Towards Real-time SAR

Georgios Pinitas

TU Delft Delft University of Technology

Towards Real-time SAR

Master's Thesis in Embedded Systems

Parallel and Distributed Systems group
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology

Georgios Pinitas

1st July 2014

Author
Georgios Pinitas

Title
Towards Real-time SAR

MSc presentation
July 10^{th}, 2014

Graduation Committee

Prof. Dr. Ir. H. J. Sips (Chair)	Delft University of Technology
Dr. Ir. Ana Lucia Varbanescu (Supervisor)	Vrije Universiteit Amsterdam
M.P.G. (Mattern) Otten MSc	TNO
Dr. Przemyslaw Pawelczak	Delft University of Technology

Abstract

Nowadays, multiple different architectures exist, each one offering a performance boost on applications of different nature. Specialized and multicore computing is clearly mainstream, most application are ported or redesigned in order to expose any level of parallelism. Thus, new software and programming infrastructures have been introduced to help programmers exploit the benefits mutli-core and specialized computing have to offer.

Radars are widely used for a variety of reasons. Their ability to operate in almost every weather condition, day and night, makes them an attractive solution. Synthetic Aperture Radar (SAR) is a radar infrastructure capable of creating high resolution spatial images. Most SAR processing chains operate on the frequency domain for lower computational demands, but the multi-core era has made time-domain processing feasible. Time-domain SAR processing can be quite demanding in terms of computational power for large problem sizes but overcomes formulation problems that frequency-domain algorithms face.

In this thesis, we investigated the possibility of achieving real-time performance on a SAR system, operating on the time-domain. For these purposes, the basic processing chain of such a system is analysed, implemented and optimized.

Overall, we manage to achieve real-time performance for different output dimensions mainly targeting Unmanned Aerial Vehicles (UAVs) and airborne solutions that operate under computational and power constraints. Furthermore, a hardware mapping of the computational components is proposed for their efficient execution. Moreover, we observe that multiple data rate reduction operations can be performed and still retain an acceptable Quality of System (QoS). Finally, the back-projection algorithm is optimized achieving a peak performance of *40 Giga Back-projections per second* using a single hardware accelerator.

Preface

From the very beginning of my postgraduate studies I was particularly intrigued by High Performance and Heterogeneous Computing. I am really glad that I had the opportunity to do my thesis in a relevant topic which also included some aspects of Embedded Systems. Doing something that you really like can only be followed by joy, independent of the difficulties that may emerge. This thesis was performed in collaboration with TNO. I hope that this work will help radar engineers and researchers by giving insights for future radar designs.

The completion of this undertaking could not have been possible without the support of so many people. Foremost, I would like to express my deepest thanks to my supervisor and mentor *Dr. Ana Lucia Varbanescu*, for her continuous support and help. Her patience, guidance and knowledge helped me throughout my postgraduate studies and my thesis. One could not have imagined having a better advisor. Further, I wish to express my gratitude to *Mr. Matern Otten* from TNO, for his help on topics I had no relevant experience. Without his help and immense knowledge this thesis would not have been feasible. Moreover, I would like to thank *Mr. Miguel Caro Cuenca* and *Mr. Wouter Vlothuizen* for their insightful conversations and their every day support. I would like also to acknowledge *Dr. Henk Sips* and *Dr. Przemyslaw Pawelczak* for being part of my graduation committee. Finally, I would like to thank *my family and friends*, I feel really blessed to have them in my life. Needless to say that this thesis is dedicated to them.

Georgios Pinitas

Delft, The Netherlands
1st July 2014

Contents

List of Figures

Acronyms

QoS Quality of System

HDL Hardware Description Language

SIMD Single Instruction Multiple Data

SPMD Single Program Multiple Data

CPU Central Processing Unit

GPP General Purpose Processors

DSP Digital Signal Processor

ISA Instruction Set Architecture

MAC Multiply-accumulate

FFT Fast Fourier Transform

GPU Graphics Processing Unit

GPGPU General-purpose computing on Graphics Processing Unit

GPC Graphics Processor Cluster

FPGA Field-programmable Gate Array

ASIC Application Specific Integrated Circuit

API Application Programming Interface

OpenCL Open Computing Language

CUDA Compute Unified Device Architecture

OpenMP Open Multi-Processing

MPI Message Passing Interface

SAR Synthetic Aperture Radar

UAV Unmanned Aerial Vehicle

TDC Time-Domain Correlation

GMTI Ground Moving Target Indication

INS Inertial Navigation System

DBF Digital Beamforming

GPS Ground Positioning System

PGA Phase Gradient Algorithm

FMCW Frequency-Modulated Continuous-Wave

SRF Signal Repetition Frequency

AoS Array-of-Structures

SoA Structure-of-Arrays

PSNR Peak signal-to-noise ratio

Chapter 1

Introduction

1.1 Context

For many years, CPU designers used to achieve higher performance by mainly focusing on increasing the frequency of a system. Exploitation of frequency scaling became harder over the years due to physical limitations like the dramatical increase in power consumption and thermal dissipation. In order to fulfil the needs for higher performance out of a system, designers turned into exploiting the small transistor size by deploying multiple cores with lower frequencies on a single chip. This led to both an increase in performance but also a relative smaller and more controllable power consumption.

Applications also used to drive the design of hardware architectures in order to increase their performance; a glaring example are DSPs and Application Specific Integrated Circuits (ASICs). But the mutlicore trend gave birth to even more. The type of parallelism an application could express led to the design of hardware architectures with varying core complexity and core number.

Many practical problems emerged by multicore and specialized computing. The most important one was the dramatic shift that had to take place in the programming area. This led into renovating existing or creating new programming languages capable of expressing parallelism and as a result exploiting the benefits multicores have to offer.

Radars are widely used nowadays for a variety of reasons. Their ability to operate in almost every weather condition, day and night, makes them an attractive solution. SAR is a form of radar deployed on moving platforms like satellites and aircrafts and is capable of creating high resolution spatial images. The computational demands of such a radar system are high, thus achieving real-time performance, at first glance at least, seems impossible.

In this thesis, we focus on analysing the processing steps of a SAR system and investigating the possibility of achieving real-time performance under multiple processing scenarios. Our main interest lies in achieving real-time performance for UAVs and airborne solutions in general, that operate under computational and

power constraints.

1.2 Research Questions

This section presents the research questions that this thesis aims to answer. The main research addressed is:

Can the proposed SAR processing chain achieve real-time performance under computational and power constraints?

More research questions that this thesis addresses are:

- What are the current hardware and software trends in heterogeneous computing?
- What are the main processing modules of a SAR systems and what are their requirements in terms of computational power?
- How can multi-channel back-projection be efficiently parallelized and what are its bottlenecks and limitations?
- Is the performance of different optimisations affected by problem size and input data variability?
- Can the computational needs of the application be reduced and still retain an acceptable high QoS?
- Is there an efficient hardware mapping and a custom architecture for the processing modules of the system?

1.3 Contributions

The main contributions of this thesis are the following:

- We survey current trends in terms of hardware and software.
- We analyse the processing chain of a state of the art SAR system, investigating in detail all the processing steps and proposing efficient implementations of each one separately.
- We investigate how the optimisations effect change with problem scalability and input data variability.
- We propose a module mapping along with a custom hardware architecture where each module of the processing chain is deployed on the appropriate hardware architecture to achieve higher performance and lower power consumption.

1.4 Thesis Organization

This thesis is organized as follows. Chapter 2 provides all the background information required for further understanding the contents of the thesis. It presents the concept of parallelism along with the models used for expressing parallel computations. Moreover, a thorough analysis of the current hardware diversity takes place, along with the parallel programming language trends that exist nowadays. Finally, an introduction in SAR and radar systems in general takes place.

Chapter 3 lists the distinct processing modules that a SAR system requires nowadays. In addition, the specifications of the experimental setup that was used are presented.

Chapters 4 and 5 present and analyse the multi-channel back-projection algorithm, the most compute intensive component of the the SAR system. Different parallel implementations are shown along with the optimization steps. In addition, a multi-node multi-GPU implementation is analysed and a model is proposed.

In Chapter 6 two techniques are analysed specialized in reducing the data rate of the radar system and as a result the computational requirements of compute intensive processing steps of the processing chain.

Chapter 7 describes the autofocus algorithm that we used for correcting large motion errors.

In Chapter 8 we propose a custom architecture, specialized for the presented software architecture. Moreover, three different possible processing scenarios are analysed along with their performance and limitations.

Finally, our concluding remarks and future work directions are gathered in Chapter 9

Chapter 2

Background

In this chapter we present the basic concepts required to ease the understanding of the more complex material in this thesis. Specifically, we discuss parallelism models, hardware trends, programming models and the basic principles behind SAR.

2.1 Parallelism

Due to the the explosive trend of multi-core architectures, the urge of exploration and exploitation of the benefits that they have to offer is inevitable. Clearly, exploiting hidden or evident *parallelism* is one way to do so.

Programs or tasks may have computational workloads that can inherently run in parallel with one another or be remodelled to do so. The fundamental argument of parallelism is to run compute intensive tasks, that need rather a long amount of time, in parallel. More and more legacy application are being ported or remodelled in order to exploit such a possibility. But still, limitations exist to the level of parallelism that can be exploited.

2.1.1 Amdhal's Law

Blindly parallelizing sequential applications may lead to very limited performance gain. In 1967, G. Amdahl [1] stated that the performance gain of a parallel application over its sequential version is limited. This statement is widely known as *Amdahl's Law*.

In more detail, *Amdahl's Law* specifies that if s is the sequential part of the program and $1 - s$ the part of program that can be parallelized, then the maximum expected improvement S using P number of processors is:

$$S(P) = \frac{1}{s + \frac{1-s}{P}} \tag{2.1}$$

As a consequence, when $P \rightarrow \infty$, then the maximum possible speedup is bounded by the portion of the sequential part of the program. For this reason,

best candidates for parallelization tend to be portions of a program that account for a significant amount of the total execution time.

Finally, we have to point out that *Amdahl's Law* is a generalized argument, which over the years has been re-evaluated or complemented to adapt to the current multi-core era [23, 39]. Multiple models have been introduced based on scalable computing where an increase in computing power can lead to an increase in the problem size as well.

2.1.2 Models of Parallel Computation

Implementing a parallel version of an algorithm is considered, for many, the *"trivial"* part of the whole paralellization process. The most important part tends to be the deep understanding of the underlying algorithm along with the modelling of parallelism that can be exposed. For this reason, conceptually at least, we can classify the way a parallel program can be modelled in one the following categories [38]:

- **Farmer-Worker/Master-Slave:** In this model the workload can be distributed among algorithmically identical processes, each one taking different input arguments than the other. Usually, a Farmer thread or process is responsible into splitting and assigning work among the Worker threads/processes. The idle Worker threads are usually part of a thread pool, where the Farmer thread checks for available threads to assign work. When a Worker thread finishes its work, it becomes again part of the thread pool waiting for further assignments.

- **Divide and Conquer:** In this model, the problem can be divided recursively in smaller problems until they become small enough to be fast to solve. Finally, all the partial solutions are combined to solve the initial problem. It is important to point out that each sub-problem is theoretically independent of the others, which makes concurrent or parallel execution feasible.

- **Data Parallelism:** In this model, computation follows the data. The data are distributed in different computing nodes each one performing the same task (executing the same code) on its own data. When finished, the data are usually gathered on a single node forming the final solution.

- **Task Parallelism:** In this model the basic tasks/function can be isolated, each one being responsible for solving a specific sub-problem. Each task acts on specific arguments providing specific outputs. The dependency between these tasks can be represented using an acyclic directed graph. The way these tasks can be mapped onto a structure of processors is not trivial, thus two types of execution models can be concluded: data-driven and demand-driven. In the data-driven execution model, a task proceeds to execution when the input arguments are available, while in the demand-driven, execution of a task occurs when the provided data are required further in the pipeline.

- **Bulk-synchronous** [59]: In this model, the problem can be solved though an iterative process, until solution convergence. Every iteration consists of three distinct steps: a computation step, a communication step and finally a synchronisation step. Each process performs the computation steps asynchronously with respect to the other processes. When computations end, processes can exchange data if required. Finally, for consistency, a synchronisation step follows where all individual processes reach a certain point before proceeding further.

- **Hybdid:** Every possible combination of the above models falls in this category.

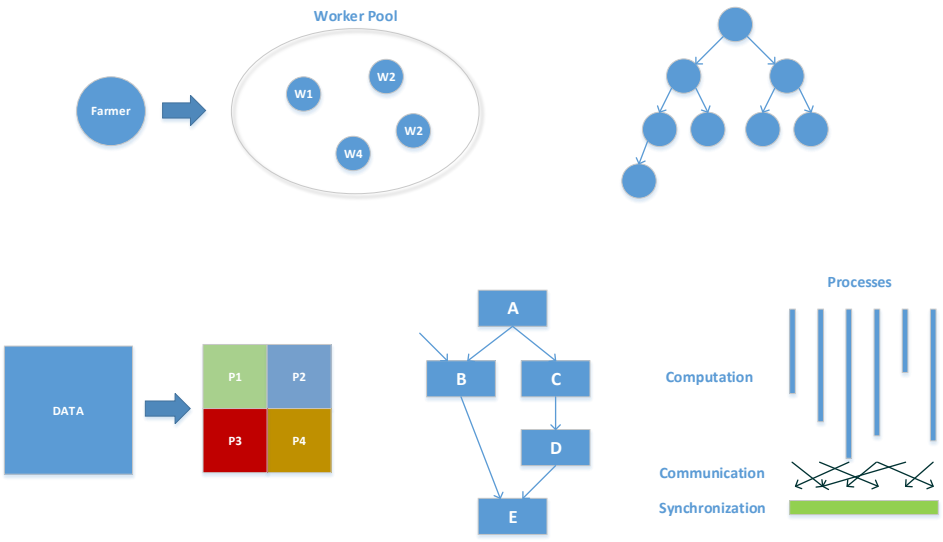


Figure 2.1: Parallel Computation Models [38, 66]

2.2 Heterogeneous Computing: A Hardware Approach

The need for specialized computing has led to the hardware diversity we experience nowadays. The fundamental premise of specialized or, as it is widely known, heterogeneous computing is having a task or a specialized workload to run on the most appropriate hardware platform in order to achieve the best performance and the lowest power consumption.

In this section, we describe the most important hardware categories that can be found nowadays in computing infrastructures.

2.2.1 General Purpose Processors (GPP)

All the conventional, general purpose, computer architectures which consist of one (rarely) or several homogeneous cores fall in this category.

They are shared-memory architectures with a multi-layer cache hierarchy and usually facilitate complex performance oriented hardware level optimisations (e.g. branch prediction, dynamic execution). They are mostly used as stand-alone processors, but newer designs, with significantly improved performance capabilities, have been created to be used as high performance coprocessors, like Knights Corner and Knights Landing (brand name Xeon Phi) [21, 13].

They target almost the whole market spectrum, from mobile devices to servers, and are offered by many industrial vendors like Intel, AMD, IBM and ARM.

Intel Haswell is the latest generation of Intel's GPPs [12, 46] [1].

A large of variety of models based on Haswell architecture are released or planned to be released soon, depending on the target market (e.g. High Performance, Low Power).

Most high performance designs come with four *physical cores*. Each one, due to Hyper-Threading Technology, can be seen as consisting of two individual cores, usually called *logical cores*. Hyper-Threading tries to exploit the fact that not all execution resources are being used during the execution of an instruction. By duplicating some resources, for example resources that keep the state of the program, gives the illusion of two distinct cores and the ability to run two independent processes or tasks in parallel.

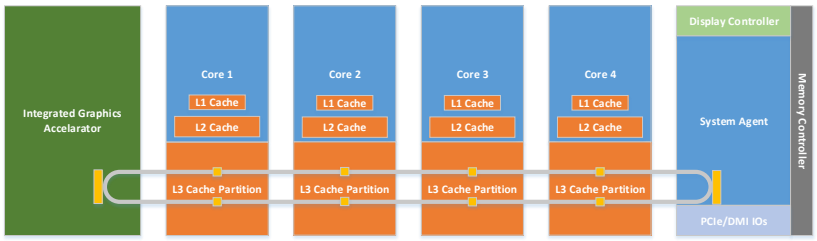


Figure 2.2: Haswell system architecture

Cores have private L1 and L2 8-way associative caches with sizes of 32 KB and 256 KB respectively. A L3 cache, shared between the processors, and has varying size depending on the model.

Haswell also supports out-of-order execution, being enhanced with multiple resources required for dynamic scheduling.

Both vector and integer registers are present in Haswell. In addition, multiple execution and address generation units exist. There is support for both scalar and

[1] At the time of writing, May 2014

SIMD operations.

Interesting is the fact Haswell contains an *Integrated Graphics Accelerator* connected to the bus ring, giving the ability to run graphics (and not only) tasks. The Graphics Accelerator was updated with compute capabilities. Different versions exist across the GPP models. The latest one is the Iris Pro 5200 version.

The L3 cache, along with the ring bus, run at a higher frequency than the cores in order to ensure GPU's high performance while keeping the cores in a low power state.

An example model of Haswell architecture is *i7-4770K* with and integrated Haswell GT2 graphics card is presented in Figure 2.3. The CPU itself has a peak performance of 448 Gflops (AVX with FMA) while the GPU 400 GFlops. The power consumption is close to 85 W [19].

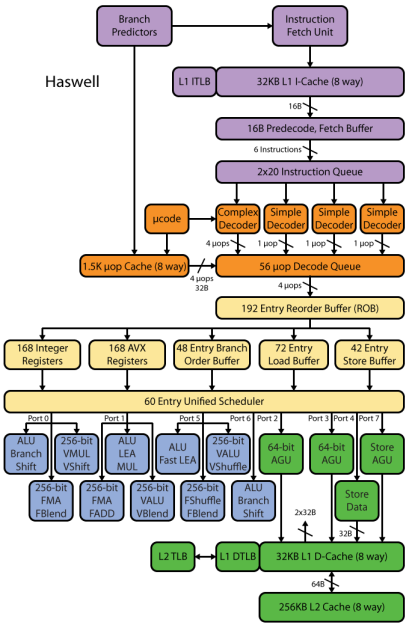


Figure 2.3: The Haswell micro-architecture [12]
(Courtesy of Real World Tech)

2.2.2 Digital Signal Processor (DSP)

A significant number of applications are signal processing oriented. Typical examples are sound or image applications. The need to accelerate such an application and the power constraints of mobile devices led to the design of DSPs. DSPs are processors designed for the efficient manipulation of digital signals.

A DSP's architecture differs in many ways from other processors. First of all, they have a signal processing oriented Instruction Set Architecture (ISA), which enables easier and faster implementation of digital signal processing applications. As required from these algorithms they perform a significant amount of operations

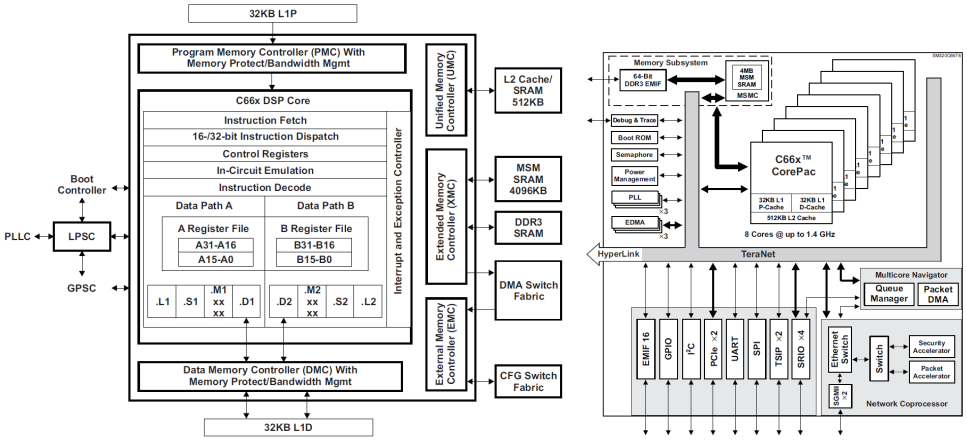
in parallel per work cycle. Such operations may be Multiply-accumulate (MAC) operations or even Fast Fourier Transform (FFT) loops. Moreover, in order to provide multiple operations per cycle and low power at the same time, they operate in lower frequencies.

Example companies designing DSP architectures are Texas Instruments, NXP and Freescale.

Keystone multicore DSP is one state-of-the-art multicore DSP architecture (see Figure 2.4) [53, 54] [1]. It consists of 8 DSP core packs each one running up to a frequency of 1GHz (see Figure 2.4b). Each DSP pack has a two-layer cache, L1 and L2 with 32KB and 512 KB respectively, and they all share a third one with a size of 4096 KB. Additionally, a memory interface exists for fast access to a DDR3 external memory. A large amount of standard interface is supported, some of the are PCIe2, Ethernet, UART and many more.

Each core has two datapaths with one register file and 4 functional units each (see Figure 2.4a). There are four kinds of functional units, $.L$ and $.S$ which perform general arithmetic and logical operations, $.M$ for multiply operations and finally $.D$ for transactions between the memory and the register file. Each functional unit can perform multiple operations per clock cycle depending on the operands' lengths.

Finally, the *SM320C6678 DSP* based on Keystone architecture has a performance of *16Gflops/32GMACs* per core when cores operate at full frequency, which sums up to a total performance of *128Gflops/256GMACs*.



(a) CorePac Block Diagram [54] (b) Functional Block Diagram

Figure 2.4: DSP C66x [54]
(Courtesy of Texas Instruments®)

2.2.3 General-purpose computing on Graphics Processing Unit (GPGPU)

In the beginning of the 21st century an interesting twist took place in the field of Graphics Processing Unit (GPU). These previously graphics dedicated architectures were enhanced with compute capabilities by deploying multiple simple cores. Nowadays, GPGPU is constantly attracting more and more people from both academia and industry.

GPUs are ALU-heavy, which means they contain multiple small and simple cores with compute capabilities and limited control logic. For this reason, suitable GPGPU applications mostly have high arithmetic intensity (arithmetic intensity is defined as the ratio between the number of operations and the data transfers needed for those operations), large input data sets and minimal dependencies between data elements. Evidently, GPUs target data parallel applications.

Furthermore, they have a complex memory-hierarchy which consist of on-chip memories, with multiple cache layers, and off-chip memories.

GPUs are mostly used as accelerators, but integrated graphics solutions also exist. Examples are Intel's new processor designs (Sandy Bridge, Ivy Bridge, Haswell) and ARM's Mali T60x architecture where GPUs and GPPs share the same physical memory space.

Example companies that provide GPUs are NVIDIA, AMD, Intel, ARM, Imagination Technologies and Qualcomm.

NVIDIA GK110 is a state-of-the-art NVIDIA Kepler based architecture [50].

GK110 architecture consists of five Graphics Processor Cluster (GPC), where each containing three "next-generation Streaming Multiprocessors", which are widely known as *SMX* (see Figure 2.5). Most designs utilize 13 or 14 out of the total 15 SMXs, probably to increase yield during manufacturing. Moreover, across the edges there are six 64-bit memory controllers, summing up to a 384-bit path to the off-chip memory.

