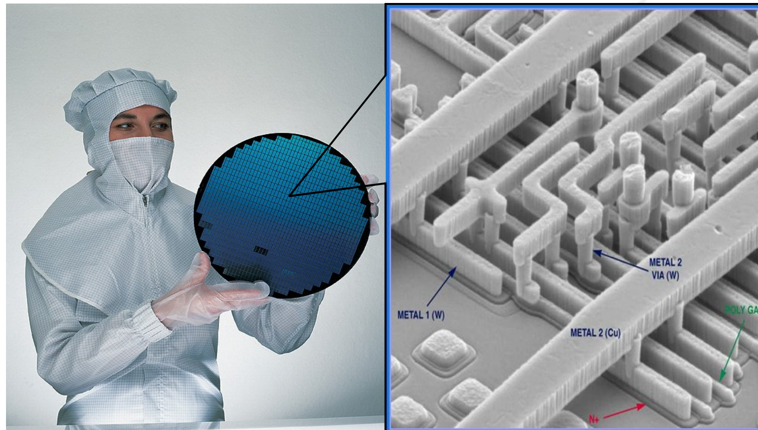
MSc Thesis

Accelerating Computational Electromagnetic Diffraction Model on Programmable Graphics Processors

by

Shams Alumairy



TU Delft
Delft University of Technology

ASML

Accelerating Computational Electromagnetic Diffraction Model on Programmable Graphics Processors

---

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING
Faculty of Electrical Engineering, Mathematics, and Computer Science
Delft University of Technology
by

Shams Alumairy

December, 2009

Parallel and Distributed Systems Group
Department of Computer Software Technology
Delft, The Netherlands

www.pds.ewi.tudelft.nl

Metrology Department
Eindhoven, The Netherlands

www.asml.com

Author:

Name: Shams A. H. Alumairy

Email: s.a.h.alumairy-1@student.tudelft.nl

Committee Members:

Chairperson	Prof. Dr. Henk Sips, Faculty EEMCS, TU Delft
University supervisor	Alexander van Amesfoort, Faculty EEMCS, TU Delft
Company supervisor	Dr. Irwan Setija, ASML
Graduation committee Member	ir. Dr. Hai Xiang Lin, Faculty EEMCS, TU Delft

MSc Presentation

Abstract

EDM, stands for "Electromagnetic Diffraction Model" is used in wafer metrology in order to deduce the quality of the photolithographic process. This numerical model is dedicated to solve sets of linear algebraic equations— i.e. electromagnetic wave equations — by means of computing Fast Fourier Transforms (FFT). The time complexity of EDM is possessed by computing the 3D electromagnetic wave equation that is solved by 2D convolution. It solo consumes about 50% of the total solving time of this method on serial computers. Therefore, in this thesis, the main focus is on accelerating these computations on massively parallel hardware. Driven by the huge numerical computing demand of this application, Graphic Processing Unit (GPU) has become the top choice to be used throughout this thesis, because of its tremendous performance. Thus, this thesis introduces a framework for the GPU-based parallel implementation and explorers the performance of solving such computations on general purpose GPUs using NVIDIA CUDA programming model. This thesis highlights modest algorithm modifications that could significantly increase the data parallelism. The overall results show that the proposed parallel algorithms have been able to fully utilize CUDA architecture features justifying the use of such technology for general purposes. It reveals that the GPU-based parallel implementation for a big enough problem size yields a speedup factor of about 6-19 times faster than its counterpart that runs serially on the CPU.

Preface

This document describes my MSc thesis research concerning accelerating an electromagnetic diffraction model using GPUs. It forms the final element required for obtaining the Master of Science degree in Computer Engineering by the Delft University of Technology. The research has been performed at Metrology Research group of ASML along with Parallel and Distributed Systems group, TU Delft.

I want to thank a number of people who have spent their time helping me with this thesis project and made it in one way or another possible. First and foremost, I would like to thank my thesis advisor, prof. Dr. Ir. Henk Sips for his patience and guidance in this project and for chairing the examination committee. I am also greatly thankful to Alexander van Amesfoort for steering my research efforts with his motivating suggestions, supports and numerous reviews in many stages of my work till the final outcome.
I am also grateful for the opportunity to be part of the Metrology Research group at ASML. Many thanks to my thesis enthusiastic supervisor Irwan Setija, for his always positive attitude, the helpful discussions about the application related topics as well as for the continuous feedbacks throughout the writing phase of the thesis, and also for managing everything I needed at ASML. Special thanks goes out as well to Martijn van Beurden for his down-to-earth advices and for continuous incentive to ”go further”. He was always available to help me deal with the details of physics theory of the application. I also thank Mark van Kraaij for his inspiring and creative ideas. I would like as well to thank all my great colleagues at this group for the support and the nice working atmosphere, and for all the fun and small talks we made during our many lunch and coffee breaks.
I would like also to thank all my teachers for the massive amount of engineering knowledge, insights and discussions. For sure, their contributions have been of great value for me; have improved my analytical thinking and motivated me for the fields of engineering, in particular computer engineering.
Furthermore, I would like to express my gratitude to my parents and my siblings for being present through every step of my life, and for their gentle nudges when needed; and I also thank them for having always faith in me, supporting me throughout my academic career, especially when my self-motivation was nowhere to be found, and for being patient during the years of my study. Last but not least, my warmest thanks go to my friends, for their cooperation; they always stood there to keep my spirit up, presenting me the opportunity for a successful career and life.
Most importantly, however, I owe all what I have been obtaining to my God “Allah”, who has always been shining my life. Now I can only reaffirm the following words – “*Our God, whatever grace we have, it is from you. No god but you*”

Shams Alumairy
Delft, The Netherlands
December 2009

Contents

Preface v

Contents vii

List of Figures ix

List of Tables xi

List of Acronyms xii

1 Introduction 1

1.1. Thesis Objectives 1

1.2. Thesis Organization 2

2 Parallel Computing and Architectures Background 3

2.1. Multi-Core Processors 3

2.2. CELL Processors 5

2.3. GPGPU 6

2.3.1. ATI GPU 7

2.3.2. NVIDIA GPU 8

2.4. GPU Hardware Model 9

2.5. GPU Programming Model 13

2.6. Summary 17

3 The Application Description 20

3.1. Lithography 20

3.2. Diffraction Grating Model 21

3.3. The Reconstruction Loop 22

3.4. The Electromagnetic Diffraction Model Algorithm 22

3.4.1. 2 Dimension Convolution Algorithm 23

4 CPU-Based Serial Implementations of EDM Algorithm 26

4.1. Serial Fortran-Based Implementation of EDM 26

4.1.1. Computing the solution of the Linear System 27

4.2. Performance Analysis of The Serial Implementation 28

4.3. Serial C-Based Implementation of Linear Solver 32

4.4. Results Precision Validation 35

4.5. Summary 36

5 GPU-Based Parallel Implementations and Optimizations 38

5.1. 2D FFT-Based Approach 39

5.2. 1D FFT-Based Approach 43

5.3. CUDA-Based Optimizations 48

5.3.1. Optimizing The Execution Configuration 49

5.3.2. Using The Shared On-Chip Memory 52

5.4. Multiple Angles Parallel Implementation 54

5.5. Summary 57

6 Experimental Results Analysis 59

6.1. Measurements Setup 59

6.2. Performance Analysis of The Parallel Implementation 60

6.3. Comparison of Single Angle version on GPU versus on one CPU Core 65

6.4. Comparison of Multiple Angles of Incidence version on GPU vs on a Cluster of CPUs 67

6.5. Summary 69

7 Conclusion and Future Work 72

Bibliography 75

List of Figures

Figure 2.1: Cell System Architecture 6
Figure 2.2: NVIDIA Tesla 10-Series Architecture........ 9
Figure 2.3: Diagram of one Tesla 10-Series Multiprocessor........ 10
Figure 2.4: Tesla Memory Model interconnected to a CPU 10
Figure 2.5: Diagram of one Stream Multiprocessor of Fermi Architecture 12
Figure 2.6: Representation of CUDA Programming Model 14
Figure 2.7: Increment Array Example in C and CUDA........ 14
Figure 2.8: Representation of CUDA Threads Blocks mapped on CUDA Memory Model........ 15

Figure 3.1: Lithography Process 21
Figure 3.2: Diagram of Grating Shape Parameters 21
Figure 3.3: Reconstruction Loop Diagram 22
Figure 3.4: Diagram of Grating Layers........ 23
Figure 3.5: Flowchart of 2D Convolution Algorithm 24
Figure 3.6: Data Structure Representation of the Electromagnetic Vector Field and Derived Field........ 24

Figure 4.1: Block Diagram of EDM Implementation in FORTRAN 26
Figure 4.2: Benchmark of the Serial Fortran-based Code, inclusive the associated subroutines........ 30
Figure 4.3: Benchmark of the Serial Fortran-based Code, exclusive the associated subroutines 30
Figure 4.4: A representation of multi-dimensional arrays........ 30
Figure 4.4: Benchmark of the Serial C-based Code, exclusive the associated subroutines 35

Figure 5.1: Schematic overview of GPU computations workflow of trivial Implementation........ 40
Figure 5.2: A summary plot of the GPU time percentage of Algorithm 5.3........ 41
Figure 5.3: Schematic representation of the levels of Parallelism........ 42
Figure 5.4: Conceptual Graph for solving 2D FFT using 1D FFT 43
Figure 5.5: Scheme for solving 2D FFT using 1D FFT and Matrix Transpose........ 43
Figure 5.6: A summary plot of the GPU time percentage of Algorithm 5.4........ 45
Figure 5.7: Scheme for solving 2D convolution algorithm using 1D FFT-based Approach........ 46
Figure 5.8: A summary plot of the GPU time percentage of Algorithm 5.5........ 47
Figure 5.9: CUDA execution representation of applying 3D blocks scheme 49
Figure 5.10: GPU exection time overview of Problem Case 1; after applying 3D blocks scheme 50
Figure 5.11: CUDA execution representation for Problem Case 1,2,3 by applying 1D Grids of 2D Blocks scheme. 51
Figure 5.12: CUDA execution representation for Problem Case 4,5 by applying 1D Grids of 2D Blocks scheme.... 51
Figure 5.13: CUDA execution representation of applying general scheme........ 52
Figure 5.14: Schematic Overview of Multiple Angles Application version for different Angles of Incident running on CPU and CPU-GPU systems........ 63
Figure 5.15: CUDA execution representation of Multiple Angles version by applying general scheme........ 56
Figure 5.16: A layout of 3x(NLayers+1) matrices of size HeightxWidth for N angles........ 57

Figure 6.1: Performance Overview of the all problem cases using different algorithms 61
Figure 6.2: Performance Overview of the all problem cases using Algorithm 5.6 with different CUDA execution configurations 64
Figure 6.3: Speedup of using different GPU-based algorithms versus their equivalent CPU-based serial algorithm, for the all problem cases; the time is exclusive the data transferring........ 65
Figure 6.4: Overall Performance of 2D Convolution Computations of single angle version on GPU versus on one CPU core 66

Figure 6.5: Speedup of 2D Convolution Computations of Multiple Angles version per Single Angle on GPU versus on one Isolated CPU 67
Figure 6.6: Speedup of 2D Convolution Computations of Single Angle on GPU when running one Angle versus 7 Angles of Incidence in Parallel in comparison to Isolated CPU 68
Figure 6.7: Speedup of 2D Convolution Computations of Single Angle on GPU when running one Angle versus 7 Angles of Incidence in Parallel in comparison to CPU in Presence of Memory Interference 69
Figure 6.8: Speedup of 2D Convolution Computations at running 7 Angles of Incidence in parallel on GPU versus 1 Angle on CPU 71

List of Tables

Table 2.1: Comparison between GT200 and Fermi Architectures 12
Table 2.2: Comparison between different High Performance Computing Solutions 19

Table 4.1: Profiling Results of Fortran Code 29
Table 4.2: A range of Computations Cases 32
Table 4.3: Profiling Results of C-based Code 34

Table 5.1: Speedup Statistics of using Algorithm 5.4 vs Algorithm 5.3 44
Table 5.2: Speedup of using Algorithm 5.5 vs Algorithm 5.3 and 5.4 46
Table 5.3: Amount of Storage needed for each Computations Case, and the Availability of the Shared Memory per MP 53
Table 5.4: Overview of the speedup achieved by applying all the Parallelization Techniques versus the Trivial Straightforward Implementations, on GPU 53

Table 6.1: An overview over the Threads Structures on GPU for Problem Case 1, 2, and 3 62
Table 6.2: Timing Results for one Angle of Incidence of different GPU-based Algorithms over a range of Problem Cases 64
Table 6.3: Speedup of GPU-based Implementations versus CPU-based Implementations 65
Table 6.4: General Overview of GPU and CPU exection Times and the Speedup of 2D Convolution Computations on GPU vs CPU and its estimated influence on entire EDM Application 66
Table 6.5: Speedup of 2D Convolution Computations of Single Angle on GPU when running one Angle and 7 angles of incidence in parallel in comparison to one Isolated CPU. 68
Table 6.6: A comparison of a cluster of CPUs versus GPUs 69

List of Acronyms

ASMP	ASymmetric Multi-Core processor
ALU	Arithmetic Logic Unit
BARC	Bottom Anti Reflection Coating
BE	Broadband Engine
BLAS	Basic Linear Algebra Subprograms
CD	Critical Dimensions
CPU	Central Processing Unit
CTM	Close to the Metal
CUDA	Compute Unified Device Architecture
CUBLAS	CUDA Basic Linear Algebra Subprograms
CUFFT	CUDA Fast Fourier Transform library
CWD	compute work distribution
1D, 2D, 3D	1 Dimension, 2 Dimension, 3 Dimension
DFT	Discrete Fourier Transforms
DP	Double Precision
DRAM	Dynamic Random Access Memory
EDM	Electromagnetic Diffraction Model
E^{inc}	incident Electromagnetic Field
E^{tot}	total Electromagnetic Field
FFT	Fast Fourier Transform
FLOP	Floating Point Operation
GB	Giga Byte
GDDR3	Graphics Double Data Rate 3
GFLOPs	Giga Floating Point Operation per second
GHz	Giga Hertz
GPU	Graphics Processing Unit
GPGPU	General Purpose Graphics Processing Unit
HPC	High Performance Computing
IC	Integral Circuit
ID	Identifier
ISA	Instruction Set Architecture
LAPACK	Linear Algebra Package
MFC	Memory Flow Controller
MKL	Math Kernel Library
MP	Multi Processor
MatVec	Matrix-Vector
OS	Operating System
PPE	Power Processing Element
RAW	Read After Write
SIMD	Single Instruction Multiple Data
SIMT	Single Instruction Multiple Thread
SMP	Symmetric Multi-Core processor
SP	Single Precision
SPE	Synergistic Processing Element
SWA	Side-Wall-Angle

1 Introduction

This thesis was carried out within the ASML Metrology Research group in the framework of wafer metrology. A metrology tool has been developed by this group— to be used during the photolithography process, in order to obtain optimum focus and exposure control by measuring and analyzing the linewidth on a wafer with sub nanometer precision. It is built based upon the idea of angular resolved scatterometry technique. This technique requires simulating the light scattering from a grating structure. This structure is illuminated by a focused light beam, which is described by sum of a plane waves. The scattering of plane-wave by a grating structure can be explained by the wave nature of the electromagnetic field.

The scattered electromagnetic field is calculated via rigorous diffraction grating models. Those models are used over range of structures and over a range of materials. EDM (Electromagnetic Diffraction Model) is a forward diffraction grating model that is aimed at computing the diffracted field by a 2D periodic grating structure, which is expressed by sets of linear algebraic equations— i.e. electromagnetic wave equations, by applying a numerical solver method. These equations are solved by 2D and 1D convolutions by means of Fast Fourier Transforms. The time complexity of EDM is possessed by computing the 3D electromagnetic wave equation. It solo consumes about 50% of the total solving time of this method on sequential computers.

Beside developing a fast numerical model, ASML is interested in inspecting if more improvements can be achieved by implementing this model on high performance hardware architecture and determine if that can introduce better performance and accelerate certain portions of the computations (especially the most time-consuming and data independent parts).

Throughout the recent years, the numerical applications heavily relied on parallelism to deliver significant performance improvements. Modern Graphic Processing Units (GPUs) are very powerful computational, multithreaded and highly parallelism systems on a chip. They present an abundance of computational potential that has attracted the attention of the scientific community. The key point is parallelizing these algorithms in a way that fits the intrinsic parallelism architecture of the advent GPUs. Studying the best way to implement such algorithm on a GPU platform is not trivial, since many aspects must be taken into account, as it will be seen in this thesis. Thus, the suitability of such architecture to EDM application remains an open research question.

1.1. Thesis Objectives

The main objective of this thesis emphasizes on studying the applicability of the EDM application to the many-core architecture (GPU). Thereby, it focuses on investigating if a CPU-GPU based implementation of this application is feasible and how it performs compared to the serial CPU-based implementation, for one and multiple angles of incidence. Thus, in the context of this thesis, a few research goals will be considered, as follows:

- Acquiring a thorough insight about the GPU architectures and pointing out the important key features.
- Inspecting the behavioral characteristics of EDM application in order to detect its

bottlenecks and determine which parts can be accelerated.

- Translating these parts from Fortran to C language.
- Validating the stability of the linear solver at lower precision.
- Parallelizing these portions on GPGPU using CUDA programming model.
- Identifying the performance bottlenecks, by profiling the system's behavior to evaluate the impact of the identified bottlenecks, as to enable further optimizations.
- Utilizing the computing potential available in the GPU by maximizing the data parallelism that can be offloaded to the GPU.
- Comparing the CPU-GPU based implementations to their counterpart that run serially on CPU in terms of performance, power consumption, space and cost.

1.2. Thesis Organization

As this thesis intends to study the suitability of EDM application to the General Purpose Graphic Processing Units GPGPUs and to judge the effectiveness of improvements to it, knowledge of all facets of the application and the implementation architecture is required. Thus, the second chapter outlines brief background knowledge on parallel architectures as well as presents important information on GPU programming and hardware models. A concise description of the basics of the introduced application will be given in Chapter 3. To be able to estimate the performance gains and determine the parts that can be optimized and ported to GPU, the serial implementation of the application along with its performance analysis will be explained in Chapter 4. Additionally, this chapter validates the precision of the results of this model by comparing results in single and double floating point precisions. Next, the parallel GPU-based implementations will be discussed and analyzed in details in Chapter 5. This chapter explores several different parallel algorithms that make the most efficient use of the GPU architecture and demonstrates convenient descriptions of the most important optimizations that should be taken into consideration when developing a parallel algorithm on CUDA GPU architecture. Later in Chapter 6, the performance measurements and analysis will be evaluated, as well as a comparison to the CPU-based implementations will be shown. Finally, this thesis is closed by Chapter 7, in which the conclusion drawn from this study is presented and promising directions for future research are indicated.

2 Parallel Computing and Architectures Background

Parallel computing is a form of computation in which many calculations are carried out simultaneously, operating on the principle that large problems can often be divided into smaller ones, which are then solved concurrently ("in parallel"). There are several different forms of parallelism: bit-level, instruction-level, data, and task parallelism. Parallel computers can be roughly classified according to the level at which the hardware supports parallelism.

Actually, parallelism has been employed for many years, mainly in high performance applications, but it has received growing concern recently due to the physical constraints preventing clock frequency scaling.

As power consumption, and consequently heat generation by computers, has become a concern in recent years, the parallel computing has become the dominant paradigm in computer architecture, mainly in the form of multi-core and many-core processors.

This chapter gives a short history and some background information about the parallel computing, and explores different parallel architectures while showing some important aspects of each, and it will explain in detail the target architecture and programming model used throughout this thesis.

2.1. Multi-Core Processors

The first attempt in adopting parallelism relied on optimizing the operating system by supporting multi-processing/ threading, i.e. while the processes are waiting for the resources to be available, the processor can start with the next thread. But, what if multiple threads are ready to execute and they are waiting for the processor. As a result, it was necessary to find alternative ways to boost the applications performance.

Over the last decade, the application developers heavily relied on new hardware to deliver significant performance improvements while actually keeping costs at minimum. Thus, several paradigm shifts were adapted. One of them is increasing the clock frequency per a single-core processor. However, each clock frequency increase resulted in power consumption increase and that influences the efficiency, which is actually measured by performance-per-watt. Consequently, increasing power consumption and the heat dissipation along with the limits imposed by quantum physics has made this progression increasingly less feasible.

Nevertheless, this challenge motivated to scale parallelism by increasing the number of processing units per system, (i.e. a supercomputers, grids, multi-core processors… etc). Therefore, another strategy to improve the efficiency was migrating from single-core processors to multi-processors.

Earliest Multi Processor systems were multiple standalone computers connected by a high-speed interconnect, such as Ethernet, Fiber Channel,…etc. Nowadays the traditional single-core processors are being replaced by multi-core or many-core processors, where two or multiple independent processing elements are typically integrated onto a single integrated circuit die (known as a chip multiprocessor), or they may be integrated onto multiple dies in a single chip package. A *many-core processor* is an architecture in which the number of cores is

large enough that traditional multi-processor techniques are no longer efficient— this threshold is somewhere in the range of several tens of cores- and likely requires a network on chip.

Modern multi-core processors represent an important new trend in computer architecture. They may implement architectures like superscalar, vector processing, or multithreading. They reduce the power consumption and heat generation by reducing the circuitry as well as minimize wire lengths and interconnect latencies, and save more room on the motherboard. Besides, it repeals the need to additional systems to control the several single cores. Furthermore, it is much cheaper than single cores. In short, multi-core processors enable true level of parallelism with great energy efficiency and scalability. In this way, it optimizes performance against area, power, volume, and cost.

The ability of multi-core processors to increase applications performance is strongly constrained by the use of multiple threads within applications. It relies on effective exploitation of multiple-thread parallelism. For utilizing their full potential, applications will need to move from a single to a multi-threaded model. Threads can be much smaller and still be effective, and in addition to that, the automatic parallelization is more feasible, and allows spreading the workload across multiple cores. The Operating System (OS) maps threads/processes to cores. It perceives each core as a separate processor. The OS scheduler maps threads/processes to different cores.

Moreover, the performance gain of multi-core processors is governed by memory wall, i.e. memory bandwidth and the way to get data out of memory banks and to get data into multi-core processor array, and also the memory latency. Nevertheless, there are a lot of advantages of multi-core. Cache coherency circuitry can operate at a much higher clock rate than if the signals have to travel off-chip. Cores may or may not share caches, and they may implement message passing or shared memory inter-core communication methods. With a shared on-chip memory, communication events can be reduced to just a handful of processor cycles. Therefore, with low latencies, communication delays have a much smaller impact on overall performance. However, these processing cores sharing the same system bus and memory bandwidth limits the real-world performance advantage.

The cores in multi-core processor are either symmetric or asymmetric cores. Symmetric Multi-Core processor (SMP) is an architecture that has multiple identical cores on a single chip. Every single core has the same architecture and the same capabilities, so it requires that there is an arbitration unit to give each core a specific task. The cores need to communicate with each other through common shared memory if they wish to cooperate. For instance, Intel Core 2 is an example of a symmetric multi-core processor. This processor may have either 2 cores on chip (like "Core 2 Duo") or 4 cores on chip (like "Core 2 Quad"). Each core in the Core 2 chip is symmetrical, and can function independently of one another. It requires a mixture of scheduling software and hardware to farm tasks out to each core. Core i7 processor is another example of symmetric multi-core processors designed based on Intel' Nehalem microarchitecture, see [37]. It includes four CPU cores with simultaneous multithreading, 8MB of L3 cache, and on-chip DRAM controllers.

On the other hand, the cores in ASymmetric Multi-Core processor (ASMP) might have different designs. For instance, there could be 2 general purpose cores and 2 vector cores on a single chip. IBM Cell processor, used in the Sony PlayStation 3 video game console is an example of ASMC systems (next section gives a short description to Cell processors).

The SMP is easier to implement and to program since all the cores are identical. As a result, that makes it easy to keep the development speed as it is applicable to any type of system (General usage). However, it is not proper to certain specific systems such as

Audio/video processing, data compression, and so on, since it wastes silicon and power because it is made for the general purpose, so it is less efficient than ASMP. These are reasons why ASMP has emerged.

The major problem is not to build multi-core hardware, but programming it in a way that lets mainstream applications benefit from the continued exponential growth in these processors performance. Parallel computer programs are more difficult to write than sequential ones, because concurrency introduces several new classes of potential software bugs, of which race conditions are the most common. Communication and synchronization between the different subtasks are typically one of the greatest obstacles to get good parallel program performance.

Before introducing the platform used in this thesis, a brief background on High Performance Computing (HPC) processors is given in the next sections.

2.2. CELL Processors

Cell processor is an asymmetrical multi-core processor that consists of 9 core processors on board, one general purpose processor, and 8 data-processing cores. The one multipurpose core, known as the *Power Processor Element* (PPE) controls the communication among the other cores, and distributes computing tasks to the other cores for processing. The other 8 cores are known as *Synergistic Processor Elements* (SPE), and are specially designed to have high mathematical floating-point throughput, especially with vector operations. The PPE core is much larger than the individual SPE cores. All cores, memory and I/O interfaces are connected with a ring bus. Every core can send/receive data to/from every other core's memory—the PPE does this through main memory or its caches.

Each SPE has a dedicated local storage of 256KB and a Memory Flow Controller (MFC) [34]. A high level view of Cell BE first implementation is seen in Figure 2.1. It must be noted that the memory bandwidth with off-chip memory is 25.6 GB/s, while Local Storage (LS) and L2 Cache reach 51.2 GB/s. The PPE is the core processor of the Cell BE which consists of a Power Processing Unit connected to a 512KB L2 cache. The general purpose of this processor is to run the operating system and to coordinate the SPEs. It selects which one is supposed to run which task, as well as fairly sharing the data for each thread. One of the key architecture features that enable the Cell BE's processing power is the SPU—a data-parallel processing engine aimed at providing several levels of parallelism at all abstraction levels. Data-parallel instructions support data-level parallelism, whereas having multiple SPUs on a chip supports thread-level parallelism.

Cell processors are currently available in different forms. One of them is IBM PowerXCell QS22 computer with Cell as core processor. IBM uses this for super computers and combines multiple blades in a chassis (6.4 TFLOPs in a BladeCenter H chassis). The Cell processor is also available on a PCI-Express card, which can be used as a processor expansion board. The board has one Ethernet port which can be used to write directly into the on board memory. Another one is the Cell processor is used in the playstation3 game console. The latter has only 256 MB local memory (256 MB), 6 SPEs available to program and limited I/O. This makes it only suitable to some HPC problems.

