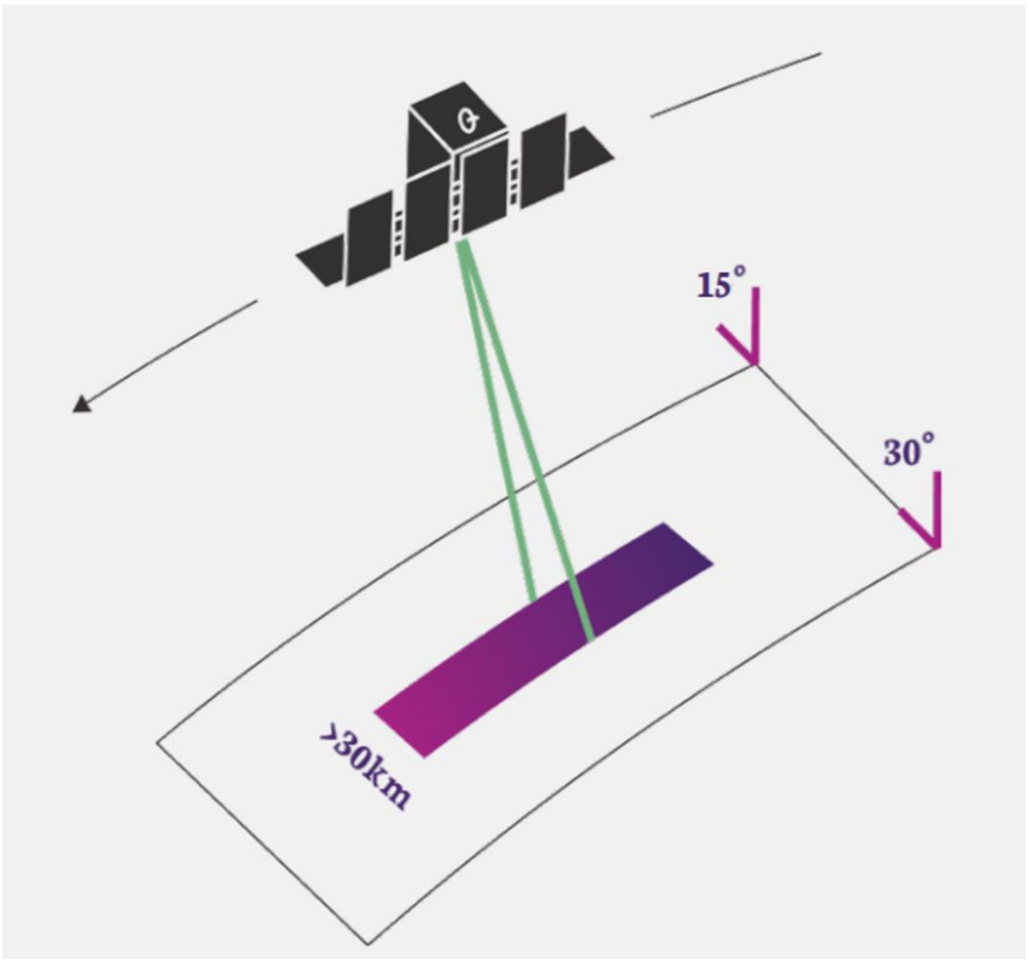


Figure 4.3 Stripmap imaging operating mode

In the **spotlight mode** the antenna is steered (both the azimuth and the elevation plane) during the total acquisition time to better illuminate the required target for a time period longer than the one of the standard strip side view, increasing the length of the synthetic antenna and therefore the azimuth resolution (at expenses of the azimuth coverage). In that configuration the acquisition is performed in frame mode, so it is limited in the azimuth direction due to the technical constraints deriving from the azimuth antenna pointing. The two different implementation allowed for this acquisition mode are SMART (only for defense purposes, not discussed here) and the Enhanced Spotlight. In the latter, the extension is achieved by an antenna electronic steering scheme requiring the centre of the beam steering to be located beyond the center of the imaged spot, so increasing the observed Doppler bandwidth for each target. This mode is characterized by an azimuth frame extension of about 11 km, a range swath extension of about 11 km, PRF values ranging from a minimum of 3148.1 Hz to a peak of 4116.7 Hz, allowed chirp duration in a range between 70 and 80 microseconds (depending on specific access area), a chirp bandwidth (accommodated along range on the basis of the required ground resolution) ranging from 185.2 MHz to 400.0 MHz and finally (due to the de-ramping processing) a sampling frequency equal to 187.5 MHz (same for each acquisition configuration.

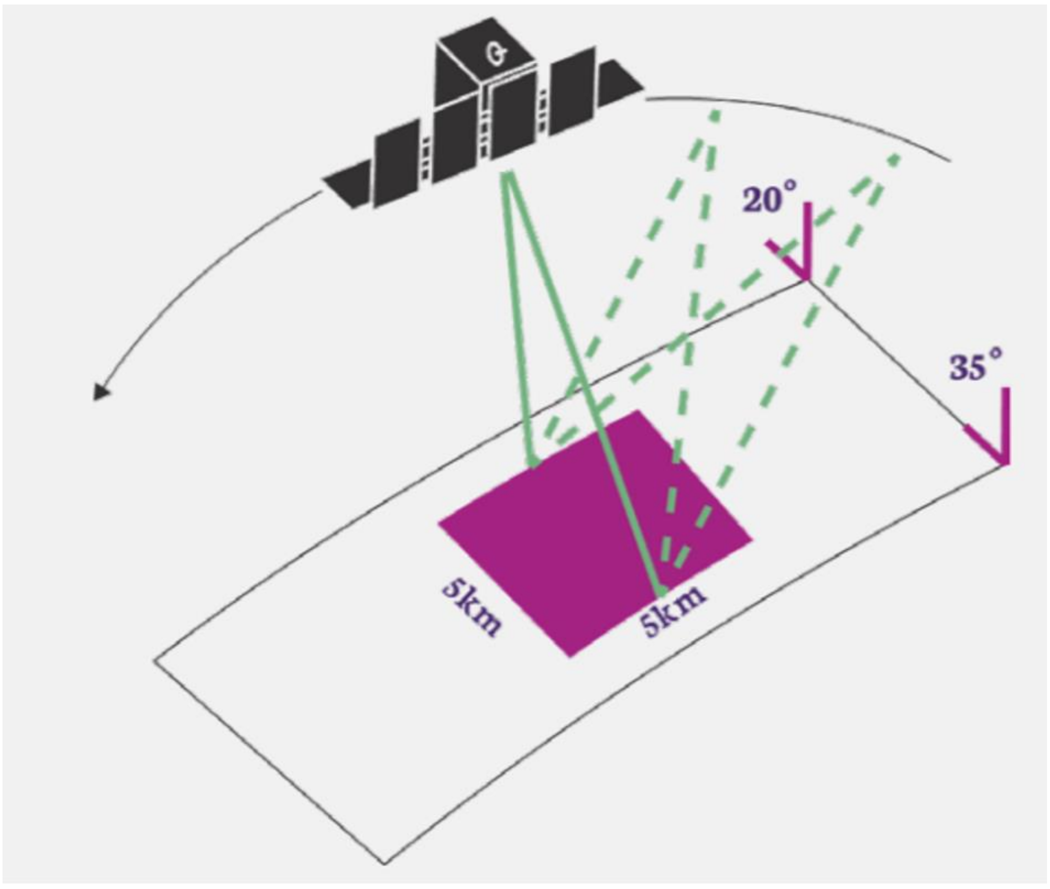


Figure 4.4 Spotlight imaging operating mode

In the **ScanSAR mode** a larger swath in range compared to the Stripmap mode is provided, but with a less spatial resolution, obtained by periodically stepping the antenna beam to closest sub-swaths. Since only a part of the synthetic antenna length is available in azimuth, the azimuth resolution is hence limited. In this configuration the acquisition is performed in adjacent strip mode, so it is virtually unlimited in the azimuth direction, but the reality is different due to duty cycle limitations of the SAR system (about 600s). The two different implementations provided by this acquisition mode are WideRegion and HugeRegion. In the first one, acquisition is grouped with three adjacent sub-swaths which permits to achieve a ground coverage of about 100 km in the range direction. The azimuth extension for the standard product is about 100 km (so forecasted for the origination of a square frame) corresponding to an acquisition of about 15 seconds. This mode is characterized by a PRF values which goes from a minimum of 2905.9 Hz to a peak of 3632.4 Hz, a chirp duration in a range between 30 and 40 microseconds, a chirp bandwidth accommodated along range on the basis of the required ground resolution and which goes from 32.74 MHz at the far range (with a sampling rate of 41.25 MHz) to 86.34 MHz at the far range (with a sampling rate of 108.75 MHz) In the latter, acquisition is grouped with up to six adjacent sub-swaths which allows to achieve a ground coverage of about 200 km in the range direction. The azimuth extension for the standard product is about 200 km (so forecasted for the origination of a square frame) corresponding to an acquisition of about 30 seconds. This mode is characterized by a PRF values which goes from a minimum of 2905.9 Hz to a peak of 3632.4 Hz, a chirp duration in a range between 30 and 40 microseconds, a chirp bandwidth accommodated along range on the basis of the required ground resolution and which goes from 8.86 MHz at the far range (with a sampling rate of 11.25 MHz) to 23.74 MHz at the far range (with a sampling rate of 30.0 MHz).

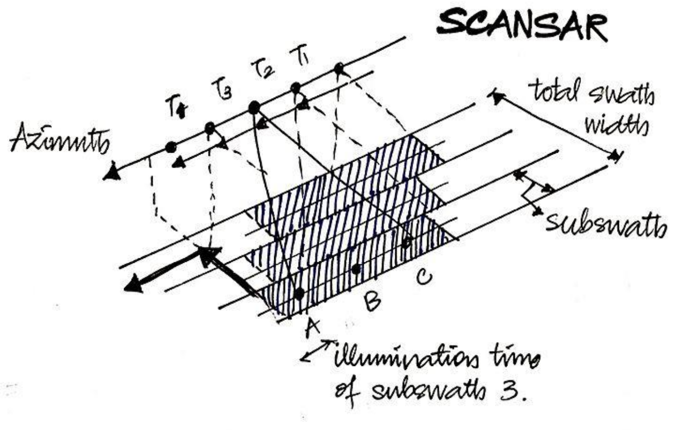


Figure 4.5 ScanSAR imaging operating mode

## 4.6 How SAR pictures the world

SAR system transmit microwave signals at an oblique angle and measure the backscattered portion of the sent signal to be able to analyze characteristics of the surface. This measurement can be described in a mathematic way with the Radar Cross Section (RCS) term σ, which is equal to the ratio between the incident and received signal intensity:

$$\sigma = \frac{I_{received}}{I_{incident}} * 4\pi R^2 \ [m^2]$$

The RCS recorded by a SAR for specific characteristics of a surface is not always easy to analyze, due to the fact that it is influenced both by a range of scene characteristics as well as by the parameters of the imaging sensor. The most important scene parameters driving RCS are surface roughness $h_{rough}$ and the dielectric properties of the imaged object quantified by its complex relative dielectric constant $\varepsilon_r$. While the first parameter describes how much of the scattered radar energy is directed back to the sensor, the dielectric properties guide whether or not signals may penetrate into the scattering surface. The fact that both of these parameters are a function of sensor wavelength explains why the characteristics of the sensor play a role when trying to interpret the measured signature of real-life objects in a SAR image. The dielectric properties of a medium decide how a microwave signal of wavelength λ interacts with a scattering medium such as the Earth's surface. These properties decide how much of the incoming radiation scatters at the surface, how much signal penetrates into the medium, and how much of the energy gets lost to the medium through absorption. In Figure 4.6 an overview of the influence of sensor wavelength λ on signal penetration into a different surface typologies is provided. The radar signals penetrate deeper as sensor wavelength increases, due to the dependence of the dielectric constant $\varepsilon_r$ on the incident wavelength, allowing for higher penetration at L-band (frequencies in the radio spectrum from 1 to 2 GHZ) than at C (frequencies in the radio spectrum from 4.0 to 8.0 GHZ) or X (frequencies in the radio spectrum from 7.0 to 11.2 GHZ) ones. The rule of increasing penetration with increasing sensor wavelength is valid for different surface typologies, such as high density vegetation and bar surfaces as alluvium soils and glacier ice. To quantify penetration depths $\delta_p$ into bare surfaces, information about the dielectric properties $\varepsilon_r$ of the medium are requested; in that case, it can be approximated as:

$$\delta_p \approx \lambda \sqrt{\frac{\varepsilon_r'}{2\pi \varepsilon_r''}}$$

where $\varepsilon_r'$ is the real component and $\varepsilon_r''$ is the imaginary one of the complex relative dielectric constant. In addition to soil density and sensor wavelength, these terms are strongly dependent on the moisture content of the medium.