Each SMX unit accompanies 192 single-precision CUDA cores, 64 double-precision units, 32 special function units and finally 32 load-store units (see Figure 2.6). Four Warp Schedulers and eight Instruction Dispatch units are present, giving the ability to execute eight instructions from four selected warps (each warp contains 32 threads) per clock cycle.

Each SMX comes with a 64 KB on chip memory which can be configured as 16/48 or 48/16 or 32/32 between the L1 cache and the shared memory. One more addition to the GK110 architecture is the 48KB read-only data cache, which targets unaligned access patterns for data that are meant to be read-only during the execution of the program.

In conclusion, *GK110* leads to extreme raw computational power. For example *GeForce GTX Titan* based on *GK110* architecture offers up to *4 TFlops* of raw performance, *288 GB/s* memory bandwidth with a maximum power consumption of *250 W* [49].

Figure 2.5: GK110 architecture [50]
(Courtesy of NVIDIA ®)

Figure 2.6: SMX architecture [50]
(Courtesy of NVIDIA ®)

2.2.4 Field-programmable Gate Array (FPGA)

The inherent high performance of hardware based solutions is tempting for everyone. Unfortunately, designing ASIC for every possible application is both time consuming and not flexible.

FPGAs are ASICs which consist of a matrix of reconfigurable logic blocks connected with each other with programmable interconnects (see Figure 2.7) [55]. Thus, they can be reprogrammed to facilitate any desired algorithm. Of course, their performance is lower compared to a custom made ASIC solution but this is compensated by the flexibility that they offer through re-programmability. It's important to mention that FPGAs offer high speed I/Os and data buses which make them appropriate for real-time or high volume applications. Xilinx and Altera are the two biggest FPGA providers worldwide.

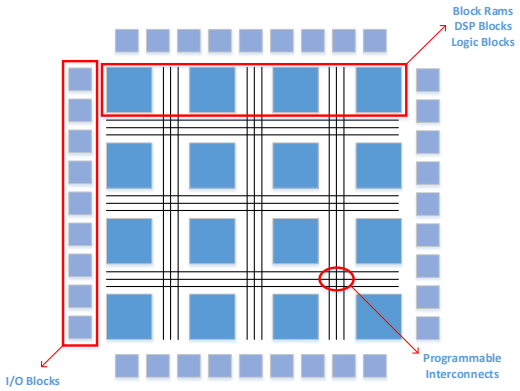


Figure 2.7: FPGA architecture

2.3 Parallel Programming Languages

Nowadays, a variety of parallel programming languages exist, giving everyone the ability to utilise and benefit of the advantages that multicore systems have to offer at a level of abstraction that suits their competency.

We will briefly describe OpenCL and CUDA, the state-of-the-art languages we extensively used, along with OpenMP and MPI which exist for decades now but still have a dominant position.

2.3.1 OpenCL

OpenCL is a royalty-free industrial standard designed for programming heterogeneous architectures which consist of collections of Central Processing Units (CPUs), GPUs, DSPs, FPGAs or other hardware accelerators in a single platform [57]. OpenCL was initially an Apple Inc. project, but from 2008 on it is managed by the non-profit Kronos Group which has the immediate support of vendors like Intel, NVIDIA, ARM, AMD and Qualcomm. Each of these providers release their own OpenCL implementation targeted for their own hardware platform.

The OpenCL platform model considers a single host connected to one or more OpenCL Compute Devices. Each Compute Device consists of one or more Com-

pute Units where each Compute Unit is made of one or more Processing Elements. Finally, each Processing Element executes code as SIMD or SPMD.

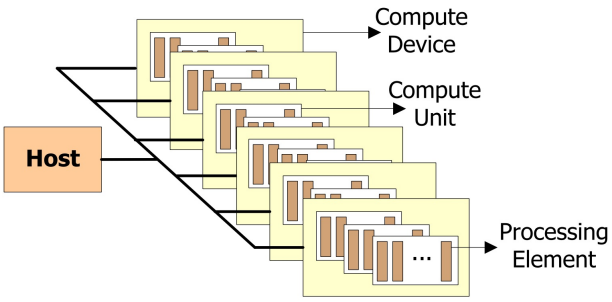


Figure 2.8: OpenCL Architecture Model
(Courtesy of Khronos Group)

An OpenCL program consists of two distinct parts [25], the host and the device code. The device code is actually a kernel, or a collection of kernels, which run on the Compute Devices; they are written in a data/task parallel manner (see Figure 2.8). The host code is responsible for orchestrating the initialization, communication, and execution of the kernels among the Compute Devices.

The data are processed over an index space which can have varying dimensionality from one to three. Each element of this index space is a work-item and a collection of work-items is called work-group. Each work-group is independent of any other one and multiple work-groups can run in parallel. Thus, work-groups should not directly share data. However, work-items of a work-group can communicate and synchronize.

Finally, the OpenCL memory model, which is a relaxed memory model, is divided in four different memory spaces: the global memory, the constant memory, the local memory and the private memory. Each one has its own limitations and regulations concerning consistency between processing elements.

The greatest advantage of OpenCL is the code portability that it offers, but this does not mean that it will run optimally across different OpenCL devices without hardware or vendor specific optimizations [26, 37, 61].

2.3.2 CUDA

CUDA is the parallel programming model and platform developed by NVIDIA in order to harness the power of the NVIDIA GPUs [56]. The first public version was released in early 2007 while the latest stable version is 5.5 and was released in 2013. CUDA is quite popular in both academia and industry, leading to a variety of compute intensive applications to be ported on GPUs.

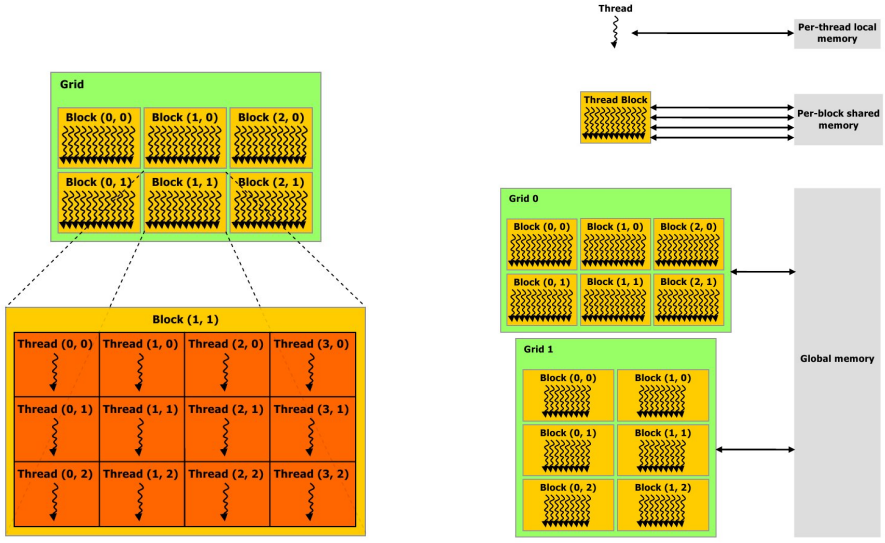
CUDA and OpenCL programming models are quite similar. Just like OpenCL, CUDA assumes a single host with one or more CUDA capable devices. The host is responsible for both the communication and synchronisation between the processor

and the CUDA devices, and for the invocation of compute intensive parts also called kernels to be executed on the GPUs. As the host code used to bare all the coordination burden of the application, NVIDIA ultimately tackled this problem by supporting in their new architectures (from Kepler on) a level of autonomous and dynamic parallelism, where a CUDA capable device has the ability to generate work for itself.

CUDA utilizes the GPUs through a hierarchy of threads [25, 69]. Threads are grouped into blocks called threadblocks and threadblocks are further grouped creating what is known as grid. The grid may have dimensionality up to three (see Figure 2.9a).

Two distinct physical address spaces are part of the CUDA memory system, an off-chip DRAM and an on-chip memory. From CUDA's perspective four different spaces are available, the global memory, the constant memory, the local memory and finally the shared memory (see Figure 2.9b).

Overall, CUDA is a major solution when targeting NVIDIA GPUs. CUDA its quite friendly as a language, making the implementation of parallel applications simple for someone that has a clear understanding of the programming model. The implementation of optimised versions may require deep understanding of both the target application and the underlying hardware, and lies on the capabilities of each programmer. Moreover, CUDA comes with a significant number of pre-optimized libraries for different scientific domains, like linear algebra (CUBLAS) and signal processing (CUFFT) which decrease programmer's effort.



(a) CUDA Thread Hierarchy (b) CUDA Memory Model

Figure 2.9: CUDA Thread and Memory Model

2.3.3 OpenMP

OpenMP is an Application Programming Interface (API) [58] that extends different programming languages in order to express shared memory parallelism. It is a collection of compiler directives, environmental variables and routines that hide the implementation details from the programmers. OpenMP is used to parallelize a sequential application. Although OpenMP can be considered a productive solution due to its abstraction level, parallelizing a legacy or inherently sequential application may limit the performance gain. OpenMP was first released in 1997 for Fortran from the OpenMP Architecture Review Board. Many specification updates have followed this up with the last one being Version 4.0.

In more detail, OpenMP is based on the fork-join [67] parallization model, where a master thread is split in a pre-specified number of work or slave threads which share the tasks among them (see Figure 2.10). Once the threads are forked, they run concurrently or in parallel. Depending on the nature of the parallelized application synchronization or communication between threads may be important for this reason appropriate pragmas exist.

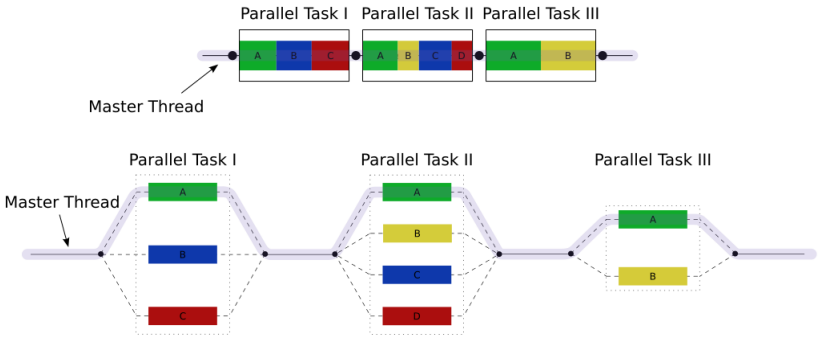


Figure 2.10: OpenMP Example [67]
(Courtesy of Wikipedia Commons)

2.3.4 MPI

MPI is a parallel programming model that enables the message passing paradigm for programming of parallel applications on distributed machines with separate memory spaces. The MPI specification standard was introduced in 1992 with the release of MPI 1.0 taking place in 1994. The latest version of MPI is MPI 3.0 and was released in 2012 [17]. After so many years, MPI is still the main solution when developing distributed applications. MPI, initially targeted only distributed memory architectures, but as multi-core processors trend took place and were connected over network, hybrid architectures were born. For this reason MPI was extended to support all kinds of memory models including hybrid (distributed and shared). Different implementations of MPI exist (e.g MPICH, LAM-MPI).

A generic MPI program consists of the environment initialization step, where the

communication world between the distributed machines is created, then the parallel execution starts where each machine performs its work and, finally, in the end the MPI environment is terminated [4] . During the parallel work step, machines can communicate using different message passing calls depending on the nature of the needed communication. Different communication types are being supported like broadcast, gather/scatter, reduce.

2.4 Real-time Systems

Real-time systems are computing systems that have to react to external stimuli within some timing constraints. As a result, the correctness of such a system is not specified only by the correctness of the calculated output, as in conventional systems, but also by meeting the imposed deadlines [5].

2.4.1 What Real-time Means

The term *"real-time"* is falsely interpreted by many [5]. Some assume a system to be real-time when it produces results really fast, but as the meaning of *"time"* is relative such is the meaning of *"fast"*.

To characterize a system as real-time we have to have a clear understanding of the environment that it operates in and, as a result, of its timing characteristics. We have to make sure that the system operates in the same time scale with the environment, and guarantee that it will respond fast with respect to the environment's evolution. Sometimes, such systems may be compromised when external, *"non-natural"*, events disrupt the timing characteristics of the environment it operates in.

Many real-time systems have been reported to malfunction due to false modelling of the environment timing characteristics, which may lead to incorrect correlation between the system's *"time"* and the environment's *"time"*.

2.4.2 Types of Real-time Tasks

At conceptual level, a real-time task differs from a non-real-time one on the timing constraints that it has to meet. These constraints come in a form of deadlines. Imposing deadlines to a task introduces the probability of missing one of them as well, which means that the task is unable to produce the required data before the deadline. As a consequence, the criticality of a deadline miss to the system's behaviour helps to distribute real-time tasks in three distinct categories [5]:

- **Hard:** A real-time task is characterised as hard when the miss of a deadline causes catastrophic consequences to the system or the environment. Example of such task is a task that controls an aeroplane's motors and/or sensors, where a deadline miss may lead to a catastrophic result from the control feedback loop.

- **Firm:** A real-time task is characterised as firm when a deadline miss produces useless data for the system but do not damage the system or the environment. Most of the times, a deadline miss in such tasks may lead to quality degradation. Examples of such tasks can be found in the signal processing domain, like video encoding/decoding or video streaming.

- **Soft:** A real-time task is characterised as soft when the data produced after a deadline are still useful but lead to a degradation to the system's performance. Example is a graphic user interface of an application.

2.4.3 Features of Real-time Systems

It is important for a real-time system to support six fundamental properties [5] in order to be able to host a critical application. These properties are:

- **Timeliness:** Such a system has to be able to guarantee that all the imposed deadlines are going to be met.

- **Predictability:** It is important for real-time systems to have been developed in a way where extensive analysis schemes can be applied to them, and varying behaviour over different scheduling techniques can be predicted.

- **Efficiency:** Usually, real-time systems come with not only timing constraints but also with power, area and computational power constraints. Thus, such a system has to operate correctly given all this constraints.

- **Robustness:** Aperiodic events may have a varying pattern depending on the evolution of the environment. Consequently, when event burst occur the response of such a system may be compromised. For this reason, we have to guarantee the correct behaviour in extreme workload cases.

- **Fault tolerance:** Due to either materials' physical properties and limitations or software errors, faults may introduced to a system. As a consequence, the system should be able to tolerate such permanent or transient faults and continue working in order not to compromise the system's operation.

- **Maintainability:** The system has to be developed in such a way that possible extensions and/or modifications can take place without much integration effort.

2.5 Synthetic Aperture Radar (SAR)

SAR is a form of radar imaging methodology where a large antenna is emulated by moving a smaller antenna over a specific region. SAR creates finer spatial resolution images than any other conventional beam scanning techniques. SAR

technology is usually deployed on satellites, aircraft and more recently also on Unmanned Aerial Vehicles (UAVs).

Figure 2.11: MQ-9 Reaper UAV equipped with SAR
(Courtesy of U.S. Air Force)

2.5.1 Radar Principles

Radar stands for *Radio Detection and Ranging* and is an electrical system the main purpose of which is to detect objects and define their distance [36]. The fact that radar systems operate in almost every weather condition, day and night, makes them a really attractive solution compared to others.

In more detail, radars transmit electromagnetic waves over a target area and receive reflections of these waves from objects that may lie in this area of interest (see Figure 2.12). The angle the electromagnetic waves are received and the time difference between the transmission and reception of these waves reveal information about the position and the range of the objects in this transmission area.

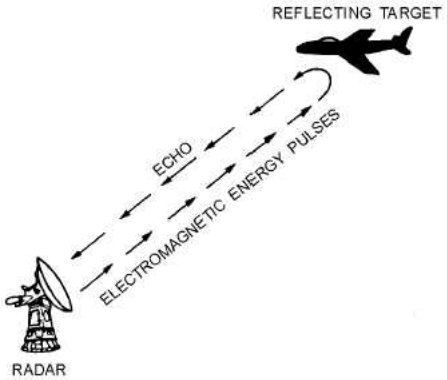


Figure 2.12: Radar Principle

Identifying an object sometimes is not that straight-forward, as multiple sources of both external and internal interference exist. Examples are the noise of the electronic circuitry (e.g. thermal noise) and reflected electromagnetic waves from areas outside the area of interest.

Radars operate in frequencies between *300MHz* and *110GHz*, and limitations exist in both low and high frequencies. Achieving high resolution in low frequencies require large antennas while high frequencies increase the effect of atmospheric attenuation [36].

The transmitter along with the receiver are the main components of a radar system. The antenna is the part of the system which transfers the electromagnetic waves from/to system to/from the medium. Multiple transmitter/receiver configurations exist. They may share the same antenna (monostatic) or be separated, each one having its own antenna (bistatic). The main reason of bistatic radar systems is to provide physical isolation of the transmitting and the receiving subsystems. This is due to the fact that transmitters usually transmit high-power electromagnetic waves (KW or MW) which may affect the sensitive subsystem of the receiver, which receives signals with orders of magnitude less power (mW or nW).

Two distinct categories of radar waveforms exist: continuous wave and pulsed. In continuous wave waveform, transmitters and receivers operate at all times with the transmitter continuously transmitting a signal. On the other hand, in pulsed waveforms, short signals are being transmitted with a short period, while the receiver receives in between transmissions.

When relative motion between the target and the radar exist, the frequency of the received signal will be different from the frequency of the transmitted one. This commonly known as the Doppler effect. Proper utilisation of the Doppler effect helps to identify moving targets along with their direction and speed.

2.5.2 SAR Principles

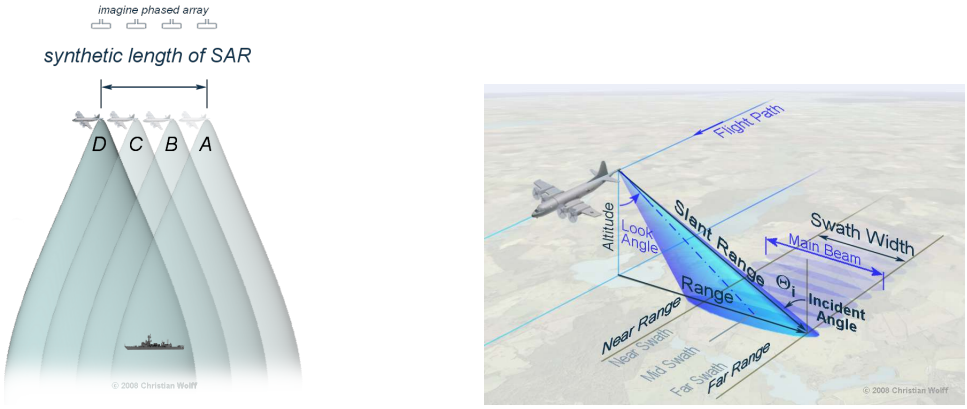
The fundamental premise of a SAR radar is to be able to emulate a rather large antenna, which would require a large amount of power and area, by moving a smaller antenna over a target area. The longer this synthetic antenna is, the higher the spatial resolution we can achieve.

The length of the synthetic antenna is determined by the distance between the start and the end of the data collection process (see Figure 2.13). Finally, by coherently integrating the collected data we can construct high resolution spatial images of the illuminated area.

The resolution of a SAR system can be expressed in two terms along two perpendicular axes: the *range resolution* and the *azimuth resolution* (see Figure 2.14).

The range resolution takes place along the range axis which is perpendicular to the antenna moving path. Calculation of range resolution does not differ of that of a conventional radar.

The azimuth resolution is the resolution along the moving track. Multiple observations take place along the track, which in the end are coherently integrated to form high resolution images.



(a) Length of Synthetic Antenna (b) SAR Geometry

Figure 2.13: SAR Principle [70]
(Courtesy of Radartutorial.eu)

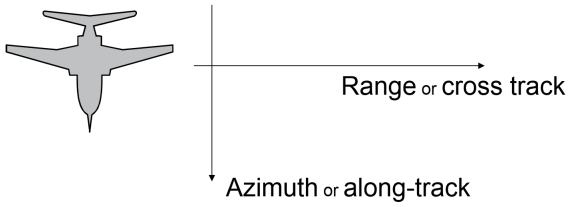


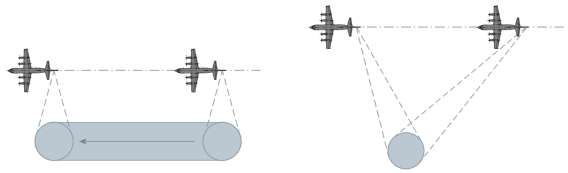
Figure 2.14: SAR Resolution

SAR Operation Modes

SAR systems support different illumination modes depending on the needs of the application. Figure 2.15 illustrates the most common SAR modes .

In *strip map mode* (see Figure 2.15a) the antenna has a fixed direction resulting in successive high resolution images along the illumination path of the antenna.

On the other hand, in *spot mode* (see Figure 2.15b) the antenna is steered to point to a desired target in the ground which leads to longer integration time and therefore very high resolution images. With digital beam forming, as mentioned later, multiple beams can be formed so that Strip and Spot can be performed simultaneously. This puts heavy demands on the processor.



(a) Strip Map Mode (b) Spot Mode

Figure 2.15: SAR Modes [70]
(Courtesy of Radartutorial.eu)

SAR Processing

As we previously mentioned, the main purpose of a SAR system is to transform the sampled raw data into a spatial image. This image formation procedure consists of multiple processing steps, with the most important being *pulse compression* and *azimuth compression*. Other steps include motion compensation, autofocus and contrast scaling which improve image's focussing and sharpness.

Over the years multiple ways of performing these important SAR processing steps have been proposed. We distinguish three distinct types of SAR processing techniques, each one having different weight distribution between accuracy and performance [34].

- Frequency domain algorithms like Spatial Matched Filter Interpolation, Range-Doppler, ω-K and Chirp Scaling perform both range and azimuth compression in the frequency domain. The utilization of FFTs to do that decreases considerably the required computational time but may lead to defocused images, wrap-around errors or even spread artifacts all over the processed image. This mainly happens because assumptions for the data layout are being made which may lead to formulation problems and approximations especially for wide-angle processing. Further, spatial based processing techniques are sensitive to motion errors, which are most of the times present in airborne SAR.

- Back-Projection is an alternative that offers accurate results in exchange of higher computation requirements as it operates on the time-domain. Back-Projection stems from tomographic imaging, where one-dimensional line projections taken from a two or three dimensional target scene from different angles, are used to create a spatial image of the target scene. Being a time-domain approach, makes back-projection able to deal with non-linear motion paths. Back-Projection was not possible years ago, but has become feasible thanks to fast computing.

SAR applications

The ability of radar systems to operate independent of the weather conditions, along with the flexibility and high resolution output of the SAR processing scheme makes SAR have numerous applications from military to environmental based ones [36].