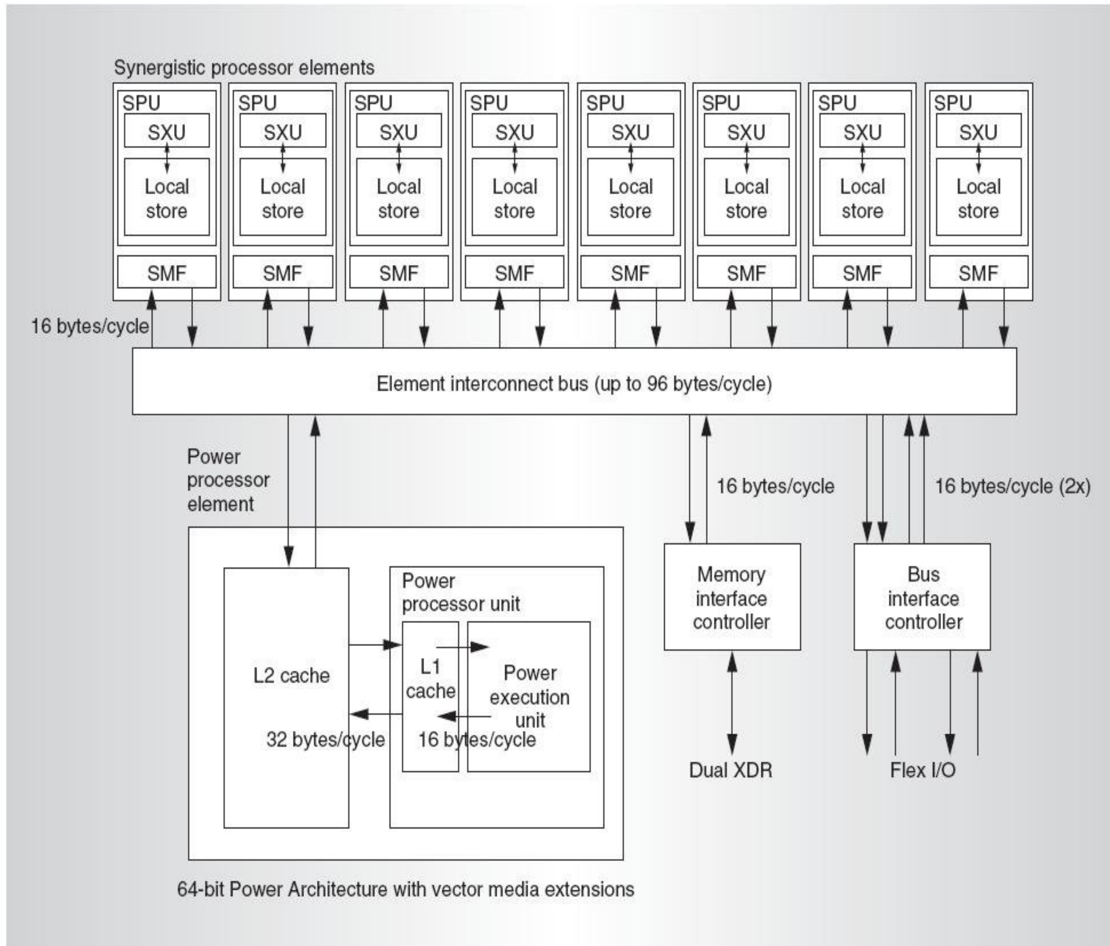


Figure 2.1: Cell System Architecture, taken from [34]

2.3. GPGPU

One of the parallel architectures that received more interest during the past years is Graphics Processing Unit (GPU). Driven by the increasing industry demand for realtime, high-definition 3D graphics, the programmable GPU has evolved into a highly parallel, multithreaded, and many-core processor. It is preliminary dedicated to efficiently support the graphics “shader” programming model, in which a program for one thread draws one vertex or shades one pixel fragment. The GPU excels at fine grained, data-parallel workloads consisting of thousands of independent threads executing vertex, geometry, and pixel-shader program threads concurrently.

GPUs have a long history of application programmability. Programming on GPUs is not new—traditionally known as GPGPU (General Purpose Graphics Processing Unit). It has been performed initially on machines like the Ikonas [England 1978], the Pixel Machine [Potmesil and Hoffert 1989] and Pixel-Planes 5 [Rhoades, et al. 1992].

Nowadays, the GPUs are not limited to its use as a graphics engine; there is a rapidly growing interest in using these units as parallel computing architecture due to the tremendous performance available in them. The importance of the emergence of GPGPU technology is to provide heterogeneous computations where applications use both the CPU and GPU. Simply, it can increase the speed of many data-processing-intensive applications just by using the GPU as

co-processors to CPU to accelerate its general purpose computations that was once only handled by the CPU alone. The advent of GPUs with their growing amount of parallel horsepower— by means of multiple pipelines, makes them a tempting resource for numerical computations.

NVIDIA [7] and ATI [8] have been the main GPU manufacturers with a long list of different models and features. Both of these companies have been producing different platforms that can use parallel computing architectures to utilize the GPU's stream processors, in tandem with the CPU, to significantly accelerate any computing process.

A typical GPGPU program uses this processor as the computation engine. The GPGPU systems have produced some impressive results, but the limitations and difficulties of doing this via graphics APIs (application programming interface) are legendary, because the developer must have in depth knowledge of graphics programming and APIs. This desire motivated graphics card manufacturers to develop a new unified graphics and computing GPU architecture and introduce new programming models dedicated to general purpose computations, such as CUDA from NVIDIA and CTM from ATI, which provide low-level or direct access to the multithreaded computational resources and associated memory bandwidth of GPUs [12], exposing them as large arrays of parallel processors.

2.3.1. ATI GPU

ATI Stream technology is based on a set of advanced hardware and software technologies that enable AMD graphics processors working in concert with the CPU, to accelerate many general purpose applications beyond just graphics. The streaming technology enables hundreds of parallel Stream cores inside AMD graphics processors.

ATI Stream processors use parallel computing architecture that employs the graphics card's stream processors to compute problems, applications or tasks that can be broken down into parallel, identical operations and run simultaneously on a single device. Stream computing takes advantage of a SIMD (Single Instruction, Multiple Data) execution methodology.

AMD Firestream is a stream processor developed by ATI Technologies, designed to utilize the stream processing for GPGPUs concept for heavy floating-point computations to support various industries, such as the High performance computing (HPC), scientific, engineering, and financial applications. The AMD Firestream, therefore, can be used as a floating-point co-processor for offloading CPU calculations [24].

The line was previously branded as both ATI *Firestream* and *AMD Stream* Processor**.** AMD refers to the Radeon RV670-based AMD Firestream 9170 because no R600-based AMD stream processors were released under the stream processing lineup [24]. The AMD Firestream 9170 processor supports double-precision floating point calculations and massively parallel processing between its 320 stream cores, driving up to 512 GFLOPs of throughput for single precision. With 2GByte of on-board memory and power consumption of 150 Watts, the Firestream will weigh in at a stout $800.

To take advantage of the massively parallel computational elements in AMD Stream Processors, by effectively unlocking the hardware, a Low Level device specific language like assembly is required. That gives the developers unfettered access to the native instruction set and memory. But at same time it requires a deep knowledge of the hardware and device specific Instruction Set Architecture (ISA). So, the difficulty of programming in device assembly language raises the need to use High Abstraction Level, device independent languages, such as OpenCL. This gives open access to the massive computational power of stream processors, and that in turn allows application developers to create their own stream applications significantly

easier. However, the available OpenCL is not yet optimal as well as still not well documented.

Beside the compiled high level languages and device assembly language, AMD introduced in a new lower level API known as Close to the Metal (CTM) which can help the application developers to have a broader choice in way to develop and deploy their applications. CTM is designed to open up the high-performance parallel processor array found in ATI graphics hardware.

2.3.2. NVIDIA GPU

NVIDIA GPUs are designed on a set of MultiProcessors (MP) in order to simultaneously process a great amount of data. Transparent scaling across this wide range of available parallelism is a key design goal of this architecture.

NVIDIA has released a wide range of architectures for the mainstream market—ranging from the high-performance enthusiast GeForce 8800 GPU, a variety of inexpensive, mainstream GeForce GPUs to professional Quadro and Tesla computing products.

Recent graphic architectures are probably today's most powerful computational hardware. For example, NVIDIA's GeForce 8800 GTX features 86.4 GB/s memory bandwidth (NVIDIA, 2009) and it costs about $600, whereas NVIDIA Tesla C1060 provides 102 GB/s memory bandwidth and it costs about $1300. Besides having high bandwidth, GPUs have been getting faster quickly. GeForce 7800 GTX (165 GFLOPS) more than triples its predecessor GeForce 6800 Ultra (53 GFLOPs) and Tesla C1060 reaches 933 GFLOPs.

As graphic processing units have become more powerful, each generation of graphics hardware has focused on more general purpose computations. NVIDIA GT200 and Tesla unified graphics and computing architectures significantly extends the GPU beyond graphics. They are massively multithreaded processor array, which became a low cost highly efficient unified platform for both graphics and general-purpose parallel computing applications.

The available graphic APIs impede development of non-graphics applications by hiding hardware resources behind its graphics-oriented interface since they are wrapped up in libraries. Moreover, using graphics APIs is not a trivial task for general programmers, so some alternatives have been proposed. Along with these releases NVIDIA distributed also a new programming model known as CUDA (Compute Unified Device Architecture) for developing innovative applications on GPUs and making GPGPU programming more straightforward (See section 2.4).

CUDA has several important aspects and different characteristics that make it eye catching for numerical applications developers (See page 2 in [2]). CUDA is compatible with NVIDIA's 8-Series, Quadro FX 5600/4600, and Tesla solutions. CUDA applications perform well on GT200 and Tesla architectures GPUs because CUDA's parallelism, synchronization, shared memories, and hierarchy of thread groups map efficiently to features of this GPU architecture. This technology allows programmers to use an extension of the C language for a minimum learning curve and it also makes use of on-chip shared memory which is faster access than DRAM, and hence making applications less dependent on DRAM memory bandwidth. The shared memory here can be used as a cache to the device memory. CUDA permits working with familiar programming concepts while developing software that can run on a GPU. Additionally it also avoids the performance overhead of graphics layer APIs by compiling the software directly to the hardware (i.e. GPU assembly language), thereby providing better performance. These features enable straightforward programming of the GPU cores.

The combination of massive speedups with an intuitive and affordable C-like

programming environment makes NVIDIA GT200 and Tesla series architectures in the spotlight among the other supercomputing platforms.

Next section describes the hardware model of NVIDIA Tesla architecture, which is targeted to be used as a numerical co-processer in this thesis, while Section 2.4 gives key details on how to program a GPU using CUDA technology.

2.4. GPU Hardware Model

NVIDIA Tesla computing solutions enable the necessary transition to fast and energy-efficient parallel computing power. With many-core architecture and a standard C compiler that simplifies application development and unlocks the stream processing power of GPUs, Tesla architecture scales to solve the most complex and toughest computing challenges including scientific applications more quickly and effectively. Capable of a Tera GFlops of processing performance puts it near the performance of small supercomputers.

The Tesla architecture is built around a scalable array of multithreaded streaming MultiProcessors (MPs). It is viewed as a compute device capable of executing a huge number of threads in parallel. It can operate as co-processor to the main CPU—or *Host*; so, data-independent, compute-intensive portions of applications running on the host are off-loaded onto the GPU—or *device*.

Throughout this thesis, NVIDIA Tesla C1060 card was used. This architecture consists of 240 (Thread Processor) cores that can collectively run thousands of computing threads. It is implemented as 30 streaming MPs. Each of MPs involves eight thread processor cores. A multiprocessor comprises eight single-precision floating point ALUs (one per core) and only one double-precision ALU (shared by the eight cores). The peak computation rate accessible from CUDA is around 933 GFlops for single floating point precision, while the performance is around only 78 GFlops for the double floating point precision. Figure 2.2 shows a general architecture of Tesla 10-series architecture with 30 MPs. The processors are clocked at 1.296 GHz. It can reach approximately a Tera FLOPs peak performance with 160 watt as typical power consumption and a maximum 225 Watt of consumption.

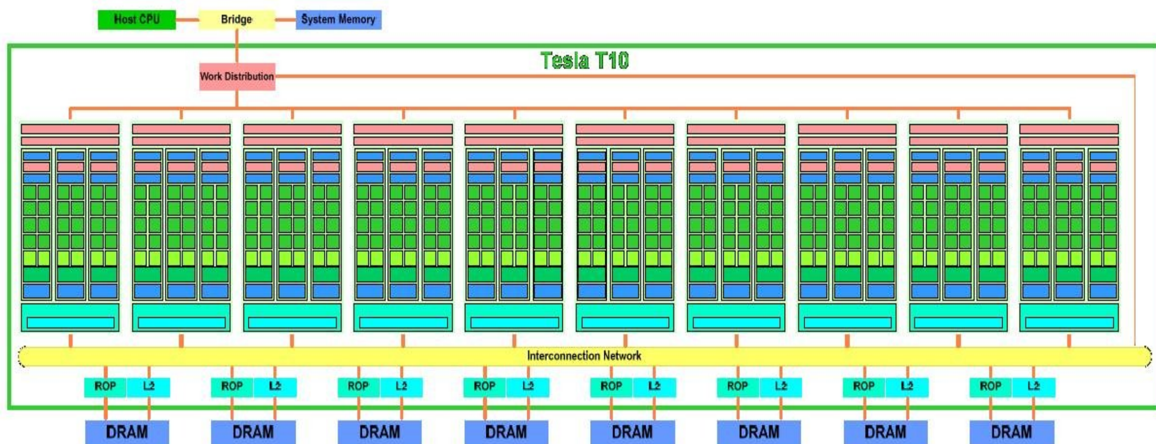


Figure 2.2: NVIDIA Tesla 10-Series Architecture

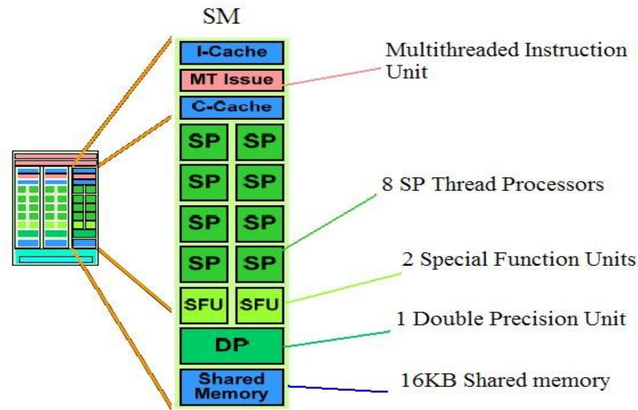


Figure 2.3: Diagram of one Tesla 10-Series Multiprocessor, SM: Stream Multiprocessor; SP: Scalar Processor

This device includes 4 GB of onboard DRAM memory that can be accessed by all the threads. The host has access only to the external device memory. Device to host memory bandwidth is much lower than device to device bandwidth, where 4GB/sec peak (PCI-e x16 Gen 1) versus 102 GB/sec peak (Tesla C1060). Thus, the off-chip device memory has a lower latency comparing to the host memory due to a larger bandwidth, and the very high concurrent threads help to hide any additional latency.

All cores in a multiprocessor have on-chip shared resources, including 16384 local 32-bit registers and one on-chip fast memory of size 16Kbyte, which enable threads cooperation.

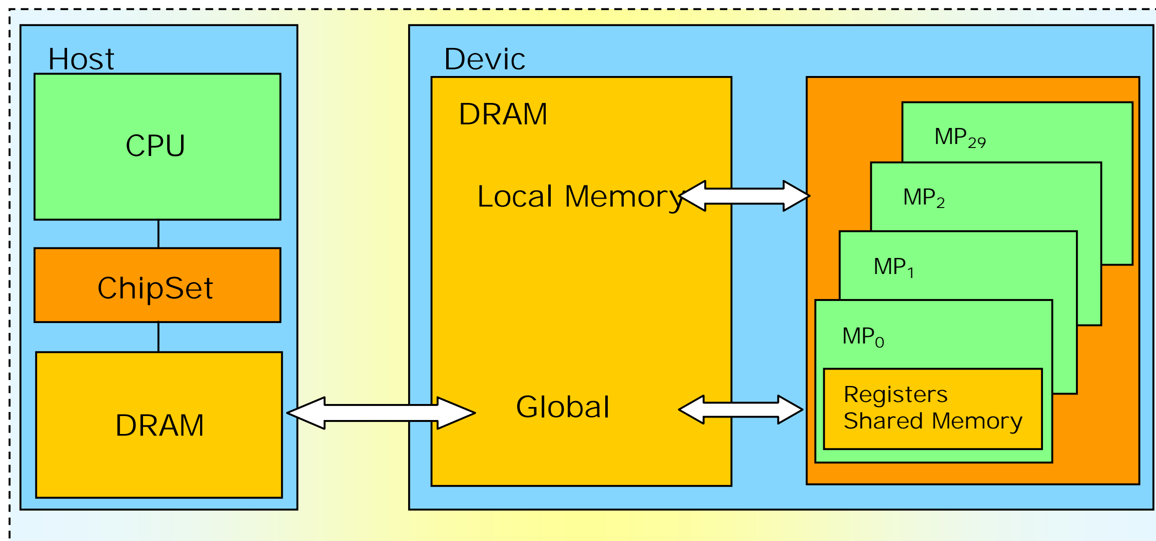


Figure 2.4: Tesla Memory Model interconnected to a CPU

Thread's variables typically reside in live registers. The on-chip shared memory has very low access latency and high bandwidth similar to an L1 cache; it holds CUDA variables for the active thread blocks. The shared memory allows the parallel threads run on the cores in a MP to share data without sending it over the system memory bus. Thus, that provides much speedup if there is no bank conflicts (See page 56 in [2]), however this memory is too small.

The MP provides load/store instructions to access CUDA variables in GPU external DRAM. It coalesces individual accesses of parallel threads in the same warp into fewer memory-block accesses when the addresses fall in the same block and meet alignment criteria. As stated in [28], *"A multiprocessor takes 4 clock cycles to issue one memory instruction for a warp. When accessing global memory, there are, in addition, 400 to 600 clock cycles of memory latency*". Sometimes it's better to re-compute than to cache because GPU spends its transistors on ALUs, not on memory.

Because global memory latency can be hundreds of processor clocks, CUDA programs may copy the data to shared memory when it must be accessed multiple times by a thread block. Tesla load/store memory instructions use integer byte addressing to facilitate conventional compiler code optimizations. The large thread count in each MP, together with support for many outstanding load requests, helps to cover load-to-use latency to the external DRAM. The latest Tesla-architecture GPUs also provide atomic read-modify-write memory instructions, facilitating parallel reductions and parallel-data structure management.

Different Tesla series are available from NVIDIA, like Tesla 10-series, which are either solo GPU cards such as Tesla C1060, or combinations of multiple GPUs in a single card such as Tesla S1070. Tesla S1070 is a 1U plate server that includes four GT200-series GPUs (960 Cores) running at a higher speed than that in the C1060 (up to 1.5 GHz core clock vs. 1.3 GHz), so the S1070's peak performance is over 4.6 times higher than a single C1060 card. It has a 16GB DDR3 with 408 GB/s bandwidth and it can reach 4.14 TFLOPS peak performance in a 1U chassis with maximum 800 Watt of power consumption.

By this time, NVIDIA has announced a new CUDA computing architecture, called "Fermi" [37]. This architecture consists of 16 Stream Multiprocessors (SM). Each SM includes 32 cores, 16 load/store units, four special-function units, a 32K-word register file, 64K of local SRAM split between cache and shared memory, and thread control logic, see Figure 2.5. Each core has both floating-point and integer execution units. This architecture has up to 6GB of GDDR5 memory. In addition, it maximizes bandwidth between the host system and the Tesla processors. As well as it maximizes the throughput by faster context switching, concurrent kernel execution, and improved thread block scheduling. In Fermi, two warps from different thread blocks (even different kernels) can be issued and executed concurrently, increasing hardware utilization. Fermi supports simultaneous execution of multiple kernels from the same application, each kernel being distributed to one or more SMs on the device. Furthermore, switching from one application to another is about 20 times faster on Fermi (25 microseconds only) than on previous-generation GPUs. Moreover, the double precision performance is 8x faster compared to Tesla 10-series GPUs. Table 2.1 shows a comparison of Fermi and GT200 computing processors. Fermi supports for multiple languages like, C, Fortran, Java, MatLab, and Python.

NVIDIA claims that Tesla 20-series, which contains 4 Fermi GPUs delivers equivalent cluster performance at 1/20th the power and 1/10th the cost of CPU-only systems based on the latest quad core CPUs. Since it is not yet released, the price is still unknown.

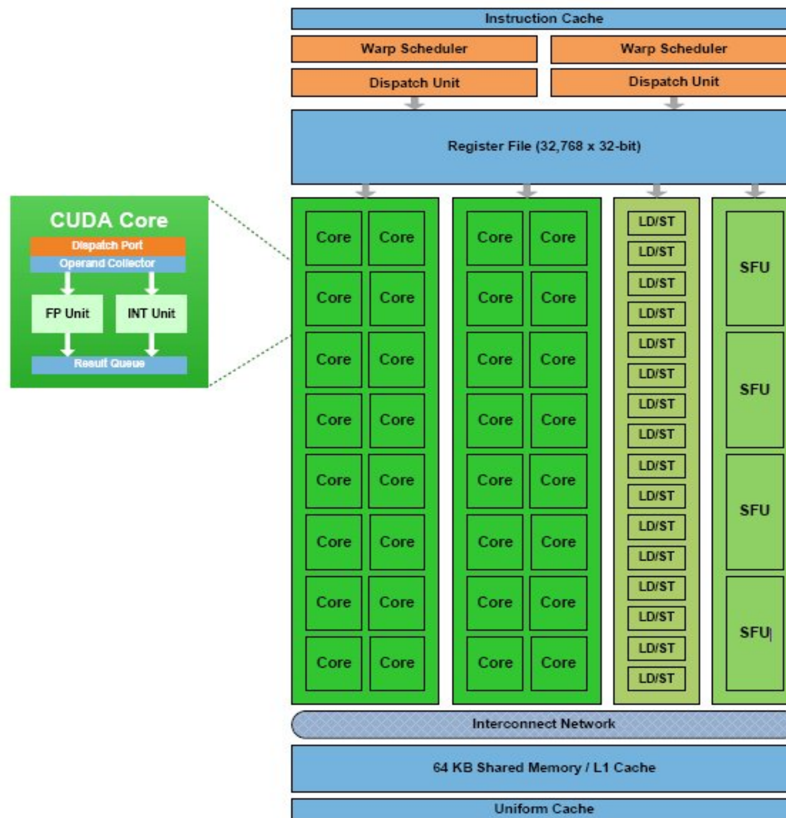


Figure 2.5: Diagram of one Stream Multiprocessor of Fermi Architecture, taken from [37]

Metrics	GT200	Fermi
# Transistors	1.4 billion	3.0 billion
# of cores	240	512
Double Precision Floating Point Capability	30 FMA ops / clock	256 FMA ops / clock
Single Precision Floating Point Capability	240 MAD ops / clock	512 FMA ops / clock
Double Precision FP Peak Performance	~87GFlops	~624GFlops
# Warps schedulers (SM)	1	2
# Special Function Units (SFUs) / SM	2	4
Shared Memory/ SM	16KB	Configurable 48KB or 16KB
L1 Cache / SM	None	Configurable 16KB or 48KB
L2 Cache / SM	None	768KB
Dedicated Memory	4GB DDR3	up to 6GB of GDDR5
ECC Memory Support	No	Yes
Concurrent Kernels	No	Up to 16
Load/Store Address Width	32-bit	64-bit
Power Consumption	175Watt (typical) 200Watt (max)	225Watt (typical) 275Watt (max)

Table 2.1: Comparison between GT200 and Fermi architectures

2.5. GPU Programming Model

CUDA technology is a new hardware and software solution for general purpose parallel computing from NVIDIA. CUDA is a parallel programming model and software environment that leverages the parallel computational horsepower of GPU for non- graphics computing in a fraction of the time required on a CPU.

The programming paradigm provided by CUDA has allowed developers to utilize the power of these scalable parallel processors with relative ease, enabling them to achieve speedups of several times on a variety of sophisticated applications. Since NVIDIA released CUDA in 2007, a lot of scalable parallel programs were rapidly developed for a wide range of applications, including matrix solvers, sorting, searching, computational chemistry, and physics models. These applications scale transparently to hundreds of processor cores and thousands of concurrent threads.

CUDA provides a few easily understood abstractions that allow the programmer to focus on algorithmic efficiency and develop scalable parallel applications by expressing the parallelism explicitly. It provides three key abstractions—a hierarchy of thread groups, shared memories, and synchronization barrier —that provide a clear parallel structure to conventional C code for one thread of the hierarchy. The abstractions guide the programmer to partition the problem into coarse sub-problems that can be solved independently in parallel, and then into finer pieces that can be solved cooperatively in parallel. The programming model scales transparently to large numbers of processor cores: a compiled CUDA program executes on any number of processors, and only the runtime system needs to know the physical processor count [2][3].

As we explained before, CUDA can support also heterogeneous computation. Serial portions of applications are run on the CPU, and parallel portions are offloaded to the GPU. The CPU and GPU are treated as separate devices that have their own memory spaces. This configuration also allows simultaneous and overlapped computation on both the CPU and GPU without contention for memory resources.

The fundamental part of the code for CUDA is the *kernel* program. The kernel is a program that operates on the entire stream of data. The context of a CUDA kernel is simply a C code for one thread of the hierarchy, but it executes in parallel across a set of parallel threads. These threads are organized into a hierarchy of a grid of thread blocks. A *grid* is a set of thread blocks that can be processed on the device independently by scheduling blocks for execution on the MP— and thus they may execute in parallel. A *thread block* is a set of concurrent threads that can cooperate among themselves through synchronization barrier (where the threads that are setup by a kernel call must wait to synchronize) and shared access to a memory space private to the block. Thread blocks are executed as smaller groups of threads known as "*warps*"— this term originates from weaving. So, individual threads composing a warp, start together at the same program address but they are free to branch and execute independently (See [2] page 14). The warp size is 32 threads on Tesla architecture.

Each thread is given a unique thread ID— *threadIdx* within its thread block, numbered 0, 1, 2, ..., blockDim–1, and each thread block is given a unique block ID— *blockIdx* within its grid. CUDA supports thread blocks containing up to 512 threads. The thread blocks may have one, two, or three dimensions, accessed via .x, .y, and .z index fields (See [2], page 7, 8). Parallelism is determined explicitly by specifying the dimensions of a grid and its thread blocks when launching a kernel. Each kernel launch creates a grid of blocks that assigns one thread to each element of the vectors and distributes the threads over the blocks. Each thread computes an element index from its *thread and block IDs,* and then performs the desired calculation on

the corresponding vector elements.