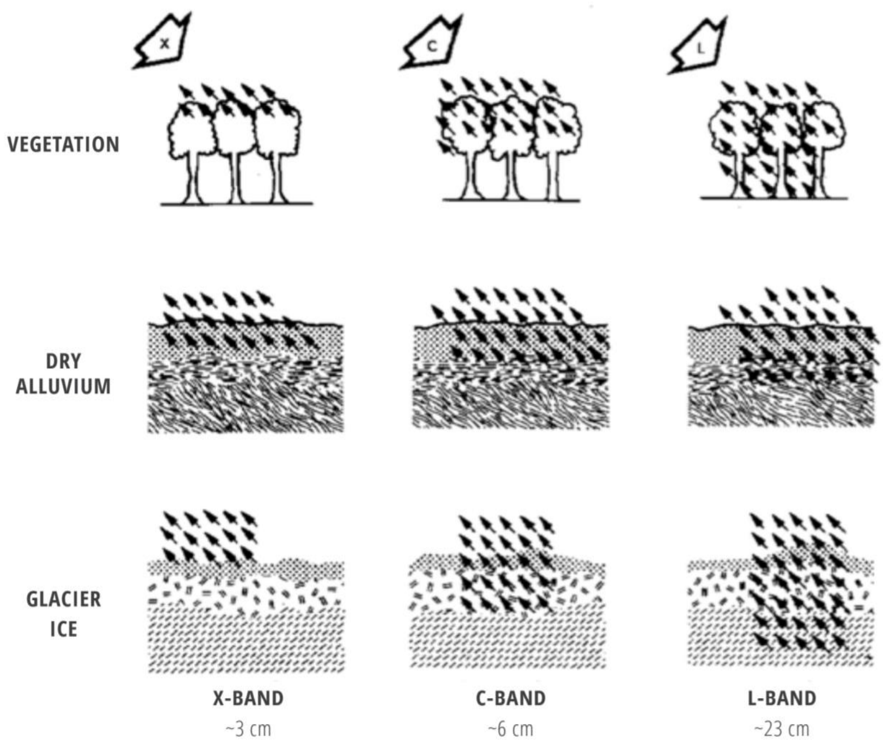


Fig 4.6 SAR signal penetration in terms of sensor wavelength λ

With few exceptions, such as dry snow and dry sandy soils, most bare or low vegetation surfaces allow a very little penetration for microwave radiation, such that surface scattering dominates the measured radar response. In these cases, the roughness of the scattering surface is the main driver defining the observed RCS in a SAR scene. For a narrow-band imaging system as it is SAR, if a surface appears rough or not can only be decided by taking into account the sensor wavelength. If the scale of roughness of a randomly rough surface is characterized using the standard deviation of the height deviation $h$ from some mean height $\bar{h}$ of the surface, then we know how large $h$ has to be for a surface to appear rough to an observing SAR system. According to the Fraunhofer criterion, a surface is defined as rough if the height deviations exceed the value $h_{rough}$, which is defined as:

$$h_{rough} = \frac{\lambda}{32 * cos\theta_i}$$

The above relationship depends on the signal wavelength λ and indicates that a surface with fixed height variations *h* may qualify as rough in X-band but possibly not in C or L ones. This concept of wavelength-dependent roughness is visualized in the following image, which shows increasing roughness conditions from left to right and identifies the transition from smooth, to intermediately rough, to rough surfaces, in accordance with the Fraunhofer criterion in the above equation.

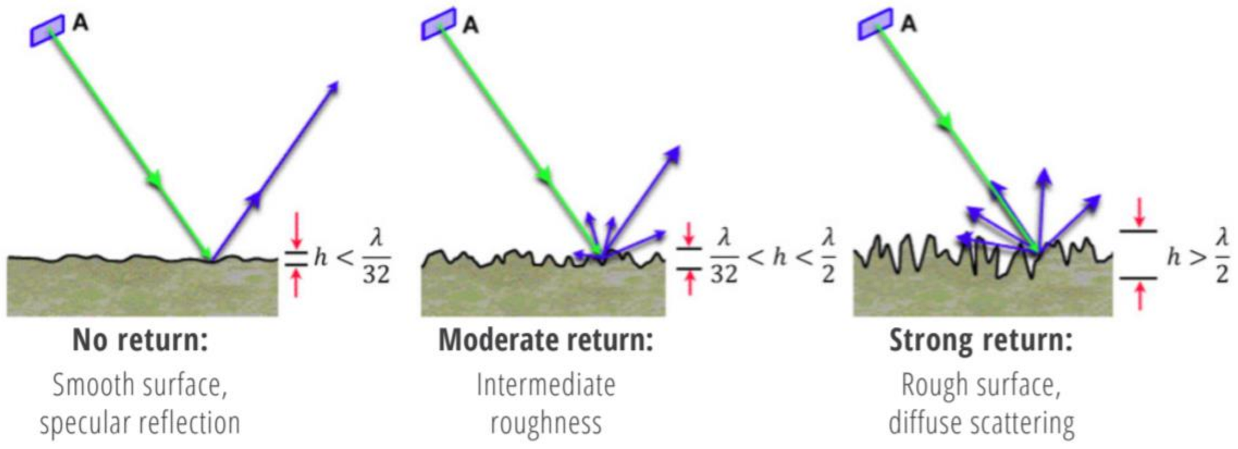

A
$h < \frac{\lambda}{32}$
No return:
Smooth surface, specular reflection
A
$\frac{\lambda}{32} < h < \frac{\lambda}{2}$
Moderate return:
Intermediate roughness
A
$h > \frac{\lambda}{2}$
Strong return:
Rough surface, diffuse scattering

Fig. 4.7 Dependence of surface roughness on the sensor wavelength λ

It can be seen that the amount of backscatter increases (length of blue arrows pointing toward the sensor) as roughness increases such that rough surfaces (at wavelength λ) have higher RCS than intermediately rough or smooth surfaces. The wavelength dependence also means that a surface will look increasingly darker as wavelength increases from X-band through C-band to L-band.

A SAR is an active instrument with its own source of illumination and it is one of the few sensing instruments which allows the full control and exploit of the polarization of the signal on both the transmit and the receive paths. Polarization describes the orientation of the plane of oscillation of a propagating signal. In linearly polarized systems, the orientation of that plane of oscillation is constant along the propagation path of the electromagnetic wave. In other systems, such as circular or elliptical polarized SARs, the orientation of the oscillation plane changes, describing geometric shapes such as circles or ellipses. Today a lot of SAR sensors are linearly polarized and transmit horizontal and/or vertical polarized waveforms. Many of the heritage SAR satellites carry single-polarized sensors, which support only one linear polarization. These sensors predominantly operate in HH (horizontal polarization on transmit; horizontal polarization on receive) or VV polarization (vertical polarization on transmit; vertical polarization on receive), while single-polarized sensors transmitting one linear polarization and receiving the other, such as HV or VH, are rare in practice.

More recent sensors provide either dual-polarization or quad-polarization capabilities. In the latter, the sensor alternates between transmitting H and V polarized waveforms and receiving both H and V simultaneously, providing HH, HV, VH and VV polarized imagery. Knowing the polarization from which a SAR image was acquired is fundamental, as signals at different polarizations interact differently with objects on the ground, affecting the recorded radar brightness in a specific polarization channel. Let's see some rules of thumb which should aid in the interpretation of polarimetric SAR data. For simplicity, it is assumed that a natural scene can be described as a combination of three types of scatterers: rough surface scatterers, double-bounce scatterers, volume scatterers, as we can see in the following figure. The first category in made up of low vegetation fields and bare soils, as well as roads and other paved surfaces. The second category includes buildings, tree trunks, light poles, and other vertical structures that deflect an initial first forward reflection back to the sensor. The third and last category is composed by vegetation canopies, as the signals bounce multiple times as they propagate  through the vegetation structure.

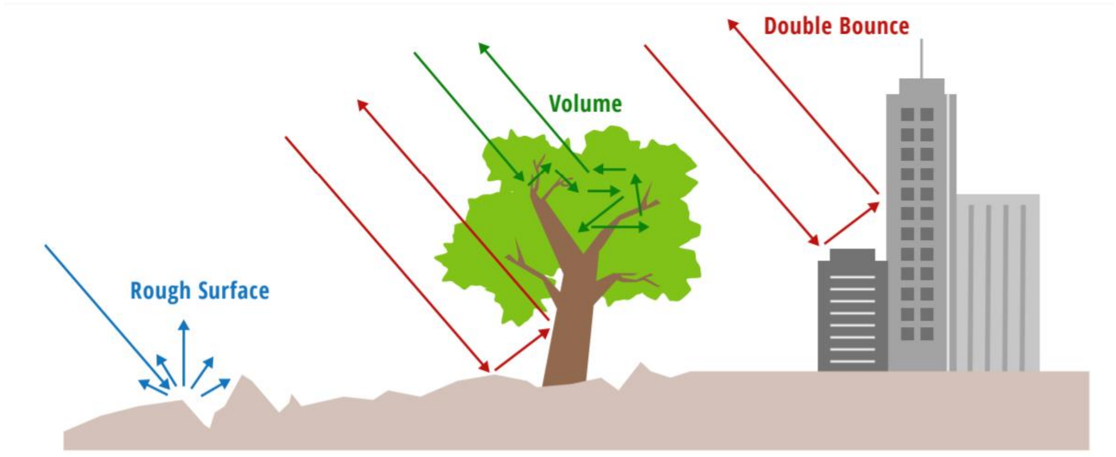


Fig. 4.8 Three main scattering types considered for SAR data

It turns out that these scattering types do not contribute to all polarimetric channels equally. Instead, each polarimetric channel "prefers" certain scattering types such that the scattering power $|S|$ in the individual polarimetric channels follows the following general scheme:

- Rough surface scattering → $|S_{VV}| > |S_{HH}| > |S_{HV}|$ or $|S_{VH}|$
- Double bounce scattering → $|S_{HH}| > |S_{VV}| > |S_{HV}|$ or $|S_{VH}|$
- Volume scattering → Main source of $|S_{HV}|$ and $|S_{VH}|$

These general rules should help when comparing the RCS in different polarimetric channels. They can be applied to perform an automatic classification of scattering types if data with all relevant polarizations (i.e. quad-polarization data) are available.

# Chapter five: porting CUDA to OpenCL in SAR focusing system

## 5.1 Introduction

In this last section I will present the changes which have to be applied to porting the SAR focusing system written in CUDA to OpenCL. Moreover, I will present some improvements which could be applied to have better performance results in focusing acquisition.

## 5.2 The initial work

The SAR focusing system was already well working with a CUDA implementation. In particular, I am referring to the work presented by Prof. Claudio Passerone, together with two colleagues of his, in a paper called "High Performance SAR Focusing Algorithm and Implementation". We will talk about this work and, after that, we will see the improvements which could be done to reach a finer work with the current technologies and instruments. They implemented the Stripmap and Spotlight imaging acquisition modes using the RDA (Range Doppler Algorithm) and the ω-k algorithms. They are both based on the compression of linearly frequency modulated signals (the so called chirp) along the range and azimuth directions, with the use of a matched filter in the frequency domain. The chirp in the range direction is generated by the radar antenna itself, while the chirp along the azimuth direction is generated by the Doppler effect, caused by the change of relative speed between the radar and the targets.

