SOFTWARE-DEFINED PULSE-DOPPLER RADAR SIGNAL PROCESSING ON GRAPHICS PROCESSORS

by

Christian Jacobus Venter

Submitted in partial fulfilment of the requirements for the degree

Master of Engineering (Computer Engineering)

in the

Department of Electrical, Electronic and Computer Engineering
Faculty of Engineering, Built Environment and Information Technology
UNIVERSITY OF PRETORIA

May 2014

SUMMARY

SOFTWARE-DEFINED PULSE-DOPPLER RADAR SIGNAL PROCESSING ON GRAPHICS PROCESSORS

by

Christian Jacobus Venter

Supervisor(s):	Mr. H. Grobler
Department:	Electrical, Electronic and Computer Engineering
University:	University of Pretoria
Degree:	Master of Engineering (Computer Engineering)
Keywords:	Software-defined, pulse-Doppler, radar, graphics processing unit, digital pulse compression, corner turning, Doppler filtering, envelope function, constant false-alarm rate, high-performance computing

Modern pulse-Doppler radars use digital receivers with high speed ADCs and sophisticated radar signal processors that necessitate high data rates, computationally intensive processing, and strict latency requirements. Data-independent processing is performed as the first stage and requires the highest data and computational rates of between 1 Gigaops to 1 Teraops, traditionally reserved for specialized circuits that typically employ restrictive fixed-point arithmetic. The first stage generally requires FIR filters, correlation, Fourier transforms, and matrix-vector algebra on multi-dimensional data, which provides a range of demanding and interesting computational challenges, and that present ample opportunities for parallel processing. Modern many-core GPUs provide general-purpose computation on the GPU (GPGPU) for high-performance computing applications through fully programmable pipelines, high memory bandwidths of up to hundreds of Gigabytes per second and high floating-point computational performance of up to several Teraflops on a single chip. The massively-parallel GPU architecture is well-suited for intrinsically parallel applications that require high dynamic range, such as radar signal processing. However, numerous factors have to be considered in order to realize the massive performance potential through a conventionally unfamiliar stream-programming paradigm. Explicit control is also granted over a deep memory hierarchy and parallelism at various granularities within an optimization space that is considered non-linear in many respects.

The aim of this research is to address and characterize the challenges and intricacies of using modern GPUs with GPGPU capabilities for the computationally demanding software-defined pulse-Doppler radar signal processing application. A single receiver-element, coherent pulse-Doppler system with a two-dimensional data storage model was assumed, due to widespread use and the interesting challenges and opportunities that it provides for parallel implementation on the GPU architecture. The NVIDIA Tesla C1060 GPU and CUDA were selected as a suitable GPGPU platform for the implementation using single-precision floating-point arithmetic. A set of microbenchmarks was first developed to isolate and highlight fundamental traits and relevant features of the GPU architecture, in order to determine their impact in the radar application context. The common digital pulse compression (DPC), corner turning (CT), Doppler filtering (DF), envelope (ENV) and constant false-alarm rate (CFAR) processing functions were then implemented and optimized for the GPU architecture. Multiple algorithmic variants were implemented, where appropriate, to evaluate the efficiency of different algorithmic structures on the GPU architecture. These functions were then integrated to form a radar signal processing chain, which allowed for further holistic optimization under realistic conditions. An experimental framework and simple analytical framework was developed and utilized for analyzing low-level kernel performance and high-level system performance for individual functions and the processing chain.

The microbenchmark results highlighted the severity of uncoalesced device memory access as well as the importance of high arithmetic intensity to achieve high computational throughput, and an asymmetry in performance for primitive math operations. Further, the microbenchmark results showed that memory transfer performance for small buffers or effectively small radar bursts is fundamentally poor, but also that memory transfer can be efficiently overlapped with computation, reducing the impact of slow transfers in general. For the DPC and DF functions, the FFT-based variants using the CUFFT library proved optimal. For the CT function, the use of shared memory is vital to achieve fully coalesced transfers, and the lesser-known, but potentially highly detrimental, partition camping effect needs to be addressed. For the CFAR function, the segmentation into separate processing stages for rows and columns proved the most vital overall optimization. The ENV function along with several simple GPU helper-kernels with low arithmetic intensity such as padding, scaling, and the window function were found to be bandwidth-limited, as expected, and hence performs comparably to a pure copy kernel. Based on the findings, pulse-Doppler radar signal processing on GPUs is highly feasible for medium to large burst sizes, provided that the main performance contributors and detractors for the target GPU architecture is well understood and adhered to.

OPSOMMING

SAGTEWARE-GEDEFINIEERDE PULS-DOPPLER RADAR SEINVERWERKING OP GRAFIKA VERWERKINGSEENHEDE

deur

Christian Jacobus Venter

Studieleier(s):	Mnr. H. Grobler
Departement:	Elektriese, Elektroniese en Rekenaar-Ingenieurswese
Universiteit:	Universiteit van Pretoria
Graad:	Magister in Ingenieurswese (Rekenaaringenieurswese)
Sleutelwoorde:	Sagteware-gedefinieerd, puls-Doppler, radar, grafika verwerkingseenheid, transponeer, digitale pulskompressie, Doppler filter, omhulling funksie, konstante vals-alarm tempo, hoë werkverrigting verwerking

Moderne puls-Doppler radars gebruik digitale ontvangers met hoëspoed analoog-na-syfer omsetters en gesofistikeerde radar seinverwerkers wat hoë datatempo's, verwerkings-intensiewe rekenkunde en streng vertraging teikens noodsaak. Data-onafhanklike verwerking word verrig tydens die eerste fase en benodig die hoogste data-en verwerkingstempo's van 1 Gigaops tot 1 Teraops, wat tradisioneel gereserveer word vir gespesialiseerde stroombane wat tipies beperkte vastepunt rekenkunde aanwend. Die eerste fase benodig tipies FIR filters, korrelasie, Fourier transforms, en matriks-vektor algebra op multidimensionele data, wat 'n reeks veeleisende en interessante rekenkundige uitdagings bied, asook genoegsame geleenthede vir parallel verwerking. Moderne veel-kern GPUs verskaf algemene verwerking op die GPU (GPGPU) vir hoë werkverrigting verwerking toepassings deur volledig programmeerbare pyplyne, hoë geheue bandwydte van honderde Gigagrepe per sekonde en hoë wisselpunt werkverrigting van vele Teraflops op 'n enkele vlokkie. Die massief-parallel GPU argitektuur is gepas vir toepassings wat intrinsiek parallel is en hoë dinamiese bereik vereis, soos radar seinverwerking. Verskeie faktore moet egter in ag geneem word om die reuse werkverrigtings potensiaal te realiseer deur middel van 'n stroom-programmering paradigma wat ongewoon is vir algemene verwerkers. Die programmeerder ontvang eksplisiete beheer oor 'n diep geheuehiërargie en parallelisme op verskeie vlakke, binne 'n optimeringsruimte wat in baie opsigte as nie-lineêr beskou word.

Die oogmerk van hierdie navorsing is om die uitdagings en kompleksiteit van die verwerkings-intensiewe, sagteware-gedefinieerde puls-Doppler radar seinverwerking toepassing op GPUs met GPGPU vermoëns, te karakteriseer en te oorkom. ’n Enkel-element, koherente puls-Doppler stelsel met ’n tweedimensionele databergingsmodel is aanvaar, gaande wydverspreide gebruik en interessante uitdagings en geleenthede wat gebied word vir parallel implementering op die GPU argitektuur. Die NVIDIA Tesla C1060 GPU en CUDA is gekies as ’n geskikte GPGPU platform vir die implementering, met die gebruik van wisselpunt rekenkunde. ’n Stel mikromaatstawwe is aanvanklik ontwikkel om die fundamentele eienskappe en kenmerke van die GPU argitektuur te isoleer en uit te lig, om die ooreenkomstige impak daarvan in die konteks van die radar toepassing vas te stel. Die digitale pulskompressie (DPC), transponeer funksie (CT), Doppler filter (DF), omhulling funksie (ENV) en konstante vals-alarm tempo (CFAR) verwerkings funksies is volgende geimplementeer en geoptimeer vir die GPU argitektuur. Verskeie algoritmiese variante is geimplementeer, waar van toepassing, sodat die effektiwiteit van verskillende algoritmiese strukture op die GPU argitektuur vasgestel kon word. Hierdie funksies is volgende geïntegreer om ’n radar seinverwerkingsketting te vorm, wat toelaat vir holistiese optimering onder realistiese kondisies. ’n Eksperimentele raamwerk en eenvoudige analitiese raamwerk is ontwikkel en aangewend om lae-vlak *kernel* werkverrigting en hoë-vlak stelsel werkverrigting vir individuele fuksies en vir die seinverwerkingsketting te analiseer.

Die mikromaatstaf resultate het die erns van nie-saamvloeiende geheue toegang, asook die belangrikheid van hoë rekenkundige intensiteit om hoë verwerkingsdeurset te bereik, en asimmetriese werkverrigting vir primitiewe wiskundige verwerkings, uitgelig. Verder dui die resultate van die mikromaatstaf aan dat die geheueoordragstempo vir klein buffers, of effektief klein sarsies, fundamenteel swak is, maar ook dat geheueoordrag oorvleuel kan word met verwerking, wat die impak van stadige oordrag verminder. Vir die DPC en DF funksies, was die FFT-gebaseerde variante optimaal. Vir die CT funksie, was die gebruik van gedeelde geheue onmisbaar om volledig saamvloeiende oordrag te verkry, asook die minder bekende, maar potensieel hoogs nadelige, *partition camping* effek aan te spreek. Vir die CFAR funksie was die verdeling in aparte verwerkingstappe vir rye en kolomme algeheel die belangrikste optimering. Die bevinding is, volgens verwagting, dat die ENV funksie tesame met verskeie eenvoudige GPU helper-*kernels* met lae rekenkundige intensiteit, soos stoffering, skalering, en die venster funksie, bandwydte-beperk is, soos verwag, en derhalwe vergelykbaar met ’n suiwer kopie *kernel* presteer. Luidens die bevindinge, is puls-Doppler radar seinverwerking op GPUs hoogs vatbaar vir medium tot groot sarsie groottes, mits die hoof prestasie faktore vir die teiken GPU argitektuur deeglik verstaan word en effektief aangespreek word.

ACKNOWLEDGEMENTS

The author would like to thank the following persons and institutions for their support:

- My supervisor, Mr. Hans Grobler, for all the guidance and advice that he has given me throughout the course of my research.
- The South African Department of Science and Technology (DST) for funding of my research.
- The King Abdulaziz City for Science and Technology (KACST) for their support of a parallel architecture investigation.
- My colleagues at the Council for Scientific and Industrial Research (CSIR) for your personal support, insights on Radar, and for the use of the GPU computing resources.
- My friends and family for your patience and encouragement, with special thanks to Elize for proofreading my dissertation.
- My girlfriend, San-Marié, for your love and support.

LIST OF ABBREVIATIONS

ADC	Analog-to-digital converter
AMD	Advanced micro devices
ALU	Arithmetic logic unit
API	Application programming interface
ASIC	Application-specific integrated circuit
BE	Broadband engine
BW	Bandwidth
CA-CFAR	Cell-averaging CFAR
CC	Compute capability
CFAR	Constant false-alarm rate
CPU	Central processing unit
CUDA	Compute unified device architecture
CUT	Cell under test
CMEM	Constant memory
CPI	Coherent processing interval
CT	Corner turning
CTM	Close to the metal
CW	Continuous wave
DF	Doppler filtering
DFT	Discrete Fourier transform
DMA	Direct memory access
DPC	Digital pulse compression
DRAM	Dynamic random access memory
DSP	Digital signal processor
DtoH	Device to host
ECC	Error-correcting code
EMI	Electromagnetic interference
ENV	Envelope function
FFT	Fast Fourier transform
FIR	Finite impulse response

FLOPs	Floating-point operations
FLOPS	Floating-point operations per second
FMA	Fused multiply-add
FPGA	Field-programmable gate array
GB	Gigabyte (10^9)
GBF	GPU benchmarking framework
GiB	Gibibyte (2^{30})
GLSL	OpenGL shading language
GMEM	Global memory
GPU	Graphics processing unit
GPGPU	General-purpose computation on GPUs
HLSL	High-level shading language
HPEC	High-performance embedded computing
HPC	High-performance computing
HtoD	Host to device
I	In-phase
IF	Intermediate frequency
IFFT	Inverse fast Fourier transform
ILP	Instruction-level parallelism
ISA	Instruction set architecture
KB	Kilobyte (10^3)
KiB	Kibibyte (2^{10})
LO	Local oscillator
MAD	Multiply-add
MB	Megabyte (10^6)
MiB	Mebibyte (2^{20})
MKL	Math kernel library
MS	MegaSamples
MT	Multithreaded
MTI	Moving target indication
OpenCL	Open Computing Language
OPS	Operations per second
PC	Partition camping

PCA	Polymorphic computing architecture
PCIe	Peripheral component interconnect express
PCL	Passive coherent locator
PRF	Pulse repetition frequency
PRI	Pulse repetition interval
PS3	PlayStation 3
PTX	Parallel thread execution
Q	Quadrature
RCS	Radar cross-section
RF	Radio frequency
RISC	Reduced instruction set computing
RNR	Random noise radar
ROP	Raster operation processor
RSP	Radar signal processor
SAR	Synthetic aperture radar
SFU	Special function unit
SIMD	Single instruction, multiple data
SIMT	Single instruction, multiple thread
SISD	Single instruction, single data
SIR	Signal-to-interference ratio
SM	Streaming multiprocessor
SMEM	Shared memory
SP	Scalar processor
SPA	Streaming processor array
SAT	Summed-area table
SWaP	Size weight and power
TLP	Thread-level parallelism
TMEM	Texture memory
TPC	Texture processor cluster

TABLE OF CONTENTS

CHAPTER 1

INTRODUCTION

1.1 PROBLEM STATEMENT

A problem statement is first formulated, by describing the context of the problem and identifying the research gap.

1.1.1 Context of the problem

Modern pulse-Doppler radars use digital receivers with high speed ADCs and sophisticated radar signal processors that necessitates high data rates and computationally intensive processing. Due to demanding real-time mission-level requirements, the radar signal processor is typically required to provide high throughput, with some applications also requiring low latency. The data-independent processing stage is the first processing stage in the radar signal processing chain with the highest data and computational rates, requiring between 1 Gigaops to 1 Teraops, which can be appropriately considered a high-performance computing application. The digital pulse compression (DPC), corner turning (CT), Doppler filtering (DF), envelope (ENV), and constant false-alarm rate (CFAR) functions are performed within the first processing stage and are common in a variety of pulse-Doppler radar applications. These functions generally require FIR filters, correlation, Fourier transforms, and matrix-vector algebra performed on multi-dimensional data, which provides a range of demanding and interesting computational challenges, and that present ample opportunities for parallel processing. The data-independent processing stage is traditionally reserved for specialized circuits and processors, such as ASICs, FPGAs and more recently DSPs, that often employ fixed-point arithmetic.

Fixed-point arithmetic is used traditionally as a result of the high computational rates, as it is faster, simpler to implement and requires less die space. The traditional approach presents several challenges with regard to the complexity, cost and skill sets required to design and develop ASICs and HDL code for FPGAs. In addition, fixed-point arithmetic has limited dynamic range and is prone to overflow and underflow errors. Due to Moore's law, increasingly general architectures can be used to perform software-defined radar signal processing closer to the radar front-end, which can alleviate many of the traditional challenges and limitations.

In recent years GPUs have been generalized to also support non-graphics processing with the advent of general-purpose computation on GPUs (GPGPU), which is used extensively for scientific and other high-performance computing applications. Modern graphics processing units (GPUs) provide fully programmable pipelines, high memory bandwidths of up to hundreds of Gigabytes per second, and high computational performance of up to several Teraflops on a single chip. GPUs have a massively-parallel many-core architecture that is well-suited for applications similar to graphics processing, with a high degree of intrinsic parallelism that can be exploited. Native floating-point support is rooted in the graphics processing heritage, but provides significant advantages for radar applications with regard to dynamic range, numerical accuracy and ease of implementation. High-level parallel computing frameworks such as NVIDIA CUDA and OpenCL have also become available which can be used to ease GPGPU implementation.

However, there are considerable differences between the GPU architecture and programming model compared to conventional general-purpose architectures such as the CPU, which introduces additional complexities that have to be considered to realize the massive performance potential of GPUs. CPUs use a single instruction, single data (SISD) and to some extent a single instruction, multiple data (SIMD) vector-programming paradigm. Conversely, GPUs use a stream-programming paradigm that is closely related to SIMD, called single instruction, multiple thread (SIMT) for NVIDIA GPUs. The familiar CPU architecture features tens of sophisticated cores, that run hundreds of heavyweight threads, and favor task-level and instruction-level parallelism. Conversely, the GPU architecture features hundreds of simple processing cores, that run thousands of lightweight threads, and favors data-parallel processing. In addition to these fundamental architectural differences, GPUs also explicitly expose a deep memory hierarchy, control over fine-grained and coarse-grained parallelism, and data transfer to and from the GPU, to the programmer. Appropriate partitioning and decomposition of the problem domain to the GPU memory and threading model is therefore critical, and algorithms may need to be adapted to suit the massively-parallel architecture.

1.1.2 Research Gap

Many implementations of individual signal processing algorithms and primitives on GPUs are available. Nonetheless, a research gap was identified, where the implementation of an entire radar signal processing chain on the GPU architecture is seldom attempted. Many studies also exclude data transfer times over the PCI express bus to and from the GPU, where input and output data for a processing chain ultimately resides on the host, and realistically needs to be considered. Furthermore, high throughput and low latency is required for certain the radar applications, whereas only high throughput is primarily required for many scientific and other high-performance computing applications that utilize GPUs. Therefore, latency is also an important factor to consider, where it is often a secondary concern in many other applications.

1.2 RESEARCH QUESTIONS

Several research questions are posed that are relevant to the problem:

1. How well do the individual pulse-Doppler radar signal processing algorithms suit implementation on the GPU architecture?
2. How well does the complete pulse-Doppler radar signal processing chain suit implementation on the GPU architecture?
3. How do we evaluate the performance of the radar signal processing chain?
4. Are there any features of the GPU architecture that constrain the efficiency of pulse-Doppler radar signal processing?

1.3 HYPOTHESIS AND APPROACH

The hypothesis presented in this research states that software-defined pulse-Doppler radar signal processing may be implemented efficiently on modern graphics processors with GPGPU capabilities. The level of parallelism exposed by the computationally demanding radar signal processing problem is considered to be intrinsically high, and is expected to be sufficient to enable efficient implementation on the highly-parallel GPU architecture, which has the potential for massive computational performance. However, the actual performance that is achieved depends on a variety of factors, including the algorithmic structure and characteristics. The pulse-Doppler radar signal processing algorithms perform a wide range of computational and data manipulation tasks, and are expected to have varying algorithmic properties. Consequently, each algorithm may require dedicated analysis and optimization to implement efficiently on the GPU architecture, if at all suitable for efficient implementation. The overall performance of the radar signal processing chain is expected to depend on the efficiency and level of optimization that can be achieved for the constituent building blocks. The approach to the research problem is described in the rest of this section.

An extensive literature survey was conducted on pulse-Doppler radar and graphics processing units (GPUs), in order to identify typical pulse-Doppler radar signal processing algorithms and appropriate techniques, software environments, and hardware for GPU implementation. A set of metrics were identified to objectively characterize the performance of algorithms implemented on the GPU architecture analytically and experimentally. A benchmarking process was developed for reliable experimental validation of GPU implementations. Initial experimentation with the core features and high priority optimization techniques was performed on the target GPU architecture, in order to isolate and characterize fundamental behaviors and effects up front. Subsequently, these behaviors and effects can be more easily identified, when present, in the more complex primary implementations.

A systematic approach was subsequently followed to analyze, implement, and progressively optimize individual radar signal processing algorithms. Opportunities for parallel execution were identified and exploited in conjunction with identifying and refining the relevant GPU implementation techniques for each algorithm. A software-defined pulse-Doppler radar signal processing chain was then developed, by chaining the individual optimized GPU implementations for each algorithm together, and performing additional holistic optimization. Empirical validation of the performance for the individual algorithms and the radar signal processing chain, on the target GPU architecture, followed.

1.4 RESEARCH CONTRIBUTION

A number of general contributions are made with this research, as follows:

1. A GPU benchmarking framework (GBF) was developed in software to aid with the benchmarking of GPU codes. The GBF automates core benchmarking steps and provides common building blocks, in order to ensure repeatable results for experiments. The GBF is generic and can be applied to, and extended for, applications other than radar signal processing.

2. The summed-area table (SAT) technique, a new method of software implementation of the CA-CFAR algorithm, was identified and implemented on a GPU architecture [1]. The SAT technique is based on processing techniques that originate from the computer vision field. The performance of the SAT technique exceeds the performance obtained with other techniques for very large CFAR window sizes.

A conference paper, that is based on the initial CFAR results, was authored, published and presented at the *2011 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)* in Amman, Jordan:

1. C.J. Venter, H. Grobler, and K.A. AlMalki, "Implementation of the CA-CFAR Algorithm for Pulsed-Doppler Radar on a GPU Architecture", [1].

A journal paper, that is based on the overall results of this research, was authored and submitted to *IEEE Transactions on Parallel and Distributed Systems*:

1. C.J. Venter and H. Grobler, "Real-Time Pulsed-Doppler Radar Signal Processing on Graphics Processors", submitted for publication.

1.5 OVERVIEW OF STUDY

The rest of this dissertation is structured as follows. Chapter 2 provides background on pulse-Doppler radar. It is mainly focused on identifying the functions and requirements for a typical pulse-Doppler radar signal processor, and identifying the relevant algorithms. Chapter 3 provides background on graphics processors, with emphasis on the programming languages and programming model that is used to program these highly-parallel devices. A summary of advanced technical concepts that are directly relevant to the NVIDIA CUDA parallel framework, which is utilized in this research, is also presented in Chapter 3. In Chapter 4, an overview of existing implementations and earlier work on radar signal processing and graphics processors is presented, based on an extensive literature survey. Chapter 5 describes the systematic design and implementation of the radar signal processing chain on the target GPU architecture. The experimental results that were achieved for a number of standalone benchmarks, the individual radar signal processing algorithms, and the final radar signal processing chain follows in Chapter 6.

CHAPTER 2

PULSE-DOPPLER RADAR

2.1 INTRODUCTION

Radar is an electromagnetic system for the detection and location of objects by transmitting a waveform and detecting the nature of the received echo signal [2]. An elementary radar consists of a transmitter, antenna, receiver and signal processor [3]. The transmitter generates radio frequency electromagnetic waves that are emitted directionally by the antenna into the propagation medium, which is typically the atmosphere. The energy is intercepted by objects in the environment and re-radiated in various directions. Some of the energy is reflected back to the radar antenna and is received. The receiver amplifies, filters and down converts the signal from the antenna to an intermediate frequency (IF) for the signal processor to process. The signal processor performs processing in order to detect and extract information about objects of interest, usually called targets.

2.2 RADAR SYSTEM TAXONOMY

A brief taxonomy of radar systems is provided in this section in order to qualify the research focus with respect to the broader radar field.

2.2.1 Monostatic versus Bistatic Systems

A monostatic radar system utilizes a common antenna for both transmitting and receiving, whereas a bistatic radar uses separate antennas, that are a considerable distance apart, for either transmitting or receiving [2]. Most modern radars are monostatic, which is considered a more practical design as it requires only a single antenna [3].

2.2.2 Continuous Wave versus Pulsed Systems

Radar signal waveforms can be classified as either continuous wave (CW) or pulsed. With a CW waveform the transmitter and receiver operate continuously and concurrently transmit and receive signals. Radars that use CW waveforms are often bistatic to isolate the transmitter and receiver from each other. A pulsed waveform, on the other hand, has repeated cycles where a short pulse is transmitted, after which the receiver is turned on in order to detect the reflected target signal. Pulsed radars can make use of a monostatic configuration due to the alternating transmit and receive cycles by using a circulator or switches to achieve the necessary isolation. Most modern radars are designed to use pulsed waveforms [3] and therefore pulsed radar is considered for this research.

2.2.3 Non-coherent versus Coherent Systems

Non-coherent radar systems only detect the amplitude of received signals, whereas coherent radar systems are capable of detecting both the phase and amplitude of the received signal. With non-coherent radar systems the amplitude of the target in the received signal must exceed the amplitude of surrounding clutter in order to be detected. Pulsed coherent systems measure the phase of each received pulse relative to a reference signal, which is typically generated by a local oscillator (LO). Coherent radar can detect the motion characteristics of the target in the received signal using the measured phase information. Most modern radar systems are coherent [3] and therefore coherent radar is considered for this research.

2.2.4 Doppler Radar

Doppler radars rely on the Doppler effect in order to improve target detection [4]. Doppler shift occurs when there is relative motion between the target and the radar radially which results in a frequency shift of the return signal. Pulsed radars that employ Doppler processing are required to be coherent in order to establish a deterministic phase relationship between pulses. Doppler processing uses filtering or spectral analysis in the pulse dimension for each sampled range increment.

Moving target indication (MTI) is one method of Doppler processing which operates in the time domain and uses a high-pass filter to simply detect either the presence or absence of a moving target in a given region. MTI has a low computational intensity and produces limited information. Pulse-Doppler processing is another method of Doppler processing where the signal is processed in the frequency domain, which is computationally intensive, but provides an improvement in the signal-to-interference ratio (SIR) and potential additional target information. The pulse-Doppler method provides more interesting implementation and optimization challenges and opportunities than MTI. Pulse-Doppler radar, which utilizes pulse-Doppler processing, is therefore considered for this research.

2.3 THE DIGITAL RECEIVER

The availability of high-speed ADCs have led to the widespread adoption of digital receivers. Modern radars increasingly use digitization at IF, especially with coherent systems [3].

2.3.1 I/Q Demodulation

Conventional receivers perform analog I/Q demodulation where digital receivers use an ADC combined with digital signal processing to do the down conversion to baseband I and Q signals [4]. The digital in-phase (I) and quadrature (Q) signals preserve the amplitude and phase information required by a coherent system. The digital IQ samples are provided as input to the radar signal processor for further digital signal processing.

2.3.2 Beam Forming

The monopulse technique can be used to estimate the angular position of a target within the 3 dB antenna beamwidth using a single pulse, which allows for accurate tracking [3]. A monopulse antenna transmits multiple beams with slight offsets and on reception produces a sum signal and two difference signals in the azimuth and elevation dimensions respectively. The sum and difference signals are received and provided as three separate digital channels to the signal processor.

2.4 THE RADAR SIGNAL PROCESSOR

The primary purpose of the radar signal processor is to maximize the SIR in order to detect targets in the presence of the various interference sources that exist.

2.4.1 Interference

A received target signal competes with a variety of interference sources [3]. In addition to energy from desired targets, reflections from other undesired objects and surfaces in the environment are also unintentionally received by the radar. These undesired signals are called clutter. Thermal noise from the external environment and internal to the radar receiver itself is also present in the received signal. Both unintentional electromagnetic interference (EMI) and intentional electronic countermeasures (ECM) are also forms of interference from man-made sources.

2.4.2 Sampling Rates

The pulse repetition frequency (PRF) is the rate at which the alternating pulse transmit and receive cycle occurs, which is the reciprocal of the pulse repetition interval (PRI). The output data rate for the receiver, which is equivalent to the input data rate, DR_{in}, for the signal processor, is shown in Equation 2.1. It can be expressed as the product of the number of received samples per pulse, N_{sp}, the number of bits per sample, N_b, the PRF, and a constant factor of 2 for the I and Q channels that are required for a coherent receiver:

$$DR_{in} = 2N_{sp}N_bPRF \text{ bps} \tag{2.1}$$

The input data rate of the signal processor can be very high as a result of the multiplicative nature of the constituent parameters. If we substitute the number of received samples per pulse, N_{sp} in Equation 2.1 as the product of the sampling window period, T_f, which may not exceed the PRI, then $\frac{T_f}{PRI} \leq 1$, and the fast time sampling rate f_s then we get:

$$DR_{in} \leq 2f_sN_b \text{ bps} \tag{2.2}$$

The number of bits per sample N_b is determined by the ADC used by the receiver and is typically between 6 and 12 bits for most radar applications [3]. The sampling rate that is required is determined by the instantaneous bandwidth of the system according the well-known Nyquist sampling criteria. The instantaneous bandwidth is a critical factor for pulse-Doppler radar, as it determines the range resolution that the system can achieve. Range resolution defines the ability of the system to separate radar returns in range [3].

2.4.3 Performance Parameters

Almost all radar applications require high throughput and some applications also require low latency [3]. Throughput is amount of data that is processed by the radar signal processor in a given period of time, which is usually specified in MegaSamples per second (MS/s). Latency is the elapsed time for a burst to pass through the entire radar signal processing chain, which is usually specified in milliseconds (ms).

A higher processing throughput is required to accommodate a higher sampling rate in order to achieve a higher instantaneous bandwidth. With pulse compression, which is discussed later in this chapter, the instantaneous bandwidth becomes inversely proportional to the range resolution [3]. High throughput in the radar signal processor can therefore be beneficial due to the potential improvement in range resolution. The latency requirements for a radar signal processor is generally determined by the later processing stages, where the mission-specific data processing occurs. Low latency is required, for instance, to close a tracking loop on a fast-moving target with high dynamics. As the required throughput increases and the required latency decreases, the computational and data transfer demands on the radar signal processor increase.

2.4.4 Data Collection and Storage Model

A pulse-Doppler radar measures in spatial dimensions using a spherical coordinate system of range, azimuth, and elevation, as well as in spectral dimensions using Doppler shift for radars with a single receiver element [5]. Systems with multiple receiver elements, such as a phased array, adds another spatial dimension, which is angle of arrival.

A conceptual data model to represent these dimensions is that of a datacube, which consists of three dimensions, namely fast-time, slow-time and receiver element. The fast-time dimension contains samples directly from the receiver output after a pulse has been transmitted that can be equated to range by using the measured time delay of each sample. Individual range samples are also called range bins or range cells. The slow-time dimension contains all the pulses that were transmitted in a single coherent processing interval (CPI). For pulse-Doppler radar the CPI consists of a burst of pulses that are transmitted during a dwell at a common PRF, received, and processed coherently as a group. The CPI is casually referred to simply as a burst. Individual samples in the pulse dimension are also called Doppler bins after the Doppler processing stage. The receiver element dimension contains the samples for each radar receiver element. A single receiver element system is assumed for this research and therefore the datacube is reduced to a 2D data matrix.

2.4.5 Burst Dimensions

The characteristics of a radar system are ultimately determined by the radar waveform design, which takes design factors from a variety of radar subsystems into consideration. The effect of the waveform design on the radar signal processors parameters, such as the burst dimensions, is relevant as the radar signal processor is the primary focus of this research. However, the process of waveform design is complex and system-specific, as it generally involves considering trade-offs between a wide variety of interdependent radar parameters to achieve specific system-level goals. Waveform design will therefore not be attempted as a means to relate hypothetical radar system parameters to particular burst dimensions that are evaluated. Instead, a summary of basic relationships between the burst dimensions and radar parameters is provided, with some examples of potential trade-offs.

The number of range bins to be processed in a burst is determined by the PRI and excludes the transmit duty cycle, which is typically around 5%. The unambiguous range, which is the maximum range at which returns can be resolved unambiguously, is determined by number of range bins and the range resolution. For a specified unambiguous range, an increase in the number of range bins allows for an improved range resolution. Alternately, for a specified range resolution, an increase in the number of range bins allows for a greater unambiguous range.

The number of Doppler bins to be processed in a burst is equal to the number of pulses that are transmitted and received. The unambiguous velocity, which is the radial velocity space wherein returns can be resolved unambiguously, is determined by the PRF and the radio frequency (RF) carrier signal frequency. For a specified unambiguous velocity, an increase in the number of Doppler bins increases the integration gain. Alternately, for a specified CPI, an increase in the number of Doppler bins necessitates and increase in PRF, which in turn improves unambiguous velocity.

2.4.6 The Radar Signal Processing Chain

The input to the radar signal processing chain is a stream of digital I/Q samples, which is interpreted as the 2D data matrix structure described in Section 2.4.4. The data is processed sequentially through a number of processing stages in order to produce high-level, mission-specific radar outputs.

The first processing stage involves fixed operations that are applied to all data samples, irrespective of the content of the data, which can be referred to as data-independent processing. The primary purpose of this stage of processing is generally to improve the SIR. The radar signal processing functions in this stage include digital pulse compression (DPC), Doppler filtering (DF), constant false-alarm rate (CFAR) and synthetic aperture radar (SAR) image formation [3]. The core operations required to implement these functions include finite impulse response (FIR) filters, correlation, Fourier transforms and matrix-vector algebra, which have high computational complexity. The high computational complexity at high data rates means that high computational rates on the order of 1 GOPS to 1 TOPS can be required for the data-independent processing stage [3].

The second stage of processing consists of operations that depend on the content of the data, which can be referred to as data-dependent processing. As an example, track measurements may be made per burst during this stage and only for targets detected by the CFAR detector in the preceding stage [3]. Computational rates are lower at around 1-10 GigaOPS and data rates are also typically reduced during this stage.

The third stage of processing performs high-level, mission-specific processing such as track filtering, which typically operates over multiple bursts. This high-level processing is sometimes referred to simply as data processing and the computational rates are further reduced to around 10-1000 MegaOPS along with a reduction in data rates.

The second and third processing stages typically perform processing functions that are specialized for a particular radar application or mission. On the other hand some of the processing that is performed as part of the first stage, such as pulse compression and Doppler filtering is used across a wide range of pulse-Doppler radar applications in general. The CFAR detector is also used commonly across a broad range of pulse-Doppler radar applications for surveillance and tracking. SAR systems, on the other hand, fall under imaging radar applications, which tend to be more specialized.

2.4.7 Fixed-Point versus Floating-point

The arithmetic format that is used in the radar signal processor has a significant impact on its design and the choice is in principle between fixed-point and floating-point arithmetic [3]. In terms of digital logic, fixed-point implementations typically require less silicon real-estate and execute computations faster, due to fewer logic stages required compared to floating-point implementations. The main disadvantage of fixed-point implementation for radar applications is its limited dynamic range, with the associated potential for overflow and underflow of radar signal processing computations. Arithmetic operations such as multiplication and addition increase the dynamic range during processing. Measures can be taken to prevent overflow and underflow, such as scaling prior to certain arithmetic operations, but this reduces the precision of the result and adds extra complexity in the design.

Floating-point arithmetic overcomes virtually all dynamic range and scaling issues with the IEEE P754 32-bit single-precision format as the most common implementation. This format uses a 24-bit mantissa with an 8-bit exponent and can represent a range of about 72 orders of magnitude [3]. Another advantage of using floating-point arithmetic in the signal processor is that algorithms are often developed using floating-point arithmetic in high-level environments such as MATLAB [6] or C/C++ and the porting effort to the operational hardware is therefore reduced. The numerical accuracy and ease of use of floating-point arithmetic remains beneficial to radar signal processing applications.

2.4.8 Processor Technologies

The first processing stage may require the use of dedicated application-specific integrated circuits (ASICs) or reconfigurable logic devices such as field-programmable gate arrays (FPGAs). The second processing stage can typically be performed by dedicated programmable processors such as digital signal processors (DSPs). The third processing stage is commonly performed using general-purpose programmable processors such as central processing units (CPUs) on single-board computers that are essentially equivalent to a standard workstation.

Although floating-point arithmetic is preferred, some systems use fixed-point arithmetic for the data-independent processing, due to the high computational requirements. Floating-point arithmetic may be utilized throughout the radar signal processor by systems with low-end computational requirements or systems capable of high-end computational performance in order to ease development and provide greater arithmetic precision [3]. As processor technologies evolve due to Moore's law, more general processor technologies can be used progressively earlier in the processing chain.

2.5 PULSE-DOPPLER RADAR SIGNAL PROCESSING CHAIN DESCRIPTION

The digital pulse compression (DPC), Doppler filtering (DF) and constant false-alarm rate (CFAR) functions that form part of the data-independent stage of the signal processing chain, were selected as the focus for this research. As discussed in Section 2.4.6, the DPC, DF and CFAR functions are common in pulse-Doppler radar and are used throughout a variety of applications. The first processing stage also requires the highest computational and data rates and is therefore expected to provide the most interesting challenges and opportunities for efficient implementation on a GPU architecture, which provides both high computational performance and high bandwidth as discussed in the upcoming Section 3. Synthetic aperture radar (SAR) processing is also common, although it is more specialized and complex and is already closely related to traditional graphics implementation due to its application for imaging radar. As a result, numerous contributions of SAR processing on GPUs have been made already. Therefore due to its specialized nature, complexity and the existing body of knowledge, SAR was not included in this research.

A typical pulse-Doppler radar signal processing chain is shown in Figure 2.1 with a breakdown of the constituent primitive operations that are required to implement the DPC, DF and CFAR functions in a combined chain. The high-level functions and the required operations for the typical pulse-Doppler radar signal processing chain are described in more detail in the rest of this section.

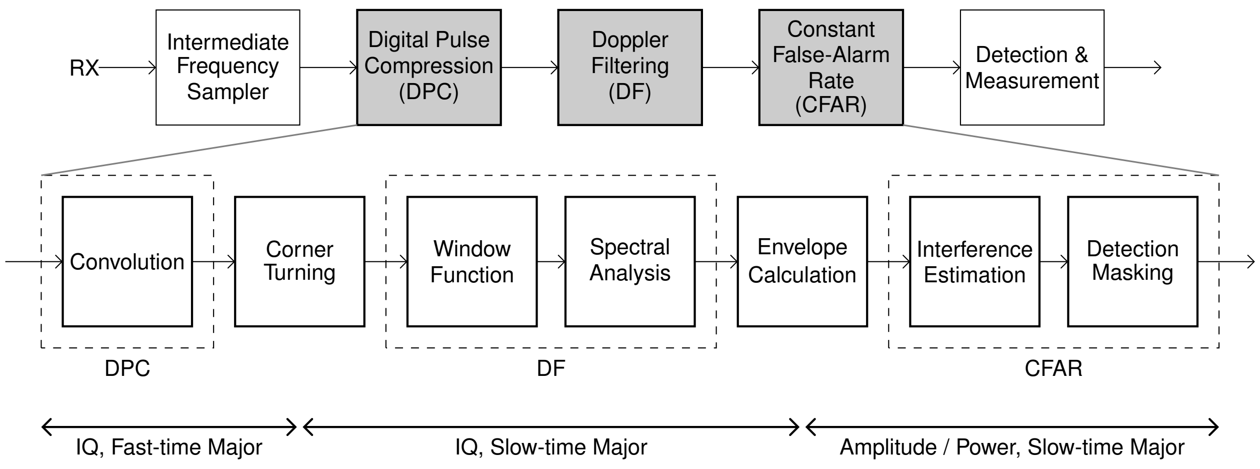

Figure 2.1: A typical pulse-Doppler radar signal processing chain for a single channel showing basic functions and constituent operations for the data-independent processing stage. The data format and processing order for different sections of the processing chain are also indicated.

2.5.1 Digital Pulse Compression (DPC)

Digital pulse compression correlates the transmit pulse waveform with received target echoes in order to effectively compress pulses to form peaks where targets are present. In their compressed form the pulses are narrower than the width of the transmit pulse waveform. Target range is indicated by the peaks with a narrower range resolution than would be otherwise possible [5]. Pulse compression in conjunction with frequency-modulated pulses also effectively decouples the pulse width and range resolution, where the latter is an important system performance parameter. Consequently, longer pulses with greater average power can be utilized, which has distinct benefits for the transmitter design, without compromising on range resolution.

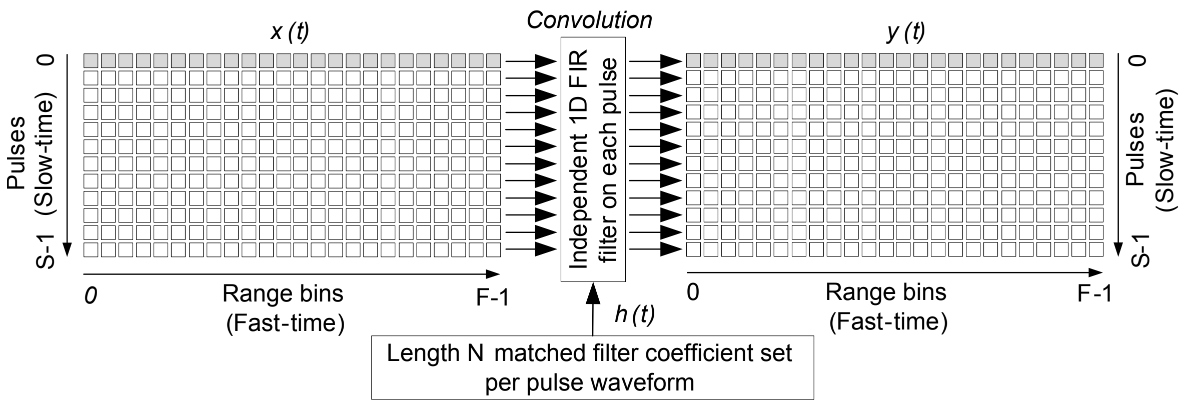
x (t)
Convolution
y (t)
Pulses (Slow-time)
0
S-1
Independent 1D FIR filter on each pulse
0
Range bins (Fast-time)
F-1
h (t)
Length N matched filter coefficient set per pulse waveform

Figure 2.2: Digital pulse compressor structure showing samples for a single pulse highlighted.

A matched filter is typically implemented to perform the correlation using a finite impulse response (FIR) filter with filter coefficients matched to each transmit pulse waveform. As the pulse width determines the required filter length, longer filter lengths are required to process longer pulses. The FIR filter output is obtained by the convolution of its input signal $\mathbf{x}$ with its impulse response $\mathbf{h}$ as $\mathbf{y} = \mathbf{x} \circledast \mathbf{h}$. The optimum matched filter coefficient vector $\mathbf{h}$, can be shown to equal the complex conjugate of the desired signal, where interference is white noise only. Hence, with the transmit pulse waveform $\mathbf{s}$ as the desired signal for digital pulse compression, $\mathbf{h} = \mathbf{s}^*$ represents the optimum matched filter coefficient vector. Figure 2.2 shows the overall structure for DPC where 1D convolution is performed independently on every pulse using the appropriate coefficient set for the given transmit pulse waveform.

2.5.1.1 Time-domain Convolution Method

Time-domain convolution or direct convolution of two finite length discrete sequences can be implemented using the convolution sum shown in Equation 2.3, where $h(k)$ and $x(n-k)$ are zero outside of their respective ranges.

$$y(n) = \sum_{k=0}^{n} h(k) \cdot x(n-k) \tag{2.3}$$

Direct convolution is very computationally intensive, especially for long input sequences.

2.5.1.2 Frequency-domain Convolution Method

Convolution can be performed in the frequency-domain using multiplication according to the convolution theorem $\mathbf{x} \circledast \mathbf{h} \leftrightarrow \mathbf{X} \circ \mathbf{H}$. A Fourier transform of the input signal $\mathbf{x}$ and impulse response $\mathbf{h}$ is required, as well as an inverse Fourier transform on the multiplication result in order to revert the final result back to the time-domain, as shown in Equation 2.4. For matrices the element-wise multiplication operation takes the form of a Hadamard product.

$$\mathbf{y} = \mathscr{F}^{-1}\{\mathbf{Y}\} = \mathscr{F}^{-1}\{\mathbf{X} \circ \mathbf{H}\} = \mathscr{F}^{-1}\{\mathscr{F}\{\mathbf{x}\} \circ \mathscr{F}\{\mathbf{h}\}\} \tag{2.4}$$

For the DPC function $\mathbf{H} = \mathscr{F}\{\mathbf{h}\} = \mathscr{F}\{\mathbf{s}^*\}$ can be precomputed [3], as each transmit pulse waveform $\mathbf{s}$, that is usable by the system is generally known *a priori*.

2.5.2 Corner Turning (CT)

A signal processing chain that operates on multidimensional data may have differing optimal data layouts for different processing stages. A modification of the underlying storage order of the data may be desirable for efficiency reasons. A corner turn (CT) operation can be used in cases where a signal processing chain first operates in one dimension such as rows and subsequently in a different dimension such as columns [7]. The corner turn operation transposes the data matrix dimensions in order to preserve data locality in the dimension operated on as shown in Equation 2.5, although a mathematical transpose is not necessarily implied and depends on the implementation.

$$\mathbf{O} = \mathbf{I}^{\mathsf{T}} \tag{2.5}$$

The DPC operates in the range dimension or fast-time dimension, which corresponds to the temporal sampling order. Data samples that are temporally adjacent are also typically stored in adjacent memory locations. Therefore, the default data arrangement is range major. However, the DF operates in

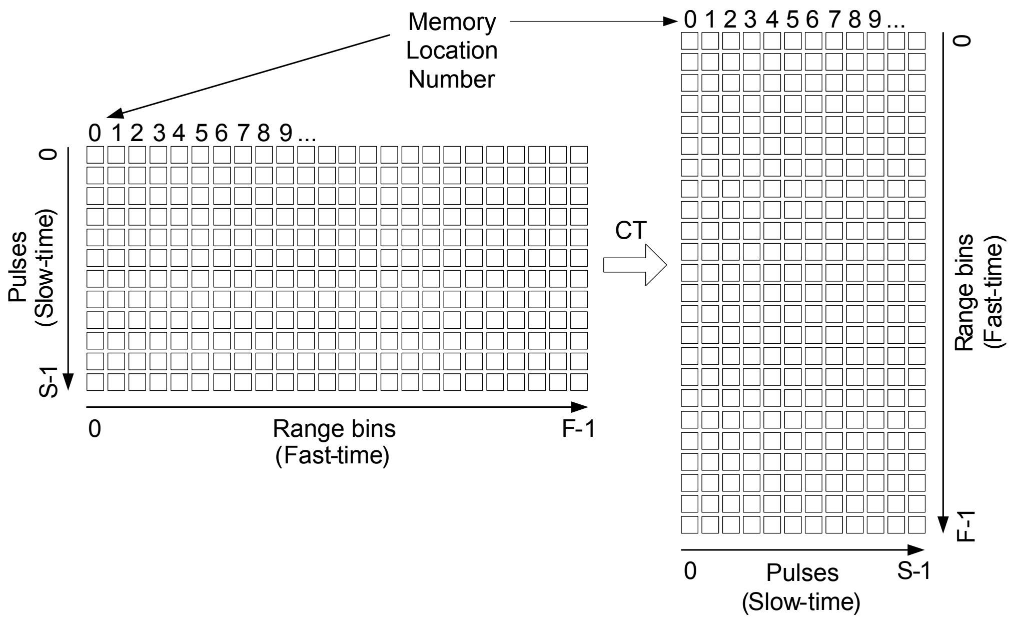

Figure 2.3: Corner turning structure showing transformation of memory layout.

the pulse / Doppler or slow-time dimension. As a result, a CT operation is commonly performed prior to the DF in order to rearrange the data matrix into a pulse / Doppler major format for optimal data processing efficiency, as shown in Figure 2.3. Generally the CT operation does not mathematically transform the data matrix at all and thus, effectively performs no processing.

2.5.3 Doppler Filter (DF)

The Doppler filter performs spectral analysis in the slow-time dimension across all pulses in a burst. Accordingly, a pulse / Doppler major data layout, as provided by the CT function, is typically used for optimal data locality to achieve optimal performance.