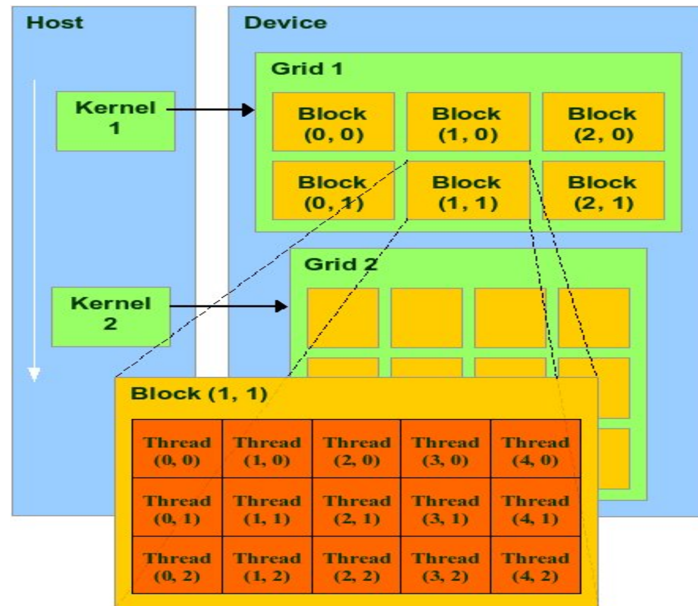


Figure 2.6: Representation of CUDA Programming Model, taken from [2]

According to conventional wisdom, parallel programming is difficult. However, this scalable parallel programming model by using a standard language that is essentially 'C' shows that many sophisticated programs can be readily expressed with a few easily understood abstractions. CUDA is a minimal extension of the C and C++ programming languages. The GPU code is implemented as a collection of functions, which may be simple functions or full programs, in a C language, but with some annotations for distinguishing them from the host code, plus annotations for distinguishing different types of data memory that exist on the GPU. Such functions may have parameters, and they can be 'called' using a syntax that is very similar to regular C function calling, but slightly extended for being able to specify the matrix of GPU threads that must execute the 'invoked' function [2]. Figure 2.7 shows an example of array incrementing in C and in CUDA.

C program (CPU)

```
void inc_cpu(int *a, int N)
{
    int idx;
    for (idx = 0; idx<N; idx++)
    a[idx] = a[idx] + 1;
}
int main()
{
    ...
    inc_cpu(a, N);
}
```

CUDA program (CPU-GPU)

```
__global__ void inc_gpu(int *a, int N)
{
    int idx = blockIdx.x * blockDim.x + threadIdx.x;
    if (idx < N)
    a[idx] = a[idx] + 1;
}
int main()
{
    …
    dim3 dimBlock (blocksize);
    dim3 dimGrid( ceil( N / (float) blocksize) );
    inc_gpu<<<dimGrid, dimBlock>>>(a, N);
}
```

Figure 2.7: Increment Array Example in C and CUDA

As we see, CUDA code is generally straightforward to write and is typically simpler than writing parallel code for vector operations. But, when developing CUDA programs, it is important to understand the ways in which the CUDA model is restricted, largely for reasons of efficiency.

Kernel invocation in CUDA is asynchronous, so the driver will return control to the application as soon as it has launched the kernel. But, for instance, CUDA functions that perform memory copies are synchronous, and implicitly wait for all kernels to complete.

To manage huge number of threads running in parallel, CUDA broadly follows a data-parallel model of computation—so-called SIMT [31]. Typically, at any given clock cycle, each thread executes the same instruction, but operates on different elements of the data set in parallel. The MP maps each thread to one thread core, so each scalar thread processed independently with its own instruction address and register state. SIMT enables thread-level parallelism for independent, scalar threads, as well as data-parallelism for coordinated threads.

Tesla architecture GPU performs the parallel execution and thread management automatically and directly in hardware. All thread creation, scheduling, and termination are handled by the underlying system through the compute work distribution (CWD) unit—the threads scheduler [6]. When a CUDA program on the host invokes a kernel grid, the CWD unit enumerates the blocks of the grid and begins propagating them to MPs with available execution capacity. Each MP processes batches of blocks one batch after the other. The threads of a thread block execute concurrently on only one MP. As thread blocks terminate, the CWD unit launches new blocks on the vacated multiprocessors.

During the threads execution, individual threads have access to data that settle in different memory spaces as illustrated by Figure 2.8. Each thread has a private local memory and register file. Each thread block has a shared memory visible to all threads of the block that have same lifetime as the block. In addition, all threads of different blocks may access same global memory. There are two additional read-only memory spaces accessible by all threads: the constant and texture memory spaces, see Figure 2.8.

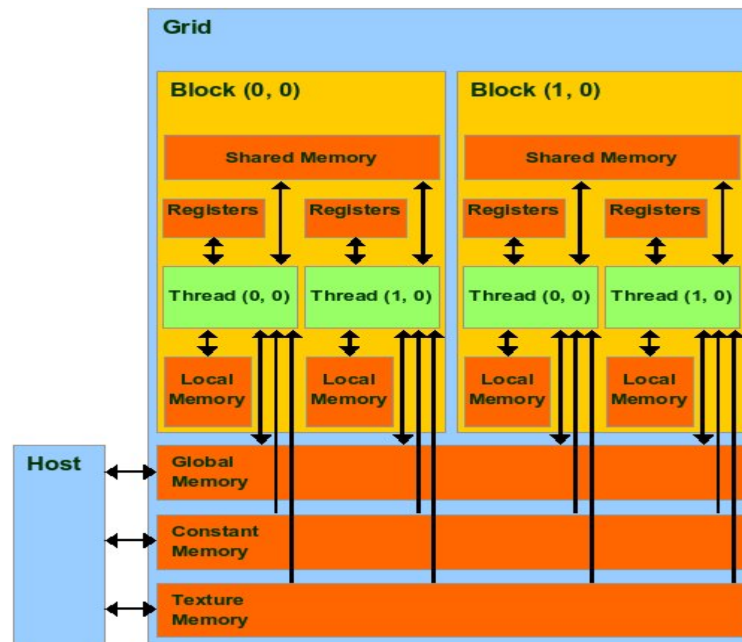


Figure 2.8: Representation of CUDA Threads Blocks mapped on CUDA Memory Model, taken from [2]

The proper choice of the memory to be used in each kernel depends on many factors such as the speed, the amount needed, and the operations to be performed on the stored data. An important restriction is the limited size of shared memory, which is the only available read-write cache. Unlike the CPU programming model, the data needs to be explicitly copied from the global memory to the shared memory and backwards.

The real advantage of the shared memory appears as a communication media to allow threads within a thread block to cooperate. This allows caching of frequently used data and can provide large speedups over using external memory to access data.

The threads scheduler creates, schedules, and manages scheduling threads in warps across thread processing elements. Each MP executes a pool of maximum 24 active warps of 32 threads per warp, a total of 768 active threads, (See Section A.1, appendix A in [2]).

Every instruction issue time, the threads scheduler selects a warp that is ready to execute and issues the instruction to the active threads of the warp. A warp executes one common instruction at a time, so full efficiency is realized when all 32 threads of a warp agree on their execution path. If threads of a warp diverge via a data-dependent conditional branch, the warp serially executes each branch path taken; disabling threads that are not on that path, and when all paths complete, the threads converge back to the same execution path. Branch divergence occurs only within a warp; different warps execute independently regardless of whether they are executing common or disjointed code paths [6]. For the best utilization: if a warp is waiting for memory access, the scheduler can perform a zero-cost, immediate context switch to another warp.

To manage this processing element virtualization and provide scalability, CUDA requires that thread blocks execute independently. It must be possible to execute blocks in any order, in parallel or in series.

The threads of a block execute concurrently and may synchronize at a barrier by calling the __syncthreads() intrinsic. This guarantees that no thread participating in the barrier can proceed until all participating threads have reached the barrier. After passing the barrier, these threads are also guaranteed to see all writes to memory performed by participating threads before the barrier. Thus, threads within a same block may coordinate their communication with each other by writing and reading per-block shared memory at a synchronization barrier. (See [2], page 26).

Thread synchronization combined with using on-chip shared memory allow cooperative parallel processing of active threads, thereby greatly reducing the expensive off-chip bandwidth requirements for many parallelizable algorithms, such as linear algebra and Fast Fourier Transform. As a result, the threads are resided on the same physical processor or MP. The number of thread blocks can, however, greatly exceed the number of processors. This allows intuitive problem decompositions, as the number of blocks can be dictated by the size of the data being processed rather than by the number of processors in the system. In addition, this also allows the same CUDA program to scale to various numbers of processor cores, thereby keeping current and future GPU fully busy.

Each MP has to have at least one block to execute, but having only one block per a MP may force the MP to idle during thread synchronization or memory reads. Therefore, the number of blocks must be at least twice the number of multiprocessors or more because multiple blocks can run simultaneously on a multiprocessor (See [2], page 62). Blocks that are not waiting at a synchronization point keep the hardware busy, however, that is subject to resource availability—registers, shared memory.

The current GPU architectures limit the maximum number of threads per thread block

and thread blocks per computation grid; these limits are 512 and 65535 respectively. These limits restrict the input sizes that can be handled on GPU. This can be overcome by using threads virtualization as in [33].

Occupancy is the number of the active warps running concurrently on a multiprocessor divided by maximum number of warps that can run concurrently which is limited by resource usage: registers and shared memory. So, it must be noted that full occupancy does not mean 100% optimization. Special attention must be paid to that using careful execution configuration (i.e. blocks and grids dimensions) can help the hardware perform a very big amount of work in parallel which leads to confer remarkable benefits, whereas careless execution configuration may force parallel code into serial execution.

CUDA's software stack is composed of a hardware driver, an application programming interface (API) and its runtime, and two higher-level mathematical libraries, CUFFT [4] (CUDA Fast Fourier Transform library) and CUBLAS [5]– an implementation of Basic Linear Algebra Subprograms (BLAS) on top of CUDA (See [2], page 3). CUFFT, the "CUDA" FFT library, provides a simple interface for computing parallel FFTs on an NVIDIA GPU. This allows leveraging the floating-point power and parallelism of the GPU without having to develop a custom, GPU-based FFT implementation. The main supported features of this library interface are batched execution for doing multiple 1D transforms in parallel, and it does 1D transform of size up to $8x10^3$ elements and 2D transform of size up to $1x10^{14}$. Besides CUDA FFT library there is also a CUDA Basic Linear Algebra Subprograms (CUBLAS) which is a standard library to perform basic linear algebra operations such as vector and matrix multiplication at level 1, 2 and 3 (vector-vector, matrix-vector and matrix-matrix operations respectively).

After describing the GPU programming model, one can notice that some applications may fit the model perfectly and may as well benefit from very high performance. On the other hand, there are applications which may not fit the aforementioned programming model and may not present speedup running on the GPU because of some factors that can influence the performance, for instance, using careless execution configuration which may not exploit all the available resources on the GPU, or insufficient memory management. If the algorithm is well-suited to the architecture, dramatic improvements can be achieved in several key metrics such as GFlops per Watt and cost.

So, using Tesla architecture coupled with CUDA technology we shall investigate throughout this thesis whether our application fit this model or not.

2.6. Summary

Nowadays, many scientific and engineering applications tend towards using more specialized high-performance processor technologies over conventional CPU processors. Some of the graphics computing solutions have been expanding their intended use in current and future graphic systems, as well as in high-performance computing systems (Such as, Cell/BE and the NVIDIA and ATI GPUs).

Semiconductor capability and advances in fabrication process have increased for both CPUs and HPC platforms at the same rate, but the main growth disparity is due to architectural differences: CPUs have large caches and they are optimized for high performance on sequential code, focusing branch prediction and out-of-order execution. On the other hand, GPUs that focus on highly parallel general purpose computations achieve higher arithmetic intensity by using roughly the same number of transistors. Thus, it increases the system performance without compromising the overall system power consumption.

Making the best choice of platforms is greatly depending on applications. If an application contains a massive amount of independent-data and still needs to gain performance while maintaining the power consumption, then GPU is a better choice. Since the GPU is a massively multithreaded parallel machine with high-end shared memory, it is a great choice for fine-grained parallelism. However, there are some difficulties and limitations found in GPU programming such as:

- New programming paradigm– programs and algorithms need to go through a completely different design process in order to exploit full parallel advantages of high processing power available in this architecture.
- Double precision floating-point computations– most of the available architectures currently only make available single precision floats, although double-precision can be emulated with some performance penalties; However, NVIDIA recently supported the double precision floats but with less parallelism power.
- GPU Programming – in order to achieve vendor independence, graphic APIs such as OpenGL or DirectX are usually recommended. However, CUDA and CTM make it possible and simpler for programmers to program the GPU without need to use graphic APIs, and with a hope, that in next few years NVIDIA will improve the performance of CUDA even more.

CUDA has several advantages over traditional general purpose computation on GPUs using other graphics APIs, like:

- CUDA has much more freedom to program GPU and to access memory. Code can read to arbitrary addresses in memory. It supports the architectures on higher abstract level.
- CUDA exposes a fast shared memory region that can be shared amongst threads per MP. This can be used as a user-managed cache, enabling higher bandwidth than is possible using external DRAM.
- It is high level-basically an extension to C language. So the learning rate is much higher compared to the traditional GPGPU. CUDA kernels can also be launched from other languages (i.e. Java, Fortran, Python, etc) [38], while the kernels still need to be programmed in "C".
- CUDA has much more comfortable APIs. People that are working in physics and mathematics, who have no experience, and do not want to know about assembler, shaders programming and etc, CUDA is a better choice for them.

Table 2.1 shows a summary that compares the three High performance processors. see [19][20][21][22]. From this comparison, we see that there is no winner from technical perspectives and they all have different qualities. If we compare them in term of the performance per watt, GPUs perform more efficient than Cell processors. In addition the GPUs are much cheaper; however, Cell/BE used in Sony's PlayStation3 is reasonably cheap. Furthermore, NVIDIA GPU has almost a double size of memory compared to AMD and Cell. The Stream Processing Units (SPUs) in ATI hardware implementations are architecturally different from NVIDIA's implementation of Stream Processors in Tesla products. The Stream Processor in NVIDIA's implementation has a higher clock frequency than the other parts of the core, while SPUs in ATI's implementation have the same clock frequency as the core. And as we see in the Table 2.1, a single AMD stream core is slower than its counterpart in NVIDIA GPU. Moreover, AMD supports less memory bandwidth than NVIDIA Tesla GPUs.

In this thesis, we intended to use GPU processors, namely Tesla architecture, for

technical as well as for non technical reasons.

HPC Solutions	Board	Clock Rate And Memory	Peak Performance	Power in Watt (TDP)	Price
Intel Quad Core	Xeon E5472	3.0 GHz	96 GFlops (SP) 48 GFlops (DP)	Max 95	~$950
	Core i7 965XE (Nehalem)	3.2 GHz	102 GFlops (SP) 51 GFlops (DP)	Max 130	~$1500
IBM Cell	Sony's PlayStation3	3.2 GHz 25.6 GB/s	153.6 GFlops (SP) 9.6 GFlops (DP)	Max 260	~ $350
	PowerXCell 8i	3.2 GHz Up to 32 GB of slotted Memory	205 GFlops (SP) 102.4 GFlops (DP)	Max 260	~ $1700
ATI/AMD (RV770)	FireStream 9250	625 MHz 2GB Memory 63.5 GB/s	1 TFlops (SP) 200 GFlops (DP)	Max 150	~ $900
	FireStream 9270	750 MHz 2GB Memory 108.8 GB/s	1.2 TFlops (SP) 240 GFlops (DP)	Max 160	~$1400
NVIDIA GPU	Tesla C1060 (1 GPU)	1.3 GHz 4GB Memory 102 GB/s	933 GFlops (SP) 87 GFlops (DP)	Typical 160 Max 225	~ $1240
	Tesla S1070 (4 GPUs)	1.5 GHz 16GB Memory 102 GB/s	4.14 TFlops (SP) 346 GFlops (DP)	Max 800	~$7000

Table 2.2: Comparison between different High Performance Computing Solutions

NVIDIA claims to have sold more than 100 million CUDA-enabled GPUs, and they are also being supported by thousands of software developers who NVIDIA says are already using the free CUDA software development tools to solve problems in a variety of professional and home applications.

In addition, the difficulty of programming Cell, which requires assembly level intrinsics for the best performance, makes this model not useful as an initial step in algorithm design and evaluation. Similarly, AMD processors are not easy to program. In contrast, the relatively ease of programming the NVIDIA GPU, using CUDA technology in particular, comparing to the other HPC platforms— was another motivation to choose NVIDIA GPUs, namely Tesla architecture.

Beside the comparison in regard to the efficiency (performance/watt), price, reliability, ease of use, and programmability. Given this background, and before going on with further details on how a parallel solution is implemented on GPU architecture, the next chapter gives a brief description over the application and the physics behind it.

3 The Application Description

A concise description of the proposed application and the motivation behind the study will be given in this chapter.

Lithography is one of the most important steps in the manufacturing process of integrated circuits. The challenge in wafer manufacturing is combining high accuracy and reliability with maximum throughput at minimum cost. Thus, it is driven by optimizing the yield in a given time. This means finding the cause of problems quickly to minimize system downtime is an opportunity to huge cost savings and greater productivity. For that purpose, ASML has developed a Diagnostic-tool that is used to inspect the quality of the imaging and overlay on the wafer at high speed and with sub nanometer precision. In this way, this tool may be able to boost the productivity through finding problems faster and to reduce downtime.

As mentioned before, this diagnostic tool provides a way to accurately measure the overlay on a wafer, and secondly determines the quality of the imaging, which is done “in situ”, i.e. without removing the wafer from the production track. In the scope of this thesis, only the imaging qualification will be considered. For this purpose, ASML has developed a reconstruction algorithm that estimates shape parameters from the measured spectrum. (See Section 3.3)

In the next section, we will briefly dwell on the lithography process of chip-making and the rigorous diffraction model that is used to determine the shape of the lines on a wafer. The rest of this chapter explains the technique that was adapted to compute the reflected field coming from a grating structure and the method that was employed to do the reconstruction.

3.1. Lithography

Photolithography is a process to create a pattern onto a wafer by imaging a mask, which is a sheet of glass, partially covered by an opaque material. The objective of the lithography process is to produce an IC image (See Figure 3.1). A chip consists of 20 to 30 connected layers. Through Wafer Alignment and subsequent exposure, lithography determines layer-to-layer positioning and critical dimensions (CD) with sub nanometer accuracy. Thus, to achieve a high resolution it must meet some criteria: the overlay must be better than ~15% of the line width. So, for instance, a line width of 32 nm requires approximately 5 nm of overlay. For wafer metrology, this poses very stringent specifications on measurement accuracy. Optical Scatterometry is the technique that is used for metrology of LineWidth and Overlay. It simulates light scattering via rigorous diffraction grating models. For CD, the line profile is reconstructed from light that is scattered by a test pattern.

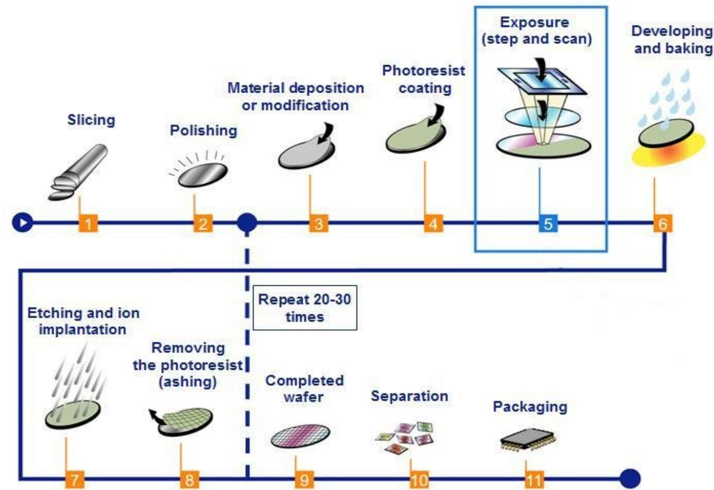


Figure 3.1: Lithography Process

3.2. Diffraction Grating Model

Periodic structures play an important role in optical lithography. To measure the shape of the structures with sub nanometer precision, a special metrology target is put on the wafer. Such grating structure can be a two-dimensional pattern, which is a repetitive array of either apertures or obstacles that has the effect of producing periodic alternations in the phase and amplitude of a scattered wave. The diffraction of the incident field in multiple diffraction orders provides a way to accurately determine overlay on the one hand and on the other hand, it provides a scatter spectrum from which the quality of the photolithographic process can be deduced. A grating shape is described by a number of physical parameters which are taken into account to determine the resolution of the lithography process. In its simplest form in Figure 3.2, these features are:

- Pitch, is the period.
- BARC thickness ($\boldsymbol{h}_{BARC}$).
- Linewidth (*midCD*) = (*top* + *bottom*)/2.
- The height of a trapezoidal grating structure (h_{res}).
- The Side-Wall-Angle (*SWA*) = arctan($2* h_{res} /\Delta CD$), where ΔCD is bottom subtracted by top.

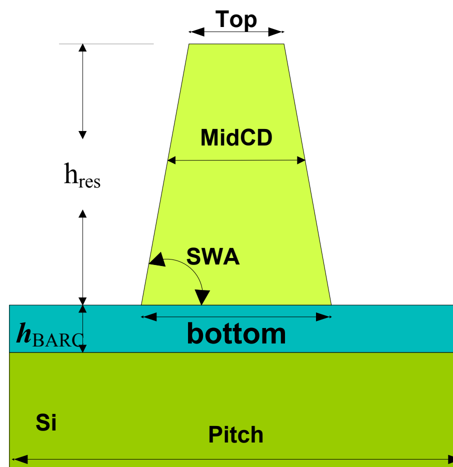


Figure 3.2: Diagram of Grating Shape Parameters

These parameters can be reconstructed by minimizing the difference between the measured scattered spectrum and the computed spectrum. This process is the so-called Reconstruction Loop, which will be explained in more details in the next section.

3.3. The Reconstruction Loop

The process of finding the actual shape parameters of a grating structure from a measured scattered field is called: the Reconstruction Loop. The scattered field for a given grating structure is computed iteratively from Forward Diffraction Model (See Figure 3.3). The process starts at an initial guess of the shape parameters under consideration. Putting this initial guess for the profiler in the forward model gives a calculated scattered field, which is compared to a measured one. The difference between both spectra is minimized by computing new values for the shape parameters from the spectral difference. The whole procedure is repeated until the computed scattered spectrum is sufficiently close to the measured spectrum.

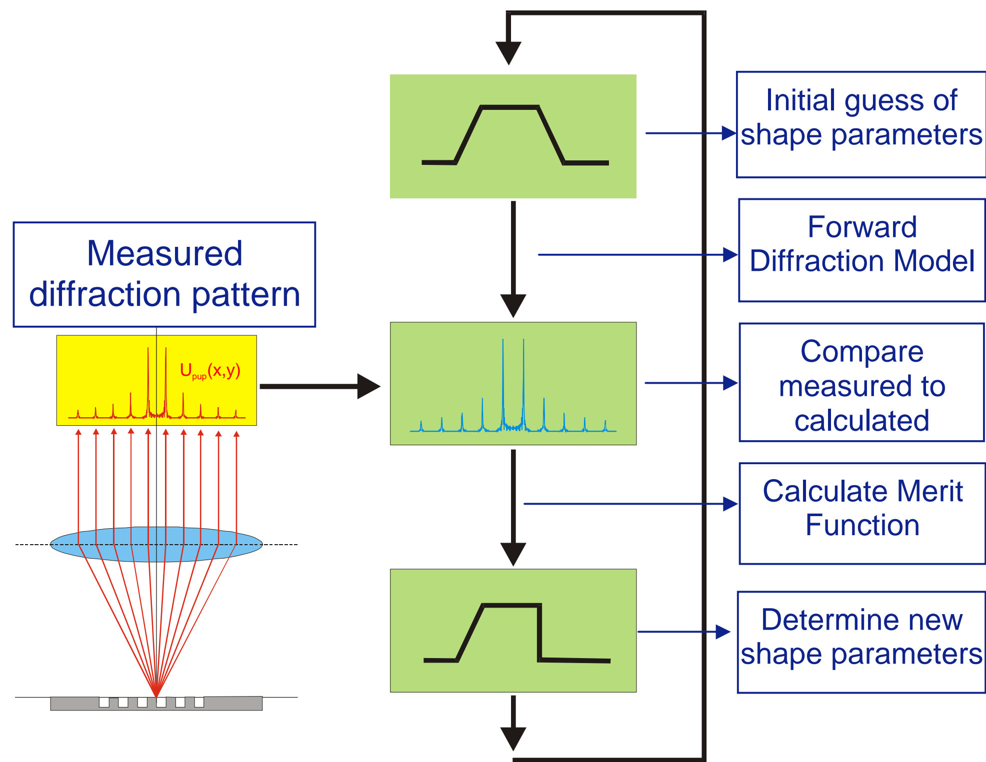


Figure 3.3: Reconstruction Loop Diagram

3.4. The Electromagnetic Diffraction Model Algorithm

The Electromagnetic Diffraction Model (EDM) is ASML proprietary information and hence cannot be explained in detail in this thesis. For the remainder of this thesis it suffices to know that the electromagnetic wave equation is discretized through a Fourier expansion of the fields in the periodic x- and y-direction and through slicing the object in layers along the z-direction. The electromagnetic response of the scattering object is described by a shape function. This discretization results in a set of linear equations which is numerically solved. In the solution process of the linear system, the matrix is multiplied by a vector (the so-called "matrix-vector product") which is computed using a 1D and 2D convolutions. EDM computes scattered field for one angle of incidence.