Some example applications are:

- Target Surveillance: Used mainly for military purposes for the surveillance of specific target areas. Surveillance reasons may be treaty preservation between countries.
- Ground Moving Target Indication (GMTI): Moving targets are being monitored. Informations like the direction and the speed of moving targets can be extracted and utilized in a variety of ways.
- Navigation: The ability of radar systems to work in every weather condition, day or night, can be used in navigation systems. Cross-correlation of images with reference images that were previously obtained may reveal location and/or direction information.
- Environmental Monitoring: Environmental changes can be monitored with the use of SAR technology. Such changes may be deforestation, ice level or even oil spills as different materials have different backscatter characteristics.
- Foliage and Ground Penetration: Interesting is the fact that low frequency radar system can penetrate foliage and sometimes decades of meters beyond ground surface. Such an important attribute can reveal information about underground targets.
- Interferometry: When two or more SAR images (or data from 2 or more antennas) are combined, generation of 3-D surface images is feasible.

Chapter 3

Software Architecture

3.1 Processing Requirements

As we mentioned in the previous chapter multiple processing steps are required in order to produce high quality two-dimensional spatial images. Figure 3.1 illustrates the processing chain of a multi-channel SAR[62].

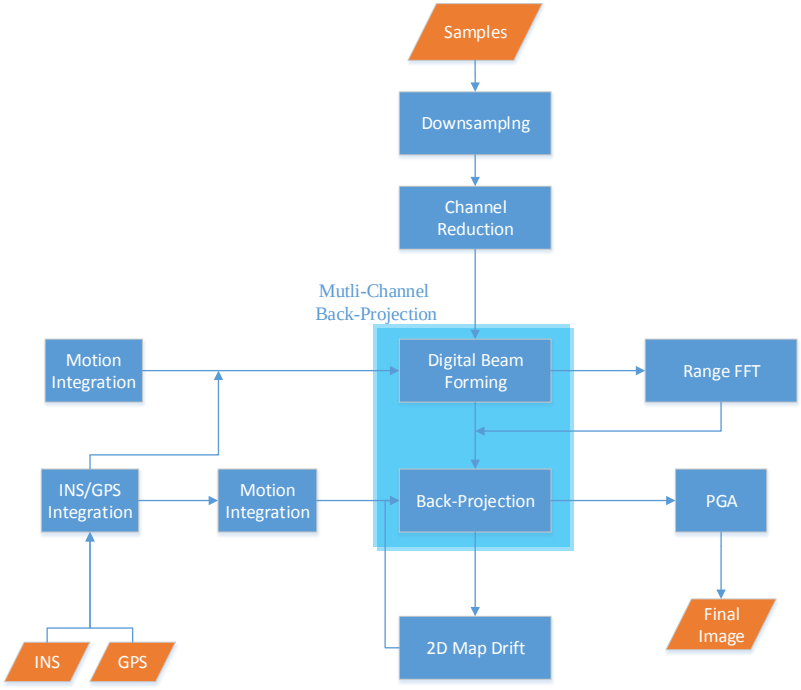


Figure 3.1: SAR Software Architecture

Initially, after the sampling phase a Decimation step takes place. Decimation [33] filters and down-samples the input data in order to reduce the input data rate.

This could be helpful for two main reasons. First of all, the sampling rate may be high enough leading to a high data rate, which may not be able to be supported by the underlying communication interface. Second, the amount of gathered samples may be more than enough to reconstruct high quality images, so a down-sampling step will reduce the processing requirements of the following steps of the chain and still provide high-resolution images.

Having multiple channels in the azimuth direction increases the active sampling rate which leads to higher resolution images or to wider swath width [65]. Channel reduction, tries to reduce the input data rate by adding pairs of channels together. Adding all the channels in this level of the processing chain leads to a conventional SAR configuration with one transmitter and a single channel receiver.

Multi-channel Back-Projection is the heart of the SAR system. It operates over three sampling domains: the channels, the fast time (cross-track) and the slow time (along-track). Back-projection constructs the spatial image of a target area by coherently summing the appropriate reflections of each voxel on the grid. In addition, Digital Beamforming (DBF) [20] helps improve the final image by focusing the receiver in the required direction. Moreover, DBF gives the ability to point different beams in multiple direction, thus multiple images can be constructed. In back-projection we perform explicitly an ultimate form of beamforming by focusing and steering beams to each voxel independently. Range resolution is achieved using FFTs while azimuth resolution by coherent integration along the flight path.

Information about the flight track are obtained through the Inertial Navigation System (INS) and the Ground Positioning System (GPS). By combining these information we can have a clear view about the radar positioning and motion.

Unfortunately, high accuracy INS and GPS systems are expensive and power hungry. Moreover, the accuracy required by a SAR systems strongly depends on the operating frequency of the SAR which for example when is in a magnitude of GHz, millimetre resolution is required by the INS and GPS sensors. In order to compensate potential motion errors multiple steps of iterative autofocus are performed [7]. With *Map Drift*[6] we can compensate large motion error while residual motion errors can be compensated using the *Phase Gradient Algorithm (PGA)* [64].

It is important to mention that this processing chain is just an example. More steps can be added or changed in order to support different functionality or meet specific power or time requirements. An example could be a step for identifying moving targets.

Finally, our main goal is to identify the requirements of each step of the processing chain and investigate possible real-time solutions under area, power and computational power constraints. We will provide this analysis in two phases. First we will discuss each module (Chapters 4-7). Next, we integrate and analyze these modules (Chapter 8).

3.2 Hardware Platforms

A variety of hardware architectures were used to evaluate empirically our SAR implementation.

To start with, Table 3.1 summarizes the specifications of the used GPPs: Intel® Core™ i7-2670QM[42], Intel® Core™ i7-3612QM[43] and Intel® Core™ i7-4960X Extreme Edition[45].

Model	Intel Core i7-2670QM	Intel Core i7-3612QM	Intel Core i7-4960X Extreme
Cores/Threads	4/8	4/8	6/12
Core Frequency (GHz)	2.2	2.1	3.6
Last Level Cache (MB)	6	6	15
Throughput (GFlops)	70.4	67.2	130
Memory Bandwidth (GB/s)	21.3	25.6	59.7
Memory Channels	2	2	4
Thermal Design Power (W)	45	35	130

Table 3.1: GPPs' Specifications

Further, multiple GPUs were used: a mobile GPU, NVIDIA® GeForce™ GT540M[48], a mid-range GPU, NVIDIA® GeForce™ GTX650 Ti[48] and finally two high-end GPUs, NVIDIA® Tesla™ K20[52] and NVIDIA® GeForce™ GTX Titan[49]. Table 3.2 presents the most important specifications of the previously mentioned architectures.

Model	NVIDIA GT540M	NVIDIA GTX680	NVIDIA GTX Titan
Cores/Threads	2/96	32/1536	14/2688
Core Frequency (Hz)	1344	1006	837
Off-chip Memory (GB)	1	2	6
Throughput (GFlops)	258	3090	4500
Memory Bandwidth (GB/s)	28.8	192	288
Memory Frequency (Hz)	900	6000	6000
Memory Interface	GDDR3	GDDR5	GDDR5
Memory Width (bits)	128	256	384
Compute Capability	2.1	3	3.5
Thermal Design Power (W)	35	195	250

Table 3.2: GPUs' Specifications

The experimental data that we used come from AMBER [60], a *X-band Digital Array SAR* which targets UAVs. AMBER has a maximum bandwidth of 1 GHz and thus a maximum resolution of 15 cm. Moreover, it is equipped with a 24 element receiver array with a sampling frequency of 20 MHz.

3.3 Metrics

Throughout this thesis multiple implementations of the all the algorithms will be presented. In order to compare those and be able to evaluate their performance on different hardware platforms we use the following metrics:

- **Execution Time** (T) is the main decisive metric when comparing different implementations. The execution time is obtained using the *wall timer* or using profiling tools like *NVIDIA's nvprof* [51] and *Intel's VTune Amplifier* [44].
- **Speedup** (S) represents the absolute performance gain between two different implementations with execution times T_1 and T_2 respectively:

$$S = \frac{T_1}{T_2} \tag{3.1}$$

- **Throughput**
  - Computational Throughput, is a measure of the amount of work that a computational system can perform in a given time period, and is calculated as the number of floating point operations that are being performed per second:

$$FLOPS = \frac{FLOPs_{total}}{T} \tag{3.2}$$

  - Memory Bandwidth, is a measure of the rate that data can be stored and read from a memory subsystem, and is calculated as the number of bytes transferred through memory transactions from and to the main memory (read and writes) per second:

$$MBw = \frac{bytes_{read} + bytes_{written}}{T} \tag{3.3}$$

- **Back-projections per second** (BPs), the number of back-projections that can be performed per second:

$$BPs = \frac{Back - projections_{total}}{T} \tag{3.4}$$

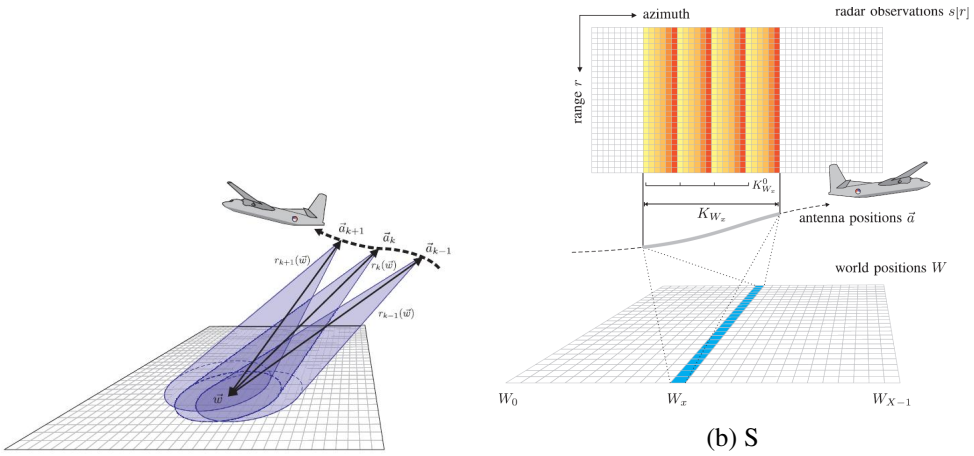
Chapter 4

Multi-channel Back-Projection

4.1 Algorithm

Back-Projection is one of the most used processing techniques in SAR systems. We focus on the time-domain "brute-force" approach presented in [63].

Along a flight path each radar observation $s\left[r\right]$ is associated with a three-dimensional antenna position $\vec{a_k}$. Each radar observation is a collection of range compressed, complex data indexed by range r (see Figure 4.1b).



(a) SAR scheme

(b) SAR data representation

Figure 4.1: SAR back-projection

Range can be specified by calculating *the euclidean distance* between the position of the voxel $\vec{w}$ where we need to measure the radar reflectivity and the k-th position of the antenna $\vec{a_k}$ (see Figure 4.1a).

$$r_n(\vec{w}) = |\vec{w} - \vec{a_k}| \tag{4.1}$$

Having calculated the range between these two points, we can calculate the one-way phase shift $\phi_k(\vec{w})$ and consequently the two-way phasor $\theta_k(\vec{w})$.

$$\phi_k(\vec{w}) = 2\pi \frac{r_k(\vec{w})}{\lambda} \tag{4.2}$$

$$\theta_k(\vec{w}) = e^{2j\phi_k(\vec{w})} \tag{4.3}$$

where λ is the wavelength of the transmitted wave.

Further, the partial reflectivity of this voxel from a single observation can be reconstructed as follows:

$$u_k\left[\vec{w}\right] = s_k\left[r_k(\vec{w})\right] \cdot \theta'_k(\vec{w}) \tag{4.4}$$

Finally, the total reflectivity of a voxel can be calculated by coherently summing all the partial contributions to the voxel's reflectivity from all the relevant radar observation.

$$u_k\left[\vec{w}\right] = \sum_{k \in K_{\vec{w}}} u_k\left[\vec{w}\right] \tag{4.5}$$

Clearly, in order to compute the reflectivity of a collection of voxels representing the grid of interest, this procedure has to be performed for each voxel independently, leading to high computational demands for high resolution wide grids.

For multi-channel receive antennas the number of radar observations is increased by a factor equal to the number of the receive channels. To be more specific, for m channels, the total reflectivity of a voxel is:

$$u_k\left[\vec{w}\right] = \sum_{k \in K_{\vec{w}}} \sum_{m} u_{k,m}\left[\vec{w}\right] \tag{4.6}$$

4.2 Sequential Implementation

Initially, we implemented a sequential version of the multi-channel back-projection to reveal any computational bottlenecks and identify potential candidates for parallelization.

Figure 4.2 illustrates the processing steps of the presented algorithm. In the initialization step, the required data structures are initialized and the observations along with the position of the antenna are read. Having the antenna positions helps us calculate the position of each channel respectively.

Next, the main processing routine follows, where for each pulse and for each channel we calculate the reflectivity of each voxel in the target grid.

Overall, the whole process to calculate each voxel's reflectivity is an independent summation over three distinct domains: fast-time (across range), slow-time(across

track) and channels. As time-domain processing in the fast-time domain (across-range) lead to no significant improvements in output's quality, frequency-domain processing is preferred (Range Compression box). The order these summations occur does not change the final result.

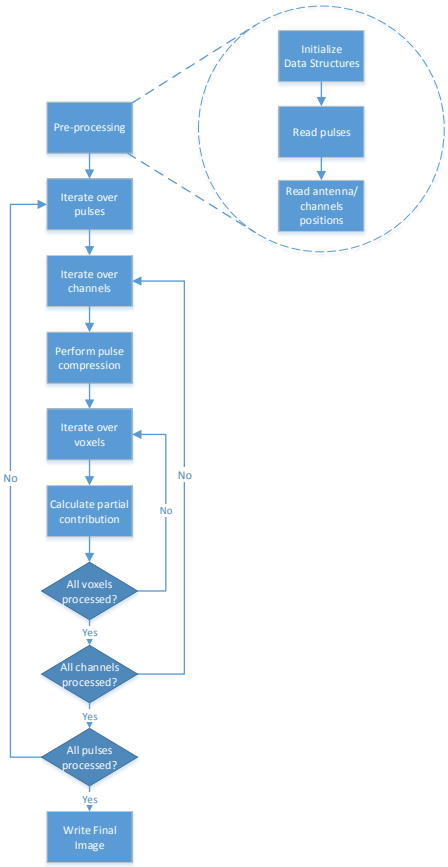


Figure 4.2: Data flow diagram

Profiling this application will help measure the execution time of each individual part along with its contribution to the overall execution time of the program. We reconstructed voxel grids of varying sizes by integrating over 5 seconds of samples which is approximately 1506 sweeps.

Table 4.1 presents the execution time of the most important functions along with their contribution to the total execution. As we can see, the contribution calculation function and the range compression account together for the most part of the total time. When reconstructing voxel grids of larger sizes, we observe that the execution time of the contribution function is increasing while range compression

remains constant.

Evidently, we conclude that the contribution function along with the range compression one are the two main processing steps that urge parallelization as they account for approximately the 95% of the total execution time.

Size	Functions	T[s]	%
100mx100m	contribution	1276	74
	rangeCompress	301	17
	others	154	9
200mx200m	contribution	5320	92
	rangeCompress	304	5
	others	162	3

Table 4.1: Profiler Results on GPP (single-threaded)

4.3 Parallel Approaches

Typically, applications use either input-based parallelization or output-based parallelization.

In input-based parallelization, the input elements are processed concurrently or in parallel. This can be advantageous when the number of the input elements is significantly larger than the output elements, but on the other hand can also lead to synchronization problems when multiple input items affect the same output element.

In output-based parallelization, the output elements are processed in parallel. While this can lead to redundant reads of input elements, and less parallelism in case the output is much smaller than the input, it resolves the synchronization problem.

Below we describe how these approaches can be applied to the given back-projection algorithm along with arguments of which one is friendly for a GPU implementation.

4.3.1 Input-based parallelization

In this parallelization approach, each thread takes care of one or several input element(s). Thus, our main goal is to calculate which output voxels this input element/s affects and, as a result, coherently add its contribution to these voxels.

The number of elements of the input data is orders of magnitude larger than the output elements, a factor suitable for a gpu-level parallelism. Moreover, threads will read consecutive memory elements (coalesced accesses) which will lead to better utilisation of the memory hierarchy.

In order to follow this approach, each input element has to iterate over all the output elements and check which ones it affects. In [10] the authors mathematically

prove that the voxels that have to be checked can be limited to a certain amount, but still depending on the size of the grid and radar specific parameters their number could vary. Moreover, iterating over output elements may lead to a computational explosion and also increases dramatically the lifetime of each individual thread. Another way is to use a mathematical way to specify, with an acceptable accuracy, the elements that each input item affects. This may of course introduce more complex control logic and highly depends on many radar and sampling specific factors which are explained further during the data analysis step. Finally, due to the fact that multiple threads may access similar output voxels, atomic access to the output voxels is required.

4.3.2 Output-based parallelization

The output-based parallelization can be implemented by making each individual parallel task responsible for calculating the contribution to a specific voxel of the output voxel grid. Such an approach may initially imply a lower level of parallelism as the number of the voxels is orders of magnitude less than the input data, but in reality the computational part is much more parallel-efficient and straightforward.

The main problem of this approach is that, as mentioned in the algorithm presentation, for each voxel we have to calculate the distance from the antenna for each aperture. Depending on this distance we have to access the appropriate range bin. These accesses to the corresponding range bins for the voxels may not be consecutive between voxels which may lead to lower utilization of the memory hierarchy or restrictions for further optimisations. Further analysis on the access patterns take place later in this chapter.

We chose to proceed with this approach as it turns out to be more parallel efficient and has less drawbacks and limitations compared to the input-based approach.

4.4 Analysis

Before proceeding to the parallelization process is important to have a deep understanding of the computational and memory requirements of the algorithm.

We start with a basic setup where we assume a single antenna consisting of M channels in the azimuth direction and a Frequency-Modulated Continuous-Wave (FMCW) radar.

4.4.1 Complexity Analysis

Input

In SAR processing, an image is reconstructed using a start-stop approximation, where a synthetic aperture is created consisting of multiple individual radar observations. The number of the individual apertures, P, depends on the start and stop

time of the integration process and the Signal Repetition Frequency (SRF), which is the frequency a pulse is transmitted (see Equation 4.7).

$$P = |t_{start} - t_{stop}| \cdot SRF \tag{4.7}$$

During consecutive apertures there are M channels sampling over the fast-time domain. The number of the collected samples across range per channel, R, depends on the hardware's sampling frequency (f_{sample}) and the upsweep's time ($T_{upsweep}$) (see Equation 4.8)

$$R = f_{sample} \cdot T_{upsweep} \tag{4.8}$$

For back-projection to achieve better resolution, an *interpolation* or *upsampling* takes place in the fast-time domain. This leads to an increased number of range bins, $\tilde{R}$, and depends on the upsampling factor ($n_{upsample}$) (see Equation 4.9).

$$\tilde{R} = R \cdot n_{upsample} \tag{4.9}$$

Figure 4.3 illustrates the structure of the observations in the three distinct domains.

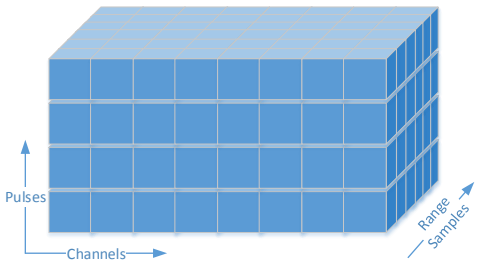


Figure 4.3: Data Layout

As a result, the number of input observations equals to the number of apertures, times the number of channels, times the number of observation of each channel at each aperture:

$$input = obs = P \cdot M \cdot \tilde{R} \tag{4.10}$$

Output

The number of the output voxels in each dimension depends on the size of this dimension and the resolution we want to achieve in this dimension (see Equation 4.11).

$$N_d = \frac{Size_d}{res_d} \tag{4.11}$$

where $Size_d$ is the length of the voxel grid in this dimension in meters and res_d is the resolution in this dimension.

The total number of voxels or output out is the product of the voxels in each dimension (see Equation 4.12).

$$out = N_{total} = \prod_{i}^{d} N_i \tag{4.12}$$

Overall, the complexity of the algorithm is $O(PM(N_xN_y + (\tilde{R}log_2\tilde{R})))$. This comes from the fact that for every pulse P and for every channel M we perform range compression over the interpolated range $\tilde{R}$ samples using FFTs[1] and we calculate the contribution for each pixel (where N_xN_y is the total amount of pixels of a 2D voxel grid). Dominating factors are the number of channels, the number of pulses and the number of voxels.

4.4.2 Operational Intensity Calculation

It is important to calculate the needs of our application in terms of computational intensity and memory bandwidth. This will provide us insight in our application's bounds and will guide our optimization strategy.

First of all, we focus on the contribution function requirements. Let's assume we operate on range compressed data. Moreover, we do not account the requirements for the pre-processing step as they are negligible compared to the main processing procedure.

The equation below (Equation 4.14) gives an estimate of the requirements of the application in terms of FLOPs, which in turn determines its computational needs and intensity.

$$\begin{aligned} FLOPs =& \text{\#iterations} \cdot \\ & \text{(The computational intensity of the contribution function)} \\ =& P \cdot M \cdot N_xN_y \cdot \\ & 8 \cdot addf + 8 \cdot subf + 20 \cdot mulf + 3 \cdot divf + 1 \cdot sqrtf + 1 \cdot expf \end{aligned} \tag{4.13}$$

where :

- $addf$, $subf$, $mulf$ and $divf$ are single-precision floating point addition, subtraction, multiplication and division respectively.
- $sqrtf$ is single-precision floating point square root calculation.

[1] N-point FFT has a complexity of $O(Nlog_2N)$

- $expf$ is single-precision floating point exponential calculation.

The requirements of each individual floating-point operations for an x86 architecture are [2] :

- $addf$ 1 FLOP
- $subf$ 1 FLOP
- $mulf$ 1 FLOP
- $divf$ 15 FLOPs
- $sqrtf$ 15 FLOPs
- $expf$ 20 FLOPs

By taking into account the above numbers, the computational intensity of the kernel is *116 FLOPs*.

On the other hand, the memory requirements of the application are:

$$\begin{aligned} Mem_{Bytes} =& \text{\#iterations} \cdot \\ & (\text{Floating points read + Floating points written}) \cdot (\text{Bytes per floating point}) \\ =& P \cdot M \cdot N_x N_y \cdot \\ & (8 \text{ reads + } 2 \text{ writes}) \cdot 4B \end{aligned} \tag{4.14}$$

We can calculate the operational intensity of the application as the operations per byte of DRAM accesses. For the back-projection applications the operational intensity equals :

$$\frac{P \cdot M \cdot N_x N_y \cdot 116}{P \cdot M \cdot N_x N_y \cdot 40} = \frac{116}{40} = 2.9 \tag{4.15}$$

The *roofline model*[68] is a theoretical model having as its goal to provide realistic insights about an application's performance and reveal its bounds. The model ties together peak performance, memory performance and operational intensity, thus a different output is associated with different architectures due to different performance and memory capabilities. The model is represented as a two dimensional graph having as an upper bound a horizontal line which represents the peak computational performance of the targeted platform and a steep line which reaches eventually the horizontal line and represents the memory bound of the platform. Further, X axis represents Flops per byte while Y axis the attainable performance in $Gflops/s$. The attainable performance of an application can be calculated using the following Equation:

$$\begin{aligned} &\text{Attainable GFlops/sec} = \text{Min}(\text{Peak Floating Point Performance}, \\ &\qquad \text{Peak Memory Bandwidth x Operational Intensity}) \end{aligned} \tag{4.16}$$

Depending on the *"roof"* that application's operational intensity reaches, it can be characterized as compute or memory bound. Figure 4.4 illustrates the roofline model for our application. Clearly, our application is considered memory-bound.