2.5.3.1 Window Function

Suppression of sidelobes in the Doppler spectrum is important for radar applications in order to avoid potential masking of the main lobe of a weaker target return in the sidelobes of a stronger target return. Received power levels for different targets can vary with several tens of decibels due to differences in radar cross-section (RCS) and range [3].

A window function is often applied prior to spectral analysis, in order to reduce the sidelobe levels of the Doppler spectrum in the frequency domain. The window typically consists of a smooth curve of real values with a maximum value at the center that is tapered down to a lower value at the endpoints. An element-wise multiplication of the window coefficients with the input sequence is used to apply the window function, as shown in Figure 2.4.

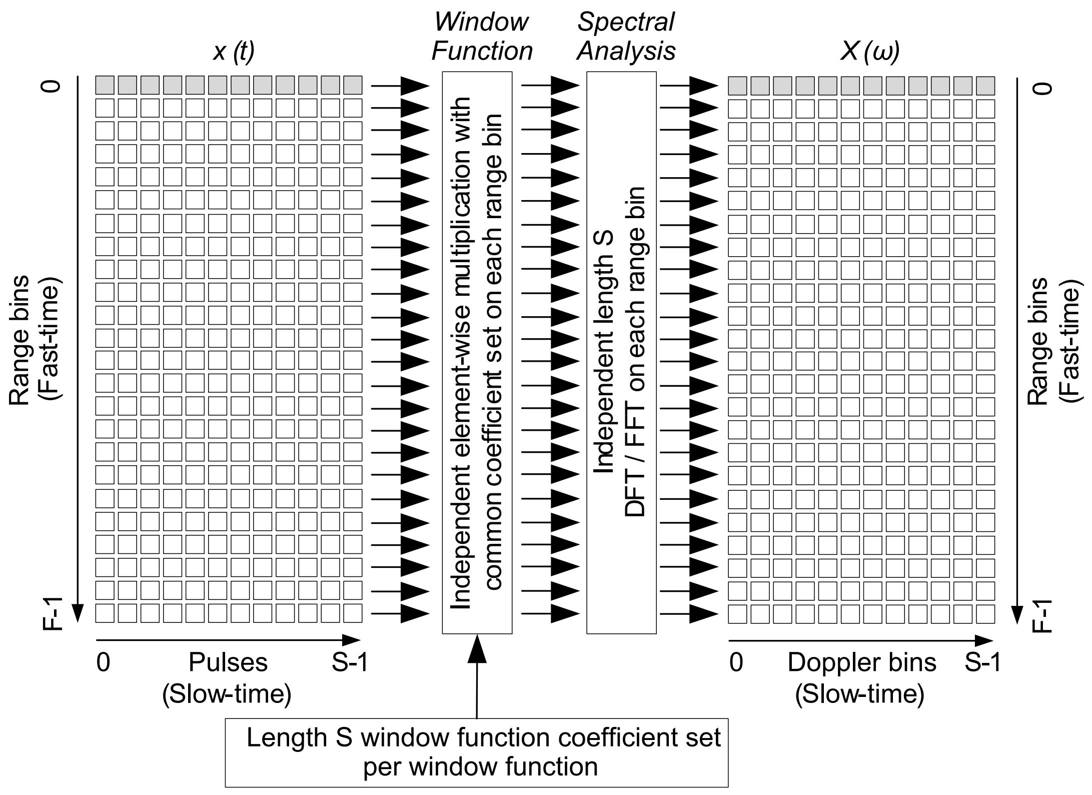
x (t)
Window Function
Spectral Analysis
X (ω)
0
Range bins (Fast-time)
F-1
Independent element-wise multiplication with common coefficient set on each range bin
Independent length S DFT / FFT on each range bin
0
Range bins (Fast-time)
F-1
0 Pulses S-1
(Slow-time)
0 Doppler bins S-1
(Slow-time)
Length S window function coefficient set per window function

Figure 2.4: Doppler filter structure showing samples for a single range bin highlighted.

The typical window shape reduces the energy at the edges of the input sequence, effectively reducing the width of the signal in the time domain. This leads to a wider main lobe in the frequency domain, due to the reciprocal spreading property of the Fourier transform. Most importantly, the sidelobe levels are greatly reduced in the process. Hamming, Kaiser, Taylor, Hann, Dolph-Chebyshev, and rectangular window functions are commonly used, where the latter effectively disables the window function [3].

2.5.3.2 Spectral Analysis

During spectral analysis the slow-time dimension is transformed from pulses into Doppler bins where each cell contains frequency information that can be used to extract radial velocity. Spectral analysis is typically performed using 1D direct discrete Fourier transforms (DFTs) or fast Fourier transforms (FFTs) across all pulses in a burst for each range bin as shown in Figure 2.4.

2.5.4 Envelope Calculation (ENV)

An envelope calculation, or linear rectifier, outputs amplitude or voltage by calculating the magnitude of the complex samples, as shown in Equation 2.6. A square law rectifier outputs power that is calculated as the envelope or magnitude squared, as shown in Equation 2.7. [3].

$$y = |x| = |a+bi| = \sqrt{a^2+b^2} = A \tag{2.6}$$

$$y = |x|^2 = |a+bi|^2 = a^2+b^2 = P \tag{2.7}$$

The output of the envelope calculation is typically passed to a detector that does not require the phase information, which is lost during the envelope calculation, but is required for coherent Doppler processing earlier in the processing chain. The square law detector is considered more convenient for implementation as it does not require a square root operation. Nonetheless, the type of rectifier that is used ultimately depends on whether a linear or square law detector is required, where the respective linear and square law rectifiers are appropriate.

2.5.5 Constant False-Alarm Rate (CFAR)

Varying interference from a variety of potential sources as described in Section 2.4.1 is assumed to be present in all radar measurements. Basic target detection is therefore performed by comparing radar measurements to a threshold. Target returns are assumed to be present where measurements exceed the threshold. When the threshold is exceeded due to interference alone, a false alarm occurs, degrading system performance due to false detections that are passed on for further processing [3]. A radar system can typically process a certain maximum number of detections per time period downstream and therefore a constant false-alarm rate is desirable.

A CFAR detector uses an adaptive detection threshold to maintain a constant false-alarm rate in the presence of varying interference. The CFAR calculates a local interference estimate and subsequently a detector threshold for each cell, based on a desired overall false-alarm probability, $\overline{P}_{fa}$. The CFAR is commonly implemented as a square law detector for radar applications as the requisite mathematical analysis is relatively tractable [3] and it will therefore be considered for this research. The CFAR input is consequently considered to be sample power values generated by a square law rectifier.

A very common class of CFAR detector is the Cell-Averaging CFAR (CA-CFAR), which averages the power of a block of cells around the cell under test (CUT) to obtain a local interference estimate. The CA-CFAR is a basic detector that assumes a homogeneous interference environment in a localized area, which is unlike operational environments where interference is generally heterogeneous [3]. The CA-CFAR was selected as the focus of this research due to its simplicity and ubiquity in literature.

2.5.5.1 Interference Estimation

The CA-CFAR estimates interference for each cell by averaging the interference power in a reference window or CFAR window of N cells around the CUT, as shown in Equation 2.8 and Figure 2.5.

$$\widehat{\beta^2} = \frac{1}{N}\sum_{i=1}^{N} x_i \tag{2.8}$$

When a target is located in the CUT, some returned power may spill into nearby cells. Inadvertently including target power in the interference estimate raises the detector threshold unnecessarily, lowering its sensitivity and causing degraded detection performance. As a result, guard cells are placed around the CUT to exclude residual target power from the interference estimate.

2.5.5.2 Detection Masking

The detection threshold, T can be calculated as a function of the estimated interference power, $\widehat{\beta^2}$ and the CFAR constant, α as shown in Equation 2.9 and Figure 2.5. The α constant is a function of the desired false-alarm probability, $\overline{P}_{fa}$ and the CFAR window size, N [5], as shown in Equation 2.10.

$$T = \alpha\widehat{\beta^2} \tag{2.9}$$

$$\alpha = N(\overline{P}_{fa}^{-1/N} - 1) \tag{2.10}$$

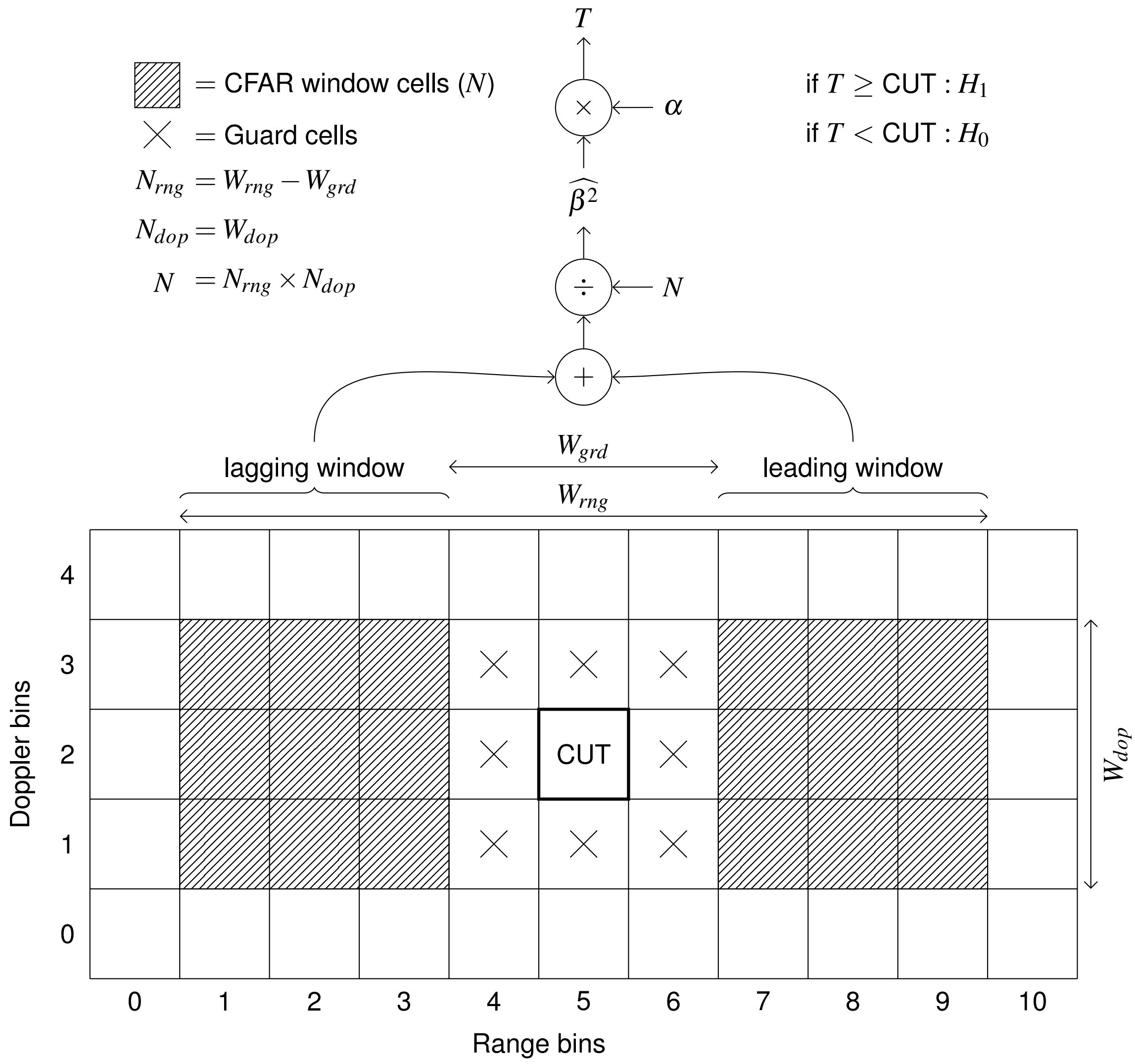

Figure 2.5: Constant false-alarm rate structure showing a 2D CA-CFAR detector with the calculation that is performed for each cell to determine whether a target is present in the given CUT.

The detector compares the measured value to the detection threshold for each cell to choose between two hypotheses. Where the measured value equals or exceeds the detector threshold, the H_1 hypothesis, which states that the cell contains a target plus interference, is selected. The H_0 or null hypothesis, which states that the cell only contains interference, is selected where the measured value is lower than the detector threshold [5]. A detection mask is then generated by marking cells where H_1 is selected. The detection mask is used downstream in the processing chain to extract and process detections further.

2.6 CONCLUSION

The data-independent stage of a typical pulse-Doppler radar signal processing chain is computationally intensive and presents implementation challenges to satisfy demanding real-time requirements. Data dependencies exist between the individual radar signal processing functions in the processing chain, that does not entirely preclude task-level parallelism, but may require the use of pipelining techniques. Multiple radar channels may also be processed in parallel for the data-independent processing stage, where combination of inter-channel data is not required yet. Individual processing functions also present considerable data-level parallelism that may be exploited, due to the multidimensional nature presented by the typical data storage model.

CHAPTER 3

GRAPHICS PROCESSORS

3.1 INTRODUCTION

Graphics processing units (GPUs) were originally designed as dedicated co-processors to offload graphics processing from the CPU. There are several characteristics of graphics processing that differentiate it from traditional general-purpose computation. High data rates and computational performance is required to render complex scenes, especially in 3D, at interactive update rates. However, graphics operations are typically applied to large sets of independent graphical data elements, such as pixels and therefore substantial parallelism is exhibited. Hence, the GPU architecture is optimized to exploit the intrinsic parallelism in graphics processing to achieve high performance.

3.2 ARCHITECTURAL OVERVIEW

High computational performance is achieved in GPUs by dedicating transistors to computation logic in the form of abundant, but simple, processing cores for data-parallel processing. General-purpose processors such as CPUs favor large caches and fewer, but more complex, processing cores with sophisticated logic for instruction-level parallelism. GPU computation is structured as a graphics rendering pipeline which is designed to sustain high computation rates through data-level parallelism in the absence of large caches to reduce latency [8]. To complement the high computational performance, memory controllers were also integrated into GPU chips and specialized graphics memory modules were developed to increase bandwidths well beyond what is available to CPUs [9].

GPUs traditionally featured fixed-function pipelines with limited programmability. The vertex and fragment processing stages of the graphics pipeline were later made programmable by introducing vertex and fragment or pixel shader units [10]. The shader units are floating-point stream processors that provided increasing flexibility to implement a variety of custom real-time visual effects by using specialized programs, written in low-level shader languages. The separate shader units have since been unified, allowing execution of any type of shader program. Continued improvements in the programmability of the graphics pipeline has led to the development of high-level GPU programming languages. Modern GPUs have evolved into generalized, many-core, SIMD architectures with powerful instruction sets that provide rich opportunities for general-purpose computation on GPUs (GPGPU).

3.3 GPGPU

GPGPU is the use of GPUs as general-purpose parallel processors through accessible programming interfaces. The term GPGPU was coined in 2002 and GPGPU.org [11] was founded by Mark Harris when he recognized the trend for non-graphics use of GPUs. High-level languages have contributed to the increasingly widespread use of GPGPU in a wide variety of computationally intensive applications [8].

3.4 PROGRAMMING LANGUAGES

High-level GPU programming languages have evolved significantly over the past few years. A summary is presented in this section, highlighting the key milestones.

3.4.1 High-level Shader Languages

Low-level, hardware-specific shader languages that resemble assembly language were introduced initially to program GPU shader units. High-level shader languages such as NVIDIA's Cg [12], Microsoft's high-level shading language (HLSL) [13], and OpenGL shading language (GLSL) [14] were developed to increase developer productivity and code portability across hardware generations. These high-level shader languages are C-like and allow for real-time procedural shading. Cg or C for graphics is a high-level shader language developed by NVIDIA that runs on Direct3D [15] and OpenGL [14]. Cg is intended to be a general-purpose, hardware-oriented language similar to C, but for graphics processing [16]. HLSL only supports the Direct3D API. GLSL consists of several closely related language variants for each of the processors in the OpenGL graphics pipeline [17].

These shader languages provide great flexibility to the graphics pipeline though high-level programmable shaders. Even so, shader languages do not provide control over other aspects of the graphics pipeline such as memory allocation and loading of shader programs, which requires a main program written in traditional general-purpose languages such as C/C++. Algorithms also have to be expressed in terms of graphics primitives such as polygons and textures, which limits general-purpose GPU computing implementation to advanced graphics developers.

3.4.2 Brook

Brook [18] is a streaming language that is an extension of ANSI C, which incorporates parallel computing paradigms into a familiar, efficient language. Brook was extended for GPUs [19] by abstracting the GPU as a streaming processor through the use of streams, kernels, and reduction operators, such as arithmetic sum. Brook is open source and overcomes many of the limitations faced by using standalone shader languages for GPGPU. Brook can target a variety of streaming architectures and is therefore devoid of any explicit graphics constructs.

Streams are represented as floating point textures on the graphics hardware. Kernels are first compiled into Cg code as an intermediate shader language by the Brook compiler, after which standard vendor shader compilers are used to compile executable shaders. Brook also has a runtime component, which supports both Direct3D and OpenGL backends. Brook+ is an implementation of Brook for Advanced Micro Devices (AMD) processors which offers some additional features, but it is no longer actively maintained. Although Brook paved the way to widespread GPGPU adoption, it suffers from a variety of intrinsic problems caused by operating via native graphics APIs and their drivers. Brook remains in beta and is primarily used by curious programmers and researchers.

3.4.3 CUDA

NVIDIA Compute Unified Device Architecture (CUDA) [20] is a proprietary parallel computing platform and programming framework that is supported on a variety of NVIDIA GPU devices. At the hardware level, NVIDIA GPUs are built on the CUDA architecture, which allows the GPUs to perform both traditional graphics rendering and general-purpose processing tasks [21]. At the software level, CUDA provides the CUDA C language, which is essentially C with extensions to program massively parallel, CUDA-enabled GPUs. A variety of CUDA libraries are provided, such as the CUBLAS [22] linear algebra library and CUFFT [23], which is a fast Fourier transform (FFT) library. CUDA also provides a hardware driver that operates as part of a CUDA runtime component. CUDA C is the first language designed by a GPU manufacturer specifically to facilitate general-purpose computation on GPUs [21]. CUDA has been used in a wider variety of applications than any other GPU computing technology and is also used in a number of commercial software packages [9].

3.4.4 OpenCL

Open Computing Language (OpenCL) [24] is an open standard for parallel programming of heterogeneous systems that includes CPU, GPU, DSP and other architectures. OpenCL includes support for a wide variety of NVIDIA and AMD GPUs. It is a portable language, which is also based on C with a number of extensions for parallel programming. OpenCL is currently the prescribed parallel framework for AMD GPUs.

Although OpenCL and CUDA have a similar architecture with respect to memory model, programming model and a C-based syntax, OpenCL tends to be more tedious to program [25]. OpenCL has been found to provide worse performance in straightforward comparisons with CUDA on NVIDIA GPUs [26, 27, 28]. The performance differences are generally attributed to the greater generality of OpenCL, limitations in terms of automatic compiler optimizations and the behavior of the programmers and users themselves. Nonetheless, some conclude that OpenCL is a good alternative to CUDA under fair comparisons and that its portability does not fundamentally affect its performance [28].

3.5 HARDWARE VENDORS

NVIDIA and AMD own the majority share of the discrete GPU market that is driven primarily by high-performance desktop computing, primarily in the form of computer gaming. The majority of these devices are therefore aimed at the mainstream gaming and graphics markets. AMD's Radeon X1900 series gaming graphics cards, based on the R580 architecture and released in 2006, were the first stream processors designed for GPGPU to be accessed via a thin hardware interface called Close To the Metal (CTM). NVIDIA's GeForce 8800 series gaming cards, based on the G80 architecture and released later in 2006, were the first devices with the NVIDIA CUDA. The 8800 GTX in particular has been used widely for GPGPU and GPGPU research since the release of the first version of CUDA shortly after the release of the card itself. Most of the subsequent mainstream graphics cards from

NVIDIA and AMD also support GPGPU, although dedicated GPGPU cards have also since become available.

The NVIDIA Tesla series [29] and AMD FirePro series [30] are aimed specifically at GPGPU and typically provide more main memory and often even exclude the graphics hardware required to output a video signal. These GPGPU cards also provide error-correcting code (ECC) memory and better support for double-precision floating-point computation. Specialized tools for managing and monitoring the GPGPU series devices, such as the NVIDIA System Management Interface, `nvidia-smi`, are also available and provide limited support for mainstream graphics cards.

NVIDIA graphics hardware with the NVIDIA CUDA GPU computing technology was selected as the focus for this research and for the implementation. NVIDIA currently owns most of the high-performance computing (HPC) market share with regard to GPU computing. NVIDIA Tesla series and GeForce series graphics cards were also already available to the author for use in this research. NVIDIA CUDA is very widely used as a GPU computing technology and provides enough low-level control over GPU resources to accommodate a thorough exploration of the GPU architectural features, while still providing the advantages of a high-level language. Although CUDA does not provide for parallel programming of heterogeneous systems as OpenCL does, a homogeneous GPU-based radar signal processor is proposed and investigated in this research. The performance and usability of CUDA compared to OpenCL is expected to be the same or better based on the initial survey that was performed.

3.6 NVIDIA CUDA OVERVIEW

The GT200 architecture, which followed the first CUDA-capable G80 architecture, added a number of features and enhancements, including native double-precision floating-point support. The GT200 and G80 architectures are both first generation CUDA-capable architectures that are also codenamed Tesla, where Fermi and Kepler are second and third generation, respectively. Note that the Tesla brand name was eventually adopted by NVIDIA to refer to all GPUs in the GPGPU or server GPU series, irrespective of the architecture generation, where its initial use was for the architectural codename only. The GT200 architecture will form the basis of the GPU architecture considered for this work. A description of the GT200 microarchitecture drawn from [9] and [31] follows.

3.6.1 Microarchitecture

A scalable streaming processor array (SPA) performs all the programmable computations on a GPU chip. The SPA contains 10 texture processing clusters (TPCs), which are independent processing units that make the architecture scalable. Low-end GPUs may feature only 1 or 2 TPCs where high-end GPUs may feature several dozen. Each TPC provides hardware texturing support and contains 3 streaming multiprocessors (SMs), which share a texture cache. The SM is a unified processor that can execute a variety of shader programs and compute (GPGPU) programs. Each SM provides 8 RISC scalar processor (SP) cores, which are 32-bit ALUs that each contain a scalar multiply-add (MAD) unit. The SPA therefore contains a total of $10 \times 3 \times 8 = 240$ processor cores that support single-precision floating-point and integer arithmetic. In addition, the SM also contains two special function units (SFUs), capable of performing transcendental functions; a double-precision floating-point unit; a multithreaded (MT) instruction unit with cache; a constant cache; a shared memory region; and a register file.

TPCs in the SPA are issued with work by dedicated graphics and compute (GPGPU) work distribution units. A host interface communicates with the host CPU to provide commands to the work distribution units and it transfers data between the CPU and GPU memory via a memory subsystem. The memory subsystem provides an interconnect fabric and an interface to the GPU external DRAM. Fixed-function raster operation processors (ROPs) are also available and can perform color and depth operations directly on the memory, but their use is generally reserved for graphics processing.

3.6.2 Single Instruction Multiple Thread (SIMT)

The SM uses an SIMT streaming architecture, which is similar to an SIMD vector architecture, but with important differences nonetheless. With both paradigms, a single instruction is executed for multiple data elements in parallel. However, with SIMT an instruction is executed on independent threads in parallel and not on a data vector alone. With SIMD the entire vector width is exposed to the software, whereas SIMT automatically manages execution and branching in hardware according to the behavior that is defined for each individual thread in software. The SIMT streaming architecture also provides for arbitrary function evaluation, where SIMD vector architectures provide for simple math operations only [19]. The SIMT width is defined as a group of 32 threads, called a warp, that are managed in hardware by the MT instruction unit on the SM.

Each SM can manage a pool of up to 32 concurrent warps, based on resource availability. The SM only executes a single warp at any given stage and uses a hardware warp scheduler to perform a zero-latency warp context switch to transfer execution to another warp. To hide latency, the warp scheduler switches to a warp that is ready to execute when the active warp has stalled due to a memory request, a barrier synchronization or as a result of register dependencies.

A single instruction is broadcast to all 32 threads in a warp to be executed in parallel, since each SM only contains a single MT instruction unit. All threads in a warp therefore simultaneously execute a single instruction in lockstep, although individual threads can be selectively made inactive due to the SIMT architecture. When threads within a warp diverge due to conditional branching, the execution of each branch is serialized, with threads not participating in a given branch made inactive. Each thread has its own registers and instruction address to maintain its context.

3.6.3 Memory Hierarchy

CUDA exposes a deep memory hierarchy that necessitates a good understanding in order to achieve optimal performance for GPU implementation. A summary of the memory hierarchy is presented in this section with advantages and pitfalls highlighted.

3.6.3.1 Host Memory

Host memory refers to memory that is accessible by a host CPU and device memory refers to memory that is accessible by the GPU. In the general case, host and device memory spaces are distinct and therefore require copy operations across a peripheral component interconnect express (PCIe) bus to exchange data. The CUDA API exposes access to certain operating system facilities related to host memory allocation to allow for optimal performance [9], which are described in this section.

Pageable memory is the default host memory allocation scheme. Pageable memory is subject to demand paging by the operating system, which means that it may be swapped out to disk or moved by the operating system. Pageable memory is therefore not suitable for direct access by the GPU. CUDA uses staging buffers that are suitable for data transfer to and from the GPU to overcome this problem, but at a substantial performance penalty due to the additional copy operation that is required [9].

Pinned memory or page-locked host memory can be allocated, which means that the operating system guarantees that it is backed by physical memory and that it will never be paged out to disk. This allows the GPU to use dedicated on-chip direct memory access (DMA) hardware to copy the data directly, as the physical address of the buffer is immutable and known [21]. Certain CUDA functions, such as asynchronous memory copies, also specifically require pinned host memory buffers. However, the amount of pinned host memory that can be allocated is strictly limited to the amount of physical memory available in the system and should therefore not be used indiscriminately.

Mapped pinned memory can also be allocated to map the host memory buffer into the GPU address space for direct access from kernels, which obviates the need for explicit memory copies between the host and device. However, the penalties associated with uncoalesced memory transfers are more severe when kernels access mapped pinned memory on the host, compared to accessing device memory. Mapped pinned memory is also referred to as zero-copy memory [21], since it removes the need for explicit memory copies and it further completely removes the need for a copy in systems with integrated GPUs.

Write-combined pinned memory can be allocated to bypass the primary CPU cache and prevent snooping of data on the PCIe bus in order to improve copy performance between the host and device. However, write-combined memory is generally best reserved for buffers that will only be read by the GPU, as CPU reads from these buffers can be up to 6 times slower. Systems with a front-side bus can generally benefit from write-combined memory, but on newer systems without a front-side bus it is best avoided entirely [9].

Portable pinned memory can be allocated to map host memory into the memory space for all GPUs in the system instead of only the active GPU at the time. No additional performance benefit is realized, but it is required to allow all GPUs to enjoy the benefits of the pinned memory allocation, should they access the buffer. It is recommended to allocate portable host buffers for multi-GPU applications [9].

3.6.3.2 Device Memory (Off-chip)

The device memory is a large dedicated pool of DRAM that is accessible to the GPU via an integrated memory controller. The device memory is accessible in the CUDA address space and it is embodied by a variety of fundamental CUDA memory abstractions that are discussed in this section.

Global memory provides the primary method for kernels to read and write device memory [9]. Global memory is not cached on the Tesla architecture under consideration. The hardware optimizes global memory data access for a warp by attempting to combine individual requests into the minimum number of 32-byte, 64-byte or 128-byte transactions, which is called coalescing. The efficiency of the coalescing depends on specific size and alignment requirements and generally requires contiguous data access patterns for optimal performance.

The requirements for different hardware generations vary [32], but are generally less strict with newer architectures. Global memory coalescing is rightly regarded as a high priority optimization as it determines how efficiently device memory bandwidth is utilized. A performance improvement of up to an order of magnitude may be observed for a kernel with optimal global memory coalescing versus a kernel with poor coalescing.

Local memory resides in device memory and is used to store kernel stack variables under certain conditions. When register space is exceeded by a kernel, the compiler will automatically spill registers into local memory. In addition, when local arrays are declared in a kernel and accessed with indices that the compiler cannot resolve, the array will be allocated in local memory [32]. The use of local memory is generally undesirable, as it is also not cached and the access latency is two orders of magnitude greater than the register access latency.

Texture memory also resides in the device memory, but provides an alternative to global memory with specific differences. Texture memory is read-only with respect to kernel access. Texture memory is cached in a dedicated texture cache per TPC that is optimized for spatial locality [32], which reduces the device memory bandwidth required. Dedicated texturing hardware also provides special addressing and conversion functions that are performed outside of the kernel. As a general rule the use of texture memory is not essential to achieving optimal performance. Even so, texture memory is a good candidate for consideration in cases where kernel access patterns do not allow for optimal global memory coalescing, but still exhibit spatial locality.

Constant memory resides in device memory and is cached in a dedicated constant cache per SM, which reduces the device memory bandwidth required. Constant memory is read-only with respect to kernel access and optimized to broadcast constant data to multiple threads.

3.6.3.3 Shared Memory (On-chip)

Shared memory provides a fast on-chip memory block per SM that can be used to share data between threads in a block and as a cache to reduce the requisite device memory bandwidth. In contrast to implicit caches that are managed in hardware, the allocation and access of shared memory is managed explicitly by the programmer. The shared memory access latency from a kernel is two orders of magnitude lower than global memory access latency and it can be a vital resource to achieving high performance. A common technique is to load blocks of data that are accessed repeatedly from the global memory into shared memory. Intermediate results may also be stored in shared memory with the final results stored back to global memory. However, in the general case, synchronization is required among threads when using shared memory between load, process and store stages, which is accomplished by calling special instructions.

The shared memory block for each SM is divided into banks that are accessed simultaneously in order to achieve high bandwidth. An n-way bank conflicts occurs when $n > 1$ simultaneous memory requests are issued to a same bank, which leads to serialization of the requests and therefore reduced performance. Special cases do exist where performance is not compromised, such as a conflict-free broadcast that is performed when all threads access the same data element. For optimal performance, bank conflicts should be minimized by structuring memory access according to the documented guidelines for each architecture [32].

3.6.3.4 Registers (On-chip)

A large register file is present on each SM and provides the fastest memory access to kernels [9]. Registers are allocated to each thread when a block is assigned to an SM during a kernel launch. The scope for a register is local to the individual thread that it is allocated to. The number of registers used per thread is important since it effectively determines the maximum number of resident threads per SM, as it is often the limiting factor. The number of registers that are used per kernel can be reported and/or limited by the `nvcc` toolchain using appropriate flags.

Access to registers generally introduces no extra latency per instruction, although read-after-write register dependencies do introduce additional latency [32]. Latency hiding can reduce or even eliminate this additional latency due to the hardware warp scheduler in the SIMT architecture, as described in Section 3.6.2.

3.6.4 CUDA C

CUDA C is based on the industry standard C programming language and adds a small set of language extensions for parallel programming of the GPU. CUDA C provides an easy transition to parallel programming of GPUs for developers that are familiar with C. Additional keywords are added for function type qualifiers, variables type qualifiers, built-in types, built-in variables and intrinsic device functions for synchronization, memory fencing, and mathematical calculations. Many of the language extensions are reserved exclusively for use in device code. Despite the similarity of CUDA C to C, it also has several restrictions such as function recursion, function pointers and static variables that cannot be used in device code.

A high-level CUDA Runtime API and a low-level CUDA Driver API are provided for execution on the host, where the former is built on top of the latter [32]. CUDA applications can be written using CUDA C with a choice of one of the two APIs. The Runtime API provides language integration for launching kernels inline using special syntax and automation of low-level resource management tasks for initialization, context management, and device code module management [33]. The Driver API provides for more direct control over resource management for applications that require the extra level of control. Due to the higher level of abstraction and language integration, Runtime API applications require the `nvcc` toolchain where Driver API applications are compiler independent as it provides a pure C API. In general, the Driver API does not provide a performance benefit over the Runtime API [9] and consequently the Runtime API was selected for the implementation as part of this research.

3.6.5 Kernels

A CUDA kernel is a function that is defined in CUDA C and which is executed N times in parallel by N threads on the GPU device [32]. A special, compact syntax is used to launch a kernel with the Runtime API where several API calls are required with the more verbose Driver API. A kernel does not have a return type, but instead reports results back via device memory [9]. A kernel describes the innermost loop of work per thread where outer loops are configured with the kernel launch configuration.

3.6.6 Threading Model

With the CUDA threading model a kernel is launched as a grid of blocks of threads [9]. The total number of threads that are available for a given kernel launch is determined by the grid and block dimensions, which are specified explicitly by the programmer as the kernel launch configuration. For convenience, the grid and block dimensions may be specified using multiple dimensions to more closely resemble the problem dimensionality. The maximum grid dimensions are specified as 65535 by 65535 with the maximum block dimensions as 512 by 512 by 64.

Each block in a grid is assigned to an SM and executed independently of all other blocks, which enables transparent scalability of the architecture by varying the number of SMs on a GPU. The execution order of blocks is also not guaranteed and all blocks in a grid do not necessarily run concurrently, which means that grid size is not restricted by the resources that would be required to run all blocks concurrently. Due to the independent execution of blocks, there is also no mechanism to communicate between blocks during kernel execution. The decomposition of the problem into independent blocks that can execute in parallel is therefore an important consideration for GPU implementation.

Each thread in a block has access to its own set of registers, which include special-purpose read-only registers for built-in variables that provide thread and block indices and dimensions. These built-in variables have reserved keywords in CUDA C and they are commonly used to determine which part of the problem is allocated to a thread [9]. A global thread index can be calculated from the built-in variables and it is commonly used directly by threads as a unique index to a global memory buffer. It is also possible to calculate the warp index and lane index, which is the index of a thread within a warp, as the mapping of threads to warps within a block is deterministic. Intra-warp behavior must be considered for optimal performance, but can safely be ignored by assuming fully independent thread execution, which will still produce functionally correct results. Nonetheless, in rare cases some authors have found that SIMT behavior differs from the expected behavior for truly independent threads [34]. Shared memory provides a mechanism for threads within a block to share data, communicate and synchronize, as described in Section 3.6.3.3.

3.6.7 Concurrent Host/Device Transfer and Kernel Execution

The standard `cudaMemcpy` data transfer function and the typical kernel execution syntax performs all operations in sequence in the so-called NULL stream. However, the CUDA API provides functionality in the form of CUDA streams and asynchronous data transfers, which allows for independent operations to potentially overlap. A CUDA stream is an abstraction for a logical sequence of operations, where operations from different streams may execute concurrently. Whether operations actually execute concurrently depends on architectural support, operation types, command issue order, and actual runtime timing conditions. Asynchronous data transfers can be performed using the `cudaMemcpyAsync` function, which allows for a stream ID to be specified. However, asynchronous transfers explicitly require host memory buffers to be allocated as pinned and cannot function on pageable host buffers. The kernel execution syntax also provides an optional argument for a stream ID to be specified.

3.6.8 nvcc

A CUDA Compiler Driver, `nvcc` is provided by CUDA to hide the intricate details of the CUDA compilation steps from developers [35]. The compilation process for the `nvcc` toolchain involves several steps of splitting host and device code, preprocessing, compilation, linking and merging. All steps that are not CUDA-specific are forwarded on to standard host compilers, such as `gcc` on Linux platforms, to preprocess, compile, and link host code for instance. For device code `nvcc` can invoke stages to compile to an intermediate parallel thread execution (PTX) instruction set architecture (ISA) that may be stored in intermediate `.ptx` files. Additionally, `nvcc` can invoke the necessary stages to compile directly to binary executable device code, for each ISA in a specified set of GPU architectures, that may be stored in intermediate `.cubin` files. Options are also provided by `nvcc` to embed the compiled `.ptx` and/or `.cubin` device code into the final code image with the executable host code, when using the Runtime API.

3.6.9 Compute Capability

NVIDIA assigns a compute capability (CC), which is defined by a major and minor revision number, to each CUDA architectural baseline. The major revision number indicates devices with the same core architecture, where 1 is the Tesla architecture, 2 is the Fermi architecture and 3 is the Kepler architecture [32]. The minor revision number indicates incremental improvements in the architecture, which can include new features.

The CC can therefore be used to identify the set of features and capabilities provided by a given CUDA-capable NVIDIA card. Compatibility of compiled binary GPU code across different architectures can also be determined using the CC. Code that is compiled for one major revision is not guaranteed to execute on a device with another major revision. However, binary compatibility is guaranteed within a specified major revision where code compiled for CC $X.y$ is executed on a device with CC $X.z$ where $z \geq y$ [32].

3.6.10 Floating-point Arithmetic

A discussion on the floating-point and IEEE 754 compliance of NVIDIA GPUs [36], supplementary to the CUDA C Programming Guide [32], summarizes the support for different NVIDIA hardware generations as follows. NVIDIA devices of compute capability 1.2 and below only support single-precision and not all operations are fully IEEE 754 compliant in terms of accuracy. Denormal numbers are flushed to zero and the result of some operations will not always result in the floating point value closest to the correct mathematical result. Compute capability 1.3 adds support for double-precision that is fully IEEE 754 compliant, but the single-precision support remains unchanged. With compute capability 2.0 and higher devices, single and double-precision support are fully IEEE 754 compliant. GPU floating-point performance is typically measured in single-precision floating-point operations per second (FLOPS), but is occasionally also quoted explicitly according to double-precision performance figures.

3.6.11 Tools

CUDA provides a variety of complementary development tools that are useful during various stages of development. A quick summary of useful tools that were used during the implementation is presented in this section.

3.6.11.1 Compute command-line profiler

CUDA provides a Compute Command-line Profiler that may be used to gather timing information and measure performance in order to identify optimization opportunities. The profiler can be configured to log counters during kernel execution where the set of counters that are available depends on the CC of the device. However, limited counters are available, especially on first generation hardware, such as the Tesla architecture. Some profiler metrics are also calculated by extrapolating from counters for a single SM. The use of a profiler alone is therefore typically not sufficient for accurate performance measurement.

3.6.11.2 NVIDIA Visual Profiler

An NVIDIA Visual Profiler is provided with CUDA which also performs gathering and measurement of performance data, similar to the Compute Command-line Profiler, but with a with a variety of additional features for automated analysis, visualization and comparison of the data.

3.6.11.3 `cuda-memcheck`

A CUDA memory checking tool, `cuda-memcheck`, which is designed to detect device memory access errors is provided with CUDA. The `cuda-memcheck` tool can typically detect out-of-bounds and misaligned access and memory leaks for the device. The `cuda-memcheck` tool executes the application to detect memory errors in runtime and can operate either in a standalone mode or integrated with a debugger.

3.6.11.4 `cuobjdump`

CUDA provides `cuobjdump`, which is similar to the Linux command-line tool `objdump`, but for CUDA object files created using the `nvcc` toolchain. It provides a variety of options for extracting various forms of device code in a human-readable format.

3.6.11.5 `nvidia-smi`

The NVIDIA System Management Interface, `nvidia-smi` ships with the NVIDIA GPU display drivers and can be used query and modify the GPU device state for management and monitoring purposes. It is primarily targeted at the Tesla and Quadro series cards.

3.7 LIGHTWEIGHT VERSUS HEAVYWEIGHT THREADING MODEL

The NVIDIA CUDA C Best Practices Guide [33] generally recommends a lightweight threading model, where a large number of lightweight threads are used. A lightweight thread commonly produces only a single output element. Arithmetic pipeline latency and memory access latency can be hidden effectively when using this model, granted that sufficient parallelism in the form of independent threads in separate warps, called thread-level parallelism (TLP), is present. Occupancy is a hardware utilization metric for kernels, expressed as a percentage of the maximum TLP for the architecture, based on the number of active warps per SM. Accordingly, it is widely recommended to optimize for high occupancy as a means to achieve latency hiding. The CUDA C Best Practices Guide recommends a minimum of 192 active threads per SM, for CC 1.x devices, and 768 active threads per SM, for CC 2.0 devices, in order to hide latencies associated with register dependencies in particular.

An advanced alternative is a heavyweight threading model, where fewer heavyweight threads are used [25, 37]. The heavyweight threading model relies on independent intra-thread instructions, called instruction-level parallelism (ILP), for latency hiding. This approach is based on the fact that independent instructions within a warp of threads may also overlap in execution and hide latency, enabled by the warp scheduler architecture, without the need for a context switch to a different warp. ILP is typically increased by computing multiple independent outputs per thread, thereby also reducing the number of threads required. Additionally, registers are sometimes used to store a shared working set instead of using shared memory, due to superior memory size and throughput. This approach becomes viable with the heavyweight threading model, where multiple outputs are calculated per thread, as registers only have thread local scope where shared memory is visible by the entire thread block. The heavyweight threading model has been shown to provide equivalent, and even superior, results to the lightweight threading model in certain cases [37], using as little as 64 active threads per SM, for CC 1.x devices, and 192 active threads per SM, for CC 2.0 devices. This is in spite of remarkably low occupancy, which only accounts for TLP.

A loss in generality of the code, and challenges surrounding the management of register allocation and usage are highlighted as potential reasons why a heavyweight threading model is not generally promoted by NVIDIA [25]. Consequently, the more general lightweight threading model is adopted for the GPU implementations in this research.

3.8 CONCLUSION

The high-performance computing benefits of the modern generalized many-core GPGPU architecture have been highlighted. However, these benefits are entwined with the need for a good understanding of the underlying microarchitecture, the deep memory hierarchy, and algorithmic decomposition into coarse-grained and fine-grained parallelism, in order to achieve good performance. A stream-programming paradigm also needs to be adopted to program the highly-parallel device, which presents a learning curve when coming from conventional general-purpose programming paradigms.

Additional intricacies are also present due to remnants from the graphics processing heritage of the architecture, that are not necessarily present on other general-purpose architectures and need to be taken into consideration. It is therefore vital to understand how to optimally utilize the architecture for the radar signal processing application at hand, prior to implementation.

CHAPTER 4

RADAR SIGNAL PROCESSING ON GPUS

4.1 INTRODUCTION

In order to determine the state of the art, a literature survey of existing implementations and previous investigations of software-defined pulse-Doppler radar signal processing on the GPU architecture was conducted. A broad survey on the radar application was conducted initially, followed by a more focused survey on the individual signal processing and radar signal processing functions identified. A summary of the relevant contributions is provided, with respect to implementation approach and techniques, results, and conclusions.

4.2 APPLICATIONS ON GPUS

A number of radar applications were identified, where the GPU architecture is utilized for radar signal processing. Most notably, passive radar and random noise radar applications were found to employ GPU processing, especially for computationally intensive correlation calculations, with substantial performance improvements over other platforms. However, there are substantial differences between these specialized radar applications and the more general and more common monostatic, single-element, coherent pulse-Doppler radar application under consideration. Most notably, neither of these two applications transmit pulsed waveforms, with passive radar systems completely devoid of a dedicated transmitter. To name a few other differences, these systems are bistatic, they use multiple receiver elements and perform different processing functions such as digital beam forming and adaptive filtering. Nonetheless, a summary of the most relevant findings is provided in this section, with respect to GPU implementation of the passive radar and random noise radar applications.

4.2.1 Passive Radar

A passive radar or passive coherent locator (PCL) system in [38] uses NVIDIA GPUs with CUDA and the CUFFT library for signal and data processing to accelerate the application. The PCL processing chain consists primarily of adaptive spatial and temporal filtering, range-Doppler processing that performs cross-correlation and FFTs, and CA-CFAR processing with bearing extraction. All these functions were implemented on a variety of architectures including a 16-core Intel Xeon processor (single-core and 16-core implementations), IBM Cell Broadband Engine (BE), an NVIDIA GeForce GTX 480 GPU and an NVIDIA Tesla C2050 GPU. The range-Doppler processing and CA-CFAR processing were found to be highly parallelizable, with the GPU implementations respectively providing speedups of around 4 times and 5 times over the best performing implementations on other platforms. The algorithms that were used for adaptive filtering were found to be intrinsically serial and performed orders of magnitude worse on the GPUs than the best-in-class implementations. A hybrid approach was followed for the final implementation and evaluation, where the CPU performs the adaptive filtering and the final extraction of CFAR detections using the mask, with the GPU performing the remaining steps.

Real-time software implementation of passive radar is compared using an Intel Core2 Quad 2.4 GHz CPU, NVIDIA GeForce GTX 280 GPU and an IBM Cell BE in a Sony PlayStation 3 (PS3) system in [39]. Two of the most computationally intensive processing functions in the PCL processing chain, namely, adaptive filtering and cross-ambiguity calculation, were implemented on all platforms. The GPU implementations are developed in CUDA using the CUBLAS and CUFFT libraries and the CPU implementations are written in ANSI C with single-core, dual-core, and quad-core support. The GPU implementations achieved speedups of 5 times and 2.5 times over the highest performing (quad-core) CPU implementations, and speedups of 5 times and 4 times over the Cell BE implementations for adaptive filtering and cross-ambiguity, respectively. However, the CPU and PS3 implementations were developed using double-precision float where the GPU implementations all use single-precision float, due to CUDA library limitations.

4.2.2 Random Noise Radar

A random noise radar (RNR) in [40] uses an FFT segmentation method for simultaneous range and velocity estimation, which requires reference signal generation and correlation to be performed. The range and velocity estimation was implemented using the MATLAB parallel computing toolbox for multi-core CPUs and then GPUs. The GPU implementation showed an overall speedup of around 2 times over the CPU implementation on a system with an Intel Xeon W5590 3.33GHz CPU and NVIDIA Tesla C1060 GPU. Similarly, a speedup of around 3 times was achieved on a second system with an Intel Xeon X5667 3.07GHz CPU and NVIDIA Tesla C2070 GPU. The overall achieved speedup was primarily attributed to speedup of the correlation function. The highest performing GPU implementation achieved an overall speedup of more than two orders of magnitude over an older MATLAB implementation that does not make use of the FFT segmentation method and which runs on a CPU.

4.3 SIGNAL PROCESSING PRIMITIVES ON GPUS

A variety of primitive signal processing operations and functions are identified in the breakdown of the typical radar signal processing chain in Section 2.5. Evidence of several existing GPU implementations for each of these signal processing primitives was found, excluding the simple envelope function, where no directly relevant references were found. The majority of the implementations focus on isolated optimization for GPU execution, where a single function is optimized, with the data transfer time between the host and device typically not accounted for. Nonetheless, a number of implementations target real-time operation and specifically optimize the host and device data exchange as well. The most relevant findings, with respect to GPU implementation of the identified primitives, are summarized in this section.

4.3.1 Convolution

Several implementations of 1D [41, 42, 43] and 2D [44, 45] convolution on GPUs both in the frequency domain [41, 43, 44] and time domain [41, 42, 45] exist. The “overlap-save“ method [46] for fast frequency-domain filtering is frequently employed in cases where long signals need to be processed in real-time [43], or where available resources are otherwise exceeded [41], and in some cases even explicitly as a general performance optimization [41]. A pipelined structure, which overlaps data transfer between the host and the GPU with computation on the GPU, is used in a number of real-time audio applications [43, 42] to improve performance. Circular buffer schemes and GPU support for asynchronous copy and compute is typically required to implement such pipelines.

In [41] a comparison of floating-point 1D convolution on GPUs, FPGAs, and multicore CPUs, convolution on the GPU using CUDA performed the best overall for large filter and signal sizes. The frequency-domain implementations outperformed the time-domain implementations, except where either the filter or signal size was small. The overlap-save method was used for signal sizes greater than 8 million elements, due to memory limitations, and also as a further optimization for signal sizes greater than 1 million elements. The comparison used an NVIDIA GeForce GTX 295 GPU card, Intel Xeon Quad-Core W3520 CPU and an Altera Stratix III E260 FPGA. Only one of the GPUs on the GTX 295, which is a dual-GPU card, was used for a fair single-chip comparison.