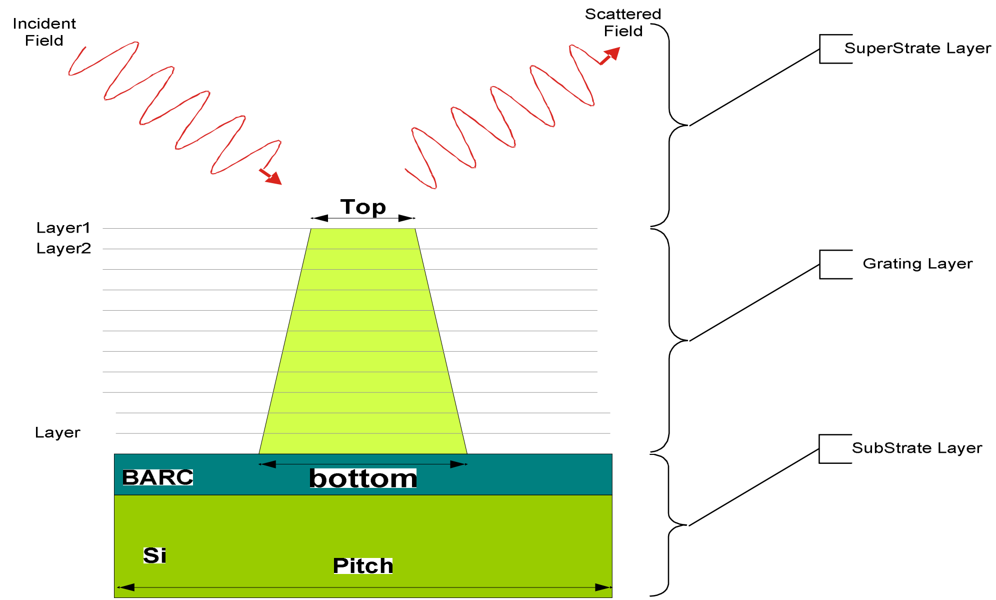


Figure 3.4: Diagram of Grating Layers

3.4.1. 2 Dimension Convolution Algorithm

Mathematically, a convolution measures the amount of overlap between two functions [17]. It can be thought of as a blending operation that integrates the point-wise multiplication of one dataset with another. This convolution, in frequency space, amounts to a point-wise multiplication in a real space.

The 2D convolution is computed via direct/inverse 2D FFT and scalar product of the electromagnetic vector field by the shape function for each sample point in *z* direction and for each field component. Thereby, computing the 2D Fourier-based convolution is done in 3 phases:

1. Apply direct FFT to 2D arrays on a layered stack of slices of the electromagnetic vector field ($\mathbf{E}^{tot}$).
2. Perform element-wise multiplication of the preceding result with the shape function (S).
3. Apply inverse FFT to the result of the multiplication and save the results back to the derived field ($\mathbf{E}^{d}$).

Figure 3.9 depicts a detailed level of this algorithm.

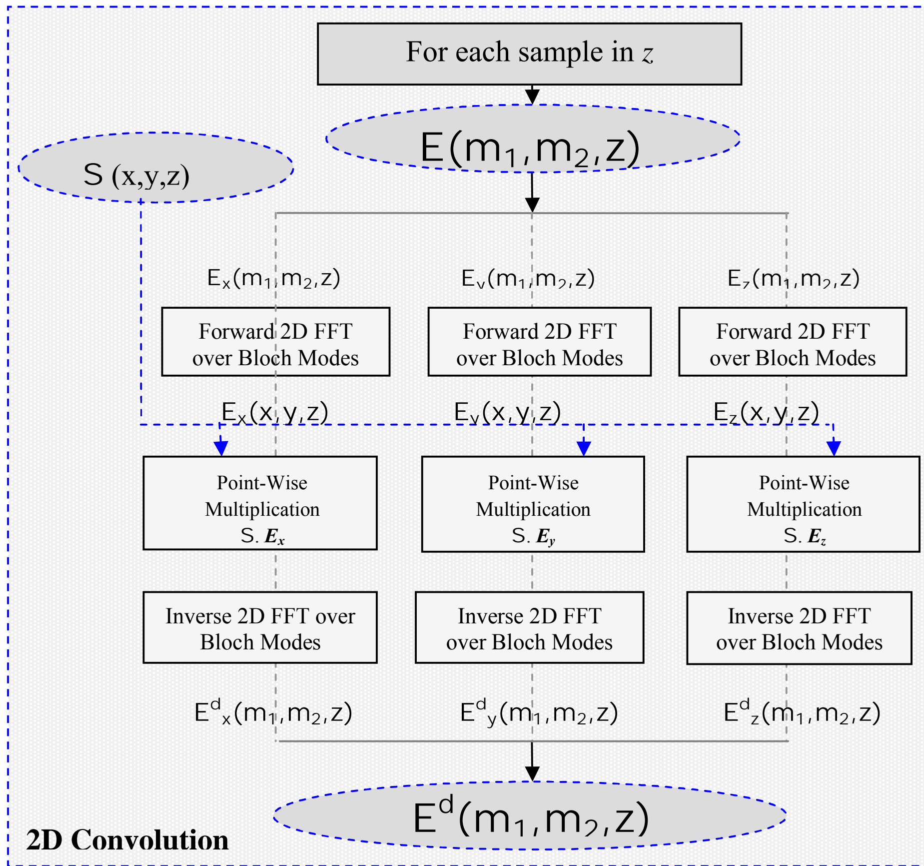


Figure 3.5: Flowchart of 2D Convolution Algorithm

Note that, the data set of the electromagnetic vector field Etot and Ed are structured in a 3D stack of slices of size (M_1xM_2) as illustrated in Figure 3.6, where M_1 is the number of modes in x direction and M_2 is the number of modes in y direction.

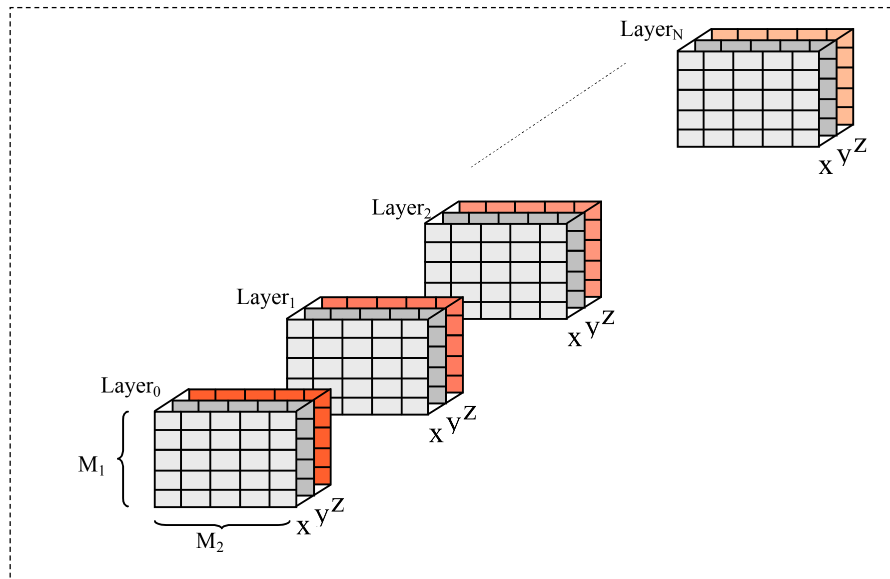


Figure 3.6: Data Structure Representation of the Electromagnetic Vector Field (E^{tot}) and Derived Field (E^d)

Given this background on the application, and before going on with the parallel implementation, the next chapter gives further details about serial implementation of EDM algorithm and investigates the bottlenecks of the application.

4 CPU-Based Serial Implementations of EDM Algorithm

4.1. Serial Fortran-Based Implementation of EDM

The Fortran-Based serial implementation of EDM algorithm was initially provided by ASML. It is mainly built upon the FFT, BLAS and LAPACK libraries. This implementation is basically divided into three main sections, as demonstrated in Figure 4.1, as follows:

1. **Preprocessing:** this section is responsible for creating workspace that is required for the 1D and 2D FFT configurations and initializations, and allocating all data arrays. As well as retrieving/collecting all parameters with respect to the grating from input files (such as the number of layers in the grating, thickness of the grating slab, number of material parameters, material parameters of the grating and index of these parameters, geometry and permittivity of the grating, frequency permittivity of surrounding medium in the layer of the grating, direct lattice vectors, …etc). Moreover, the grating shape function is calculated here by computing 2D FFT and stored in *shapefunc* array.
2. **Compute the solution of the linear system:** Computing the solution is organized in 3 parts: first compute the incident field, and then compute the propagation operator, and finally the linear solver. (See Section 4.1.1)
3. **Post-processing:** this section is responsible for computing the reflection coefficients, where scattered field =total electrical field - incident field. As well as freeing up the all memory allocations and destroying the FFT configurations.

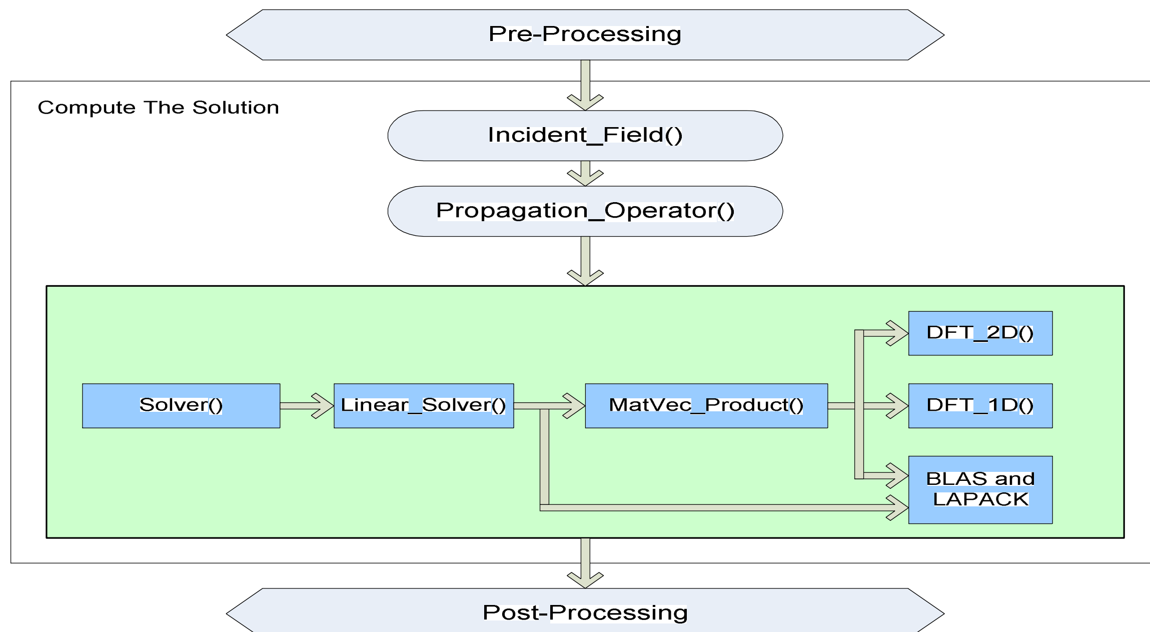


Figure 4.1: Block Diagram of EDM Implementation in FORTRAN

4.1.1. Computing the solution of the Linear System

In this section, the serial implementation of computing the approximated solution will be briefly described. It is split up into the following routines:

1. `Incident_Field(Etheta,Ephi,theta,phi)`: is a routine to compute incident field due to a plane wave and store it in *incfield* array, where *Etheta* is strength of electric field in theta direction, *Ephi* is of electric field in phi direction, theta is angle of incidence w.r.t. the *z*-axis (in deg) and phi is angle of incidence w.r.t. the *x*-axis (in deg). They are all set in main routine.

2. `Propagation_Operator()`: is a routine to compute the Propagation operators for a quasi-periodic source in *x* and *y* directions, expressed in Bloch modes, and a periodic source distribution in *z* direction, given in piecewise linear functions.

3. `Solver()`: is a routine that serves as a wrapper to `Linear_Solver()` routine. It performs the following operations:
    a) It copies incident field array (*incfield*) to right hand side vector (b).
    b) It stores the initial guess of the solution (*x*).
    c) It calls `Linear_Solver()` routine to compute the iterative solution.
    d) It copies back the solution (*x*) to the total electric field (*elecfield*).

4. `Linear_Solver()`: this routine is addressed to compute the solution for the diffraction problem. It requires computing matrix-vector products, and that is done in `MatVec_Product()` routine.

5. `MatVec_Product()`: this routine is responsible for computing the matrix-vector product $A.x \rightarrow y$, where x is a 1-dimentional vector of size $(3*M_1*M_2*(nlayer+1))$ that enters through the argument list of the routine, and M_1 is the number of Fourier modes in the x direction, M_2 is the number of Fourier modes in the y direction, nlayer is the number of the sample points in the z direction, while y is the output vector that is exported via the argument list. The following is the pseudo code of this function:

    a) It copies x to 4-dimentional array elecfield($1{:}3,\ M1_{l}{:}M_{1h},\ M_{2l}{:}M_{2h},\ 0{:}nlayer$).
    b) It calculates derived field as the interaction between the electromagnetic vector field (*elecfield*) and shape function (*shapefunc*). For each Layer, take 3 2D arrays out of the electromagnetic vector field array (*elecfield*).

```
- for each layer n, where n ∈(0 to nlayer)
    - for i= 1 to 3
        - slice(0:mfft-1,mfft-1) = 0;
        - slice(0:M1l-M1h, 0:M2l-M2h) = elecfield(i,:,:,n);
        - Perform forward 2D FFT on slice(0:mfft-1,mfft-1);
        - slice = slice * shapefunc(:,:,n);
        - Perform backward 2D FFT on slice(0:mfft-1,mfft-1);
        - Edev(i,:,:,n)= slice(0:M1l-M1h, 0:M2l-M2h);
    - Endfor
- Endfor
```

c) It calculates the interaction between the derived field and the Propagation operator; this gives the scattered electric field. For each Fourier mode, take 3 1D arrays out of the derived field array (*Edev*).

- for each mode (m_1, m_2) where $m_1 \in (M_{1l};M_{1h})$ and $m_2 \in (M_{2l}:M_{2h})$

  - Compute the contribution of Propagation operator
    - for *i*= 1 to 3
      - elecfieldz(1:nlayer-1,i) = Edev(i,m1,m2,1:nlayer-1);
      - elecfieldz(0,i) = 0 and elecfieldz(nlayer:nfft-1,i) =0;
      - Perform forward 1D FFT on elecfieldz(0:nfft-1,i);
    - Endfor
    - for each layer *n*, where n ∈(0 to *nlayer*)
      - Apply element-wise multiplications on elecfieldz(n,:) with P(n,m1,m2), the derivative of the Propagation operator dPdz(n,m1,m2), the wave vector in the x-direction kxi and the wave vector in the y-direction kyi, where the elements over the second index of elecfieldz are mixed. The result is placed back in elecfieldz(n,:).
    - Endfor
    - for *i*= 1 to 3
      - Perform backward 1D FFT on elecfieldz(0:nfft-1,i);
    - Endfor
  - Compute the contribution of Propagation operator for the first and last points.
    - Perform multiplications on Edev(:,m1,m2,0) with P0(0:nlayer,m1,m2), dPdz0(0:nlayer,m1,m2), kxi and kyi; the result is added to elecfieldz and placed back in elecfieldz.
    - Perform multiplications on Edev(:,m1,m2,0) with PN (0:nlayer,m1,m2), dPdzN(0:nlayer,m1,m2), kxi and kyi; the result is added to elecfieldz and placed back in elecfieldz.
  - Compute the substrate reflections
    - for *i*= 1 to 3
      - Edevz(0:nlayer,i) = Edev(i,m1,m2,0:nlayer);
      - Take the real (non complex, zdotu) inner product with wPr(0:nlayer,m1,m2), the result is stored in innerwPrEdev(i)
      - Determine the vector za(1:3) by multiplying Rdyad(1:3,1:3, m1, m2) with the vector innerwPrEdev (i);
      - elecfieldz(0:nalyer,i)= elecfieldz(0:nalyer,i) + za(i)*vPr(0:nlayer,m1,m2);
    - Endfor

    - for *i*= 1 to 3
      - for each layer *n*, where n ∈(0 to *nlayer*)
        - Edev(i,m1,m2,n) = elecfield(i,m1,m2,n) - elecfieldz(n,i);
      - Endfor
    - Endfor
- Endfor

d) It copies 4-dimentional array Edev(*1:3, M1l:M1h, M2l:M2h, 0:nlayer*) to a *Y* vector of size (*3*M1*M2*nlayer+1*), where M_2 is the Fourier modes in *y* direction, *nlayer* is the number of the sample points in the *z* direction.

4.2. Performance Analysis of The Serial Implementation

A detailed performance analysis of the algorithm may give good insight about application behavior that can help to estimate the performance gains, as well as points to be optimized.

In order to analyze the performance and investigate the behavior of the sequential code, some measurements were done on the building blocks of the sequential code on the CPU, which were generated by hardware profilers. Table 4.1 shows the first timing analysis of the sequential Fortran-based code, which was provided by the Fortran compiler. It reveals that the computation time is largely consumed by *Solver()* and its associated routines, which consume about 88.2% of the total CPU time (See Figure 4.2). This routine takes almost neglectible time by its self computations, because as we saw in Section 4.1.1, it does only a few arrays copying. It is clear from the table that the *MatVec_Product()* routine was the time cost of the this algorithm, which forms about 80% of the total solving time, because this routine is responsible for solving most of the linear equations of the system. And the 2D convolution alone (including the 2D FFT computations and inner product) takes about 50% of the total elapsed time, while about 10% of the time is spent in calculations of the *Linear_Solver()* routine itself, which includes some BLAS and LAPACK routines (i.e. *zcopy*, *zaxpy* … etc). The rest of the execution time is spent in the rest computations and initializations of the code.

Thus, the *MatVec_Product*() is the most important numerical building block in the code that has to receive particular attention. It requires massive computations, mainly computing 1D and 2D FFTs and some arithmetic manipulations of complex data (i.e. inner products, additions,... etc), and that was solved though using FFTW and BLAS libraries.

	Exclusive associated subroutines		Inclusive associated subroutines	
Functions	**CPU sec**	**CPU%**	**CPU sec**	**CPU%**
GRATING2D (main function)	0.0062	0.014	45.5051	100.00
SOLVER	0.0158	0.035	40.1358	88.201
LINEAR_SOLVER	0.4877	1.072	39.6409	87.113
MATVEC_PRODUCT	5.9022	12.970	36.1930	79.536
MYDFT2D	21.0175	46.187	21.6575	47.593
MYDFT1D	4.6390	10.194	4.9279	10.829
ZCOPY	4.3701	9.604	4.3701	9.604
MYDFT_INIT	0.0004	0.001	0.0004	0.001
MYDFT_EXIT	0.0000	0.000	0.0000	0.000
PROPAGATION_OPERATOR	0.4724	1.038	3.4913	7.672
BLAS_ZDOTC	0.3566	0.784	0.3566	0.784
BLAS_ZDOTU	0.000	0.485	0.2208	0.485
BLAS_ZDSCAL	0.9288	2.041	0.9288	2.041
BLAS_ZAXPY	0.9090	1.998	0.9163	2.014
BLAS_DZNRM2	0.4904	1.078	0.4904	1.078
Total Run Time 48.114 sec. Timed Program 45.505 sec. % Timed 94.577				

Table 4.1: Profiling Results of Fortran Code

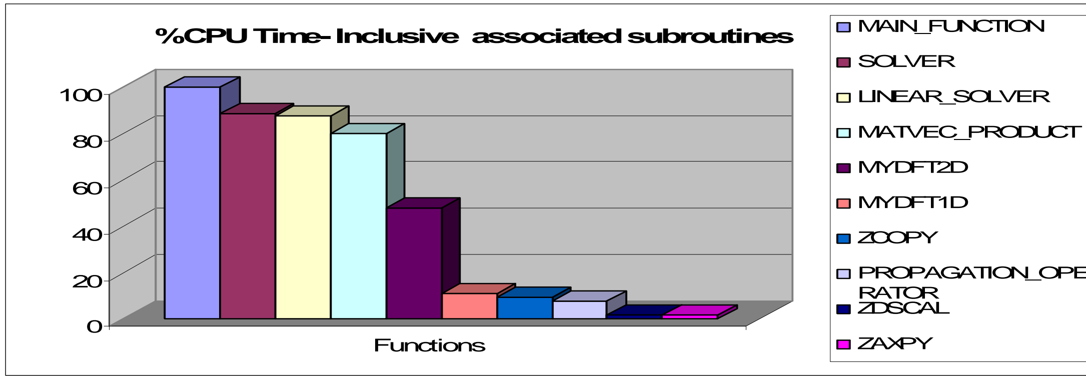


Figure 4.2: A benchmark of the Serial Fortran-based Code, inclusive the associated subroutines

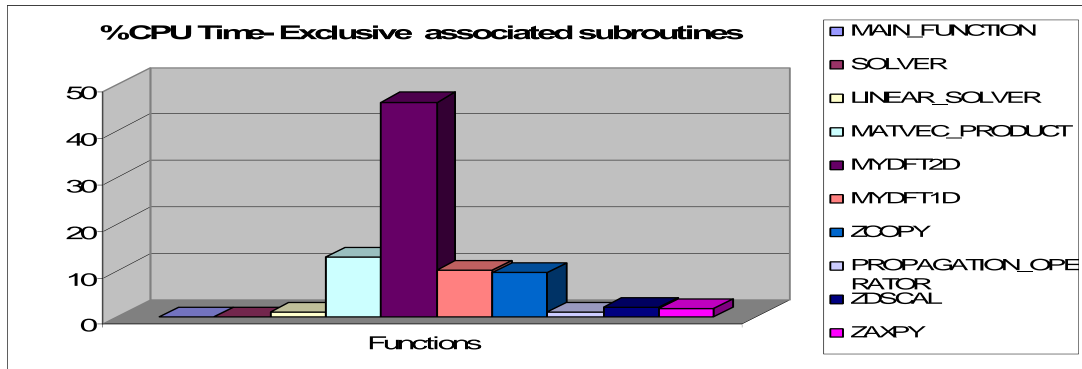


Figure 4.3: A benchmark of the Serial Fortran-based Code, exclusive the associated subroutines

Obviously, from Figure 4.3, we can see that the 2D/1D FFTs require massive processing time, as well as BLAS routines. It is well known that FFT is an intrinsically parallel algorithm; it has the interesting property that its computation is completely data independent, i.e. the operations to be processed are the same whatever the input data. Thus, it can be parallelized quite directly using the abstractions provided by CUDA, such as CUFFT and CUBLAS libraries.

Furthermore, if we take a look at the other parts of the algorithm, we see that some computations are data dependent that must be carried out sequentially— which means it is more efficient to be executed on the CPU (since one CPU core is much faster than one GPU core)— whereas some other portions are data independent, and can therefore be efficiently mapped on the GPU to be executed in parallel. This means, a hybrid system of CPU-GPU will be well suited to such application. On the other hand, as stated in Chapter 2, because of the low memory bandwidth, the overhead caused by transferring the data back and forth to the CPU may limit the parallelism.

Thus, the initial plan was to implement the whole EDM algorithm on GPU to avoid this overhead. However, running a program on a multi-core processor may not always lead to a big factor in speedup, even when the number of the processors is increased because the performance gain is also bounded by the amount of work running sequentially. Thus, if we suppose that we have f fraction of the work done in parallel (on a GPU), that means the performance gain will be constrained by $(1\text{-}f)$ of the work.

Therefore, in order to boost the performance, we should maximize the parallel execution, which means to perform as much work as possible on the GPU. Implementation of

the entire EDM algorithm in CUDA was beyond the scope of this thesis. It was therefore decided to concentrate on parallelizing the most time consuming building blocks of the code (See Table 4.1), namely the *Linear_Solver()* and its associated routines. Within these routines, parallelization is restricted to the most time-consuming routines, e.g. the 2D convolution. So, as a start this thesis focuses on optimizing only this part of EDM algorithm.

Theoretically, the maximum improvement to the overall EDM algorithm when only part of the system is improved can be estimated by Amdahl's law [16]. According to Amdahl's law the relationship between the expected speedup of parallelized implementations of an algorithm relative to the serial algorithm, under the assumption that the problem size remains the same when parallelized is:

$$MaxSpeedup \leq \frac{1}{(1-P)} \tag{4.1}$$

where P is the parallelizable port of the algorithm. Thus, if we can parallelize the matrix-vector product routine, which runs about 80% of the computation, then the maximum speedup of the parallelized version of EDM algorithm according to Amdahl's law is 1/(1-0.8) <= 5x faster than the serial implementation.

But the law is concerned with the speedup achievable from an improvement to a computation that affects a proportion P of that computation where the improvement has a speedup of S.

$$MaxSpeedup \leq \frac{1}{(1-P)+\left(\frac{P}{S}\right)} \tag{4.2}$$

For instance, if the parallelization of that routine makes it 5 times as fast, then the overall achievable speedup of applying the improvement will be 1/((1-0.8)+(0.8/5)) => 2.77x faster than the non-parallelized implementation.

From Table 4.1, we notice that the 2D FFT required for computing the 2D convolution takes about 50% of the total running time. Thus, for example, if we can gain a factor of 5 speedup from parallelizing this portion then the overall speedup of EDM will be 1/((1-0.5)+(0.5/5)) => 1.66x faster. The overall speedup increases when the factor S increases.

However, one should take into account that this estimation neglects other potential bottlenecks such as memory bandwidth. This means, the parallel portion needs to be mapped on the hardware as efficiently as possible.

Therefore, in order to expose the parallelism, preferably entire *MatVec_Product()* routine has to be processed in parallel as kernel programs and offloaded to the GPU. As we just saw from the calculations above parallelizing the whole computation of *MatVec_Product()* routine will lead to a factor of 2.77x performance gain versus only 1.66x by parallelizing the 2D convolution computations alone.

Furthermore, it must be noted that, several parameters can be varied in EDM algorithm in regard to the geometry size— which influence the size of the problem, such as the number of harmonics in x and y directions, the size of the Fourier Transforms and the number of layers in z direction. Table 4.2 shows a wide range of different problem cases that have been approached in this thesis. At present, the problem cases 1 to 5 are the most interesting cases from an computational point-of-view.

Problem Case	# Harmonics in X and Y	FFT Size	# Layers	# Slices (3x(# Layers+1))
1	-3 to 3	16x16	16	51
2	-3 to 3	16x16	24	75
3	-3 to 3	16x16	32	99
4	-7 to 7	32x32	32	99
5	-7 to 7	32x32	64	195
6	-15 to 15	64x64	64	195
7	-31 to 31	128x128	128	387

Table 4.2: A range of Computations Cases

The GPU code will be written in CUDA, and the CPU code can be written in Fortran or C languages. Although Fortran is more suitable for numerical computations, while C is well suited for system programming tasks, I preferred to write the code of the linear solver portion in C, as first task in this thesis (since CUDA is a C-like language and because I am not Fortran expert). The next section gives a brief explanation of translation the code into C and the main issues that must be taken into account.