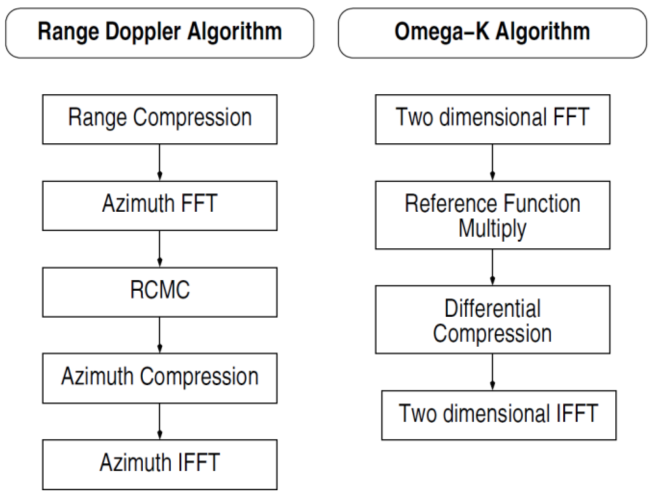


Figure 5.1 The flow's differences between the used algorithms

The Range Doppler Algorithm performs two separate compressions in the range direction and in the azimuth one, with the Range Cell Migration Correction performed in the Doppler domain. To achieve the finest accuracy and resolution the correct estimation of several parameters, such as the doppler frequency and the azimuth FM rate, is required. These values are not constant within an image, so polynomial models are used to characterize their variations, and the model parameters have been directly derived from the image data itself. The input to the algorithm is a *W x H* image with complex numbers in 32-bit floating point format (8 bytes for each pixel, 4 for the real part, and 4 for the imaginary part). This image corresponds to the samples received by the SAR system for each of the *H* transmitted pulses, with the high frequency carrier already removed on the spot in the space system. More auxiliary data are anyway present in the image file, and some of them are extracted for following processing phases. Among the available data there are biasing and antenna pattern values, which allow to correct the received samples to obtain a better quality signal with an higher signal to noise ratio. The corrections are realized using dedicated parallel kernels which operate on pixel values directly. The first step of the algorithm involves the convolution of each range row with a matched filter, that depends upon the chirp used during signal transmission. Chirp samples are contained in the raw image data, and are extracted during the loading phase and stored in the frequency domain, as the complex product of the spectrum of each range scan line with the conjugate of the spectrum of the chirp. Then for each line are required, and so performed, an FFT (Fast Fourier Transform), a complex multiplication, and an inverse FFT. To perform the above operations Nvidia CUDA comes in help, with a library named CuFFT which provides highly optimized functions to perform Fast Fourier Transforms. The library is widely used through the entire algorithm and requires the initial definition of a plan, with all the relevant information for the transformation, e.g. the number of samples and the direction of the transformation. An additional parameter is the number of concurrent FFTs to perform, called batch: computation is parallelized over hundreds of cores, so increasing the opportunities to exploit parallelism is a main target to improve performances. In particular, it emerged that an 8192 sample FFT takes more or less the same amount of time as a batch of 8 concurrent 8192 sample FFTs; higher values for the batch parameter leaded to a linear increase in the running time. Thus, in their implementation they have defined three plans: two of them have the batch parameter equal to *H*, and differ only for the direction of the transformation (forward and inverse, respectively). The third one is a forward FFT of a single line, which is applied to a zero padded version of the chirp, extended to the same size of a range scan line. The whole image is initially forward transformed with a single call of the FFT library function and, as the raw data is not needed anymore, the transformation is in place to save memory. Then the complex multiplication takes place: a dedicated pointwise kernel has been developed to perform it in parallel. At last, a second in place FFT, an inverse one, is applied to the entire image to return to the spatial domain. The original image is lost at this point, and only the range compressed image is kept for the subsequent processing phases. The SAR concept is based on the Doppler effect that changes the frequency of the transmitted signals as the system moves with respect to ground targets. For that reason, a good knowledge of Doppler parameters is a very important aspect to obtain the highest possible resolution along the azimuth direction. In theory, Doppler parameters can be computed from the relative geometry of the radar and target system. In practice, while this is a common practice for airborne SARs, the resulting accuracy for a satellite is often not adequate enough, as not only the satellite speed and altitude should be precisely known, but also Earth rotation and curvature should be taken into account. Hence, estimation of the Doppler parameters from the received data is required.

Two main parameters are estimated, using three different procedures, which share many similar processing steps:

- Doppler centroid: the Doppler frequency for a ground target when the satellite is at closest approach. This value is 0 for a SAR with antenna perfectly orthogonal to the flight path, and it is non-zero otherwise. The estimation is divided in two phases: first, the fractional part of the Doppler frequency with respect to the PRF is computed, then the integer part, also known as the ambiguity number $M_{amb}$ is estimated
- Azimuth FM rate: the rate of change of the Doppler frequency along the azimuth direction. This value is necessary to derive the correct matched filter for the subsequent azimuth compression step

In all cases, the estimated values differ slowly with the image location. Hence, for the estimation process the image itself is divided into several contiguous small blocks, parameters are computed for each individual block, and smooth polynomial fitting surface is then obtained using the Nelder-Mead method (a commonly applied numerical method used to find the minimum or maximum of an objective function in a multidimensional space), initialized with data obtained from geometrical and orbital parameters. This procedure leads to a better accuracy due to the fact that any error in the estimation process on a block is averaged with the other blocks. Most computations in the estimation processes are carried out in the range-Doppler domain (the spatial domain for the range direction) and in the frequency domain for the azimuth direction. Unfortunately, FFT functions in the CUDA library did not provide a stride parameter (they used CUDA toolkit 3.5; nowadays, toolkit 11.0 is available), so the samples of the arrays to be transformed needed to be contiguous in memory. Transposition of individual blocks, or of the entire image, was hence needed before applying the FFT. In order to be efficient, parallel matrix should exploit the memory access capabilities of the GPU hardware in such a way to group many main memory transactions (also known as coalescing) in a single wider transaction to optimize access time. In a linear parallel implementation of transposition, each thread reads one piece of data from the source matrix, and writes it to the destination buffer. Anyway coalescing is possible either while reading data or while writing it, not both of them at the same time, due to the matrix memory layout. An alternative implementation uses shared memory to accomplish the same task, while exploiting coalescing in main memory in both reading and writing. Threads are divided in an *n x n* matrix, where each thread reads a value from an *n x n* block in the source image, and stores it transposed in a temporary buffer in shared memory, which is much faster than main memory, and coalescing is not needed, while reading from main memory enjoys coalescing because threads read contiguous data. Once the temporary buffer is completely filled in, the destination main memory image is written. The destination block is in a transposed location compared to the source block, and threads are organized to store data contiguous in memory, in such a way to achieve coalescing also in writing. The idea behind this method is that a thread reads a value, but writes another one read by the transposed thread in the matrix.

Three individual procedures can now be analyzed in a deeper way to better understand how the mechanism works:

- Fractional Doppler centroid estimation: the range compressed image is divided into a number of blocks with a size defined by the user. For each block, the Doppler centroid fractional part is estimated as the peak of the Doppler magnitude spectrum. As the spectrum for a single azimuth line in a block is very noisy, the magnitude is averaged over all lines of the block; then, its maximum is found by convolving it with a power filter with a sinusoidal shape and by looking for the zero crossing. In the following, there are the detailed steps followed by each block:
  - The power filter is generated and its Fourier transform is computed (the power filter is identical for all blocks, and it is computed only once)
  - The block is transposed to get the azimuth direction in the rows, which are then transformed to the frequency Doppler domain using FFTs in batches equal to the block height
  - The magnitude of each azimuth line is computed, and the average of all lines gives the Doppler power spectrum. In both situations, pointwise parallel kernels were developed to increase performances
  - The power spectrum is then transformed using a forward FFT, multiplied with the power filter transform to perform the convolution, and finally inverse transformed to get back to the Doppler domain

    - The zero crossing of the convolution is determined on the host, as this computation is not time critical because it involves a single line

  Once all blocks are completed, a first order polynomial surface is fitted to the fractional centroid values, such that the final outcome of the procedure is given by:

$$d_{frac} = d_0 + d_1 r + d_2 a + d_3 ra$$

  where *r* and *a* are respectively coordinates of the range and azimuth directions. A second order polynomial model can also be used, with very few modifications and a slightly higher running time. Before fitting, the signal to noise ratio of the average power spectrums are also computed, and potentially inaccurate values are removed.

- Azimuth FM rate estimation: one more time, the range compressed image is divided into user defined blocks, possibly with a different size compared to the fractional Doppler centroid estimation procedure. Each block is transformed in the Doppler domain, and decomposed into two looks with non-overlapping spectrum. One by one, each look is separately compressed along the azimuth axis using a nominal FM rate $K_a$ derived from orbital data. In the case in which the nominal FM rate is not precise enough, it results in a mis-registration of the two compressed looks, and the amount of mis-registrations gives the error on the FM rate. Averaging over all range lines in the block allows to reduce noise and achieve a better accuracy. The following are the implemented steps:
    - The block is transposed to get the azimuth direction in the rows, and then rows are transformed to the frequency Doppler domain using FFTs in batches equal to the block height (if the block size is the same as the one used for the fractional Doppler centroid estimation, this step can be shared with the second step of the previous procedure, with a considerable saving in running time)
    - The block is compressed along the azimuth direction, using a matched filter derived by the complex multiplication with the transformed matched filter, and the result is kept in the Doppler domain at this stage. The matched filter is computed for each azimuth line separately, as the coefficients change from one scan line to another

  - The Doppler spectrum is divided in two looks, which are one by one inverse transformed back to the spatial domain; the magnitude of each look is also calculated
  - The mis-registration along the range direction of the two magnitude looks is computed using their mutual cross-correlation. To be able to do so, looks are transposed, transformed, complex multiplied, and transformed back. The average of all range lines is computed to reduce noise
  - The maximum of the cross-correlation is computed on the host processor; its position gives the amount of FM rate error $\Delta K_a$ for the block

  A first order polynomial model is used also in this case to get a smoothly varying FM rate over the entire image. The Nelder-Mead algorithm is started with the nominal FM rate, with the average FM rate error from all blocks already added to improve convergence.

- Integer Doppler centroid estimation: the last Doppler parameter which is estimated is the integer part $M_{amb}$ of the Doppler centroid. The procedure is similar to the FM rate estimation, as it is based on the detecting the mis-registration in the range direction of multiple looks. Anyway, in contrast to the technique described before, it starts from the range-Doppler image, rather than the range compressed one. Hence, the whole image needs to be transformed along the azimuth direction. To avoid allocating memory for the transposed image, the operation is carried out in vertical slices: a slice is first transposed in a temporary buffer, a batch of FFTs equal to the number of lines in the transposed buffer is applied, and then the slice is written back in its original position by transposing it again. After that, the integer Doppler centroid is estimated using the following steps:
  - The estimated fractional part of the Doppler frequency is used to determine the vertical location of the maximum of the Doppler power spectrum
  - The image is divided in slices (which are different and often larger than the slices used to transform the image in the range-Doppler domain), and for each one of them a set of pair looks, which are symmetric with respect to the maximum of the Doppler power spectrum, are defined at increasing distances from the estimated maximum

    - Each look is compressed in the azimuth direction by multiplying it with the frequency domain matched filter and transforming it back to spatial domain; furthermore, the magnitude of each look is computed
    - Magnitudes of compressed looks are compared in pairs to determine the range mis-registration. The displacement is computed by finding the maximum of the cross-correlation, averaged over all the lines of the looks to reduce noise. The cross-correlation is upscaled by a factor of 4 to get an higher accuracy. Pairs which are further away with respect to the maximum of the Doppler power spectrum show a linearly increasing displacement, where the rate of increase is related to the Doppler frequency
    - A fitting first order surface without the mutual *r x a* term is computed, and the integer part of the Doppler frequency is determined as the coefficient along the range direction, rounded to the lower integer value

For all three estimation procedures, a magnitude based method has been adopted. The literature also contains several phase based methods, which were in some cases also implemented. Anyway, the ones presented in the paper "High Performance SAR Focusing Algorithm and Implementation" gave the best results from both an accuracy and a performance point of view, so only some of them had been analyzed, while the others are not discussed.

Let's move on the Professor analysis. After estimations are over, the image is in the Range-Doppler domain, as it was transformed during the evaluation of the Doppler ambiguity number. If estimations are skipped, the azimuth FFT using coalesced transposition and slicing is carried out anyway, as it is necessary for the next steps in the algorithm.

- Range Cell Migration Correction (RCMC): Range cell migration occurs because the distance (range) between the antenna and the target changes along the flight path. This effect must be compensated, if that does not happen severe smearings appear in the final focused image. This is best achieved in the Doppler domain, as the spectrum for a target is also skewed, with the amount of skew depending on the effective Doppler frequency. Therefore, RCMS usually consists in a remapping of samples in the range-Doppler domain, using some sort of interpolation mechanism.

  The most important detail to an accurate RCMC is the precise knowledge of the Doppler centroid, given by both the integer and fractional parts estimated in the previous phases. Remapping is along the range direction only, and has been implemented in two user-selectable ways:

  - Using a nearest neighborhood method, to achieve high performance with a slightly less accurate correction
  - Using a weighted sinc interpolator, for maximum accuracy

In both cases, a GPU kernel performs the correction one range line at a time, concurrently for all pixels of each line. The result overwrites the initial image, in such a way that at the end of processing a corrected range-Doppler array is available.

- Azimuth compression: the last step in the algorithm performs the final compression along the azimuth direction. Differently to the matched filter in range, which is extracted from the chirp data, for azimuth compression the matched filter must be properly computed, and it depends on both the accurate estimations of the Doppler centroid and of the azimuth FM rate. Moreover, it changes with range, so different filters are used for each azimuth scan line.
  As the image is already in the Doppler domain, compression corresponds to a complex multiplication, followed by a last inverse FFT. To reduce the number of transpositions, the matched filter is computed directly in the frequency domain along range lines, and complex multiplication directly applied. This is performed using two successive parallel kernels, but they can in theory be grouped into a single one.

The final IFFT uses the usual coalesced transposition, followed by a batch of IFFTs, and terminated with a last coalesced transposition which overwrites the initial data. The resulting image is hence in Slant-range projection, and all pixels carry a complex data. To visualize it, the magnitude for each pixel must be computed.

Contrary to the RDA, the ω-k algorithm performs all compressions and corrections within the frequency domain in both range and azimuth directions, and it is able to achieve better accuracy for large antenna squint angles. The complexity is comparable to the RDA, and many of the processing steps are shared with it, so the authors of the paper "High Performance SAR Focusing Algorithm and Implementation" decided to reference them instead of repeat them.