An investigation [44] aimed at small 2D (16 by 16 to 128 by 128) frequency-domain convolutions on GPUs, found straightforward implementations using CUDA libraries inefficient. Firstly, batching of convolutions is recommended to increase the independent parallelism and data transfer sizes for higher efficiency. Secondly, the tuning of execution configurations and merging of kernels is recommended. An Intel Core i7 920 CPU, NVIDIA GeForce 8800 GTX, NVIDIA Tesla C1060, and NVIDIA GeForce GTX 280 GPUs were used for the investigation.

A real-time 1D frequency-domain convolution is implemented in [43] on an undisclosed NVIDIA GPU using CUDA with the CUFFT library. The implementation uses a pipeline structure to achieve real-time massive convolution of up to 16 audio channels on a single GPU, which traditionally required specialized hardware found in theatres and funfairs. The overlap-save method is employed to split the audio signal into multiple fragments that are packed into a matrix structure, which allows for efficient GPU computation using batched 1D transforms and element-wise multiplication.

Both real-time and offline 1D time-domain convolution is implemented on NVIDIA GPUs for audio processing in [42]. The highest priority optimizations that made significant performance improvements to the offline kernel in particular, were the use of shared memory in conjunction with texture memory, as well as manual unrolling of unconditional inner loops, which provided a speedup of greater than 3 times over the baseline offline kernel implementation. The main difference between the offline and real-time kernel, is that the latter requires a circular buffer check and cannot make use of texture memory for signal data, which is not available upfront to be bound to a texture. The final offline kernel implementation performed up to 2.4 times faster than the final real-time kernel. When compared to the standard MATLAB time-domain convolution function, `conv`, on a CPU, the offline kernel provides a maximum speedup of around 130 times and the real-time kernel provides a maximum speedup of around 50 times. An NVIDIA GeForce GTX 260 GPU with compute capability 1.3 was used with an Intel E5320 CPU for the comparison.

Simple initial tests were also performed on a newer NVIDIA Tesla C2050 with compute capability 2.0. The C2050 showed a speedup of more than 3 times over the GTX 260 for the real-time kernel and was capable of processing an impulse response of 14.1 seconds in real-time instead of only 4.6 seconds. Note that at the 44.1 kHz sampling rate, which was used for this investigation, an impulse response of a few seconds in length already equates to a filter length of several hundred thousand samples. For radar applications the filter length is typically on the order of a few hundred or a few thousand samples, despite the use much higher sampling rates. This is due to the filter length corresponding to the transmit pulse width, which is often on the order of a few hundred microseconds only.

A 2D time-domain image convolution implementation for a separable filter is presented in an NVIDIA CUDA SDK application note [45]. The 2D filter is separated into two consecutive stages of 1D filtering. Global memory coalescing is ensured by explicitly aligning warps with memory boundaries where necessary. The implementation uses shared memory and ensures bank conflict free access by accessing consecutive shared memory addresses with consecutive threads only. Filter coefficients are stored in constant memory and each thread in a half-warp always accesses the same constant address to ensure no penalties due to constant memory bank conflicts. As a final optimization step the inner loop is unrolled, which more than doubles the performance.

4.3.2 Finite Impulse Response (FIR) Filter

A time-domain FIR filter is implemented in [47] on a GeForce 6600 GPU using Cg and compared with an SSE-optimized implementation from the GNU Radio software on a Pentium 4-HT 3.2 GHz CPU. The GPU implementation outperformed the CPU implementation when using a large number of taps greater than 60000, but performed worse for smaller sizes. The input data and filter tap vectors are represented as matrices in two-dimensional textures in the GPU memory, due to constraints on maximum texture dimensions. The matrix representation also allows for a native dot product instruction to be used on the GPU for computation. The FIR filter on the GPU was identified to be computation-bound as texture upload and download time combined was on average around only 10% of the total time, with the rest of the time spent on computation.

A time-domain and frequency-domain complex FIR filter is implemented in [25] on an NVIDIA Quadro FX 4800 GPU using CUDA and compared to a CPU reference implementation on an Intel Core2Duo E8400 CPU. Additionally, for the time-domain FIR filter an OpenCL implementation, nearly identical to the CUDA implementation, is performed on the same GPU hardware and used in the comparison. Minimal implementation details are provided in general, but for the TDFIR the input data is purposely split up between data parallel blocks to exploit the parallelism of the problem. For the CUDA TDFIR implementation a maximum kernel speedup of 100 times was achieved excluding data transfers and up to 37 times when data transfer between host and device is included. The maximum computational performance achieved for the CUDA TDFIR implementation is 182 GFLOPS. Based on visual interpretation of the results in a direct comparison between the CUDA and OpenCL TDFIR implementations, OpenCL performed within 15% of the CUDA implementation. The authors concluded that OpenCL and CUDA performed similarly, but OpenCL was always slower, which they attributed to potential driver quality differences. The FDFIR implementation was not compared to a CPU implementation but achieved a maximum of 160 GFLOPS based on an estimation of the FFT workload by the authors.

A time-domain and frequency-domain FIR filter is implemented in [48] on an NVIDIA GeForce 8800 GTX GPU using CUDA and evaluated against the high-performance embedded computing (HPEC) Challenge [49] benchmark suite. The HPEC suite is based on a set of kernel-level benchmarks for a polymorphous computing architecture (PCA) [7] that was developed for a defence research programme. The equivalent CPU reference implementations of the benchmark suite, written in ANSI C, are evaluated on an Intel Core2 Q6600 CPU. The GPU implementation provided speedups of around 22 times and 150 times for the time-domain and around 12 times and 20 times for the frequency-domain over the reference implementations using the two standard HPEC Challenge datasets, respectively.

The time-domain FIR filter implementation assigns one filter bank to a thread block, uses shared memory to store the filter coefficients and employs an unrolled loop structure for optimal performance. Note that the maximum number of filter coefficients in the datasets is 128, which is still small enough to fit into the shared memory and allow for the approach followed here. However, with larger filter lengths this simplistic, but efficient, scheme would not be viable. The frequency-domain FIR filter implements fast convolution using the CUFFT library to perform the necessary Fourier transforms. The speedup that the frequency-domain FIR filter provides is therefore also greater for large input vectors, which corresponds to the trend for the FFT and IFFT operations that are used.

4.3.3 Corner Turning (CT)

Several corner turning and equivalent matrix transpose implementations exist [25, 48, 50, 51] that emphasize the role of shared memory to ensure coalesced global memory access, shared memory bank conflict resolution, and in some cases a lesser-known partition camping effect. Global memory is also divided into partitions, similar to how shared memory is divided into banks. Partition camping is an effect that occurs when particular partitions of global memory become oversubscribed by memory requests, while other partitions sit idle. The transpose implementations are generally performed out-of-place using floats, which could be used to represent real-valued samples. No explicit evidence of available implementations that support `float2`, `cuFloatComplex` or equivalent data types that could be used to represent complex-valued samples, required for the radar application, was found. An empirical model has been developed [52] for memory-bound GPU kernels under the partition camping effect to predict a performance range only, as the dynamic nature of the problem makes exact performance prediction difficult, and also renders static analysis techniques ineffective.

In [50] a matrix transpose implementation is optimized for CUDA and is now available as a CUDA SDK example. Shared memory was used to achieve optimal global memory coalescing during read and write cycles, as it has no performance penalty for noncontiguous access, unlike global memory. Shared memory bank conflicts were resolved by padding the shared memory buffers with an extra column in order to spread columns across all shared memory banks. Partition camping was addressed using a diagonal thread block ordering scheme, instead of the typical cartesian mapping, effectively spreading the coarse-grained memory access at the thread block level over multiple partitions. It is also mentioned, but not demonstrated, that padding can be used to alleviate partition camping, similar to the solution employed for shared memory bank conflicts. A speedup of greater than 30 times was achieved in terms of effective bandwidth, with the final optimized GPU transpose kernel over a naïve GPU transpose kernel on an NVIDIA GeForce GTX 280 GPU.

A corner turn benchmark is implemented in [25] on an NVIDIA Quadro FX 4800 GPU using CUDA along with a single-core reference implementation on an Intel Core2Duo E8400 CPU. Few implementation details are revealed, however, global memory coalescing, shared memory bank conflicts and partition camping are highlighted as important effects, as with the CUDA SDK transpose implementation. A GPU kernel speedup of up to 260 times was achieved over the CPU implementations, with a speedup of up to 7 times when data transfer between host and device is included.

The corner turn benchmark from HPEC Challenge [49] benchmark suite is implemented in [48] on an NVIDIA GeForce 8800 GTX using CUDA and evaluated against the reference implementations on an Intel Core2 Q6600 CPU. The corner turn was implemented with an out-of-place transpose of floats and global memory coalescing was achieved using shared memory. The GPU transpose implementation provides speedups of around 8 times and 11 times over the CPU implementation for the two standard HPEC Challenge datasets respectively.

Fast CUDA data rearrangement kernels for multidimensional data are developed in [51]. The importance of optimal data access patterns to maintain global memory coalescing and avoid partition camping effects are highlighted and solutions very similar to the CUDA SDK matrix transpose implementation are presented. Shared memory is used extensively as a user managed cache to improve performance and to allow access patterns that would otherwise be uncoalesced when performed directly on global memory. A diagonal block ordering scheme is again employed to counter the partition camping effect.

A model is developed in [52] to predict the worst-case and best-case bounds of execution time under the partition camping effect for memory-bound kernels, such as the matrix transpose, on NVIDIA GPUs. Partition camping occurs at a macro level between all active warps for all active blocks that are scheduled and executed independently as SMs become available and is therefore undetectable with static analysis, unlike shared memory bank conflicts. Hence, the model was developed to predict only a performance range. The CUDA SDK matrix transpose example forms part of the application suite that was used to test the model, but no attempt was made to further analyze or improve it as this was not the stated goal. A spreadsheet-based tool, *CampProf* was developed as a front-end to the performance model, as the standard CUDA tools cannot detect the partition camping effect. However, the *CampProf* tool and the model coefficients that were obtained from empirical data do not appear to be publically available. The study showed that partition camping can result in up to a sevenfold degradation in performance.

4.3.4 Discrete Fourier Transform (DFT)

A variety of fast Fourier transform (FFT) [53] implementations of the discrete Fourier transform (DFT) have been developed for GPUs and evaluated [23, 54, 55, 56, 57]. The implementations generally support 1D and 2D transforms for both real and complex data where an interleaved I and Q scheme is typically used for the latter. Implementations that make use of texture memory [55, 56] do not support in-place transforms, as textures cannot have simultaneous read and write access from within GPU kernels or programs. However, in-place transforms can be supported with implementations that use shared memory [23, 54, 57]. In-place transforms can be beneficial for very large transform sizes due to memory constraints, as out-of-place transforms typically require twice as much memory to be allocated. Benchmark results that are obtained generally exclude data transfer times between the host and device over the PCIe bus. They also exclude FFT planning time for plan generation before execution, which is based on the FFTW model. The FFT performance that is generally achieved on GPUs is low for short FFT lengths, unless many FFTs are batched and performed together. The performance achieved on GPUs for longer FFT lengths is much higher and typically greatly exceeds what is achieved on other general-purpose processors such as CPUs.

An NVIDIA GeForce FX 5800 GPU, which predates the generalized CUDA architecture, is used in [56] to implement an FFT algorithm in Cg and OpenGL. The FFT was implemented using trivial vertex programs, which were only used to limit each fragment to a single pixel, followed by the execution of fragment programs for each pixel to perform the processing. The fragment program needs to exhibit slightly different behaviors based on the position of the pixel. However, branching in the fragment programs is not supported on the older graphics hardware. The implementation therefore loads and executes different fragment programs for different sections of the texture, although this introduces additional overhead. The GPU implementation was compared to an implementation that uses the FFTW library on a 1.7GHz Intel Xeon CPU. The GPU implementation performed worse, taking 2.7 seconds, compared to the CPU implementation which took 0.625 seconds or 2 seconds under some conditions, although the authors considered their GPU implementation comparable to the highly optimized FFTW implementation.

A collection of FFT algorithms that efficiently support arbitrary FFT lengths that use the complex input format are implemented in [54] using CUDA on a number of NVIDIA GPUs, of which the NVIDIA GeForce GTX 280 is the most capable. The GPU implementations are compared to the CUFFT library version 1.1 on the same GPU and the Intel Math Kernel Library (MKL) version 10.2 on an Intel QX9650 3GHz quad-core CPU. The GPU implementations provided typical performance improvements of 2 times to 4 times over CUFFT and 8 times to 40 times over MKL for large sizes. For the prime length cases that were evaluated the CUFFT library was very slow, as it uses a direct DFT implementation in these cases. Scaling tests were also performed with a global memory FFT implementation, as well as a hierarchical FFT implementation that uses shared memory on the GTX 280 GPU. The GPU FFT performance in GFLOPS scales linearly with the core clock rate for all but the smallest sizes. Performance also scales with increasing memory clock rate, but the FFT becomes compute bound for higher memory clock rates, particularly with the shared memory implementation for larger sizes.

The FFT is implemented using the DirectX 9 API on an NVIDIA GeForce 8800 GTX GPU and ATI XT1900 GPU in [55]. Arbitrary length FFTs are supported, provided the total number of elements can fit into a single texture. A ”four-step“ algorithm is used to wrap long 1D FFTs into the 2D texture buffer used for computation, in order to avoid exceeding the maximum texture dimensions in any one dimension. The computation of real-valued FFT is optimized by packing the real sequence into a complex sequence and exploiting the symmetry of this type of transform. The GPU implementations running on the NVIDIA and ATI GPUs were compared to CUFFT on the NVIDIA GPU and Intel MKL on an Intel Core 2 Duo 2.6GHz E6600 CPU. The GPU implementations on the NVIDIA GPU consistently outperformed the MKL implementation and the CUFFT implementation for real-valued power-of-two FFTs. The CUFFT implementation outperformed the GPU implementations in all cases for complex-valued power-of-two FFTs, which was attributed to the fact that CUFFT uses shared memory. The GPU implementations on the older, less expensive ATI GPU for power-of-two sizes performed much lower than on the NVIDIA GPU, but they were still competitive when compared to MKL. For non-power-of-two sizes the results are generally quite erratic, but show similar trends. The GPU FFT implementations were found to be mostly bandwidth limited as performance improved by less than 10% when computations were removed entirely from the GPU fragment programs, leaving only the memory read and write operations.

The CUFFT library performance is investigated for potential future application in a radar pulse compression application on an NVIDIA GeForce 8800 Ultra GPU in [57]. A comparison is made with the FFTW library on two dual-core AMD Opteron CPUs at 2.6GHz. Unlike most cases, the GPU benchmark that was used includes the data transfer time between the host and the device. Page-locked host memory was used to optimize data transfer speed. Short 1D FFTs, which have low total computational intensity compared to total data transfer size, is highly inefficient on the GPU compared to the CPU, especially when the data transfer time is taken into consideration. For a 1D FFT the GPU implementation provides a speedup only for large power-of-two FFT lengths up to a maximum speedup of 2.5x, where the CPU is superior for smaller and non-power-of-two cases. For the 2D FFT the GPU implementation shows a consistent advantage with a maximum speedup of 6x, which is attributed to the fact that the number of operations per data element is greater than with the 1D case. The results shows that the pulse compression application can be expected to have a speedup at least equivalent to the 2D FFT case, as the standard convolution operation that is used has a similar operation count.

The latest CUFFT library [23] version 5.5 now supports 1D, 2D and 3D transforms in single or batched modes for real and complex sequences. Both in-place and out-of-place transforms are supported for FFT lengths up to 128 million elements in single precision. The library now also supports an $O(n \log n)$ algorithm for every input length, where certain cases, such as prime lengths, traditionally reverted to a direct DFT computation of $O(N^2)$ complexity.

4.3.5 Cell-Averaging Constant False-Alarm Rate (CA-CFAR)

A variety of software implementations of the CA-CFAR algorithm on GPUs [1, 38, 48] and CPUs [39, 58] exist. The CA-CFAR algorithm allows for embarrassingly parallel implementation and therefore GPU implementations generally provide significant speedups over CPU implementations when considering solely the kernel execution time, especially for larger datasets. The most common approach to GPU implementation of the CA-CFAR, which will be referred to as the naïve technique, favors parallelism but performs a considerable amount of redundant computation. A sliding window technique [59] is popular for software implementation in general as it reduces redundant computation, although it also introduces additional data dependencies that lower parallelism.

The HPEC Challenge [49] benchmark suite provides a 1D CA-CFAR algorithm and reference implementation that operates in the range dimension only and uses averaging to compute the local noise estimate. Four datasets are also provided, that contain beam, range and Doppler dimensions. The beam dimension adds an additional processing dimension, which is not specifically considered for this research.

In [48] a CFAR is implemented on a GPU and compared to the HPEC Challenge [49] benchmark suite CFAR reference implementation. In [48] the CA-CFAR GPU implementation on an NVIDIA GeForce 8800 GTX using CUDA provided speedups of between 2.3 times and 166 times over the reference implementation on an Intel Core2 Q6600 CPU using the standard HPEC Challenge datasets. The performance of the CA-CFAR GPU implementation was found to be bound by memory bandwidth and not computation.

A 1D CA-CFAR is also implemented in [25] and evaluated on an NVIDIA Quadro FX 4800 GPU using CUDA and compared to a CPU reference implementation on an Intel Core2Duo E8400 CPU using the HPEC Challenge [49] benchmark suite datasets. Implementation details are not revealed, but a maximum kernel speedup of 30 times was achieved with the GPU implementation over the CPU implementation, with a maximum speedup of 1.1 times only, when data transfer between host and device is included. It is recommended that the CFAR not be used as a standalone kernel, as the data transfer cost is high relative to the computation that is performed, due to a low overall arithmetic intensity (AI) for the algorithm.

The CA-CFAR algorithm is implemented in [1] on a GPU using the naïve and sliding window techniques, as well as a new summed-area table (SAT) technique. The naïve and SAT techniques were found to be most efficient for GPU implementation. The former is preferable under typical conditions and the latter is preferable with large CFAR window sizes, due to a constant-time summation lookup scheme that uses summed-area tables. The naïve GPU implementation on an NVIDIA Tesla C1060 outperformed all other GPU implementations and multicore CPU implementations on dual Intel Xeon X5355 quad-core CPUs in terms of kernel throughput for input sizes greater than 8192 cells. The use of shared memory and the segmentation of summation into separate row and column stages were key factors in achieving high performance.

Also, as described in Section 4.2.1, the CA-CFAR implementation for a PCL system on a GPU [38] provided a speedup of around 5 times over a multicore CPU implementation.

4.4 CONCLUSION

In general it is found that GPU implementations provide large performance improvements over conventional general-purpose architectures with sufficient data-parallelism and arithmetic intensity (AI). Long input lengths and large batches of independent work items are generally required to achieve peak performance. It is also found that the identified primitive operations provide sufficiently parallel structures, in principle, for efficient GPU implementation. No directly relevant references to the envelope or rectifier function on GPUs were found, which is attributed to the relative simplicity of this function, which is perhaps incorporated into larger functions. Some algorithms, such as the CFAR, generally provide a low AI, which makes it unattractive for standalone implementation when considering data transfer times between the host and device. In such cases it is generally recommended that a function not be implemented in isolation, but instead as part of a chain of functions.

A number of real-time audio applications use a pipelined structure to overlap data transfer and execution efficiently to improve throughput by hiding latency. Such a pipelined structure is also an important consideration for the radar signal processing chain, which is also a real-time application. Examples of radar signal processing on GPUs exist for RNR and PCL applications with a significant speedup over alternative approaches. However, there are substantial differences between these systems and conventional pulse-Doppler radar that precludes direct comparison. No evidence was found of a complete pulse-Doppler radar signal processing chain implementation on the GPU, which is considered a research gap. It was further found that the majority of contributions used the CUDA parallel framework for implementation.

CHAPTER 5

DESIGN AND IMPLEMENTATION

5.1 INTRODUCTION

The design and implementation approach, as well as core steps that were followed are described in this section. The target hardware platform that was used for the implementation is described initially. A set of high-level and low-level metrics are defined that are used to qualify and quantify performance of the implementation at a system and architectural level respectively. A GPU benchmarking framework (GBF) that was developed and used extensively during implementation is also described. Finally, the implementation details of the radar signal processing algorithms and the radar signal processing chain are discussed.

5.2 HARDWARE PLATFORM

A host system with two discrete NVIDIA graphics cards was used for the implementation and to generate experimental results. Summaries of the host system specifications and GPU specifications are provided in Table 5.1 and Table 5.2, respectively. The NVIDIA Tesla C1060 [60], which is a GPGPU card that uses the GT200 architecture, was the main focus of the research as it is most representative of a device tailored for GPGPU and it therefore provides additional features that are disabled or unavailable in mainstream graphics cards. The algorithms and the radar signal processing chain were implemented and optimized for the C1060 and the architectural features that it provides. An NVIDIA GeForce GTX 480, which is a newer mainstream graphics card aimed primarily at computer gaming, was also available in the host system. The GTX 480 was therefore also used, but only for selective testing and inspection of results without any additional architecture-specific optimization or use of new features that it provides.

Table 5.1: Summary of specifications for host system.

CPU	#Cores	Memory	OS	CUDA	NVIDIA Driver	gcc
Dual Intel Xeon X5550 @2.67 GHz	4 (x2)	50 GB	Ubuntu 12.04	4.2.9	304.88	4.6.3

The results generated using these two cards from different GPU families can still be reasonably compared since single-precision floating-point computation is used exclusively for this implementation. Enhanced double-precision performance is reserved for the GPGPU cards as a selling point due to the attractiveness of additional precision for many scientific GPGPU applications. Where the total device memory size plays a role, the Tesla C1060 will perform better as it provides more than double that of the GTX 480. Nonetheless, the GTX 480 card uses a newer generation GPU architecture and is at the top of the range for its generation. The newer GTX 480 is therefore expected to outperform the older Tesla C1060 card overall.

Table 5.2: Summary of specifications for NVIDIA GPUs in host system.

GPU	#Cores	Memory	Peak BW	Peak GFLOPS	CC	Arch.	Released
Tesla C1060	240	4 GiB	102 GB/s	933	1.3	(Tesla) GT200	Aug '08
GeForce GTX 480	480	1.5 GiB	177 GB/s	1345	2.0	(Fermi) GF100	Mar '10

Both cards support PCIe 2.0 $\times16$ for the host interface, which provides a maximum theoretical bi-directional bandwidth of 8 GB/s. However, the ECC memory feature, which is generally supported by the Tesla (GPGPU) series cards starting from the Fermi architecture onwards (CC 2.0+), is not available on either of the two cards that were used. ECC is a highly desirable feature for large-scale HPC cluster computing, since it enhances data integrity. With ECC memory, single-bit errors can be corrected and double-bit errors can be reported. However, enabling ECC also reduces usable memory by 12.5% and also reduces memory bandwidth for uncoalesced memory transfers beyond the usual penalties [9].

5.3 METRICS

The metrics that were used to characterize the GPU radar signal processing implementation on various levels are presented and described in this section.

5.3.1 High-level

High-level metrics that are related to system-level performance of the GPU radar signal processing chain are presented in this section.

5.3.1.1 Latency

Figure 5.1 shows an illustration of the different latency metrics that are required to characterize kernel-level, function-level, and chain-level performance. An advanced case is used where pipelining leads to overlap between certain operations for clear illustration of the metric definitions and their purpose. In simple sequential cases some of the metrics that have been defined become redundant. The latency metrics are defined in Table 5.3 and described in further detail in the rest of this section.

Kernel latency is defined as the time required from start to finish to execute one or more consecutive kernels on the GPU for a particular burst, excluding any host to device (HtoD) or device to host (DtoH) data transfers over the PCIe bus. It therefore represents exclusively the GPU computation time required to perform a set of operations once the data already resides in the GPU device memory. The kernels that are executed depend on the experiment. This metric is important to consider for a chain of functions where intermediate data may remain on the GPU without the need for transferring data back to the CPU after each step. Additionally, under certain conditions, data transfers may be hidden using pipelining, which leaves the kernel latency as the main performance indicator.

Burst latency is defined as the time required from start to finish to execute one or more consecutive kernels on the GPU for a particular burst, including HtoD and DtoH data transfers over the PCIe bus. The burst latency therefore includes the kernel latency as well as any HtoD and DtoH transfers over the PCIe bus, in order to transfer data between the GPU and host memory. The burst latency is an important metric as it represents the achieved per-burst system latency.

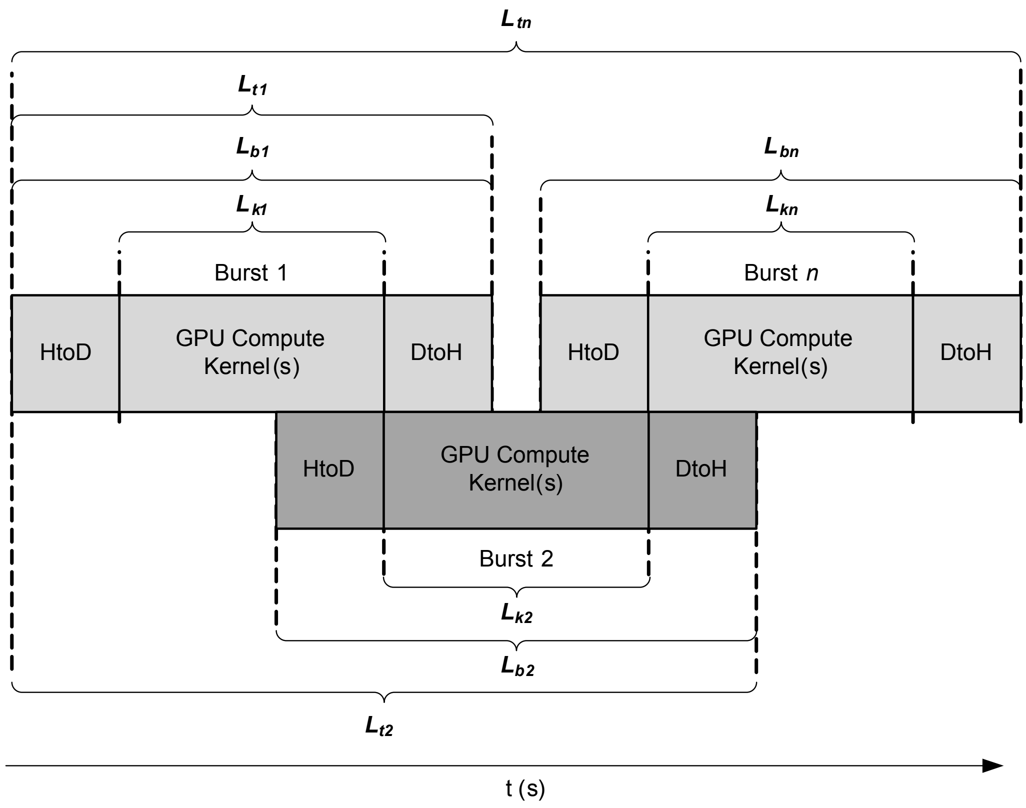

Figure 5.1: Latency metrics for a series of n consecutive bursts for an advanced case that uses pipelining, selected in order to clearly distinguish differences between similar metrics.

Table 5.3: Definition of high-level metrics.

Metric	Notes	Expression	Definition	Unit
Kernel latency	Burst i	L_{ki}	(fig. 5.1)	seconds
Average kernel latency	For n bursts	$\bar{L}_{kn}$	$\sum_{i=1}^{n} \frac{L_{ki}}{n}$	seconds
Burst latency	Burst i	L_{bi}	(fig. 5.1)	seconds
Average burst latency	For n bursts	$\bar{L}_{bn}$	$\sum_{i=1}^{n} \frac{L_{bi}}{n}$	seconds
Total latency	For n bursts	L_{tn}	(fig. 5.1)	seconds
Signal processing workload	F fast-time samples S slow-time samples	W_s	$F.S$	samples
Kernel throughput	For n bursts	T_{kn}	$\lim_{n\to\infty} \frac{n.W_s}{\sum_{i=1}^{n} L_{ki}}$	samples/s
Total throughput	For n bursts	T_{tn}	$\lim_{n\to\infty} \frac{n.W_s}{L_{tn}}$	samples/s

Total latency is defined as the time required from start to finish to execute one or more consecutive kernels on the GPU per burst for a series of consecutive bursts, including HtoD and DtoH data transfers over the PCIe bus. In advanced cases where overlap occurs, the total latency does not equal the sum of the burst latencies for all bursts, as illustrated in Figure 5.1. The total latency is useful for calculation of throughput rates and as a measurement of overall run time.

5.3.1.2 Throughput

The throughput metrics are defined in Table 5.3 and described in further detail in the rest of this section.

Signal processing workload is defined as the total number of samples that need to be processed in a burst with F fast-time (range) samples by S slow-time (pulse / Doppler) samples for the high-level metrics. The workload is therefore independent of the data storage format and other implementation details, which is useful for high-level throughput metrics that can be easily related to system performance.

Kernel throughput is defined as the rate at which the signal processing workload for consecutive bursts is processed on an infinite timeline based on the kernel latency. For practical purposes a set of 3 or more consecutive bursts are processed and considered sufficient to approximate the kernel throughput. As with the kernel latency, the kernel throughput is the main performance indicator in cases where data transfer can be minimized and/or hidden using pipelining techniques.

Total throughput is defined as the rate at which the signal processing workload for consecutive bursts is processed on an infinite timeline based on the total latency. For practical purposes a set of 3 or more consecutive bursts are processed and considered sufficient to approximate the total throughput. The total throughput is an important metric as it represents the achieved overall system throughput.

5.3.2 Low-level

Low-level metrics that are related to kernel-level performance and optimization for the GPU architecture are presented in this section.

5.3.2.1 Occupancy

Occupancy is a metric which determines how effectively the hardware is kept busy by relating to the number of active warps available for the warp scheduler to choose from when performing a zero-latency context switch, as described in Section 3.6.2. The maximum number of warps per SM is determined by physical limits of the architecture as outlined in Table 5.4 for CC 1.3 cards, which is applicable to the Tesla C1060 GPU. Occupancy is defined as the ratio of the number of active warps per SM to the maximum number of warps supported per SM [33], as shown in Table 5.5.

Table 5.4: Core architectural limits coupled to occupancy for CC 1.3 GPUs.

Quantity	Upper limit
#Threads/ block	512
#Threads/ warp	32
#Threads/ SM	1024
#Warps/ SM	32
#Blocks/ SM	8
#32-bit registers/ SM	16384
Shared memory/ SM (bytes)	16384

The number of active warps per SM is therefore affected by the kernel launch configuration, register count and shared memory usage. A CUDA GPU occupancy calculator spreadsheet is available as part of the CUDA SDK, which can be used to analyze kernel occupancy by employing kernel resource usage statistics. Some of these statistics are parameters that are explicitly specified by the programmer, such as the number of threads per block and dynamically allocated shared memory. The other implicit statistics, such as the register count and automatically allocated shared memory, can be gathered using tools such as the NVIDIA CUDA profilers or by compiling the code to the intermediate PTX format that contains this information.

Optimizing occupancy is one of the key factors to obtain high performance, when using the lightweight threading model, by keeping the SMs as busy as possible. However, higher occupancy does not necessarily imply higher performance for all kernels. It is primarily effective at keeping the hardware busy by avoiding idle compute time while waiting for memory transfers to complete.

Kernels that are heavily bound by computation may show little or no improvement with increased occupancy beyond a certain point.

5.3.2.2 Throughput

Several low-level metrics that are related to throughput are defined in Table 5.5 and described in the rest of this section.

Device memory workload is defined at the total number of bytes that are read from and written to device memory during a single kernel invocation. This metric is generalized to device memory, but can be applied to any of the CUDA device memory abstractions as listed in Section 3.6.3.2, although certain restrictions apply to cases where the memory is cached.

Table 5.5: Definition of low-level metrics (single kernel).

Metric	Notes	Expression	Definition	Unit
Occupancy	[Derived]	occupancy	$\frac{\text{active warps}/SM}{\text{maximum warps}/SM}$ [33]	N/A
Device memory workload	B_r bytes read B_w bytes written [Derived]	W_{dmem}	$B_r + B_w$	bytes
Computational workload	[Derived]	W_c	$\sum FLOPs$	FLOPs
Arithmetic intensity	[Derived]	AI	$\frac{W_c}{W_{dmem}}$	FLOPs/byte
Peak theoretical computational throughput	(Tesla C1060) [Derived]	$T_{c(peak)}$	$\min(AI.102 \times 10^9, 933 \times 10^9)$ $\min(AI.BW_{peak}, FLOPS_{peak})$	FLOPS
Computational throughput	[Measured + Derived]	T_c	$\frac{W_c}{L_k}$	FLOPS
Effective bandwidth	[Measured + Derived]	T_{eb}	$\frac{W_{dmem}}{L_k}$ [33]	bytes/s

Computational workload is defined as the total number of single-precision floating-point operations that need to be performed in a single kernel invocation. A multiply-add (MAD) instruction, which combines multiplication and addition in a single operation, is counted as two FLOPs. Transcendental functions are counted as a single FLOP each.

Arithmetic intensity (AI) is defined as the ratio of the computational workload to the device memory workload for a single kernel invocation. The AI is an important factor in determining whether a kernel implementation is limited by either compute or memory throughput in order to determine further optimization steps that may be followed. Low AI kernels are more likely to be limited by memory throughput, with high AI kernels more likely to be limited by computation or, effectively, instruction throughput.

Peak theoretical computational throughput is defined as the maximum computational throughput that a kernel with a given arithmetic intensity (AI) is expected to achieve under ideal conditions on a specified device. The calculation assumes that the kernel achieves either the peak theoretical bandwidth or peak theoretical computational performance for the device.

Computational throughput is defined as the rate at which the computational workload is processed by a kernel based on measured kernel latency.

Effective bandwidth is defined as the combined rate at which a kernel reads and writes data on device memory. The effective bandwidth is determined using knowledge of how a kernel accesses memory and by measuring the kernel latency accurately. The effective bandwidth that is calculated using measured values can then be compared to the theoretical peak bandwidth of the target GPU device in order to determine whether additional optimization of the kernel may yield improved results. If the effective bandwidth is much lower than the theoretical bandwidth of the device then the implementation details are likely to be limiting the bandwidth and further optimization is warranted [33].

5.4 GPU BENCHMARKING FRAMEWORK (GBF)

A lightweight software framework, called the GPU benchmarking framework (GBF), was developed to aid in the benchmarking of all GPU codes, as significant empirical validation is often required to achieve GPU implementations that perform well. The aim of the GBF is to standardize and coordinate the benchmarking process to ensure that results are reliable and reproducible by providing common functions and by largely automating functional verification and benchmark data gathering. Rudimentary processing and visualization of benchmark data is also provided, but detailed analysis and interpretation of the data is left for the user.

The GBF provides common benchmarking functionality and infrastructure in the form of a set of C/C++ and CUDA C source files, utility functions, and Makefiles, that are used directly by the GPU implementations; benchmarking scripts that drive the benchmarking process itself; and MATLAB scripts that are employed to analyze benchmarking results. A particular experiment is defined by incorporating all these GBF elements into an experiment-specific configuration built on top of the common building blocks.

The outer steps of the GBF benchmarking process are performed by scripts as shown in Figure 5.2. The online benchmarking process is driven by a Linux BASH [61] script, which performs all the steps necessary to produce a full set of benchmarking results for an experiment. A common script provides common functions that are called from a high-level experiment-specific script. Functions are provided to automatically invoke standard NVIDIA tools during the process that are useful for logging hardware details and performing memory access error checking, ancillary profiling, and disassembling of the code. A typical experiment script defines multiple code variants, multiple input parameter sets and multiple timed runs.

Code variants are implemented using conditional compilation blocks supported by the underlying GPU code that embodies the experiment and is generally used to invoke different variants of the same core function for comparison. Variants may differ algorithmically or with respect to specific kernel implementation details under investigation. An input parameter set is passed to a compiled variant when it is executed and used during runtime to determine algorithmic and other general behavior. Typical input parameters include algorithmic parameters, burst dimensions, filenames and additional flags used by the GBF functions to enable or disable certain options.

Parameters can easily be swept across a range by specifying multiple input parameter sets for an experiment. Multiple timed runs are performed for the same variant and input parameter set in order to obtain benchmark timing results that are statistically significant. Standard MATLAB scripts are provided to parse, check, and consolidate benchmark timing logs, with detailed analysis and plotting functions generally being experiment-specific.

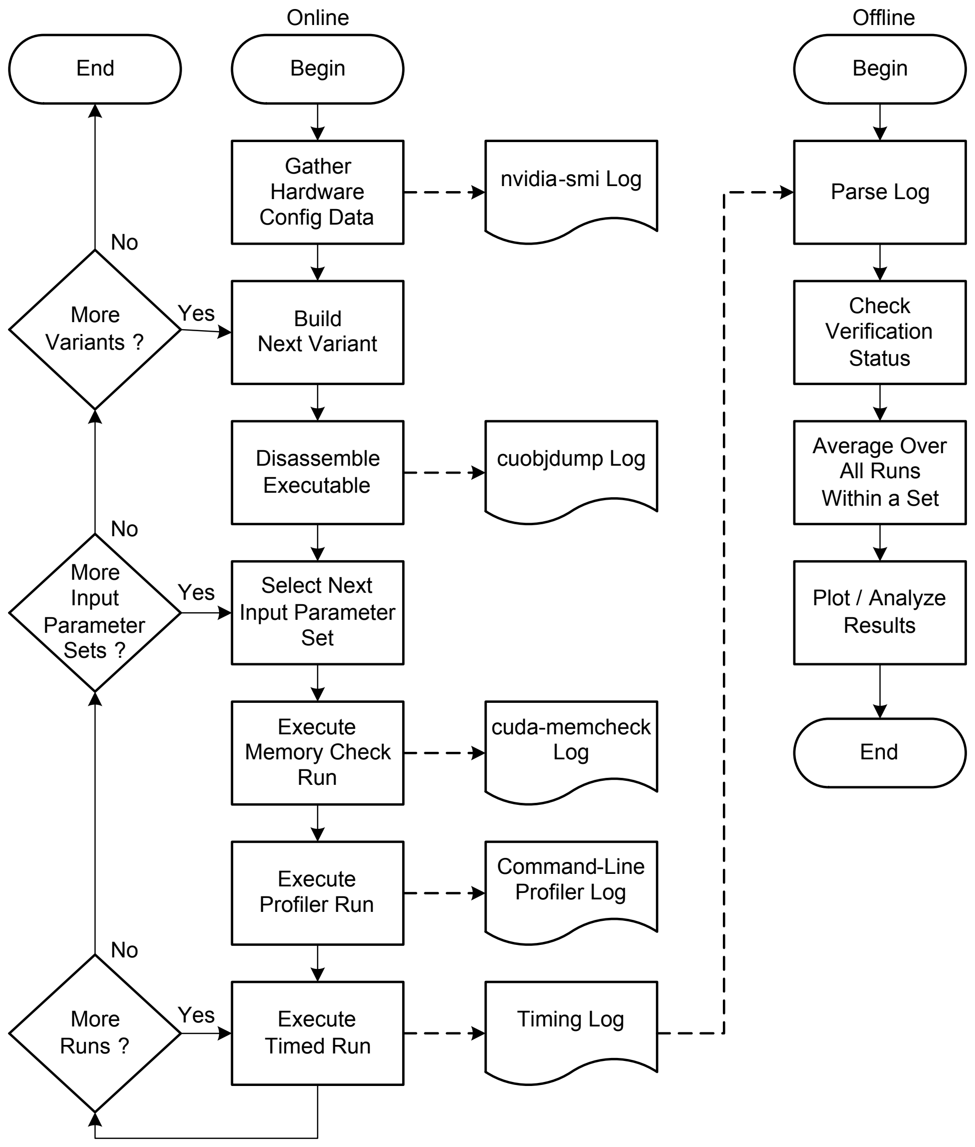

Figure 5.2: Typical GBF benchmarking flow showing outer steps with the online process driven by a BASH script during benchmarking and the offline process driven by a MATLAB script to analyze benchmarking results.

The inner steps of the GBF benchmarking process are performed during an individual timed run, as shown in Figure 5.3. A single timed run corresponds to an invocation of the target executable for a particular compiled variant with a particular input parameter set. After the basic CUDA initialization and buffer allocation, burst data is generated according to burst dimensions and number of bursts that are specified as input parameters. The multi-burst mode supported by the GBF is typically only required when considering pipelining techniques where data transfer and execution are overlapped.

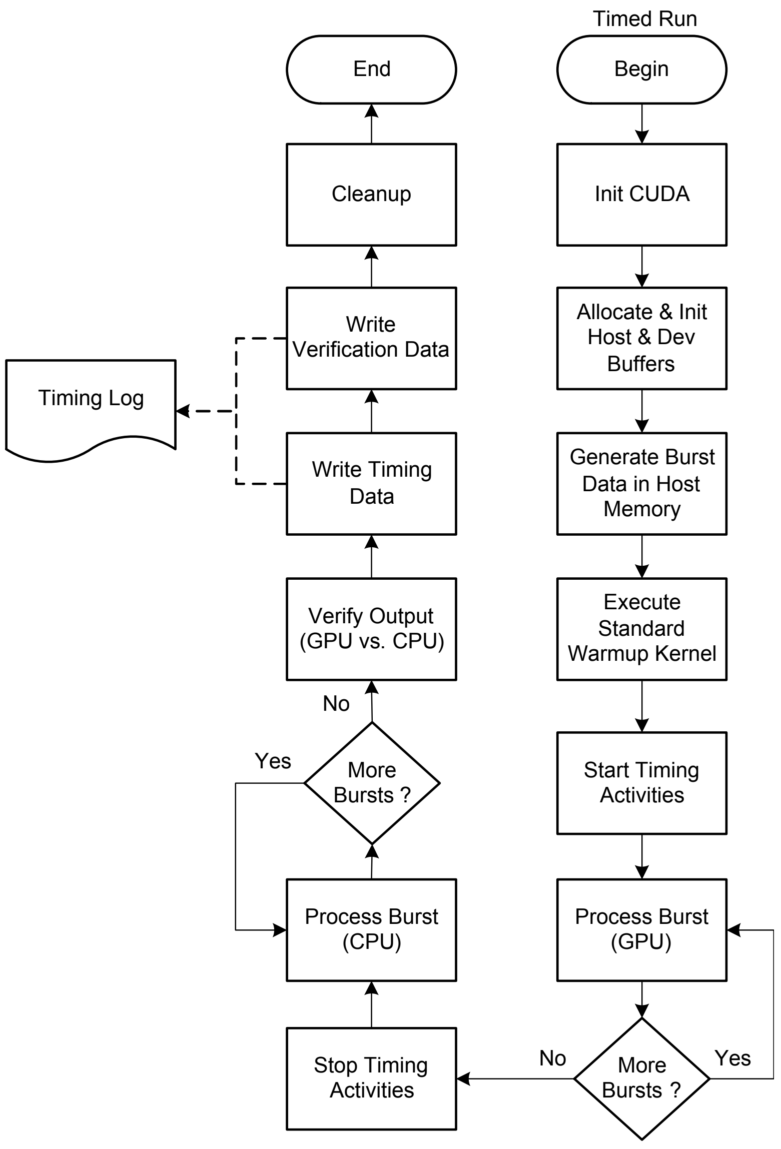

Figure 5.3: Typical GBF benchmarking flow showing inner steps that are performed for a single timed run of the target executable that contains host and device code.

Accordingly, only a single burst is often used. A standard warmup kernel, that is invoked from a self-contained function that also performs the necessary memory management, is also provided. As a standard practice a warmup kernel is typically called prior to any GPU performance measurements when using the CUDA Runtime API, which uses lazy initialization, to ensure that any API startup costs are not included in measurements. Performance measurements are then made by timing GPU memory transfer and kernel operations during execution using CUDA events.

The CUDA events API provides functions to record timestamps using the GPU clock with a resolution of approximately 0.5 μs [33], with additional helper functions to calculate elapsed time. With asynchronous operations, as described in Section 3.6.7, special care is required when using CUDA events to ensure that concurrency is not disrupted and that timing results are valid. CPU reference implementations are typically performed for each experiment or algorithm to allow GPU results to be validated automatically. The GBF provides utility functions that perform functional verification by comparing the output of GPU implementations with the output generated by the CPU reference implementations. Each experiment defines a high-level Makefile that includes a common Makefile, which in turn defines a standard compilation process using the `nvcc` and `g++` toolchains with typical compilation flags and linking options to a number of common libraries. Each experiment also implements its own `main` function, based on the standard process described here.

5.5 IMPLEMENTATION

The implementation details for each of the radar signal processing algorithms that were implemented are described in this section. Each implementation is also analyzed abstractly in this section prior to experimentation. Finally the implementation of the complete processing chain is discussed.

5.5.1 Digital Pulse Compression (DPC)

The DPC function was implemented as a 1D complex FIR filter using time-domain (TDFIR) and frequency-domain (FDFIR) implementation approaches. A burst with F fast-time range samples and S slow-time pulses is filtered using S 1D filter banks with input length F and a 1D complex impulse response with length N, that represent the matched filter coefficients for the transmit pulse waveform. The DPC variants and kernels that were implemented are shown in Figure 5.4.

5.5.1.1 TDFIR

The TDFIR is implemented as a single kernel that performs a batch of S 1D TDFIR filters with length F using time-domain convolution. Batching of convolutions is recommended to increase the independent parallelism and data transfer sizes to increase efficiency as discussed in Section 4.3.1. Each pulse is filtered independently by separate thread blocks. Each thread block contains 256 threads and each thread calculates 1 filter output sample.

Multiple thread blocks are used per filter where the input length, or number of range bins, exceeds 256 in order to ensure high parallelism within each filter. This is contrary to certain implementation approaches where an entire filter is mapped to a single thread block, as discussed in Section 4.3.2. The chosen mapping is further aimed at maximizing the potential for data reuse and spatial locality within a thread block, as the data reuse within each filter is high. There is no data reuse between filters as they are independent, apart from the matched filter coefficients that are shared between all filters. Texture memory (TMEM), shared memory (SMEM), and constant memory (CMEM) are frequently used in addition to global memory (GMEM), in order to improve the memory throughput for FIR filter implementations, as discussed in Section 4.3.1. When employed, CMEM is typically used to store filter coefficients with TMEM and SMEM used in various combinations for coefficients and/or input data.