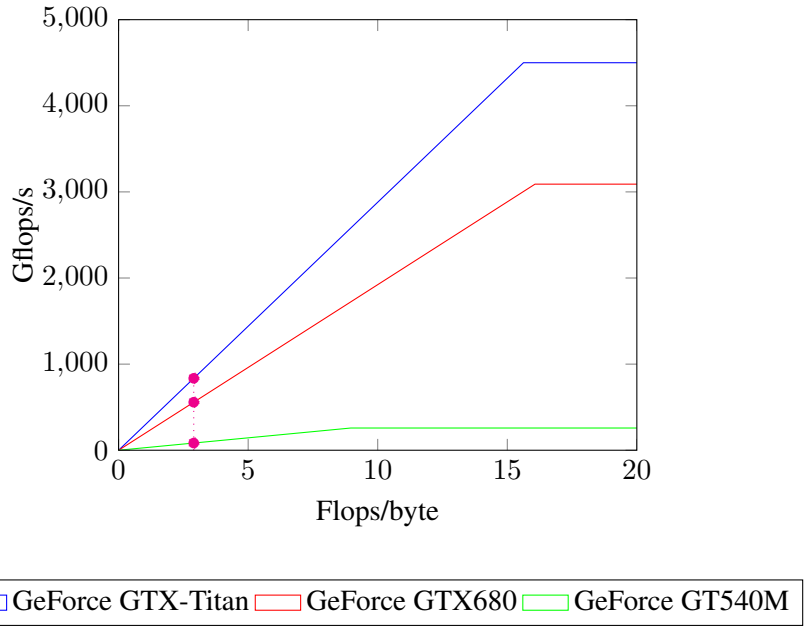


Figure 4.4: Roofline model before applying any optimizations

Finally, assuming a second of integration which, for our radar hardware, consists of *301 pulses* and different voxel grid dimensions of the highest possible resolution(15cm), the computational and memory requirements for the contribution calculations are presented in Table 4.2.

Size	Back-Projections[GBP]	Computational Requirements [GFLOPS]	Memory Requirements [GB]
100m x 100m	16.08	372.8	128.555
300m x 300m	144.57	3351.93	1155.84
1km x 1km	1606.56	37247.46	12843.95

Table 4.2: Computational and memory requirements per second of integration

4.5 Parallel Implementation

4.5.1 Naive Implementation

Our first attempt was to actually port the serial implementation on a GPU using the finest task granularity possible, where each execution element was responsible for

the calculation of the reflectivity of a single output voxel. Although this is a simple and straight-forward approach, it lead to a significant speedup over the sequential implementation (see Table 4.3).

4.5.2 Optimization Steps

In this section we describe a series of optimizations we applied to improve the performance. Both architecture independent and vendor specific optimizations were applied mainly targeting the NVIDIA GPU series. The optimization and tuning process is an iterative procedure where in each step the performance evaluation is revisited to make the appropriate decisions towards further possible improvements. All the optimization results are reported in Table 4.3.

Range Compression Optimization

Our first goal was to optimise the range compression step. For every second of integration we have to approximately integrate over *301 pulses* for the specific radar hardware. Each pulse consists of independent collections of range samples. The number of these collections is equal to the number of channels, which is 24 in our case, and each collection has *29160 samples*.

As a result, for every second of integration we have an input data size of approximately $850MB$. In strip mode, where not maximum possible resolution is required, the integration of $2s - 3s$ could be enough, but for spot mode, where high resolution images are constructed, up to $10s$ are required.

Fast-time integration is performed in the frequency domain using FFTs which leads to lower computational demands than time-domain processing. A variety of FFT libraries exist which provide highly optimized routines (e.g. FFTW [16], CUFFT [47], clMath [41]) and are generally preferred. Initially, we used *FFTW* to perform range compression on the CPU, using the maximum number of possible hardware threads, but the fact that we perform up-sampling over range to achieve better accuracy lead to large range compressed data sizes. In more detail, to achieve better range resolution we increase the number of bins by a constant factor k. This results in hundreds of thousands bins. For example, for an up-sampling factor of 8 we have to perform a $233280 - point$ FFT for each channel of each pulse. This FFT leads to an output of $13.5GB$. If we take into account that for real inputs the Hermitian property is satisfied, the size can be reduced to half ($7GB$).

Consequently, for every second of integration we have to transfer $7GB$ of input data to the GPU which with the current state-of-the art communication interface PCI Express 3.0 requires more than one second. This immediately imposes enough problems into achieving real-time performance.

The most straight-forward solution is to perform both the up-sampling and range compression inside the GPU. Because the output of the FFTs is larger than the memory that GPUs have available, we perform range compression and eventually the contribution calculation over small chunks of input pulses.

In total, we manage to reduce the data transfer time only to the time required for transferring the initial input data, which for example, for $5s$ of integration ($4GB$ of input data) less than one second is required. In addition, performing FFTs inside the GPU gives a significant speedup of approximately 10 over the execution time required by the range compression, and especially for this size of FFTs.

For the CUDA version *CUFFT* library is used. For OpenCL, *clMath* is preferred. Moreover, as we send batches (chunks) of pulses for processing, we have the ability to perform multiple FFTs in parallel and leverage the compute capabilities of the GPU.

Limitations of these libraries are the supported FFT sizes, which have to be any mix of powers of 2,3,5,7 for *CUFFT* and 2,3,5 for *clMath*. Moreover, *CUFFT* operates on data structures with interleaved complex values, while clMath gives the ability to also use planar input/output arrays where real and imaginary values are stored in different arrays. Such an option can give an significant advantage for architectures with vector hardware extensions and achieve better memory coalescing.

Basic Optimizations

Compiler Flags In most GPU architecture there are hardware units able to perform special math functions like square root and sinusoid calculations. Using this option may increase the performance of our application but will also decrease the accuracy as these operations are usually approximations. Apart from that, GPUs can usually perform both multiplication and addition in a single instruction, something that is referred as fused multiply-accumulate.

In order to force the compiler to use these hardware capabilities, special compilation options have to be used. For OpenCL *"-cl-fast-relaxed-math"* and *"-cl-mad-enable"* options enable the use of native mathematical functions and the use of fused multiply-accumulate operations respectively. In CUDA the use of native functions can be achieved using the *"–use_fast_math"* flag which also automatically enables the contraction of multiplications and additions to fused.

By mapping the math functions on native GPU instructions we reduce the number of flops required for each one of them. The requirements of each individual native floating-point operation for a GPU architecture is [2] :

- $addf$ 1 FLOP
- $subf$ 1 FLOP
- $mulf$ 1 FLOP
- $divf$ 1 FLOP
- $sqrtf$ 1 FLOP
- $expf$ 1 FLOP

By taking into account the new numbers, the computational intensity of the kernel drops from *116 FLOPs* to *41 FLOPs*. This leads to a reduction of the arithmetic intensity of the back-projection kernel and makes our kernel memory-bound. Additional compiler optimizations reduce further the number of flops.

AoS vs SoA Channel and voxel grid positions along with pulse compressed data are grouped into structures. A position structure contains x, y, z elements which represent the position of an element and a pulse compressed structure contains the real and imaginary elements of a complex number.

Depending on the application requirements and the underlying hardware, different ways to represent the data are preferred to achieve better performance and better memory utilisation. Two of them are AoS and SoA (see Figure 4.5). In AoS, an input array is a collection of identical complex elements, where each one consists of multiple sub-elements. On the other hand, in SoA, input is an element which consists of arrays each one containing exclusively one of the sub-elements of the complex structure.

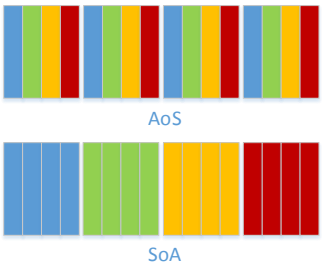


Figure 4.5: AoS vs SoA

In each contribution calculation, the positions of the voxel and the channel along with the value of the calculated range bin are needed. Access granularity inside threads occurs in terms of structures, which means that each thread reads a whole struct and not selective elements of a structure. In GPUs, memory accesses occur with a warp granularity. Consequently, threads inside a warp will access consecutive voxel position structures, leading to coalesced memory accesses. Moreover, access patterns for the range elements are irregular, something that is later explained. Thus, transformation from AoS to SoA will lead to almost no speedup to the execution time and it requires additional time for the transformation process of all the input data.

Finally, we have to point out that for hardware architectures that deploy SIMD units, using SoA is preferred. For CUDA implementation we use the AoS approach while for OpenCL the AoS is also implemented for efficient execution on vector based architectures.

Shared Memory Leveraging the underlying access patterns is important towards achieving high performance applications. Understanding and knowing possible access patterns help us utilise the fast, on-chip, shared memory which has low access times. This helps us achieve higher memory bandwidth and as a result improve the performance of memory-bound applications. Our application is memory bound which, for this reason, makes any possible exploitation of access patterns beneficial.

Figure 4.6 illustrates the phase history of some grid voxels, which also represents the access pattern between consecutive pulses for a voxel. We can clearly see that there is a regularity in the accesses for a single voxel. Unfortunately, for different voxels, we observe that the curvature of the access patterns changes. This mainly depends on two factors, the distance of each voxel grid with respect to the antenna and the sampling space of the observations.

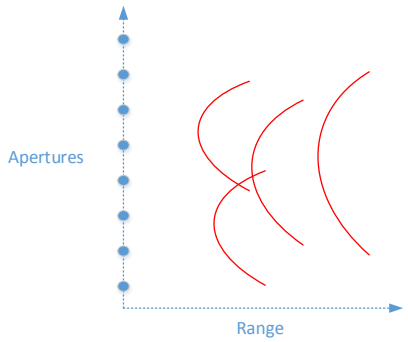


Figure 4.6: Phase history

For uniform sampling we can assume a regularity in the accesses, but a non-uniform sampling space may change instantaneously the curvature of the phase history.

Even if we assume uniform sampling space the curvature change for different voxels depends on their distance, something that requires additional control logic in order to be realized.

Overall, trying to exploit access patterns in our application is *"tricky"* and may lead to wrong or blemished results. Consequently, in order to preserve the correctness of our application we avoided this kind of optimizations.

Instruction Throughput NVIDIA's profiler provides information about the device's ability to issue instructions. Many reasons lead to an instruction stalling, making unable the GPU to issue the next kernel instruction. Instruction fetch stall, execution dependency stall, data request stall and synchronization stall are some of them. For our application, profiler reports an instruction stalling factor because of execution dependencies of up to 40%. This means that the input operands that

instructions require in order to execute are not yet calculated.

This can be resolved by making each thread do some additional work. For example, a thread could be responsible of calculating more than one voxel or responsible for calculating the contribution of two or more independent pulses.

We chose to make each thread calculate the contributions of all the channels of a pulse. This helps reduce the instruction stalling factor as we introduce more instructions, independent with the previous ones, helping the compiler rearrange them to achieve higher throughput.

Moreover, by having each thread calculating the contribution of all the channels of a pulse, we reduce the required memory bandwidth, as we now need to read the voxel position only once and not multiple times.

Data Locality Until now, all our approaches use 1D thread grids. Using a 2D approach, we can achieve better data locality as neighbouring voxels tend to access close range bins. Figure 4.9 illustrates the difference of using 1D grids and blocks instead of 2D. As we can see the threads of the 2D approach are more close to each other while in the 1D approach they are spread.

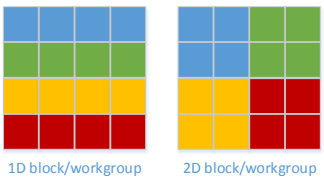


Figure 4.7: Thread block/workgroup dimensionality

By changing the grid and block dimensionality to 2D we managed to achieve better data locality.

Antenna position on constant memory Both antenna and voxel positions are constant and read-only throughout the program lifetime. When all the threads of a warp access the same element, constant memory leads to beneficial gains.

We perform an output based parallelization scheme which makes all threads read different voxel positions, but operate on the same pulse which is characterized by a single antenna position. Thus, placing the antenna positions on constant memory complies to the usage of constant memory.

Pointer Aliasing Compilers account the possibility of aliasing between different pointers when applying optimizations strategies. This decreases the efficiency of the applied optimizations. The programmer can introduce hints to the compiler in order to limit the effect of pointer aliasing. This can be done by using the keyword *restrict* for pointer declarations, when the pointed memory elements are accessed

through this and only this pointer. If in any case another independent pointer accesses the object this will lead to undefined behaviour.

Automatic Unrolling and Tuning Instead of performing manual loop unrolling in order to achieve better performance and achieve lower instruction stalling factors, one can explicitly indicate to the NVIDIA compiler to perform unrolling using the *#pragma unroll* directive. Usually compilers are more *"clever"* into performing such kind of optimizations.

Unfortunately, sometimes compiler may proceed to conservative or brute unrolling, for this reason tuning and specifying the unrolling factor is possible. We tested our applications with different unrolling factors. Figure 4.8 illustrates the execution time for some of them. We concluded that an unrolling factor of 4 is the most appropriate for our underlying hardware as it provides the lowest execution time.

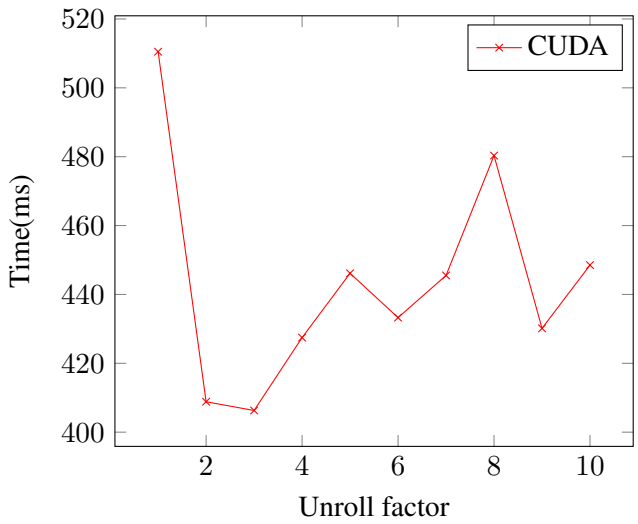


Figure 4.8: Execution time with varying unrolling factor

Vendor Specific Optimizations

Texture Cache Optimization The newer Kepler architectures introduce a new, 48KB read-only data cache, which targets unaligned access patterns for data that are meant to be read-only during the execution of the program. Unable to exploit access patterns lead us into utilising this new cache for accessing the range compressed data.

Use of this cache can be automatically done by the compiler by specifying the specific input data as *const __restrict__* or by using the *__ldg* intrinsic.

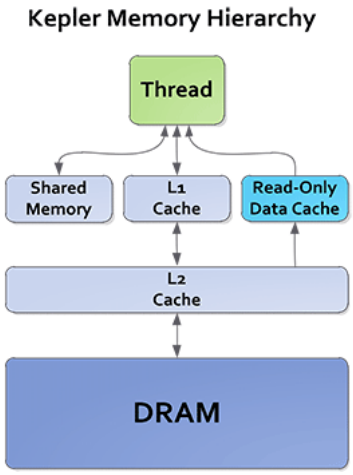


Figure 4.9: Kepler cache hierarchy

Re-evaluate "Roof" Figure 4.10 depicts the roofline model after having applied all the optimisation steps. The operational intensity is increased to:

$$\frac{P \cdot M \cdot N_x N_y \cdot 41}{P \cdot M \cdot N_x N_y \cdot 9} = \frac{41}{9} = 4.56 \tag{4.17}$$

We can conclude that the application is still memory bound, something that is also backed up by the statistically obtained results. The profiler reports a performance of 1371 GFlops/s, close to the one that the roofline model reports, something that leads to a device utilization of 30%

Finally, all the optimization results are gathered in Table 4.3. T reports the execution time, S the speedup over the reference solution, S_r the relative speedup of the solution over the previously reported implementation and finally, GBP/s the number of giga back-projections per second.

Optimizations Scalability

We performed each optimization with different grid sizes and varying input data (see Figure 4.11 and Figure 4.12). As expected most of the optimizations had the same efficiency, with some small absolute variability, independent of the output size. On the other hand, data coarsening and constant memory efficiency seems to vary with the output size. The first one is due to L2 caching limitations as all the global accesses are cached in L2 memory, while the second depends on the warp number and context switch overhead.

In addition, for different interpolation factors data coarsening, data locality and texture cache optimizations efficiency increase as threads access data closer to each other.

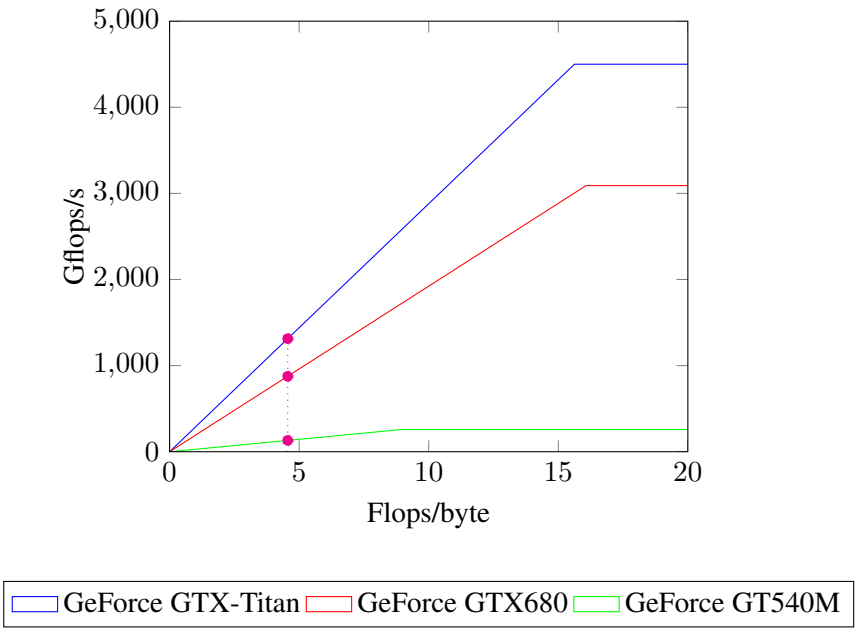


Figure 4.10: Roofline model after applying optimizations

Solution	T[s]	S	S_r	GBP/s
Reference	1276	1	-	0.12
Naive	4.57	279.2	1	3.51
Compiler Flags	3.63	351.5	1.26	4.43
AoS vs SoA	-	-	-	-
Coarsening	1.58	807.6	2.3	10.18
Data Locality	1.3	981.5	1.22	12.37
Constant Memory	0.73	1747.95	1.78	22.03
Pointer Aliasing	0.51	2501.96	1.43	31.53
Texture Cache	0.50	2502	1.02	32.16
Unrolling	0.406	3142.86	1.23	40.01

Table 4.3: Optimization Results

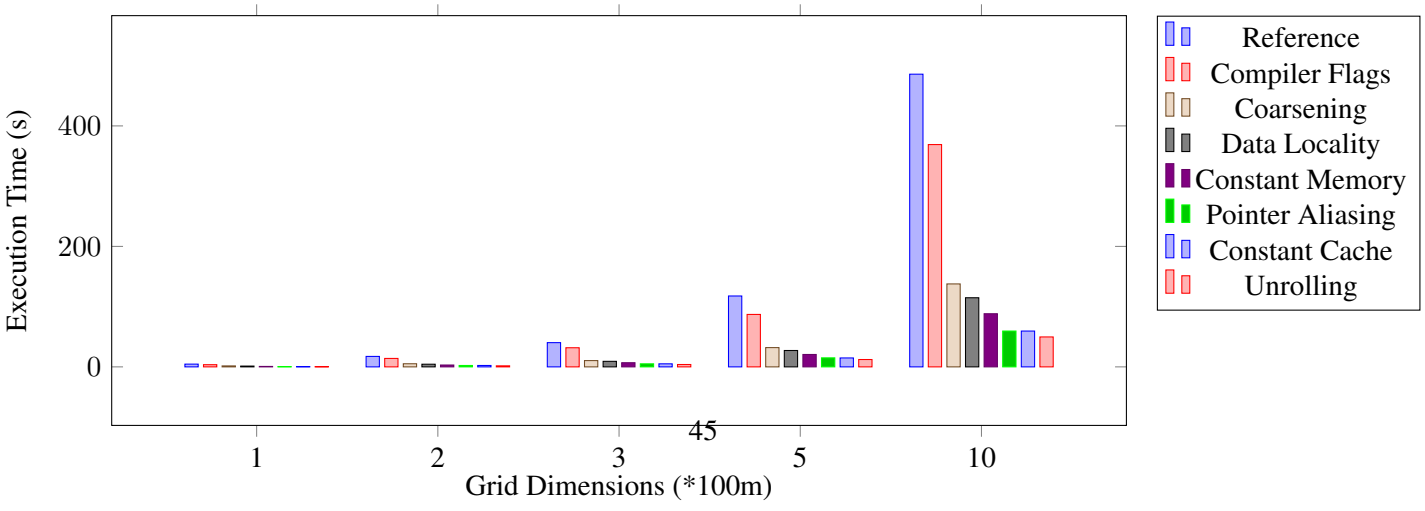


Figure 4.12: Optimization impact on execution time

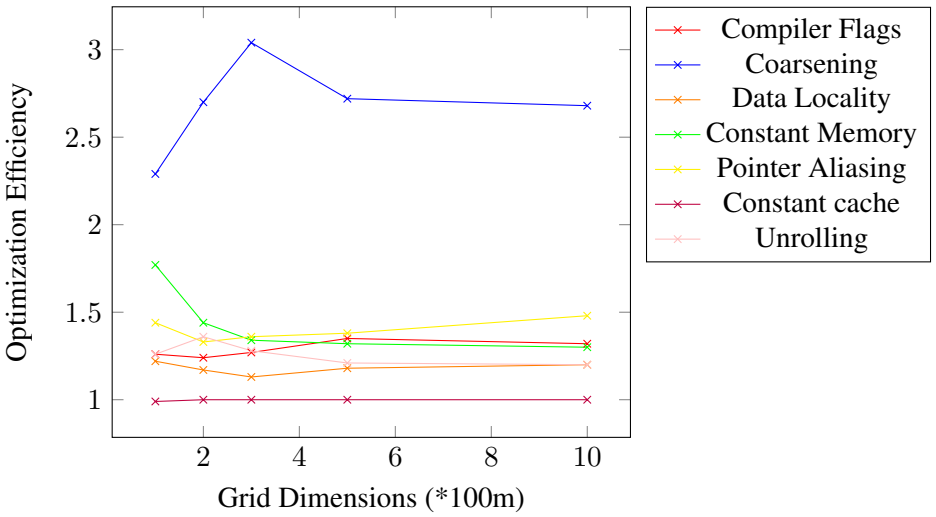


Figure 4.11: Optimizations efficiency

4.5.3 Task-level parallelism

Until now, we have focused on fine-grain parallelism, trying to optimise the performance of individual tasks. When tasks are independent with each other, exploitation of task-level parallelism is feasible. Mapping algorithms on hardware accelerators with dedicated memory usually comes at cost. As GPUs target data-level parallelism usually have to operate on large input data which have to be transferred on its memory. Thus, a communication overhead required for data transfers between the host and the accelerator's memory is inevitable. A common optimization applied to such a problem is to overlap any memory transfers with independent computation workload.