4.3. Serial C-Based Implementation of Linear Solver

The first plan was to cut down the green colored region of the code shown in Figure 4.1, and translate it to C. Then all the required data for this code segment can be initialized before going through these computations as well as all the calculated results by this code segment can be written back to be used for the rest of the computations.

In the first place, F2C tools were used to translate the Fortran code to C, but it produced a very naïve code. Therefore, a manual translation was performed in order to enhance the quality of the code. All the variables and macros were defined in header files, and every workspace was declared as a structure …etc.

However, this task took longer than expected because many bugs appeared, which needed much time and efforts to be figured out and fixed. A couple of things that one should take into the account when translating a code from Fortran to C are listed below:

1. Array Indices:

   In Fortran the array indices start by default at 1 while in C they start at index 0; hence a passed array Fortran_array[1:10] must be declared as C_array[0:9].

2. Array Storage Order:

   Fortran uses a column wise storage of matrices while C stores them row wise. This means either all the matrices must be transposed before entering the routine or adapting the source code by either changing the order of the matrix dimensions by declaration or switching all the indices by any matrix indexing. Otherwise, the program may probably run but it will generate wrong results.

   We may have to write a transposition wrapper to not modify already existing Fortran and C routines. This can be advisable for reasons of clarity (i.e. keeping the documentation of the code and the math in sync.) However the transposition is expensive, it is better to avoid it by adapting the source code. When the Fortran source

declares an array as Fortran_array(3,4), in C it must be C_array[4][3], or when accessing an array Fortran_array(j+1,i+1) it should mean a[i][j] in C.

3. **Array Dimensions Order:**

Moreover, in Fortran the dimensions are ordered from right to left while in C it is ordered and located in memory in a reverse way. A Fortran_array[w][x][y][z] must be declared in C as c_array[z][y][x][w] to keep *z* dimension as the most outer dim and *w* as the least inner dim. In other words, the order of data located in memory is the same. Only the way how the language represents it—thus, how the compiler interprets the order— is switched.

I had to change the order, because the API of precompiled libraries is in language-native dimension order. That API is a given and transposing back and forth on every call is slow.

4. **Multidimensional Arrays Vectorization:**

The multi-dimensional arrays in C are arrays of pointers that are located in disjoint memory spaces, whereas in Fortran they are stored in contiguous memory spaces as one joint block. Thus, since all the routines as well as the BLAS and FFTW3 libraries expect to receive vectors that are located contiguously in the memory, this introduced a big issue while translating the EDM application. The best that can be done is to declare the multi-dimensional arrays in C as one dimensional arrays with a comment stating the actual number of dimensions. But, the code must carry out array indexing using the dimension information.

However, converting the array indexing, of such big and complicated code as EDM code, requires a lot of effort and time. This problem was solved eventually by using the scheme shown in Figure 4.4, where we still can declare the multi-dimensional arrays and indexing them in the same standard way known in C, while locating them in the memory in contiguous memory spaces as one vector as it happens in Fortran. Thus, instead of making the pointers point to separated addresses in the memory they can point to adjacent locations. Figure 4.4 clearly depicts this behavior. Note that programming it is very difficult since it deals with nested C pointers.

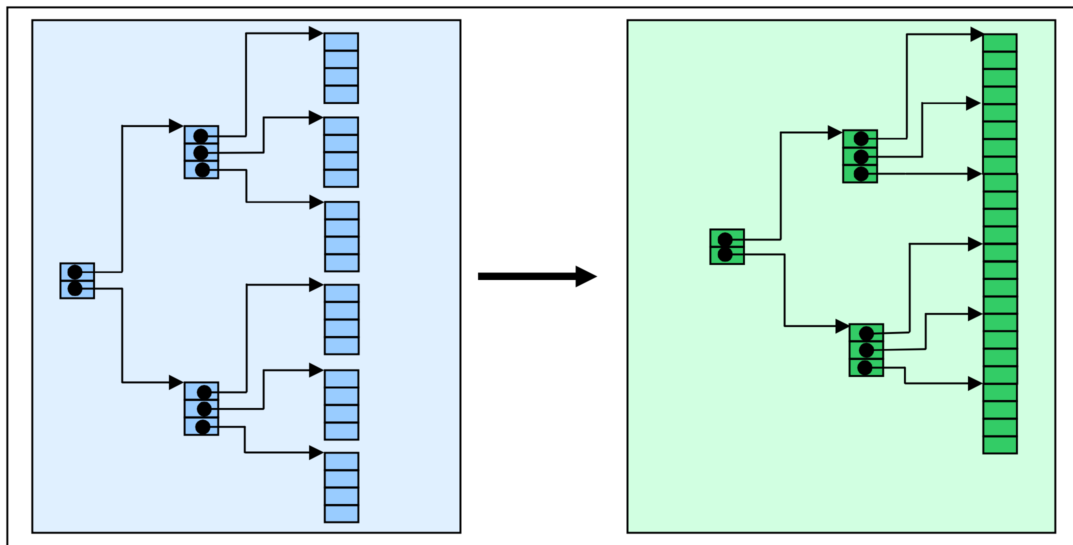


Figure 4.4: A representation of multi-dimensional arrays for an array C_array[2][3][4]

5. Call by Reference:

   Fortran by default passes its variables by reference (passes pointers). That means one should pass addresses when calling C-functions.

6. **Variable Type Matching:**

   The size of the variables in the C-routine must be identical to the size in the Fortran routine. Thus, one should carefully consider the data type of the variables, especially with float and double data types. For instance:

   float --- REAL

   double --- REAL*8

7. **Character Strings:**

   Fortran may pass string lengths by value in the order the strings appear in the argument list. These will not appear in the Fortran argument list, but will appear in the C argument list. Thus, one may need to pay attention for nulls and blanks spaces explicitly if wants to ignore this.

Furthermore, some libraries were used in this application such as Complex, FFTW3, BLAS and LAPACK Libraries. For compiling the C code I used gcc compiler under windows (cygwin), where the Complex and LAPACK libraries were not supported by default. Due to difficulties encountered when installing and configuring Complex and LAPACK libraries, I implemented a trivial version of the Complex library. Some other implementations provided by ASML in Fortran, replaced the required LAPACK routines.

As a final step in this stage of the thesis work, a hardware profiling was done on the C-Based building blocks. Table 4.3 displays a summary of the timing results that were measured by Oprofiler tools. It must be noted, that it shows only the results for the relevant functions to the Solver() routine and its associated routines. The results were obtained on one Core CPU of speed 1.86 GHz. As we see from Table 4.3, reading the inputs and writing the outputs consume much time. For fair comparison to the Fortran profile, we may ignore the I/O time, since it is not relevant to the Fortran-based code but it was used only for verifying the results integrity and debugging the code. Consequently, the profiling results (Figure 4.5) are roughly the same as for Fortran, (see Table 4.1). Note that, about 9% of the time taken by the main function is spent in arrays allocation and deallocation as well as FFTW3 library initializations.

	Inclusive associated subroutines		
Function Name	**CPU sec**	**%CPU of the total Time**	**%CPU of the total Time excluding I/O**
Main Function	90.2	100.0%	100.0%
READING+WRITING	16.99	18.83%	-
SOLVER	66.71	73.95%	91.12 %
LINEAR_SOLVER	66.15	73.32%	90.35%
MATVEC_PRODUCT	60.58	67.16%	82.74%
FFTW3 LIBRARY	43.53	48.26%	59.45%
ZCOPY	3.97	4.41%	5.42%

Table 4.3: Profiling Results of C-based Code

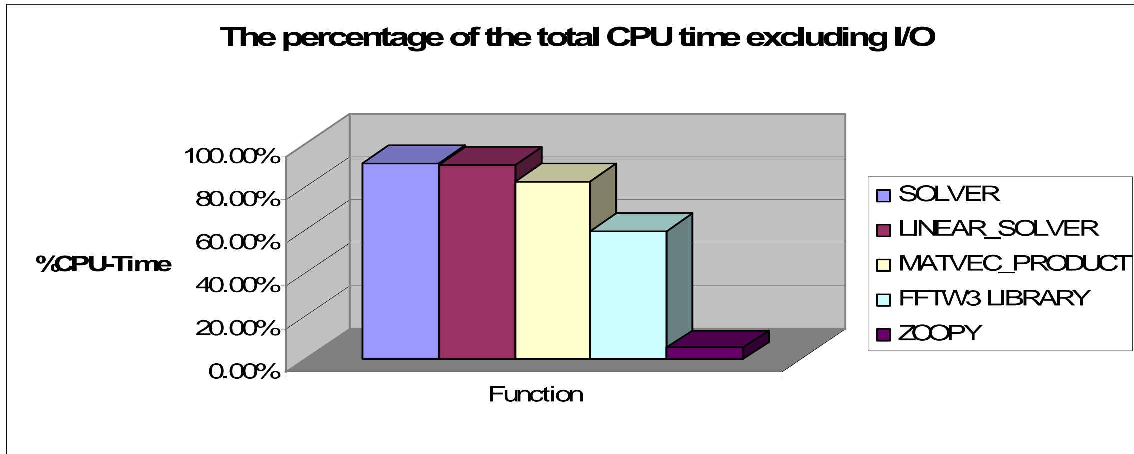


Figure 4.5: A benchmark of the Serial C-based Code, exclusive the associated subroutines

Although tests with small geometry size that have 7 or 15 harmonics in x and y directions are the most desired from an applicational point-of-view, larger problem sizes were also experimented and profiled in order to obtain reasonable performance results, as well as to see whether the memory on the targeted GPU is sufficient for the application for such cases.

The next section deals with examining the accuracy of the output results with respect to the precision of the input data. It explains and compares different precision-based implementation versions.

4.4. Results Precision Validation

The floating-point hardware performs calculations in either IEEE single-precision (SP) or IEEE double-precision (DP). DP provides higher precision (64-bits, which is about 16 decimal digits) than SP (32-bits, which is about 8 decimal digits of precision). DP is always more accurate, however it does run slower (on 32 bit machines at least) and takes up more memory storage than SP.

All the data of EDM application are complex numbers which are composed of two floating-point values. To achieve higher accuracy and rate of convergence, the computations of this model were implemented based on double floating-point precision. Earlier tests of EDM in the framework of reconstruction demonstrated the importance of DP. But, when I started my thesis, NVIDIA did not yet release a version of CUDA FFT library that supports DP floating point calculations.

Since the GPU architecture has a factor 12x lower (theoretical) DP throughput compared to the SP, this motivated extending our study to SP. However, it is expected that the next generations of NVIDIA GPUs will dramatically increase DP performance. This has recently been confirmed with Fermi GPUs. Besides, programs that manipulate large amounts of floating-point data may run faster if they use single rather than double variables, because the smaller data size reduces memory traffic, which can form a performance bottleneck for the application. But, it is necessary to ensure that float variables provide an acceptable range of precision. Therefore, as a result, we decided to validate the precision of the model's results by running the application in SP and comparing the results against the ones in DP.

Actually there was a suspicion whether the solution will not have enough resolution to resolve the small value of the residual accurately, leading to problems with accurate predictions

of the solution etc, ultimately leading to unstable linear solver behavior and probable divergence. To verify the application behavior and the stability of the linear solver, I ran EDM application in SP. The results show that single precision version works safely and faster while sacrificing a little amount of precision. Moreover, the linear solver converges also with the SP and with same number of solving steps as its counterpart in DP.

The timing comparison (on an Intel Xeon/Linux machine) reveals that the DP took 30%-50% longer than SP. This assumes that the linear solver converges OK with both double as well as single precision.

The results of the solution (total electrical field) obtained from the C code, comparing to the ones obtained from the Fortran code, shows that the SP provides a variable range of precision with a minimum of about 4 to 7 decimal digits, while the DP format introduces accuracy of about 13 to 14 decimal digits or more.

Therefore, mixed precision was proposed that mixes the single and double precision to significantly increase the accuracy of the application through exploiting the accuracy of double precision computations, while keeping optimal execution time by exploiting the efficiency of the single-precision calculations. This technique runs the matrix-vector product calculations in SP while the linear solver is carried out in DP, which means calculating the residual in DP. The overall results conduct same number of solving steps with some gain in performance comparing to the double precision version, however they still show almost same precision as SP, because most of the computations (matrix-vector product) are performed in SP. Besides, it shows slightly lower performance than the single precision version.

Usually, the computations that mix single and double operands require conversions of the SP operands to DP and/or vice verse. These conversions cause a noticeable performance impact to the complete program. On the other hand, DP values that are converted to SP require rounding operations. The values, thus, may be less precise than the original DP values, as a result of rounding error. Beside, due to the rounding operations, conversions from DP to SP values may reduce the performance a little bit more.

Thus, due to the data conversions impact and running part of the computations in DP, the proposed mixed precision version slowed down the application performance in comparing to the single precision version. As a result, we notice that not much benefit as regards to the performance and/or the accuracy may be gained using this mixed precision version in comparison with the counterpart versions. However, it must be noted that this investigation was done on a limited range of computational problem cases (i.e. Case 7, Table 4.2).

Nevertheless, to increase the precision of the mixed precision version, one should attempt to increase the number of solving steps. But, note that this may sacrifice the performance. As the prototypical algorithm proposed in [36], the linear solver runs over two loops (outer and inner loops), where the matrix-vector product computations run over inner loop in SP, whereas the other computations (computing the residuals) is done within outer loop in DP.

From this, one can conclude that the single precision version of the application is faster than the equivalent versions that run in double and mixed precisions while it keeps almost same precision as the mixed precision calculations. Despite this result, DP is preferred for reconstruction to prevent unexpected error contribution from the rigorous modeling. Future GPU generations like Fermi, fulfils the need for DP high performance computing and are definitely worth a careful evaluation.

4.5. Summary

This chapter describes how EDM application is implemented serially in Fortran and C languages. In addition, it highlights the important points of the implementation that have to receive special attention. Besides, it evaluates the performance of the application and investigates the behavior of the code running on CPU. Thereby, we could detect the bottlenecks and the duration and relation of the building blocks of the serial code.

Moreover, it estimated the performance gain, and found out the parts where we can start optimizing to boost the program speed and memory usage. Thus, this thesis will focus on to parallelization and implementation of 2D convolution algorithm on NVIDIA Tesla C1060 GPU using CUDA version 2.3. Next chapter will describe the parallel implementation of this algorithm on the CUDA architecture and determine how much performance improvements can be achieved by implementing it on such highly parallel hardware.

5 GPU-Based Parallel Implementations and Optimizations

This chapter focuses on how to implement an optimal parallel code that computes the 2D convolution in parallel across the multiprocessors of GPU and how to achieve more parallelism of this algorithm to take full advantage of the underlying architecture features.

Firstly, the pseudo-code in Algorithm 5.1 needs to be carefully analyzed in order to find further details that may need to receive special attention. As we see, this algorithm traverses all the slices of the layered stack sequentially. Each time it executes multiple sequentially dependent kernels; but per each kernel, the data are independent. Besides, since the data are also independent per one slice, thus, this can be processed in parallel. Therefore, that may make the algorithm very well-suited for multi-threaded hardware like GPU. Furthermore, this algorithm is typically memory intensive transferring, so the memory may form a bottleneck for the performance (in particular for the large problem cases). The total computation time is spent in performing the forward and backward FFT computations; element-wise multiplication, determining indices of each slice, loading and storing data and computing loop parameters, but computing the FFT consumes the majority of cycles.

- Algorithm 5.1: Pseudo Code of the serial implementation of the 2D convolution algorithm:

- For n=0:Nlayers+1
  - For i=1:3
    - slice(0:mfft,0:mfft)=0 //zero padding
    - slice(0:MBlo1, 0:MBlo2)=elecfield(i, :, :, n)//cutting off a slice from elecfield
    - Forward 2D FFT(slice(0:mfft, 0:mfft))
    - For M_1=0:mfft
      - For M_2=0:mfft
        - slice(M_1, M_2)=nevt(M_1,M_2)*shapefunc(M_1, M_2) //element-wise multiplication
      - EndFor
    - EndFor
    - Inverse 2D FFT(slice(0:mfft, 0:mfft))
    - Edev(i, :, :, n)=slice(0:MBlo1, 0:MBlo2) //inserting a slice into Edev
  - EndFor
- EndFor

This chapter presents a sequence of kernels, which progressively address various performance bottlenecks, and shows how convolution of a 2D data array can be efficiently implemented on GPU using the CUDA programming model. The first part of this chapter shows a straightforward parallel implementation of the 2D convolution computations which is solved by using CUDA FFT library— through the use of the standard available release. Then we walk through optimizing this algorithm using different implementations, ranging from

batching 2D FFTs, stripping out some transpose kernels and merging the point-wise multiplication kernel, varying the execution configurations for various problem sizes, to using the shared memory, for one and multiple versions of EDM application.

5.1. 2D FFT-Based Approach

Since performing the direct/inverse FFTs consumes most of the time, the first obvious idea to parallelize this algorithm was computing only the 2D FFT in parallel on GPU using CUFFT library, while the element-wise multiplication kernel was left to be processed on CPU because it does not consume much time. This strategy did not yield any speedup. Especially with small problem cases, it even gave delays, because it is limited by the communication overhead between CPU-GPU in each iteration. Thus, to reduce the costly data transfers, it might be better to store the intermediate data in the device memory, if possible. If we take a look at the serial code, we find that the element-wise multiplications consists of loops where each loop is independent of all the others. Such loops can be basically batched into a parallel kernel; where each loop iteration becomes an independent thread.

In this way, we can operate on the results of the forward 2D FFT without ever being copied to host memory, so we can calculate the scalar products and hence apply the backward 2D FFT on the results before the data would be mapped to the host memory. That could introduce some performance gain because low parallelism computations can sometimes perform faster than transferring back and forth to the host (See page 68 in [2]).

The first parallel version of this algorithm is strikingly similar to the serial one. Such a parallel algorithm can be written in CUDA in a fairly straightforward manner. The host code performs typical tasks: data allocation and transfer between host and device, launching and timing the parallel kernels, and the de-allocation of host and device memory. The data-independent sections of the program were identified and implemented as kernels and mapped to the GPU. The input and output of the kernels are data arrays transferred and stored as stream vectors in GPU memory. The following are the steps to perform 2D convolution:

- For each point in z direction and for each field component:
    1. Padding each stream with zeros.
    2. Upload the data stream to the GPU.
    3. Setting up the CUDA execution configuration.
    4. Forward 2D FFT is applied onto the input data.
    5. Multiplying the results data array with the shape function.
    6. Backward 2D FFT is calculated for the results of the multiplication.
    7. The resulting array is copied back to the CPU.

As a first attempt, the 2D FFT was solved based on using the latest available release of CUFFT library from NVIDIA. For each point in z direction and for each field component, we transfer one slice (*slice*), of the electrical field array (*elecfield*), of size MFFTxMFFT to the GPU to compute a forward 2D FFT in parallel and multiplying the results by the shape function (*shapefunc*)— which is stored beforehand on the GPU— also in parallel, and then computing the inverse 2D FFT on the results in parallel as well, later sending it back to the CPU to be stored in the derived field array (*Edev*), see Algorithm 5.2. This procedure will be repeated serially on the GPU over all the slices as demonstrated in Figure 5.1. In this way, only one slice can be processed at a time and the parallelism will be done only among slice's elements per one kernel launch. It must be noted, that all the intermediate values will need to be stored temporarily in device global memory per each iteration.

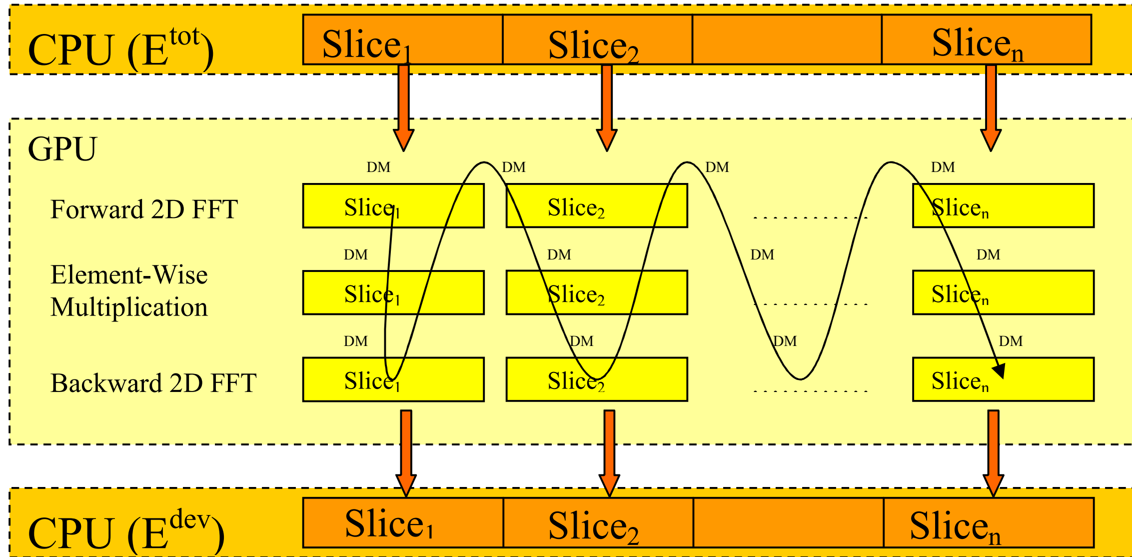


Figure 5.1: Schematic overview of GPU computations workflow of trivial Implementation; DM: Device Memory

- Algorithm 5.2: The pseudo code of the first parallel implementation of the 2D convolution algorithm, using the standard Imp. of 2D FFT from Nvidia, where *elecfield* on CPU:

```
• For n=0:Nlayers+1
  o For i=1:3
     ▪ slice(0:mfft,0:mfft)=0; //zero padding
     ▪ slice(0:MBlo1, 0:MBlo2)=elecfield(i, :, :, n); //cutting off a slice from elecfield
     ▪ gpu_Ptr(mfftxmfft)=slice(0:mfft,0:mfft); //Copy the data to the GPU
     ▪ Forward 2D FFT(gpu_Ptr(mfftxmfft));
     ▪ Multiply<<<gridDim, blockDim>>>(gpu_Ptr(mfftxmfft)*gpu_shapefunc(0:mfft,0:mfft))
     ▪ Inverse 2D FFT(gpu_Ptr(mfftxmfft));
     ▪ slice(0:mfft,0:mfft) = gpu_Ptr(mfftxmfft); //Copy the data back to the CPU
     ▪ Edev(i, :, :, n)=slice(0:MBlo1, 0:MBlo2); //inserting a slice into Edev
  o EndFor
• EndFor
```

However, the communication overhead between the CPU and GPU is still high and hides the performance gained by the naïve way of parallelizing the computations. So, the best sequential implementations are barely outperformed by parallel algorithms because of long memory latencies and high synchronization costs. The memory latency is a fixed time which is associated with each data transfer between the CPU and the GPU. Consequently, by many transfers, the total latency time will be too high. According to NVIDIA *"because the overhead associated with each transfer, batching many transfers into one always performs much better than making each transfer separately"*, [2]. As a result, all the data streams can be grouped in one transfer and stored in the device memory; thus, in this way, we can save the costly small multiple data transfers.

On the other hand, the data structure of the total electrical field *(elecfield)* is not allocated in the memory as simple regular grids as shown in Figure 3.6. Therefore, the data of the total electrical field must be rearranged to be a continuous sequence of slices as presented in that figure. Afterwards, it can be packed into one stream vector and sent to the GPU in one step followed by looping over all the slices on the GPU (see Algorithm 5.3). This obviously reduced the memory latencies.

- Algorithm 5.3: Pseudo Code of the parallel 2D convolution algorithm (Restructring all the data in one vector), where looping over all the slices is done on the GPU:

- conv(3x(Nlayers+1)x mfft x mfft)=0; //zero padding
- conv(:)=elecfield(:, :, :, :); //Reconstructing the data of elecfield
- gpu_Ptr(3x(Nlayers+1)x mfft x mfft)=conv(:); //Copy the data to the GPU
- For i= 1: (3x(Nlayers+1))
    - Forward 2D FFT(gpu_Ptr(i*mfft*mfft));
    - Multiply<<<gridDim, blockDim>>>(gpu_Ptr(i*mfft*mfft),shapefunc(0:mfft,0:mfft));
    - Inverse 2D FFT(gpu_Ptr(i*mfft*mfft));
- EndFor
- conv(:)=gpu_Ptr(3x(Nlayers+1)x mfft x mfft); //Copy the data back to the CPU
- Edev(:, :, :, :)=conv(:); //save the results back to Edev

Since all the matrices were stored in the device memory as one big vector, we need to calculate the based address of each matrix (of size MFFTxMFFT) in each iteration. Calculating the offsets may be done in two different fashions: calculating it by kernel launches, so the outer-loop adds additional offsets to the index, thereby a relative pointer is passed, or calculating it within the kernel by each thread over the load and stores. The time spent in calculating the matrix indices in the latter way shows more overhead than calculate it by the kernel launch becase the base address will be computed by each thread versus computing it only once on the host. Figure 5.2 shows an overview over the GPU behavior with Algorithm 5.3 for various computational problem cases (Table 4.2).

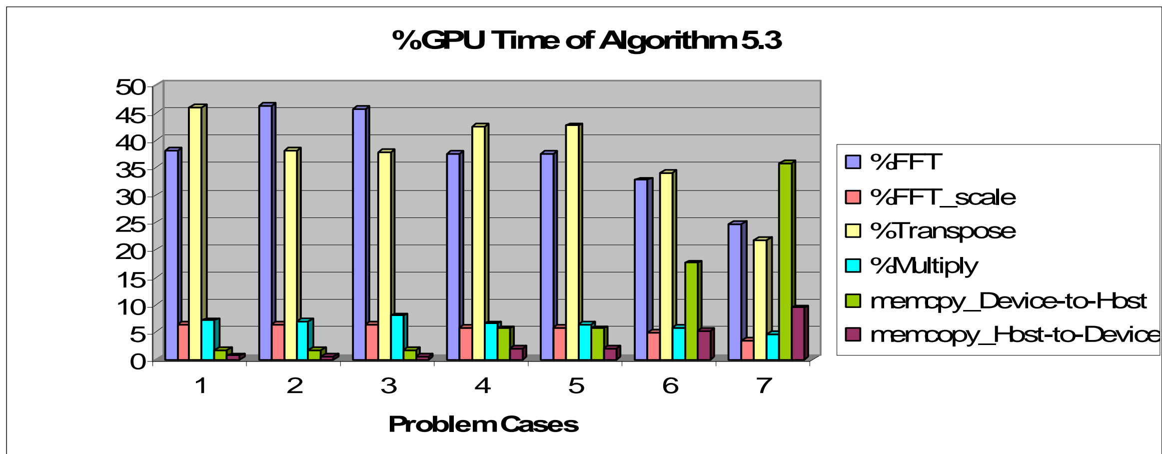


Figure 5.2: A summary plot of the GPU time percentage of Algorithm 5.3, excluding the data transfer

Minimizing the data transfers between the host and the device could enhance the

running time a little bit, because that almost neglects the latency. However, these changes could not consistently improve the application performance, because the 2D FFT computations dominate as it is clear from Figure 5.2. That might be because the device memory accesses forms a bottleneck for the performance, since the device memory bandwidth is too low. Therefore, in order to maximize performance and hide memory-latencies, more computations should be done per one memory access or more threads should be run in parallel. This way, enough background is given to show that a different programming paradigm must be developed to take full advantage of the high processing power available in GPUs.