For the TDFIR implementation the filter coefficients are stored in constant memory (CMEM) in order to make use of the constant cache, which greatly reduces the device memory workload required for each thread to read the coefficients. CMEM is optimized to broadcast read-only data to multiple

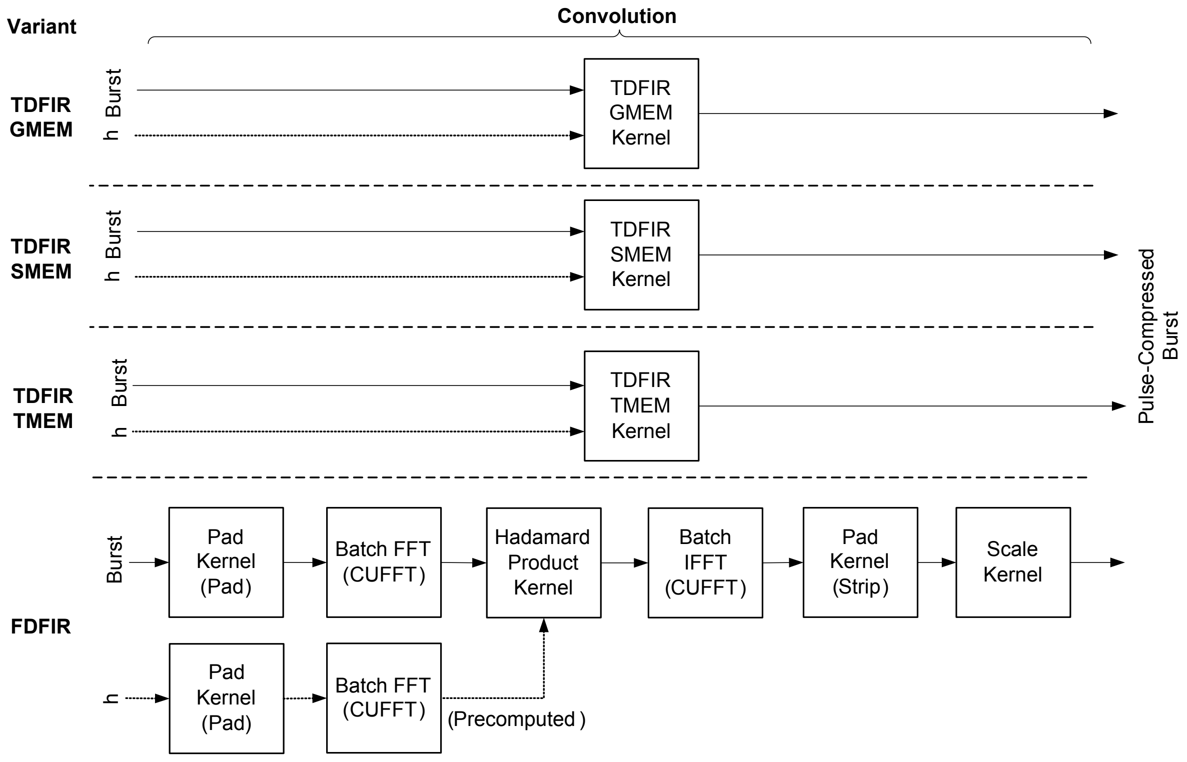

Figure 5.4: DPC implementation showing all variants and kernels that were implemented with the CUFFT library indicated where used to perform certain functions for one variant.

threads, as discussed in Section 3.6.3.2, hence the kernels are designed to access the same filter coefficient within a warp when performing the convolution to make efficient use of this mechanism. Note that with the current implementation a single filter is applied to all pulses using a single coefficient set, which implies that the same pulse waveform is used throughout the burst, which is not always the case for advanced radar systems. A total of 64 KiB CMEM is available, which allows for a maximum of 8192 complex filter samples to be stored if this resource is not used for any other purpose by the kernel. Therefore, if numerous separate filter coefficients sets are required for a single burst, an alternative to using CMEM, such as TMEM or SMEM needs to be sought.

Three variants that use the TDFIR approach were implemented that use GMEM, TMEM and SMEM respectively to read the input data, with all variants using CMEM for the filter coefficients and GMEM to write the output data. The GMEM variant suffers from significant redundant reads of the same input data for multiple threads and poor coalescing due to misaligned reads. The latter is caused by the access pattern where each thread reads N adjacent input samples in turn and accordingly the data access for the warp as a whole is not optimally aligned in all cases. The misalignment effects that are shown for the global memory microbenchmark in Figure 6.3 are, therefore, expected for the GMEM kernel.

The TMEM variant is expected to improve on the GMEM variant as texture memory is cached and, furthermore, it does not impose the same coalescing requirements as global memory, as discussed in Section 3.6.3.2. Significant texture cache hits are expected due to the spatial locality and level of data reuse within each filter due in turn to the chosen thread block mapping.

For the SMEM variant a shared memory buffer is declared with storage for one complex sample per thread for a total of 256 locations that is used to selectively cache input data blocks. As with the other variants each thread calculates the convolution output for a single filter output sample. However, with the SMEM variant multiple stages of loading and processing are required, due to the limited size of the SMEM. Each output sample requires N adjacent input samples that extend into adjacent data blocks. Each thread reads a single complex input sample from GMEM into SMEM per stage. Each thread then iterates through the relevant filter coefficients and input samples in the input block that is cached in SMEM and updates a local running sum that is stored in a register with the convolution calculation result. If additional input data is required to calculate the final convolution result for any of the output samples that correspond to the current thread block, another stage is initiated.

The multi-stage approach is implemented as an outer loop within the SMEM kernel with the necessary `__syncthreads()` block-wide synchronization primitive used after loading to SMEM and again after data access to SMEM for the current stage is complete, as the same SMEM buffer is reused. After all stages are complete each thread writes the corresponding filter output sample to global memory. The SMEM variant is also expected to improve on the GMEM variant by reducing the device memory workload significantly, due to the explicit caching. Furthermore, the SMEM variant does not suffer from poor global memory coalescing that is encountered in the GMEM variant due to misalignment, as each thread only needs to load a single sample during each stage. Consequently, the warp can easily be aligned. However, 2-way shared memory bank conflicts are expected for the SMEM variant due to the 8 byte size of the `float2` datatype, that represents a complex value, and may impact on performance. Bank conflicts are largely unavoidable for 64-bit access by each thread in a half-warp, and typically generates bank conflicts on the Tesla architecture [32].

Loop unrolling was implemented for all variants using conditional blocks that contain unconditional inner loop statements with unroll factors of 16, 64 and 256 for N. The block with the largest unroll factor that does not exceed N is selected using conditional statements in the kernel. Any residual of the filter length is processed using a standard loop, after the unrolled block is processed. For unroll factors greater than 256 the total code expansion became excessive, and loop unrolling was no longer being performed by the compiler, according to advisory warnings that were received.

5.5.1.2 FDFIR

The FDFIR is implemented as a series of kernels that perform frequency-domain convolution on the input burst and is based on the CUDA *FFT-Based 2D Convolution* example, but adapted to perform a batch of 1D convolutions using complex-to-complex transforms. Firstly, the input burst is zero-padded in the range dimension using a padding kernel to a total of $M = F + N - 1$ samples, as required with respect to the output length for convolution to produce the padded input matrix, x. A batch of 1D complex, out-of-place FFTs is performed on x using the CUFFT library to produce the frequency-domain equivalent, X. A batch size of S and FFT length of M is used and configured using the CUFFT library `cufftPlanMany` function prior to execution. The frequency-domain equivalent, H, of the impulse response matrix, h, for all filters in the burst is precomputed, as described later on in this section, with dimensions matching X. A Hadamard product kernel then performs an element-wise multiplication on the X and H matrices to produce Y. A batch of 1D complex, out-of-place IFFTs is then performed on Y with a batch size of S and IFFT length of M to produce the time-domain equivalent y. The padding kernel is then used again and configured to strip output values beyond F in the range dimension to produce an output buffer with the same dimensions as the original input buffer. Finally, a scale kernel scales complex values by multiplying with a real scale factor of $1/M$, which is implemented as the CUFFT library produces un-normalized outputs, that are scaled by the number of elements for a forward transform (FFT) followed by an inverse transform (IFFT) [62]. Scaling is left to the user to perform as seen fit and was included in this implementation to remain functionally equivalent to the TDFIR variant and other standard convolution implementations.

The FFT and IFFT performance is typically the determining factor with respect to the overall FDFIR performance, and thus it is important to be aware of performance considerations for the CUFFT library that is used to perform this operation. For transform lengths that can be factored as $2^a.3^b.5^c.7^d$ with $a,b,c,d \geq 0$, CUFFT employs the well-known Cooley-Tukey algorithm [53], with large prime lengths handled by other algorithms that have known disadvantages [62]. For the Cooley-Tukey path a brief summary of the constraints that provide the most efficient implementation for 1D single-precision transforms on the Tesla architecture from most to least general is:

1. Restrict FFT length to a multiple of 2,3,5, or 7 only.
2. Restrict 2^a term to a multiple of 16.
3. Restrict 2^a term to a multiple of 256.
4. Restrict FFT length to between 2 and 2048 and strictly 2^a term only.

These constraints are related to the efficiency of the recursive decomposition, global memory coalescing, and efficient shared memory implementations in the underlying library. Each subsequent constraint has the potential for providing an additional performance improvement. The exact FFT length is therefore an important factor in the FFT performance, and the minimum required FFT length of $M = F + N - 1$ for the FDFIR implementation is not likely to provide a good result in all cases. A simple scheme is proposed and implemented to ensure that listed constraint 1 through 3 are met in all cases, as follows. The actual FFT length is taken as the minimum FFT length snapped to the next higher multiple of 256 and power of 2. This scheme is expected to improve the overall FFT efficiency, but also introduces wasted computation proportional to the difference between the actual and minimum FFT lengths, which does not contribute to the computation of the problem. Constraint 4 cannot be met easily when the input or length approaches or exceeds 2048 without using the overlap-save or similar approach, which was not implemented.

With respect to implementation details of the supporting kernels, the pad kernel allows for both padding and stripping operations on 2D buffers by performing a copy operation between an input and output buffer with specified dimensions and pitch. The output buffer is zero-padded using a `cudaMemset` operation after the initial buffer allocation. For the scale complex kernel a common scale factor is applied to each sample, independent of the sample position, which allows the kernel to access the input buffer linearly as a 1D buffer, which means that sequential thread IDs map to sequential data indices in the input buffer. Optimal alignment and coalescing is then ensured irrespective of the pitch of the input buffer and dimensionality of the input data.

The H matrix is precomputed as follows. A matched filter coefficient set for each of the S pulses in the input burst is packed into a buffer. Using the padding kernel, the buffer is then zero-padded for sample positions beyond N in the range dimension to form the h matrix, which matches the X matrix dimensions of M fast-time samples by S slow-time samples. A batch of 1D complex, out-of-place FFTs is then performed on the h buffer to obtain the H matrix for the entire burst, again using a batch size of S and FFT length of M. Note that with the FDFIR implementation, multiple filter coefficient sets to support multiple pulse waveforms per burst can potentially be specified more easily than with the TDFIR implementation. For the FDFIR implementation the frequency-domain matched filter coefficients in H are accessed only once by the Hadamard product kernel and can therefore be stored and accessed in global memory directly. Conversely, the TDFIR coefficients are accessed repeatedly and subsequently typically require fast on-chip storage, which has limited capacity.

5.5.1.3 Kernel Analysis

The low-level metrics, as defined in Table 5.5, were derived for all the DPC kernels that were implemented, as shown in Table 5.6. A kernel occupancy of 1 was achieved for all kernels, which is set as a general design goal. The time complexity for the TDFIR kernels is considered $O(n^2)$ as the number of operations that need to be performed for each sample is a function of the filter length N, for a total operation count on the order of N^2 per filter when $N = F$. The time complexity for all other kernels is $O(n)$ due to a constant-time workload per sample for a linear function of the number of samples per burst.

Each thread for the TDFIR GMEM variant reads N complex input samples from GMEM, N filter coefficients from CMEM, and writes a single complex output sample to GMEM. Each thread performs N complex multiplications and additions to calculate its filter output sample. The GMEM kernel is expected to be bandwidth limited due to a relatively low AI of approximately 2 for large N. Somewhat better than predicted performance is expected due to the use of CMEM for the window coefficients, where the analysis here assumes a worst-case W_{dmem} without any constant cache hits. The TDFIR TMEM variant is identical to the GMEM variant, but reads the complex input samples via TMEM. According to the simple analysis presented here, which assumes a worst-case W_{dmem} without any texture or const cache hits, the TMEM kernel is also expected to be bandwidth limited. However, actual performance is expected to be better than predicted with cache hits. The SMEM variant is also identical to the GMEM variant, except from complex input samples that are read from GMEM into SMEM and used as an explicit cache. Each input sample is read a minimum of 2 times, which matches the minimum number of stages. For $N \geq 256$ in steps of 256, an additional stage and, therefore, an additional load per sample is required. As with the other variants, the analysis assumes a worst-case W_{dmem} with respect to const cache hits for the window coefficients and actual performance may exceed predicted performance.

Table 5.6: Derived low-level metrics for all DPC kernels for processing of a single burst with F fast-time and S slow-time complex samples where each sample consists of two 4 byte float words for a filter length of N. Occupancy is 1 for all kernels.

DPC Kernel	$\mathbf{W_{dmem}}$[*]	$\mathbf{W_c}$[**]	**AI**[†]	$\mathbf{T_c}$**(peak)**[††]	**Predicted Limitation**
TDFIR (GMEM)	$2(2N+1)$[‡]	$8N$	$2\frac{N}{N+0.5}$[‡]	51.0[‡]	BW
TDFIR (TMEM)	$2(2N+1)$[‡]	$8N$	$2\frac{N}{N+0.5}$[‡]	51.0[‡]	BW
TDFIR (SMEM)	$2(N+\lceil\frac{N}{256}\rceil+3)$[‡]	$8N$	$4\frac{N}{N+\lceil\frac{N}{256}\rceil+3}$	102.0[‡]	BW
Pad	4	0	N/A	N/A	BW
Hadamard Product	6	6	1	25.5	BW
Scale Complex	4	2	0.5	12.8	BW

[*]Normalized to words/sample for clarity - multiply by $4FS$ for W_{dmem} in bytes for entire burst.
[**]Normalized to FLOPs/sample for clarity - multiply by FS for W_c in FLOPs for entire burst.
[†]Specified as FLOPs/word for clarity - divide by 4 for AI in FLOPs/byte.
[††]Specified in GFLOPS based on simple theoretical calculation by using $\lim_{N\to\infty} AI$ for AI to predict limiting factor.
[‡]Based on worst-case W_{dmem} with respect to CMEM and TMEM, as cache hits reduce workload.

Each thread for the pad kernel only reads and writes a single complex sample and is, therefore, expected to be bandwidth limited, as it performs no computation that is considered algorithmically relevant. Each thread for the Hadamard product kernel reads a complex sample for each of the two product terms and writes a complex sample output for the product result. A complex multiplication of the two product terms is computed for the output. The kernel is expected to be bandwidth limited due to a constant, low AI per sample. Each thread of the scale complex kernel reads a complex input sample and writes a complex output sample. A scalar multiplication of the I and Q components is performed using a real scale factor. The scale factor is common among all threads and is accordingly supplied as a kernel argument, which is efficiently broadcast to all threads and is therefore not counted towards W_{dmem}. The scale kernel is also expected to be bandwidth limited due to a constant, low AI per sample.

5.5.2 Corner Turning (CT)

The CT function implementation was performed based on the CUDA SDK matrix transpose example that is described in [50] and discussed in Section 4.3.3. The SDK example only supports the `float` datatype and hence support for the `float2` datatype was added as the corner turn function is required to operate on complex data samples. The kernels were also expanded to include arguments for the buffer pitch to be specified in addition to the actual data dimensions, which allows for padded buffers to be used. Range checking of buffer indices against data dimensions was also added to the kernels in order to support arbitrary dimensions. The CT function is implemented as a single kernel with different variants as shown in Figure 5.5.

Initially a GMEM, SMEM and SMEM with diagonal block ordering kernel were implemented, that are representative of major optimization steps in [50] for a progression from a naïve to optimally coalesced and bank conflict-free to a partition camping free implementation. There is general consensus in literature regarding the solutions that are employed for these optimization steps, as summarized in Section 4.3.3.

In [50] instruction optimization is noted as a potential area for further improvement, but details for a solution are not presented. The SMEM with diagonal block ordering kernel calculates the block index within each thread using the slow modulus operator that is not available as a native instruction. Hence, an additional SMEM with diagonal block ordering plus block index broadcast kernel was implemented to attempt to improve instruction throughput by calculating the block index in a single

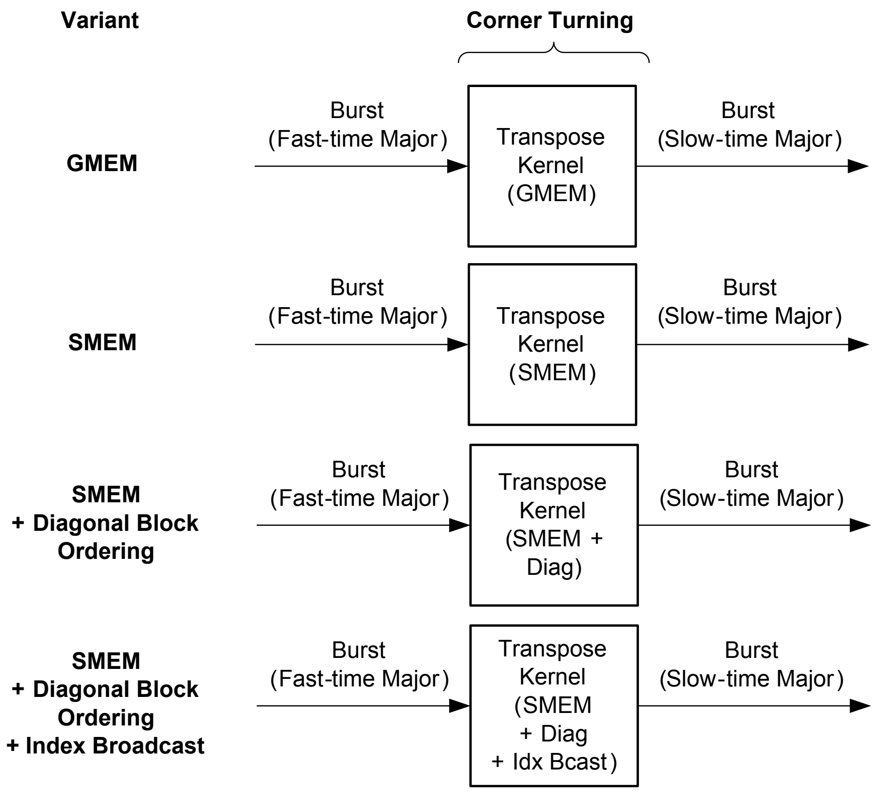

Figure 5.5: CT implementation that uses a single transpose kernel with varying implementations.

thread and communicating it to other threads via shared memory. As all threads read the same shared memory index the data is broadcast to each thread in a special bank conflict-free case as described in Section 3.6.3.3.

In [50] padding is also briefly mentioned as a potential alternative option to alleviate partition camping and was therefore explored. The Tesla C1060 global memory is divided into 8 partitions of 256 bytes [50]. The standard SMEM kernel could be used, as it supports the buffer pitch argument, and the host code was expanded to allow padding of input and output buffers to be specified in number of bytes.

5.5.2.1 Kernel Analysis

The low-level metrics, as defined in Table 5.5, were derived for all the CT kernels that were implemented, as shown in Table 5.7. A kernel occupancy of 1 was achieved for all kernels, which is set as a general design goal. All CT kernels have a time complexity of $O(n)$ as the workload per sample

Table 5.7: Derived low-level metrics for all CT kernels for transpose of a single burst with F fast-time and S slow-time complex samples where each sample consists of two 4 byte float words. Occupancy is 1 and time complexity is $O(n)$ for all kernels.

CT Kernel	$\mathbf{W_{dmem}}^{*}$	$\mathbf{W_c}^{**}$	$\mathbf{AI}^{\dagger}$	$\mathbf{T_c(peak)}^{\dagger\dagger}$	**Predicted Limitation**
GMEM	4	0	N/A	N/A	BW
SMEM	4	0	N/A	N/A	BW
SMEM + Diag	4	0	N/A	N/A	BW
SMEM + Diag + Idx Bcast	4	0	N/A	N/A	BW

*Normalized to words/sample for clarity - multiply by $4FS$ for W_{dmem} in bytes for entire burst.
**Normalized to FLOPs/sample for clarity - multiply by FS for W_c in FLOPs for entire burst.
†Specified as FLOPs/word for clarity - divide by 4 for AI in FLOPs/byte.
††Specified in GFLOPS.

is constant. The CT kernels do not perform any computation that is algorithmically relevant and are therefore expected to all be bandwidth limited, based on the simple analysis presented here.

5.5.3 Doppler Filter (DF)

For the DF implementation a window function kernel was implemented with both DFT and FFT implementation approaches to spectral analysis. The DF function performs Doppler processing on an entire burst with F fast-time range samples and S slow-time pulses. The window function applies a separate real coefficient as a scaling factor to each Doppler bin across all range bins prior to spectral analysis. For spectral analysis a batch of 1D DFTs is performed in the Doppler dimension for each range bin in parallel across the entire burst to maximize the data parallelism. The variants and kernels that were implemented for the DF are shown in Figure 5.6.

The window function coefficient set for a burst contains S entries in total, that are applied in a one-to-one relationship to the S Doppler bins in the burst across all range bins. Constant memory is used to store the window coefficients as the same window coefficient set is reused for all range bins and threads only require read access. A total of 64 KiB constant memory is available on the C1060 that is cached per SM, which means that a maximum of 16384 window coefficients can be stored in constant memory if this resource is not used for any other purpose by the kernel. This greatly exceeds the number of Doppler bins that are typically required, which is on the order of a few hundred pulses

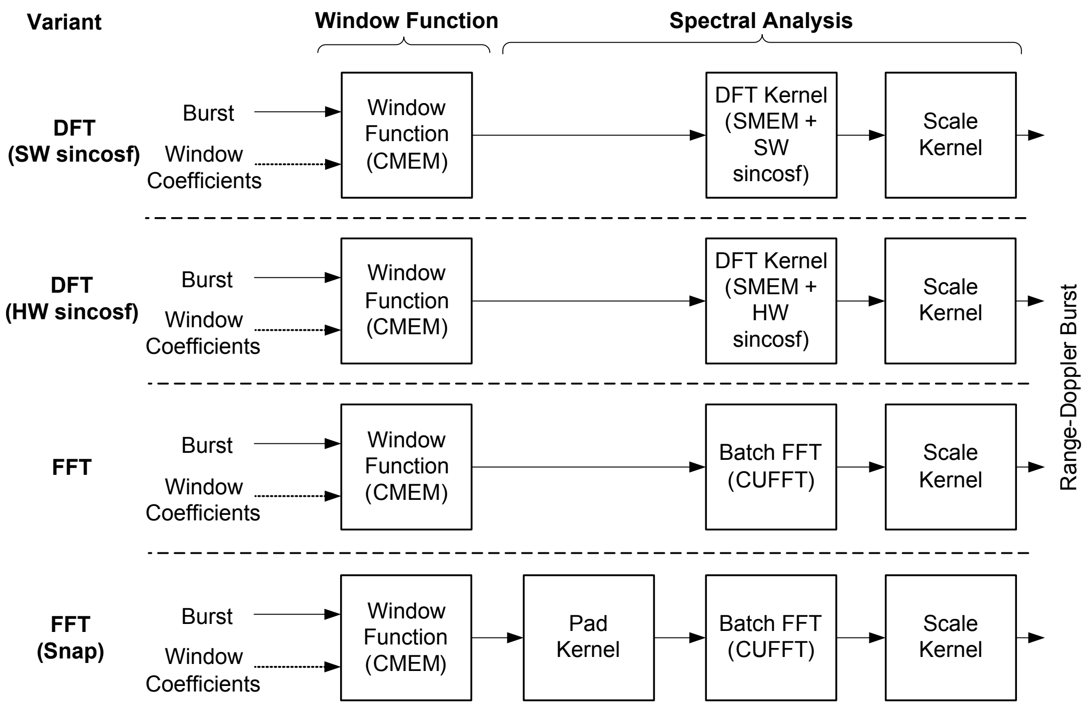

Figure 5.6: DF implementation showing all variants and kernels that were implemented with the CUFFT library indicated where used to perform certain functions for one variant.

at most. Global memory is used to read input samples, as global memory coalescing can easily be achieved with the simple one-to-one access pattern between input and output samples.

5.5.3.1 DFT

For the DFT variant a DFT kernel is implemented, which operates on the output of the window function and performs a direct DFT that produces un-normalized outputs. The DFT, $X[k]$ of an input sequence, $x[n]$ is given by Equation 5.1 [5].

$$X[k] = \sum_{n=0}^{N-1} x[n]e^{-j2\pi kn/N},\ k \in [0, N-1] \tag{5.1}$$

The calculation of the $e^{-j2\pi kn/N}$ term is implemented by precomputing the constant $c = \frac{2\pi}{N}$ on the CPU and computing e^{-jx} as $\cos(x) - j\sin(x)$, according to the trigonometric identity, with $x = ckn$, in a kernel on the GPU. Two DFT kernel variants are implemented, that use the software `sincosf` function, and the intrinsic, or hardware, `__sincosf` function that can only execute in device code, respectively. The `sincosf` function calculates $\sin(x)$ and $\cos(x)$ in a single function call. The hardware version of this function is aimed at faster computation through fewer native instructions, with slightly reduced accuracy.

The DFT kernels use shared memory for input data with a block of 512 threads, which is chosen as the maximum number of threads supported per block for the architecture, as shown in Table 5.4. A large thread block size allows an entire 1D DFT to be performed in a single thread block using SMEM to efficiently cache input data, which is reused extensively. Each thread loads a single DFT input sample into shared memory and outputs a single DFT sample. Therefore, a maximum DFT length of 512 is supported by the DFT kernels. In order to maintain high occupancy with very short DFT lengths, the maximum number of whole 1D DFTs that can fit into the thread block is performed by the kernels.

To avoid shared memory bank conflicts, the SMEM index is mapped sequentially with respect to the thread ID within a warp, when writing to shared memory. When reading from SMEM, the broadcast mechanism is used where each thread in the warp accesses the same index. However, 2-way shared memory bank conflicts are expected in general, as the `float2` datatype, that is used to represent a complex value, is 8 bytes in size. Bank conflicts are largely unavoidable for 64-bit access by each thread in a half-warp, and typically generates bank conflicts on the Tesla architecture [32]. Manual loop unrolling of the inner computation loop in the kernels is also performed, as the DFT kernels are expected to be compute bound due to the use of SMEM, which reduces total global memory bandwidth requirements significantly. The inner loop was unrolled for DFT lengths of 16, 32, 64, and 128 for the hardware variant, but could only be unrolled for lengths of 16 and 32 for the software variant. The total code expansion became excessive at this point, and loop unrolling was no longer being performed by the compiler, according to advisory warnings that were received. The DFT kernel is followed by a scaling kernel to normalize the output by multiplying with a factor of $\frac{1}{\sqrt{S}}$.

5.5.3.2 FFT

For the FFT variant a batch of 1D complex, out-of-place FFTs is performed on the window function output using the CUFFT library. A batch size of F and FFT length of S is used and configured using the CUFFT library `cufftPlanMany` function prior to execution. A scale kernel is used to scale data after the FFT transform due to the un-normalized outputs produced by CUFFT, as with the DPC function. In this case a real scale factor of $\frac{1}{\sqrt{S}}$ is appropriate as only a forward transform is performed. As with the FDFIR implementation for the DPC, the minimum required FFT length, determined by the number of pulses S in the slow-time dimension for the DF, is not guaranteed to be an efficient length for the CUFFT library. A simple FFT snap scheme is also employed for a second FFT variant that snaps to an efficient length using zero-padding to extend the FFT input buffer.

The scheme employed for the DF is similar to the scheme employed for the DPC FDFIR implementation described in Section 5.5.1.2. However, the number of pulses in a burst is typically much lower than the number of range bins, which was the primary determining factor for the FDFIR, around a few hundred at the very most. Snapping to a multiple of 256 is therefore expected to lead to significant wasted computation, hence the third constraint is relaxed, restricting the FFT length to a power of two and a multiple of 16 instead. A padding kernel, identical to the padding kernel used for the DPC FDFIR implementation, is used prior to the FFT step to perform zero-padding of the input buffer to the snapped FFT length. When the original FFT input length already matches the snapping criteria, the padding kernel is bypassed to avoid unnecessary overhead. It is important to note, however, that the number of output Doppler bins is directly proportional to the FFT length that is used and will only match the number of input pulses when the FFT length is not extended. In the cases where the FFT length is extended the Doppler dimension is also extended and may, therefore, increase the processing workload downstream in the processing chain.

5.5.3.3 Kernel Analysis

The low-level metrics, as defined in Table 5.5, were derived for all the DF kernels that were implemented, as shown in Table 5.8. A kernel occupancy of 1 was achieved for all kernels, which is set as a general design goal. For the window function kernel each thread reads a complex sample and a real window coefficient, performs two floating-point multiplies to scale the I and Q components by the real window coefficient, and then writes the complex sample output. The window function kernel has a constant, low AI and is therefore expected to be bandwidth limited. The scale complex kernel implementation is identical to the kernel used for the DPC function and is discussed in Section 5.5.1.3. The time complexity of the DFT algorithm is $O(n^2)$ as the workload for each sample is a linear function of the number of slow-time samples, S. The algorithms for the other kernels are all $O(n)$ time complexity due to a constant workload per sample. For the DFT kernels, each thread reads a single input sample from GMEM into SMEM, and outputs a single DFT sample. Each thread performs S iterations of the inner loop, where a sin and cos calculation is performed to determine the e^{-jx} term, which is multiplied with an input sample from SMEM using a complex multiply, and added to the output sample total using a complex add. Note that sin and cos are counted as a sin-

Table 5.8: Derived low-level metrics for all DF kernels for processing of a single burst with F fast-time and S slow-time complex samples where each sample consists of two 4 byte float words. Occupancy is 1 for all kernels.

DF Kernel	$\mathbf{W_{dmem}}^{*}$	$\mathbf{W_c}^{**}$	$\mathbf{AI}^{\dagger}$	$\mathbf{T_c(peak)}^{\dagger\dagger}$	Predicted Limitation
Win. Fn. (CMEM)	5‡	2	0.4	10.2	BW
DFT (SMEM, SW `sincosf`)	4	$10S$	$2.5S$	933.0	Compute
DFT (SMEM, HW `sincosf`)	4	$10S$	$2.5S$	933.0	Compute
Scale Complex	4	2	0.5	12.8	BW

*Normalized to words/sample for clarity - multiply by $4FS$ for W_{dmem} in bytes for entire burst.
**Normalized to FLOPs/sample for clarity - multiply by FS for W_c in FLOPs for entire burst.
†Specified as FLOPs/word for clarity - divide by 4 for AI in FLOPs/byte.
††Specified in GFLOPS based on simple theoretical calculation by using $\lim_{S\to\infty} AI$ for AI to predict limiting factor.
‡Based on worst-case W_{dmem} for constant memory (CMEM) implementations as const cache hits reduce workload.

gle FLOP each, and are implemented using the `sincosf` function for the software variant and the `__sincosf` function for the hardware variant.

5.5.4 Envelope Calculation (ENV)

For the envelope function two functional variants for a linear and square law rectifier were implemented. For the linear rectifier three variants were implemented where a software `hypotf` function, software `sqrtf` function, and hardware `__fsqrt_rn` function, which can only be executed in device code, is used respectively in the kernel as part of the calculation. In all cases the ENV kernel is implemented as a single kernel, as shown in Figure 5.7, for all variants.

All ENV kernels use a pure global memory implementation as there is no data reuse and the processing on each cell is identical, irrespective of its position in the range-Doppler space. This allows thread blocks to access a linear input buffer where sequential thread IDs map to sequential data indices in the input buffer. This ensures optimal alignment and coalescing irrespective of the actual row pitch or number of Doppler bins of the input buffer, which is otherwise an important factor when processing 2D data blocks from the input buffer. A block of 256 threads is used to achieve optimal occupancy and each thread operates on a single sample that is complex at the input comprised of two 4 byte float words and real at the output comprised of a single 4 byte float word.

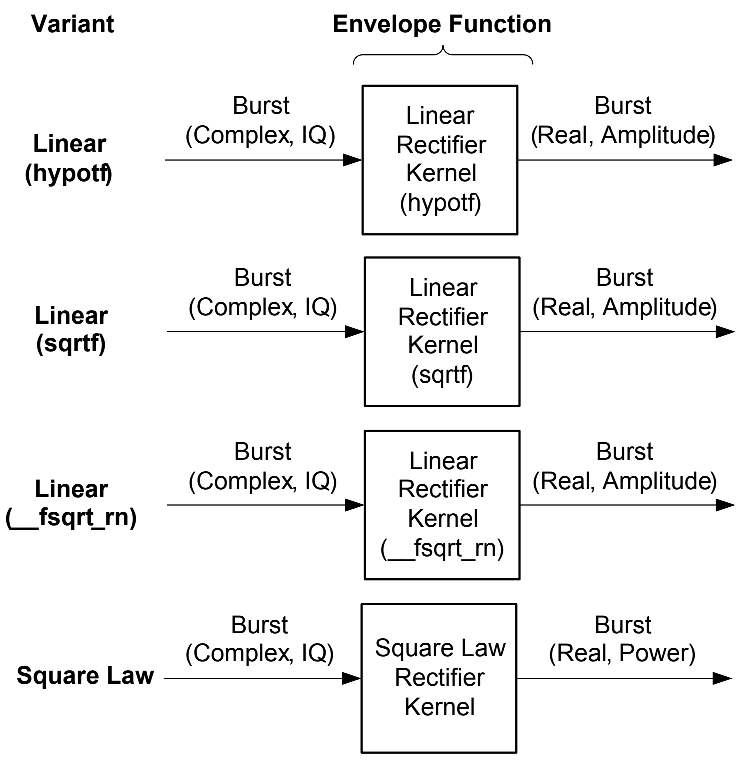

Figure 5.7: ENV implementation showing all variants that were implemented where the linear and square law variants differ functionally and produce amplitude and power outputs respectively.

The linear rectifier implementations differ only in the inner calculation of the amplitude value and are identical in all other respects. For the `hypotf` variant the input sample I and Q components are provided directly to the `hypotf` software function that calculates the hypotenuse of a triangle, which is equivalent to the magnitude function that is required. According to the inline implementation of the CUDA `hypotf` function in the `math_functions.h` header file, it uses a method that is not straightforward and that performs various checks and corrections for special case handling, such as divide by zero, NaNs and infinities, in some cases seemingly required due to the particular computation method that is used. The special case handling that is performed here is not considered a necessity for a straightforward ENV implementation for the radar application where input data is expected to be fairly homogenous with respect to floating-point math. The `hypotf` function also eventually calls the `sqrtf` function. For the `sqrtf` variant, I and Q components are first squared and added in the kernel and then provided to the `sqrtf` software function, which calculates the square root that is required to calculate the magnitude. The final `__fsqrt_rn` variant is identical to the prior variant, but uses an intrinsic or hardware `__fsqrt_rn` function that forms part of the GPU architecture instruction set directly.

The intrinsic functions are expected to be faster as they map to fewer native instructions, but typically provide lower accuracy than their software counterparts [32]. However, the `__fsqrt_rn` is documented as IEEE-compliant with respect to error bounds [32]. The SFU provides support for some of the additional native instructions that are used for intrinsic functions. Note that the GPU instruction set does not support a native single-precision floating-point square root instruction. It is generally implemented by the compiler as a single-precision floating-point reciprocal square root instruction and a single-precision floating-point reciprocal instruction that are supported natively [32]. The square law rectifer kernel merely squares and adds the I and Q components of each sample without any further calculation.

5.5.4.1 Kernel Analysis

Table 5.9: Derived low-level metrics for all ENV kernels for processing of a single burst with F fast-time and S slow-time samples where input samples are complex consisting of two 4 byte float words and output samples are real consisting of 4 byte float words. Occupancy is 1 and time complexity is $O(n)$ for all kernels.

ENV Kernel	$\mathbf{W_{dmem}}$[*]	$\mathbf{W_c}$[**]	AI[†]	$\mathbf{T_c(peak)}$[††]	Predicted Limitation
Linear (hypotf)	3	4	4/3	34.0	BW
Linear (sqrtf)	3	4	4/3	34.0	BW
Linear (__fsqrt_rn)	3	4	4/3	34.0	BW
Square Law	3	3	1	25.5	BW

[*]Normalized to words/sample for clarity - multiply by $4FS$ for W_{dmem} in bytes for entire burst.
[**]Normalized to FLOPs/sample for clarity - multiply by FS for W_c in FLOPs for entire burst.
[†]Specified as FLOPs/word for clarity - divide by 4 for AI in FLOPs/byte.
[††]Specified in GFLOPS.

The low-level metrics, as defined in Table 5.5, were derived for all the ENV kernels that were implemented, as shown in Table 5.9. A kernel occupancy of 1 was achieved for all kernels, which is set as a general design goal. All kernels also have a time complexity of $O(n)$, as the workload is a linear function of the number of samples in the burst. All kernels exhibit a constant device memory and computational workload per sample with a constant, low AI and are therefore expected to be limited by bandwidth. Note that the square root operation is counted as 1 FLOP for the analysis presented here.

5.5.5 Constant False-Alarm Rate (CFAR)

For the CFAR a 2D CA-CFAR was implemented with a CFAR window that wraps in the Doppler dimension and clips in the range dimension. An apron of cells is generated around the input burst data as an initial specialized padding step to simplify and improve the performance of the subsequent interference estimation step. The interference estimation is then performed by calculating the average power over N cells in the CFAR window for each cell in the burst. Lastly, a detection mask is generated by marking detections according to a threshold calculated using the alpha constant and the interference estimate for each cell compared to its input power. The CA-CFAR variants and kernels that were implemented are shown in Figure 5.8.

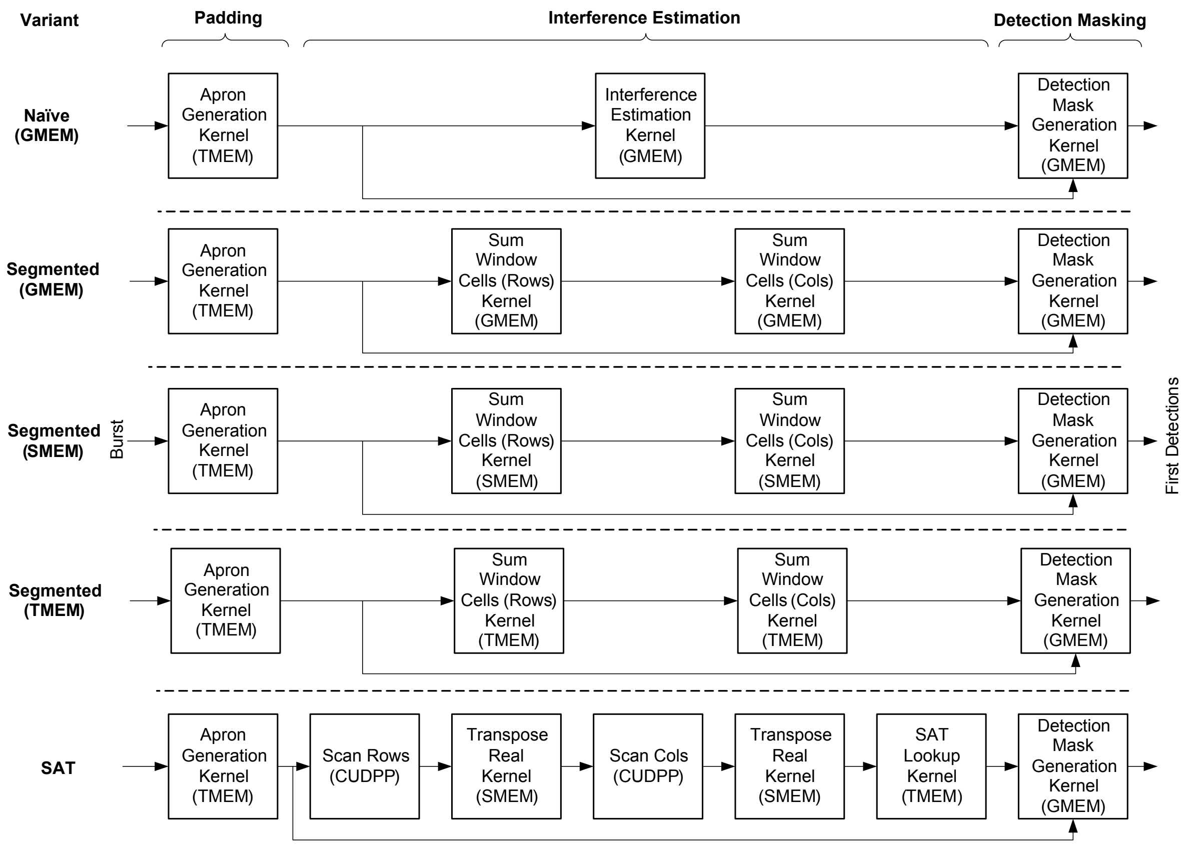
Variant
Padding
Interference Estimation
Detection Masking
Naïve (GMEM)
Apron Generation Kernel (TMEM)
Interference Estimation Kernel (GMEM)
Detection Mask Generation Kernel (GMEM)
Segmented (GMEM)
Sum Window Cells (Rows) Kernel (GMEM)
Sum Window Cells (Cols) Kernel (GMEM)
Segmented (SMEM)
Burst
Sum Window Cells (Rows) Kernel (SMEM)
Sum Window Cells (Cols) Kernel (SMEM)
First Detections
Segmented (TMEM)
Sum Window Cells (Rows) Kernel (TMEM)
Sum Window Cells (Cols) Kernel (TMEM)
SAT
Scan Rows (CUDPP)
Transpose Real Kernel (SMEM)
Scan Cols (CUDPP)
SAT Lookup Kernel (TMEM)

Figure 5.8: CFAR implementation showing all variants and kernels that were implemented, with the CUDPP library indicated where used to perform certain functions for one variant. The variants that are presented differ only with respect to the interference estimation step implementation. Note that kernels indicated as texture memory (TMEM) only read from a texture and write to global memory.

5.5.5.1 Apron Generation

For pulse-Doppler radar it is possible to wrap the CFAR window around the burst edge in the Doppler dimension, as the corresponding cells at the opposite burst edge are at the equivalent range and adjacent in Doppler space, in particular when using low PRF waveforms that cause Doppler ambiguities. The CFAR window may be wrapped instead of clipped in Doppler to ensure that enough cells are contained in the CFAR window to still provide a good statistical interference estimate. There are various additional factors with regard to the range dimension to consider, which makes a similar wrapping scheme problematic. Therefore, clipping in range at the burst edge is considered more practical for most systems.

The apron generation step alleviates the need for explicit checks and logic during interference estimation to implement the behavior of the CFAR window beyond the burst edges, that is otherwise required. For this implementation the apron generation implementation determines the CFAR window behavior beyond the burst edge, by enabling the interference estimation kernels to transparently extend the CFAR window beyond the boundary of the original input burst into regions selectively padded by the apron generation kernel. An apron generation kernel was implemented for CFAR window wrapping in the Doppler dimension and clipping in the range dimension, as shown in Figure 5.9. However, other schemes can be employed by merely altering the apron generation kernel implementation without affecting the implementation for subsequent CFAR processing stages. Cells that fall within half of the CFAR window extent from the burst edge in the Doppler dimension are duplicated at the opposite burst edge to implement wrapping. All remaining cells in the padded buffer are zero-padded using a `cudaMemset` operation after initial buffer allocation in order to implement clipping. The regularity and spatial locality of memory access patterns at the burst edge in subsequent stages is improved in addition to reduced complexity, compared to explicit wrapping or clipping of data indices instead.

Furthermore, the CFAR input burst dimensions may not allow for optimal 2D data access patterns that are convenient for the CFAR. For the CFAR, the input burst data is a 2D range-Doppler map, that contains power values for each cell organized in the slow-time or Doppler-major data format. The row pitch for the input buffer is therefore equal to the number of Doppler bins in the burst, which is not guaranteed to be an efficient stride with regard to alignment between rows, when accessed as a 2D buffer during processing. The apron generation step provides an opportunity to add additional padding to ensure optimal memory alignment for arbitrary burst and CFAR window sizes.

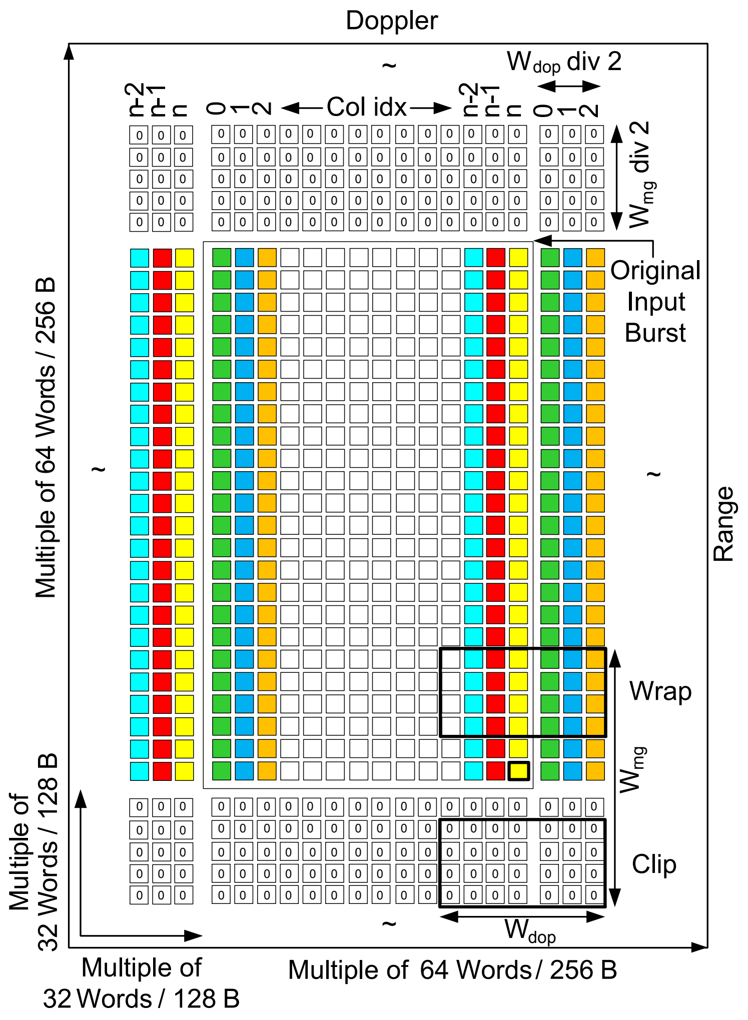
Doppler
W$_{dop}$ div 2
Col idx
n-2
n-1
n
0
1
2
W$_{mg}$ div 2
Original Input Burst
Multiple of 64 Words / 256 B
Range
Wrap
W$_{mg}$
Clip
W$_{dop}$
Multiple of 32 Words / 128 B
Multiple of 32 Words / 128 B
Multiple of 64 Words / 256 B

Figure 5.9: CFAR apron generation example output showing columns that are duplicated from the input burst and rows that are zero-padded to achieve CFAR window wrapping in Doppler and clipping in range at the burst edge during the subsequent interference estimation processing stage.

As shown in Figure 5.9, the burst origin is placed at an offset in the padded buffer which is a multiple of 128 bytes or 32 single-precision float words of 4 bytes, which aligns with the largest single global memory transaction size for the architecture. The minimum number of 128 byte padding words that can accommodate half the CFAR window extent in each respective dimension, which represents the maximum CFAR window overlap beyond the burst edge, are automatically allocated. A single 128 byte padding word is generally more than sufficient for the typical CFAR window dimensions.