Multi-channel back-projection performs three main tasks: memory transfers, range compression and back-projection. The memory transfer task is responsible for copying the appropriate pulses and antenna position on the memory of the GPU. Range compression is then applied on the transferred pulses using FFTs and finally, back-projection is performed with the pulse compressed data as an input. We observe that a dependency exists between the tasks, meaning that in order for back-projection to be performed, pulse compression has to have taken place in advance and memory transfer before that. Consequently, overlapping memory transfer with range compression can lead to invalid results as the new pulses will overwrite the existing ones before range compression ends. On the other hand, while back-projection is performed, the new data can be prefetched and be ready for the range compression step.

The top part of Figure 4.13 illustrates two iterations of multi-channel back-projections on a single stream while in the bottom a parallel execution is shown with three distinct streams, each one dedicated to a task. Arrows represent depend-

encies between instructions and the vertical lines synchronization points.

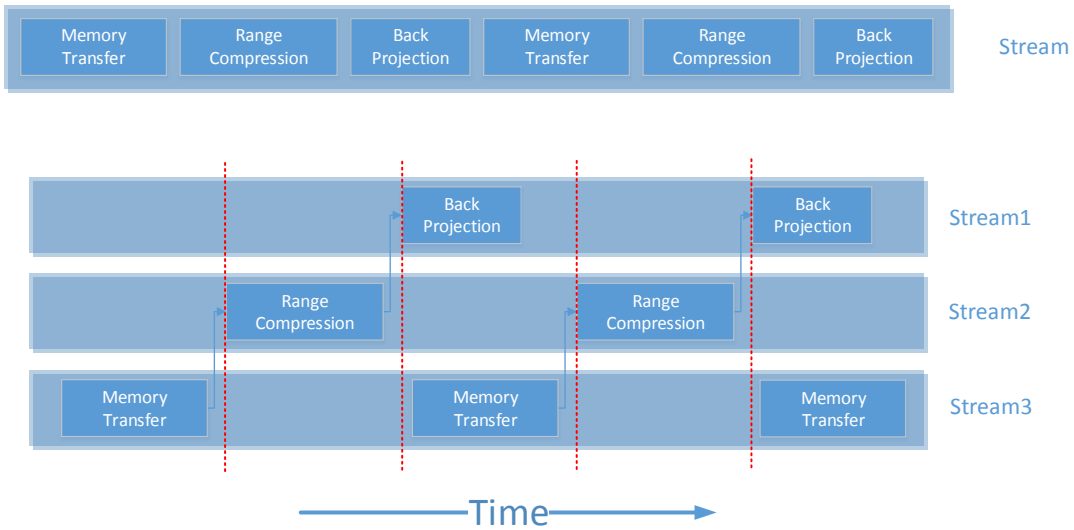


Figure 4.13: Task parallelism on multi-channel back-projection

With this approach we managed to hide a fraction of the memory transfer overhead. The exact amount depends on the execution time required for memory transfer and back-projection. In our NVIDIA Titan platform, back-projection is faster compared to memory transfer thus 75% of the communication overhead can be hidden. On the other hand, on slower platforms all the memory transfer overhead can be compensated.

4.6 Evaluation

4.6.1 Scalability

The computation time required for back projection scales roughly linearly with the number of output voxels.

Moreover, for a state-of-the-art GPUs with $6GB$ of memory, voxel grids of up to $750Mpixel$ can be supported. For larger grids, back-projection has to be performed multiple times, each time operating on smaller sub-grids. A feature already supported by our implementation.

4.6.2 Output Quality

Originally, back-projection uses double precision arithmetic. Migrating to single precision arithmetic led to an absolute error compared to the double precision solution, but also to a reduction of the execution time roughly by a factor of two. Fortunately, the Peak signal-to-noise ratio (PSNR) of the final image is more than *30db*

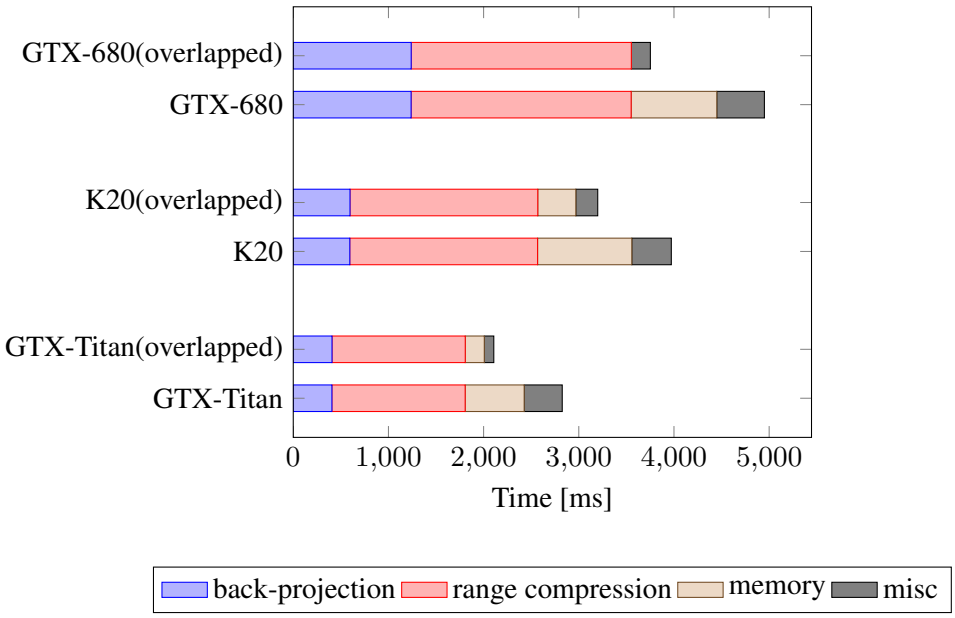


Figure 4.14: Overlap impact for different devices (grid size of 100 by 100 m)

which is acceptable in image processing. Moreover, quality degradation is rarely visible. Better quality can be achieved by performing some of the calculations using double precision arithmetic (e.g. range and one-way shift calculation) under a small performance degradation.

4.6.3 Why both OpenCL and CUDA?

CUDA is more mature and targets NVIDIA GPUs. For this reason using CUDA helps one squeeze more performance when using NVIDIA hardware accelerators. Furthermore, the CUDA ecosystem is well supported, with enhanced debugging and profiling capabilities, something that clearly increases productivity.

On the other hand, OpenCL is an attractive solution when multiple hardware solutions have to be supported and/or compared. In addition, most mobile companies designing GPUs support OpenCL, something that makes OpenCL an attractive solution for embedded systems. Unfortunately, the OpenCL ecosystem is quite immature, making application development more time consuming.

Overall, when addressing the same hardware and no vendor specific optimizations are performed, CUDA and OpenCL should be able to achieve more or less the same performance and mainly depends on the driver efficiency and not on any framework limitations.

4.7 Related Work

Two dimensional SAR image reconstruction is a well researched topic. Multiple approaches exist utilizing either or both the time and frequency domain.

Real-time FGPA-based [11, 14, 28, 32, 40] and GPU-based [27, 29, 30, 71] solutions exist but they mostly operate on the frequency domain.

Concerning back-projection, different algorithmic modifications have been proposed [8], to increase its accuracy with a computing time of the same order.

In [63], the Cell architecture is utilized achieving, a total of 177 million back-projection per second.

In addition, [31] presents a multi-node implementation based on linear interpolation, where each node consists of one Intel Xeon processor and two Xeon Phi accelerators. Each node is capable of 35 billion back-projections.

Moreover, multiple CUDA based solution have been proposed [3, 10, 15, 18, 72]. In [3], using a linear interpolator, authors can achieve gigapixel image reconstruction within minutes, while in [8] better accuracy is achieved compared to different interpolators with a small performance loss.

OpenCL solutions also exist and have been presented in [24, 35] but target different SAR domains.

In this chapter we presented a multi-channel back-projection implementation based on [8, 63] with increased accuracy and the ability of performing up to *40 billion back-projection per second* on a single NVIDIA GTX-Titan GPU.

Chapter 5

Multi-node Multi-GPU Back-Projection

In this chapter, a multi-node multi-GPU back-projection implementation is thoroughly analysed to investigate the problem's scalability of the ability to reconstruct high resolution images. Moreover, potential bottlenecks and limitations are presented.

5.1 Approaches

Due to the *"embarrassing"* level of parallelism that back-projection exposes, acceleration using multiple nodes and multiple hardware accelerators is feasible. Figure 5.1 illustrates a system with multiple computation nodes and multi-GPUs.

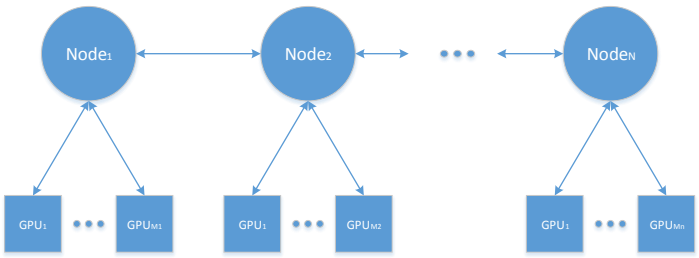


Figure 5.1: Multi-node Multi-GPU implementation

As proposed in [22], which is now expanded for multiple nodes and multiple GPUs, there are two different ways to tackle this problem. Each one has its own advantages and limitations.

5.1.1 Input-based parallelization

In this approach we distribute the input data across the nodes and each node further to its GPUs. Each GPU performs back-projection with its assigned data and creates a partial output. All the partial outputs are then collected by the node which is also responsible for coherently summing them. The final step is to gather the results of all the nodes to a single node and reduce them to construct the final output.

5.1.2 Output-based parallelization

In the output based approach each node is responsible for fully computing a part of the output image. As a result, all the input data are distributed to all the nodes and further to all the GPUs. Interesting is that no reduction or further calculations have to take place. The final step is to collect all the portions of the output along the nodes to a single node and as a result construct the final image.

5.2 Model

Symbol	Description
$\#GPUs$	Number of GPUs per node
$\#Nodes$	Number of nodes
$S_\star$	Size of the $\star$ component
$T_\star$	Time required for $\star$ operation
$T_{comm}(S_\star)$	Time required to transfer the $S_\star$ data
$f(\star)$	Function with $\star$ as independent variables

Table 5.1: Symbols used in the model

5.2.1 Single Node - Single GPU implementation

In general the execution time of the serial implementation over one GPU is the summation of the time required for computation and communication. The communication part accounts the time required for the data to be transferred to and from the GPU.

$$T_{ser} = T_{comp} + T_{comm} \tag{5.1}$$

The communication between two devices has two distinct parts, T_L which is the latency between these devices and depends from their distance as well as of the physical properties of the underlying communication interface and the time required to transfer the data T_D, which depends on the size of the data that have to be transferred S and the bandwidth of the intermediate channel B.

$$T_{comm} = T_L + T_D, where \quad T_D = \frac{S}{B} \tag{5.2}$$

5.2.2 Input-based Multiple Node - Multiple GPU parallelization

As we mentioned splits all the input data along the available GPUs. Each GPU calculates a partial output voxel grid by using the assigned input data. At the end, all the output voxel grids are gathered and reduced to a single one. This method keeps steady the time required by memory transfers but imposes a reduction overhead. Figure 5.2 illustrates this approach.

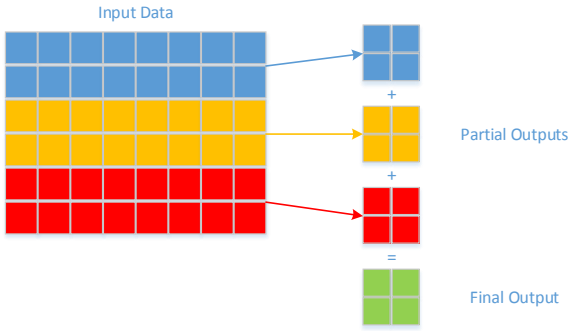


Figure 5.2: Input-based parallelization

Equation 5.3 specifies the execution time of this parallel approach, which is the summation of the time required for computations, the time required for communication and the time required for the output reductions.

$$T^1_{par} = T^1_{comp} + T^1_{comm} + T^1_{reduction} \tag{5.3}$$

In more detail, as the GPUs run in parallel each one processing its part, the final computation time is the maximum of the execution times of each individual GPU (see Equation 5.4).

$$T^1_{comp} = \max(T_{GPU_i}) \tag{5.4}$$

Because we may run the parallel algorithm on multiple nodes, each one with multiple GPUs, the communications time is split in two parts, the communication time inside a node and the communication time across nodes (see Equation 5.5).

$$T^1_{comm} = T^1_{comm_{node}} + T^1_{comm_{nodes}} \tag{5.5}$$

The communication inside a node has two distinct parts, the time required to distribute the already reduced input data to the GPUs and the time required to transfer the partial calculated outputs from the GPUs to the node for the reduction process. As a result, the total communication time inside a node is a function of the number of nodes, something that affects the size of the input data, the number of the GPUs and the size of the output data (see Equation 5.6).

$$
\begin{aligned}
T^1_{comm_{node}} =& T^1_{input} + T^1_{output} = f(\#GPUs, \#Nodes, S_{input}, S_{output}) \\
=& \frac{T_{comm}(S_{input})}{\#Nodes} + \#GPUs \cdot T_{comm}(S_{output})
\end{aligned} \tag{5.6}
$$

On the other hand the communication across the nodes depends on the number of nodes, the size of the input data and the size of the output voxel grid (see Equation 5.7).

$$
\begin{aligned}
T^1_{comm_{nodes}} =& f(\#Nodes, S_{input}, S_{output}) \\
=& T_{comm}(S_{input}) + (\#Nodes - 1) \cdot T_{comm}(S_{output})
\end{aligned} \tag{5.7}
$$

Finally, the time required for the reduction of the partial outputs is also split in two parts, the time required to perform reduction inside a node and the time require to perform reduction across nodes (see Equation (5.8)). Reduction inside a single node depends on the number of GPUs and the size of the output solution (see Equation (5.9)). On the other hand, reduction across nodes depends on the number of nodes and the size of output solution (see Equation (5.10))

$$
T^1_{reduction} = T^1_{reduction_{node}} + T^1_{reduction_{nodes}} \tag{5.8}
$$

$$
T^1_{reduction_{node}} = f(\#GPUs, S_{output}) \tag{5.9}
$$

$$
T^1_{reduction_{nodes}} = f(\#Nodes, S_{output}) \tag{5.10}
$$

We have to point out that in case of overlapping of the computations and the communication inside a node the execution time is reduced to the maximum time between the computation and the inter-node communication along with the time required for across-node communication and the reduction time (see Equation 5.11).

$$
T^1_{par} = \max(T^1_{comp}, T^1_{comm_{Node}}) + T^1_{comm_{nodes}} + T^1_{reduction} \tag{5.11}
$$

5.2.3 Output-based Multiple Node - Multiple GPU parallelization

The output-based parallelization splits the output elements over the GPUs, which means that each GPU is responsible for calculating a part of the output voxel grid. Figure 5.3 illustrates this approach. The benefit of this approach is that we overcome the time required for reductions (see Equation (5.12). On the other hand, we have to transfer a larger amount of data. In order for each GPU to calculate the final value of the assigned part of the output voxel grid, all the input data have to be processed by each GPU.

$$
T^2_{par} = T^2_{comp} + T^2_{comm} + T^2_{reduction} = T^2_{comp} + T^2_{comm} \tag{5.12}
$$

In more detail, as the GPUs run in parallel each one processing its part, the final computation time is the maximum of the execution times of each individual GPU (see Equation 5.13).

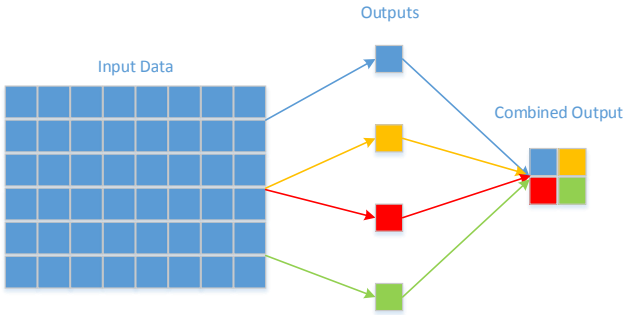


Figure 5.3: Output-based parallelization

$$T^2_{comp} = \max(T_{GPU_i}) \tag{5.13}$$

Because we may run the parallel algorithm on multiple nodes, each one with multiple GPUs the communication time is split in two distinctive parts, the communication time inside a node and the communication time across nodes (see Equation 5.14).

$$T^2_{comm} = T^2_{comm_{node}} + T^2_{comm_{nodes}} \tag{5.14}$$

The communication inside a node depends on the number of Nodes, GPUs, the size of the input data and the size of the output (see Equation 5.15). The input has to be processed by all the GPUs, so the communication time for distributing the input (when no overlapping exists) is equal to time time required to send the input to a single GPU multiplied by the number of GPUs. In addition, as each GPU processes a part of the output, the total communication cost for the output gathering after the completion of the back-projection equals to the number of voxels that have been assigned to the specific node.

$$\begin{aligned} T^2_{comm_{node}} =& f(\#GPUs, S_{input}, S_{output}) \\ =& \#GPUs \cdot T_{comm}(S_{input}) + \frac{T_{comm}(S_{output})}{\#Nodes} \end{aligned} \tag{5.15}$$

The communication cost for communication across nodes depends on the size of the input and the number of nodes, because the input has to be broadcasted to all the nodes. Moreover, in the end all the calculated voxels have to be collected from all the nodes.

$$T^2_{comm_{nodes}} = f(\#Node, S_{output}, S_{input}) = (\#Nodes - 1) \cdot T_{comm}(S_{input}) + T_{comm}(S_{output}) \tag{5.16}$$

It is important to mention that for a broadcasting operation (e.g. used in MPI), the time required is not linear with the number of nodes that the message has to be sent to. Advanced algorithms are used to reduce the time required for broadcasting.

Most of them use a tree-based approach where the time required is equal to :

$$T_{broadcast} = f(\#Nodes, S_{data}) = \log_2(\#Nodes) \cdot T_{comm}(S_{data}) \quad (5.17)$$

As with the previous approach, if overlapping of the computations and the communication inside a node exists, the execution time is reduced to the maximum time between the computation and the inter-node communication along with the time required for across-node communication (see Equation 5.18).

$$T^2_{par} = \max(T^2_{comp}, T^2_{comm_{node}}) + T^2_{comm_{nodes}} \quad (5.18)$$

5.2.4 Approach Comparison

Both approaches have their own advantages and when having to choose among them, careful consideration of the system's input and output requirements is necessary. The number of GPUs, the number of computational nodes and the size of the input and output data are of high importance.

When having large output images and the overhead of communication and reduction is high then the output-based approach is preferred. On the other hand, for small output sizes the input-based approach is preferred as the reduction overhead is negligible.

For an embedded environment where a small number of nodes is deployed and usually the output data size is small in order to achieve real-time response the most appropriate approach is the first one.

5.3 Communication Analysis

We distinguish here two data distribution policies: the block distribution and the block-cyclic distribution.

In block distribution the data are distributed in blocks across the nodes, while on the other hand, in block-cyclic distribution the input data are distributed in a cyclic manner across the nodes.

We see no clear advantage with any of these two. The most important parameter in tuning the performance of the overlapping is the granularity of the partition, which can be tuned in both solutions. Therefore, we choose block-based partitioning for its low complexity in terms of coding.

5.4 Implementation

The implementation procedure can be split in two stages: the multi-GPU stage and the multi-node stage.

In the multi-GPU stage, the multi-channel back-projection is parallelized across the available GPUs. This can be performed either by using a multi-threaded approach where each thread is responsible for a given number of GPUs or by us-

ing multiple compute streams (CUDA) or command queues(OpenCL), where each stream is associated with one of the available GPUs. For CUDA and OpenCL implementations the second approach is preferred as it is less error-prone and moreover it reduces the thread creation and handling overhead.

In the multi-node stage, one node is responsible for coordinating and transferring the appropriate data to the remaining nodes. In the input based parallelization approach parts of the input data are distributed across the nodes, while in the output base approach all the input data are transferred to all the nodes through broadcast. For the distributed multi-node environments MPI is used.

The model itself can be used to determine the performance for a given experiment on a given configuration. To do so, some data have to be provided such as, the time required for transferring the input and output data, or instead, some specifications about the transferring rate of the communication interface and the sizes of the input and output data. In addition the time required for performing back-projection on a single machine has to be provided.

One could easily modify the model in order to predict the number of GPUs (same hardware model) and their configuration (intra- or inter-) that would be best for an experiment of given size. Still some appropriate information have to be provided along with bounds such as the maximum number of the inter-node GPUs.

5.5 Evaluation

Unfortunately, our experimental setup did not contain multiple nodes with more than one GPUs. For this reason we evaluated our implementation in two steps.

First, we evaluated the multi-GPU environment under a single node and second, the multi-node one under multiple nodes consisting of one GPU. The multi-GPU environment consists of two *NVIDIA Tesla K10*, one *NVIDIA Tesla K20x* and one *NVIDIA Tesla K40c* GPUs. On the other hand each node in the multi-node processing environment consists of a *NVIDIA GeForce GTX480* GPU.

5.5.1 Multi-GPU Evaluation

For the multi-GPU implemenation both the input based and output based approaches were implemented.

Input-based Approach

Figure 5.4 illustrates the results of two experiments of different voxel grid sizes. As the node consist of multiple GPUs each one having different compute capabilities we first executed the application on each one independently and then to all of them together. As we can see the total execution has been improved and is actually closest to the execution time required from the slowest GPU to run its assigned part, something that verifies our model.