In theory, as before described, the ω-k algorithm consists in a two dimensional FFT, the product with the reference function followed by the differential Stolt interpolation, and a concluding two dimensional IFFT. However, their implementation of fractional and integer Doppler centroid parameter estimations required a range compressed and a range-Doppler image respectively. While in RDA these images are readily available along the standard processing flow, they must be created ad hoc in the ω-k algorithm flow. Hence, the sequence of steps is different whether estimation of Doppler parameters from received data is performed or not; it will follow the flow which includes estimations, but is important to say that the other one is faster because it need less domain transformations.

The ω-k algorithm starts with range compression (a range FFT, the chirp matched filter multiplication, and a range IFFT). It follows a fractional Doppler centroid estimation, as the image is now in the range compressed state. The integer Doppler centroid estimation requires the image to be in the range-Doppler domain, so a coalesced transposition and an azimuth FFT are executed before invoking the estimation procedure. Up to this stage, the algorithm is actually identical to the RDA, except that the Doppler rate estimation is not performed, as it is not needed in the following steps.

The RDA and ω-k algorithm diverge at this point. As the image is currently in the range-Doppler domain, a forward range FFT is needed to transform it from range time to range frequency. RDA does not need it, as both RCMC and azimuth compression are performed in the range-Doppler domain. The reference multiply and differential Stolt interpolation are then executed.

- Reference function multiply: the complex reference function is computed for each scan line with a parallel pointwise kernel, using the estimated Doppler centroid two dimensional model at a reference range, corresponding to the middle of the image swath. This function consists of a phase term only (magnitude equal to 1), and its full formulation includes a square root; to improve performance, the square root is actually implemented using its second order Taylor approximation, with all intermediate computations carried out in double precision, and the final value converted to a single precision floating point complex value.

  After computing the reference function, a second pointwise kernel performs the actual complex multiplication, and this is repeated for all range lines in the image. The final output is an image in both the range and azimuth frequency domain, which is correctly focused at midrange, but still has a residual error along the range frequency direction, due to how the reference function has been computed.

  It has to be noted that the reference function multiplication is applied to an already range compressed image, rather than the original one transformed in the two dimensional frequency domain. The reason is that parameter estimations were carried out before this step. Anyway, this is acceptable, as this step is invariant with respect to range compression, and it allows to save the memory required to store the original image, while keeping the compressed one only.

- Stolt interpolation: as the residual error after the reference function multiplication depends on range frequency, rather than range time, it can be eliminated with a mapping interpolation along the range frequency dimension. This step, called Stolt interpolation, finally focuses the whole image.

  At first, a vector containing the value of range frequencies is computed. This is useful, as the interpolation is azimuth dependent, but range frequencies are not, so they can be stored and reused for all lines without modifications.

  Then, looping on all lines of the image, a parallel kernel computes the amount of displacement of each range frequency, as a floating point number, again using an approximation of the exact function, which includes a square root. All displacements are stored in a vector in the device memory, which is used as the input of an interpolation kernel, that generates the new line using a nearest neighborhood approach. A linear interpolation has also been implemented, but it did not show an appreciable better accuracy in the final result.

  Following the Stolt interpolation, the image is focused but still in the two dimensional frequency domain. A range IFFT, followed by a transposition and an azimuth IFFT brings the image back in the spatial domain, where each pixel is a complex value in the Slant-range projection. Similarly to the RDA, a visible image is obtained by computing the magnitude at each pixel.

The whole system for focusing using both the RDA and the ω-k algorithm had been implemented on a Tesla C1060 GPU with 4 Gbyte of main memory, on an Intel i7 host processor with 8 Gbyte of memory, running a 64 bit Linux operating system. It had been used the CUDA toolkit version 3.5. The system performance is better highlighted as follows:

- The transfer of a 1 GByte image from host to device memory (or the opposite direction) takes around 270 ms, with a throughput of more than 3.5 GByte/s
- The most used kernels are the one about Fourier transforms and transpositions.
- The range compression of a 16384 x 8192 image takes 330 ms (including forward FFT, complex multiplication and IFFT).
- The compression along the azimuth direction takes 3.9 s, with an initial transposition, forward FFT, matched filter multiplication, IFFT and final transposition
- In general, the FFT library reaches more or less 140 Gflops performance.

- For a 400 Mpixel input image the system takes about 20 s to complete the processing, with an equivalent throughput of 20 Mpixel/s
- The results show similar timing performance for both RDA and ω-k algorithms, with no very big differences between the two both in stripmap or spotlight mode.

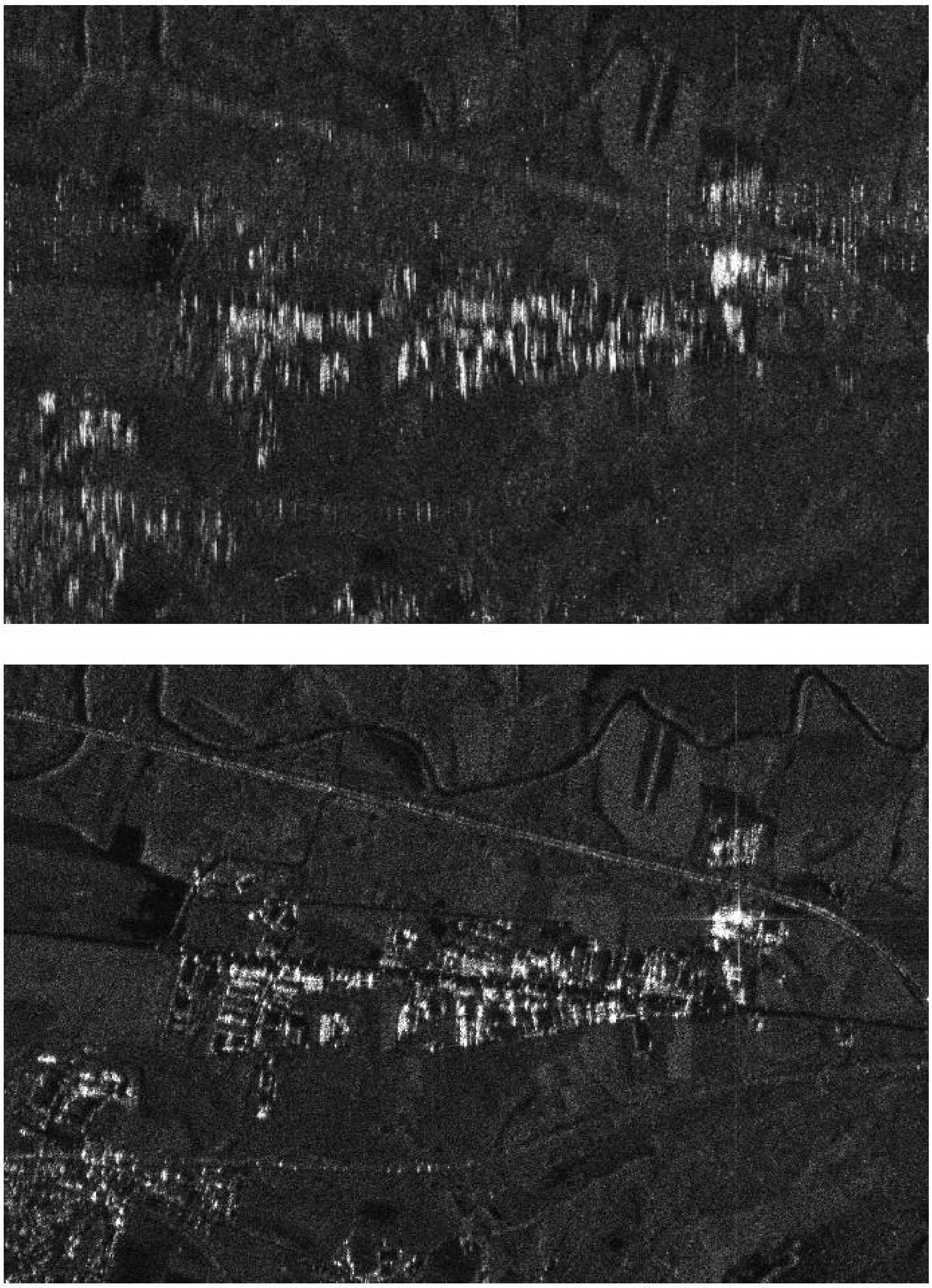


Fig. 5.1 Bulk compression without and with differential Stolt interpolation

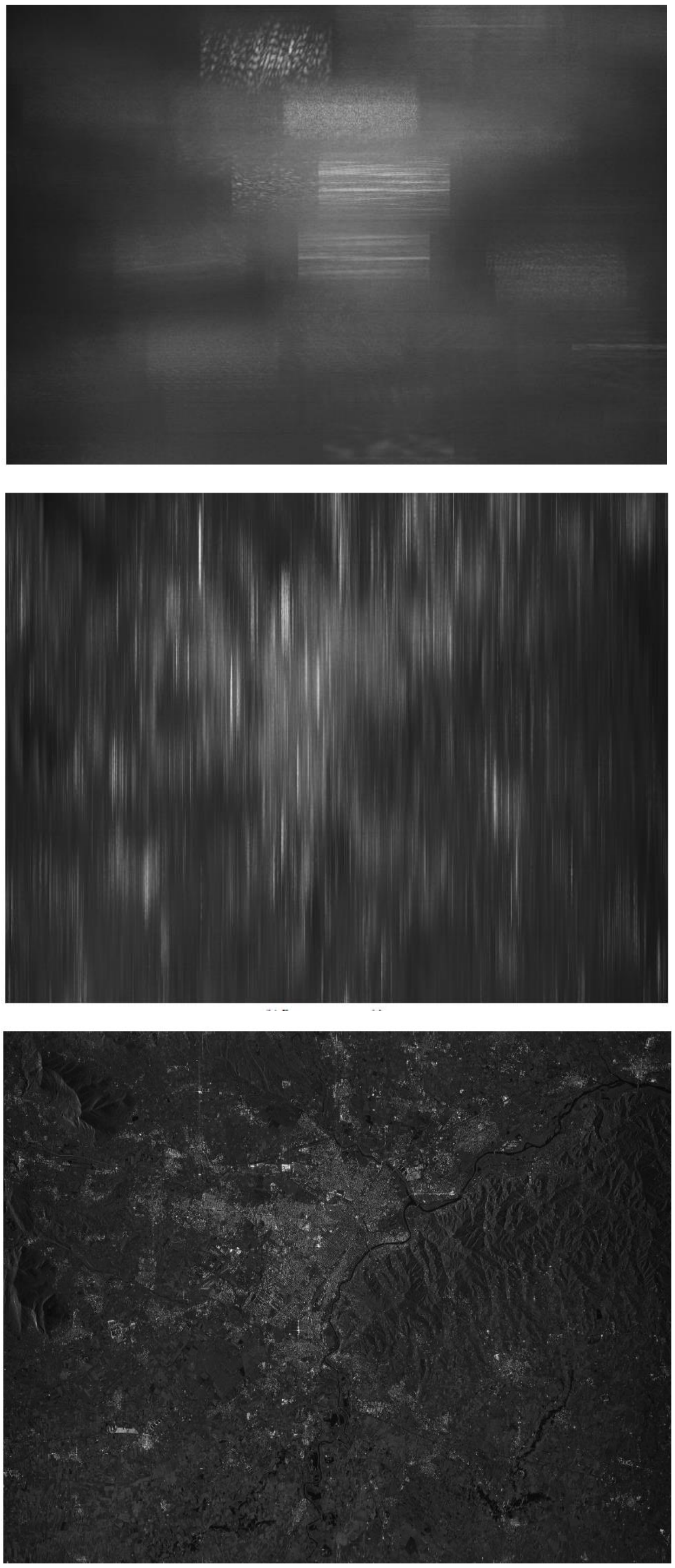


Fig. 5.2 Focusing of a stripmap image using the ω-k algorithm

The images above are the result of the optimum work done and explained in the paper "High Performance SAR Focusing Algorithm and Implementation". They are virtually indistinguishable to the ones from the Cosmo-SkyMed official processing toolchain.

## 5.3 Achievable improvements

The work which has be done is actually stunning, but some time has passed from that work so technology has improved and the same system could be improved not only to obtain a better performance but also to obtain a better portability. This is the key point of the thesis: how to improve both performance and portability all at once?

The idea is to porting the actual CUDA software in OpenCL, adding new features such as the use of a pipeline for memory transfers and a pipeline among kernels, doing the padding of the image not only on powers of two, using the new cuFFT library which allows to do the stride also in the vertical direction, and using more than one GPU at the same time.

### 5.3.1 Porting CUDA to OpenCL

Porting CUDA to OpenCL could seem an easy task, since actually some little changes appears to be applied. The reality is quite different and consequently, two different approaches could be used. The first one is to use automatic converting tools which read the code and change it accordingly to the requested characteristics; the second one is to manually  change the code.