The width and height of the padded buffer is also allocated as a multiple of 256 bytes or 64 words in order to present an optimal row pitch for 2D access. The choice of 256 bytes for alignment is equal to the device memory partition width and reported texture alignment boundary, and also in accordance with the observed behavior for the `cudaMallocPitch` function that allocates buffers optimally for 2D access on the architecture in question. Note that the standard `cudaMalloc` function to allocate linear device memory was used instead, with manual adjustments for buffer allocation size, as concurrent kernel execution and data transfer is not supported on CC 1.x devices for buffers allocated as CUDA arrays or 2D arrays using `cudaMallocPitch` [32]. It is desirable to retain the option to overlap kernel execution and data transfer due to the advantages observed for the corresponding microbenchmark in Section 6.2.4. Note that the same padding logic is applied throughout by the apron generation kernel in both dimensions, even though the row pitch is generally the only relevant factor with regard to performance with the number of rows not playing a major role. However, the same logic is applied in both dimensions in order to ensure an optimal data layout even when the buffer is transposed, which is required for some of the variants that were implemented.

The apron generation kernel thread blocks operate on 16x16 element blocks. Each thread reads a word from the input burst buffer and writes the word at least once to the padded buffer, with a second conditional write to a different address required for the duplicated cells in one of regions adjacent to the burst edge. For the final implementation each thread instead operates on two words in adjacent rows to alleviate an instruction throughput limitation caused by index calculation and the conditional checks for the burst edge regions, by amortizing these costs across double the amount of data reads and writes to allow for more optimal latency hiding. Also, the apron generation kernel was initially implemented to read and write via global memory. However, in cases where the input burst row pitch is suboptimal due to the particular burst dimensions, the performance for global memory reads suffer, due to misalignment between rows that causes suboptimal coalescing and, therefore, degraded performance, despite good performance on writes due to the padded buffer layout. As a final optimization the apron generation kernel uses texture memory as a read path for the input burst data in order to reduce the misalignment effects on reads.

5.5.5.2 Interference Estimation

The interference estimation step is the most processing and memory intensive step and, accordingly, a variety of approaches were therefore considered here as different variants that were implemented, as shown in Figure 5.8. A naïve variant was initially implemented with a single interference estimation kernel that uses global memory, where each thread calculates the interference estimate for a single CUT. The naïve variant performs $N = N_{rng} \times N_{dop}$ global memory reads, $N-1$ floating-point additions, a single floating-point multiply by $1/N$, and a single global memory write per CUT in the input burst, according to the illustration and definitions in Figure 2.5. A series of variants were then implemented where the interference estimation is segmented into two summation stages for rows and columns that map to range bins and Doppler bins. The two stages are implemented as two separate kernels that execute sequentially.

The segmented approach only performs $N_{rng} + N_{dop}$ memory reads, $(N_{rng} - 1) + (N_{dop} - 1)$ floating-point additions, a single floating-point multiply by $1/N$, and two global memory writes per CUT in the input burst. The memory and computational workload is therefore reduced to an additive instead of multiplicative relationship with regard to the CFAR window size. The additional global memory write is required to store the partial sum results between the two stages. Lastly, a summed-area table (SAT) variant that also operates in two stages and that exploits the constant-time sum lookup properties of a SAT was implemented. During the first stage a 2D SAT is generated from the input data requiring a series of kernels, with the second stage using the SAT to perform lookups in order to determine the CFAR window sums required to calculate the interference estimate for each CUT in the input burst. The SAT variant is aimed at decoupling the CA-CFAR performance from the CFAR window size. No variant that is based on the sliding window technique was implemented, due to the low degree of parallelism and correspondingly poor GPU performance it exhibits, as discussed in Section 4.3.5.

5.5.5.3 Interference Estimation Segmented Variants

The variants that use the segmented structure all write output data to global memory, but are distinguished by the method used for reading of input data. The efficiency of reading input data is deemed the main performance contributor for the interference estimation kernels, due to the potentially high ratio between the input and output memory workloads and the nature of the access patterns required for the input data based on the CFAR window size and position. The kernels for the GMEM segmented variant read input data directly from global memory, but suffer partially from misaligned reads and therefore suboptimal coalescing due to the memory access pattern exhibited when each thread reads each CFAR window cell in turn, which is not optimally aligned for all combinations. The misalignment effects that are shown for the global memory microbenchmark in Figure 6.3 are therefore expected for the GMEM kernels.

A second, SMEM segmented variant was implemented where kernels read a block of input data from global memory into much faster on-chip shared memory to perform the summation. Performance improvements are expected, due to the data reuse associated with significant overlap in CFAR window input data for adjacent CUT positions. The structure used for the SMEM segmented implementation is loosely based on [45] to implement an efficient separable filter using shared memory. With the SMEM segmented kernels each thread loads three input values into shared memory from three adjacent blocks in global memory, as shown in Figure 5.10 and Figure 5.11 for rows and columns respectively. The access from each thread is separated by 16 rows or columns between blocks, which leads to an efficient access pattern that does not suffer from the misalignment effects experienced by the GMEM segmented kernel. As demonstrated for the global memory microbenchmark in Figure 6.3, memory access with an offset of 16 words, or columns in this case, shows no degradation due to optimal alignment. Any offset in rows is also guaranteed to produce optimal alignment due to the additional padding that is performed during the apron generation, as shown in Figure 5.9, which provides an aligned pitch. Sequential shared memory locations are accessed using sequential thread IDs to ensure bank conflict free access. There are 16 shared memory banks that correspond to consecutive 32-bit words. A half-warp of 16 threads access the shared memory at once and the CFAR implementation uses 32-bit `float` words, meaning bank conflict free access is achievable.

The center memory block cell positions are evaluated, with the two outer memory blocks providing access to the corresponding CFAR window cells that are required, which extend beyond the center memory block by definition. Therefore, three memory blocks are loaded to calculate the outputs that correspond to a single memory block, as shown in Figure 5.10 and Figure 5.11, which means that each cell will be loaded 3 times by 3 independent thread blocks in order to calculate the outputs for all memory blocks. Consequently, for the rows and columns kernel combined, each cell is loaded a constant 6 times irrespective of N, where the device memory workload for the GMEM kernels is a function of N. However, the SMEM segmented implementation in its current form supports maximum CFAR window dimensions in range W_{rng} and Doppler W_{dop} of 33 as the CFAR window extent from the CUT is defined as $W_{rng}/2$ with thread and memory block sizes currently defined as 16 by 16. The limit imposed by the current implementation is not considered a limitation for the radar application which typically uses CFAR window sizes on the order of $N = 20$ to $N = 40$ cells, where the implementation supports sizes of hundreds to more than a thousand cells, depending on the window shape. The CFAR window dimensions in conjunction with the guard cell dimensions determine the CFAR window size N as defined in Figure 2.5.

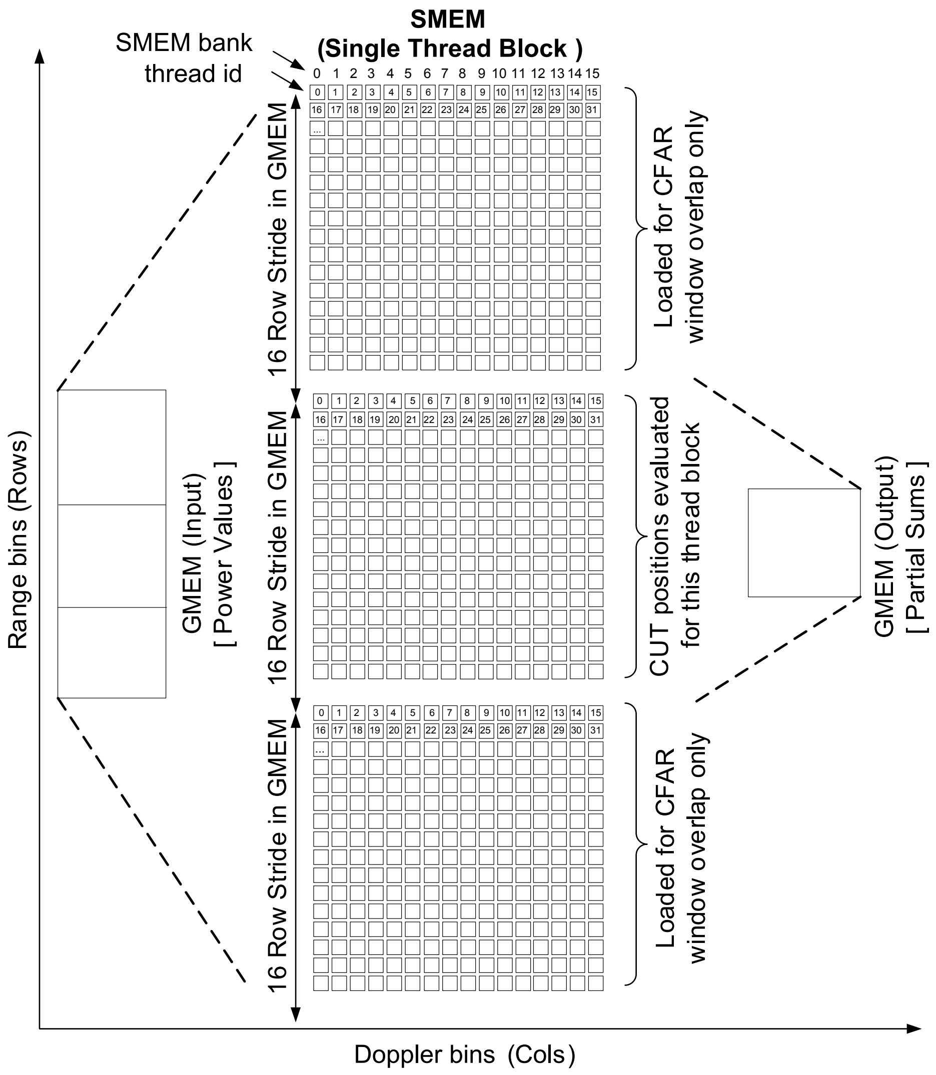

Figure 5.10: CFAR SMEM segmented rows kernel shared memory structure that achieves optimal global memory coalescing for reads and writes with no shared memory bank conflicts.

The kernels for the third, TMEM segmented variant read inputs values via read-only texture memory, which is cached and does not impose the same coalescing requirements as global memory, as discussed in Section 3.6.3.2. Texture memory in general does not provide improved performance under all conditions, but was considered for implementation due to the significant spatial locality exhibited by the rectangular CFAR windows, due to the suboptimal memory coalescing experienced with the GMEM implementation, and as a potential alternative to the SMEM segmented implementation. The texture cache is expected to provide improved performance over the GMEM segmented implementation on cache hits due to the level of data reuse and spatial locality exhibited by the CFAR. The TMEM segmented implementation can be considered an implicit caching implementation, whereas

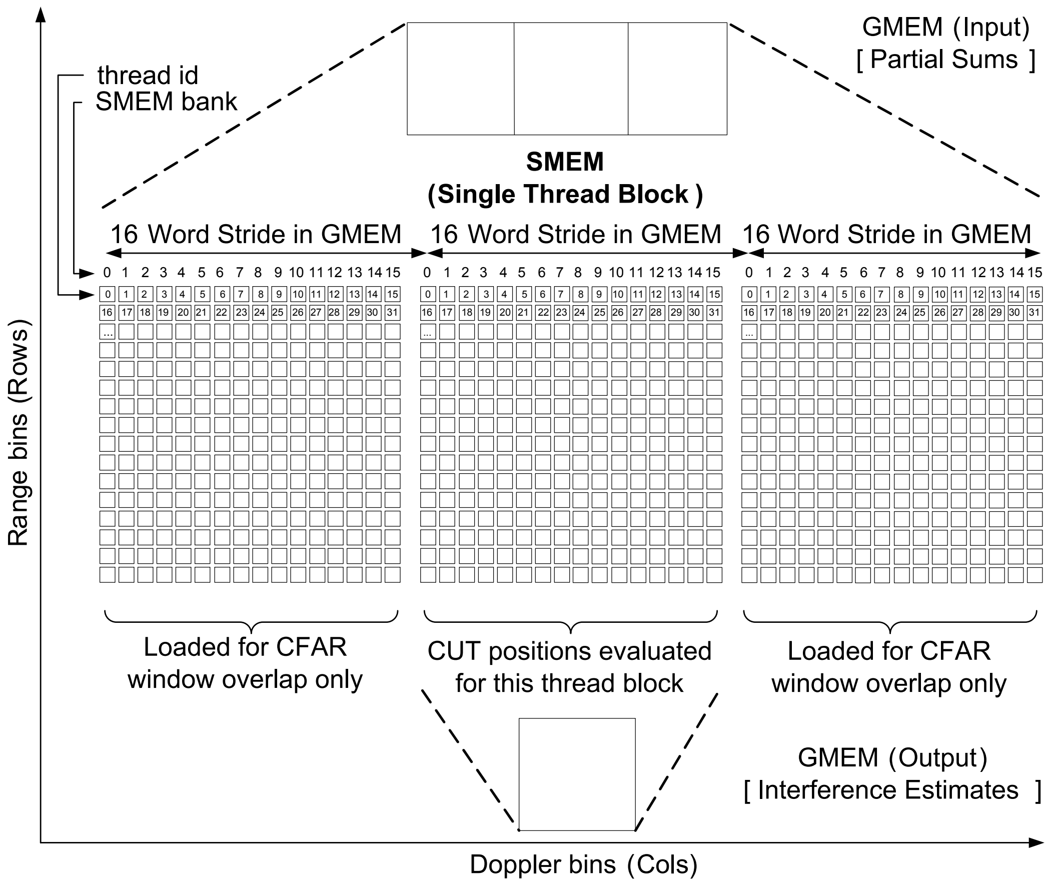

Figure 5.11: CFAR SMEM segmented cols kernel shared memory structure that achieves optimal global memory coalescing for reads and writes with no shared memory bank conflicts.

the SMEM segmented implementation uses explicit caching, but both are aimed at achieving the same objectives with regard to an improvement over the GMEM segmented variant.

Note that the reciprocal of the CFAR window size, $1/N$ is a constant that is pre-calculated on the CPU and passed to the GPU interference estimation kernels for all variants to calculate the average interference power from the sum over all CFAR window cells. By pre-calculating the reciprocal, a floating-point multiplication can be performed on the GPU instead of a floating-point division (RCP), which has a much lower throughput, as observed from the microbenchmark in Section 6.2.1.

5.5.5.4 Interference Estimation SAT Variant

A summed-area table (SAT) provides a 2D data structure that allows for efficient calculation of the sum of all elements in any rectangular subset of the source matrix that the algorithm is applied to.

The SAT algorithm was originally introduced in the computer graphics domain [63] and is also used in image processing fields, where it is called integral images [64]. However, the SAT algorithm also offers an interesting and potentially efficient implementation approach to the CA-CFAR algorithm which requires summation of the rectangular leading and lagging CFAR windows for each cell during interference estimation, as shown in Figure 2.5.

A SAT S that is generated from an input matrix i has the property that the value of a cell (x,y) in S is equal to the sum of all values within a block in i with indices less than or equal to (x,y) in either dimension, as shown in Equation 5.2.

$$S(x,y) = \sum_{\substack{v \leq x \\ w \leq y}} i(v,w) \tag{5.2}$$

A SAT can be generated using a prefix sum or scan operation, which produces a sequence of running totals from an input sequence, where each element j in the output sequence is equal to all preceding elements including j in the input sequence. A naïve scan operation implementation that calculates the scan in a single pass is inherently sequential and, therefore, inefficient for data-parallel GPU implementation. However, algorithms for parallel computation of the scan operation have been available for many years [65, 66] and more recently specifically for the GPU architecture initially with $O(n \log n)$ work complexity [67] and later optimized to $O(n)$ work complexity [68]. This optimized parallel scan algorithm [68] now forms the basis of the implementation used in the CUDA Data-Parallel Primitives (CUDPP) library [69].

Furthermore, the CUDPP library also provides a `cudppMultiScan` function which performs a batch of parallel scans on multiple rows simultaneously. The CUDPP library provides a satGL example that generates a SAT using `cudppMultiScan`, where the SAT is used subsequently to simulate depth of field blur in a scene. The SAT generation step for the CA-CFAR was implemented using the CUDPP library based on the satGL example. In satGL the SAT is generated by performing a scan across rows, a transpose to transform columns into rows, followed by a second scan to generate the complete SAT. The transpose is required as CUDPP only supports scanning across elements with a unit stride. The same steps were used for the generation of the SAT for the CA-CFAR implementation, although an additional transpose was added after the second scan stage in order to retain the original data ordering for consistency with other variants, as shown in Figure 5.12. The transpose operations were implemented using a CT kernel that is equivalent to the SMEM with diagonal block ordering plus block index broadcast kernel discussed in Section 5.5.2, but which uses the `float` datatype for a real transpose as required for the CFAR.

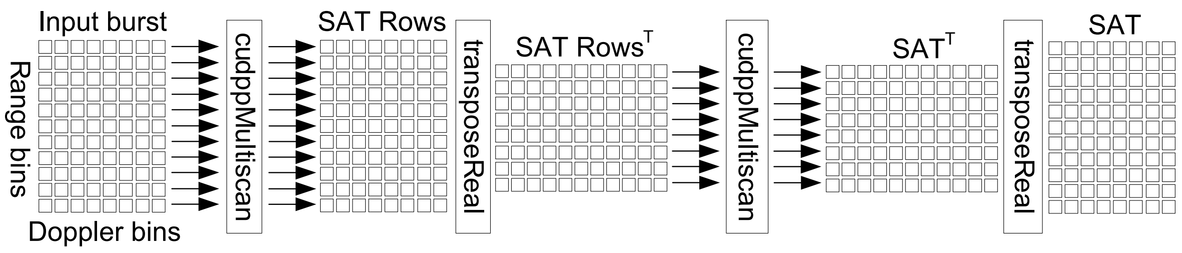

Figure 5.12: CFAR SAT generation steps using the `cudppMultiScan` library function to perform the scan operation with a custom transpose kernel implementation for real values to reorder the data into a format that is usable by the library for the different stages.

The CUDPP library and `cudppMultiScan` function also requires an execution plan to be generated prior to execution, similar to the well-known adaptive FFTW CPU library, using the `cudppPlan` function where configuration settings and buffer dimensions are specified. The plan creation generates internal infrastructure and storage for algorithms and optimization information for present GPU hardware in some cases. A plan may be reused for execution repeatedly, provided the same plan configuration settings and buffers are used. The configuration settings that were specified for the `cudppMultiScan` function is to perform an inclusive forward scan using the add operation and `float` datatype, which corresponds to the SAT definitions provided. It should also be noted that the documentation for the `cudppMultiScan` function indicates that rows need to be aligned to the appropriate boundaries for the architecture to achieve good performance, which is achieved due to the additional padding to a multiple of 64 words that is performed by the apron generation kernel, as shown in Figure 5.9.

Using the generated SAT S, Equation 5.3 can be used determine the sum of all elements within an arbitrary rectangle in the original input matrix i, where $S(x,y) = 0$ for $x,y < 0$.

$$\sum_{\substack{x_0 \le x \le x_1 \\ y_0 \le y \le y_1}} i(x,y) = S(x_1,y_1) - S(x_0-1,y_1) - S(x_1,y_0-1) + S(x_0-1,y_0-1) \tag{5.3}$$

For the SAT lookup stage a lookup kernel is implemented which reads 4 values from the SAT for both the leading and the lagging CFAR windows for a total of 8 reads per CUT. Texture memory was also used as a read path for the lookup kernel to alleviate misaligned access and to serve as an implicit cache as with the segmented variants. Misalignment is expected for the lookup kernel as the lookup offsets depend on the respective CUT position and the CFAR window size, which cannot be regulated to ensure optimal access.

The SAT lookup kernel then performs 3 floating-point summations for each of the two windows to determine their individual sums, one floating-point summation to determine the overall CFAR window sum, and a floating-point multiply by $1/N$ to calculate the interference estimate for a total of 8 FLOPs per CUT. The SAT lookup kernel therefore has a constant memory and computational workload for each CUT, in contrast with all other implemented variants that have workloads that are a function of N. The SMEM segmented implementation also features a constant device memory workload. However, the computational workload is still a function of N.

5.5.5.5 Detection Masking

The final detection masking stage is implemented with a single detection mask generation kernel which uses the interference threshold for each cell to calculate a threshold by multiplying with the CFAR constant α, which is pre-calculated on the CPU using Equation 2.10. The calculated threshold is then compared to the sample power for each cell. A 1 is written to the detection mask for hypothesis H_1 when the threshold is matched or exceeded and a 0 is written otherwise for H_0, the null hypothesis. The detection masking kernel therefore performs 1 read for the interference estimate, 1 read for the sample power, 1 floating-point multiply, 1 comparison and 1 write for the detection mask flag for every cell. The detection mask generation kernel uses global memory as optimal coalescing can easily be achieved using the padded structure for input and output.

A further optimization that may be performed for the CA-CFAR is to merge the detection masking kernel with the preceding interference estimation kernel that calculates the final interference estimate value for each variant. By merging these kernels the cell threshold can be calculated directly in the merged kernel and used to generate the detection mask, which alleviates the fixed overhead of transferring the intermediate interference estimate values to another kernel via global memory. Care also has to be taken when merging kernels with regard to register usage that will typically increase and the potential negative effect thereof on occupancy. An option was implemented that is controlled via compile-time flags to either use a dedicated detection masking kernel, or perform the detection inline with the interference estimation in a merged kernel for all variants. For simplicity, and to emphasize the differences between the interference estimation variants, the case where a dedicated detection masking kernel is used is considered as shown in Figure 5.8, unless otherwise noted.

5.5.5.6 Kernel Analysis

The low-level metrics, as defined in Table 5.5, were derived for all the CFAR kernels that were implemented, as shown in Table 5.10. One of the design goals was to attempt to always achieve a kernel occupancy of 1, which was achieved for all the CFAR kernels. The time complexity for all kernels is $O(n)$ as the time is a linear function of the total number of samples in the burst, despite the actual workload per sample being a function of the CFAR window size N, or a constituent thereof, in some cases. The kernels that are invoked by the CUDPP library functions that are used for the SAT variant are not included, as these kernels were not implemented by hand, and consequently, the necessary working knowledge required to derive the metrics is not sufficient.

Based on the analysis, all CFAR kernels are expected to be bandwidth limited by device memory throughput, except for the Segmented (SMEM) kernels that are expected to be compute bound for sufficiently large N_{rng} and N_{dop}, which are the constituent dimensions of the CFAR window N. This is due to the AI which is constant, or which approaches a constant for sufficiently large N for other kernels, whereas the AI for the Segmented (SMEM) kernel increases with N due to a constant device memory workload with an increase in the computational workload for increasing N. The SMEM kernel operates on the data that is loaded into the on-chip shared memory for each block, which does not utilize device memory bandwidth once loaded. For sufficiently large N the total execution time for the segmented variants for the rows and columns kernels are also expected to be less than the total execution time for the single-kernel naïve variant. This performance difference is expected in spite of the identical $T_c(peak)$ theoretical throughput, but instead due to the reduced overall memory and computational workloads from the multiplicative compared to additive relationship with respect to N_{rng} and N_{dop}.

The SAT Lookup kernel also has a constant device memory bandwidth, but also features a constant computational workload. Again, despite a lower overall $T_c(peak)$, the SAT lookup kernel is expected to exceed the performance of the other kernels, with respect to total execution time for sufficiently large N, due to the benefits of a constant workload over a workload that is a function of N. However, the overhead of the SAT generation is expected to be a major factor for this variant. Furthermore, the apron generation and transpose kernels perform no computation and are therefore expected to be bandwidth limited by definition.

Table 5.10: Derived low-level metrics for all CFAR kernels for processing of a single burst with F fast-time and S slow-time samples that are real and represented by 4 byte float words for a CFAR window containing N cells with N_{dop} cells in the Doppler dimension and N_{rng} cells in the range dimension. Occupancy is 1 and time complexity is $O(n)$ for all kernels.

CFAR Kernel	$\mathbf{W_{dmem}}^{*}$	$\mathbf{W_{c}}^{**}$	$\mathbf{AI}^{\dagger}$	$\mathbf{T_c(peak)}^{\dagger\dagger}$	**Predicted Limitation**
Apron Generation					
Apron Gen. (TMEM)	$2^{\ddagger}$	0	N/A	N/A	BW
Interference Estimation					
Naïve (GMEM)	$N+1$	N	$\frac{N}{N+1}$	25.5	BW
Seg. Rows (GMEM)	$N_{rng}+1$	$N_{rng}-1$	$\frac{N_{rng}-1}{N_{rng}+1}$	25.5	BW
Seg. Cols (GMEM)	$N_{dop}+1$	N_{dop}	$\frac{N_{dop}}{N_{dop}+1}$	25.5	BW
Seg. Rows (TMEM)	$N_{rng}+1^{\ddagger}$	$N_{rng}-1$	$\frac{N_{rng}-1}{N_{rng}+1}^{\ddagger}$	$25.5^{\ddagger}$	BW
Seg. Cols (TMEM)	$N_{dop}+1^{\ddagger}$	N_{dop}	$\frac{N_{dop}}{N_{dop}+1}^{\ddagger}$	$25.5^{\ddagger}$	BW
Seg. Rows (SMEM)	4	$N_{rng}-1$	$\frac{N_{rng}}{4}-0.25$	933.0	Compute
Seg. Cols (SMEM)	4	N_{dop}	$\frac{N_{dop}}{4}$	933.0	Compute
SAT Transpose (SMEM)	2	0	N/A	N/A	BW
SAT Lookup (TMEM)	$9^{\ddagger}$	8	$8/9^{\ddagger}$	$22.7^{\ddagger}$	BW
Detection Masking					
Det. Mask Gen. (GMEM)	3	2	$2/3$	17.0	BW

*Normalized to words/sample for clarity - multiply by $4FS$ for W_{dmem} in bytes for entire burst.
**Normalized to FLOPs/sample for clarity - multiply by FS for W_c in FLOPs for entire burst.
†Specified as FLOPs/word for clarity - divide by 4 for AI in FLOPs/byte.
††Specified in GFLOPS based on simple theoretical calculation by using $\lim_{N\to\infty} AI$ for AI to predict limiting factor.
‡Based on worst-case W_{dmem} for texture memory (TMEM) implementations as texture cache hits reduce workload.

It should be noted, however, that the analysis presented here only takes a limited set of parameters into consideration, which excludes for instance the potential impact of on-chip memory bandwidth limitations and caches, where appropriate. The computational workload metric also only considers the primary algorithmic computation and excludes all other computation and instruction overhead to calculate data indices and loop overhead, for instance. The results presented here should, therefore, only be treated as indicative for a first order analysis and will not necessarily compare directly with actual measurements.

5.5.6 Radar Signal Processing Chain

A pulse-Doppler radar signal processing chain was constructed from the individual radar signal processing functions that were implemented. The variants shown in Table 5.11 are recommended for the radar application, based on initial findings for each function, and were used for the radar signal processing chain implementation. The radar signal processing chain performs the entire typical data-independent processing stage by concatenating the kernels for the selected variants, as shown in Figure 5.13. Additional holistic optimization may be applied to the complete radar signal processing chain in order to further improve performance.

Table 5.11: Function variants selected for radar signal processing chain implementation.

Function	Variant
DPC	FDFIR (Snap)
CT	SMEM + Diag + Idx Bcast
DF	FFT (Snap)
ENV	Square Law
CFAR	Segmented (SMEM)

Firstly, there are number of architectural constraints to highlight and take into consideration at this stage with respect to the design and optimization of the radar signal processing chain, as follows.

1. Single kernel execution - Concurrent kernel execution not supported on Tesla C1060.
2. HtoD and DtoH transfers are slow - PCIe bus much lower bandwidth than device memory.
3. HtoD transfer cannot overlap with DtoH - Only 1 copy engine provided by Tesla C1060.

Only a single kernel can be actively executed at a time on the Tesla C1060 GPU. Task-level parallelism in the form of executing multiple processing functions in parallel on independent data, for instance, individual pulses during pulse compression, is therefore not easily supported. Data-level parallelism is intrinsically well-supported by the GPU architecture and exploited by performing a single function on an entire burst in parallel, per kernel launch. This approach was followed for the implementation

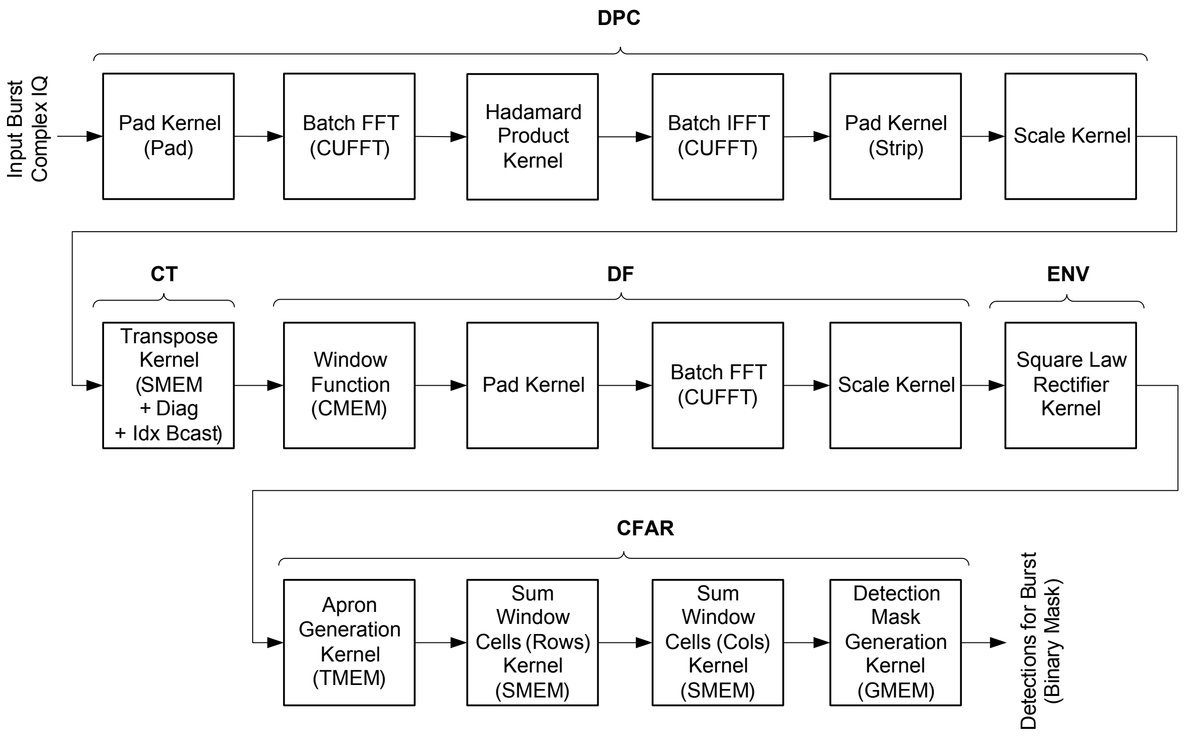
DPC
Input Burst Complex IQ
Pad Kernel (Pad)
Batch FFT (CUFFT)
Hadamard Product Kernel
Batch IFFT (CUFFT)
Pad Kernel (Strip)
Scale Kernel
CT
DF
ENV
Transpose Kernel (SMEM + Diag + Idx Bcast)
Window Function (CMEM)
Pad Kernel
Batch FFT (CUFFT)
Scale Kernel
Square Law Rectifier Kernel
CFAR
Apron Generation Kernel (TMEM)
Sum Window Cells (Rows) Kernel (SMEM)
Sum Window Cells (Cols) Kernel (SMEM)
Detection Mask Generation Kernel (GMEM)
Detections for Burst (Binary Mask)

Figure 5.13: Radar signal processing chain showing all kernels that are executed in sequence.

of the individual radar signal processing functions. The approach is aimed at maximizing kernel occupancy and the overall buffer size, which was identified as important factors for good performance in the microbenchmark in Section 6.2.2.

Data transfer rates between the host and device is slow, especially for small transfer sizes, as illustrated in the microbenchmark in Section 6.2.3. Consequently, data transfer between the host and device is minimized, which is also in accordance with general recommendations [32]. Furthermore, complete bursts are transferred in order to maximize the transfer size for better performance. The input to the radar signal processing chain is a stream of complex IQ samples from the receiver, which is transferred from the host to the device, once an entire burst is received. The radar signal processing chain then processes the burst from start to finish in device memory using GPU kernels, without transferring intermediate data back to the host. The output of the radar signal processing chain is a binary detection mask, that indicates the cell positions where targets are detected for the entire burst, which is the result that are transferred back from the device to the host.

Overlap of HtoD and DtoH transfers are not supported by the Tesla C1060 GPU, which only provides a single copy engine. Nonetheless, the overlap of either HtoD or DtoH transfers with kernel execution is supported and can be used to create a pipelined structure. A number of GPU implementations for real-time audio convolution in Chapter 4 effectively used pipelining techniques to improve performance. The audio convolution implementations all used pinned memory with asynchronous transfers and multiple CUDA streams to achieve pipelining. The use of mapped pinned memory is another technique for achieving pipelining by overlapping host and device data transfer with kernel execution, where kernels access host memory directly. These techniques were investigated in the microbenchmark in Section 6.2.4. A discussion on how these techniques, that are considered as mutually exclusive optimization options, were implemented to optimize the radar signal processing chain follows.

5.5.6.1 Optimization for Pinned memory with Asynchronous Transfers

Asynchronous HtoD and DtoH transfers require pinned host memory and may overlap with computation in a different CUDA stream, as described in Section 3.6.7. A CUDA stream is an abstraction in the CUDA API that represents a set of operations that are performed sequentially, where operations in different streams are considered independent and may overlap. Certain libraries provide functions to specify a stream ID for subsequent library function calls to use, such as CUFFT that provides the `cufftSetStream` function. This allows library operations to also execute in a desired stream and achieve pipelining, where the default NULL stream that is used otherwise does not allow for any concurrency.

A pipelined structure was implemented for the radar signal processing chain where 2 CUDA streams are used to provide the desired 2-way overlap between data transfer and execution, as shown in Figure 5.14. All operations for a burst are assigned to a single stream due to the direct data dependencies between consecutive functions, with consecutive bursts assigned to alternating streams. The global order that operations are issued among streams is also of importance to achieve pipelining, as the copy engine executes data transfer commands in the overall order that they are issued. Explicit, asynchronous HtoD and DtoH transfers are performed to transfer the input and output burst data, respectively. A double buffering scheme was implemented for device memory as each CUDA stream processes an independent burst in parallel and requires its own set of buffers.

Overlap is achieved inter-burst between consecutive bursts, once the pipeline is filled. Due to computational intensity of the processing chain and the design goal to minimize data transfer, the compute duration is expected to exceed the data transfer time considerably, which is represented by case 1

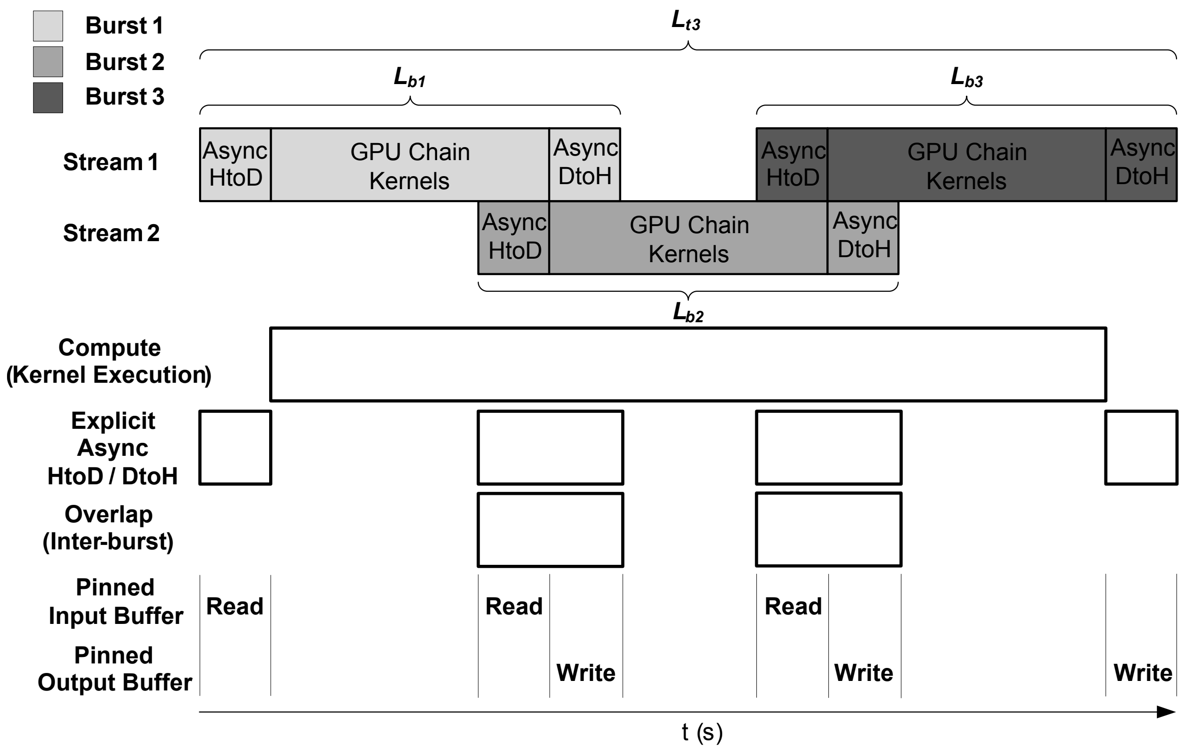

Figure 5.14: Pipelined structure for radar signal processing chain using pinned memory with asynchronous transfers shown for 3 consecutive bursts.

as shown in Figure 6.6 for the microbenchmark in Section 6.2.4. Based on the conclusions of this microbenchmark the total throughput is expected to increase as a function of the exact compute to data transfer ratio, with the average burst latency expected to be equivalent to using a single CUDA stream without overlap.

5.5.6.2 Optimization for Mapped Pinned Memory

Mapped pinned host memory allocation removes the need to perform explicit memory copies between the host and device, as discussed in Section 3.6.3.1. Kernels can read and write mapped host memory directly over the PCIe bus, which allows for implicit overlap between kernel execution and data transfer. Mapped pinned memory is generally only recommended where data is read or written once by a kernel. Therefore, mapped pinned memory may be considered appropriate for reading the input data and writing the output data for the radar signal processing chain.

The input buffer is read by the pad kernel and the output buffer is written by the detection mask generation kernel, as shown in Figure 5.13. These buffers were allocated as mapped pinned host memory buffers, without any modifications required to the kernels. The pad kernel then performs implicit HtoD transfer during execution as it reads directly from the input mapped host buffer and writes to the padded device memory buffer, as shown in Figure 5.15. Similarly, the detection mask generation kernel reads from device memory buffers and writes the detection results directly to the output mapped host buffer, performing an implicit DtoH transfer. Multiple CUDA streams are not required, or useful, as the overlap that is achieved is within each burst, instead of between bursts, which was the case with the asynchronous transfer scheme. No double buffering scheme is therefore required either.

Overlap is achieved intra-burst during the pad kernel and detection mask generation kernel execution. Again, the compute duration is expected to exceed the data transfer time considerably. Based on the microbenchmark results in Section 6.2.4 the total throughput is expected to increase and the average burst latency is also expected to decrease, both as a function of the exact compute to data transfer ratio. The expected decrease in latency is due to the overlapping of data transfer and kernel execution within each burst.

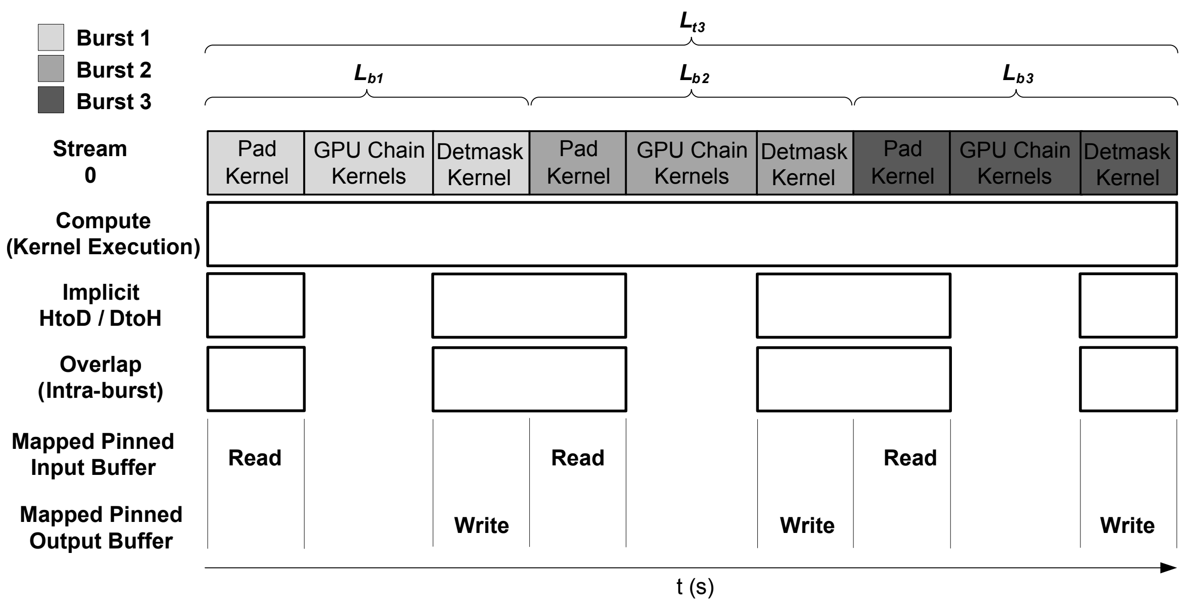

Figure 5.15: Pipelined structure for radar signal processing chain using mapped pinned memory shown for 3 consecutive bursts.

5.6 CONCLUSION

The individual radar signal processing functions were implemented on the target Tesla C1060 architecture using CUDA, based on findings in the literature and considerations for the radar application, where appropriate. Multiple variants were implemented to allow the most relevant algorithmic and architectural implementation options to be compared. The low-level metrics that were defined were subsequently derived for the GPU kernels that were implemented and used to perform initial analysis and comparison. A radar signal processing chain was constructed using the most appropriate variants for the radar signal processing application, based on initial results. The radar signal processing chain was then holistically optimized further, to better suit the requirements for the radar application. Experimental validation on the target hardware platform is required to evaluate the actual performance on the target architecture using experimental results, and to further characterize performance that can be achieved under realistic conditions for the radar application. All implementations were performed using the GBF building blocks, which subsequently allows for easy data gathering, functional verification and general experimentation.

CHAPTER 6

EXPERIMENTAL RESULTS

6.1 INTRODUCTION

The experimental validation that was performed is described in this section. Results were gathered on the hardware platform for the GPU implementations using the GBF. The high-level metrics that have been defined are used to measure the system-level performance of the radar signal processing functions and radar signal processing chain. The low-level metrics that have been defined are used to measure the kernel-level performance on the GPU architecture for custom kernels that were implemented. A set of microbenchmarks were first performed, followed by benchmarking of the individual radar signal processing functions and concluding with the benchmarking of the complete radar signal processing chain.

6.2 MICROBENCHMARKS

A number of microbenchmarks were performed in order to characterize the performance of fundamental architectural features on the Tesla C1060 GPU.

6.2.1 Device Computational Performance

A microbenchmark was developed to investigate the floating-point computational performance of the device. It is important to characterize the computational performance of the device in order to understand the conditions that are required for optimal performance and to be able to identify when a kernel is compute-bound.

For this benchmark a series of kernels were developed where each kernel performs one type of primitive mathematical operation exclusively, as summarized in Table 6.1. The `cuobjdump` tool was used to extract the intermediate PTX and SASS microcode instructions from the compiled kernels. The extracted code was analyzed in order to determine the exact microcode instructions that the high-level kernels compile into. The peak theoretical throughput for each of the primitive math operations was calculated for the Tesla C1060 using a core clock rate of 1296 MHz with 30 SMs, along with a FLOP count for each operation and the corresponding native arithmetic instruction throughput from the CUDA C Programming Guide [32], by multiplying the factors.

Table 6.1: Primitive math operations used for the microbenchmark.

Floating-point Operation	Instructions	FLOPs	Peak Theoretical Throughput (GFLOPS)
Add	FADD	1	311
Multiply	FMUL	1	311[†]
Multiply-add	FMAD	2	622
Divide	RCP + FMUL	1	78[††]

[†]Based on SP throughput alone (excl. SFU contribution).
[††]Based on assumption that FMUL (on SPs) is hidden by RCP (on SFUs) with lower instruction throughput.

The throughput for native arithmetic instructions are summarized and grouped according to the units within each SM that executes the instructions in Table 6.2. The SP and SFU units can execute independent instructions in parallel and both, in principle, be kept fully occupied by the warp scheduler under ideal conditions [31]. The peak theoretical throughput for divide in Table 6.1 is therefore based purely on the throughput of the RCP operation that executes on the SFUs at a much lower instruction

Table 6.2: Single-precision floating-point microcode instructions used for the microbenchmark.

Opcode	Description	SPs Throughput[††]	SFUs Throughput[††]
FADD	Add	8	-
FMUL	Multiply	8	Not documented clearly
FMAD[†]	Multiply-add	8	-
RCP	Reciprocal	-	2

[†]Not to be confused with single-precision fused multiply-add (FFMA), which is N/A on CC1.3.
[††]Instructions per clock cycle per SM.

throughput (1:4 ratio), as it is assumed that the FMUL operation that executes on the SPs will be hidden due to the overlap in computation. The SPs are the MAD units that can perform multiply, add and multiply-add operations natively. The SFU unit can perform transcendental functions and the floating-point reciprocal function that is used for floating-point division.

Note that each SFU is also known to contain up to four floating-point multipliers, according to certain sources [31], that is used for interpolation in texturing operations. Indications are that the additional multipliers in the SFUs can also be used for general floating-point multiplication (FMUL) in CUDA kernels, when the SFU is otherwise idle. However, the exact instruction throughput that can be expected for pure multiplication from the SFU alone is not documented clearly in the CUDA documentation. The peak theoretical throughput for multiply in Table 6.1 is therefore based purely on the throughput on the SPs, without any potential contribution from the SFU. The overall peak theoretical throughput for the Tesla C1060 is widely reported as 933 GFLOPS [60], which is said to be based on the peak FMAD rate of 622 GFLOPS plus the overlapped computation from the SFU. Again, the exact calculation of the SFU contribution is not widely known.

For the microbenchmark kernels each thread reads and writes a single 4 byte float word and the number of iterations of the math operation on the data word is made variable through a loop, effectively allowing the arithmetic intensity (FLOPs/word) to vary. This inner loop is fully unrolled in all cases in order to exclude any loop overhead that constitutes non-floating-point operations. Kernel occupancy is kept at 100% with a block size of 256 and all memory transactions are fully coalesced in order to attempt to isolate computational effects. An overall buffer size of 32 MiB is used for all runs. The microcode was also inspected to ensure that compiler optimizations had not inadvertently removed code explicitly included for the purposes of the artificial microbenchmark.