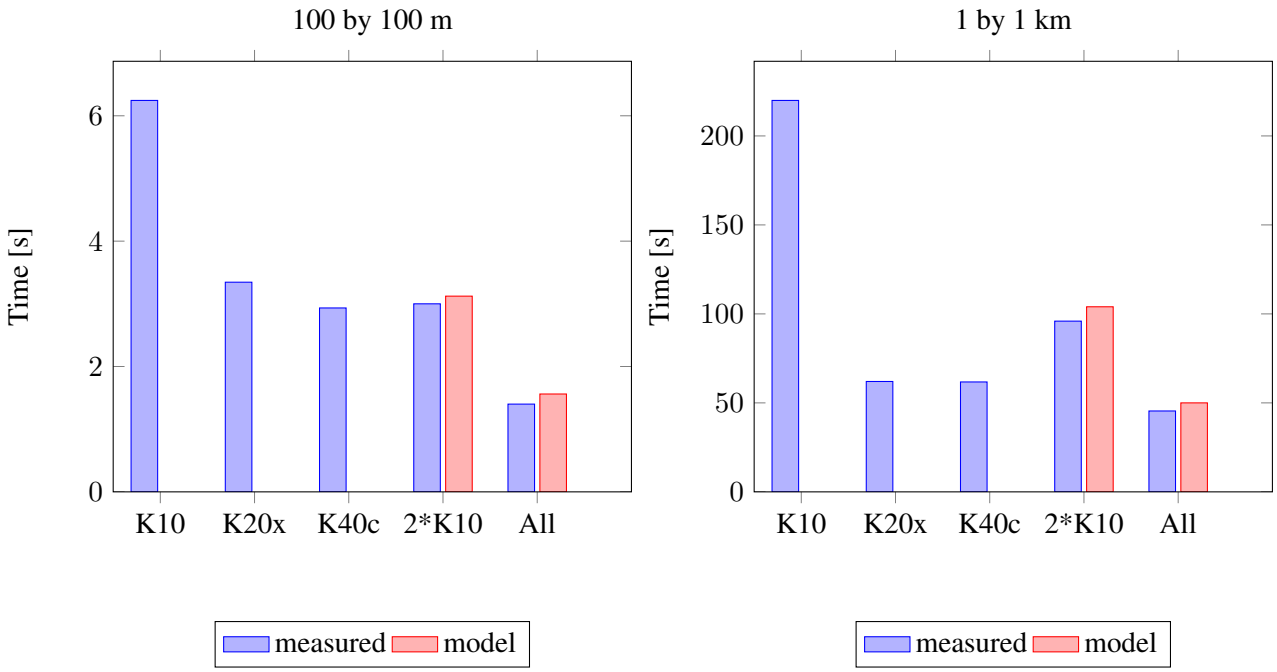


Figure 5.4: Expiremental Results for the input-based multi-GPU approach

Output-based Approach

Figure 5.5 illustrates the results of the two experiments under the output-based parallelization. We notice that the execution time is significantly higher for smaller grid sizes due to increased number of memory transfers. This can be partially compensated by using memory-compute overlap techniques. Moreover, for larger grid sizes the performance seems to be worse, we suspect that this is due to the block geometry we imposed. For this reason we can also see a large difference from the predicted execution time. Better blocking sizes which take into account the GPU configuration may lead to better results. We strongly believe that this approach leads to lower execution times compared to the input-based approach for larger grids, because it decrease the problem size and achieves better locality. Thus, autotuning for better suitable block geometries is the next step to optimize this solution.

5.5.2 Multi-node Evaluation

For the multi-node approach only the input-based parellization was implemented. We performed two different experiments: one with a small voxel grid and one with a much larger one. Our main purpose was to investigate the scalability and behaviour of the parallel implementation.

Figure 5.6 presents the results of the first experiment. For the first experiment

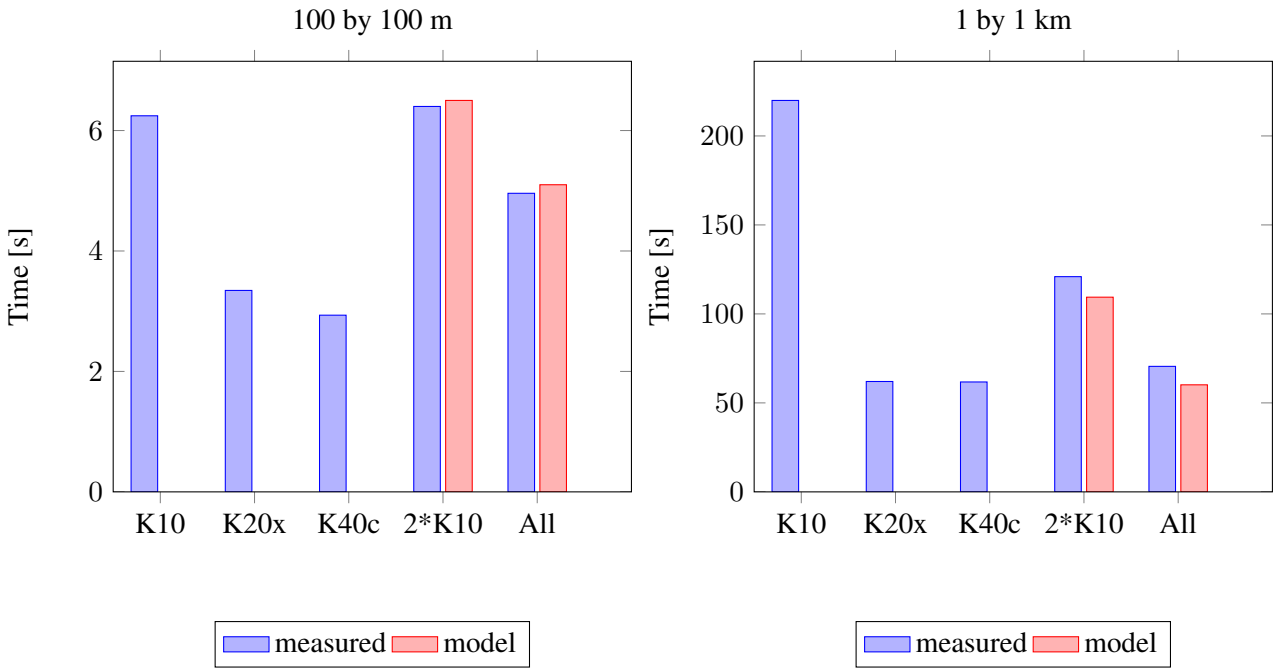


Figure 5.5: Expiremental Results for the output-based multi-GPU approach

we used a small voxel grid with 100 m in each dimension and the highest possible resolution. The left figure presents the execution time of the experiment when varying the number of nodes. It includes both the prediction of the model along with the actual measured execution time when accounting the communication and reduction overhead. Moreover, the right figure illustrates the achieved speedup.

We notice that for such problem sizes where the communication overhead is too high compared to the actual execution time, the speedup is too low. The maximum achieved speedup is 2.5 when using 16 GPUs. In addition, we do not perform any communication and computation overlap at node level, something that could increase by a small portion the achievable speedup.

Figure 5.7 presents the results of the second experiment. For the second experiment we used a much larger voxel grid with 1km in each dimension and the highest possible resolution. Again, the left figure presents the execution time of the experiment when varying the number of nodes, while the right one illustrates the speedup of the experiment.

We notice that for larger problem sizes the communication and reduction overhead is almost completely compensated. Moreover, for eight nodes or less the speedup is linear. On the other hand, for more nodes the speedup is below the ideal due to the congestion that the master node experiences for the final reduction of the outputs of each individual node. In addition, we have to point out that the reduction cost for such large outputs can be reduced by using multi threading and SIMD approaches or more advanced reduction schemes (e.g. tree based reduction)

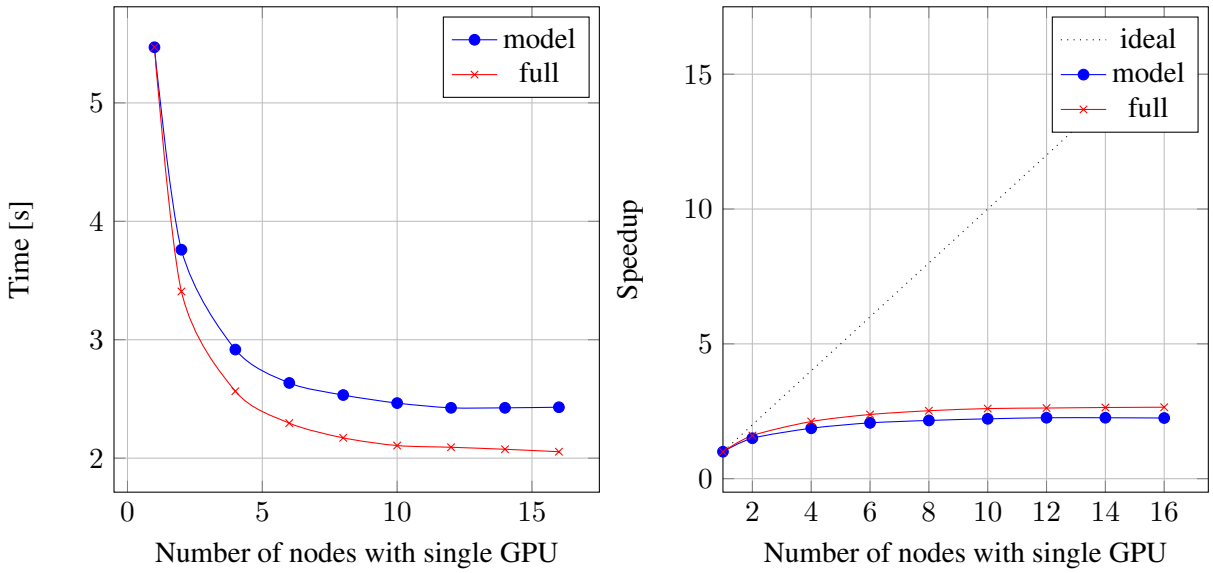


Figure 5.6: Expiremental Results for a 100 by 100 meter voxel grid

in order to reduce both the communication and computational cost.

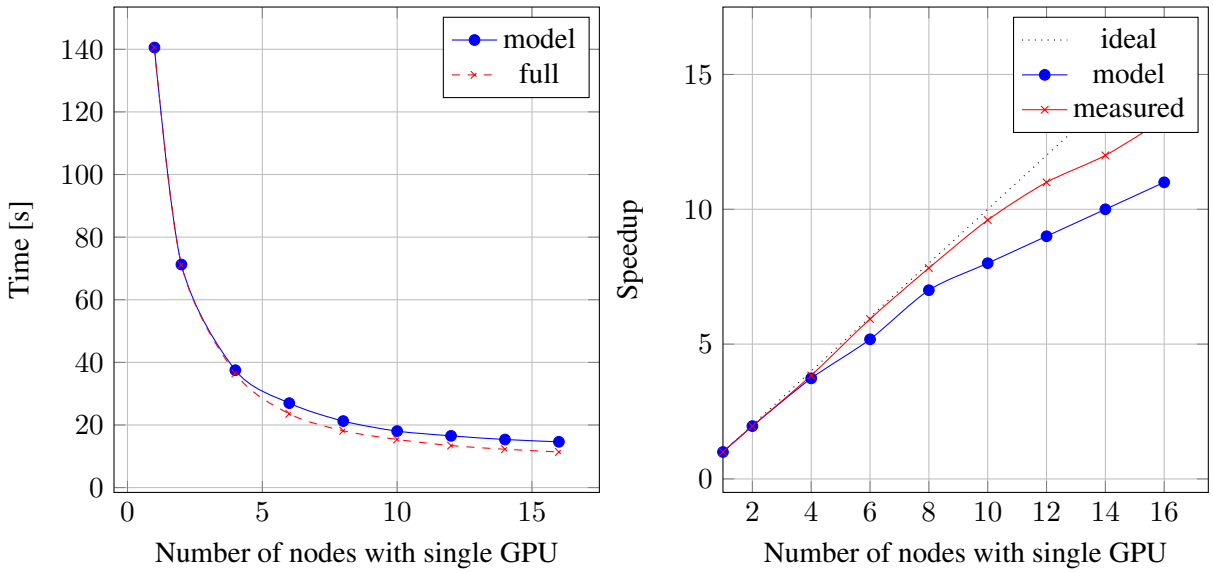


Figure 5.7: Expiremental Results for a 1 by 1 km voxel grid

Overall, in our experiments, we have shown a grid of 1km in each dimension (aproximately 50 MPixels) being processed in only 7 seconds. These results show that gigapixel image reconstruction can be indeed performed in a few minutes. We also point out that the bottleneck of this solution remains the data transfer. Thus,

multi-node approaches are preferred when large voxel grids have to be processed.

5.6 Related Work

In [31] an implementation with multiple nodes and multiple Xeon Phi accelerators is presented, achieving a performance of 35 billion back-projections per node. Moreover, in [9] a multi-GPU implementation using the NUFFT approach is presented. Finally, in [3] an output-based multi-node multi-GPU implementation of back-projection is implemented and is capable of gigapixel image reconstruction using nodes consisting of Xeon X5660 processors and Tesla C2050 GPUs. Our implementation is different and outperforms previous attempts. Moreover, we are the first to propose a model to evaluate and predict the performance of multi-node multi-GPU back-projection on modern cluster-like architectures.

Chapter 6

Decimation and Channel Reduction

6.1 Decimation

Decimation or *Downsampling* [33] is the process of reducing the frequency of a signal. Decimation operates in the discrete-time domain and creates a new discrete signal $y[n]$ using sub-samples of another signal $x[n]$.

Decimation by an integer factor M reduces the sampling frequency from f_s to $\frac{f_s}{M}$. The main concept is for every M samples to discard $M-1$ of them. An anti-aliasing filter $h[n]$ is then applied to avoid aliasing effects from high frequency components (see Equation 6.1).

$$y[n] = \sum_{k=0}^{K-1} h[k] \cdot x[nM - k] \tag{6.1}$$

6.1.1 Requirements

Decimation reduces the data rate of the system. The data rate of a system is given by Equation 6.3 depends on the the sampling frequency f_s, the size of the samples n_r (in bits) and the number of channels m.

$$\text{Data rate} = m \cdot f_s \cdot n_r \quad [Mb/s] \tag{6.2}$$

Decimation reduces the data rate by a decimation factor k.

$$\text{Data rate}_{dec} = \frac{\text{Data rate}}{k} \quad [Mb/s] \tag{6.3}$$

Decimation requirements in terms of computational power are quite low. For our radar hardware they are approximately 0.1 GFlops/s per channel.

6.1.2 Evaluation

Decimation can lead to a significant reduction of the input data rate and size. On the other hand, by reducing the number of samples, we reduce the accuracy and as a result the quality of the system's output.

To be more specific, a decimation step in our system reduces the number of samples we have to integrate over the fast-time domain. As the calculations on the fast-time domain take place in the frequency domain, such an operation will reduce the size of the FFTs that have to be performed during the range compression step.

As presented in Chapter 4, the computational demands of the range compression step depend on the number of sweeps, the number of channels, the number of range samples we have to integrate on, and it is independent of the size of the output image.

As a result, by performing decimation, we only reduce the time required for performing range compression and not the time required for the calculation of the reflectivity contributions. Table 6.1 presents the execution time of the range compression process when using different interpolation factors for 29120 samples per channel and 1506 sweeps in the slow-time domain. Moreover, Figure 6.1 illustrates the impact of the upsampling factor to the quality of the output image.

Important is the fact that with lower upsampling factors, the performance of the contribution calculation was also improved. The main reason is the tighter data locality that is achieved, as the range differences between consecutive voxels are getting smaller.

For a system that needs to calculate the radar reflectivity over small grids this may lead to a significant reduction of the execution time, but when large voxel grids have to be constructed, the speedup is minimal.

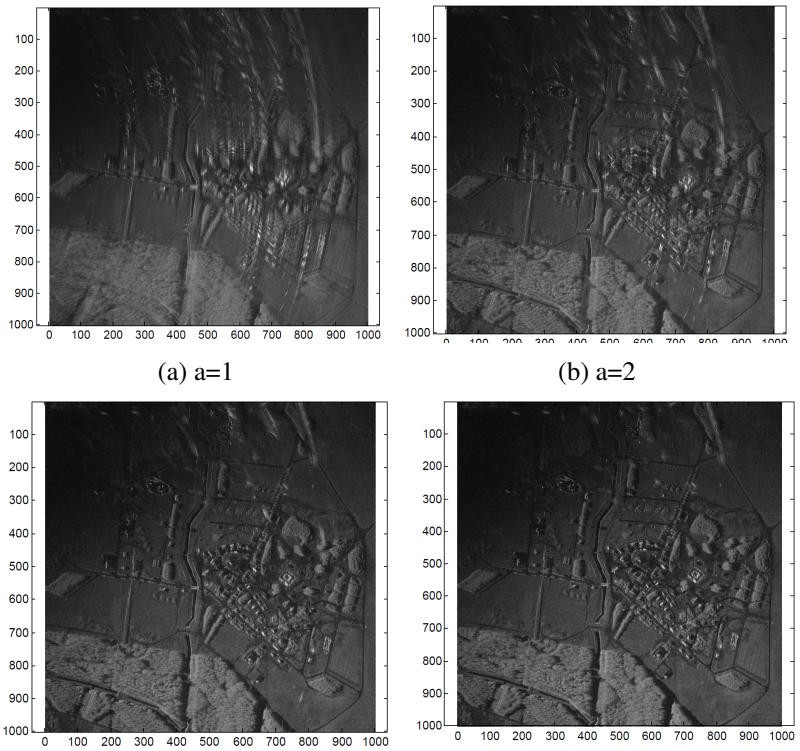
Interpolation factor	GT540M	GTX-Titan
1	2.01	0.195
2	3.68	0.376
4	8.37	0.746
8	16.92	1.493

Table 6.1: Range compression performance (execution time[s]) with varying upsampling factor

6.2 Channel Reduction

Another way to reduce the input data rate is by adding channels either in RF or at digital level. Channel addition reduces the coverage of the radar, and when it is not performed carefully it leads to problems.

Having a large number of channels leads to a wide area coverage, sometimes wider than the area covered by the illumination beam. For this reason, channel



(a) a=1 (b) a=2

(c) a=4 (d) a=8

Figure 6.1: Image quality with varying upsampling factor

reduction may lead to no or small quality degradation depending on the position and the area of the grid we need to reconstruct (see Figure 6.2).

On the other hand, when channel reduction is performed in such a manner, we cannot control the directivity of the new beam. So, if a voxel grid that we want to reconstruct was previously inside the coverage of the receive beams, after channel reduction it may not be, something which will lead to the inability of reconstructing the needed area. Moreover, when the level of channel summation is high and result to new channels with length greater than the wavelength ambiguities may took place, where the origin of the received signal can not be exactly specified.

Other approaches can be used in order to improve the behaviour of channel reduction. One for example could be to change the directivity of the beam by using information obtained by the INS about the position of the radar and the position of the area we want to reconstruct. Such an approach is much simpler to realize during the digital beamforming step and not directly at RF level.

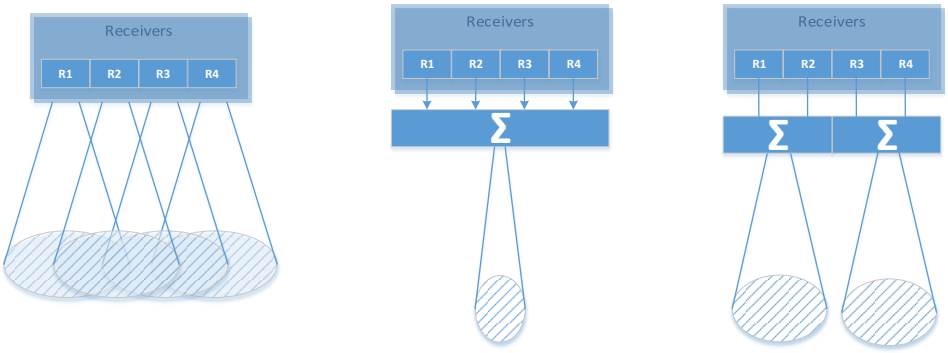


Figure 6.2: Channel Reduction

6.2.1 Requirements

Channel reduction also reduces data rate by a specific factor depending on the number of channels we want to add together.

The maximum computational demands of channel reduction occurs when we want to add all the channels together, where $m-1$ additions have to be performed (see Equation 6.4).

$$\text{Channer Reduction}_{max} = (m-1) \cdot f_s \quad [FLOPs] \tag{6.4}$$

For the radar hardware that we use with 24 channels channel reduction requires less than one MFLOP.

6.2.2 Evaluation

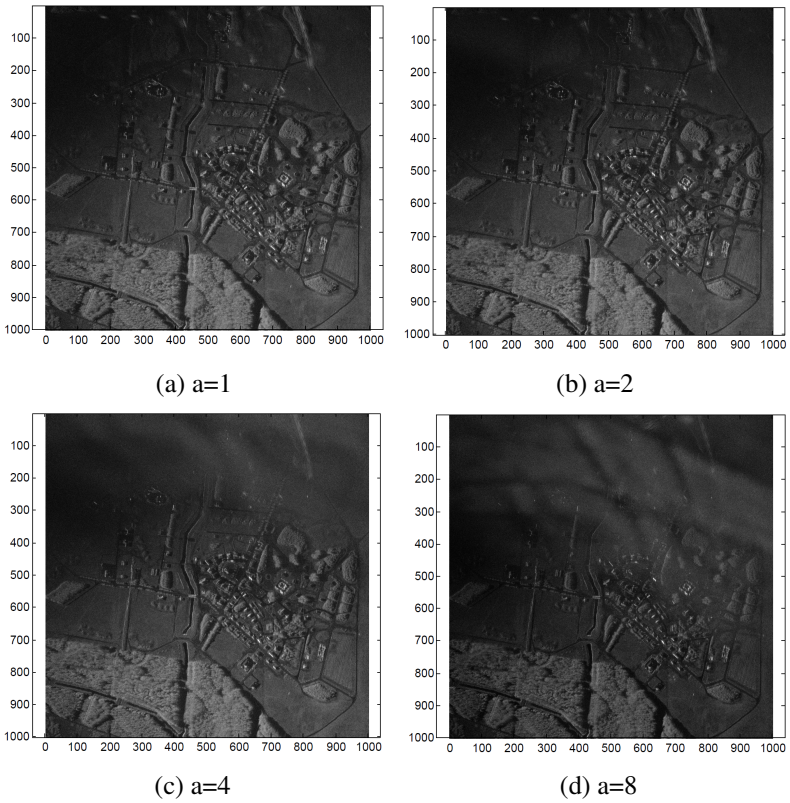
The execution time of Multi-channel back-projection can be significantly reduced through the addition of channels. The number of channels affects the whole processing chain of the multi-channel back-projection: the contributions calculation part and the range compression one.