The principal automatic converting tools actually known are Project Coriander, CU2CL, GPUOpen HIP. They are a reliable and quick method to convert an existing CUDA file in an OpenCL one, which allows programmers who already know CUDA to easily translate their job, but they have some disadvantages. The principal ones are that each one of them works with different versions of OpenCL and some of them do not have constant updates, so they do not run together with OpenCL updates. This is a limit to the potentiality of those methods. The first and the second methods are available on GitHub, while the third it is now called AMD ROCm$^{TM}$ Open Ecosystem which is property of AMD but anyway an open source tool. Let's have a deeper look inside one of those tools, the CU2CL one. CU2CL, which stands for CUDA-to-OpenCL, is a translator implemented as a Clang tool, a C language family fronted for LLVM. The tool interface is able, with a single invocation, to translate all the CUDA source files which will create a complete and working executable. CU2CL uses Clang's CUDA front-end to perform preprocessing, parsing, and abstract syntax tree generation, using a certain number of Clang's libraries through the translation process, which provide core functionalities to the translator such as file management, abstract tree traversal and information retrieval, the tool interface, preprocessor token access, and rewriting mechanisms.

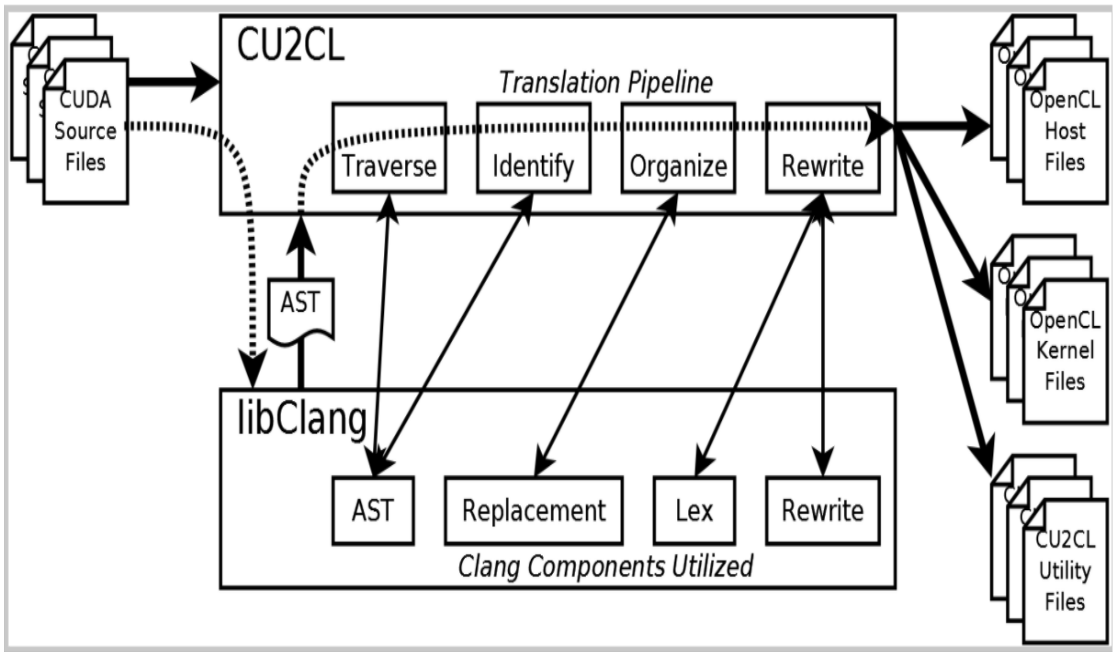
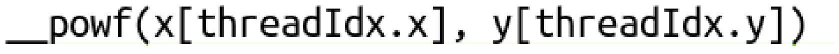
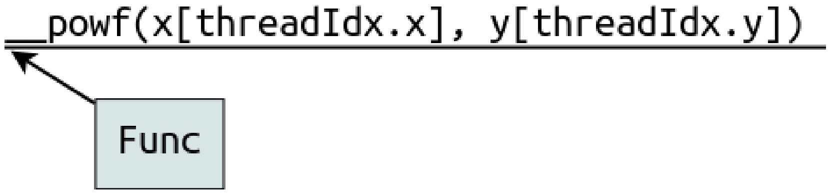


Figure 5.3 CU2CL architecture

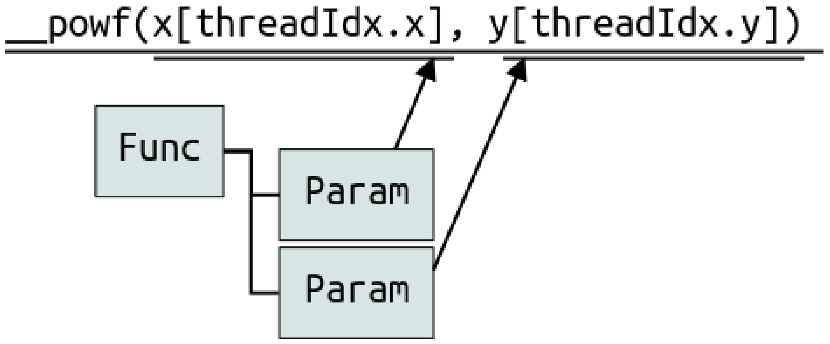
CU2CL performs a translation process called "AST-Driven, String Based Translation", which uses the AST generated by Clang to identify sections of source code which contain CUDA code which has to be translated. After the identification, the translator recurses into each individual component of the code searching for further sub-code structures which may need translation. After that all the code has been checked, the translator performs highly localized string based rewrites of the CUDA code from the bottom up. In the following, an example of how it works explained through different images, directly taken from the CU2CL website[21]:


```
__powf(x[threadIdx.x], y[threadIdx.y])
```
A typical CUDA kernel math function, the native power.

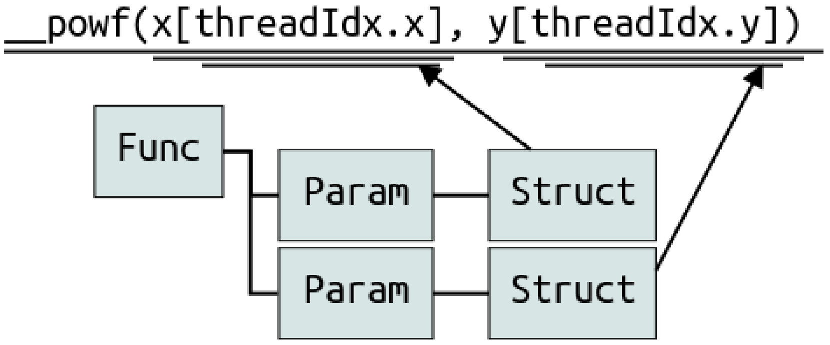
[21] http://chrec.cs.vt.edu/cu2cl/overview.php#translation



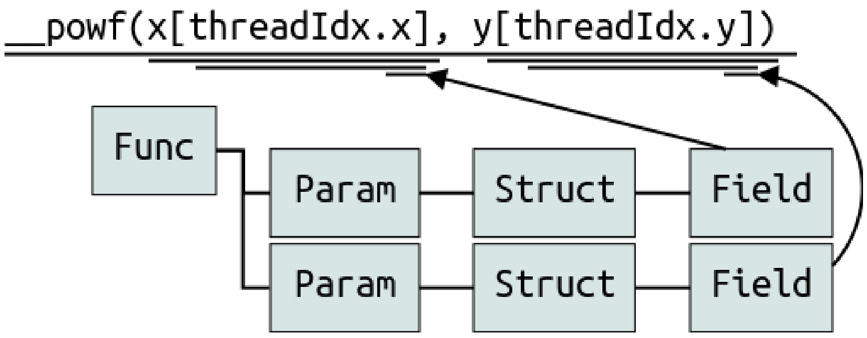
CU2CL identifies that a CUDA kernel function has to be translated.



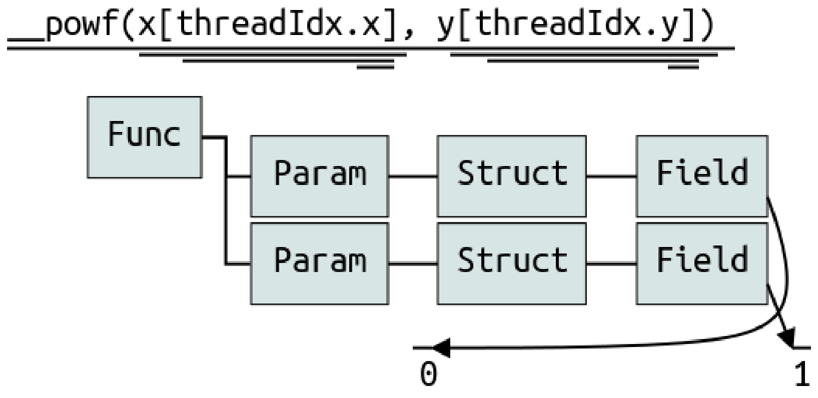
CU2CL identifies which are present two children, for its first and second parameter.



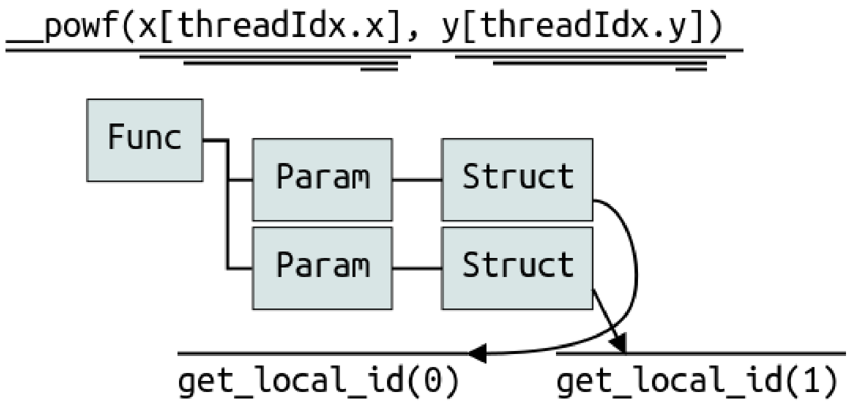
These are standard C arrays which do not need to be translated. Anyway, CU2CL has to check if their indices are CUDA structures which, in that case, are the CUDA specific threadIdx structures.



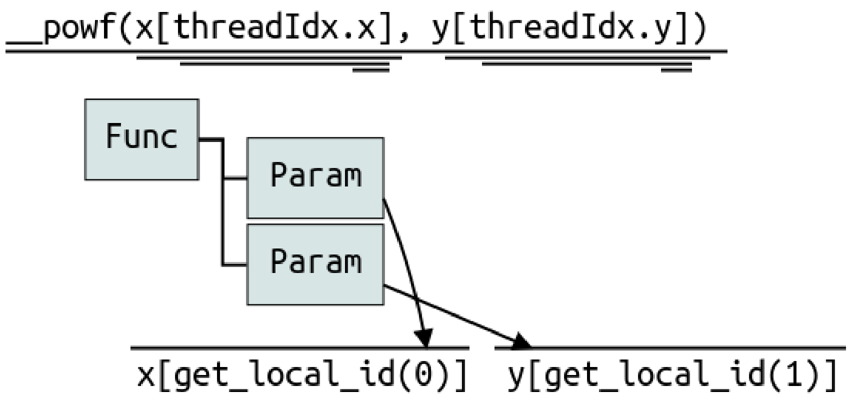
As the array indices were structs, CU2CL recurses one step further to determine the specific fields which are referenced.



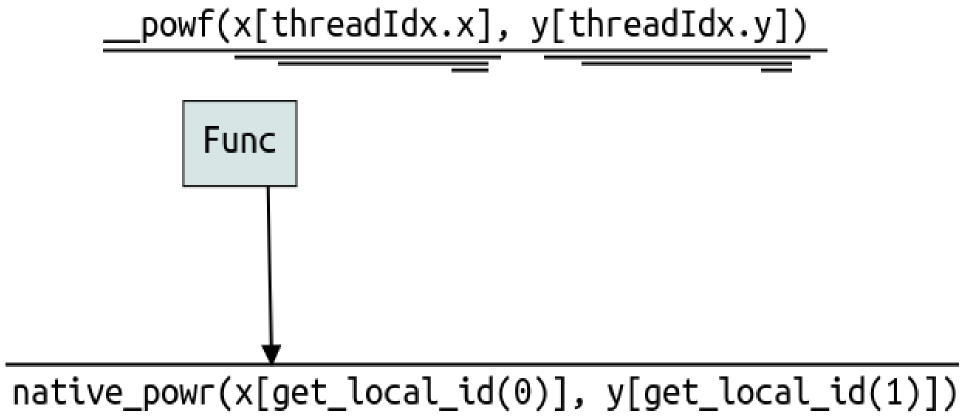
After reaching the AST leaf nodes, the translator must rewrite the text for each of these nodes that is either a CUDA structure or contains a child which is a CUDA structure. Upon returning from each recursive call the rewritten text corresponding to child nodes is integrated with the rewritten text from the current node, and returned to the parent. So, for the native power translation, CU2CL first rewrites the threadIdx fields for each of the arguments.