Figure 6.1 shows the theoretical peak throughput that was calculated for the Tesla C1060 along with the results for each kernel that performs a primitive operation. For all primitive operations 50% of the peak theoretical throughput is reached with an arithmetic intensity of 32 FLOPs/word. To obtain 85% or more of the peak theoretical throughput an arithmetic intensity of at least 256 FLOPs/word is required. For small arithmetic intensity values the throughput is shown to be unaffected by the operation type within a particular category that executes on the SPs or SFUs respectively. On closer inspection the timing results show a constant-time response in this region. The results therefore show that for workloads that execute on the SPs (FADD / FMUL / FMAD) the performance constraint transitions at an arithmetic intensity of 16 FLOPs/word. For arithmetic intensity below this point the kernels are

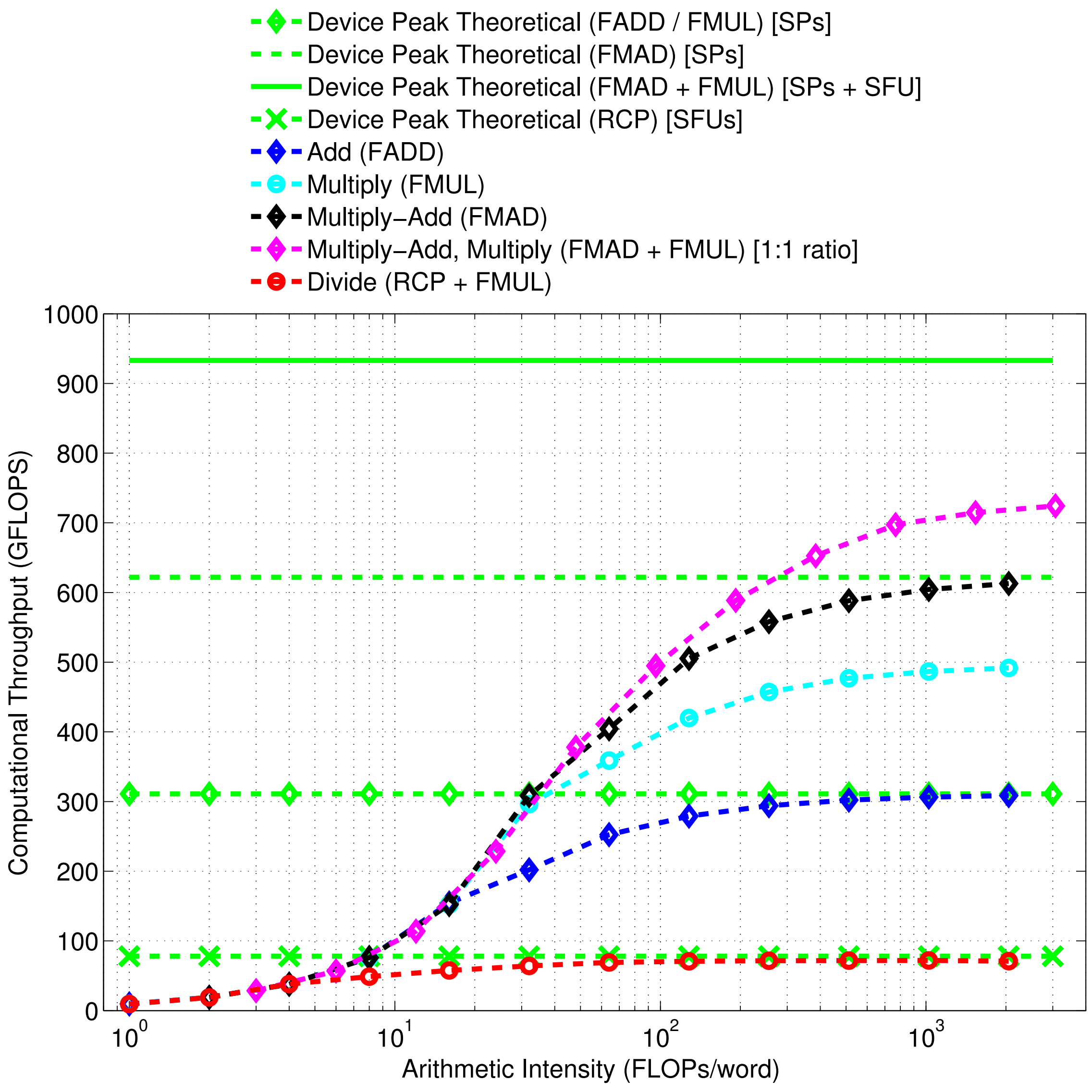

Figure 6.1: Single-precision floating-point performance of primitive math operators as a function of arithmetic intensity.

bound by global memory bandwidth plus instruction overhead to perform indexing, for instance. For arithmetic intensity above this point the kernels are bound by the primary computation. For workloads that execute on the SFUs (RCP) the same is observed with an arithmetic intensity of 4 FLOPs/word as the transition point. The difference in transition points is attributed to the 1:4 ratio in instruction throughput for the SP and SFU instructions that were utilized, as shown in Table 6.2.

For the pure multiply kernel, which uses the FMUL operation, the observed throughput is also shown to greatly exceed the peak theoretical throughput that is expected for the SPs alone by nearly 200 GFLOPS. Therefore, this additional performance has to be attributed to the additional multipliers in

SFUs for an approximate effective instruction throughput per clock cycle per SM of 5 in addition to the 8 provided by the SPs alone. Subsequently an additional kernel was developed, which is also shown, where independent multiply-add and multiply operations were interleaved at a ratio of 1 : 1 in an attempt to obtain maximum overall performance by overlapping FMAD computation on the SPs with FMUL computation on the SFUs.

The performance obtained with this kernel shows an improvement of around 120 GFLOPS obtained over both the performance and peak theoretical throughput for the pure multiply-add kernel. This result again indicates that overlapped FMUL computation on the SFUs contributed to a greater overall throughput. However, the maximum performance that could be obtained with this kernel is still only around 78% of the claimed overall computational performance for the Tesla C1060 and the net contribution attributed to the SFUs is lower than observed with the pure multiply kernel. This diminished improvement is attributed to scheduling flexibility, which causes a sensitivity to the FMAD to FMUL instruction ratio, as the FMAD instruction can only be executed on the SPs, where the FMUL instruction can be executed on the SPs and the SFUs.

6.2.2 Device Global Memory Bandwidth

Device global memory bandwidth is vital to maximizing kernel performance as it serves as the primary method for kernels to access device memory [9]. However, there are several factors that affect performance.

A microbenchmark was developed to characterize the device global memory bandwidth by allowing a number of important factors to vary. A series of copy kernels were developed which read and write a single word from global memory, but perform no explicit computation apart from index calculation. The basic copy kernel performs fully coalesced global memory transfers by reading from an input buffer and a writing to an output buffer. An offset copy kernel and a stride copy kernel were also developed based on the copy kernel, but take an additional argument to respectively specify an offset and stride in words, that is applied to read and write operations. The microbenchmark supports float (4 bytes) and float2 (8 bytes) words, which may be used to represent real and complex samples for the radar application. The microbenchmark also allows the block size to be specified in number of threads per block and the overall buffer size which is representative of a burst.

Figure 6.2 shows results that were generated using the copy kernel by varying the overall buffer size and block size respectively for fully coalesced transfers. The maximum global memory bandwidth that is achieved is around 82 GB/s which is 80% of the peak theoretical bandwidth for the device. The results also indicate that a block size of at least 128 is required for optimal occupancy and memory latency hiding to reach the peak achievable bandwidth.

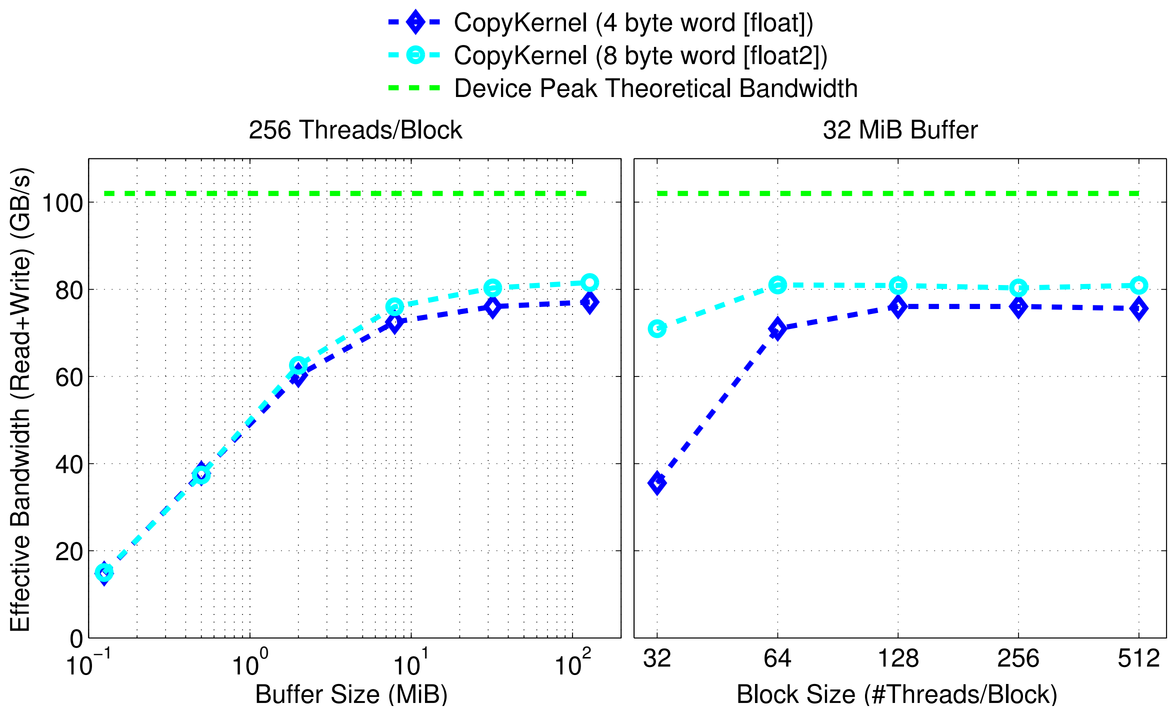

Figure 6.2: Effect of buffer size and thread block size on global memory performance with fully coalesced transfers for a simple copy kernel.

Global memory coalescing is an important factor in achieving high performance global memory access, as discussed in Section 3.6.3.2. The coalescing requirements and protocol is provided in the CUDA C Programming Guide [32] and the directly relevant aspects for CC 1.3 is summarized in the rest of this paragraph. Memory transactions are serviced per half-warp, which is 16 threads. An initial segment size of 128 bytes is used for 4 and 8 byte words. The memory segment for the requested address by the active thread with the lowest thread ID is identified, after which all other active threads with a requested address that falls in the same segment is identified. The transaction size is then reduced to 64 bytes and further reduced to 32 bytes in each case, only if either the upper or the lower half of the transaction is used. The transaction is then performed and all serviced threads are marked as inactive. This process is repeated until all threads in a half warp are serviced.

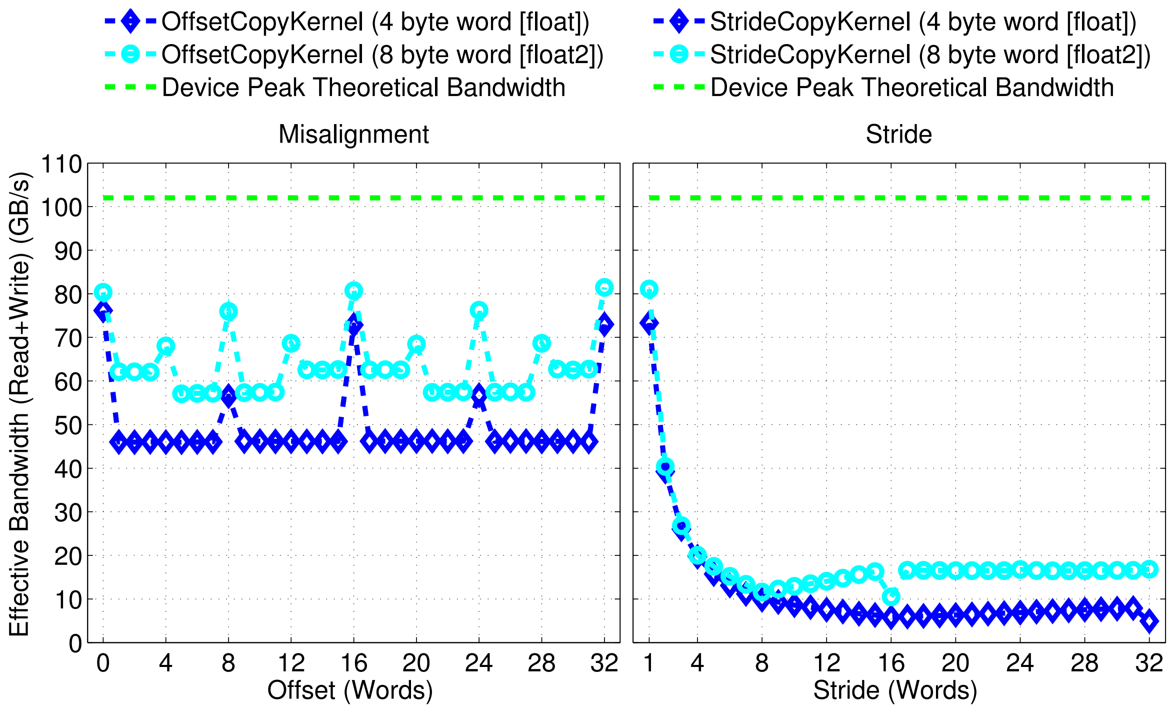

Figure 6.3: Effect of misaligned and strided access on global memory performance due to uncoalesced transfers for simple offset and stride copy kernels with optimal occupancy of 1 using a 32 MiB buffer and 256 threads/block.

Figure 6.3 shows results that were generated using the offset copy kernel and stride copy kernel by varying the offset and stride respectively, in order to investigate the causes and effects of uncoalesced memory transfers. Non-sequential access was not investigated as it is not a requirement for coalescing on devices with CC 1.3, but was however required on older devices with CC 1.0 and 1.1. A 32 MiB buffer was used with a block size of 256 based on the results shown in Figure 6.2, in order to operate in a region where fully coalesced transfers are known to reach the maximum achievable bandwidth. For the stride copy kernel with a stride of 1, all 16 threads in the half-warp are serviced by a single 128 byte transaction. For each increment of the stride an additional adjacent 128 byte transaction is required. The transaction size cannot be reduced, as the requested addresses are evenly spread throughout the segment. It is therefore expected that the global memory throughput decreases by a factor equal to the stride, up to a maximum factor of 16, where a separate transaction is performed for each thread in the half-warp, which is observed in the results.

6.2.3 Host/Device Bandwidth

A microbenchmark was developed to characterize the bandwidth between the host and device for a variety of host memory allocation options that are made available through the CUDA API, as discussed in Section 3.6.3.1. The bandwidth between the host and device is typically the lowest in the GPU memory hierarchy and can potentially be a performance bottleneck. Accordingly, it is important to characterize the bandwidth in order to understand the limitations and how to optimize the performance.

The microbenchmark performs a series of separate, non-overlapping host to device (HtoD) and device to host (DtoH) transfers for a specified transfer size using `cudaMemcpy`. The microbenchmark supports the pageable, pinned and pinned (write-combined) host memory allocation options. For each combination of parameters the microbenchmark was set to perform the copy operation 5 times consecutively in an inner loop with 10 independent runs and the results were averaged. Host memory can also be allocated as portable, although this option was not included in the benchmark as it is primarily useful in multi-GPU systems, which is not under consideration.

Figure 6.4 shows the results that were generated using the microbenchmark by varying the transfer size from 128 KiB to 512 MiB for HtoD and DtoH transfers respectively. The results indicate that pageable memory performs considerably worse than the pinned memory across the entire range, which is expected due to the additional overhead of transferring data via staging buffers, as described in Section 3.6.3.1. The two types of pinned memory allocation performed nearly identical across the range of transfer sizes, which is expected from a system without a front-side bus, such as the host system, as described in Section 3.6.3.1. The maximum throughput that was achieved for HtoD transfers is 6 GB/s and DtoH transfers is 5.6 GB/s, which is approximately 75% and 70% respectively of the peak theoretical throughput for the PCIe bus that connects the device and the host. The CUDA C Best Practices Guide [33] indicates that pinned memory allocation with the PCIe bus version and lane width that was used should be able to exceed 5 GB/s, which was achieved.

The results also show that the bandwidth is lower for small transfers, which is a known effect [33] and which is expected due to the underlying DMA setup and execution overhead per transfer. Assuming that one burst is copied between the host and device in a single transfer, Table 6.3 shows a summary of the results obtained as a percentage of peak achievable bandwidth with equivalent burst sizes. A very small burst with around 8192 complex samples arranged, for instance, as 16 pulses by 512 range bins,

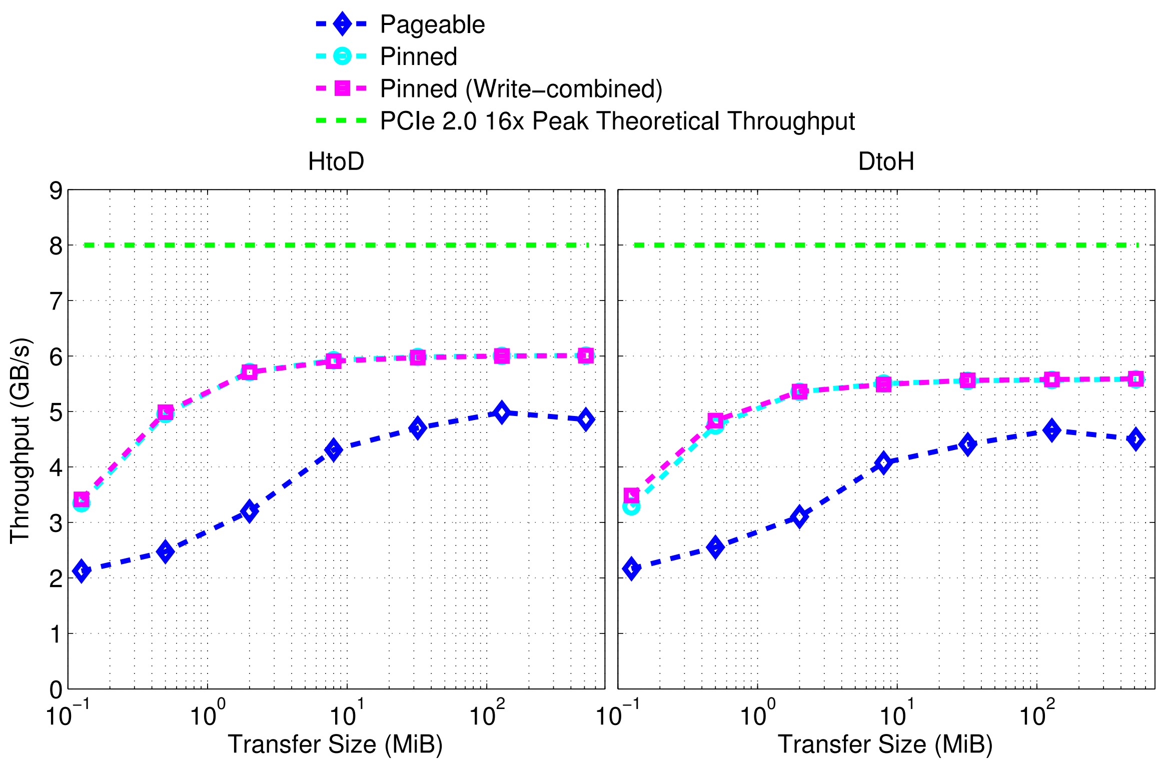

Figure 6.4: Data transfer performance between host and device in both transfer directions for different host memory allocation options and transfer sizes using `cudaMemcpy` function.

will achieve just over half of the peak achievable bandwidth when transferred between the device and host. A larger burst size of around 262144 complex samples arranged, for instance, as 32 pulses by 8192 range bins, is required to achieve more than 95% of the peak achievable bandwidth. Therefore, smaller bursts will experience reduced throughput and increased latency, which is undesirable for the radar application under consideration.

Table 6.3: Summary of results in Figure 6.4 for microbenchmark with equivalent burst sizes for the transfer size.

% of Peak Achievable BW	**Transfer Size**	**Burst Size (Samples) Real [float]**	**Burst Size (Samples) Complex [float2]**
55%	128 KiB	16384	8192
75%	512 KiB	131072	65536
85%	1 MiB	262144	131072
95%	2 MiB	524288	262144

Based on the results it is recommended that pinned memory be used whenever possible, only to be exchanged for pageable memory when there is a risk of reaching physical memory limits on the host system. It is also recommended that pinned (write-combined) memory is avoided due to the lack of a relevant performance increase, as well as the additional pitfalls associated with regard to CPU read performance, as discussed in Section 3.6.3.1. Note that on other systems the architecture, performance and memory limits may vary and similar benchmarks may need to be repeated in order to discover the optimal host memory allocation options. It is also recommended that data transfer between the host and device should also be minimized, due to the comparatively low bandwidth that it provides, by keeping data resident on the device, which is also considered a common practice [33].

Batching of small transfers into a larger transfer is generally recommended [33] to increase bandwidth, due to the decreased performance for small transfers. However, buffering of bursts will further increase burst latency and also add further complexity and overhead in extracting and processing multiple bursts from a single buffer. CUDA does, however, provide features for concurrent host/device data transfer and kernel execution, which can potentially be used to improve overall performance, given the performance limitations of host/device data transfer.

6.2.4 Concurrent Host/Device Data Transfer and Kernel Execution

The Tesla C1060 supports asynchronous data transfers as it features a dedicated copy engine, which allows for concurrent host/device data transfer and kernel execution for operations in different streams, as described in Section 3.6.7. Some newer devices also provide two dedicated copy engines, which also allows HtoD transfers to overlap with DtoH transfers, but this feature is not available on the Tesla C1060. Mapped pinned host memory allocation is another method that allows for concurrent host/device data transfer and kernel execution, as described in Section 3.6.3.1.

Unlike the asynchronous transfer method which performs explicit copies, kernels can access the mapped host buffers directly and data is transferred between the host and device implicitly. Another difference when using mapped pinned memory is that the kernel execution automatically overlaps with kernel-originated data transfer without explicit use of streams [33]. Also, for the approaches that require explicit data transfers, all intra-stream operations can only execute once the preceding operation is complete, with concurrency only supported inter-stream. Consecutive bursts are independent and may be processed in different streams, although the data transfer to and from the device and kernel execution for a given burst is dependent and must occur within the same stream.

We can therefore expect to potentially achieve better latency using mapped pinned memory. However, with mapped pinned memory it is more important to ensure that data transfers are coalesced as the penalty for uncoalesced transfers is greater than the normal penalty associated with device memory access. It also important to ensure that mapped memory is only read or written once by the kernel, as the host/device bandwidth is much lower than the bandwidth provided by device memory, which is otherwise accessed by kernels when not using mapped memory.

A microbenchmark was developed to characterize the performance that can be achieved using concurrent host/device data transfer and kernel execution. The microbenchmark supports the pageable, pinned and mapped pinned host memory allocation options. The number of CUDA streams is also adjustable where a stream count greater than 1 enables overlapped data transfer and kernel execution for the pinned option, where the mapped pinned option always implicitly performs overlap. Pinned (write-combined) memory was not considered due to the findings presented in Section 6.2.3. The pageable option does not support any concurrent data transfer and execution, but was included as a potential worst-case scenario that indicates how a naïve implementation may perform, as pageable memory allocation is considered the default memory allocation option.

Furthermore, the multiply kernel which was developed for the device computational throughput microbenchmark in Section 6.2.1 was reused for this microbenchmark as it effectively allows us to vary the compute to data transfer ratio by varying the inner loop count in the kernel. The ratio between computation and data transfer is an important factor to consider as it determines the extent of the overlap that may occur, as illustrated in Figures 6.5, 6.6, and 6.7. The microbenchmark allows a specified number of buffers to be processed where each buffer is considered a burst. The total latency

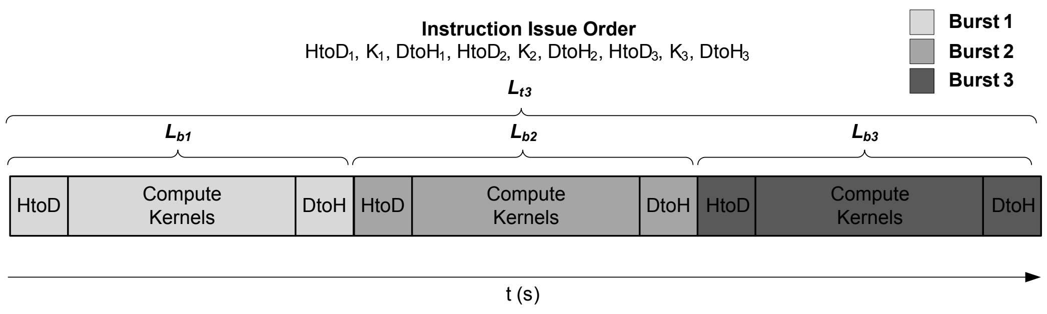

Figure 6.5: Illustration of serial data transfer and kernel execution, using Pinned (Without Overlap) scheme, for 3 consecutive bursts.

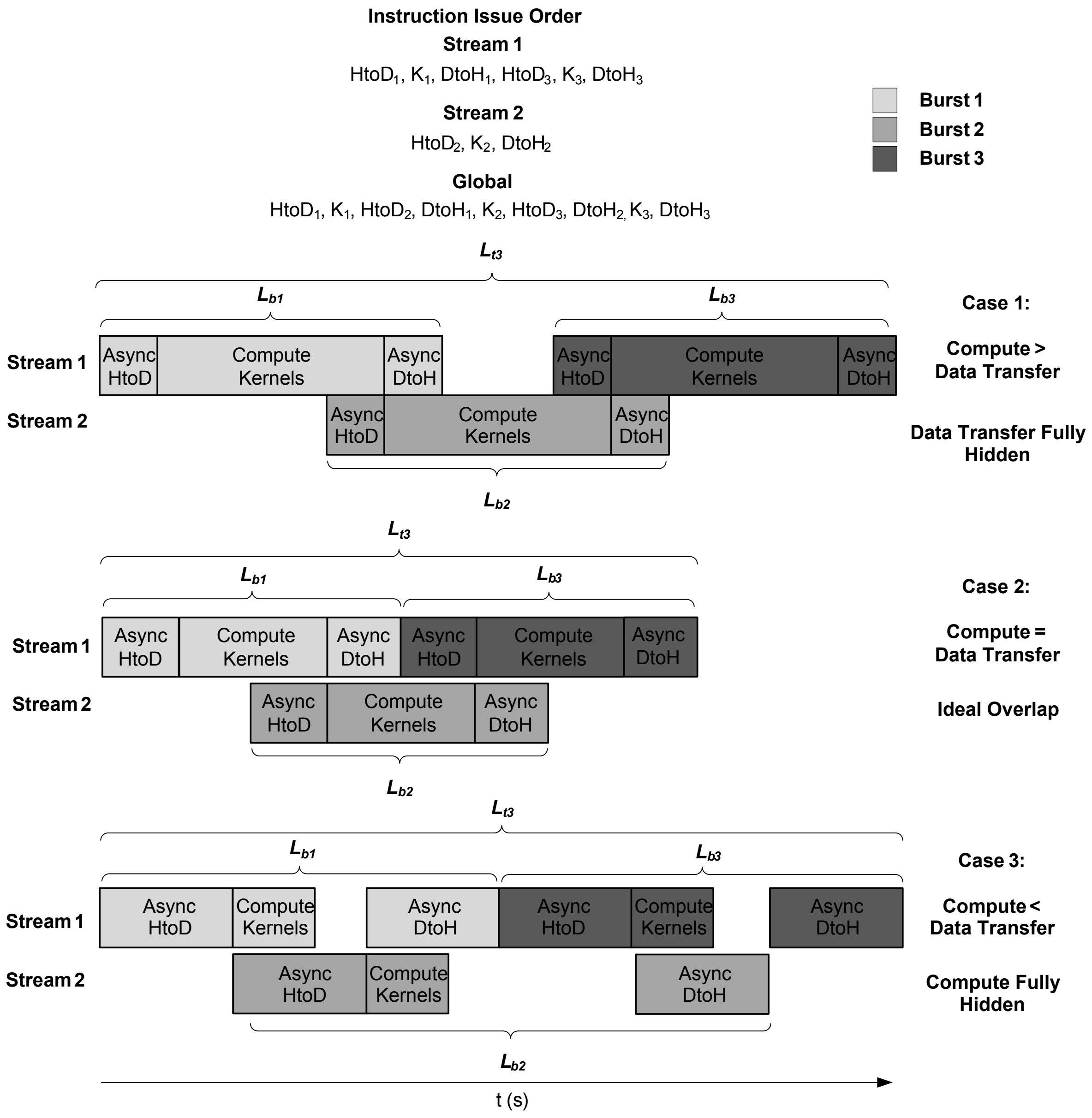

Figure 6.6: Illustration of concurrent data transfer and kernel execution, using Pinned (Overlap) with CUDA streams and asynchronous transfers scheme, for 3 consecutive bursts.

L_{tn}, burst latency L_{bi} and kernel latency L_{ki} is measured and the total throughput T_{tn}, average burst latency $\overline{L}_{bn}$ and average kernel latency $\overline{L}_{kn}$ is subsequently derived as defined in Table 5.3.

Figure 6.8 shows the results that were obtained for the microbenchmark by varying the compute to data transfer ratio for all supported host memory allocation options and combinations by processing 8 consecutive bursts. A total of 10 independent runs were performed and the results were averaged. The y-axis was normalized to the best-case performance without overlap, namely the pinned option

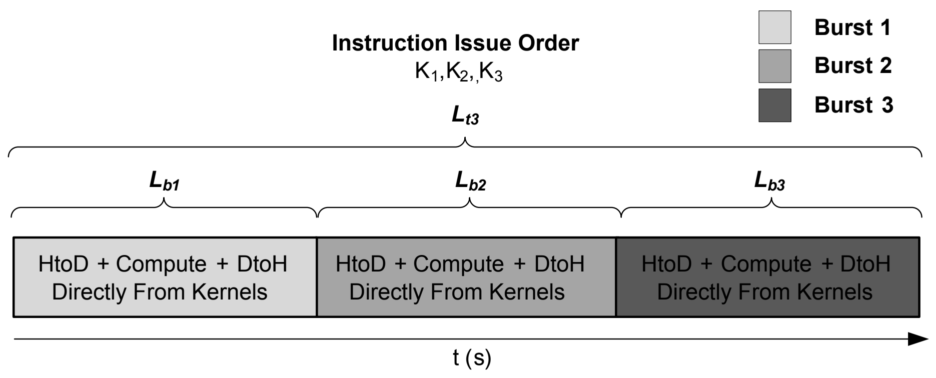
Instruction Issue Order
K1,K2,K3
Burst 1
Burst 2
Burst 3
Lt3
Lb1
Lb2
Lb3
HtoD + Compute + DtoH Directly From Kernels
HtoD + Compute + DtoH Directly From Kernels
HtoD + Compute + DtoH Directly From Kernels
t (s)

Figure 6.7: Illustration of concurrent data transfer and kernel execution, using Mapped Pinned (Overlap) scheme, for 3 consecutive bursts.

that is considered the base case, in order to clearly show any improvement or degradation. The x-axis was normalized to the ratio between compute and data transfer in terms of duration, using the average kernel latency and the average burst latency for the base case.

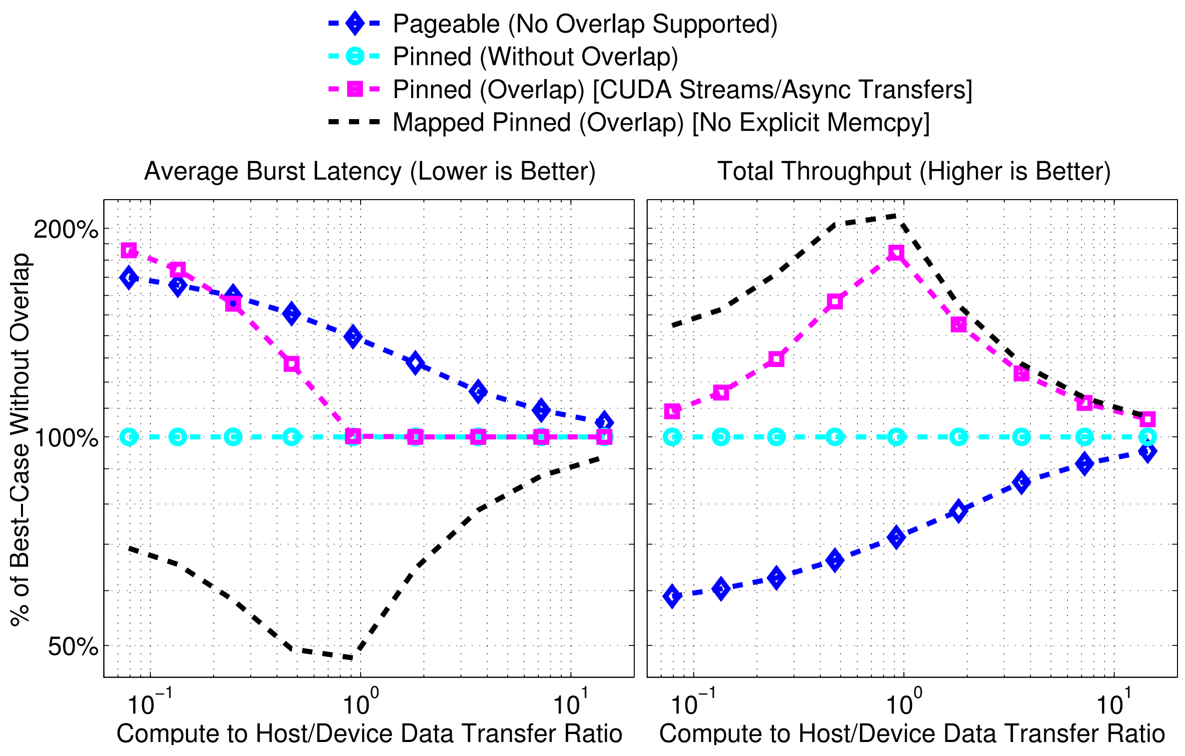

Figure 6.8: Effect of different concurrent data transfer and kernel execution schemes on latency and throughput for 8 consecutive bursts.

For the pinned (overlap) option the results show an overall improvement in throughput that is roughly proportional to the ratio of the smaller component to the larger component with respect to data transfer and execution. This is due to the smaller component being fully hidden by the larger, with performance nearly doubling when the components are equal due to ideal overlap, as illustrated in Figure 6.6. However, the results show that the latency that is achieved using the pinned (overlap) option shows no improvement, but rather significant deterioration, which approaches double the base case latency, when computation is the smaller component. If we review the representative case 3 in Figure 6.6 we observe that L_{b1} represents the burst latency with a filled pipeline. L_{b1} is constrained to a minimum latency equivalent to the data transfers for two bursts, hence the decay to double the base case latency as the computation component decreases. The constraint is imposed by the global instruction issue order, shown in Figure 6.6, that is required in order to achieve overlap between consecutive bursts.

For the mapped pinned option the results show the best overall improvement with an improvement slightly in excess of double the base case performance for equal compute and data transfer components. The excess is attributed to a reduction in overhead for the explicit memory copies that are not required, as an improvement of more than double is not realistically expected for this experiment, since it would imply an improvement beyond what ideal overlapping alone can achieve.

6.3 RADAR SIGNAL PROCESSING FUNCTIONS

The results for the individual radar signal processing functions that were implemented are presented in this section.

6.3.1 Digital Pulse Compression (DPC)

The kernel throughput that is achieved for the TDFIR variants are shown in Figure 6.9. In general the TDFIR performance is a strong function of the filter length N, as can be expected due to the linear increase in the device memory workload and computational workload with a linear increase in N, as shown in Table 5.5. For very short N of around 64 taps or less, respectable performance in excess of 200 MS/s is achieved for the best performing variant. For the shortest N of 16 that was evaluated, impressive performance in excess of 1.2 GS/s is achieved. The results also show that the variants that use TMEM and SMEM for input easily outperform the variant that uses GMEM for reading input, due to improved coalescing and on-chip caching.

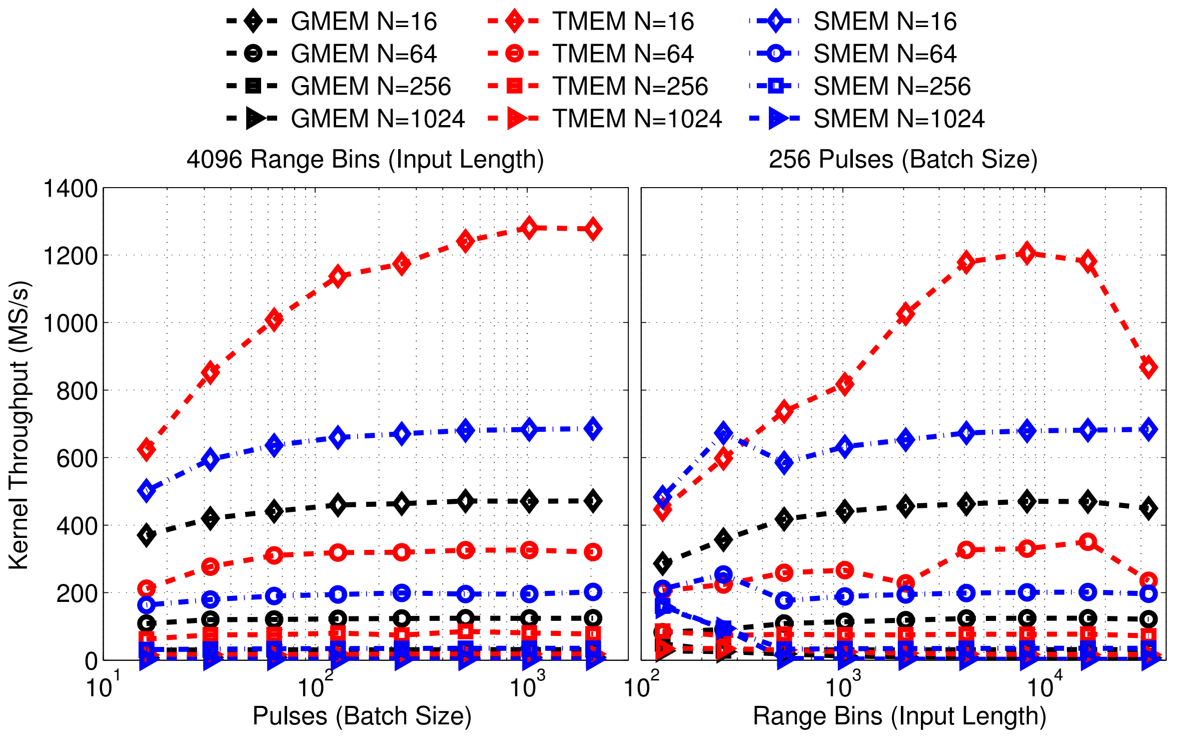

Figure 6.9: DPC function kernel throughput (T_k) for TDFIR variants for different filter lengths N and varying batch sizes and input lengths with fully unrolled inner loops.

As shown in Table 6.4, the GMEM variant achieves around 65 GFLOPS and 157% of the achievable bandwidth. The bandwidth achieved exceeds 100% due to const cache hits on the filter coefficients that are stored in CMEM, where the W_{dmem} is based on a worst-case figure with no cache hits, as shown in Table 5.6. The TMEM variant achieves around 180 GFLOPS and 441% of the achievable bandwidth, due to the implicit TMEM caching performing very well with the substantial input data reuse and high spatial locality. The high spatial locality is achieved due to the chosen thread block layout where a block maps to a section of a single input vector. The SMEM variant achieves around 103 GFLOPS and 254% of the achievable bandwidth as the explicit caching of input data for the SMEM implementation is also much more efficient than the pure GMEM variant, but is generally outperformed by the TMEM variant.

Firstly, the increased complexity of the SMEM kernel over the simple TMEM kernel adds additional instruction overhead on a kernel that is already expected to be compute bound. The SMEM kernel contains the outer loop to implement the multi-stage processing, additional instructions for index checking and multiple `__syncthreads()` synchronization instructions per stage due to the reuse of the shared memory buffer for multiple stages. Secondly, some redundant loads are still performed during the last stage with corresponding idle threads during the subsequent processing phase, as each thread always loads one sample per thread per stage. Lastly, the `warp_serialize` counter, that represents the number of shared memory bank conflicts in the NVIDIA Visual Profiler, was used to confirm that 2-way bank conflicts are indeed present as expected, which is another potential source of lower performance compared to the TMEM variant. The results that were achieved with the loop unrolling optimization also showed that the SMEM variant is limited by factors other than the inner loop overhead as almost no improvement was achieved. Conversely, for the TMEM variant the loop unrolling optimization improved performance by up to 50%.

The kernel throughput that is achieved for the FDFIR variants are shown in Figure 6.10. The only difference between the two variants is that the minimum required FFT length $M = F + N - 1$ is used in one case and the length is snapped to the next higher multiple of 256 and power of 2 in the other, according to the scheme described in Section 5.5.1.2. The scheme where the FFT length is snapped improves performance by a factor of up to 4 times. However, as M increases the amount of potentially wasted computation increases, with maximum wasted computation at the start of each snap section boundary.

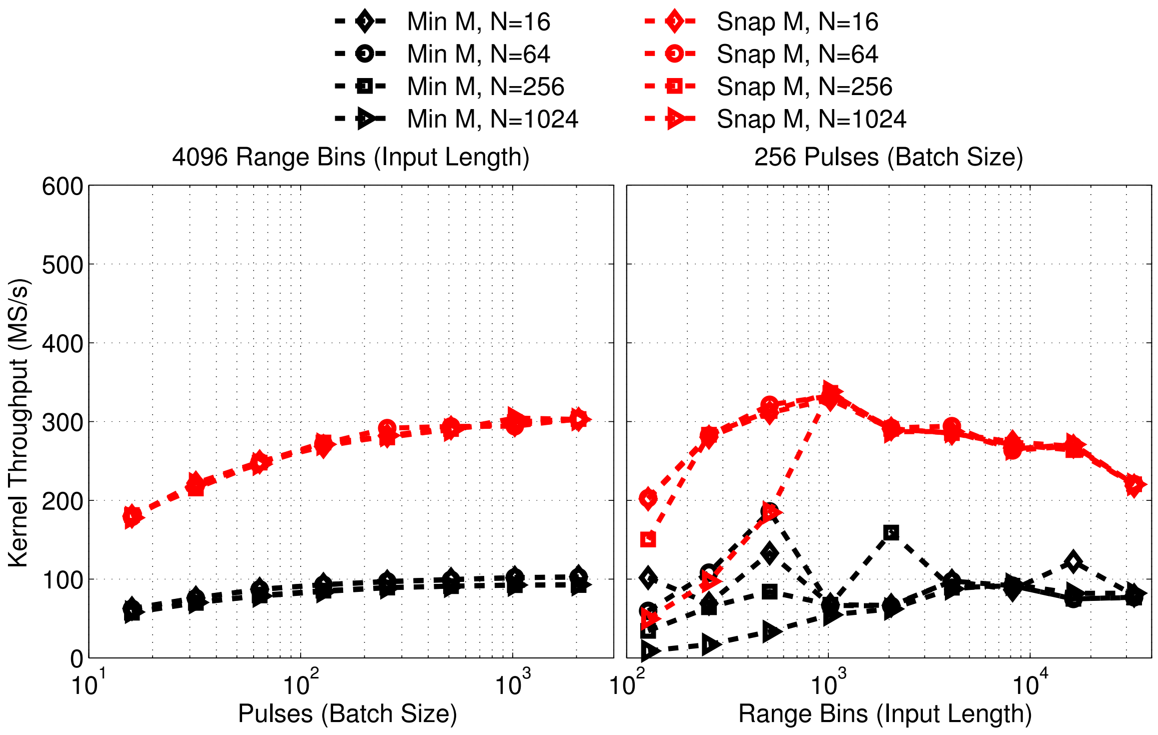

Figure 6.10: DPC function kernel throughput (T_k) for FDFIR variants with FFT length M for different filter lengths N and varying batch sizes and input lengths.

Furthermore, the performance of the FDFIR snap variant is not a strong function of N, as the FFT length is determined by the sum of the input length and N, where the input length is typically greater than N. However, a performance drop is expected as a function of increasing N when it causes this sum to fall within the next snap section. Note that for the FDFIR results the padded frequency-domain representation of the impulse response H is precomputed and, therefore, excluded from performance measurements. The FFT plan steps are also performed during initialization and therefore also excluded.

The pad kernel, Hadamard product kernel, and scale complex kernel are used for the FDFIR variant and discussed briefly, based on the measured low-level metrics in Table 6.4 and the derived low-level metrics in Table 5.6. The pad kernel, Hadamard product kernel and scale complex kernel achieve very close to the maximum achievable bandwidth. The pad kernel performs no computation and the Hadamard product kernel and scale kernel both have a constant, low AI per sample. All three kernels are therefore found to be bandwidth limited.

Table 6.4: Measured low-level metrics for individual DPC kernels.

DPC Kernel	$\lceil \mathbf{T_{eb}} \rceil$	**% of** $\lceil \mathbf{T_{eb}} \rceil$ **Achievable**[*]	$\lceil \mathbf{T_c} \rceil$[†]	**Limiting Factor**
TDFIR (GMEM)	129 GB/s	157%	64.7	BW
TDFIR (TMEM)	362 GB/s	441%	179.6	Inst. Throughput
TDFIR (SMEM)	208 GB/s	254%	103.3	BW / Inst. Throughput
Pad	78 GB/s	95%	N/A	BW
Hadamard Product	82 GB/s	100%	20.5	BW
Scale Complex	80 GB/s	98%	10.0	BW

[*]Relative to max achievable BW for device memory bandwidth microbenchmark results.
[†]Specified in GFLOPS.