By reducing the number of channels we reduce the number of pulses that we have to compress over range. In addition, we reduce the number of channels we have to integrate on during the contribution calculation part. Consequently, by reducing the the number of channels by a factor k, we roughly reduce the computations and the memory transfers between the host and the device by the same factor. Table 6.2 presents the performance of the contribution function when channel reduction is performed. Figure 6.3 illustrates the impact of channel reduction to the quality of the output image. When we reduce the number of channels by four we are able to identify the presence of ambiguities. The ambiguities are products of (Doppler) aliasing caused by the effective reduction of azimuth

sampling. Secondly, adding channels creates a narrowing of the antenna receive beam that leads to darkening of some areas

Interpolation factor	GT540M	GTX-Titan
1	11.3	0.491
2	5.7	0.245
3	2.8	0.120
4	1.4	0.061

Table 6.2: Back-projection performance (execution time[s]) with different levels of channel reduction



(a) a=1 (b) a=2

(c) a=4 (d) a=8

Figure 6.3: Image quality degradation due to channel reduction

Chapter 7

Map Drift Autofocus

7.1 Algorithm

Existence of motion errors in SAR systems is almost inevitable. High accuracy INS and GPS systems are expensive and power hungry, something that prohibits their usage in an embedded radar platform with power constraints. Moreover, the accuracy required by a SAR systems strongly depends on the operating frequency of the SAR itself. For example, a SAR with frequency of GHz, requires millimetre resolution from the INS and GPS sensors.

In order to compensate potential motion errors multiple steps of iterative autofocus are performed [7]. Multiple algorithms exist, but in this thesis we focus on correcting large motion errors using the *Map Drift*[6] algorithm.

Figure 7.1 depicts the Map Drift principle.

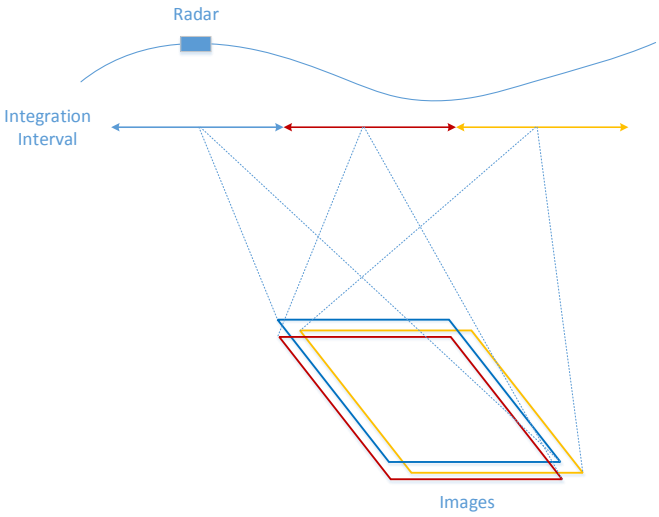


Figure 7.1: Map Drift

In the *Map Drift* algorithm, multiple images are being processed on the same grid using different non-overlapping sections of radar data. This eventually leads to images that are defocused, distorted and displaced with respect to each other. The displacement can be estimated by cross correlating these consecutive images.

It is important to calculate this displacement back to a motion error, which can then be applied on the aircraft track. Performing back-projection again using the same data and the corrected track will lead to a reduction in the absolute value of the total displacement. This procedure continues until there is no more displacement or the error is small enough. Finally, the images are coherently added together to produce the final autofocus images. Figure 7.2 illustrates the processing steps of the map drift autofocus process.

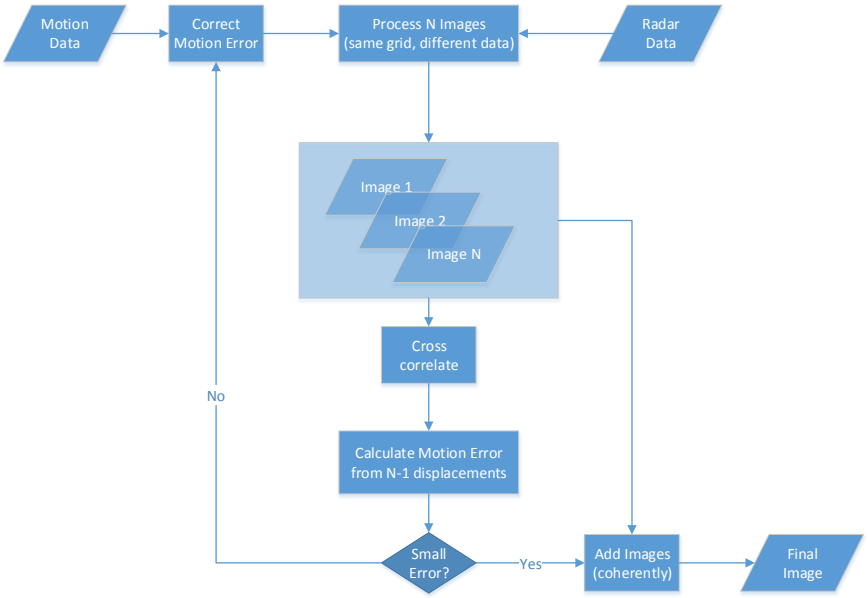


Figure 7.2: Map Drift autofocus process chain

Calculating back to motion is not trivial, but here we propose a simple form, sufficient for our case. We consider the motion only in line-of-sight (see Figure 7.3), so in the direction the radar is looking, where the motion sensitivity is higher, thus we use the displacements in the cross-track direction. We then use a polynomial approximation of the motion error, the order of which is linked to the number of consecutive images used.

7.2 Implementation

In order for our system to support the map drift autofocus algorithm first we had to alter the multi-channel back-projection module to support multiple *"looks"* on

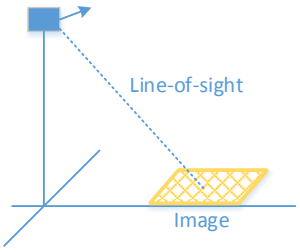


Figure 7.3: Line-of-sight displacement

the same voxel grid. Given the number of *"looks"* that we want to have, and the total integration time, multi-look multi-channel back-projection splits the total back-projection procedure into smaller ones, equal to the number of the required *"looks"*. This step is the most compute intensive step of the autofocusing process, as it contains the back-projection calculation, and has to be performed in every iteration.

Next step is to perform the cross-correlation procedure between the consecutive images. Cross correlation calculation is quite similar to convolution as the main difference between them is a time reversal in one of their inputs. As with linear convolution, there are two distinct ways to compute the linear cross-correlation of two inputs, one is performed in the spatial domain while the other in the frequency domain. Performing cross-correlation in the spatial domain through the sliding method, where we slide the one input over the other leads to high complexity. On the other hand, in the frequency domains FFTs are used in order to speedup the process. Analogous to the convolution theorem, by multiplying the Fourier transform of the one signal with the complex conjugate of the Fourier transform of the other, we obtain the Fourier transform of their correlation (see Equation (7.1)).

$$\mathcal{F}\{f \star g\} = (\mathcal{F}\{f\})^* \cdot \mathcal{F}\{g\} \tag{7.1}$$

Figure 7.4 depicts the cross-correlation process between two consecutive *"looks"*.

Cross-correlation can be performed either on the GPP or the GPU, something that depends on the size of the images that are being processed.

Having calculated the cross-correlation of two images, we are interested to find its *"peak"*. The position of the *"peak"* reveals the displacement between the two images in terms of pixels. This can be translated to meters by multiplying the offset with the radar resolution.

Although, we are interested into correcting only the motion in line-of-sight, by performing a two dimensional cross-correlation we obtain both the offset along-track and cross-track. This gives us the ability for further extending the motion correction process in the future.

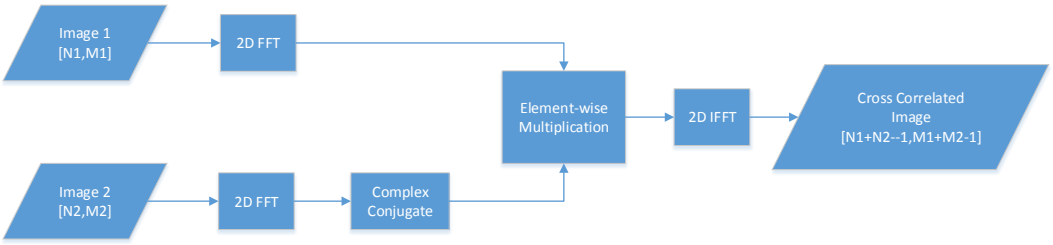


Figure 7.4: Cross-correlation using FFTs

The next step of the process is a polynomial approximation among the displacements. This will lead to obtaining the coefficients of the polynomial, which will give us the track correction of every time step along the integration interval. This correction is then applied on the used track and the back projection is performed again.

7.3 Evaluation

The computational requirements of the *Map Drift Autofocus* alone are quite small, as the most compute intensive part is the cross-correlation process. On the other hand, the fact that is an iterative process where back-projection has to be performed multiple times evidently restricts autofocus utilisation in real time processing. Iterative processes violate the deterministic behaviour that real-time systems have to follow. Clearly, some pre-defined steps can be performed in case is required and the appropriate time budget is available.

We mentioned that FFTs can be performed either on the GPP or the GPU. For images with dimensions less than 4000 pixels the GPP is preferred, while for larger images where the communication overhead can be compensated, the GPU approach leads to faster results. Moreover, interesting is the fact that by performing one dimensional FFTs, instead of a two dimensional FFT leads to faster results although a matrix transposition step has to take place.

The only limitation that *Map Drift Autofocus* has is the size of the images that it is capable of cross correlating. Very large images, along with the intermediate calculations, may not be able to fit the GPU memory. Solutions for this problem can be found but are not currently implemented in this thesis.

7.4 Related Work

Multiple autofocusing techniques targeting SAR have been proposed, map drift is one of them. In this thesis, we optimise and investigate the behaviour and limitations of this algorithm and finally, incorporate it in the whole processing chain. Our

main interest lies in the possibility of performing a number of autofocusing steps at real-time when possible. An implementation for both GPP and GPU is provided and automatically selected depending on the size of the processing images.

Chapter 8

Proposed System Architecture

8.1 System Design

In this chapter we revisit and analyse the requirements of each processing block and we propose a possible hardware mapping along with a custom architecture.

8.1.1 Hardware Mapping

Table 8.1 summarizes the computational demands of the whole system per second of integration when assuming the radar hardware described in Chapter 3, a voxel grid of maximum resolution and size of 100 by 100 meters and an upsampling factor of 8 ($29160 \cdot 8 = 233280$ range samples).

Component		GFlops/s
Decimation		2
Channel Reduction		0.001
Range FFT		75.1
Multi-channel Back-projection		74.1
Map Drift[1]	2D Cross Correlation	0.57
	Polynomial Approximation	0.01
Rest Components		2

Table 8.1: Computational requirements per second of integration

The *"Rest Components"* include the inertial data manipulation, the voxel grid creation and other pre-processing routines.

Figure 8.1 illustrates a proposed hardware mapping of all the processing blocks. Our main target is to perform each block on the most appropriate hardware platform in order to minimize both the computation time and the power requirements.

[1]We do not account the computations required for performing back-projection again. This is purely the execution time for the Map Drift process when having already two input *"looks"*. For every additional *"look"*, another cross correlation has to be performed.

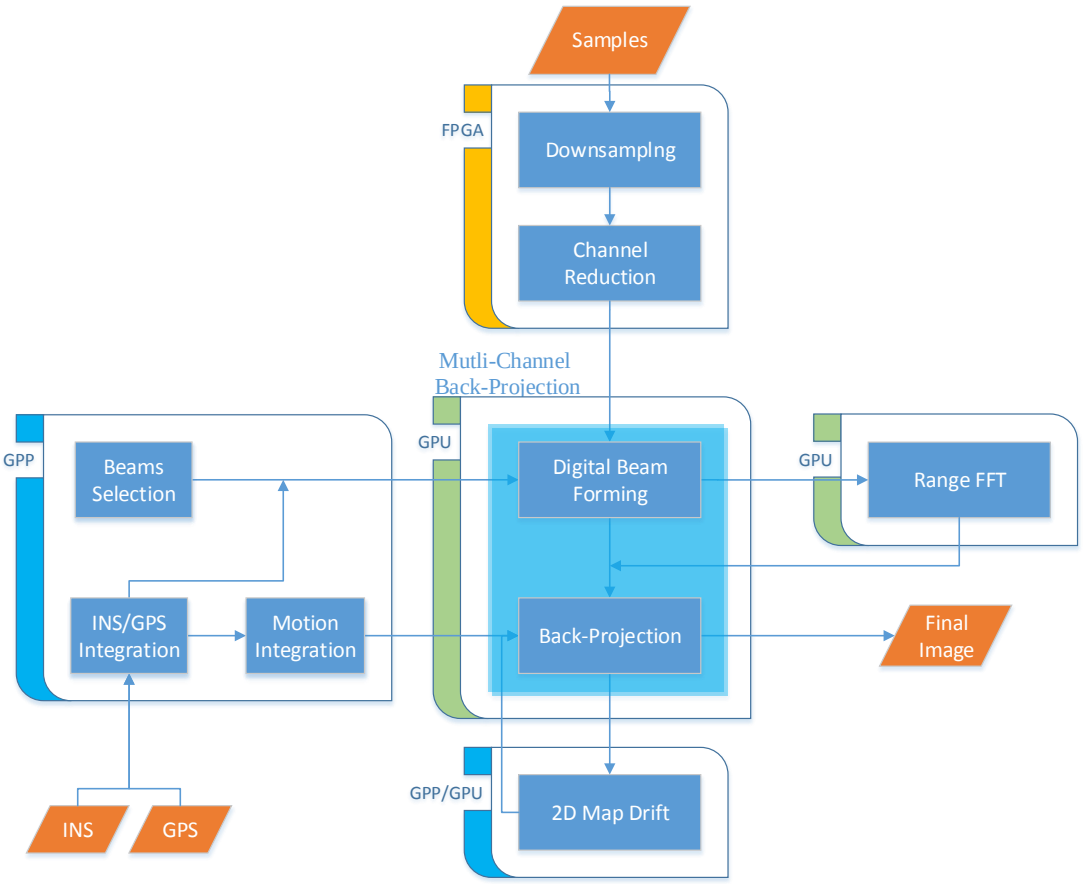


Figure 8.1: Proposed Module Mapping

Decimation and channel reduction are two processes that can be mapped on an FPGA. FPGA deploys very fast I/Os, thus has the ability to support high-speed data. In our system, an FPGA can keep up with the sampling frequency and perform decimation and channel reduction under low latency.

The data parallel nature of Multi-channel back-projection makes a GPU the most appropriate architecture for this module. Moreover, Range FFT can also be mapped on a GPU. GPUs tend to achieve a great speedup over GPP implementations for large FFTs, where the memory transfer can be compensated.

Map Drift deploys more complex logic, thus requires a combination of GPUs and GPPs. Cross correlation can be performed using FFTs. For small FFT sizes (less than 8000 elements), GPP implementations provide lower absolute execution times as they do not have a memory transfer overhead. As a result, for small images, cross correlation is performed on the GPP, for larger ones GPU is the best solution.

Polynomial approximation has a more complex logic and low computational demands. For this reason a GPP is preferred.

Finally, all the other components like pre-processing, voxel grid creation and inertial data manipulation have to be mapped on a GPP.

8.1.2 Custom Hardware Architecture

The major drawbacks of combining different hardware architectures are the overhead spent for communication and data transfer among them and the total power consumption of the whole system. Custom hardware architectures are designed in order to facilitate specific applications and have as their main goal to achieve higher performance power ratio. Figure 8.2 illustrates a possible custom design that is specialized for SAR processing but can be also used for applications which expose data level parallelism and require high-speed IO transactions.

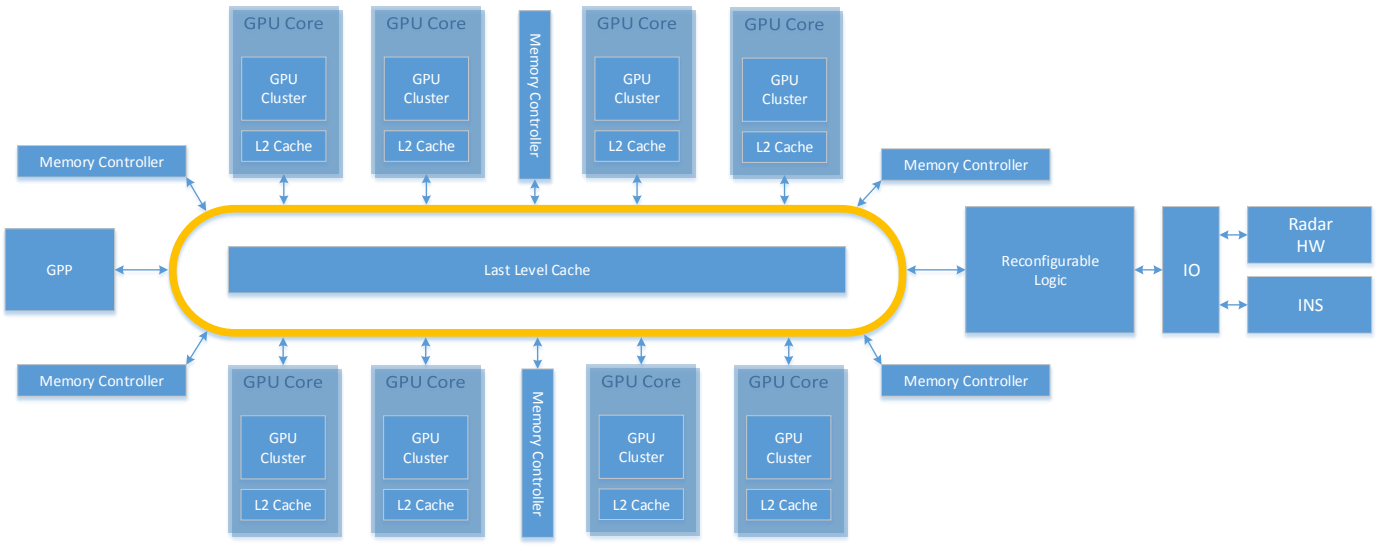


Figure 8.2: Custom Architecture

The architecture consists of a GPP which coordinates the whole system and executes control intensive and task-parallel modules, the GPU cores which are responsible for executing the data-parallel modules and a reconfigurable logic capable of high-speed IO transactions.

All these components are connected through a main ring bus which is responsible for memory coherency. Moreover, multiple memory controllers are available to achieve a wide memory path and thus higher memory bandwidth. High-speed IOs are connected to the reconfigurable logic.

An estimation of the performance of such a system is possible. As GPP a multi-core mobile GPP architecture can be used with low power consumption and mid-range performance. GPU clusters can consist of mobile GPUs which are able nowadays to achieve high raw performance of approximately 350 Gflops each under a power consumption of 2 W each. Having eight GPU clusters produces a total performance of 2.8 Tflops with a power consumption of less than 100 W [2]. Reconfigurable logic should be small and able to facilitate small hardware accelerated modules like decimation module. Such an FPGA core is estimated to have a power consumption of 10 W. Moreover, having for example six 64-bit wide memory controllers leads to a 384-bit path to the main memory, thus for memory frequencies higher than 5 Gbps an aggregate memory bandwidth of 240 GB/s can be achieved.

8.2 Scenarios

Because the ideal architecture presented in Section 8.1.2 is not (yet) available, we present and analyse three distinct possible processing scenarios. The first one targets off-line processing using high-end hardware solutions. The second scenario targets off-line processing using mid-range hardware solutions mainly used on laptops. Finally, the third one targets on-line processing with power and computational constraints, and analyse the possibility of real-time processing.

8.2.1 Off-line High-end Processing

Off-line processing using state-of-the-art hardware can not only reveal potential limitations and bottlenecks, but also help improve future radar designs and enhance radar capabilities.

For the purposes of this scenario we use an NVIDIA Titan GPU and an Intel i7 Extreme CPU (see Chapter 3) which are state-of-the-art GPU and GPP architectures, respectively.

For off-line processing, all the components mapped on an FPGA can be performed on the GPU, as the impact on the total execution time is negligible.

As described in Table 8.1 the requirements for our radar design per second of integration sum up to 150 GFlops for the multi-channel back-projection part and an additional 151 GFlops (1 GFlops for the 2D Map Drift and 150 GFlops for performing back-projection with the corrected track) for every iteration of the map drift autofocus and two images.

These workload may initially seem bearable for high-end architectures, but for larger voxel grids and a larger amount of integration time the requirements increase dramatically.

Our radar hardware targets UAVs, thus supports a maximum grid of 2 km in each dimension. The lowest feasible resolution of 15 cm can leads to images of 178 Mpixel.

[2] Power consumption does not scale linearly. Multiple factors affect the total power.

For this example we assume a voxel grid of 1 km in each dimension, maximum feasible resolution, up-sampling factor of 8 in the fast-time domain [3] and an integration time of 5 s (the average integration time for spot mode).

Moreover, map drift autofocusing can be avoided in images of this size, because, as we mentioned in Chapter 7, for very large images, space requirements for *2D Cross Correlations* are quite large. Nevertheless, track correction and autofocusing can be performed in smaller images with the same effect and this is something that depends on the absolute value of the motion error.

Table 8.2 presents the requirements of the proposed test case along with the execution time of each component. Channel reduction along with decimation can be performed but we are interested in more compute intensive scenarios.

Component	GFlops/s	T[s]	Notes
Decimation	10	0	Not performed
Channel Reduction	0.01	0	Not performed
Range FFT	375.5	1.51	24 channels
Multi-channel Back-projection	36951	45	24 channels
Rest Components [4]	2	1.1	Mostly memory transfers

Table 8.2: Computational requirements for 5 s of integration

Back-projection has a performance of 35 GBP/s and almost 1.2 TFlops/s, which is less compared to the one achieved for smaller grids. This difference stems mainly from thread handling and cache related factors.

Thus, we can conclude that maximum supported image with the highest resolution can be reconstructed in approximately 4 min. By using P GPUs, the execution time can be reduced roughly by a factor P. However, when cluster environments are used, this factor is an optimistic approximation, as more overheads appear due to the underlying physical communication interface.

8.2.2 Off-line Mobile Processing

Having the ability to perform fast off-line processing using hardware accelerators on conventional portable computers gives both the advantage of on-site radar tuning and also makes fast radar processing accessible to more users.

We assume two scenarios, a small grid of 100 m in each dimension and one of 1 km. Both with the maximum feasible resolution, up-sampling factor of 8 in the fast-time domain and an integration time of 5 s which is the average integration time for spot mode.

In the first scenario we also perform a multi-look map drift autofocus using 5 images (Each image corresponds to a second of integration).

[3] Experimental data are already decimated by a factor of 1 during the sample gathering.

[4] We do not count the time required for the parsing of the input data.

Table 8.3 presents the requirements of the proposed test cases along with the execution time of each one. Channel reduction along with decimation are also not performed.

Dimensions	Component	GFlops/s	T[s]	Notes
100m	Decimation	2	0	Not performed
	Channel Reduction	0.001	0	Not performed
	Range FFT	1502	68	24 channels, 4 times
	Multi-channel Back-projection	1478.04	35.2	24 channels, 4 times
	Map Drift	6.9	0.4	5 images, 3 iterations
	Rest Components	2	1.94	Mostly memory transfers
1km	Decimation	10	0	Not performed
	Channel Reduction	0.01	0	Not performed
	Range FFT	375.5	16.92	24 channels
	Multi-channel Back-projection	36951	1295	24 channels
	Rest Components	3	2.45	Mostly memory transfers

Table 8.3: Computational requirements for 5 s of integration

Even using mobile accelerators, with much lower compute capabilities compared to high-end solutions, fast image reconstruction is feasible. In the first scenario, multiple back-projection and map drift iterations were performed in two minutes with a throughput close to 70 GFlops/s. For larger grids more time is required. Glaring example is the second scenario, which required approximately 20 minutes. Although the execution time is much higher than the one required by the high-end solutions, it is still much lower than the initial reference solution, where days were spend in order to finish.