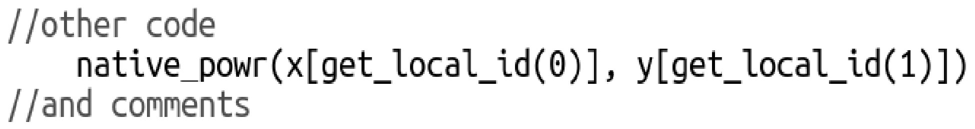
The rewritten zero and one are then passed upwards to their parent nodes, which rewrites the threadIdx structure into the corresponding OpenCL get_local_id call, using the zero and one as parameters to the call.



This rewritten structure is the passed upwards to the parent nodes associated with the x and y arrays. No special rewriting is performed on the arrays themselves, but new strings are returned to the parent node with the original CUDA threadIdx indices replaced.


__powf(x[threadIdx.x], y[threadIdx.y])
Func
native_powr(x[get_local_id(0)], y[get_local_id(1)])

Finally, after receiving the rewritten strings for both function parameters, CU2CL translates the _powf function itself.



```
//other code
    native_powr(x[get_local_id(0)], y[get_local_id(1)])
//and comments
```

Once the CUDA structure is fully translated it is then inserted into the source code as a direct replacement to the original structure, without modifying the surrounding code. By providing this highly localized translation mechanism, the original formatting and commenting in source code is preserved, significantly simplifying maintenance of the automatically translated OpenCL code.

In a work done by Wu Feng[22], it has been highlighted how using CU2CL (or in general automatic porting tools) is far better than manually porting the code. He showed that OpenCL has a too low level API compared to CUDA, so it is much easier to start with CUDA. He concluded that automatic translated code and manually translated code from CUDA to OpenCL yields the same performance and that OpenCL performance is not as good as CUDA as implementations are not as mature (but he also pointed out that the performance could be raised thanks to optimization).

On the other hand there is the manual porting of the code, which, in theory, allows to have always an updated code in correspondence to the last releases in such a way to have a final better result, but it takes time and it requires a certain knowledge in the porting itself.

Actually, already having the no bug code in CUDA is a great news, because you have already worked out how to split up the problem you had to solve to run effectively. Nevertheless some changes are needed. The first ones to be done are the ones about the different terms in the code which have to be changed, as we can see in the following:

[22] How To Run Your CUDA Program Anywhere - A Perspective from Virginia Tech's GPU-Accelerated HokieSpeed Cluster, W. Feng and M. Gardner, November 2011

| **CUDA terms** | **OpenCL terms** |
|---|---|
| **Terminology** | |
| Thread | Work-item |
| Thread block | Work-group |
| Global memory | Global memory |
| Constant memory | Constant memory |
| Shared memory | Local memory |
| Local memory | Private memory |
| **Qualifiers** | |
| _global_ *function* | _kernel *function* |
| _constant_ *variable declaration* | _constant *variable declaration* |
| _device_ *variable declaration* | _global *variable declaration* |
| _shared_ *variable declaration* | _local *variable declaration* |
| **Indexing** | |
| gridDim | get_num_groups() |
| blockDim | get_local_size() |
| blockIdx | get_group_id() |
| threadIdx | get_local_id |
| **Synchronization** | |
| _syncthreads() | barrier() |
| _threadfence_block() | mem_fence() |
| **Import API Objects** | |
| CUdevice | cl_device_id |
| CUcontext | cl_context |
| CUmodule | cl_program |
| CUfunction | cl_kernel |
| CUdeviceptr | cl_mem |
| **Important API Calls** | |
| cuDeviceGet() | clGetContextInfo() |
| cuCtxCreate() | clCreateContextFromType() |
| cuModuleGetFunction() | clCreateKernel() |

| | |
|---|---|
| cuMemAlloc() | clCreateBuffer() |
| cuMemcpyHtoD() | clEnqueueWriteBuffer() |
| cuMemcpyDtoH() | clEnqueueReadBuffer() |
| cuParamSeti() | clSetKernelArg() |
| cuLaunchGrid() | clEnqueueNDRangeKernel() |
| cuMemFree() | clReleaseMemObj() |

Table 5.1 CUDA vs OpenCL terminology changes

Porting to OpenCL requires to change both host and device code. By default, CUDA initializes the GPU automatically, while OpenCL requires an explicit device initialization, due to its portability, so the code must know which device has to be used. Let's start from the host program.

The host program is the code which runs in the host to setup the environment for the OpenCL program and to create and manage kernels. In general, it is created in this way: first, you define the platform where you are operating, so how many and which devices are used, the context and the queues; then, you create and build the program (with dynamic libraries for kernels); after that, you have to setup the memory object and to define the kernel (to link arguments to kernel functions); finally, you submit the commands (and so it happens the transfer of memory objects and the execution of kernels).

When you define the platform you have to build up also the context and the queues. The context is included in the program object, which includes also the program kernel source or binary and the list of target devices and build options. For that reason, you have to use the C API build process to create a program object, either clCreateProgramWithSource() or clCreateProgramWithBinary(). Note that OpenCL uses runtime compilation, due to the fact that the details of the target devices are not known from the beginning. The queues (command-queues) include the kernel execution, the memory object management and the synchronization. They are a main feature of the program, because the only way to send command to devices is through them. Each command-queue points to a single device within a context and multiple command-queues can feed a single device (e.g. to define independent streams of commands which do not require synchronization). They can be configured in two different ways to manage how commands execute. In the in-order queues the commands are enqueued and complete in the order they appear in the program, while in the out-of-order queues the commands are enqueued in program-order but can execute and complete in any order. Then you have to create and build the program, better if including some error messages to perform a check on the creation itself. The following step is to setup memory objects; the difference between CUDA and OpenCL to perform this operation are listed in the following:

| | CUDA | OpenCL |
|---|---|---|
| Allocate | float* d_x; cudaMalloc(&d_x, sizeof(float)*size); | cl_mem d_x= clCreateBuffer(context, CL_MEM_READ_WRITE, sizeof(float)*size, NULL, NULL); |
| Host to Device | cudaMEMcpy(s_x, h_x, sizeof(float)*size, cudaMemcpyHosttoDevice); | clEnqueueWriteBuffer(queue, d_x, CL_TRUE, 0, sizeoff(float)*size, h_x, 0, NULL, NULL); |
| Device to Host | cudaMemcpy(h_x, d_x, sizeof(float)*size, cudaMemcpyDeviceToHost); | clEnqueueReadBuffer(queue, d_x, CL_TRUE, 0, sizeof(float)*size, h-x, 0, NULL, NULL); |

Table 5.2 Memory objects

The following step is to define the kernel and, in contrast with CUDA where you have to specify the number of thread blocks and threads for each block, in OpenCL you have to specify the problem size and, optionally, the number of work-items for each work-group. The last step is to enqueue commands: once again, here is a table with the differences between CUDA and OpenCL.

| CUDA | OpenCL |
|---|---|
| dim3 threads_per_block(30,20); dim3 num_blocks(10,10); kernel<<<num_blocks, threads_per_block>>>(); | const size_t global[2]={300,200}; const size_t local[2]={30,20}; clEnqueueNDRangeKernel(queue, &kernel, 2, 0, &global, &local, 0, NULL, NULL); |

Table 5.3 Enqueue commands

The host program is now complete and working. The last step is to modify the device program, which should be easier than the host one. This is true for simple programs, but usually it is the same also for more complex ones. To better understand and highlight the above concepts, I will attach here an example of a vector addition done from Piero Lanucara, from SCAI (SuperComputing Applications and Innovation) in a work called "From CUDA to OpenCL". The vector addition is a good example, such as it stands for parallel computing as the program “hello world” stands for basic language programming. At first, I will attach CUDA host and device program (this last one highlighted in bold inside the program), then the separate OpenCL host and device programs.

## CUDA host and device vector:

```
/**
* Copyright 1993-2015 NVIDIA Corporation. All rights reserved.
*
* Please refer to the NVIDIA end user license agreement (EULA) associated
* with this source code for terms and conditions that govern your use of
* this software. Any use, reproduction, disclosure, or distribution of
* this software and related documentation outside the terms of the EULA
* is strictly prohibited.
*
*/

/**
* Vector addition: C = A + B.
*
* This sample is a very basic sample that implements element by element
* vector addition.
*/

#include <stdio.h>

// For the CUDA runtime routines (prefixed with "cuda_")
#include <cuda_runtime.h>
/**
* CUDA Kernel Device code
*
* Computes the vector addition of A and B into C. The 3 vectors have the same
* number of elements numElements.
*/
__global__ void
vectorAdd(const float *A, const float *B, float *C, int numElements)
{
        int i = blockDim.x * blockIdx.x + threadIdx.x;

        if (i < numElements)
        {
                C[i] = A[i] + B[i];
        }
}
/**
* Host main routine
*/
int
main(void)
{
        // Error code to check return values for CUDA calls cudaError_t err = cudaSuccess;

        // Print the vector length to be used, and compute its size int numElements = 50000;
        size_t size = numElements * sizeof(float);
        printf("[Vector addition of %d elements]\n", numElements);

        // Allocate the host input vector A
        float *h_A = (float *)malloc(size);
```

```
// Allocate the host input vector B
float *h_B = (float *)malloc(size);

// Allocate the host output vector C
float *h_C = (float *)malloc(size);

// Verify that allocations succeeded
if (h_A == NULL || h_B == NULL || h_C == NULL)
{
        fprintf(stderr, "Failed to allocate host vectors!\n");
        exit(EXIT_FAILURE);
}

// Initialize the host input vectors
for (int i = 0; i < numElements; ++i)
{
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
}

// Allocate the device input vector A float *d_A = NULL;
err = cudaMalloc((void **)&d_A, size);

if (err != cudaSuccess)
{
        fprintf(stderr, "Failed to allocate device vector A (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
}

// Allocate the device input vector B
float *d_B = NULL;
err = cudaMalloc((void **)&d_B, size);

if (err != cudaSuccess)
{
        fprintf(stderr, "Failed to allocate device vector B (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
}

// Allocate the device output vector C
float *d_C = NULL;
err = cudaMalloc((void **)&d_C, size);

if (err != cudaSuccess)
{
        fprintf(stderr, "Failed to allocate device vector C (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
}

// Copy the host input vectors A and B in host memory to the device input vectors in device memory
printf("Copy input data from the host memory to the CUDA device\n");
err = cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);

if (err != cudaSuccess)
{
        fprintf(stderr, "Failed to copy vector A from host to device (error code %s)!\n",
        cudaGetErrorString(err));
        exit(EXIT_FAILURE);
}

err = cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);
```

```
if (err != cudaSuccess)
{
        fprintf(stderr, "Failed to copy vector B from host to device (error code %s)!\n",
        cudaGetErrorString(err));
        exit(EXIT_FAILURE);
}

// Launch the Vector Add CUDA Kernel
int threadsPerBlock = 256;
int blocksPerGrid =(numElements + threadsPerBlock - 1) / threadsPerBlock;
printf("CUDA kernel launch with %d blocks of %d threads\n", blocksPerGrid, threadsPerBlock);
vectorAdd<<>>(d_A, d_B, d_C, numElements);
err = cudaGetLastError();

if (err != cudaSuccess)
{
        fprintf(stderr, "Failed to launch vectorAdd kernel (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
}

// Copy the device result vector in device memory to the host result vector in host memory.
printf("Copy output data from the CUDA device to the host memory\n");
err = cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

if (err != cudaSuccess)
{
        fprintf(stderr, "Failed to copy vector C from device to host (error code %s)!\n",
        cudaGetErrorString(err));
        exit(EXIT_FAILURE);
}

// Verify that the result vector is correct
for (int i = 0; i < numElements; ++i)
{
        if (fabs(h_A[i] + h_B[i] - h_C[i]) > 1e-5)
        {
                fprintf(stderr, "Result verification failed at element %d!\n", i);
                exit(EXIT_FAILURE);
        }
}

printf("Test PASSED\n");

// Free device global memory
err = cudaFree(d_A);

if (err != cudaSuccess)
{
        fprintf(stderr, "Failed to free device vector A (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
}

err = cudaFree(d_B);

if (err != cudaSuccess)
{
        fprintf(stderr, "Failed to free device vector B (error code %s)!\n", cudaGetErrorString(err));
        exit(EXIT_FAILURE);
}

err = cudaFree(d_C);
```

```
	if (err != cudaSuccess)
	{
		fprintf(stderr, "Failed to free device vector C (error code %s)!\n", cudaGetErrorString(err));
		exit(EXIT_FAILURE);
	}

	// Free host memory
	free(h_A);
	free(h_B);
	free(h_C);

	// Reset the device and exit
	// cudaDeviceReset causes the driver to clean up all state. While not mandatory in normal operation, it is a
	// good practice. It is also needed to ensure correct operation when the application is being profiled. Calling
	// cudaDeviceReset causes all profile data to be flushed before the application exits

	err = cudaDeviceReset();

	if (err != cudaSuccess)
	{
		fprintf(stderr, "Failed to deinitialize the device! error=%s\n", cudaGetErrorString(err));
		exit(EXIT_FAILURE);
	}

	printf("Done\n");
	return 0;
}
```