Figure 6.11 shows a further set of results that were generated, where the best performing TDFIR and FDFIR variants are compared using realistic burst sizes that are varied over a broad range in both dimensions. For the DPC or matched filter function in a radar system the filter length N shall at least match the pulse width in samples, as the filter coefficients are the complex conjugate of the uncompressed transmit pulse. The results therefore show that for very short pulses of approximately 64 samples or less, the TDFIR implementation provides better performance across a wide range of

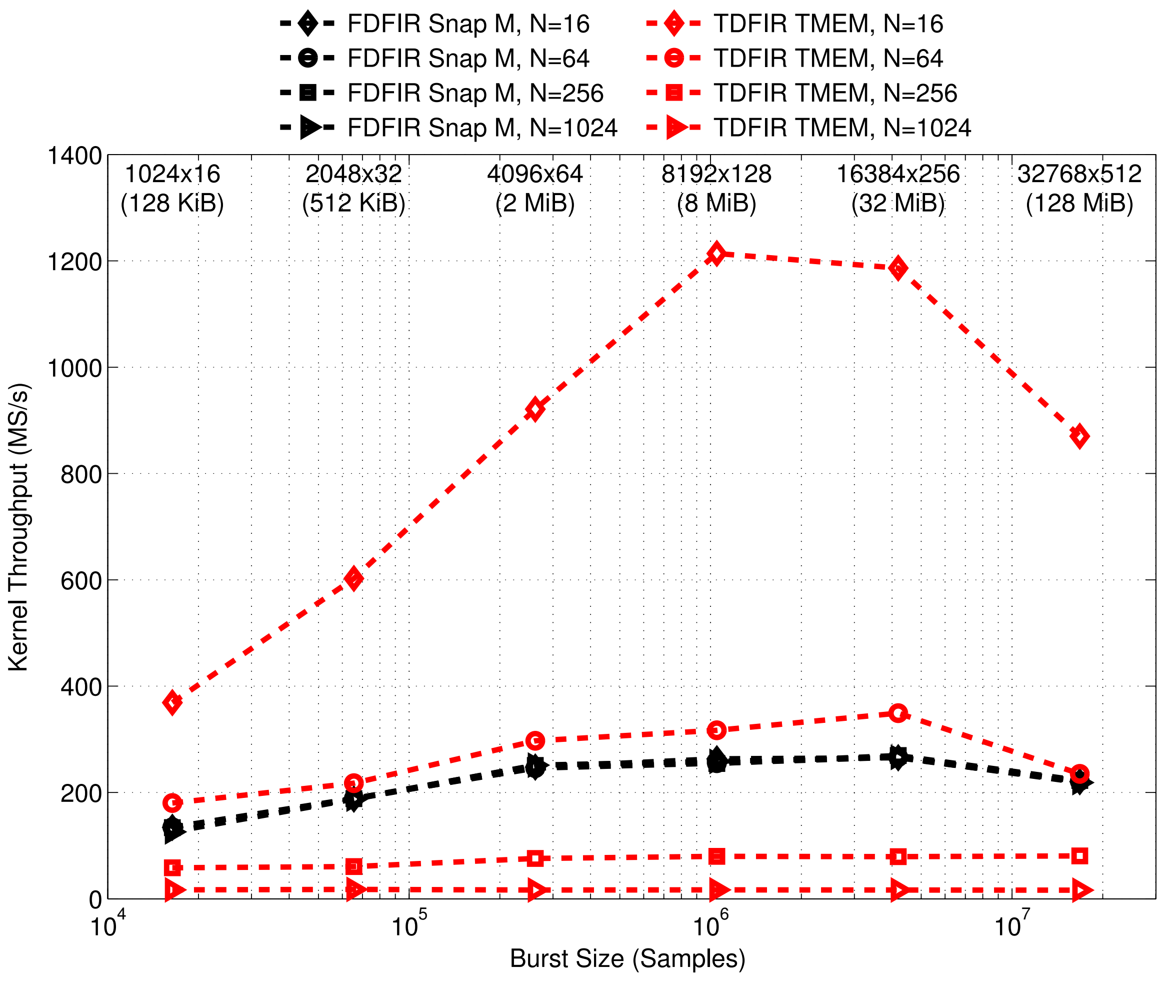

Figure 6.11: DPC function kernel throughput (T_k) for best performing TDFIR and FDFIR variants for different filter lengths N with realistic burst dimensions and sizes varied over a broad range showing labels for number of range bins by number of Doppler bins.

burst sizes. However, the FDFIR implementation provides very consistent performance across the entire range of burst sizes, and better performance than the TDFIR for pulse widths roughly in excess of 64 samples, which is typically the case. Furthermore, the current FDFIR implementation in principle allows for different pulse waveforms to be used throughout the burst as the H matrix contains a coefficient set for each pulse in the input burst. Conversely, the TDFIR implementation only supports a single filter coefficient set that is stored in CMEM, which is no longer a viable option at some point when multiple coefficient sets are required per burst. Therefore, the FDFIR is recommended in general for the radar application.

6.3.2 Corner Turning (CT)

For the CT benchmark the kernels discussed in Section 5.5.2 are evaluated over a broad range of burst sizes and compared to a pure global memory copy kernel as shown in Figure 6.12. The copy kernel also uses the same block size as the CT kernels and serves as an indication of absolute best-case expected performance, given that all the CT kernels perform an out-of-place transpose producing a transposed copy of the input data, whereas the copy kernel merely copies the data. Results for the GMEM, SMEM, and SMEM with diagonal block ordering kernels illustrate the improvements when the typical CT optimizations discussed in Section 4.3.3 are applied, showing the progression from a naïve to optimally coalesced and bank conflict-free to partition camping free implementation.

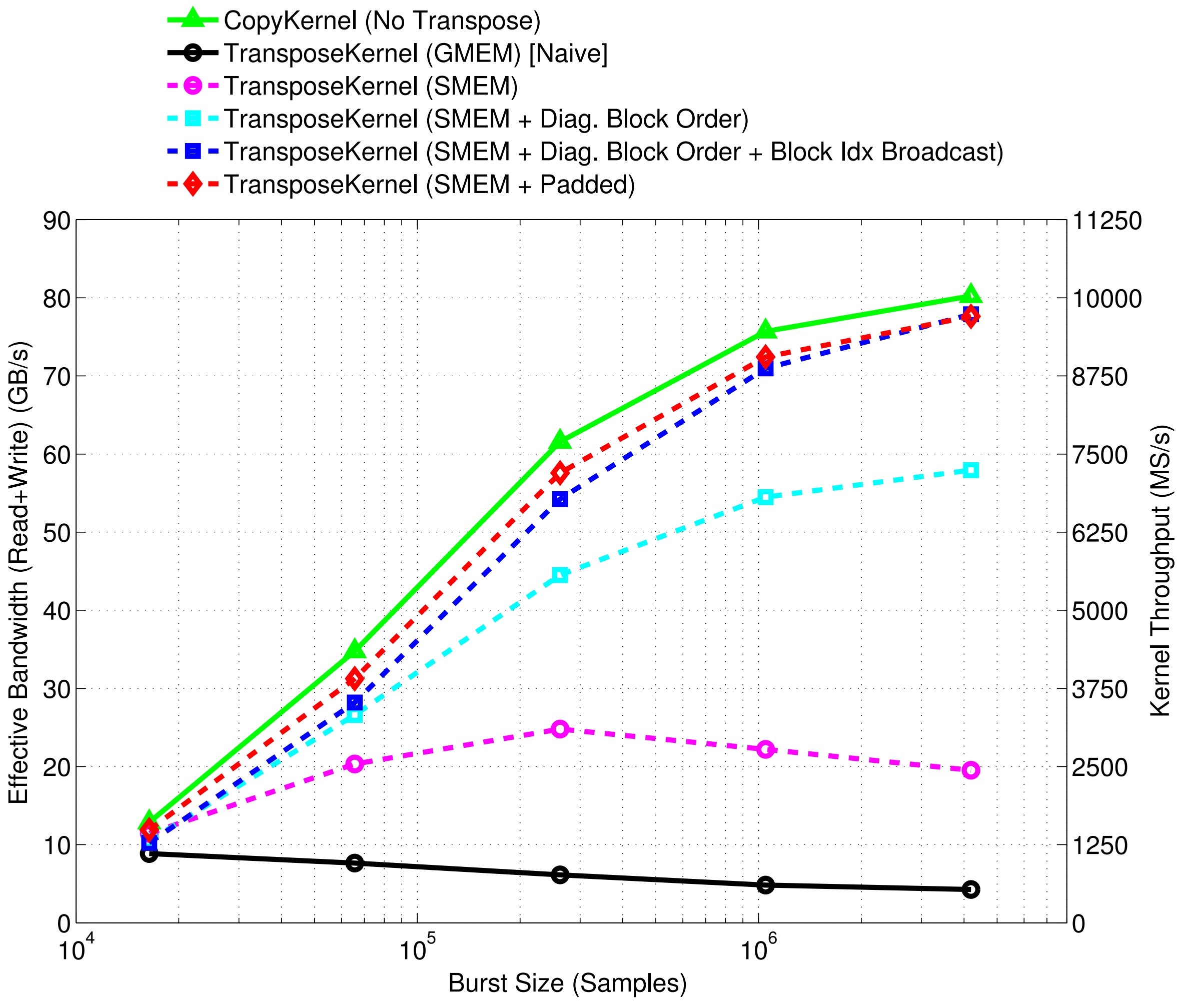

Figure 6.12: CT function benchmark results for square dimensions using the `float2` datatype to represent complex samples.

The remaining two results shown in Figure 6.12 for the SMEM with diagonal block ordering plus block index broadcast kernel and the SMEM kernel with padding are for the implementations that go beyond the final implementation that is presented in [50] and found in the corresponding CUDA SDK matrix transpose example that the CT kernels are based on. The results for the SMEM kernel with padding were generated using the standard SMEM kernel and by allocating input and output buffers where rows are padded with a single global memory partition width of 256 bytes that effectively causes the column thread blocks to be spread across memory partitions, where row thread blocks are already naturally spread across partitions. The CT function consists of a single kernel and subsequently the low-level effective bandwidth is shown on one axis along with the equivalent high-level kernel throughput for the function itself on the other axis.

Table 6.5 shows a summary of the results in Figure 6.12 with the limiting factors that were identified for each of the kernels. It was discovered that the SMEM with diagonal block ordering kernel is limited by instruction throughput, achieving only 72% of the copy kernel bandwidth, due to the slow modulus operations that are used to calculate the diagonal block indices from the standard cartesian mapping within each thread. The improved version of this kernel that was implemented calculates the block index in a single thread and broadcasts it to all threads in the block via shared memory, achieving within 97% of the copy kernel bandwidth. The SMEM kernel with padding also achieved 97% of the copy kernel bandwidth and is, therefore, confirmed as an effective alternative to using a diagonal ordering scheme.

Table 6.5: Summary of CT benchmark results with identified limiting factors for each kernel.

CT Kernel	**Padded Buffer**	**% of CopyKernel BW**[*]	**Limiting Factor**
GMEM	No	5%	GMEM BW (Uncoalesced + PC[††])
SMEM	No	24%	GMEM BW (PC[††])
SMEM + Diag	No	72%	Instruction Throughput
SMEM + Diag + Idx Bcast	No	97%	Synchronization Overhead
SMEM	Yes[†]	97%	Synchronization Overhead

[*]Taken for burst size where copy kernel reaches maximum achievable bandwidth of 80 GB/s.
[†]Input & output buffers padded with 1 partition width (256 bytes) during memory allocation.
[††]Partition Camping.

The 3% drop in performance that is observed for the best performing CT kernels compared to the copy kernel is attributed to the added synchronization overhead for calls to the `__syncthreads()` synchronization primitive, that are required to synchronize read and write access to shared memory among all threads in the block. The slightly lower performance for the SMEM with diagonal block ordering plus block index broadcast kernel compared to the SMEM kernel with padding that is observed for most burst sizes is attributed to an additional `__syncthreads()` call that is required to distribute the block index to all threads via shared memory. However, if explicit padding and stripping steps are required on the data stream of the bigger processing chain, merely for the purpose of the transpose function, additional overhead will be introduced that is not reflected as part of these results. The SMEM with diagonal block ordering plus block index broadcast kernel is therefore recommended for the radar application.

6.3.3 Doppler Filter (DF)

The kernel throughput that is achieved for the different variants of the DF function, with all processing steps performed, is shown in Figure 6.13. Firstly, the number of pulses is varied, which represents

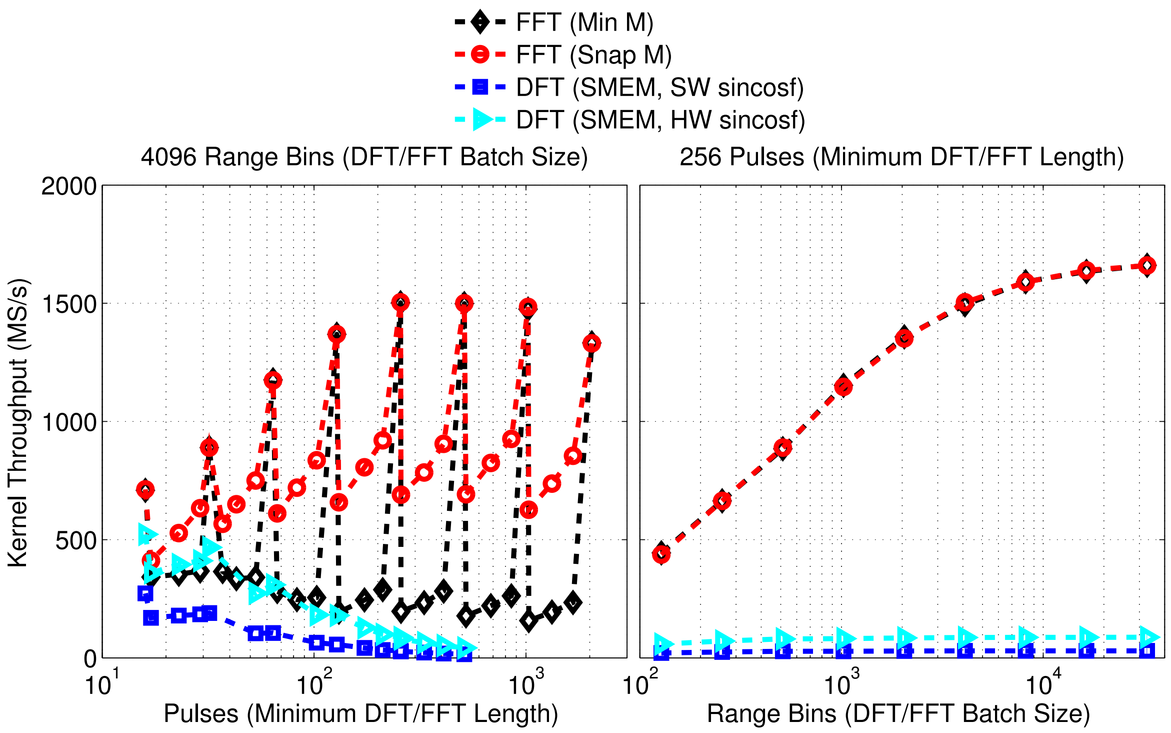

Figure 6.13: DF function kernel throughput (T_k) for all variants for varying DFT/FFT length and batch sizes.

the minimum DFT/FFT length that needs to be performed. Secondly, the number of range bins is varied, which represents the DFT/FFT batch size. Every fourth length that was evaluated, starting with the shortest, is a power-of-2 and multiple of 16. All other lengths are prime numbers in order to illustrate the effects of suboptimal DFT/FFT lengths on the different variants. Suboptimal lengths may be common for certain radar systems as the minimum DFT/FFT length is directly related to the number of pulses used in the burst as determined by the system waveform design. For the majority of the variants the minimum length matches the actual length that is performed. However, for the FFT snap variant, the actual length is the equal to the next higher power-of-2 and multiple of 16 when the snap scheme length constraint is not met by the minimum required length.

The two FFT variants perform identically for lengths that already meet the stated length criteria, as expected. However, for the suboptimal lengths, the FFT snap variant performs padding and extends the FFT length to remain in the more optimal Cooley-Tukey code path for the CUFFT library. The FFT snap variant performs better than the FFT variant that uses the minimum required FFT length by a factor of up to 3 times, despite the additional overhead of the padding step. A reduction in performance is still observed in the kernel throughput curve for the FFT snap variant, which is attributed to the overhead of the padding kernel that is active in the regions with suboptimal minimum DFT/FFT lengths. Additionally, the FFT snap variant performs wasted computation proportional to the difference between the actual and minimum required FFT length, which leads to the sawtooth shape of the steps.

The FFT batch size also has a substantial impact on the performance with bigger batches leading to better performance across the range evaluated. The FFT batch size is determined by the number of pulses in the burst and therefore increases proportional to the burst size in this dimension. This observed effect is therefore expected in accordance with the established relationship between burst size and throughput from the device global memory microbenchmark results in Section 6.2.2.

For the two DFT variants the results show that the variant with the intrinsic or hardware `__sincosf` function, instead of the software `sincosf` in the inner loop, performs better by a factor of up to 3 times. A performance dip is visible for both DFT variants between lengths 16 and 32, which is attributed to threads being assigned to independent DFTs in multiples of a warp of 32 threads within a thread block. For lengths that are not a multiple of 32, the last warp has a varying number of idle threads, as it handles the residual. Within the region in question, only two warps are used per DFT. Therefore the number of idle threads in the residual warp is a large percentage of the total threads per DFT, leading to the inefficiency. The DFT batch size has a more limited impact compared to the FFT batch size. This is due to the DFT variants being compute limited instead of bandwidth limited, which reduces the impact of device global memory for small bursts. The best-performing DFT variant exceeds the performance of the worst-performing FFT variant for certain of the shorter lengths. However, the FFT snap variant performs better than all other variants in all cases that were evaluated here.

Both DFT kernel variants use SMEM to efficiently cache the input data where each sample is read from GMEM only once, as shown in Table 5.8, instead of naïvely reading each sample S times or once per DFT output sample. The maximum bandwidth that was achieved for the DFT kernel variants is only around 11% of the maximum achievable device memory bandwidth, as shown in Table 6.6. The peak computational throughput is also shown, where the hardware DFT kernel variant achieves more than 250 GFLOPS, exceeding the performance of the software variant by a factor of more than 3 times.

The loop unrolling that is applied to both DFT kernel variants increased the performance of the hardware variant by around 20% on average, where virtually zero improvement was observed for the software variant. This discrepancy is attributed to difference in the number of instructions and overall execution time required for the hardware versus the software implementation of the transcendental function, where the software implementation requires substantially more instructions and execution time. The fact that loop unrolling was only possible for lengths of 16 and 32 for the software variant, compared to 16, 32, 64 and 128 for the hardware variant is further evidence of the difference in

Table 6.6: Measured low-level metrics for individual DF kernels.

DPC Kernel	$\lceil \mathbf{T_{eb}} \rceil$	**% of** $\lceil \mathbf{T_{eb}} \rceil$ **Achievable**[*]	$\lceil \mathbf{T_{c}} \rceil$[†]	**Limiting Factor**
Win. Fn. (CMEM)	101 GB/s	123%	10.1	BW
DFT (SMEM, SW `sincosf`)	3 GB/s	3.7%	76.8	Instr. Throughput
DFT (SMEM, HW `sincosf`)	9 GB/s	11%	253.4	Instr. Throughput
Scale Complex	80 GB/s	98%	10.0	BW

[*]Relative to max achievable BW for device memory bandwidth microbenchmark results.
[†]Specified in GFLOPS.

instruction count. The additional instructions lead to a more heavyweight inner loop, where the contribution of loop overhead to the total execution time dwindles.

General provisions were made with respect to SMEM access to avoid bank conflicts, although the `warp_serialize` counter that represents the number of shared memory bank conflicts in the NVIDIA Visual Profiler confirmed that 2-way bank conflicts are indeed present, as expected. The 2-way bank conflicts are not expected to have a performance impact due to the high AI of the inner loop for the DFT kernels. Neither of the variants are therefore considered to be bandwidth limited, but instead both are deemed compute limited by instruction throughput.

The window function kernel and scale complex kernel are common between the FFT and DFT variants. The scale complex kernel is identical to the kernel used for the DPC and the corresponding results are discussed in Section 6.3.1. The window function kernel achieves around 123% of the bandwidth, as shown in Table 6.6. The window function kernel exceeds 100% of the bandwidth due to const memory cache hits on the window coefficients that are stored in CMEM, which do not consume device memory bandwidth. A worst-case W_{dmem} is assumed, as shown in Table 5.8, without cache hits for the purpose of the effective bandwidth calculation. The window function kernel achieves low peak computational throughput of around 10 GFLOPS, despite high effective bandwidth. This is attributed to the low, constant AI per sample and it is therefore considered bandwidth limited.

Figure 6.14 shows the overall DF performance for the best-performing FFT and DFT variant, under conditions that are more specific to the radar application with respect to realistic burst sizes. The FFT snap variant is shown to be superior to the DFT hardware variant for all but the smallest burst size evaluated, where the DFT/FFT length and the batch size are small. The FFT snap variant is recommended in general for the radar application. However, in the cases where the FFT length is snapped to a longer length, the number of output Doppler bins are extended proportionally, which may impact on downstream processing after the DF function. Therefore, it is advisable to minimize the difference between the minimum and actual FFT length to minimize the downstream processing overhead. A more sophisticated snap scheme where, for instance, powers of 3, 5 or 7 are also considered for the length factorization, may yield more optimal overall performance. For systems where the number of pulses is large, an overlap-save approach could also be effective.

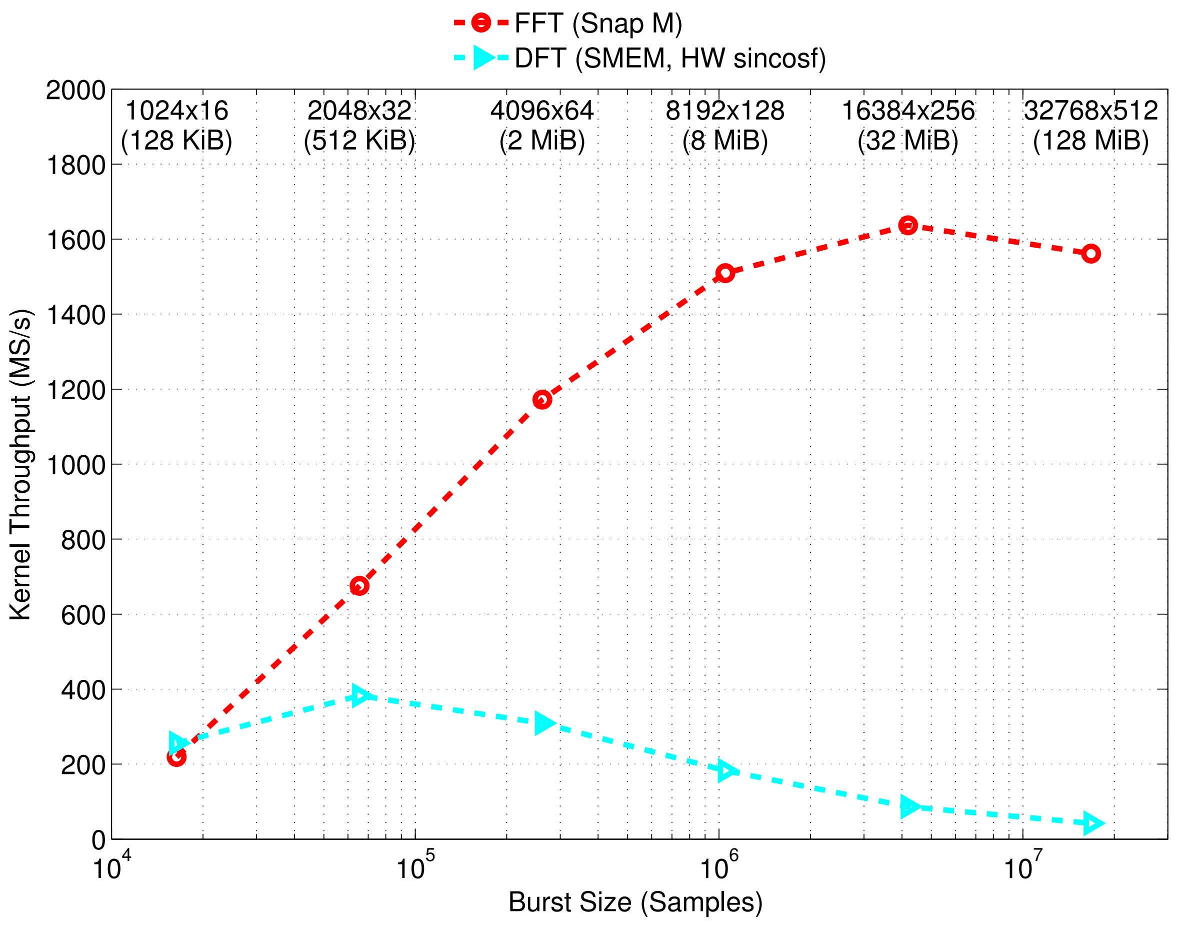

Figure 6.14: DF kernel throughput (T_k) with realistic burst dimensions and sizes varied over a broad range showing labels for number of range bins by number of Doppler bins.

6.3.4 Envelope Function (ENV)

The results for the ENV function for all variants is shown in Figure 6.15. The ENV function consists of a single kernel and therefore the low-level effective bandwidth is shown on one axis along with the equivalent high-level kernel throughput for the function itself on the other axis. The actual burst dimensions in range and Doppler have no impact on the ENV function as it accesses the input buffer linearly as a 1D buffer.

The results show that for the linear rectifier the software `sqrtf` variant performs best, followed closely by the software `hypotf` variant with the hardware `sqrtf` equivalent that uses the intrinsic `__fsqrt_rn` function at around only 50% of the peak performance of the other variants. Table 6.7 shows a summary of the measured low-level metrics for the ENV kernels, where we can see that the

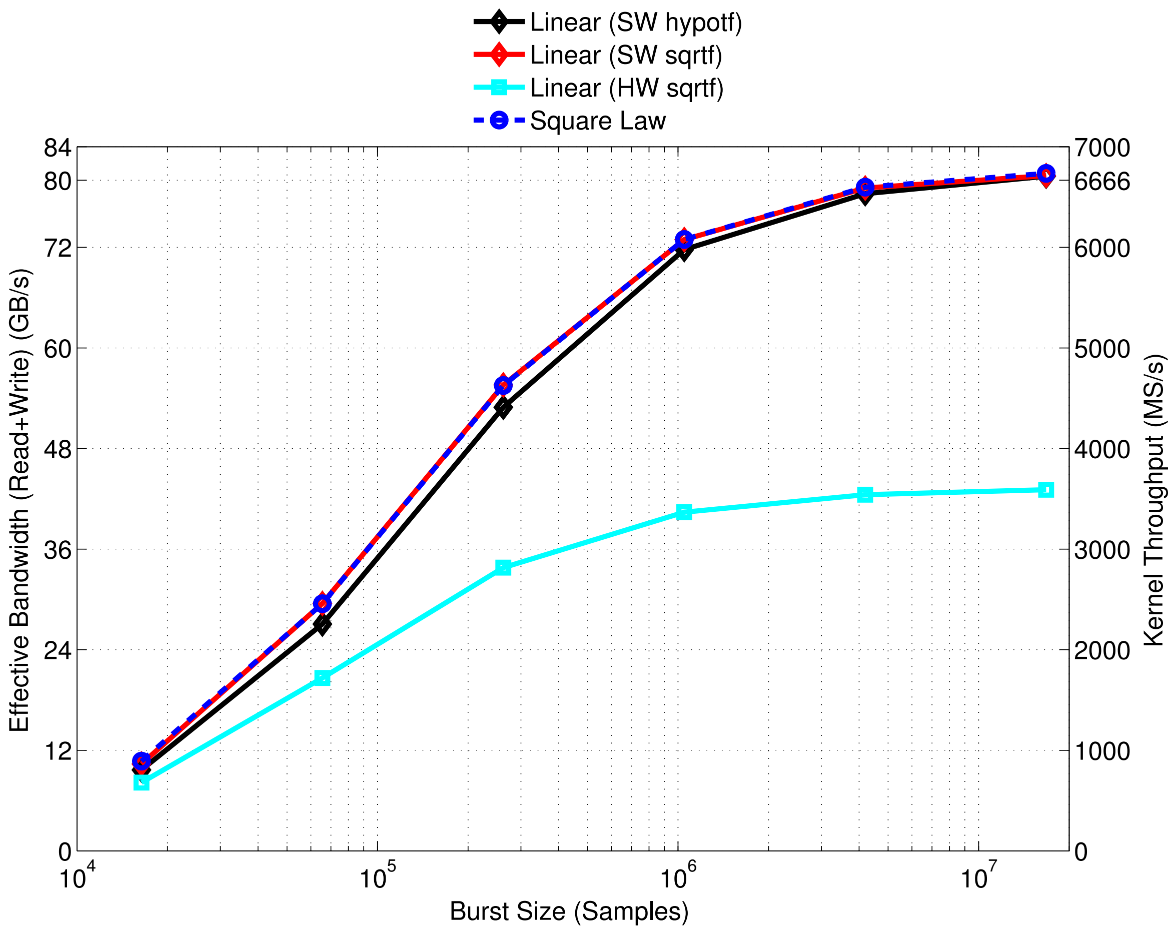

Figure 6.15: ENV kernel effective bandwidth (T_{eb}) and ENV function kernel throughput (T_k) varied over a broad range of burst sizes for all variants.

linear rectifier software variants achieve nearly 100% of the bandwidth and we can conclude that they are limited by bandwidth. The hardware variant, however, achieves only 53% of the bandwidth and we have to conclude that it is bound by instruction throughput, contrary to what was predicted for the initial analysis in Table 5.9 where the square root operation is counted as a single FLOP, and where it is assumed that the peak theoretical device bandwidth and computational throughput for the device is achievable.

The lower performance for the hardware variant is attributed to the lower instruction throughput for the SFUs, which executes additional native instructions for the intrinsic functions, compared to the SPs that perform more basic arithmetic with the standard software functions. The SFU throughput for square root is 2 operations per clock cycle per multiprocessor, where the SP throughput is 8 [32] for basic operations. It may be possible to achieve better overall performance with the hardware

Table 6.7: Measured low-level metrics for individual ENV kernels for the experiment shown in Figure 6.15.

ENV Kernel	$\lceil \mathbf{T_{eb}} \rceil$	% of $\lceil \mathbf{T_{eb}} \rceil$ Achievable*	$\lceil \mathbf{T_c} \rceil^{\dagger}$	Limiting Factor
Linear (hypotf)	81 GB/s	99%	26.9	BW
Linear (sqrtf)	81 GB/s	99%	26.9	BW
Linear (__fsqrt_rn)	43 GB/s	53%	14.4	Inst. Throughput
Square Law	81 GB/s	99%	20.2	BW

*Relative to max achievable BW for device memory bandwidth microbenchmark results.
†Specified in GFLOPS.

variant if additional computation that uses the SPs is performed in the same kernel, by merging the ENV kernel with another processing kernel, for instance, as the SP computation may overlap with SFU computation as discussed in Section 6.2.1. However for a standalone ENV kernel the software `sqrtf` variant is recommended for a linear rectifier implementation.

The results further show that the square law rectifier performs virtually identical to the best-performing linear rectifier software `sqrtf` variant with respect to effective bandwidth utilization and kernel throughput and is therefore also bandwidth limited. The computational throughput for the square law kernel, as shown in Table 5.9, is however, lower than the best-performing linear rectifier kernel, which is due to a lower computational workload, also indicated in the table, due to the absence of the square root operation and because it is limited by bandwidth.

It is concluded firstly, that the intrinsic functions do not always provide better or equivalent performance. Secondly, for a standalone ENV kernel there is no performance difference between a linear rectifier and square law rectifier implementation, as the kernel is limited by bandwidth in both cases.

6.3.5 Constant False-Alarm Rate (CFAR)

The kernel throughput for all CFAR variants that were implemented where all steps are performed is shown in Figure 6.16. The number of cells, N, in the CFAR window is varied over a broad range with the window extents in range W_{rng} and Doppler W_{dop} kept roughly square with a fixed guard cell

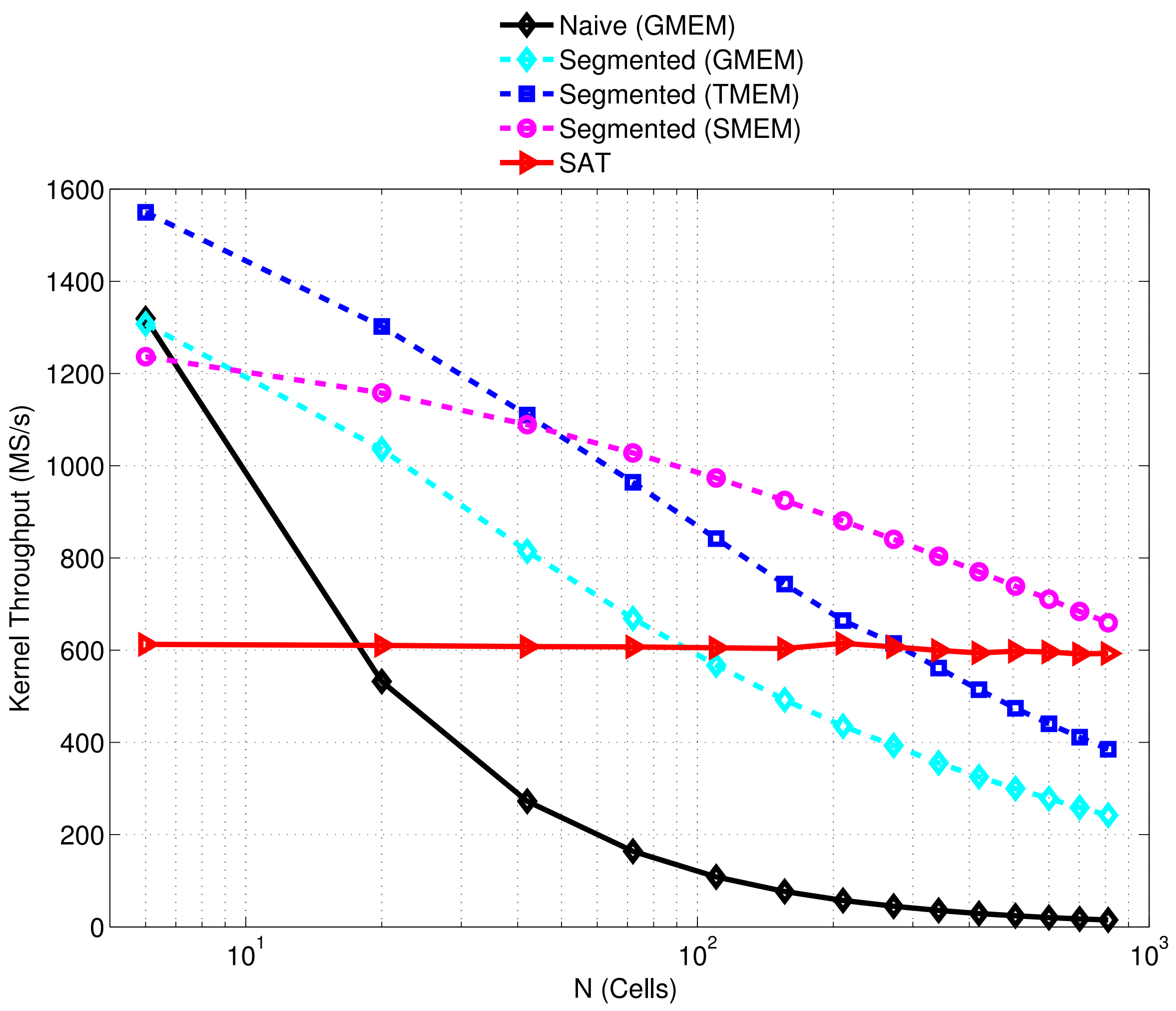

Figure 6.16: CFAR kernel throughput (T_k) for N varied over a broad range for a single large 3072 by 3072 burst (36 MiB buffer) that is processed.

extent W_{grd} of 3. To isolate the effects of the window size and to establish an expected upper bound on performance under conditions where maximum device memory bandwidth may be reached, a large square 3072 by 3072 burst is processed at first. A square burst is used to avoid potential effects related to skewed buffer width to height ratios at this stage. The burst size is also chosen so that the buffer size of 36 MiB at least meets or exceeds the roughly 32 MiB observed minimum buffer size required to reach the maximum achievable device memory bandwidth of approximately 82 GB/s for a simple copy kernel with optimal coalescing in the device memory bandwidth microbenchmark in Section 6.2.3. A known, potentially achievable upper bound for kernel effective bandwidth T_{eb} is established under these conditions, which can be used to analyze individual kernel performance and determine limiting factors, as shown in Table 6.8. Note that for the SAT variant the CUDPP planning

Table 6.8: Measured low-level metrics for individual CFAR kernels for the experiment shown in Figure 6.16.

CFAR Kernel	$\lceil \mathbf{T_{eb}} \rceil$	**% of $\lceil \mathbf{T_{eb}} \rceil$ Achievable**[*]	$\lceil \mathbf{T_c} \rceil$[†]	**Limiting Factor**
Apron Generation				
Apron Gen. (TMEM)	75 GB/s	91%	0	BW
Interference Estimation				
Naïve (GMEM)	56 GB/s	68%	12.2	BW
Seg. Rows (GMEM)	79 GB/s	96%	18.6	BW
Seg. Cols (GMEM)	59 GB/s	72%	12.0	BW
Seg. Rows (TMEM)	130 GB/s	159%	28.5	BW
Seg. Cols (TMEM)	90 GB/s	110%	21.1	BW
Seg. Rows (SMEM)	51 GB/s	62%	41.0	Inst. Throughput
Seg. Cols (SMEM)	60 GB/s	73%	48.7	Inst. Throughput
SAT Transpose (SMEM)	35 GB/s	43%	0	BW
SAT Lookup (TMEM)	81 GB/s	99%	18.1	BW
Detection Masking				
Det. Mask Gen. (GMEM)	76 GB/s	93%	12.8	BW

[*]Relative to max achievable BW for device memory bandwidth microbenchmark results.
[†]Specified in GFLOPS.

is performed during initialization and is, therefore, excluded from performance measurements, similar to memory allocation.

The results in Figure 6.16 show that the Naïve (GMEM) variant performs well for very small N, but performs worse than any of the other variants, and at less than 50% of its peak kernel throughput for $N \geq 20$. At $N \geq 100$ it performs at less than 10% of its peak kernel throughput and is therefore very sensitive to the magnitude of N. Poor comparative performance may be expected owing to a linear increase in the device memory and computational workload for the kernel with a linear increase in N, as defined in Table 5.10, where other kernels have more optimized workloads due to their structure. Furthermore, the results in Table 6.8 show that the interference estimation kernel for this variant only achieves 68% of the achievable bandwidth, which is attributed to suboptimal global memory coalescing due to misalignment on many of the read requests, due to the required access pattern. Each thread evaluates a single CUT and needs to read a contiguous block of CFAR window cells, which has a negative performance impact as discussed in Section 5.5.5.3 and illustrated in Figure 6.3 for misaligned access due to an offset.

The results for all the segmented variants show an immediate improvement in kernel throughput with respect to a reduced sensitivity to N, firstly due to the reduction in the computational workload that is common among all segmented variants, as discussed in Section 5.5.5.6. Furthermore a reduction in the device memory workload and the type of memory abstraction employed has added benefits that differ between the segmented variants. The Segmented (GMEM) variant still achieves 45% of its peak kernel throughput for $N = 100$. For interference estimation, its rows kernel, which sums across rows, achieves 96% of the bandwidth due to optimal inter-row alignment and, therefore, optimal coalescing due to the padding added during apron generation for the row pitch. However, the columns kernel, which sums across columns, still suffers from misalignment the same as the Naïve (GMEM) variant does and only achieves 72% of the bandwidth.

The Segmented (TMEM) variant shows a kernel throughput increase of around $20-60\%$ over the previous variant. The rows and columns interference estimation kernels for this variant perform at 159% and 110% of the bandwidth respectively, where the former even exceeds the peak theoretical device memory bandwidth, which is possible due to the on-chip texture cache that is invoked. Despite the overall caching benefits, the performance improvement for the columns kernel in particular is attributed to a reduced sensitivity to misalignment, as texture memory does not impose the same coalescing restrictions as global memory.

The Segmented (SMEM) variant improves kernel throughput further over the prior variant, but only in the region where $N > 50$. Peak performance is also slightly lower than all variants covered so far for the smallest N evaluated. The rows and columns interference estimation kernels achieve only 62% and 73% of the bandwidth at best for the smallest N, with a linear reduction as N increases. The computational throughput for the SMEM kernels, on the other hand, increase with increasing N to in excess of 40 GFLOPS for the largest N that was evaluated. These observations correspond to what is expected due to the increase in AI with an increase in N, as discussed in Section 5.5.5.6, and also indicates that the kernel is compute bound as computational performance increases with a decrease in bandwidth. Also, by temporarily disabling computation loops selectively and retaining shared memory load and store instructions both kernels achieve 94% of the bandwidth. This quick test confirmed that both kernels achieve optimal global memory coalescing during load to and store from the shared memory as intended with the design in Figure 5.10 and 5.11, and is therefore not limited by device memory bandwidth.

The kernel shared memory access was also confirmed bank-conflict free by verifying that the `warp_serialize` counter, that represents the number of bank conflicts in the NVIDIA Visual Profiler shows 0. The kernels are therefore considered bound by instruction throughput with respect to computation, but also in part due to loop overhead and additional calculations and checks that are required to determine which cells to load into shared memory. Loop unrolling could therefore potentially be used to further increase performance. However, the primary limiting factor, especially for small N, is currently considered to be algorithmic with respect to the shared memory loading scheme, where each thread always loads 3 cells into a block layout that provides for a CFAR window extent of up to 33 by 33 cells, regardless of the actual CFAR window extents that are required for processing. The shared memory layout can also be optimized further to minimize redundant loads, by reducing the ratio between the number of cells that are loaded into shared memory versus the number of cells that evaluated and output by each thread block, which is at $3:1$ with the current implementation, as illustrated in Figure 5.10 and 5.11.

The SAT variant shows a near constant kernel throughput across the range of N at around 40% of the peak kernel throughput observed for all other variants. By visually extrapolating, the SAT should provide the best performance across all variants for N in excess of 1000 cells, which is very far beyond what is generally required for radar applications. Nonetheless the objective to decouple performance from the CFAR window size is achieved with the SAT, as it very insensitive to N, due to the constant-time nature of the SAT generation and lookup steps. The SAT Lookup kernel achieves up to 99% of the bandwidth with texture memory that was used again to avoid misaligned access, as the access pattern also depends on the CFAR window extents and for potential caching benefits. An initial global memory implementation was confirmed to also suffer from poor coalescing and misalignment, similar to what was observed for some of the other CFAR kernels. However, the SAT Lookup step only comprises around 30% of the execution time for the SAT interference estimation step with around 70% attributed to the SAT generation, according to measurements that were made. The overhead of generating the SAT using the four steps shown in Figure 5.12 is therefore the primary performance contributor with the execution time for each scan or transpose step roughly the same, owing to the linear time CUDPP scan implementation.

As mentioned in Section 5.5.5.4 the second transpose operation is strictly speaking not functionally required, but was added in order to maintain a consistent data ordering among the different variants. Given that each of the four SAT generation steps execute for roughly the same duration, an overall performance improvement for the SAT variant of less than 20% may be expected. Considering the results in Figure 6.16 with this hypothetical performance improvement the SAT variant will still only exceed the performance of other variants for N on the order of 500 cells. Therefore the SAT variant with the current implementation should only be considered for applications that require very large CFAR windows, well beyond what is typically required for radar. Further optimization of the SAT generation into fewer steps could improve performance. Shared memory can also potentially be used to improve performance for the SAT Lookup kernel further, as there is reuse between lookup values, although limited and distributed across CUT cells that do not necessarily exhibit spatial locality themselves, which presents additional complexities.

Lastly, the apron generation and detection mask generation kernels are common to all variants and briefly discussed here. The apron generation kernel uses texture memory to read the input burst data and achieves up to 91% of the bandwidth. A quick experiment was also conducted by comparing texture memory to global memory for reading the input burst data, where the row pitch was varied through offsets of 0 to 15 words by incrementing the Doppler dimension in steps of 1 using the 3072 by 3072 burst for the base size. The worst-case performance for the texture memory implementation was 88% where the global memory implementation achieved up to 91% for the ideal cases, but as low as 70% for the suboptimal cases.

The optimization to increase the number of elements/thread to two increased peak effective bandwidth by 8% for the initial global memory implementation and 3% for the final texture memory implementation, by reducing the ratio of index calculation and conditional check instructions to memory instructions. A further increase of the number of elements/thread yielded no additional benefits and it was confirmed that the implementation was bandwidth limited beyond this point by temporarily removing the conditional checks and additional writes for the Doppler apron regions and achieving the same effective bandwidth. The detection mask generation kernel achieves 93% of the bandwidth using a pure global memory implementation. The detection mask generation kernel does not suffer from misalignments due to the efficient padded structure provided by the apron generation step.

The overall CFAR performance is now evaluated further in conditions that are more specific to the radar application with respect to the CFAR window size that is kept at a realistic 40 cells with further exploration of the impact of burst dimensions and burst size. A burst in a typical radar system will typically not be square as the number of range bins and Doppler bins are independent, where the former is determined by the range swath that is processed and the latter is determined by the number of pulses used in the burst. Figure 6.17 shows the performance for all variants for different burst dimension ratios of range bins to Doppler bins ranging from 1 : 256 to 256 : 1 for a fixed burst size. The burst size is still kept at 36 MiB total size at this stage for the same reasons as with the initial tests. The performance for most variants are fairly homogenous across the range with the most pronounced effect for ratios lower than 1 : 4 for the Segmented (TMEM) variant, where performance

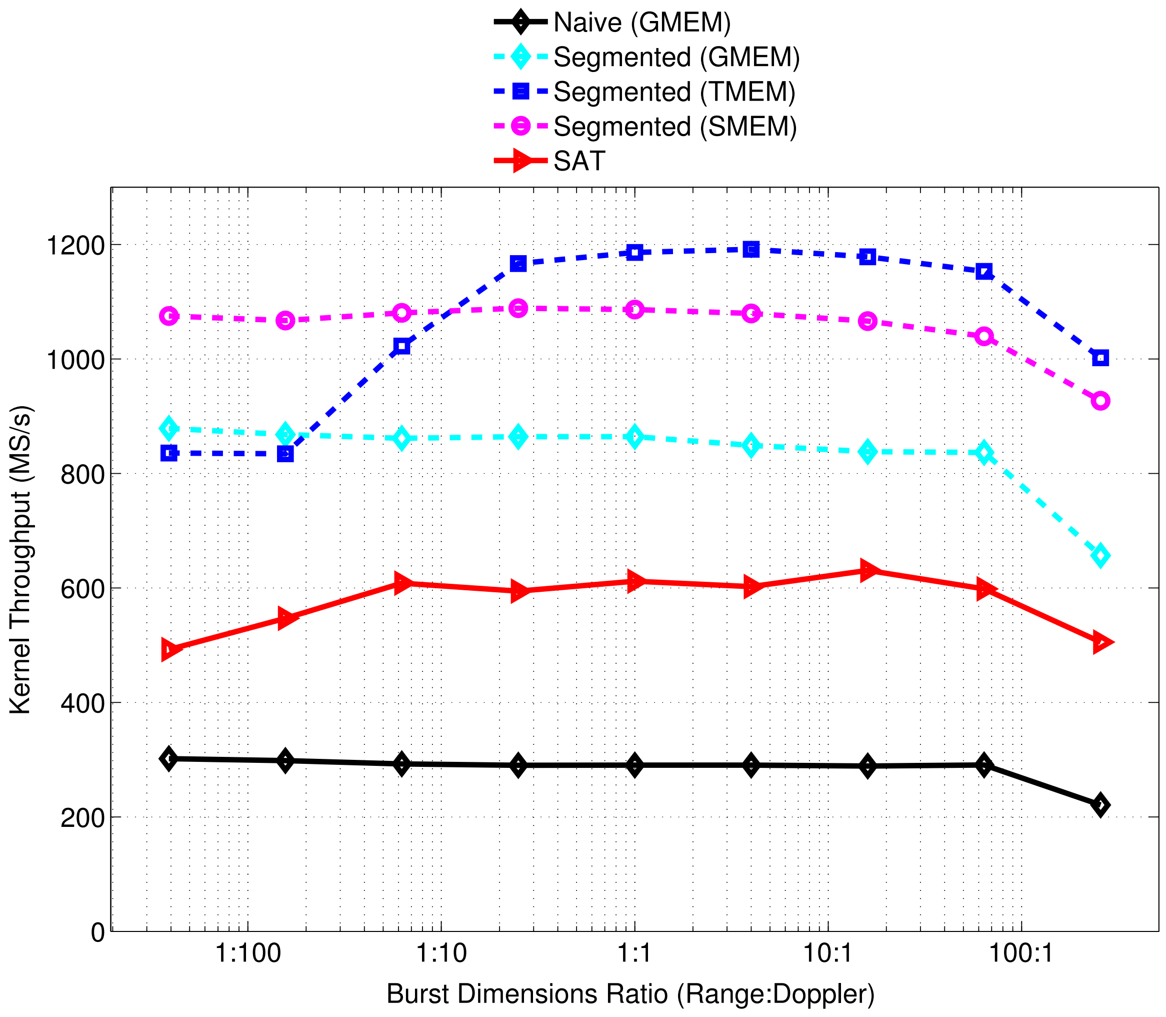

Figure 6.17: CFAR kernel throughput (T_k) for $N = 40$ using a large burst size (36 MiB buffer) with a varying ratio between range and Doppler dimensions with respect to number of samples or bins.

eventually drops to around 70% of its peak performance, which is also lower than performance for the Segmented (SMEM) variant at this point. This performance drop is attributed to caching effects. Secondly, all variants experience a performance drop at the highest ratio of 256 : 1 where the number of Doppler bins, that is the major dimension, is low. It is suspected that this effect is a mild form of partition camping that is experienced when the padded buffer row pitch, which is generated by the apron kernel, is a multiple of the partition size, with a low Doppler bin count where the ratio of useable cells compared to padding cells is low. The padding cells are not read or written and fall within particular partitions that then go underused, with other partitions oversubscribed.