In order to achieve that, two different approaches can be attempted. As it has been seen from Figure 3.9, the algorithm can be applied on the slices independently, therefore, one may apply the parallelism (vertically) in slice-level through merging the computations of the three kernels in one kernel and hence run it in parallel over all the slices as illustrated in Figure 5.3 (a). Consequently, more computations can be performed per one memory access. However, since we use a pre-compiled library (i.e. CUFFT API) to compute the 2D FFTs, it is difficult to perform this attempt because we do not have access to this library. Beside, CUFFT interface launches a kernel, rather than provides GPU routines to call from GPU code. On the other hand, implementing it is too complicated.

Furthermore, since as stated earlier, the data per each kernel are independent, one can adapt another approach, where the parallelism can be done (horizontally) in kernel-level by executing one kernel at a time, such that the computations done in each kernel can be applied on all the slices in parallel, see Figure 5.3 (b). In this way, more threads can be run concurrently. However, the current release of CUFFT library (version 2.3) does not support "batch execution" feature to execute 2D FFT for multiple matrices in parallel as it is with executing 1D FFT. That is how we came with the idea to break up the 2D FFT kernel and use the batched 1D FFT kernel to solve it in order to take the advantages of the batching option. Next section describes this technique in more details.

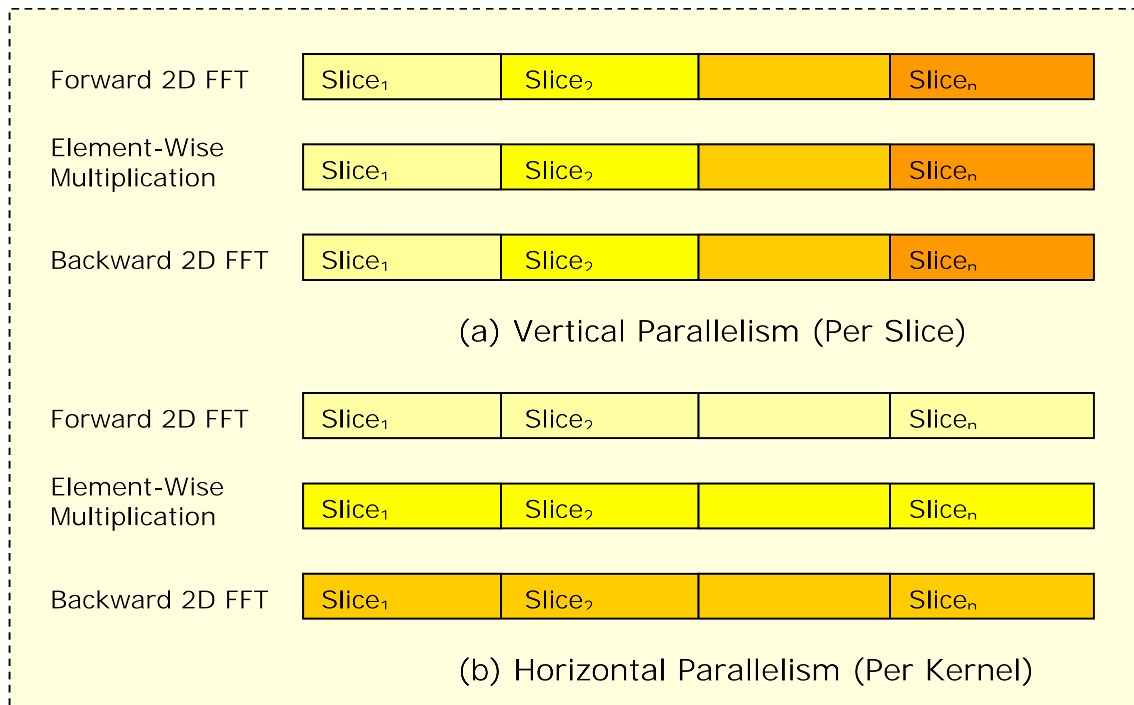


Figure 5.3: Schematic representation of the levels of Parallelism

5.2. 1D FFT-Based Approach

Computing the 2D FFT is then simply a matter of looping over all rows and computing the 1D FFT for each row and consequently looping over all columns and computing the 1D FFT for each column (See Figure 5.4).

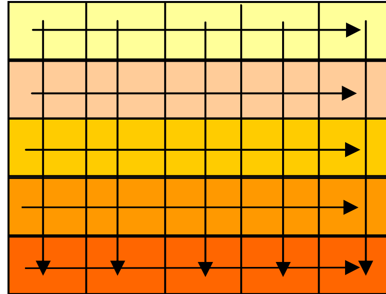


Figure 5.4: Conceptual Graph for solving 2D FFT using 1D FFT

So, we can compute 1D FFT for all the rows of a matrix in parallel and then compute the 1D FFT on all the columns we transpose the matrix and compute 1D FFT for the all rows of a matrix again in parallel and hence transpose back the matrix to get the right order of the elements as shown in Figure (see Figure 5.5).

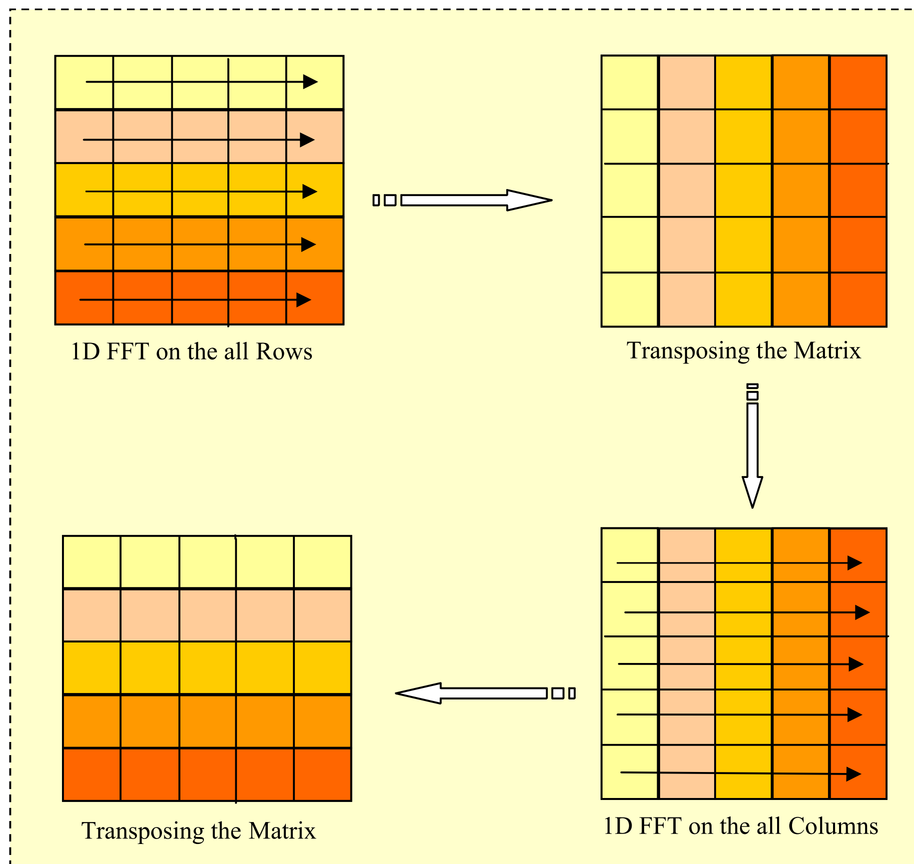


Figure 5.5: Scheme for solving 2D FFT using 1D FFT and Matrix Transpose

Since we already re-arranged the data order as a sequence of slices, we can calculate the 1D FFT of all the rows of all the matrices in one go using the batch option. Implementing the 2D FFT using this technique make the 2D convolution algorithm several times faster than the 2D FFT-based approach without the batching option. Algorithm 5.4 shows the pseudo code of this technique.

- Algorithm 5.4: Pseudo Code of the 2D convolution using 1D FFT-based approach:

```
o conv(3x(Nlayers+1)x mfft x mfft)=0;
o conv(:)=elecfield(:, :, :, :);
o gpu_Ptr(3x(Nlayers+1)x mfft x mfft)=conv(:);
o 1D FFT(gpu_Ptr(:), NSlices);                                        }
o For i= 1: (3x(Nlayers+1))                                           }
    • Transpose<<<gridDim, blockDim>>> (gpu_Ptr(i*mfft*mfft));        }  //Forward 2D FFT
o 1D FFT(gpu_Ptr(:), NSlices);                                        }
o For i= 1: (3x(Nlayers+1))                                           }
    • Transpose<<<gridDim, blockDim>>> (gpu_Ptr(i*mfft*mfft));        }
o For i= 1: (3x(Nlayers+1))
    • Multiply<<<gridDim, blockDim>>>(gpu_Ptr(i*mfft*mfft),shapefunc(0:mfft,0:mfft));
o 1D FFT(gpu_Ptr(:), NSlices);                                        }
o scale(gpu_Ptr); //normalizing the results                           }
o For i= 1: (3x(Nlayers+1))                                           }
    • Transpose<<<gridDim, blockDim>>> (gpu_Ptr(i*mfft*mfft));        }  //Inverse 2D FFT
o 1D FFT(gpu_Ptr(:), NSlices);                                        }
o scale(gpu_Ptr); //normalizing the results                           }
o For i= 1: (3x(Nlayers+1))                                           }
    • Transpose<<<gridDim, blockDim>>> (gpu_Ptr(i*mfft*mfft));        }
o conv(:)=gpu_Ptr(3x(Nlayers+1)x mfft x mfft);
o Edev(:, :, :, :)=conv(:);
```

The approach used in Algorithm 5.3 is more straightforward and leads to simpler code, however, as observed by Table 5.1, it performs several times slower than the 1D FFT-based approach.

FFT size	Algorithm 5.3 (2D FFT- based approach)	Algorithm 5.4 (1D FFT- based approach)	Speedup	Block size
128x128	0.0570	0.0279	2.0x	16x16
64x64	0.0181	0.0083	2.2x	8x8
32x32	0.0086	0.0033	2.6x	4x4
16x16	0.0045	0.0015	3.0x	2x2

Table 5.1: Speedup Statistics of using Algorithm 5.4 vs Algorithm 5.3 - Timing results in sec

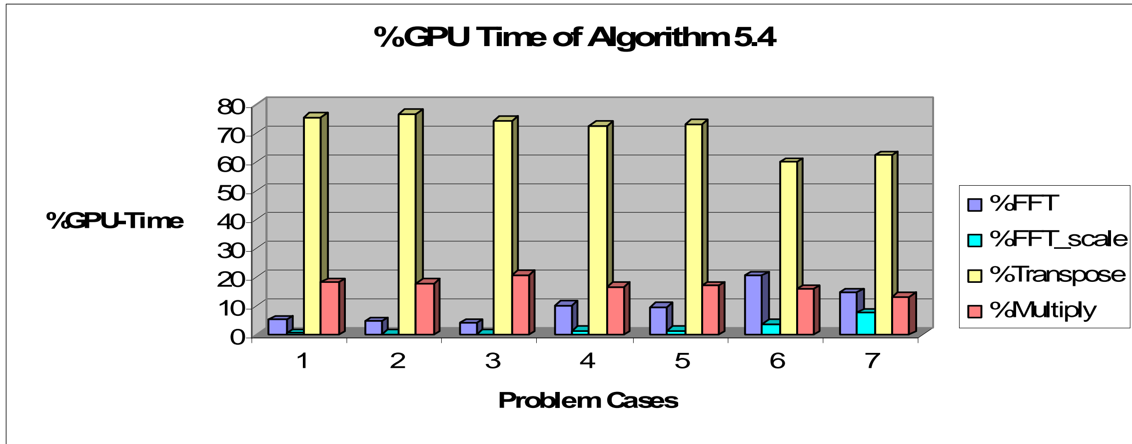


Figure 5.6: A summary plot of the GPU time percentage of Algorithm 5.4, excluding the data transfer

However, as we see in Figure 5.6, after parallelizing the FFT computations for all rows and columns, the FFT computations do not take high percentage of the total computations time anymore. But on the other hand, we see that the percentage of the transpose kernel dominates in all cases. That is because the matrix transposing is done four times, and each time it launches *((NLayers+1)*3)* parallel kernels sequentially, where one kernel launch performs one matrix transpose in parallel. That is a real problem with a small matrix size, because not many active threads per one kernel call will be executed concurrently. In this way, we can not gain much benefit from the high parallism available in the GPU. This can be optimized first by reducing the number of transposes. Since the forward 2D FTT is followed by inverse 2D FFT afterwards, and as we know applying 1D FFT on the rows first and then on the columns, or vice verse, leads to same results, we can strip out two extra transposes, and apply the multiplication in a transposed way, see Figure 5.7.

Thus if we merge the three kernels in Algorithm 5.4, then two matrix transposes can be left out as illustrated in Algorithm 5.5.

- Algorithm 5.5: 2D FFT Implementation using 1D FFT-based optimized approach (Stripping out 2 transposes):

- 1D FFT(gpu_Ptr(:), NSlices);
- For i= 1: (3x(Nlayers+1))
    - Transpose<<<gridDim, blockDim>>> (gpu_Ptr(i*mfft*mfft));
- 1D FFT(gpu_Ptr(:), NSlices);

//Forward 2D FFT

- For i= 1: (3x(Nlayers+1))
    - Multiply<<<gridDim, blockDim>>>(gpu_Ptr(i*mfft*mfft),shapefunc(0:mfft,0:mfft));
- 1D FFT(gpu_Ptr(:), NSlices);
- scale(gpu_Ptr); //normalizing the results
- For i= 1: (3x(Nlayers+1))
    - Transpose<<<gridDim, blockDim>>> (gpu_Ptr(i*mfft*mfft));
- 1D FFT(gpu_Ptr(:), NSlices);

//Inverse 2D FFT

- scale(gpu_Ptr); //normalizing the results

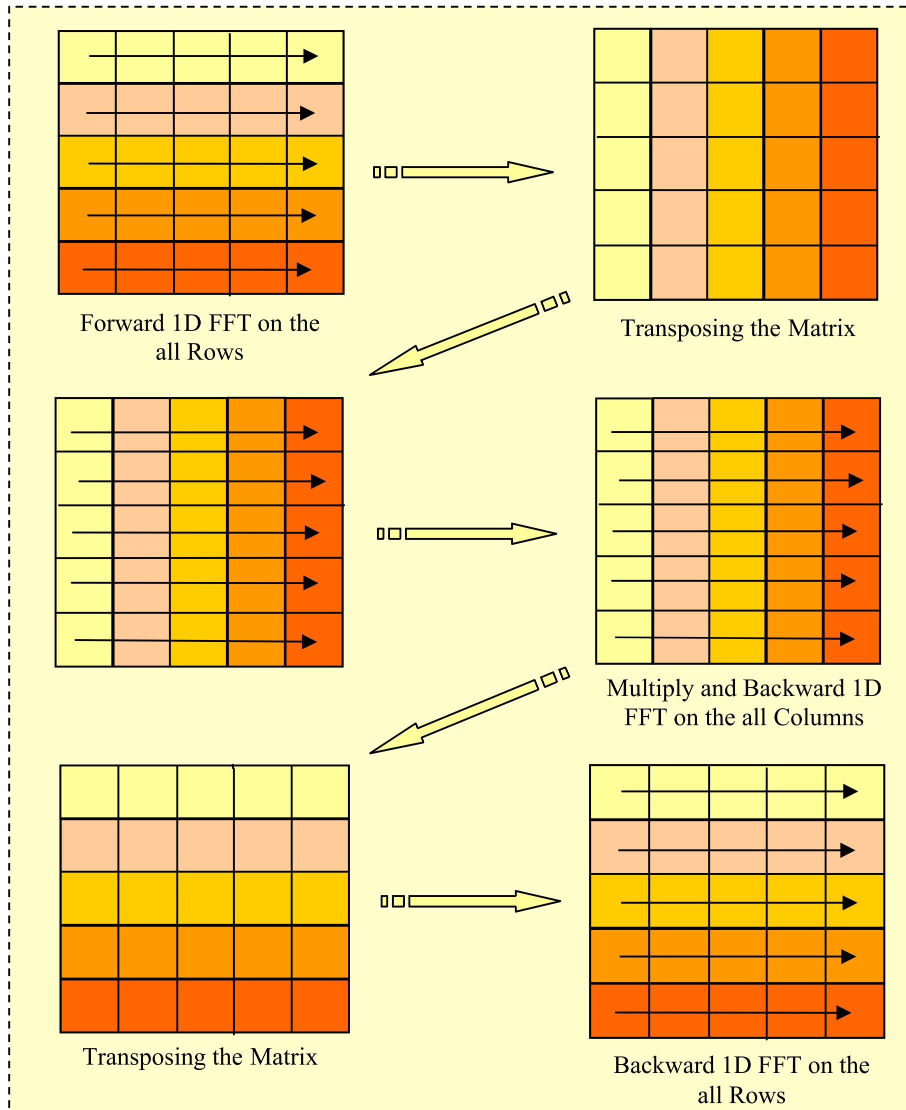


Figure 5.7: Scheme for solving 2D convolution algorithm using 1D FFT-based Approach

Table 5.2 shows some statistics about the execution time taken by the different implementations of 2D convolution for different problem cases.

FFT size	GPU Time of Algorithm 5.3 (in sec)	GPU Time of Algorithm 5.4 (in sec)	GPU Time of Algorithm 5.5 (in sec)	Speedup of Algorithm 5.5 vs Algorithm 5.3	Speedup of Algorithm 5.5 vs Algorithm 5.4	Block size
128x128	0.0570	0.0279	0.0186	3.0x	1.50x	16x16
64x64	0.0190	0.0083	0.0050	3.8x	1.43x	8x8
32x32	0.0083	0.0033	0.0021	3.95x	1.57x	4x4
16x16	0.0041	0.0015	0.0009	4.5x	1.66x	2x2

Table 5.2: Speedup of using Algorithm 5.5 vs Algorithm 5.3 and 5.4- Timing results are exclusive the data transferring

It is so clear from Table 5.2 that Algorithm 5.5 effectively shows much better performance. Besides that, there is still a room for more optimization this algorithm. As we see in Figure 5.8, the transpose kernel still dominates, so optimizing this kernel may lead to significant improvements. Likewise with the multiply kernel.

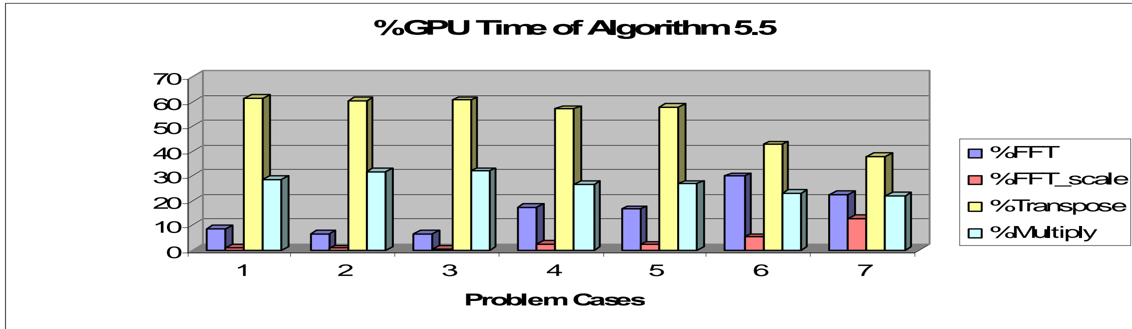


Figure 5.8: A summary plot of the GPU time percentage of Algorithm 5.5, excluding the data transfer

This can be taken one step further to find a technique that can utilize the GPU more efficiently where we can batch these two kernels as to execute more threads per a kernel launch and ensure more active threads per MP, as to avoid any serialization between the slices on the CPU, see Algorithm 5.6.

However, we need to make sure that the resources usage is low enough, to support multiple active thread blocks per multiprocessor. The profiling results show that maximum 10 registers are needed per thread. Thus, according to NVIDIA as in the equation shown in page 62 in [2], we may have up to 64 active blocks of 256 threads per each. Nevertheless, theoretically, there is an upper limit of 768 active threads that could be run concurrently per multiprocessor. Therefore, that means, registers availability will not be an issue in our case because the total number of the registers per MP is still quite enough to compile and invoke successfully as well as to run several blocks at a time.

The next section discusses all optimizations that were done to exploit the power of parallism of the GPU when it deals with small matrix size.

- Algorithm 5.6: Pseudo Code of Bached 2D convolution (Batching Transpose and Multiply):

```
o conv(3x(Nlayers+1)x mfft x mfft)=0;
o conv(:)=elecfield(:, :, :, :);
o gpu_Ptr(3x(Nlayers+1)x mfft x mfft)=conv(:);
o 1D FFT(gpu_Ptr(:));                                   }
o Transpose<<<gridDim, blockDim>>> (gpu_Ptr(:));        } // Batched Forward 2D FFT
o 1D FFT(gpu_Ptr(:));                                   }
o Multiply<<<gridDim, blockDim>>> (gpu_Ptr(:), shapefunc(0:mfft,0:mfft));
o 1D FFT(gpu_Ptr(:));                                   }
o scale(gpu_Ptr); //normalizing the results             }
o Transpose<<<gridDim, blockDim>>> (gpu_Ptr(:));        } // Batched Inverse 2D FFT
o 1D FFT(gpu_Ptr(:));                                   }
o scale(gpu_Ptr); //normalizing the results             }
o conv(:)=gpu_Ptr(3x(Nlayers+1)x mfft x mfft);
o Edev(:, :, :, :)=conv(:);
```

5.3. CUDA-Based Optimizations

In this section, we will go through optimizing the transpose and multiply kernels that operate out-of-place, where the input and output matrices address separate memory locations. Each optimization addresses different performance issues and deals with a range of computation cases. Most of the optimization techniques were adapted from NVIDIA optimization guide, see [35].

For the transpose, the relevant performance metric is the effective bandwidth, calculated in GB/s as twice the size of the matrix – once for reading the matrix and once for writing – divided by the time of execution. It must be noted that for the multiply kernel, it will be read, performing multiplication and then write.

In the begining, I started my experiments with a big problem size which is problem Case 7 (Table 4.2), where it has a high FFT size and very high number of layers, in order to have valuable measurements results and to make sure if it is applicable for GPU architecure. This case turned out to a good performance comparing to the serial implementaion on the CPU, and that due to that fact that computing FFT for all these elements (samples) sequentially takes a huge amount of time. Moreover, by each kernel invocation a grid of 8x8 blocks and each block of dimension 16x16 threads were created, and that already produces a big amount of threads that can be active during the computation stage, which means no longer necessary to load more elements. Beside that, the number of threads in a block is big and a multiple of the warp size, and that may ensure an optimal efficiency.

Afterwards, I tried problem Case 6 (Table 4.2), where we have a smaller matrix size and less number of layers. Here it is not preferable to use the same number of threads per block because then a grid of only 16 blocks would be created, and in this way almost half of the MPs would be idle during the execution stage and that means not the entire GPU potential would be exploited. Thus, it is better to reduce the number of threads per block to increase the number of blocks per grid. So, a configuration of 8x8 threads per block increases the number of blocks to 64 blocks.

Thus, parallelizing the transpose/multiply computation of all slices for these cases did not pay off much because the GPU has already reached the point where it runs everything sequentially. That may only save the cost of kernels invokation and setting up the grids and waiting for scheduling the new threads ...etc. For instance, after parallelizing the transpose kernels for Case 6, it is about 1.5x faster, and that leads to only ≈15% overall improvement.

On the other hand, in the problem Case 1 to 5, the matrix size is not big so parallelizing the slices turns into considerable performance gain. But, it must be noted that the execution configuration must be chosen very carefully. For instance, Case 1 is only 16x16 matrix size, and before batching, only one matrix is processed at a time, so with this case if we choose a block size of 16x16 threads then this imposes only one active block per MP. This is not efficient since all the threads will be in a one block and then that will be spilt into 8 warps running on only one MP of 8 cores (as explained in section 2.5), and in this way, all the rest of the cores will be idle. From NVIDIA, *"running only one block per multiprocessor will force the multiprocessor to be idle during thread synchronisation and also during device memory reads if there are not enough threads per block to cover the load latency. It is therefore better to allow for two or more blocks to be active on each multiprocessor to allow overlap between blocks that wait and blocks that can run."*.

If we consider to reduce the wasted parallelism or idle MPs, we may decrease the number of threads per block to 2x2 to increase the number of blocks per grid to 64 blocks, but

then this would introduce another problem, where each block would create one warp of size 4 threads. And that is another issue as explained earlier in section 2.5. (*"for an optimal efficiency the number of threads in a block must be a multiple of the warp size (32 threads on Tesla C1060 GPUs)"*). Perhaps a block of dimension 4x2 threads shows a roughly better performane, but then this would decrease the number of blocks to 32 blocks per grid and the warp size is still too small.

This wastes much of the availabe parallelism, and with the small transform sizes, the waste for these kernels can be quite high. Beside, the thread instructions are executed sequentially, so executing more warps is the only way to keep the hardware busy.

This bring us to the first optimization technique, where we can boost the utilization of the GPU by maximizing the blocks sizes as well as increasing the number of thread blocks to ensure that more threads are active during the computation stage, which leads to making the GPU fully loaded. Further details about these solutions can be found in the next section.

5.3.1. Optimizing The Execution Configuration

To achieve maximum performance one should pay attention to the number of active warps, so the block and grid dimensions must be chosen optimally to yield a large number of concurrent threads during the computation stage, i.e. choosing threads per block as a multiple of warp size so as to avoid wasting computation on under-populated warps. Furthermore, more threads per block can provide a better memory latency hiding. An obvious good choice can be 64 threads per block, but that will be good only if there are multiple concurrent blocks: 128 to 256 threads are better choices.