## OpenCL host vector:

```
// Fill vectors a and b with random float values
int i = 0;
int count = LENGTH;
for(i = 0; i < count; i++){
h_a[i] = rand() / (float)RAND_MAX;
h_b[i] = rand() / (float)RAND_MAX;
}

// Set up platform and GPU device

cl_uint numPlatforms;

// Find number of platforms

err = clGetPlatformIDs(0, NULL, &numPlatforms);
checkError(err, "Finding platforms");
if (numPlatforms == 0)
{
	printf("Found 0 platforms!\n");
	return EXIT_FAILURE;
}

// Get all platforms
cl_platform_id Platform[numPlatforms];
err = clGetPlatformIDs(numPlatforms, Platform, NULL);
checkError(err, "Getting platforms");

// Secure a GPU
for (i = 0; i < numPlatforms; i++)
{
	err = clGetDeviceIDs(Platform[i], DEVICE, 1, &device_id, NULL);
```

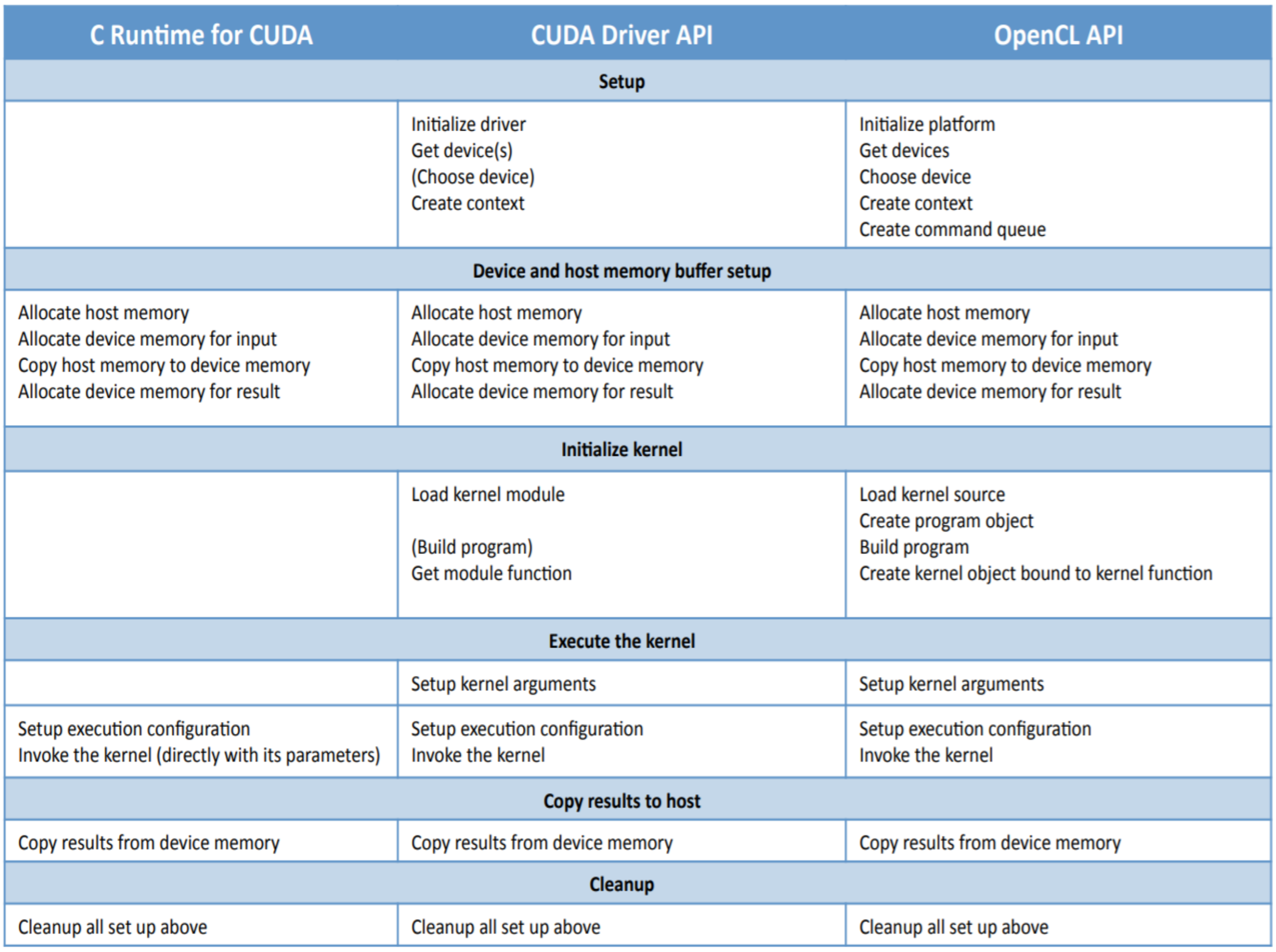
```
		if (err == CL_SUCCESS)
		{
			break;
		}
	}

	if (device_id == NULL)
		checkError(err, "Finding a device");

	err = output_device_info(device_id);
	checkError(err, "Printing device output");

	// Create a compute context
	context = clCreateContext(0, 1, &device_id, NULL, NULL, &err);
	checkError(err, "Creating context");

	// Create a command queue
	commands = clCreateCommandQueue(context, device_id, 0, &err);
	checkError(err, "Creating command queue");

	// Create the compute program from the source buffer
	program = clCreateProgramWithSource(context, 1, (const char **) & KernelSource, NULL, &err);
	checkError(err, "Creating program");

	// Build the program
	char options[] = "-cl-mad-enable";
	err = clBuildProgram(program, 0, NULL, options, NULL, NULL);
	if (err != CL_SUCCESS)
	{
		size_t len;
		char buffer[2048];
		printf("Error: Failed to build program executable!\n%s\n", err_code(err));
		clGetProgramBuildInfo(program, device_id, CL_PROGRAM_BUILD_LOG, sizeof(buffer), buffer, &len);
		printf("%s\n", buffer);
		return EXIT_FAILURE;
	}

	// Create the compute kernel from the program
	ko_vadd = clCreateKernel(program, "vadd", &err);
	checkError(err, "Creating kernel");

	// Create the input (a, b) and output (c) arrays in device memory
	d_a = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * count, NULL, &err);
	checkError(err, "Creating buffer d_a");

	d_b = clCreateBuffer(context, CL_MEM_READ_ONLY, sizeof(float) * count, NULL, &err);
	checkError(err, "Creating buffer d_b");

	d_c = clCreateBuffer(context, CL_MEM_WRITE_ONLY, sizeof(float) * count, NULL, &err);
	checkError(err, "Creating buffer d_c");

	// Write a and b vectors into compute device memory
	err = clEnqueueWriteBuffer(commands, d_a, CL_TRUE, 0, sizeof(float) * count, h_a, 0, NULL, NULL);
	checkError(err, "Copying h_a to device at d_a");

	err = clEnqueueWriteBuffer(commands, d_b, CL_TRUE, 0, sizeof(float) * count, h_b, 0, NULL, NULL);
	checkError(err, "Copying h_b to device at d_b");

	// Set the arguments to our compute kernel
	err = clSetKernelArg(ko_vadd, 0, sizeof(cl_mem), &d_a);
	err |= clSetKernelArg(ko_vadd, 1, sizeof(cl_mem), &d_b);
	err |= clSetKernelArg(ko_vadd, 2, sizeof(cl_mem), &d_c);
```

```
err |= clSetKernelArg(ko_vadd, 3, sizeof(unsigned int), &count);
checkError(err, "Setting kernel arguments");

double rtime = wtime();

// Execute the kernel over the entire range of our 1d input data set
// letting the OpenCL runtime choose the work-group size
global = count;
err = clEnqueueNDRangeKernel(commands, ko_vadd, 1, NULL, &global, NULL, 0, NULL, NULL);
checkError(err, "Enqueueing kernel");

// Wait for the commands to complete before stopping the timer
err = clFinish(commands);
checkError(err, "Waiting for kernel to finish");

rtime = wtime() - rtime;
printf("\nThe kernel ran in %lf seconds\n",rtime);

// Read back the results from the compute device
err = clEnqueueReadBuffer( commands, d_c, CL_TRUE, 0, sizeof(float) * count, h_c, 0, NULL, NULL );
if (err != CL_SUCCESS)
{
        printf("Error: Failed to read output array!\n%s\n", err_code(err));
        exit(1);
}
```

## OpenCL device vector:

```
const char *KernelSource = "\n" \
"#pragma OPENCL EXTENSION cl_nv_compiler_options :
enable          \n" \
"_kernel void vadd(             \n" \
"_global float* a,              \n" \
"_global float* b,              \n" \
"_global float* c,              \n" \
" const unsigned int count)     \n" \
"{                              \n" \
" int i = get_global_id(0);     \n" \
" if(i < count)                 \n" \
" c[i] = a[i] + b[i];           \n" \
"}                              \n" \
"\n";
//------------
```

We have just seen a practical example of how the porting of a CUDA code into an OpenCL one has to be done. The presented example is very easy, but it can highlight at best the changes which have to be done in the software, which could appear quite simple but for longer and more difficult software are challenging, mostly due to the fact that a manual porting could probably lead to many typing errors. In the following, a table to compare the differences in the usage of the host API that can be also be seen in the code above.



| C Runtime for CUDA | CUDA Driver API | OpenCL API |
|---|---|---|
| | **Setup** | |
| | Initialize driver<br>Get device(s)<br>(Choose device)<br>Create context | Initialize platform<br>Get devices<br>Choose device<br>Create context<br>Create command queue |
| | **Device and host memory buffer setup** | |
| Allocate host memory<br>Allocate device memory for input<br>Copy host memory to device memory<br>Allocate device memory for result | Allocate host memory<br>Allocate device memory for input<br>Copy host memory to device memory<br>Allocate device memory for result | Allocate host memory<br>Allocate device memory for input<br>Copy host memory to device memory<br>Allocate device memory for result |
| | **Initialize kernel** | |
| | Load kernel module<br><br>(Build program)<br>Get module function | Load kernel source<br>Create program object<br>Build program<br>Create kernel object bound to kernel function |
| | **Execute the kernel** | |
| | Setup kernel arguments | Setup kernel arguments |
| Setup execution configuration<br>Invoke the kernel (directly with its parameters) | Setup execution configuration<br>Invoke the kernel | Setup execution configuration<br>Invoke the kernel |
| | **Copy results to host** | |
| Copy results from device memory | Copy results from device memory | Copy results from device memory |
| | **Cleanup** | |
| Cleanup all set up above | Cleanup all set up above | Cleanup all set up above |

Table 5.4 Host API usage compared

Transporting the above analysis into the SAR focusing system software, there is the possibility to reach the portability achievement. It would be interesting to compare the two software at this state to see which one obtains the better performances, but maybe this reasoning is fallacious. I will explain myself better. To compare the two software under a performance point of view, we would have both of them running on the same architectures, so Nvidia's ones because we are obliged by CUDA limitations. From that perspective, obviously CUDA would exit victorious, due to all the native arrangements made ad hoc for its GPUs. So, they can not simply be compared, but they are both good GPU's programming platform which can do their best in different situations. If we have the need to program for different vendor's architecture, then OpenCL is the better choice to take. If we aim to have the maximum performances and we are comfortable in using Nvidia's architecture, then the better choice is CUDA. Now we can move on the implementations which could improve the actual work.

### 5.3.2 Modifying the code

As said at the beginning of subchapter 5.2, there are some improvements which could be applied to the actual software, thanks to the improvements in technology, not only for the hardware part, so the GPU itself, but also for the updates to the APIs used to programming the environment. The listed improvements (using a pipeline for memory transfers and a pipeline among kernels, doing the padding of the image not only on powers of two, using the new cuFFT library which allows to do the stride also in the vertical direction, and using more than one GPU at the same time) are all correlated to CUDA platform, and it is not taken for granted that the same changes could be applied using OpenCL.

Let's take the first two improvements, so using a pipeline for memory transfers and one among kernels. As we can learn directly from Nvidia developer's website, this is possible with CUDA. Some rules have to be followed and some constraints have to be applied, I quote them as it follows:

- The device must be capable of "concurrent copy and execution". This can be queried from the deviceOverlap field of a cudaDeviceProp struct, or from the output of the deviceQuery sample included with the CUDA SDK/Toolkit. Nearly all devices with compute capability 1.1 and higher have this capability.
- The kernel execution and the data transfer to be overlapped must both occur in different, non-default streams.
- The host memory involved in the data transfer must be pinned memory.