Figure 6.18 shows the performance for all variants where realistic burst sizes are used. As with the initial performance characterization in Figure 6.16 we see that the Segmented (TMEM) and Segmented

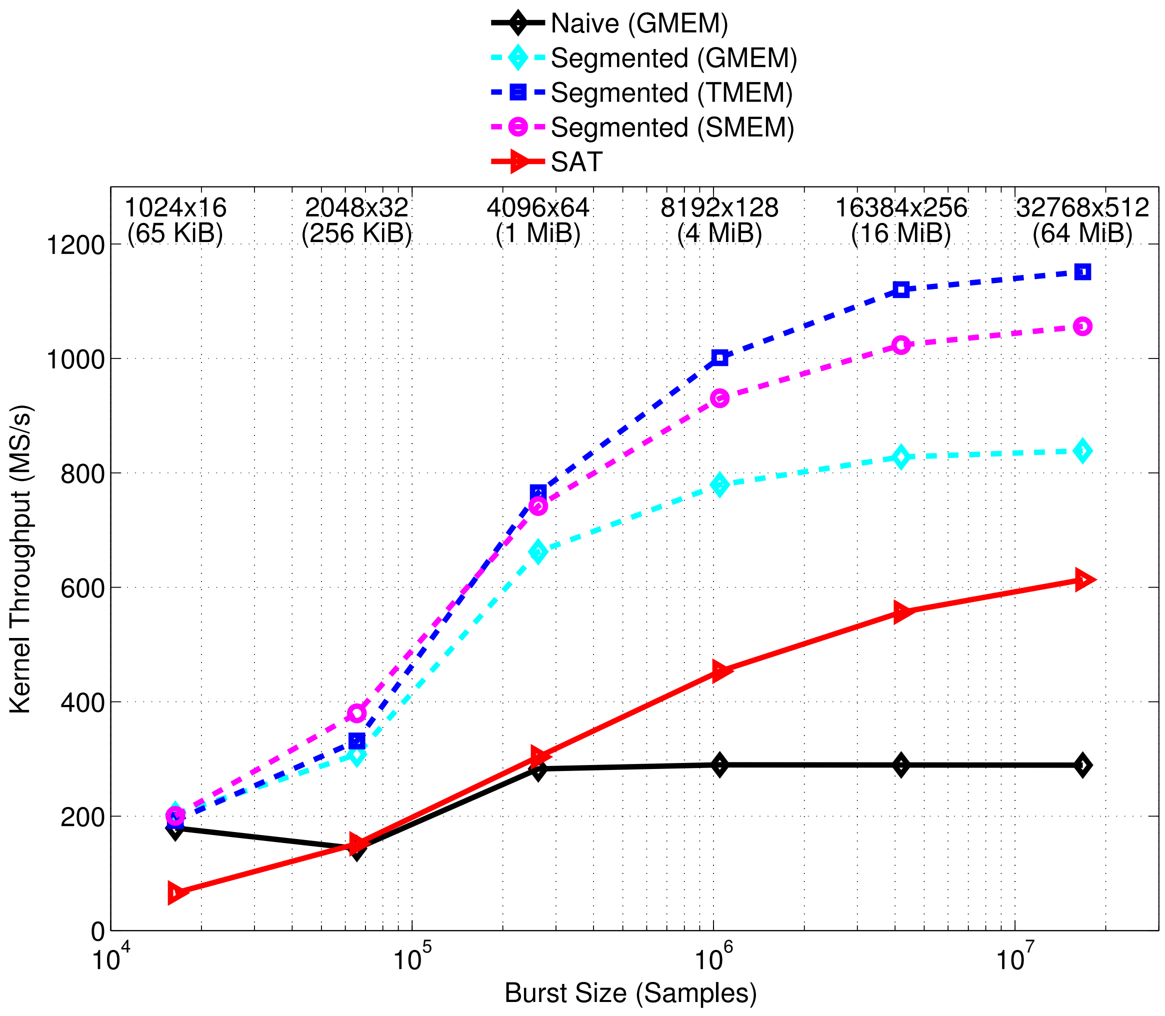

Figure 6.18: CFAR kernel throughput (T_k) for $N = 40$ and with realistic burst dimensions and sizes varied over a broad range showing labels for number of range bins by number of Doppler bins.

(SMEM) variants provide the best performance. The trend that is observed as a function of burst size for these kernels that have been well optimized is very similar to the trend observed for the simple copy kernel results in the device memory microbenchmark in Figure 6.2.

A buffer size or burst size of 32 MiB was required in the microbenchmark to achieve maximum device memory bandwidth and therefore maximum throughput. The lower performance that is achieved for smaller burst sizes is therefore innate to the GPU architecture where a certain minimum number of threads and data elements are required in order to achieve good utilization of all processing units and latency hiding. A performance drop is experienced for the 2048 by 32 burst that is also attributed to the mild partition camping effect that was observed in the burst ratio test. The effect is less severe on the Segmented variants that have reduced device memory workloads compared to the Naïve (GMEM) variant, where the effect is more severe.

6.4 RADAR SIGNAL PROCESSING CHAIN

The results for the complete radar signal processing chain with respect to throughput and latency is shown in Figure 6.19 for realistic burst sizes and processing function parameters. The GBF capability to process multiple bursts back-to-back is used to investigate the implemented pipelining techniques using 8 consecutive bursts. All measurements for the processing chain results include data transfer between the host and device, which was excluded for the individual radar signal processing functions in the preceding sections.

The results show that both the pinned (overlap) and mapped pinned variants exceed the throughput achieved with the pinned (without overlap) option in all cases, by hiding the data transfer latency using pipelining to increase throughput. The pinned (overlap) variant achieves higher throughput than the mapped pinned variant in all cases, performing up to 22% better for the largest burst size evaluated. For pinned (overlap) variant, the data transfer can overlap with multiple kernels in the processing chain, as the overlap is inter-burst. However, for the mapped pinned variant, the data transfer can only overlap with the pad kernel and the detection mask generation kernel execution, as the overlap is intra-burst. The pad and detection mask generation kernels have zero and fairly low AI respectively, therefore, data transfer is implicitly not overlapped with heavyweight computation. This leads to less efficient overall overlap of data transfer and computation, and therefore lower throughput.

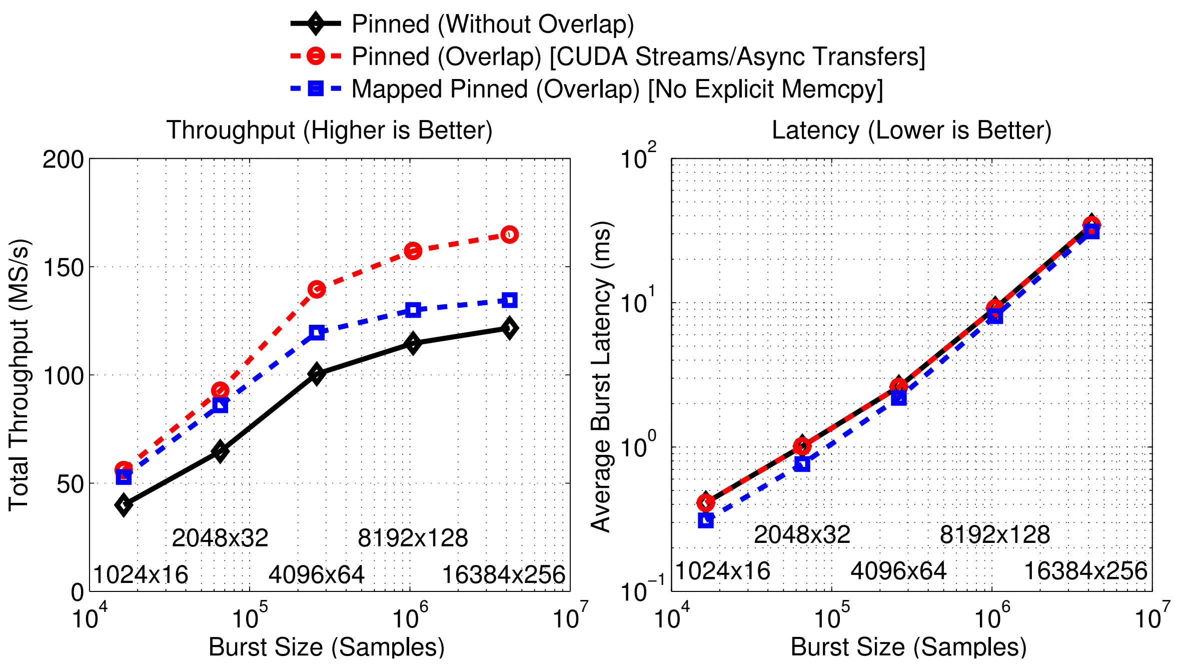

Figure 6.19: Radar signal processing chain throughput (T_{t8}) and latency ($\bar{L}_{b8}$) for 8 consecutive bursts with a CFAR window size of 40 and DPC filter length of 256 with realistic burst dimensions and sizes varied over a broad range showing labels for number of range bins by number of Doppler bins.

The results further show that the latency for the pinned (without overlap) and pinned (overlap) variants are identical, which is expected for case 1 where the compute duration exceeds the data transfer duration, as shown in Figure 6.6 for the microbenchmark in Section 6.2.4. However, the latency that is achieved for the mapped pinned variant is lower than for the pinned variants in all cases, performing up to 32% better for the smallest burst size evaluated. The reduced latency for the mapped pinned variant is attributed to the intra-burst overlap, where data transfer and kernel execution for a given burst can start simultaneously. For the pinned (overlap) variant, intra-burst kernel execution and data transfer for a given burst are mutually exclusive and occur sequentially, as they are explicit commands in the same CUDA stream.

To determine the compute to data transfer ratio for the radar signal processing chain, the average HtoD plus DtoH latency and the average kernel latency, for the pinned (without overlap) case is used. The compute to data transfer ratio for the radar signal processing chain was found to be between 1.8 : 1 and 2.3 : 1 across the broad range of burst sizes evaluated. The expectation that the compute duration exceeds the data transfer duration for the radar signal processing chain is therefore confirmed. Note that with all the radar signal processing functions active in the signal processing chain, the total available device memory of 4 GiB, as shown in Table 5.1, limited the maximum burst size that could be evaluated to 16384 by 256. With the current implementation, all functions allocate dedicated device memory buffers for every processing step to ease debugging, by allowing the intermediate output for each step to be transferred back to the host for analysis. This scheme can be optimized to reuse buffers to allow for larger burst sizes to be used.

The use of mapped pinned memory for pipelining is recommended for systems where low latency is the primary driver, especially where small bursts are required. Mapped pinned memory is also expected to perform better with respect to throughput when used in conjunction with high AI kernels. The use of pinned memory with asynchronous transfers is recommended for systems where high throughput is the primary driver, especially where large bursts are required.

CHAPTER 7

CONCLUSION

A software-defined radar signal processing chain for the data-independent processing stage of a single receiver-element, coherent, pulse-Doppler radar signal processor was implemented, optimized and evaluated, using CUDA on a Tesla C1060 GPU. The stream-programming paradigm, which is required for the highly-parallel GPU architecture, was adopted and successfully applied. Substantial effort was also invested to develop a GPU benchmarking framework (GBF) in software, to coordinate and facilitate benchmarking activities and enable repeatable experiments. The microbenchmarks were useful to characterize significant effects, which could be readily identified and addressed in the core implementations, and to identify practical upper performance limits. The individual radar signal processing algorithms were implemented and optimized for the target GPU architecture, followed by the integration into the radar signal processing chain, with further holistic optimization. The research questions that were posed are addressed in the rest of this section.

7.1 RADAR SIGNAL PROCESSING ALGORITHMS ON THE GPU

The CT and ENV functions were found to have a constant, low AI per sample. In addition, several of the helper kernels for the DPC and DF and CFAR functions, such as the pad, Hadamard product, scale complex, window function, apron generation, SAT transpose, SAT lookup, and detection mask generation kernels, were also found to have a constant, low AI per sample. Consequently, the best-performing variants of all of these kernels were all identified to be bandwidth limited, as the device memory access is the dominant factor and hides computation. Highly bandwidth limited kernels are not ideally suited to the GPU architecture, as the primary strength of high computational performance is then underutilized. Nonetheless, substantial effort was still required when optimizing these kernels to ensure that a high percentage of the available bandwidth is utilized, for instance, with respect to considerations for global memory coalescing and the choice of memory subsystem to use. It should be noted that, as the aforementioned kernels are all bandwidth-limited, these optimizations to maximize memory bandwidth were especially important.

The best-performing TDFIR (TMEM) kernel variant for the DPC function has a constant, low AI for a worst-case device memory workload where no constant cache hits for the filter coefficients and no texture cache hits for the input data are made. For the worst-case scenario, the TDFIR kernel is found to be bandwidth limited. However, the AI increases as cache hits increase. With substantial cache hits and sufficiently long filter length, the TDFIR kernel is found to be compute limited. The best-performing DFT (SMEM, HW `sincosf`) kernel variant for the DF is found to have an AI that increases linearly with the number of pulses in the slow-time dimension. For a sufficiently large number of pulses, the DFT kernel is found to be compute limited. The interference estimation kernels for the CFAR function generally have an AI that is a function of the CFAR window extent. The best-performing Segmented (SMEM) CFAR variant is optimized with respect to memory access to the point where the AI increases linearly with the CFAR window dimensions. The Segmented (SMEM) variant is therefore found to be compute limited for sufficiently large CFAR window sizes, as the computation becomes the dominant factor and hides device memory data transfer.

The compute limited nature of the aforementioned kernels, especially for larger burst sizes and CFAR windows, are ideal for the GPU architecture. It should be noted that the initial naïve implementations for these kernels were all essentially bandwidth limited, until the necessary memory optimizations were made to use the on-chip memory subsystems appropriately. Furthermore, loop unrolling and the use of faster intrinsic functions were relevant and effective at improving performance for some of the compute bound kernels.

With respect to the different algorithmic variants that were implemented, the FDFIR, FFT, and segmented approach, respectively for the core DPC, DF, and CFAR functions, were found to be generally the most efficient for the radar application on the GPU architecture. For very short filter lengths of less than 64 samples, a TDFIR approach yielded better results for the DPC. For small bursts, where the DFT length and batch size is small, the DFT approach yielded better results for the DF. The SAT is a new CA-CFAR implementation approach, which has the advantageous property that it is insensitive to the CFAR window size. Nonetheless, the SAT approach was found to only perform better than other variants for very large CFAR window sizes, approximately in excess of 1000 cells. Such a large CFAR window is impractical for radar applications, that typically require one or two orders of magnitude smaller. More efficient methods of generating the 2D SAT in a single pass, without the need for explicit transpose operations, may improve performance.

7.2 RADAR SIGNAL PROCESSING CHAIN ON THE GPU

The pulse-Doppler radar signal processing chain was constructed on the GPU, by chaining the best-performing variants of the optimized radar signal processing algorithms together. The processing chain consists of a long series of kernels that all execute on the GPU, avoiding any intermediate data transfers to the CPU, besides the input and output of the entire processing chain itself. The individual kernels execute sequentially due to the data dependencies between them, and therefore the total execution time is effectively the sum of the execution time for each individual algorithm.

Pipelining techniques were successfully implemented to overlap data transfer and kernel execution for the radar signal processing chain, in order to improve performance. Different pipelining techniques, that perform intra-burst and inter-burst overlapping, were respectively found to be appropriate where low latency and high throughput are the primary performance drivers for the radar system in question. The compute duration was also found to be around double the data transfer duration for the complete radar signal processing chain. The maximum speedup that is therefore expected with ideal overlap between data transfer and execution is around 50%, where an improvement in throughput in excess of 40% was observed using pipelining with the inter-burst overlapping scheme. In turn, an improvement in latency of more than 30% was observed with the intra-burst overlapping scheme. Performance ranging from 50 to 150 MS/s is achieved for the full radar signal processing chain across a broad range of burst sizes.

However, small bursts remain a problem with respect to performance on the GPU architecture. The GPU architecture fundamentally caters for highly data-parallel workloads, whereas data-parallelism of the workload is reduced as the burst size is reduced. The GPU architecture also relies heavily on latency hiding, where the efficiency of latency hiding is reduced as the number of threads and the number of memory accesses are reduced. For some functions, the AI per sample is also a function of the burst dimensions, such as the DPC and DF, in which case the AI per sample is proportional to the fast-time and slow-time dimensions respectively. For these functions, smaller bursts exhibit lower AI, which in turn makes the workload less optimal for the GPU architecture.

It is concluded that the software-defined radar signal processing chain using a GPU architecture is highly feasible for medium to large burst sizes. The GPU architecture is therefore expected to be highly feasible for radar systems that use medium to low PRF waveforms with medium to high pulse counts, based on the breakdown provided in Section 2.4.5. High range resolution can also be achieved, due to the high throughput on the GPU architecture under these conditions. Furthermore, it is speculated that additional optimization may be applied to the bandwidth limited kernels to increase the average AI per kernel, which is discussed as future work later in this section.

7.3 PERFORMANCE EVALUATION ON THE GPU

Performance evaluation was required in two domains, namely on the architectural level to evaluate individual kernels, that are the low-level building blocks, and on the system level to evaluate radar signal processing algorithms, that are the high-level building blocks of the processing chain. Consequently, a set of low-level and high-level metrics were defined and utilized to evaluate the performance of the radar signal processing GPU implementations. The low-level metrics are based on common metrics for GPU performance optimization and general computational analysis. The high-level metrics characterize the throughput and latency of the signal processing that is performed by the individual algorithms and the processing chain.

The low-level level metrics were applied both analytically and experimentally to both predict and measure kernel performance and performance limiting factors. The low-level metrics were mainly useful for analyzing individual kernel performance during optimization, to indicate where to focus optimization, and to also indicate whether additional optimization is expected to yield further improvement at all. The high-level metrics were mainly useful for comparing the efficiency of different algorithmic variants, such as a time-domain and a frequency-domain variant, as these metrics are independent of the underlying implementation details. The high-level metrics were also useful for characterizing the absolute signal processing performance of the final radar signal processing chain.

The GBF also proved invaluable with respect to the performance evaluation, due to the extensive experimental validation that is commonly associated with GPU implementation and optimization efforts. The practical upper performance limits that were established with the aid of the microbenchmarks were also very useful in conjunction with the low-level metrics in order to determine the level of optimization for a given kernel.

7.4 ARCHITECTURAL EFFECTS ON THE GPU

The most prominent effects with a major performance impact, that were observed on the GPU architecture for the radar application, are listed in Table 7.1. The majority of the major effects that were observed are known and documented in the NVIDIA CUDA C Programming Guide [32] and the CUDA C Best Practices Guide [33], and were confirmed to be valid in practice. However, partition camping is not explicitly documented in these guides, but references were found in other literature [25, 50, 51, 52]. The asymmetry in the trend that was observed in the performance for basic mathematical operations was expected, based on documented instruction throughputs, although the floating-point multiplication contribution of the SFUs was not clearly documented.

Table 7.1: Prominent architectural effects with relevant techniques that were successfully applied.

Effect	**Technique**
Low performance for small bursts	None
Poor global memory coalescing	Use padding / TMEM / SMEM / CMEM
High global memory latency *	If data reused, use TMEM / SMEM / CMEM Otherwise, none
Shared memory bank conflicts for general 64-bit access per thread **	If limited by bank conflicts, use TMEM / CMEM
Partition camping	Diagonal block ordering + Block index broadcast
Asymmetric performance for basic math operations	Perform common reciprocals on CPU instead Ensure multiply-add is used where possible
HtoD and DtoH are slow	Minimize HtoD and DtoH Use pinned buffers Overlap data transfer and execution
Intrinsic functions do not consistently provide better performance	Validate empirically
FFT/IFFT library performance highly dependent on exact FFT length †	Padding + Snap FFT length

*Global memory has an implicit cache on newer architectures, which is also expected to be effective if data is reused.
**Alleviated on newer architectures.
†The overlap-save method could also be effective.

The low-level metrics that were defined were used to analyze algorithmic characteristics and kernel performance. Coarse pruning of the optimization space was performed using the metrics to determine whether kernels are expected to be bound by computation or bandwidth. For bandwidth limited kernels, access patterns were analyzed to determine optimization paths with respect to memory access, where a variety of memory access optimization options are available in the deep GPU memory hierarchy. Shared memory, texture memory, and constant memory were most frequently employed, in appropriate circumstances, to realize performance benefits where pure global memory implementations are suboptimal. A lightweight threading model was also adopted, with occupancy, global memory coalescing, shared memory bank conflicts, and spatial locality as the primary drivers with respect to the thread block layout and overall kernel design.

7.5 FUTURE WORK

A wide variety of aspects have been considered for the software-defined implementation of radar signal processing on GPUs, but a number of additional aspects are recommended for future study. The overlap-save method may be implemented for the FDFIR variant of the DPC to improve overall performance with long input lengths, and to reduce the overhead of wasted computation in cases where the FFT length is snapped to a longer length. The GPU implementation of other CFAR algorithms beyond the pervasive, but overly simplistic CA-CFAR, which has known practical issues, remains. The data model may be extended to include another dimension, by processing data for multiple receiver elements. Adding additional dimensions is expected to be beneficial to GPU implementation owing to a potential increase in burst size and data-parallelism. A heavyweight threading model may also be explored, where the lightweight threading model was used for this research.

Further optimization of the GPU radar signal processing chain is possible with respect to combining operations within the chain. Low AI kernels that perform adjacent operations in the chain may be merged into a kernel with higher AI to improve overall performance. Potential kernels that may be merged for the existing radar signal processing chain implementation include the DPC scale complex kernel with the CT kernel, the DF pad kernel with the window function kernel, the DF scale complex kernel with the ENV kernel, and the CFAR interference estimation kernel with the detection mask generation kernel. Furthermore, certain operations may be eliminated altogether, such as the scale complex kernel that can be invoked once to perform the scaling for the DPC and DF transforms, instead of once per function.

Further investigation using the latest CUDA release and the latest GPU architectures, such as Fermi and Kepler, is recommended, as overall performance improvements are expected. In addition, a variety of limitations on the Tesla architecture have been addressed and additional features that may benefit the radar signal processing application have been added. On the newer architectures, global memory is cached, which implicitly diminishes the negative effects of poor global memory coalescing and partition camping, in cases where data is reused. With respect to shared memory, 64-bit access per thread is possible, without generating bank conflicts. Two copy engines are supported, which allow for overlapping of HtoD transfers with DtoH transfers, in addition to overlap of data transfer with computation. This feature can potentially alleviate the negative impact on latency that was observed when using asynchronous transfers, under certain conditions, with the single copy engine on the Tesla C1060.

Concurrent kernel execution is also supported on newer architectures where kernels executed in different CUDA streams may overlap execution, under certain conditions. Furthermore, buffers that are allocated for full texture support, that includes addressing and filtering modes, using CUDA arrays or using `cudaMallocPitch`, support overlapping during data transfer, which was not available on the Tesla. ECC memory is also supported, although this feature is not considered a critical requirement in general for the radar application, as mission-level algorithms typically integrate over several processing intervals before decisions are made, negating the effects of rare, random memory errors during data-independent processing. Single-precision floating-point math is also fully IEEE compliant, where only double-precision math is fully compliant on the Tesla. Lastly, it should be noted that peak memory bandwidth is increasing at a lower rate than peak computational performance, according to the current trend with new architectures. This indicates that high AI is becoming increasingly important with new generation GPU hardware.

REFERENCES

[1] C. Venter, H. Grobler, and K. AlMalki, "Implementation of the CA-CFAR Algorithm for Pulsed-Doppler Radar on a GPU Architecture," in *2011 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)*, Amman, Jordan, Dec. 2011, pp. 233–238.

[2] M. Skolnik, *Introduction to Radar Systems*, 2nd ed. Singapore: McGraw-Hill, 1981.

[3] M. Richards, J. Scheer, and W. Holm, *Principles of Modern Radar: Basic Principles*. Raleigh, NC: SciTech Publishing, Inc., 2010.

[4] M. Skolnik, *Radar Handbook*, 3rd ed. New York, NY: McGraw-Hill, 2008.

[5] M. Richards, *Fundamentals of Radar Signal Processing*. New York, NY: McGraw-Hill, 2005.

[6] The MathWorks, Inc. MATLAB®. [Online]. Available: http://www.mathworks.com

[7] J. Lebak, A. Reuther, and E. Wong, "Polymorphous Computing Architecture (PCA) Kernel-Level Benchmarks," MIT Lincoln Laboratory, Lexington, MA, Tech. Rep. PCA-KERNEL-1, 2005.

[8] J. Owens, D. Luebke, N. Govindaraju, M. Harris, J. Krüger, A. Lefohn, and T. Purcell, "A Survey of General-Purpose Computation on Graphics Hardware," *Computer Graphics Forum*, vol. 26, no. 1, pp. 80–113, Mar. 2007.

[9] N. Wilt, *The CUDA Handbook: A Comprehensive Guide to GPU Programming*. Upper Saddle River, NJ: Addison-Wesley, 2013.

[10] E. Lindholm, M. Kilgard, and H. Moreton, "A User-Programmable Vertex Engine," in *Proc. 28th Annual Conference on Computer Graphics and Interactive Techniques*, New York, NY,

Aug. 2001, pp. 149–158.

[11] GPGPU.org. (2013) General-Purpose Computation on Graphics Hardware. [Online]. Available: http://gpgpu.org/

[12] NVIDIA Corporation. (2013) Cg Toolkit. [Online]. Available: https://developer.nvidia.com/cg-toolkit

[13] Microsoft Corporation. (2013) HLSL. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/bb509561(v=vs.85).aspx

[14] OpenGL.org. (2013) OpenGL. [Online]. Available: http://www.opengl.org/

[15] Microsoft Corporation. (2013) Direct3D. [Online]. Available: http://msdn.microsoft.com/en-us/library/windows/desktop/hh309466(v=vs.85).aspx

[16] W. Mark, R. Glanville, K. Akeley, and M. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 896–907, Jul. 2003.

[17] J. Kessenich, D. Baldwin, and R. Rost. (2012) The OpenGL Shading Language. [Online]. Available: http://www.opengl.org/registry/doc/GLSLangSpec.4.30.6.pdf

[18] I. Buck, "Brook Spec v0.2," Stanford University, CA, Tech. Rep., Oct. 2003.

[19] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan, "Brook for GPUs: Stream Computing on Graphics Hardware," *ACM Trans. Graph.*, vol. 23, no. 3, pp. 777–786, Aug. 2004.

[20] NVIDIA Corporation. (2010) CUDA Zone. [Online]. Available: http://www.nvidia.com/object/cuda_home_new.html

[21] J. Sanders and E. Kandrot, *CUDA by Example: An Introduction to General-Purpose GPU Programming*. Upper Saddle River, NJ: Addison-Wesley, 2010.

[22] NVIDIA Corporation. (2013) CUBLAS. [Online]. Available: http://docs.nvidia.com/cuda/cublas/index.html

[23] ——. (2013) CUFFT. [Online]. Available: http://docs.nvidia.com/cuda/cufft/index.html

[24] Khronos. (2013) OpenCL™. [Online]. Available: http://www.khronos.org/opencl/

[25] J. Pettersson and I. Wainwright, "Radar Signal Processing with Graphics Processors (GPUs)," Master's thesis, Uppsala University, Jan. 2010.

[26] K. Karimi, N. Dickson, and F. Hamze, "A Performance Comparison of CUDA and OpenCL," *Computing Research Repository (CoRR)*, vol. abs/1005.2581, 2010. [Online]. Available: http://arxiv.org/abs/1005.2581

[27] K. Komatsu, K. Sato, Y. Arai, K. Koyama, H. Takizawa, and H. Kobayashi, "Evaluating Performance and Portability of OpenCL Programs," in *Proc. 5th International Workshop on Automatic Performance Tuning (iWAPT)*, Berkeley, CA, Jun. 2010.

[28] J. Fang, A. Varbanescu, and H. Sips, "A Comprehensive Performance Comparison of CUDA and OpenCL," in *Proc. 2011 International Conference on Parallel Processing (ICPP)*, Sep. 2011, pp. 216–225.

[29] NVIDIA Corporation. (2013) Tesla™. [Online]. Available: http://www.nvidia.com/object/tesla-supercomputing-solutions.html

[30] AMD. (2013) FirePro™. [Online]. Available: http://www.amd.com/us/products/workstation/graphics/firepro-remote-graphics/Pages/gpu_compute.aspx

[31] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym, "NVIDIA Tesla: A Unified Graphics and Computing Architecture," *IEEE Micro*, vol. 28, no. 1, pp. 39–55, Apr. 2008.

[32] NVIDIA Corporation. (2012) NVIDIA CUDA C Programming Guide. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf

[33] ——. (2012) CUDA C Best Practices Guide. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_C_Best_Practices_Guide.pdf

[34] M. Papadopoulou, M. Sadooghi-Alvandi, and H. Wong, "Micro-benchmarking the GT200 GPU," Computer Group, ECE, University of Toronto, Ontario, Canada, Tech. Rep., 2009.

[35] NVIDIA Corporation. (2013) The CUDA Compiler Driver NVCC. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUDA_Compiler_Driver_NVCC.pdf

[36] N. Whitehead and A. Fit-Florea, "Precision & Performance: Floating Point and IEEE 754 Compliance for NVIDIA GPUs," NVIDIA Corporation, White Paper, 2011. [Online]. Available: http://docs.nvidia.com/cuda/pdf/Floating_Point_on_NVIDIA_GPU.pdf

[37] Vasily Volkov. (2014) Better Performance at Lower Occupancy. [Online]. Available: http://www.cs.berkeley.edu/~volkov/volkov10-GTC.pdf

[38] M. Bernaschi, A. D. Lallo, A. Farina, R. Fulcoli, E. Gallo, and L. Timmoneri, "Use of a Graphics Processing Unit for Passive Radar Signal and Data Processing," *IEEE Aerosp. Electron. Syst. Mag.*, vol. 27, no. 10, pp. 52–59, Oct. 2012.

[39] K. Szumski, M. Malanowski, J. Kulpa, W. Porczyk, and K. Kulpa, "Real-time Software Implementation of Passive Radar," in *Proc. 6th European Radar Conference (EuRAD)*, Rome, Sep. 2009, pp. 33–36.

[40] T. Thorson and G. Akers, "Near Real-Time Simultaneous Range and Velocity Processing in a Random Noise Radar," in *Proc. 2012 IEEE Radar Conference (RADARCON)*, Atlanta, GA, May 2012, pp. 585–590.

[41] J. Fowers, G. Brown, J. Wernsing, and G. Stitt, "A Performance and Energy Comparison of Convolution on GPUs, FPGAs, and Multicore Processors," *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, pp. 25:1–25:21, Jan. 2013.

[42] A. LaChance, "AUDIO ON THE GPU: REAL-TIME TIME DOMAIN CONVOLUTION ON GRAPHICS CARDS," Master's thesis, Appalachian State University, May 2011.

[43] J. Belloch, A. Gonzalez, F. J. Martínez-Zaldívar, and A. Vidal, "Real-time massive convolution for audio applications on GPU," *The Journal of Supercomputing*, vol. 58, no. 3, pp. 449–457, Dec. 2011.

[44] S. A. Umairy, A. van Amesfoort, I. Setija, M. van Beurden, and H. Sips, "On the Use of Small 2D Convolutions on GPUs," in *Proc. 37th International Symposium on Computer Architecture (ISCA)*, ser. 1st Workshop on Applications for Multi and Many Core Processors (A4MMC),

Saint Malo, France, Jun. 2010, pp. 52–64.

[45] V. Podlozhnyuk, "Image Convolution with CUDA," NVIDIA Corporation, NVIDIA CUDA SDK Application Note, 2007.

[46] G. Carlson, *Signal and Linear System Analysis*, 2nd ed. Hoboken, NJ: John Wiley & Sons, 1998.

[47] A. Smirnov and T. Chiueh, "An Implementation of a FIR Filter on a GPU," Experimental Computer Systems Lab, Stony Brook University, Tech. Rep., 2005. [Online]. Available: http://www.ecsl.cs.sunysb.edu/fir/

[48] A. Kerr, D. Campbell, and M. Richards, "GPU Performance Assessment with the HPEC Challenge," in *Proc. 12th Annual HPEC Workshop*, Lexington, MA, Sep. 2008.

[49] HPEC Challenge. (2013) HPEC Challenge Benchmark Suite. [Online]. Available: http://www.omgwiki.org/hpec/files/hpec-challenge/index.html

[50] G. Ruetsch and P. Micikevicius, "Optimizing Matrix Transpose in CUDA," NVIDIA Corporation, NVIDIA CUDA SDK Application Note, 2009.

[51] M. Bader, H. Bungartz, D. Mudigere, S. Narasimhan, and B. Narayanan, "Fast GPGPU Data Rearrangement Kernels using CUDA," *Computing Research Repository (CoRR)*, vol. abs/1011.3583, 2010. [Online]. Available: http://arxiv.org/pdf/1011.3583

[52] A. Aji, M. Daga, and W. Feng, "Bounding the Effect of Partition Camping in GPU Kernels," in *Proc. 8th ACM International Conference on Computing Frontiers (CF)*, Ischia, Italy, May 2011.

[53] J. Cooley and J. Tukey, "An Algorithm for the Machine Calculation of Complex Fourier Series," *Mathematics of Computation*, vol. 19, pp. 297–301, Apr. 1965.

[54] N. Govindaraju, B. Lloyd, Y. Dotsenko, B. Smith, and J. Manferdelli, "High Performance Discrete Fourier Transforms on Graphics Processors," in *Proc. 2008 ACM/IEEE Conference on Supercomputing (SC)*, Austin, Texas, Nov. 2008.

[55] B. Lloyd, C. Boyd, and N. Govindaraju, "FAST COMPUTATION OF GENERAL FOURIER TRANSFORMS ON GPUS," in *Proc. 2008 IEEE International Conference on Multimedia and*

Expo (ICME), Hannover, Jun. 2008, pp. 5–8.

[56] K. Moreland and E. Angel, “The FFT on a GPU,” in *Proc. 2003 ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (GH)*, San Diego, California, Jul. 2003, pp. 112–119.

[57] S. Bash, D. Carpman, and D. Holl, “Radar Pulse Compression Using the NVidia CUDA Framework,” in *Proc. 12th High Performance Embedded Computing (HPEC) Workshop*, Sep. 2008. [Online]. Available: http://www.ll.mit.edu/HPEC/2008/

[58] X. Wenli, C. Hao, and Z. Mo, “Research and Realization of Software Radar Signal Processing Based on Intel MKL,” in *Proc. 1st International Conference on Computer and Management (CAMAN)*, Wuhan, May 2011, pp. 28–33.

[59] HPEC Challenge. (2013) Constant false-alarm rate detection. [Online]. Available: http://www.omgwiki.org/hpec/files/hpec-challenge/cfar.html

[60] “Tesla C1060 Computing Processor Board,” NVIDIA Corporation, Board Specification BD-04111-001_v03, Sep. 2008. [Online]. Available: http://www.nvidia.com/docs/IO/56483/Tesla_C1060_boardSpec_v03.pdf

[61] GNU. (2011) BASH. [Online]. Available: http://www.gnu.org/software/bash/

[62] NVIDIA Corporation. (2014) CUFFT Library Programming Guide. [Online]. Available: http://docs.nvidia.com/cuda/pdf/CUFFT_Library.pdf

[63] F. C. Crow, “Summed-Area Tables for Texture Mapping,” in *SIGGRAPH ’84 Proc. 11th Annual Conf. on Computer Graphics and Interactive Techniques*, New York, NY, USA, Jul. 1984, pp. 207–212.

[64] P. Viola and M. Jones, “Robust real-time object detection,” in *Proc. 2nd Intl. Workshop on Statistical and Computational Theories of Vision*, Vancouver, Canada, Jul. 2001.

[65] G. Blelloch, “Scans as primitive parallel operations,” *IEEE Trans. Comput.*, vol. 38, pp. 1526–1538, Nov. 1989.

[66] ——, *Vector Models for Data-Parallel Computing*. Cambridge, Massachusetts: The MIT

Press, 1990.

[67] D. Horn, *Stream reduction operations for GPGPU applications*. Upper Saddle River, NJ: Addison-Wesley, 2005, ch. 36, pp. 573–589, in GPU Gems 2, M. Pharr.

[68] S. Sengupta, A. Lefohn, and J. Owens, “A work-efficient step-efficient prefix-sum algorithm,” in *Proc. Workshop on Edge Computing Using New Commodity Architecture*, Chapel Hill, North Carolina, May 2006, pp. 26–27.

[69] CUDPP. (2010) CUDA data parallel primitives library. [Online]. Available: http://code.google.com/p/cudpp

APPENDIX A

DERIVATION OF LOW-LEVEL METRICS

The low-level metrics that are defined in Table 5.5 are derived for the GPU kernel implementations to perform the necessary kernel analysis. The kernel analysis for each radar signal processing algorithm is presented in Section 5.5.

Selected CFAR kernels are used to provide an example of the occupancy calculation in Table A.1, which is based on the architectural limits shown in Table 5.4. To illustrate the derivation of the remaining low-level metrics, an example of the peak theoretical computational throughput calculation is shown in Table A.2 for the same set of CFAR kernels.

Table A.1: Example of occupancy calculation for selected CFAR kernels.

CFAR Kernel	Threads / Block*	Registers / Thread**	SMEM / Block**	Registers / Block†	SMEM / Block†	Warps / Block†	Blocks / SM††	Warps / SM††	Occupancy††
Naïve (GMEM)	256	10	164 bytes	2560	512 bytes	8	4	32	1 (100%)
Seg. Rows (SMEM)	256	10	3244 bytes	2560	3584 bytes	8	4	32	1 (100%)
Seg. Cols (SMEM)	256	8	3244 bytes	2048	3584 bytes	8	4	32	1 (100%)

*Specified by programmer.
**Obtained from *ptxas* compiler output. Dynamically allocated SMEM (if any) should be added to this figure, which accounts for statically allocated SMEM only.
†Determined by block size, warp size and architectural memory allocation granularity and unit size.
††Determined by most restrictive block allocation factor according to architectural limits for maximum number of threads/SM, warps/SM, registers/SM, and SMEM/SM.

Table A.2: Example of peak theoretical computational throughput calculation for selected CFAR kernels, with a CFAR window containing N cells with N_{dop} cells in the Doppler dimension and N_{rng} cells in the range dimension.

CFAR Kernel	$\mathbf{B_r}$ bytes*	$\mathbf{B_w}$ bytes*	$\mathbf{W_{dmem}}$ bytes*	**Scalar +** FLOPs*	**Scalar ×** FLOPs*	$\mathbf{W_c}$ FLOPs*	**AI** FLOPs/byte	$\lim_{N\to\infty}$ **AI** FLOPs/byte	$\mathbf{AI.BW_{peak}}$ GFLOPS	$\mathbf{T_c(peak)}$ GFLOPS
Naïve (GMEM)	$4N$	4	$4(N+1)$	$N-1$	1	N	$\frac{N}{4(N+1)}$	0.25	25.5	25.5
Seg. Rows (SMEM)	12	4	16	$N_{rng}-1$	0	$N_{rng}-1$	$\frac{N_{rng}}{16}-\frac{1}{16}$	∞	∞	933.0
Seg. Cols (SMEM)	12	4	16	$N_{dop}-1$	1	N_{dop}	$\frac{N_{dop}}{16}$	∞	∞	933.0

*Normalized to per sample.

APPENDIX B

INITIAL RESULTS ON GEFORCE GTX 480

An NVIDIA GeForce GTX 480 GPU, which is a second-generation CUDA device, was also present in the host system that was utilized. The main differences between the GTX 480 and the Tesla C1060, which is a first-generation CUDA device that was used predominantly throughout the research, are discussed in Section 5.2. The specifications for both GPUs are also shown in Table 5.2. Initial results are presented in this section that were generated for the radar signal processing chain on the GTX 480 with the GBF. The same function variants that were used for the radar signal processing chain on the Tesla C1060, as shown in Table 5.11, were also used to generate the indicative GTX 480 results that are presented here. All implemented variants should ideally be re-evaluated for the GTX 480 using the GBF, when thorough optimization is attempted.

The GTX 480 shows a consistent speedup over the Tesla C1060, as shown in Figure B.1 and Figure B.2. A best-case speedup in throughput of greater than 2 times is achieved for the inter-burst scheme, whereas the intra-burst scheme achieves a speedup of around 1.7 times. Note that the bandwidth between the host and device is not proportional to general GPU performance improvements on the GTX 480, as both devices use a PCIe 2.0 $\times$16 bus. With the intra-burst scheme the compute to data transfer ratio for the specific kernels that access host memory directly is reduced by the comparatively slower PCIe transfers. Conversely, the inter-burst scheme allows the slower PCIe transfers to overlap with execution of all kernels in the radar signal processing chain, where the overall compute to data transfer ratio remains high enough to hide the data transfers. It is also observed that the achieved speedup is roughly proportional to the increase in device memory bandwidth, with quick inspection also revealing that around 90% instead of only 80% of the theoretical peak is readily achievable. Due to memory constraints imposed by the smaller device memory size on the GTX 480, the largest burst size could not be evaluated for the inter-burst scheme, which requires double buffering.

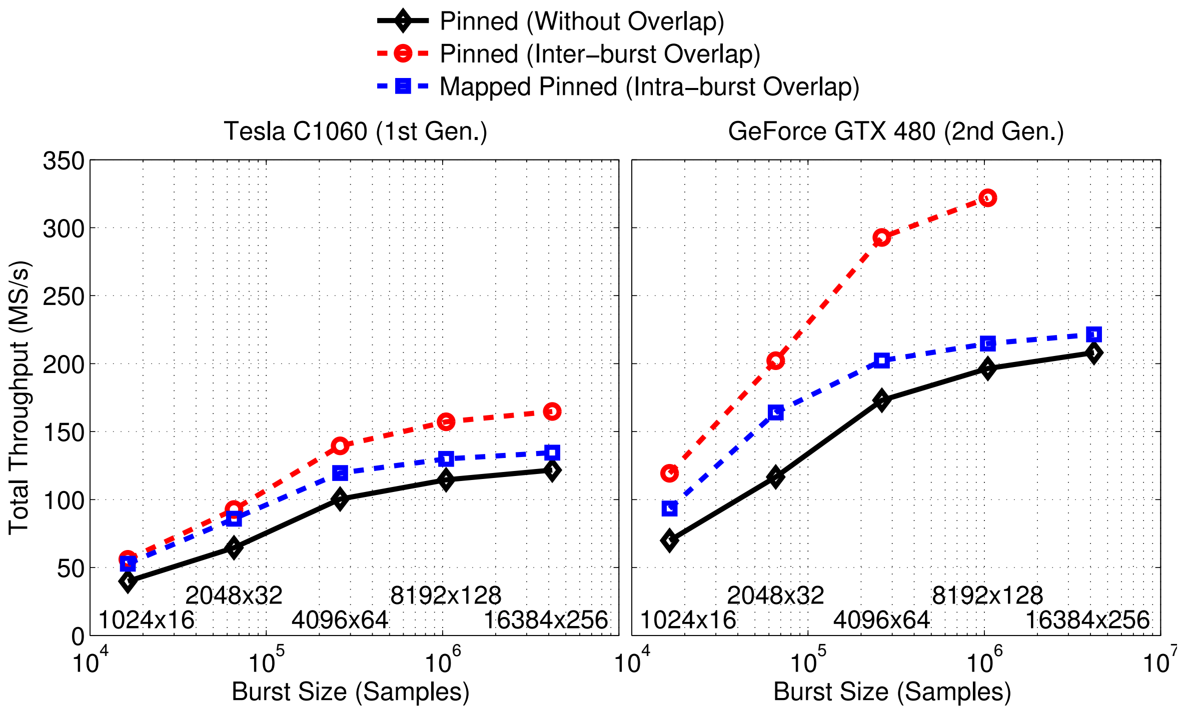

Figure B.1: Radar signal processing chain throughput (T_{t8}) for the experiment described in Section 6.4 showing initial results for the GeForce GTX 480.

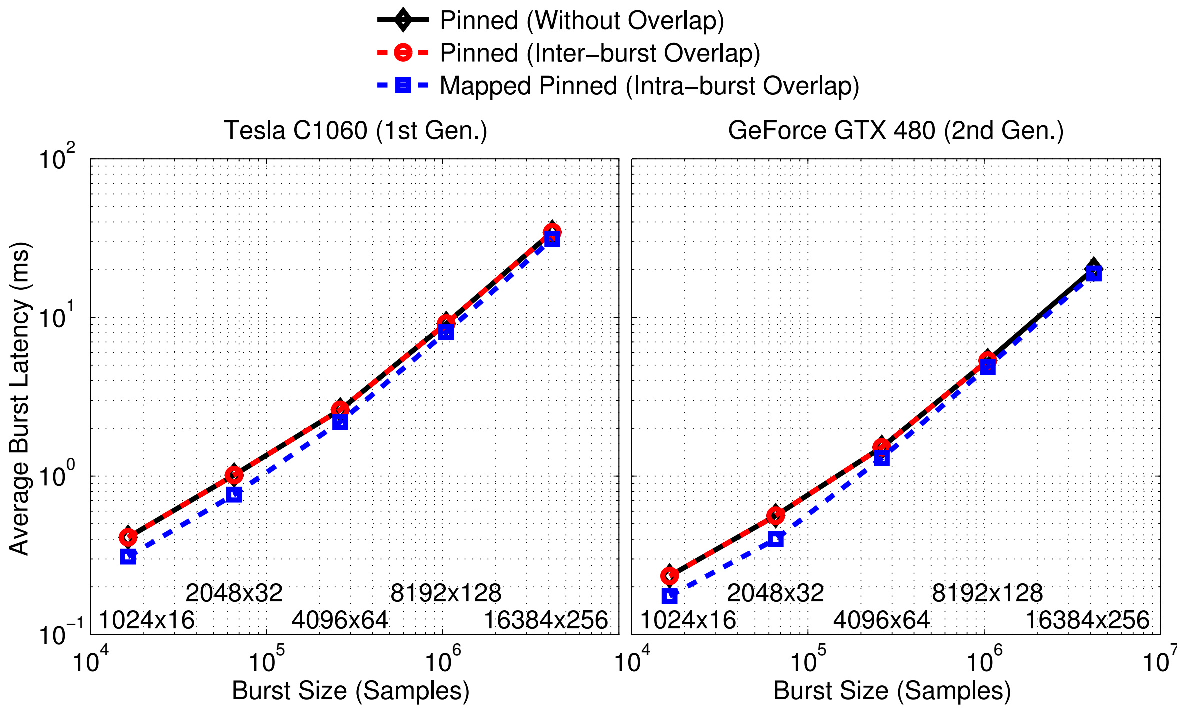

Figure B.2: Radar signal processing chain latency ($\overline{L}_{b8}$) for the experiment described in Section 6.4 showing initial results for the GeForce GTX 480.