I started optimizing from problem case 1 when the size of matrices is 16x16 and the number of slices is 51. The first optimization idea for batching all the matrices was launching one kernel that creates one grid of size 8x8 of three dimensional blocks, where the third dimension reflects the number of matrices, so each block of dimension 2x2x51 as shown in Figure 5.9. This allows more elements to be loaded for processing in each thread block.

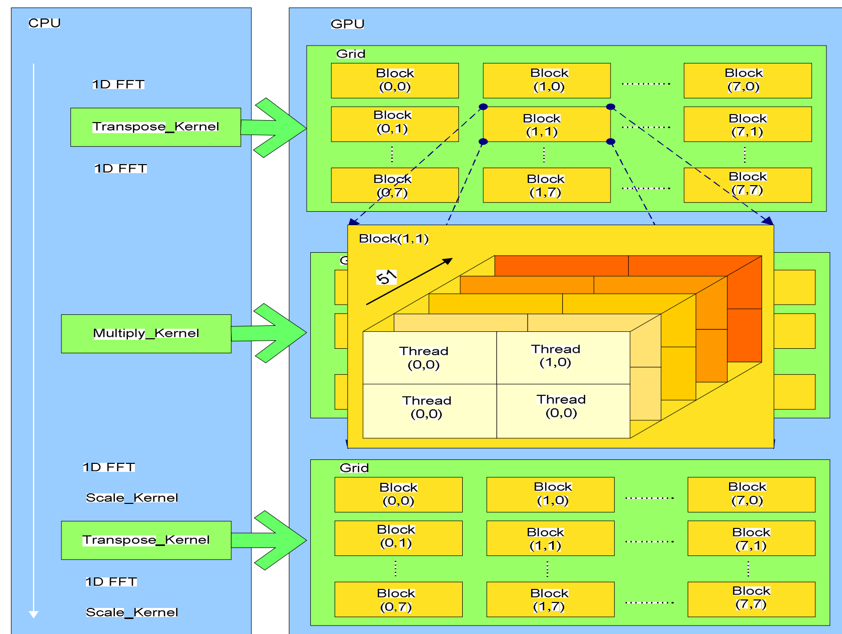


Figure 5.9: CUDA execution representation of applying 3D blocks scheme; Each kernel establishes a layout of 51 matrices of size 16x16 spreaded onto a grid of 3D blocks, where # of matrices are laid out on z dim

A block of size 4x2x51 shows a slightly better performance. This configuration leads to 32 blocks with 408 threads which each split into 12 warps of 32 threads and one of size 24. Figure 5.10 shows the percentage of GPU computation time, regarding problem Case 1. As we see in this figure, this technique reduces the percentage of transpose/multiply kernels from about 95%, to about 30% and 25% in the second and third columns, respectively. To watch the behavior of this technique for all problem cases see Figure 6.2.

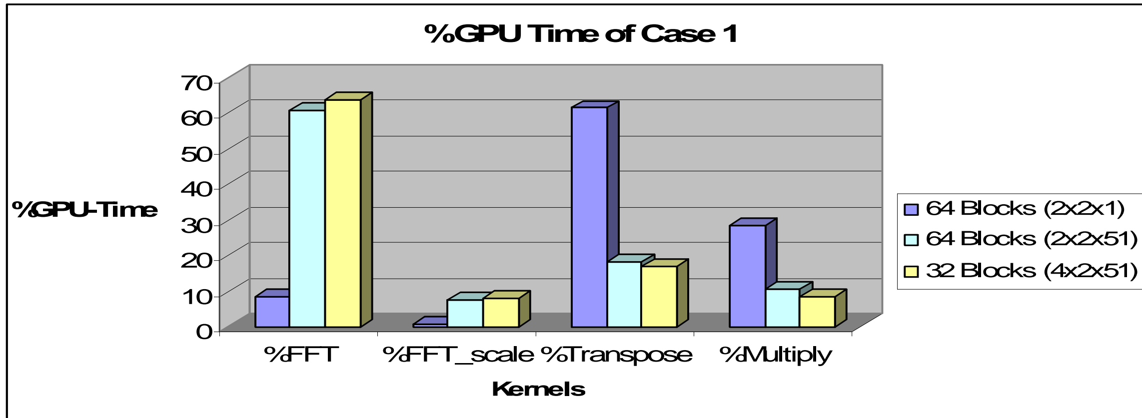


Figure 5.10: GPU execution time overview of Problem Case 1, after applying 3D blocks scheme

Although this pattern considerably improved the performance, especially for problem Case 1, it introduces a quite limited speedup with the other problem cases. That is because as we discussed before the optimal block size is preferred to be multiple of 32. Moreover, the number of blocks will be too small which produces a lower number of active blocks run on a MP, so this does not allow a full occupancy of the multiprocessors. Beside that, it does not suit the other problem cases since the third dimension of a block is limited to 64 threads only. This could be solved by laying out the number of matrices on the first or the second dimensions, which are limited to 512 threads. For instance, with problem Case 2, where we have 75 matrices, a grid of 8x8 blocks with 2x75x2 threads per each can be used. This implementation is still not very effective due to all these features. Now we propose another scheme.

As we look back, we notice how varying the CUDA execution configuration affects the performance. This brought us to propose a new scheme, where (as illustrated in Figure 5.11) all the elements per matrix can be located in one block of size 16x16, and hence we may have *3*(NLayers+1)* blocks run in parallel. The remarkable advantage of this pattern is that there is a degree of scalability available, where we have the ablility to request an execution configuration to whatever number of matrices processed by a grid; and thus, we can scale up the number of layers needed and the number of angles (as we will see later in Section 5.4), quite flexibly using this technique, and that leads to significant performance improvements since more layers and/or angles means more blocks running in parallel. It does not gain anything with problem Case 1 in comparing to the previous technique and may seem like a waste of parallelism since we could have only 51 blocks, but this is of little importance when the data processed by a single thread block is large enough and a multiple of 32, which leads to improve the performance utilization. However, the real benefit is seen when the size of the matrices increases. So, this technique shows a better performance with problem Case 2 and 3 (as we will see in Section 6.2, Table 6.1).

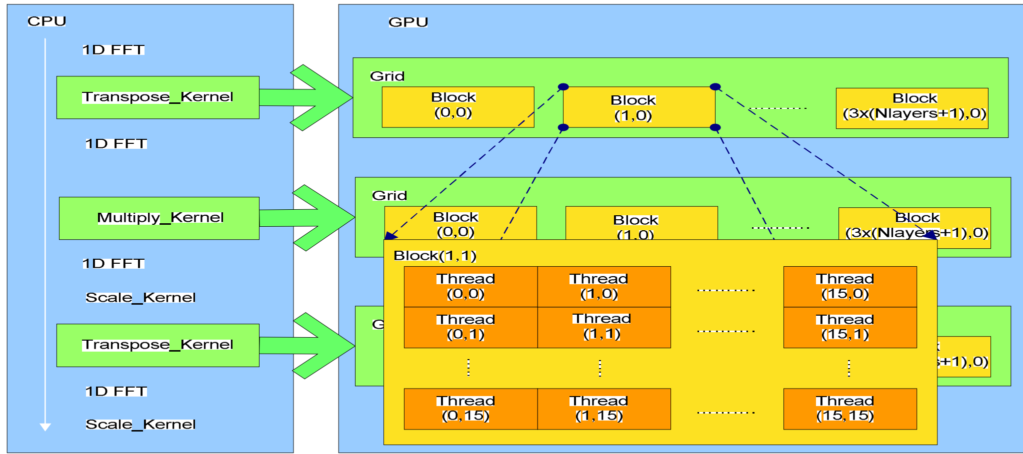


Figure 5.11: CUDA execution representation for Problem Case 1, 2 and 3 by applying 1D Grids of 2D Blocks scheme; Each kernel establishes a layout of (3x(NLayers+1)) matrices of size 16x16 laid out on horizontal row of blocks

On the other hand, with problem Case 4 and 5 the matrix is of size 32x32, which does not fit in one block. Nevertheless, we may have a 32x32 array region if we use a horizontal row of threads with the same width as the matrix we are processing, but spreaded vertically through four blocks with only 32x8 threads per each. So, each 4 blocks in *y* direction handles one matrix while the *x* dimension still can be used to run all the other *3*(NLayers+1)* matrices as demonstrated in Figure 5.12.

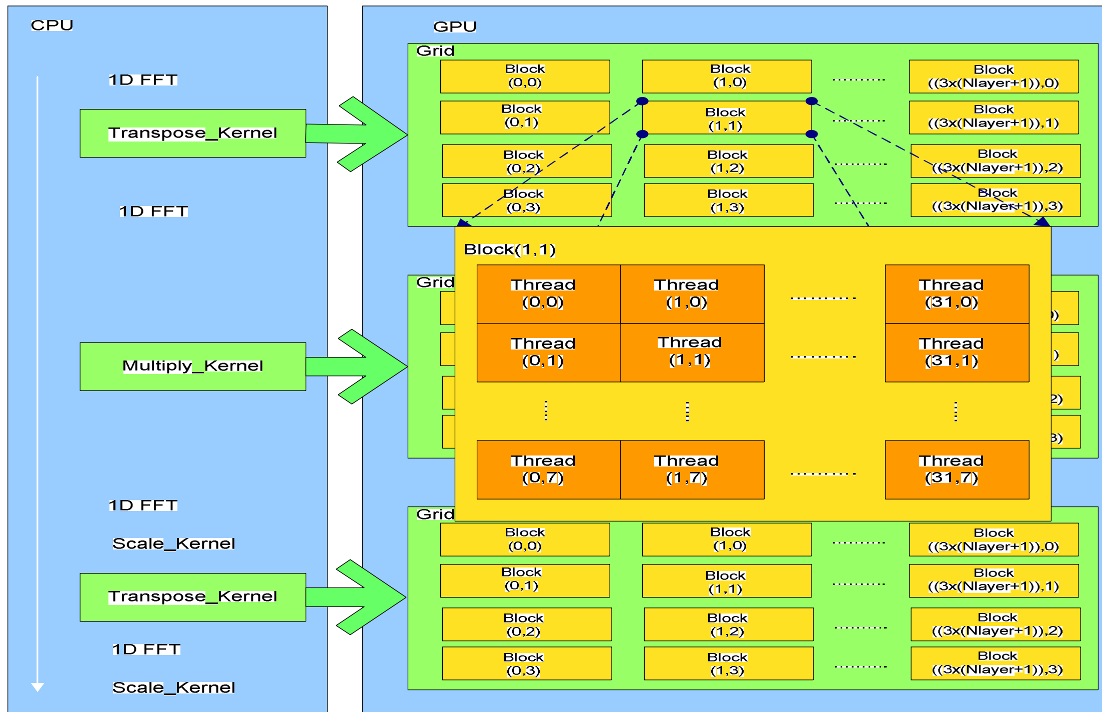


Figure 5.12: CUDA execution representation for Problem Case 4 and 5 by applying 1D Grids of 2D Blocks scheme; Each kernel establishes a layout of (3x(NLayers+1)) matrices of size 32x32, where each matrix is spread vertically on 4 blocks, whereas the number of matrices is laid out onto the horizontal row of blocks

Thus, we can increase the matrix size (the transform size) more flexibly using this approach. Therefore, that can be also used for all other problem cases, namely 5, 6 and 7. A BlockSize of (FFT_Width, 256/FFT_Height) and a GridSize of ((3*(NLayers+1), $(FFT_Height)^2/256$) should be considered in this pattern.

To have a simpler structure with square threads blocks (i.e. 16x16), we may use a square array of blocks instead of using an array of vertically spreaded blocks; and that can be done by distributing the threads horizontally and vertically on a layout of square arrays of blocks with 16x16 threads per each. As a general configuration, one can use a BlockSize of (16,16,1) and a GridSize of ((FFT_Width /16)*(3*(NLayers+1), (FFT_ Height /16)) with this pattern. See Figure 5.13.

Due to these facts, we may use this pattern as a general pattern as long as the transform size is a multiple of 256, otherwise a different grid configuration should be set up. This strategy enables a great flexibility in the implementation of EDM algorithm with variety of geometry sizes and number of angles of incidence (See section 5.4). In the next section we take advantage of on-chip shared memory to see if further significant performance optimization can be gained.

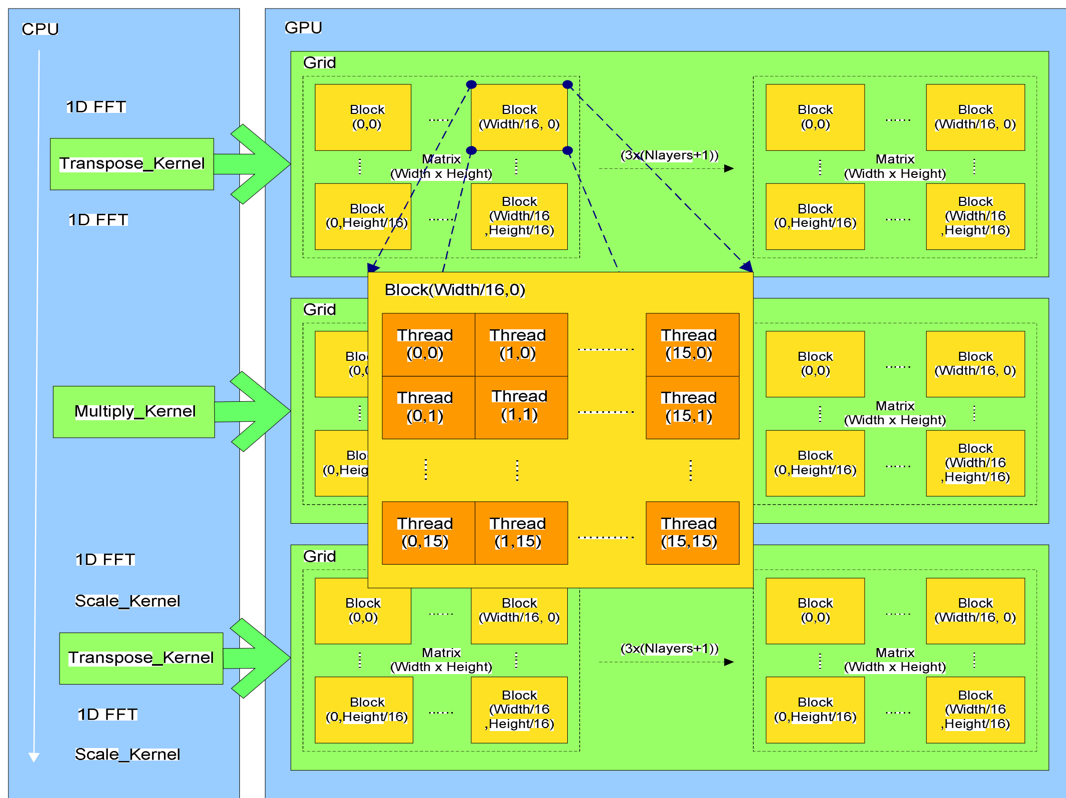


Figure 5.13: CUDA execution representation of applying general scheme; Each kernel establishes a layout of (3x(NLayers+1)) matrices of size HeightxWidth, where each matrix is spread on a square array of thread blocks of size 16x16, whereas the number of matrices is laid out onto the horizontal row of arrays of blocks

5.3.2. Using The Shared On-Chip Memory

As we saw in the last sections, configuring the dimensions of the grids and thread blocks more efficiently could considerably enhance the application performance. In addition,

more optimization may be achieved by maximizing the use of the on-chip shared memory, since it is almost not being used at all, and as explained in Chapter 2, its accesses are almost as fast as register accesses as long as there are no bank conflicts between the threads.

On the other hand, the shared memory is small. Therefore, in the beginning, we need to make sure whether it is sufficient to use. For instance, for the smallest problem Cases 1, 2, and 3, we need to have only a layout of size 16x16 in the shared memory, to store a matrix of size 16x16 float complex values, which yields 2KB per block (256 points*(8B needed for each float complex point)). Since the available shared memory is 16KB, it allows allocating up to 8 matrices. In problem Case 4 and 5, the matrix size is 32x32, and that means we need 32x32 region in the shared memory, that leads to 8KB (1024 points *8B) needed to store one matrix, which means is space can be allocated for 2 matrices. In short, the available shared memory per MP is quite enough for these 5 cases. However, it is not sufficient for the other problem cases. Table 5.3 shows in which problem cases EDM application is applicable to shared memory-based approaches.

Problem Cases	FFT Size	FFT points	Storage (Single Precision)	Availability on Shared Memory?	Storage (Double Precision)	Availability on Shared Memory?
1	16x16	256	2KByte	Yes	4KByte	Yes
2	16x16	256	2KByte	Yes	4KByte	Yes
3	16x16	256	2KByte	Yes	4KByte	Yes
4	32x32	1,024	8KByte	Yes	16Kbyte	No
5	32x32	1,024	8KByte	Yes	16Kbyte	No
6	64x64	4,096	32KByte	No	128Kbyte	No
7	128x128	16,384	128KByte	No	256Kbyte	No

Table 5.3: Amount of Storage needed for each Computations Case, and the Availability of the Shared Memory per MP

By each CUDA kernel invokation the handling time is separated into two sub stages within corresponding kernels. By the transpose kernel, the first stage loads the data from global memory into shared memory and saves it in a row-wise order, and the second stage performs the required operation which in this case reads the data from the shared memory in column-wise order and writes the results back to global memory. The multiply kernel, in the first stage, loads the data from global memory into registers and computes element products and then saves the results into shared memory, and the second stage writes the results back to global memory. Here loading and storing the data into the shared memory are done in row-wise order). It must be nothed that, probably more benefit can be gained by performing the element product on the shared memory rather than using the math library available in CUDA.

In between the two stages a __syncthreads( ) must be called to guaranty that all threads have written to shared memory before any other processing begins. This is necessary to avoid the RAW hazards because threads are dependent on data loaded by other threads (See page 26 in [2]).

If one thread is assigned to load only one element into shared memory, then the threads loading the elements will be idle during waiting for the other threads in the block to synchronize. Thus, another technique can be adopted to optimize the transpose and multiply kernels as in [29]. This technique assumes that the time wasted for threads synchronization, and/or for memory reads, can be exploited through making each single thread perform the work of multiple threads sequentially, by adding sequential loops in the kernels. The threads indices

are defined by adding an appropriate offset to them. This allows more elements to be loaded for processing in each thread block. According to Nvidia, CUDA kernels compiled in this way exhibit excellent performance and scalability, since the dimensions of the threads blocks and the grids are limited, 512 and 65536 respectively. Further attempts to optimize the performance of transpose kernels can be found in [29].

Although, the results of these techniques, from NVIDIA, show considerable performance speedup, it was not implemented in this thesis. The transpose kernel does not take high percentage, thus it was not worthwhile in terms of the performance gain/the programming efforts.

Furthermore, more optimizations could be achieved by maximizing the independent parallelism of the application. Thus, more parallelism might be found by running multiple angles of incidence concurrently, which are currently computed on a cluster of CPUs as illustrated in Figure 5.14 (a). Next section will discuss this in more details.

5.4. Multiple Angles Parallel Implementation

Another interesting parameter that can exploit the GPU parallelization is the number of angles of incidence that is required to compute the angular-resolved spectrum. The angles can actually be processed independently and in parallel. As the number of angles in the spectrum increases, more elements can be explored in parallel, which may yield an increase in performance especially with small problem cases.

Basically, computing the matrix-vector product of multiple angles can be calculated in the same way as for one angle, but only for a bigger data set. See Eq. 5.1, where the product between the matrix $\boldsymbol{A_1}$ and the vector $\boldsymbol{x_1}$, the matrix $\boldsymbol{A_2}$ and the vector $\boldsymbol{x_2}$, and the matrix $\boldsymbol{A_n}$ and the vector $\boldsymbol{x_n}$, represent the matrix-vector product of the first angle, the second angle and nth angle respectively.

$$\begin{bmatrix} [A_1] & 0 & 0 & 0 \\ 0 & [A_2] & 0 & 0 \\ 0 & 0 & \ddots & \vdots \\ 0 & 0 & \cdots & [A_n] \end{bmatrix} \bullet \begin{bmatrix} [x_1] \\ [x_2] \\ \vdots \\ [x_n] \end{bmatrix} = \begin{bmatrix} [b_1] \\ [b_2] \\ \vdots \\ [b_n] \end{bmatrix} \qquad (5.1)$$

This can be independently computed in parallel within one sequential linear solver, as depicted in Figure 5.14 (b), in each solving step. But, this requires first a paradigm shift of the serial code. In first place, all the data arrays were maximized by an extra dimension to reflect the different number of angles as shown in Algorithm 5.7. Afterwards, a great number of elements belong to different angles were grouped in a set, as a one vector, and offloaded to the GPU in once; see Figure 5.14 (c).

- Algorithm 5.7: Pseudo Code of the 2D convolution (of multiple angles version):

```
o conv(3x(Nlayers+1)x mfft x mfft x Nangles)=0;
o conv(:)=elecfield(:, :, :, :,:); // copying 5D array into 1D array
o gpu_Ptr(3x(Nlayers+1)x mfft x mfft x Nangles)=conv(:);
o 1D FFT(gpu_Ptr(:));                                  }
o Transpose<<<gridDim, blockDim>>>(gpu_Ptr(:));        } //Forward 2D FFT
o 1D FFT(gpu_Ptr(:));                                  }
o Multiply<<<gridDim, blockDim>>> (gpu_Ptr(:), shapefunc(0:mfft,0:mfft));
```

- 1D FFT(gpu_Ptr(:));
- scale(gpu_Ptr);
- Transpose<<<gridDim, blockDim>>> (gpu_Ptr(:));
- 1D FFT(gpu_Ptr(:));
- scale(gpu_Ptr);

//Inverse 2D FFT

- conv(:)=gpu_Ptr(3x(Nlayers+1)x mfft x mfft x Nangles);
- Edev(:, :, :, :,:)=conv(:); // copying 1D array into 5D array

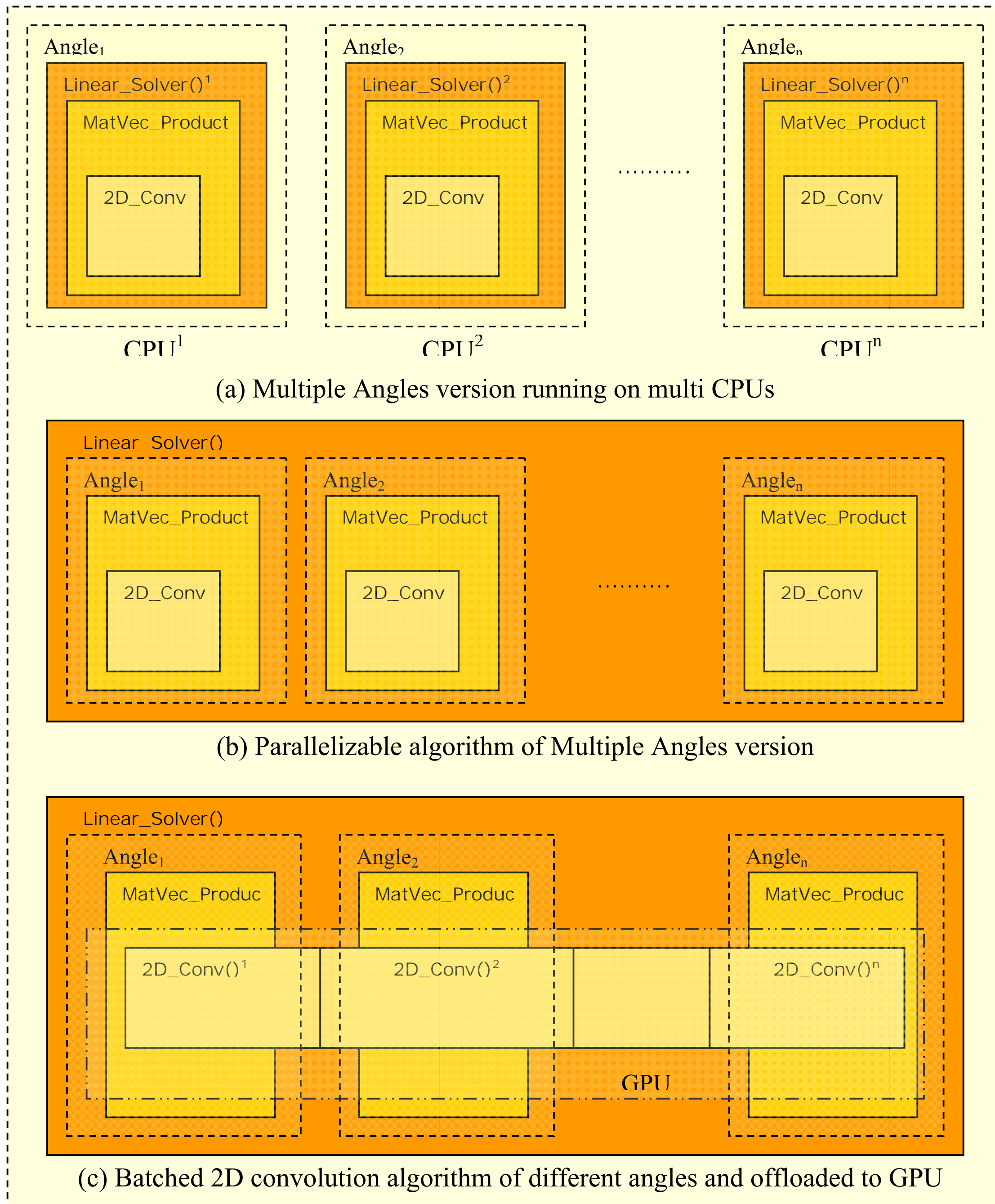


Figure 5.14 Schematic Overview of Multiple Angles Application version for different Angles of Incident running on CPU and CPU-GPU systems

The GPU-based implementation of this method faced all the aforementioned optimization phases, starting from 2D FFT-Based approach to the last optimization technique, and it was applied for all problem cases. In first place, increasing the number of angles meant only extra sequential outer loops on the GPU. Afterwards, when started using the threads structure shown in Figure 5.11— with problem Cases 1, 2 and 3, it began to be included in the *y* dimension of grid, as spreaded vertically through the grid as illustrated in Figure 5.15. As we see in this figure, the horizontal row of blocks reflects the number of matrices and the vertical direction reflects the number of angles.