The resulting code is then obtained breaking up an array of a certain size N into many blocks of streamSize elements, which can be processed separately since the kernel operates separately on each one of them. The number of [non-default] streams used us nStreams=N/streamSize. Different approaches can be found to implement the domain decomposition of the data and processing as, for example, looping over all the operations for each block of the array.

```
for (int i = 0; i < nStreams; ++i)
{
         int offset = i * streamSize;
        cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice, stream[i]);
        kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
        cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost, stream[i]);
}
```

Or, for example, batching similar operations together, issuing all the host to device transfers first, then all kernel launches, and finally all device to host transfers.

```
for (int i = 0; i < nStreams; ++i)
{
        int offset = i * streamSize;
        cudaMemcpyAsync(&d_a[offset], &a[offset], streamBytes, cudaMemcpyHostToDevice,
        cudaMemcpyHostToDevice, stream[i]);
}

for (int i = 0; i < nStreams; ++i)
{
        int offset = i * streamSize;
        kernel<<<streamSize/blockSize, blockSize, 0, stream[i]>>>(d_a, offset);
}

for (int i = 0; i < nStreams; ++i)
{
        int offset = i * streamSize;
        cudaMemcpyAsync(&a[offset], &d_a[offset], streamBytes, cudaMemcpyDeviceToHost,
        cudaMemcpyDeviceToHost, stream[i]);
}
```

The full code done by Mark Harris can be found on GitHub[23]. On the other hand, with OpenCL, the preferred way is to create separate command queues for data transfers and compute and to use events to create dependencies among steps. Then, at each iteration, have to be enqueued the:

- Upload of N+1 (the fragment of data which will be processed)
- Processing of N (the actual fragment)
- Download of N-1 (the old fragment is sent back)

So in both APIs it is possible to do this step, with some differences between the two platforms. Another improvement which could be done is using the new cuFFT library which allows to do the stride also in the vertical direction without transposing the data. Due to the fact that azimuth transformations are used quite extensively in the current implementation and constitute a main bottleneck, there should be a substantial decrease in running time by adopting the new toolkit. In CUDA this is directly appliable, while OpenCl has another library for operating with Fast Fourier transforms, which is called clFFT. Also this library supports vertical strides, as reported in the library description[24] "clFFT expects all multi-dimensional input passed to it to be in row-major format. This is compatible with C-based languages. However, clFFT is very flexible in the organization of the input and output data, and it accepts input data by letting you specify a stride for each dimension. This feature can be used to process data in column major arrays and other non-contiguous data formats. See clfftSetPlanInStride() and clfftSetPlanOutStride()." A further change in the software should be the possibility to use (so communicate and interact) multiple GPUs (so one host and multiple devices), in such a way to divide the whole work upon the devices to speed up the entire

[23] https://github.com/NVIDIA-developer-blog/code-samples/blob/master/series/cuda-cpp/overlap-data-transfers/async.cu

[24] https://clmathlibraries.github.io/clFFT/

process. *Divide et impera* approach. This could be done by both platforms; in particular, in OpenCL the way to follow is to create a context, fetch all devices that you need, place them in an array, and use that array for building your kernels, and create one command_queue for every device in them. OpenCL (clBuildProgram) supports in fact multi-device compilation, so the whole job can be split among devices. The last point is about doing the padding of the image not only on powers of two. This point deserves an initial consideration, which is that it is necessary to do a study based on the right dimension based on the FFT performance. In theory, clFFT supports lengths that are any combination of powers of 2, 3, 5, 7, as well as cuFFT. The fact is that, in any case, FFT of any length is information preserving, but programmers tend to use radix 2 transforms due to they easiness. So, it is possible to use others padding compared to the one powered by two, but, first of all, it is necessary to see if the performance increases and, then, if the actual work works well, there is not a real necessity to change it.

# Conclusion

This thesis has been conceived as a work to highlight an actual existent software and see what types of changes could be applied to it to improve its performance and portability. The initial consideration are encouraging, and some work could be made on it to actual realize a new software with the new features. Which one to choose between CUDA and OpenCL? From the different works seen in Chapter three, in particular the last one, which has been massively explained, it has been shown that both platforms are able to obtain similar performance results under some constraints. So the real question is: which one to choose between CUDA and OpenCL for the SAR focusing system environment? On the one hand, choosing OpenCL may bring the advantage of an heterogeneous system, which could be the winning point due to the possibility to configure the best system possible to reach the best performance, and also the possibility to run the system onto different computing configurations, such as different computers or the same computer with different hardware configurations. On the other hand, choosing CUDA may bring the advantage of running on a platform which has great performance results and a lot of programmers which already know how to write it in a very efficient way. It is also true that, as it is written in Chapter three, CUDA works very well with the Fast Fourier Transform library, compared to OpenCL. The SAR focusing system uses a lot this library, which is of main importance in both the algorithm used for the focusing (the Range Doppler Algorithm and the ω-k algorithm). This last point is a very strong reason to prefer CUDA platform instead of OpenCL one. It is also true that automatic porting tools are becoming more efficient and more powerful. In particular, the idea of using already written CUDA software and the porting it into OpenCL could be a very efficient way to take the best features of both platforms. The growing of this tools could be, in the future, the turning point in the choice of which platform would be better to use. In conclusion, it is a matter of choice of what is really needed between the two different platforms, taking into account also the differences between them, and maybe considering the idea to use an automatic porting tool (which are becoming better and better) to use CUDA and porting it into OpenCL, in such a way to have already expert programmers able to write efficient codes, with also the strong point of CUDA platform and libraries, but having also the possibility to run that code on different architectures or an architecture with different hardware configurations.

# References

- A. W. Love, "In Memory of Carl A. Wiley", IEEE Antennas and Propagation Society Newsletter, pp 17–18, June 1985
- Africa Ixmucane Flores-Anderson, Kelsey E. Herndon, Rajesh Bahadur Thapa, Emil Cherrington, “The SAR Handbook: Comprehensive Methodologies for Forest Monitoring and Biomass Estimation”, “Chapter 2: Spaceborne Synthetic Aperture Radar: Principles, Data Access, and Basic Processing Techniques”, April 2019
- Alexandros Papakonstantinou, Karthik Gururaj, John A. Stratton, Deming Chen, Jason Cong, Wen-Mei W. Hwu, “FCUDA: Enabling Efficient Compilation of CUDA Kernels onto FPGAs”, IEEE 7th Symposium on Application Specific Processors (SASP), 2009
- Ashu Rege, “An Introduction to Modern GPU Architecture”, Nvidia Corp., 2008
- ASI Agenzia Spaziale Italiana / Italian Space Agency, “COSMO-SkyMed System Description & User Guide”, Doc. No: ASI-CSM-ENG-RS-093-A, May 2007
- Bala Dhandayuthapani Veerasamy, G. M. Nasira, “Exploring the contrast on GPGPU computing through CUDA and OpenCL”, September 2014
- Bogdan Oancea, Tudorel Andrei, Raluca Mariana Dragoescu, “GPGPU Computing”, Challenges of the Knowledge Society. IT
- C. W. Sherwin, J. P. Ruina, and R. D. Rawcliffe, "Some Early Developments in Synthetic Aperture Radar Systems", IRE Transactions on Military Electronics, April 1962, pp. 111–115
- Claudio Passerone, Claudio Sanso, Riccardo Maggiora, "High Performance SAR Focusing Algorithm and Implementation", Department of Electronics and Telecommunications, Politecnico di Torino
- David A. Patterson, John L. Hennessy, “Computer Organization And Design, The Hardware/Software Interface: RISC-V Edition”, Morgan Kaufmann Publishers, 2018
- Denis Demidov, Karsten Ahnert, Karl Rupp, Peter Gottschling, “Programming CUDA and OpenCL: a case study using modern C++ libraries”, 2013
- FirePro Graphics AMD, “OpenCL$^{TM}$: The Future of Accelerated Application Performance Is Now”, 2011
- Giuseppe Bilotta, Emiliano Tramontana, Roberto Spina, “Porting (CUDA/OpenCL) del software MagFlow per la simulazione dei flussi lavici dell’Etna (MagFlow Porting Software (Cuda/OpenCL) for Simulation Lava Flows of Etna), July 2014
- Guido Masera, “ISA: Integrated Systems Architecture Part 3-C Processor architecture”, Politecnico di Torino, 2018-2019

- Håkon Kvale Stensland, “GPU & CUDA”, Simula Research Laboratory
- Hiroyuki Takizawa, Shoichi Hirasawa, Makoto Sugawara, Isaac Gelado, Hiroaki Kobayashi, Wen-mei W. Hwu, “Optimized Data Transfers Based on the OpenCL Event Management Mechanism”, Hindawi Publishing Corporation, Volume 2015, Article ID 576498
- How To Run Your CUDA Program Anywhere - A Perspective from Virginia Tech’s GPU-Accelerated HokieSpeed Cluster, W. Feng and M. Gardner, November 2011
- ISO/IEC 9899:TC2, WG14/N1124, Committee Draft, May 2005
- Jeff Larkin, “GPU Fundamentals”, Nvidia Corp., November 2016
- Jianbin Fang, Ana Lucia Varbanescu and Henk Sips, “A Comprehensive Performance Comparison of CUDA and OpenCL”, Parallel and Distributed Systems Group, Delft University of Technology (Delft, Netherlands), September 2011
- K. Karimi, N. G. Dickson, and F. Hamze, “A Performance Comparison of CUDA and OpenCL,” May 2010
- K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, “Evaluating Performance and Portability of OpenCL Programs,” in Proceedings of the Fifth international Workshop on Automatic Performance Tuning(iWAPT2010), (Berkeley, USA), June 2010
- M. M. Baskaran, U. Bondhugula, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, “A compiler framework for optimization of affine loop nests for gpgpus,” in Proceedings of the 22nd annual international conference on Supercomputing, ICS ’08, (New York, NY, USA), pp. 225–234, ACM, 2008
- Matt Harvey, “Experiences porting from CUDA to OpenCL”, Imperial College London CBBL IMIM, December 2009
- Matt Pharr, “GPU gems 2 : programming techniques for high performance graphics and general-purpose computation”, Randima Fernando series editor, April 2005
- Nvidia Corp., “OpenCL$^{TM}$ Programming Guide for the CUDA$^{TM}$ Architecture”, Version 4.2, September 2012
- NVIDIA Inc., NVIDIAs Next Generation CUDA Compute Architecture: Fermi, 2009
- Nvidia Inc., OpenCL Best Practices Guide, May 2010
- NVIDIA Inc., PTX: Parallel Thread Execution ISA Version 2.2, October 2010
- P. Du, R. Weber, P. Luszczek, S. Tomov, G. Peterson, and J. Dongarra, “From CUDA to OpenCL: Towards a Performance-portable Solution for Multi-platform GPU Programming,” tech. rep., Department of Computer Science, UTK, Knoxville Tennessee, September 2010
- Piero Lanucara, “From CUDA to OpenCL”, SuperComputing Applications and Innovation

- R. Amorim, G. Haase, M. Liebmann, and R. Weber dos Santos, “Comparing CUDA and OpenGL implementations for a Jacobi iteration,” pp. 22–32, June 2009
- R. van Nieuwpoort and J. Romein, “Correlating radio astronomy signals with Many-Core hardware,” International Journal of Parallel Programming, vol. 39, pp. 88–114, Feb. 2011
- R. Weber, A. Gothandaraman, R. J. Hinde, and G. D. Peterson, “Comparing Hardware Accelerators in Scientific Applications: A Case Study,” IEEE Transactions on Parallel and Distributed Systems, vol. 22, pp. 58–68, January 2011
- Sami Rosendahl, “CUDA and OpenCL API comparison”, Presentation for T-106.5800 Seminar on GPGPU Programming, Spring 2010
- T. I. Vassilev, “Comparison of several parallel API for cloth modelling on modern GPUs,” in Proceedings of the 11th International Conference on Computer Systems and Technologies and Workshop for PhD Students in Computing on International Conference on Computer Systems and Technologies, CompSysTech ’10, (New York, NY, USA), pp. 131–136, ACM, 2010

- http://chrec.cs.vt.edu/cu2cl/overview.php#translation
- http://www.cosmo-skymed.it/en/index.htm
- https://clmathlibraries.github.io/clFFT/
- https://developer.nvidia.com/blog/how-overlap-data-transfers-cuda-cc/
- https://developer.nvidia.com/cuda-zone
- https://en.wikipedia.org/wiki/C99
- https://en.wikipedia.org/wiki/COSMO-SkyMed
- https://en.wikipedia.org/wiki/CUDA
- https://en.wikipedia.org/wiki/General-purpose_computing_on_graphics_processing_units
- https://en.wikipedia.org/wiki/Graphics_processing_unit
- https://en.wikipedia.org/wiki/OpenCL
- https://en.wikipedia.org/wiki/Parallel_Thread_Execution
- https://en.wikipedia.org/wiki/Scoreboarding
- https://en.wikipedia.org/wiki/Stream_processing
- https://en.wikipedia.org/wiki/Synthetic-aperture_radar
- https://github.com/hughperkins/coriander
- https://github.com/NVIDIA-developer-blog/code-samples/blob/master/series/cuda-cpp/overlap-data-transfers/async.cu
- https://standards.ieee.org/standard/60559-2020.html
- https://www.amd.com/en/graphics/servers-solutions-rocm

- https://www.intel.com/content/www/us/en/products/docs/processors/cpu-vs-gpu.html
- https://www.khronos.org/opencl/
- https://www.khronos.org/registry/OpenCL/specs/2.2/html/OpenCL_API.html#_introduction
- https://www.nrcan.gc.ca/earth-sciences/geomatics/satellite-imagery-air-photos/satellite-imagery-products/educational-resources/9567