Finally, the larger possible image sizes requires a little more than an hour to be reconstructed.

8.2.3 On-line processing

On-line processing turns out to be the most important processing scenario. There are two main directions on on-line processing: first, to be able to achieve real-time response, and second, to be able to perform processing at time intervals specified by the user.

The second scenario is similar to the off-line processing, in terms that we are still interested in the processing time but no actual deadlines exist. The user defines the area that wants to be reconstructed and the SAR processing is performed using the appropriate radar data. In other words, the processing is performed when the user wants and is not continuous. This can be used for spot image reconstruction, where the user specifies an area of interest, gathers data and performs SAR processing.

On the other hand, in real-time processing, the processing chain is constantly executed reconstructing each time an area in the ground pointed by the radar. Such

an approach can be used for strip mode SAR, where successive images are reconstructed along the illumination path of the antenna. Figure 8.3 depicts the real-time process. Clearly, using GPUs, an image cannot be reconstructed while gathering data for this specific area, thus first the data gathering takes place with an integration time depending on the resolution and size of the final output. Consequently, the latency of this system is equal to the integration time.

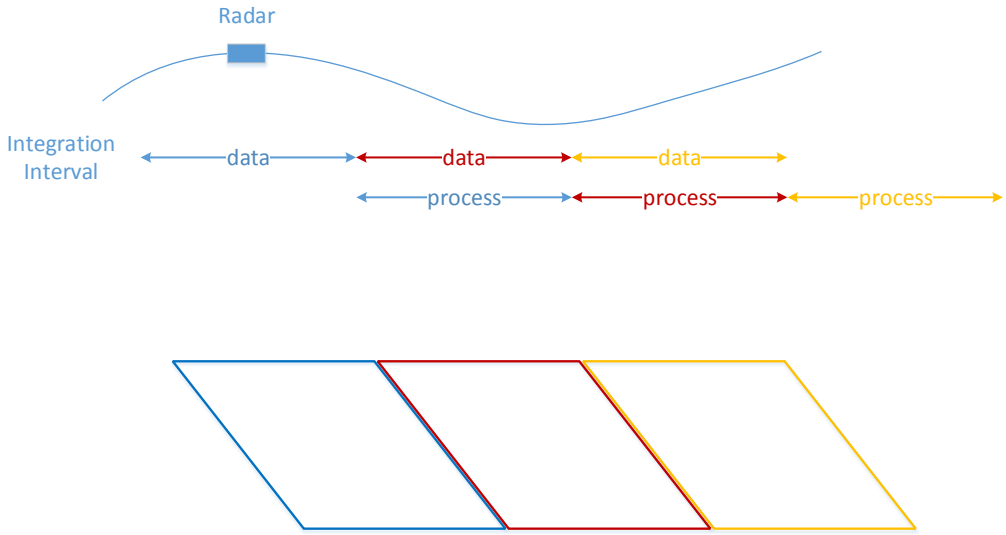


Figure 8.3: Real-time Processing

We performed two experiments for real-time processing, one using *NVIDIA Titan* and one using *NVIDIA Geforce GTX680*. Our goal is to calculate the maximum grid dimensions that we can reconstruct within the integration time interval.

We assume that the airborne platform flies with a speed of 40 m/s and we integrate for an interval of five seconds, targeting the highest possible radar resolution of 15 cm. The product of the integration time with the fly speed results in the length of the reconstructed voxel grid, which in our example is equal to 200 m. Our goal is to specify the possible width of the reconstructed grid under different configurations.

Figure 8.4 illustrates the results of the above experiment. With a mid-range accelerator, we were able to reconstruct a 200 by 100 meters grid while with the high-end GPU a 200 by 300 meter image reconstruction was feasible. Increasing the dimensions is possible by applying decimation on the input samples or reducing the number of integration channels. Decimation led to a small increase of the grid size by 100 meters in both experiments while channel reduction doubled the current

grid size. Grid dimensions can be increased further by reducing the resolution of the radar.

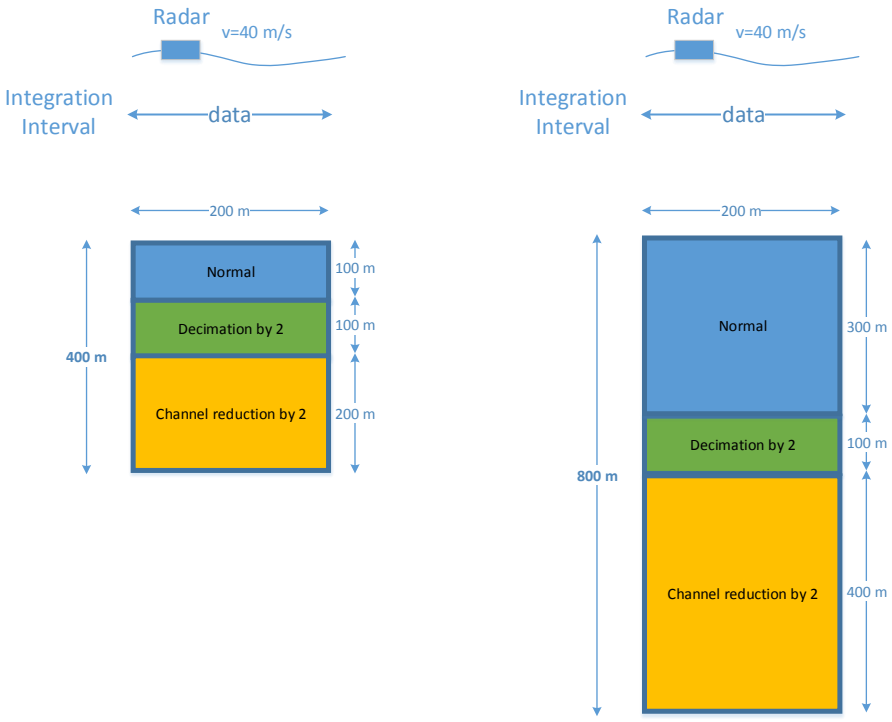


Figure 8.4: Real-time Grid Dimensions (left NVIDIA Geforce GTX680, right NVIDIA Titan)

Overall, real-time on-line processing is feasible, but strongly depends on platform specific attributes like the platform's speed, the radar hardware and the requirements of the system's output.

Chapter 9

Conclusions and Future Work

9.1 Conclusions

Nowadays, multiple different architectures exist, each one offering a performance boost on applications of different nature. Specialized and multi-core computing is clearly mainstream, as most application are ported or redesigned in order to expose any level of parallelism. Thus, new software and programming infrastructures have been introduced to help programmers exploit the benefits multi-core and specialized computing have to offer.

SAR is a radar infrastructure capable of creating high resolution spatial images. Most SAR processing chains operate on the frequency domain for lower computational demands, but the multi-core era has made time-domain processing feasible. Time-domain SAR processing can be quite demanding in terms of computational power for large problem sizes but overcomes formulation problems that frequency-domain algorithms face.

In this thesis, we investigated the possibility of achieving real-time performance on a SAR system, operating on the time-domain. For these purposes the basic processing chain of such a system is analysed, implemented and optimized.

We started by optimizing the most compute intensive component, multi-channel back-projection. Multiple optimization steps took place in both OpenCL and CUDA. We investigated the impact of each optimization step for different problem sizes. The final implementation gave a speedup of approximately *3000* at kernel level and *750* at application level over the sequential implementation. Moreover, we transferred range compression calculation to the GPU, something that lead to a reduction of a factor of 8 for the memory transfers between the host and the GPU. In addition, we exploited potential overlap between computation and communication, hiding 75% of the communication time for NVIDIA Tesla and 100% for the rest GPUs.

The *"embarrassingly"* parallel nature of the multi-channel back-projection led us into the expansion of the implementation for multiple nodes where each node had multiple GPUs. We investigated different parallel solutions and in addition,

we presented a model under different communication strategies. Finally, an almost linear scaling with the number of GPUs across the nodes is achieved.

We analysed different strategies targeting data rate reduction of our processing chain. Our main goal was the reduction of the execution time required by the back-projection module. Two techniques are proposed, decimation and channel reduction. Decimation led to a reduction of the range samples and thus in the computational time required for range compression, something that proves to be useful when calculating small grids. On the other hand, channel reduction reduced the number of the total integration sweeps and eventually the time required by back-projection. Careful utilization of the above techniques is of high importance as they may lead to low quality images and may favour the presence of ambiguities.

The final module was map drift autofocusing. Both GPU and GPP versions were implemented, each one favouring different problem sizes. Track correction was performed only in the line-of-sight, but the main infrastructure for supporting more complex correcting algorithms already exists.

Finally, a module mapping along with a custom hardware architecture are presented. Each processing module is assigned to the most appropriate architecture to guarantee efficient execution. Moreover, three distinct possible processing scenarios are presented, with the most interesting being the last one, which emphasizes achieving real-time response. Real-time response can be eventually achieved depending on the underlying hardware architecture and the requirements of the system (speed of the aircraft, size of supporting grid, resolution etc.).

9.2 Future Work

This thesis presents an initial attempt into evaluating and optimising a SAR processing chain, thus multiple potential directions for future work exist. We emphasize three such directions: the SAR system design research, the application research which focuses on back-projection itself, and finally the generic research.

The SAR system design research path mainly focuses on expanding the proposed processing chain, making it capable of supporting additional features like GMTI. In addition, more complex autofocusing techniques can be applied and investigated, which may lead to more accurate results. In general, having a fast and generic SAR processing framework gives radar engineers the ability to experiment, evaluate and introduce more complex features.

In this thesis, we restrict our experiments to NVIDIA GPUs, but GPUs from other vendors can be used and evaluated. An interesting twist, could be the utilisation of low-power mobile GPPs and GPUs. This may initially require custom-made designs, containing clusters of GPPs and GPUs in order to support the computational demands required by a SAR system. But such a design could lead to a low-power system able to be mounted on power constrained platforms and eventually on the aircraft itself.

The application research path mainly focuses into back-projection itself. First,

more optimisation steps could be tested and evaluated, with the main one being data-dependent optimisations. An appropriate sorting of the raw input data could simplify the computations and give us the ability to utilise the fast GPU memory, in turn leading to a significant reduction of the memory traffic inside the GPU. Second, back-projection is not used only in SAR processing, it has many applications in fields like medicine, astronomy and non-destructive testing. Thus, a fast, optimised and parametrizable generic back-projection implementation could be beneficial not just for radar processing.

Finally, the generic research path focuses on concepts concerning parallel programming and parallel programming frameworks. We used and compared both OpenCL and CUDA using theoretical models and performance measuring tools. CUDA is well-supported with a large amount of in-house performance measuring tools, while OpenCL lack such tools. Having the appropriate tools could help us elaborate more into the performance differences of these two parallel programming frameworks. In order to do this more applications with both simpler and complex structure should be investigated. Moreover, it would be interesting to have an insight into the impact of different optimisation strategies while accounting both problem and input scalability. Ultimately, an appropriate model could be developed capable of analysing, under pre-specified assumptions, potential optimization strategies and estimate their impact to performance.

Bibliography

[1] Gene M. Amdahl. Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. In *Proceedings of the April 18-20, 1967, Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[2] Folding at Home. FAQ: FLOPS.

[3] T.M. Benson, D.P. Campbell, and D.A. Cook. Gigapixel spotlight synthetic aperture radar backprojection using clusters of GPUs and CUDA. In *Radar Conference (RADAR), 2012 IEEE*, pages 0853–0858, May 2012.

[4] Lawrence Livermore National Laboratory Blaise Barney. Message Passing Interface (MPI) Tutorial, May 2014.

[5] Giorgio C. Buttazzo. *Hard Real-time Computing Systems: Predictable Scheduling Algorithms And Applications (Real-Time Systems Series)*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2004.

[6] T.M. Calloway and G.W. Donohoe. Subaperture autofocus for synthetic aperture radar. *Aerospace and Electronic Systems, IEEE Transactions on*, 30(2):617–621, Apr 1994.

[7] Terry M. Calloway, Charles V. Jakowatz, Jr., Paul A. Thompson, and Paul H. Eichel. Comparison of synthetic-aperture radar autofocus techniques: phase gradient versus subaperture. volume 1566, pages 353–364, 1991.

[8] A. Capozzoli, C. Curcio, and A. Liseno. Fast gpu-based interpolation for SAR backprojection. In *Progress In Electromagnetics Research, vol133*, pages 259–283, 2013.

[9] A. Capozzoli, C. Curcio, A. Liseno, and P.V. Testa. Nufft-based sar backprojection on multiple gpus. In *Advances in Radar and Remote Sensing (TyWRRS), 2012 Tyrrhenian Workshop on*, pages 62–68, Sept 2012.

[10] W. Chapman, S. Ranka, S. Sahni, M. Schmalz, U. Majumder, L. Moore, and B. Elton. Parallel processing techniques for the processing of synthetic aperture radar data on GPUs. In *Signal Processing and Information Technology (ISSPIT), 2011 IEEE International Symposium on*, pages 573–580, Dec 2011.

[11] Ben Cordes and Miriam Leeser. Parallel Backprojection: A Case Study in High-performance Reconfigurable Computing. *EURASIP J. Embedded Syst.*, 2009:1:1–1:14, Jan. 2009.

[12] Real World Technologies David Kanter. Intels Haswell CPU Microarchitecture, November 2012.

[13] Real World Technologies David Kanter. Knights Landing Details, January 2014.

[14] A.P. Delazari Binotto, E. Pignaton de Freitas, C.E. Pereira, A. Stork, and T. Larsson. Real-time task reconfiguration support applied to an UAV-based surveillance system. In *Computer Science and Information Technology, 2008. IMCSIT 2008. International Multiconference on*, pages 581–588, Oct 2008.

[15] A. Fasih and T. Hartley. GPU-accelerated synthetic aperture radar backprojection in CUDA. In *Radar Conference, 2010 IEEE*, pages 1408–1413, May 2010.

[16] FFTW. Fftw library.

[17] MPI Forum. MPI Documents, May 2014.

[18] Liang Fulai, Qu Xiaojiang, Li Yanghuan, Song Qian, and Zhang Hanhua. GPU-accelerated SAR backprojection in JACKET for MATLAB. In *Synthetic Aperture Radar (APSAR), 2011 3rd International Asia-Pacific Conference on*, pages 1–4, Sept 2011.

[19] Rahul Garg. Floating point peak performance of Kaveri and other recent AMD and Intel chips, January 2014.

[20] N. Gebert, G. Krieger, and A. Moreira. Digital beamforming on receive: Techniques and optimization strategies for high-resolution wide-swath sar imaging. *Aerospace and Electronic Systems, IEEE Transactions on*, 45(2):564–592, April 2009.

[21] Intel ®George Chrysos. Intel©Xeon Phi ™Coprocessor - the Architecture, September 2012.

[22] T.D.R. Hartley, A.R. Fasih, C.A. Berdanier, F. Ozguner, and U.V. Catalyurek. Investigating the use of gpu-accelerated nodes for sar image formation. In *Cluster Computing and Workshops, 2009. CLUSTER '09. IEEE International Conference on*, pages 1–8, Aug 2009.

[23] M.D. Hill and M.R. Marty. Amdahl's Law in the Multicore Era. *Computer*, 41(7):33–38, July 2008.

[24] V Jithesh and K Poulose Jacob. GPU Based Performance Acceleration of Radar Imaging Algorithms.

[25] David B. Kirk and Wen-mei W. Hwu. *Programming Massively Parallel Processors: A Hands-on Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[26] Kazuhiko Komatsu, Katsuto Sato, Yusuke Arai, Kentaro Koyama, Hiroyuki Takizawa, and Hiroaki Kobayashi. Evaluating Performance and Portability of OpenCL Programs. In *The Fifth International Workshop on Automatic Performance Tuning*, June 2010.

[27] F. Kraja, A. Murarasu, G. Acher, and A. Bode. Performance evaluation of SAR image reconstruction on CPUs and GPUs. In *Aerospace Conference, 2012 IEEE*, pages 1–16, March 2012.

[28] Yung Chong Lee, Voon Chet Koo, and Yee Kit Chan. Fpga-based pre-processing unit for real-time synthetic aperture radar (sar) imaging. In *Progress In Electromagnetics Research Symposium*, pages 1087 – 1091, March 2012.

[29] Bin Liu, Kaizhi Wang, Xingzhao Liu, and Wenxian Yu. An Efficient SAR Processor Based on GPU via CUDA. In *Image and Signal Processing, 2009. CISP '09. 2nd International Congress on*, pages 1–5, Oct 2009.

[30] Bin Liu, Kaizhi Wang, Xingzhao Liu, and Wenxian Yu. An Efficient Signal Processor of Synthetic Aperture Radar Based on GPU. In *Synthetic Aperture Radar (EUSAR), 2010 8th European Conference on*, pages 1–4, June 2010.

[31] Jongsoo Park, P.T.P. Tang, M. Smelyanskiy, Daehyun Kim, and T. Benson. Efficient backprojection-based synthetic aperture radar computation with many-core processors. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, Nov 2012.

[32] Song Jun Park, D. Shires, D. Richie, and J. Ross. Enhanced SAR Imaging Algorithm Development for Streaming Processors. In *High Performance Computing Modernization Program Users Group Conference (HPCMP-UGC), 2010 DoD*, pages 396–400, June 2010.

[33] B. Porat. *A course in digital signal processing*. John Wiley, 1997.
[34] X. Qiu, C. Ding, and D. Hu. *Bistatic SAR Data Processing Algorithms*. Wiley, 2013.
[35] M. Raskovic, A.L. Varbanescu, W. Vlothuizen, M. Ditzel, and H. Sips. OCL-BodyScan: A Case Study for Application-centric Programming of Many-Core Processors. In *Parallel Processing (ICPP), 2011 International Conference on*, pages 542–551, Sept 2011.
[36] M.A. Richards, J.A. Scheer, J. Scheer, and W.A. Holm. *Principles of Modern Radar: Basic Principles*. Number v. 1 in Principles of Modern Radar. SciTech Publishing, Incorporated, 2010.
[37] Sean Rul, Hans Vandierendonck, Joris D'Haene, and Koen De Bosschere. An experimental study on performance portability of OpenCL kernels. In *Application Accelerators in High Performance Computing, 2010 Symposium, Papers*, page 3, 2010.
[38] H.J. Sips. *Aspects of Computational Science a Textbook on High Performance Computing*, chapter Programming Languages for High Performance Computing, pages 125–194. NCF, Den Haag, 1995. Editor: A. van der Steen.
[39] Xian-He Sun and Yong Chen. Reevaluating Amdahl's Law in the Multicore Era. *J. Parallel Distrib. Comput.*, 70(2):183–188, Feb. 2010.
[40] Zhi-Jian Sun and Xue-Mei Liu. The realization of sar real-time signal processor by fpga. In *Computer Science and Software Engineering, 2008 International Conference on*, volume 4, pages 79–82, Dec 2008.
[41] AMD ®. AMD® clMath.
[42] Intel ®. Intel® Core™ i7-2670QM Specifications.
[43] Intel ®. Intel® Core™ i7-3612QM Specifications.
[44] Intel ®. Intel® VTune™ Amplifier XE 2013.
[45] Intel ®. Intel®l Core i7-4960X Extreme Edition Specifications.
[46] Intel ®. *Desktop 4th Generation Intel Core Processor Family: Datasheet, Vol. 1*. Intel ®, March 2014.
[47] NVIDIA ®. cuFFT CUDA Toolkit.
[48] NVIDIA ®. GeForce™ GT540M Specifications.
[49] NVIDIA ®. GeForce™ GTX TITAN Specifications.
[50] NVIDIA ®. *NVIDIAs Next Generation CUDA ™Compute Architecture: Kepler TM GK110*. NVIDIA ®.
[51] NVIDIA ®. NVIDIA® Visual Profiler.
[52] NVIDIA ®. Tesla™ K20 Specifications.
[53] Texas Instruments ®. *TMS320C66x DSP CPU and Instruction Set Reference Guide*. Texas Instruments ®, November 2010.
[54] Texas Instruments ®. *TMS320C6678 Multicore Fixed and Floating-Point Digital Signal Processor*. Texas Instruments ®, revision e edition, March 2014.
[55] XILINX ®. FPGA.
[56] NVIDIA ®CUDA ™. CUDA ™, May 2014.
[57] Khronos Group ™. OpenCL, May 2014.
[58] OpenMP ™. OpenMP™, May 2014.
[59] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Commun. ACM*, 33(8):103–111, Aug. 1990.
[60] M.W. van der Graaf, M.P.G. Otten, A.G. Huizing, R.G. Tan, M.C. Cuenca, and M.G.A. Ruizenaar. Amber: An x-band fmcw digital beam forming synthetic aperture radar for a tactical uav. In *Phased Array Systems Technology, 2013 IEEE International Symposium on*, pages 165–170, Oct 2013.
[61] Jarno van der Sanden. Evaluating the performance and portability of opencl. MSc thesis, Eindhoven University of Technology, August 2011.

[62] M. P. G. Otten & W. L. van Rossum & R. Tan & W. J. Vlothuizen & J. J. M. de Wit. Multi-channel processing for digital beam froming SAR. In *Proceedings of the 21st European Signal Processing Conference (EUSIPCO 2013)*, Marrakech, Morocco, 2013.

[63] W.J. Vlothuizen and M. Ditzel. Real-time brute force SAR processing. In *Radar Conference, 2009 IEEE*, pages 1–4, May 2009.

[64] D.E. Wahl, P.H. Eichel, D.C. Ghiglia, and Jr. Jakowatz, C.V. Phase gradient autofocus-a robust tool for high resolution SAR phase correction. *Aerospace and Electronic Systems, IEEE Transactions on*, 30(3):827–835, Jul 1994.

[65] W.Q. Wang. *Multi-Antenna Synthetic Aperture Radar*. Taylor & Francis, 2013.

[66] Wikipedia. Bulk Synchronous Parallel, May 2014.

[67] Wikipedia. OpenMP, May 2014.

[68] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4):65–76, Apr. 2009.

[69] N. Wilt. *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Pearson Education, 2013.

[70] Christian Wolff. Radar Tutorial, May 2014.

[71] Yewei Wu, Jun Chen, and Hongqun Zhang. A real-time SAR imaging system based on CPUGPU heterogeneous platform. In *Signal Processing (ICSP), 2012 IEEE 11th International Conference on*, volume 1, pages 461–464, Oct 2012.

[72] Zhang Xin, Zhang Xiaoling, Shi Jun, and Liu Zhe. GPU-based parallel back projection algorithm for the translational variant BiSAR imaging. In *Geoscience and Remote Sensing Symposium (IGARSS), 2011 IEEE International*, pages 2841–2844, July 2011.