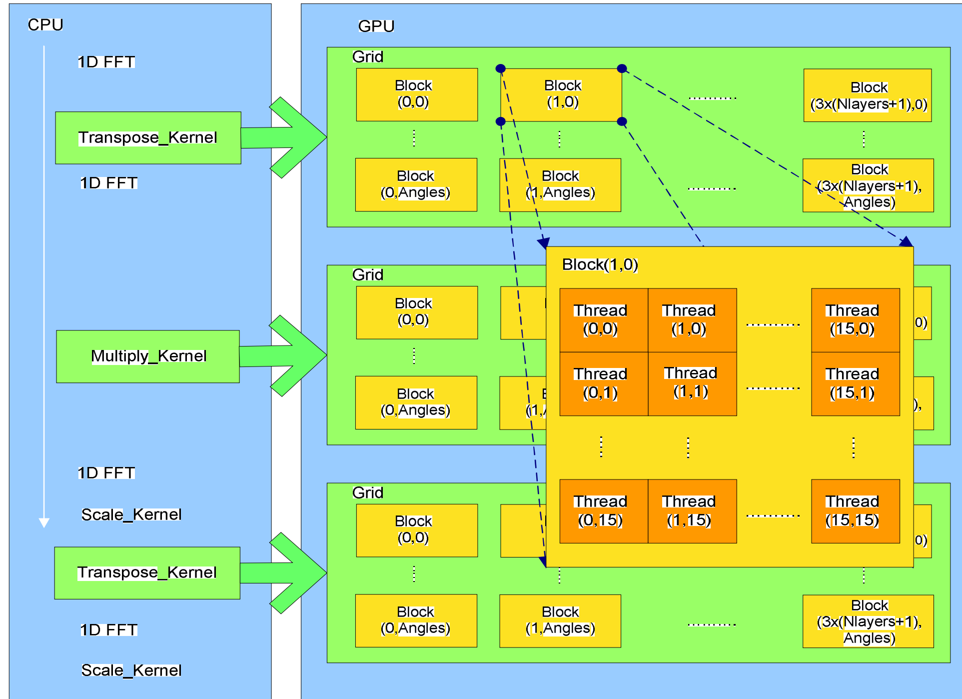


Figure 5.15: CUDA execution representation of Multiple Angles version by applying general scheme; Each kernel establishes a layout of (3x(NLayers+1)) matrices of size16x16, for Nangles; where the number of matrices is laid out onto the horizontal row of blocks, whereas the number of angles is laid out onto the vertical row of blocks

As a general pattern, we may use the threads structure demonstrated in Figure 5.16, where a square array of blocks is repeated horizontally as to represent the number of matrices and vertically to represent the number of angles.

This optimization needed a lot of programming efforts, since it required first a paradigm shift of the serial code. However, on the other hand, it yields a significant amount of parallelism which delivers decent performance improvements. The timing results show a speed increase up to a factor 2 compared to running only one angle at a time.

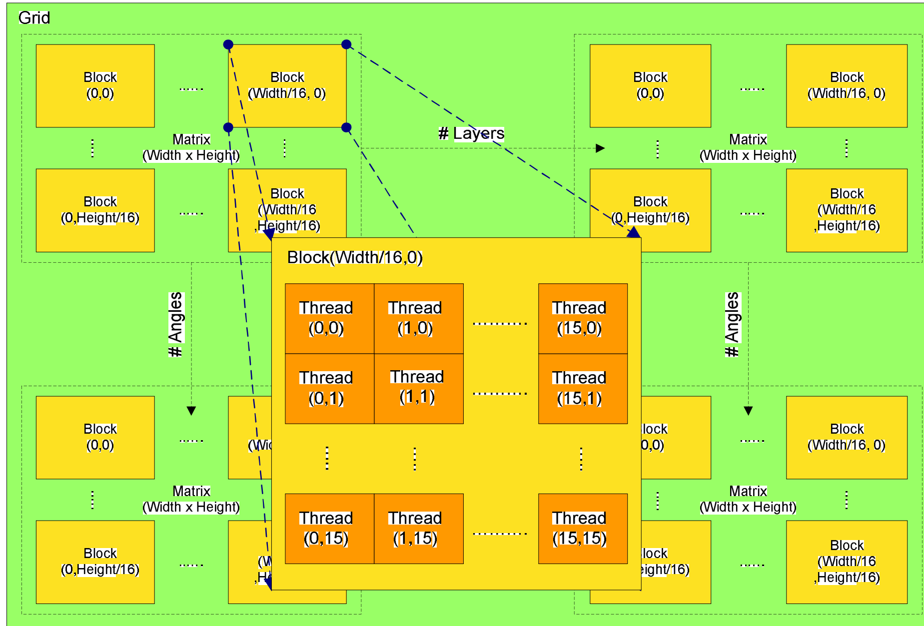


Figure 5.16: A layout of 3x(NLayers+1) matrices of size HeightxWidth for N angles; where # matrices is laid out onto the horizontal row of arrays of blocks, whereas # angles is laid out onto the vertical row of arrays of blocks

5.5. Summary

As we look back, this chapter dealt with the explanation of key points of accelerating the 2D convolution algorithm on the GPU by parallelizing the FFT computations as well as the Transpose/Multiply kernels. It, briefly, covers the sequential algorithm of the 2D convolution (Algorithm 5.1), using NVIDIA CUFFT to perform only the FFT computations in parallel on GPU, moving more kernels to the GPU (Algorithm 5.2), grouping the data transfers back and forth to the GPU (Algorithm 5.3), batching NVIDIA 2D FFT computations by using NVIDIA 1D FFTs (Algorithm 5.4), stripping out some transposes and merging the Multiply kernel (Algorithm 5.5), batching Transpose/Multiply kernels by using optimal execution configurations for various problem sizes (Algorithm 5.6), and using the GPU shared memory.

Another aspect focused in this chapter was how to benefit from parallelizing multiple independent workloads of the application, i.e. performing multiple angles of incidence simultaneously (Algorithm 5.7).

This chapter elaborates that straightforward implementations do not exhibit performance improvements and that alternative parallelizable algorithms must be adopted in order to exploit the massive potential of the GPU. It also shows that in the design of new algorithms, a few essential aspects must to be taken into account in order to benefit from the abundance of parallelism available on the GPU architecture:

- Maximizing the independent operations to be able to batch more computations at once.
- Reducing unnecessary memory copies.
- Using optimal execution configuration.
- Key memory access patterns.

This will extend very well to general 2D convolution kernels and exhibit significant reductions in computation time. Therefore, given an efficient implementation, 2D convolution computations based on optimized 1D FFT-based approch can perform much faster than a straightforward implementation for this application.

Table 5.4 compares the performance gained after applying all the parallelization techniques, mentioned earlier, on the first trivial GPU-based implementation (Algorithm 5.2). All these techniques yield more modest but worthwhile gain.

	Problem Cases						
	1	2	3	4	5	6	7
Execution Time of the First Implementation (msec) (Algorithm 5.2)	4.148	6.163	8.13	8.558	16.838	18.119	57.9
Execution Time of the Final Implementation (msec) (Algorithm 5.6 + using SM)	0.133	0.162	0.203	0.537	0.98	5.118	18.6
Speedup	31.19x	38.04x	40.05x	15.94x	17.18x	3.54x	3.11x

Table 5.4: Overview of the speedup achieved by applying all the Parallelization Techniques versus the Trivial Straightforward Implementations, on GPU. SM: shared Memory.

Although the timing measurements reveal that Algorithm 5.6 is fast and more efficient, but it suits only this application and can not be a generic algorithm. Thus, as to benefit from the batched 2D FFT approach developed in this study for general usage, one should apply all the optimization techniques appear in Section 5.3 on Algorithm 5.4 instead of Algorithm 5.5, in particular before leaving out two transposes.

Furthermore, it might also be useful to develop a new FFT algorithm in order to add more flexibility to merge the kernels, in turn allows more computations per one memory access (per element), to benefit even more from the fast shared memory, decreasing global memory access. The work done in [33] presents GPU-based parallel algorithm for computing FFT algorithm. It indicates that the proposed algorithm efficiently exploits the shared memory and reduces the memory accesses by combining the transposes operations with the FFT computations. The results show a performance gain of up to 4x over optimized GPU-based FFT algorithm using NVIDIA CUFFT on NVIDIA GTX280 GPU, and 8-40x improvements over CPU-based FFT algorithm using MKL (Intel's FFT library) on high-end quad core CPU, for large data sizes and for single floating point precision.

Another modest optimization would be reducing the time taken by the FFT normalization by merging the two normalization kernels in one kernel call and hence scaling by the total FFT size of 2D array instead of 1D at a time. Alternatively, one may merge the normalization kernels with the point-wise multiplication. The latter way performs better since this will provide much memory accesses reductions. In spite of that, the overall performance gain will not be noticeably affected since these computations take only a small percentage of the total execution time.

The important points of the implementation have been explained in this chapter, and then benchmarks and experimental studies on parallel solutions will be described, analyzed and compared with the CPU in Chapter 6.

6 Experimental Results Analysis

Throughout this thesis, different implementations and configurations have been experimented on GPU for several problem cases linked to the application (Table 4.2). This chapter analyzes and evaluates the experimental results of the GPU-based parallel implementations of single and multiple angles versions of EDM application and compares them to the CPU-based serial implementations. It must be noted that all measurements were considered in single floating point precision.

Lots of details will be involved in this analysis ranging from computing the direct/inverse FFT and scalar products, to memory transfers between CPU and GPU. These parts of the algorithm have been also benchmarked separately using the CUDA visual profiler. To remove the CUDA startup and data transferring overhead from performance measurements, the time considered for the comparisons was only the computation time spent in processing the kernel programs. So, the time taken to read data array from CPU memory, transfers to the GPU, is excluded as well as without uploading it back to the CPU. This gives the time taken by the GPU compared to the time taken by the CPU to compute the same portion of the algorithm. Furthermore, the time consumed by transferring the data back and forth to the GPU can be neglected by overlapping it with the computation time, which means "Concurrent copy and execution" [29], by using pinned (locked-paged) memory (See page 30 in [2]).

Section 6.1 shows the test setup used for the measurements. The next sections demonstrate the timing results of the framework and the performance behavior of the parallel implementation for different optimization versions of running one angle of incidence, as well as running multiple angles concurrently on GPU in comparison with using a cluster of CPUs.

6.1. Measurements Setup

The host code is written in C, whereas the device code is written in CUDA version 2.3, which is available for download at http://ww.nvidia.come/cuda/. The C Code was compiled with GNU's GCC compiler while CUDA code was compiled with *nvcc* [39].

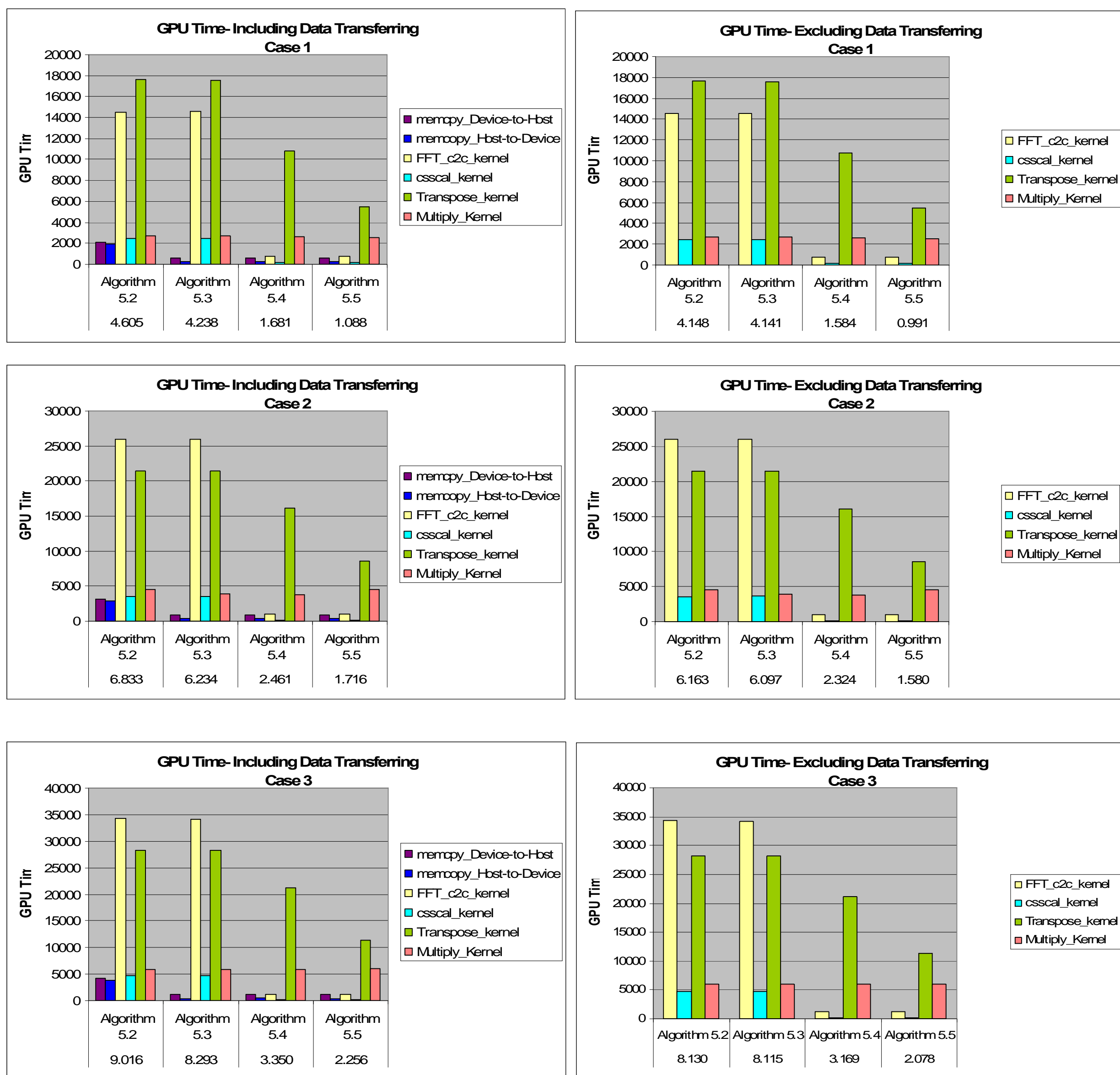
The computer's processor used for benchmark was Intel Quad Core Xeon E5450, 3.0 GHz, with 2GB of DRAM memory. The environment for the experiments has been a Linux distribution, namely Ubuntu, which is an open source Linux-based operating system available for download at http://www.ubuntu.com/. The CUDA computing architecture used for the experiments was NVIDIA Tesla C1060, with a Host-to-Device pageable memory bandwidth 1.832 GByte/sec, Device-to-Host pageable memory bandwidth 1.235 GByte/sec, and 73.657 GByte/sec of Device-to-Device bandwidth.

To measure the time elapsed on the CPU and the GPU, a heavy use of timers have been exploited. The system function *gettimeofday()* was used to measure the time taken by the CPU, which is declared in *sys/time.h*, and another timer readily available on CUDA architecture has also been used, such as *cutCreateTimer()* and *cutStartTimer()* from *cutil* library. As well as some important debug information retrieved in order to correct GPU programs.

6.2. Performance Analysis of The Parallel Implementation

The total time spent by the 2D convolution computations is dominated by the direct/inverse FFT calculations. This means the effects of optimizing these portions would be clearly visible since it covers most part of the execution time of the algorithm. Furthermore, the matrix transposing and multiplying kernels were other bottlenecks for this algorithm for some cases. Therefore this has received also special focus in benchmarking in this chapter. Main differences in optimization techniques, besides batching the computations of FFTs, are related to grid and threads block sizes as well as GPU memory model.

Figure 6.1 illustrates how the time is distributed between the kernels of the different implementations described in Chapter 5, for the variant computational problem cases. Note that the *x*-axis shows the time taken by each algorithm per one 2D convolution call (the time is in msec), whereas the *y*-axis represents the time taken by each kernel times the number of matrix-vector product calls, (the time is in usec).



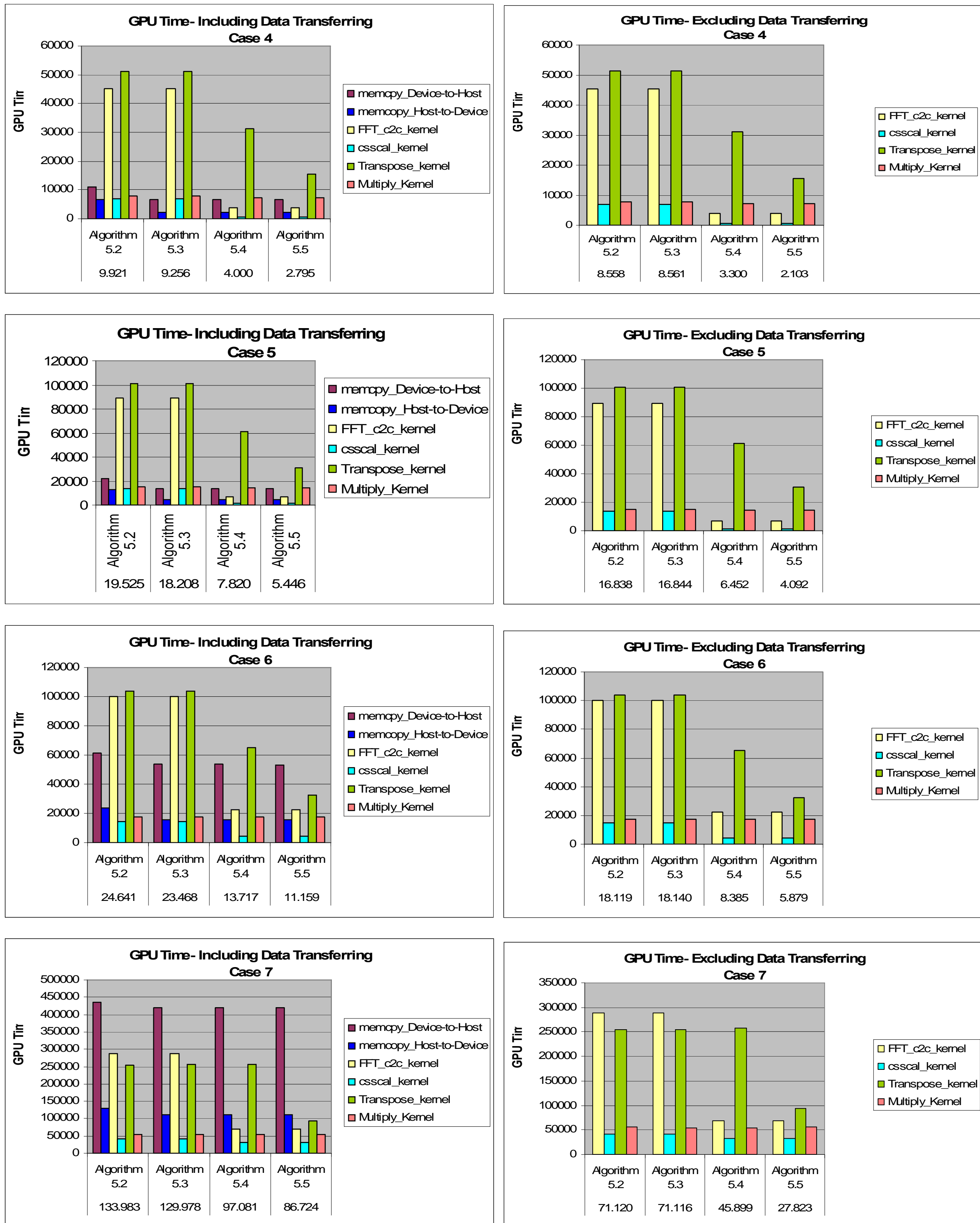


Figure 6.1: Performance Overview of the all problem cases using different algorithms

It should not come as a surprise that Algorithm 5.2 and 5.3 takes much more time than Algorithm 5.4 and 5.5. This is due to the fact that 2D FFT computations dominate. These

algorithms are built based on 2D FFT-based approach, which did not take advantage of executing multiple batches in parallel, which as a result does not fully utilize the parallelism available on the GPU. Thus, as we see the 1D FFT-based approach is the better choice for small and large 2D convolution solutions.

Although batching the computations of the FFTs shows a considerable performance gain, this approach has been constrained from achieving higher performance than CPU-based implementation especially for small problem cases because the parallelism of the GPU was not exploited enough for the other kernels (e.g. Transpose and Multiply kernels). These kernels are launched sequentially for *(3*(Nlayer+1))* times, where each kernel invocation establishes a grid of 64 thread blocks of size (2x2) with Case 1, 2 and 3, and of size (4x4) with Case 4 and 5. A block size of 2x2 threads allow only one warp of size 4 threads to be run in parallel, which as stated in Chapter 5, will force some of the GPU cores to idle. Therefore, streaming more thread blocks in pipeline fashion through the device can overcome this overhead (see Table 6.1).

Cases	# Grids # Blocks per Grid	Block size	Max # of Active Threads per MP	Max # of Active Blocks per MP	Max # of Active Warps per MP	GPU Time of Transpose Kernel (usec)
1.a	2x51 grids(64 Blocks of (2x2) Threads)	4	32	8	8	608.12
1.b	2 grids(51 Blocks of (16x16) Threads)	256	768	3	24	26.59
1.c	2 grids(64 Blocks of (2x2x51) Threads)	204	612	3	~20	25.82
1.d	2 grids(32 Blocks of (4x2x51) Threads)	408	408	1	~13	22.75
2.a	2x75 grids(64 Blocks of (2x2) Threads)	4	32	8	8	954.4
2.b	2 grids(75 Blocks of (16x16) Threads)	256	768	3	24	32.43
2.c	2 grids(64 Blocks of (2x75x2) Threads)	300	600	2	~19	44.62
3.a	2x99 grids(64 Blocks of (2x2) Threads)	4	32	8	8	1260.5
3.b	2 grids(99 Blocks of (16x16) Threads)	256	768	3	24	41.98
3.c	2 grids(64 Blocks of (2x99x2) Threads)	396	396	1	~13	52.42

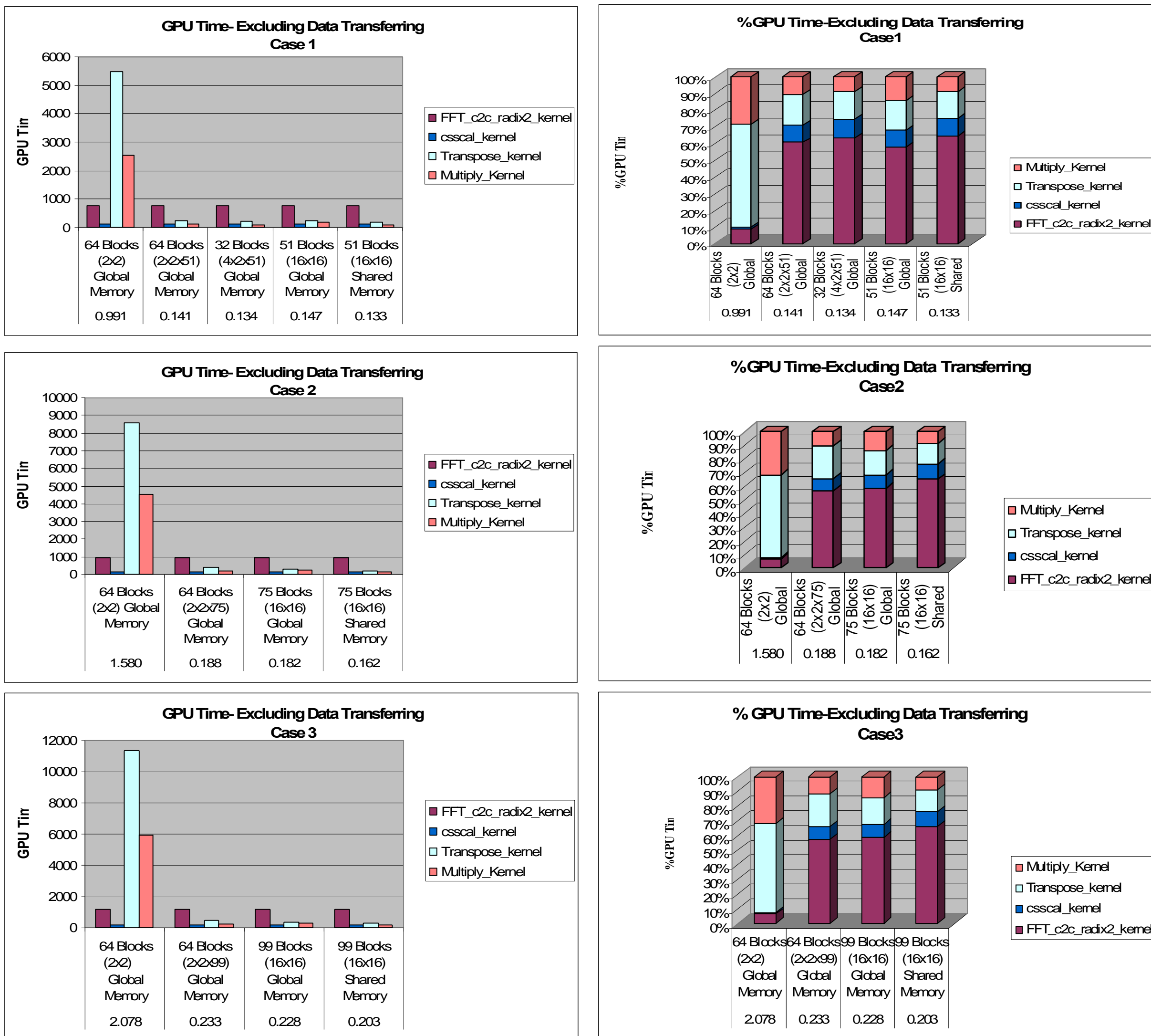
Table 6.1: An overview over the Threads Structure on GPU for Problem Case 1, 2, and 3

As noticed in Table 6.1, varying the execution configuration has a profound effect on the performance of the Transpose kernels. For instance, Case 2.b and 3.b consider the general pattern. As we see in the table, they perform the best because we have more blocks as well as because the block size is a multiple of the warp size and more blocks are active per multiprocessor. Therefore, the general

pattern exhibits a better performance than using 3D blocks scheme for "big enough" problem sizes. Only multiples of the number of warps (32) per multiprocessor and multiples of the number of multiprocessors result in optimal configurations that can fully utilize the GPU.

From Table 6.1, one can conclude that the more threads are executed the faster the execution is. It must be noted, though, that the number of threads is only a parameter passed on kernel invocations and that does not reflect the fact of having more hardware running more threads. It simply states that the thread scheduler will be able to schedule more threads during its time slices.

Figure 6.2 shows the computations time taken by the GPU when applying some improvements, in respect with CUDA execution configuration, on Algorithm 5.5: in particular batching the Transpose and Multiply kernels (as it has been seen in Algorithm 5.6).



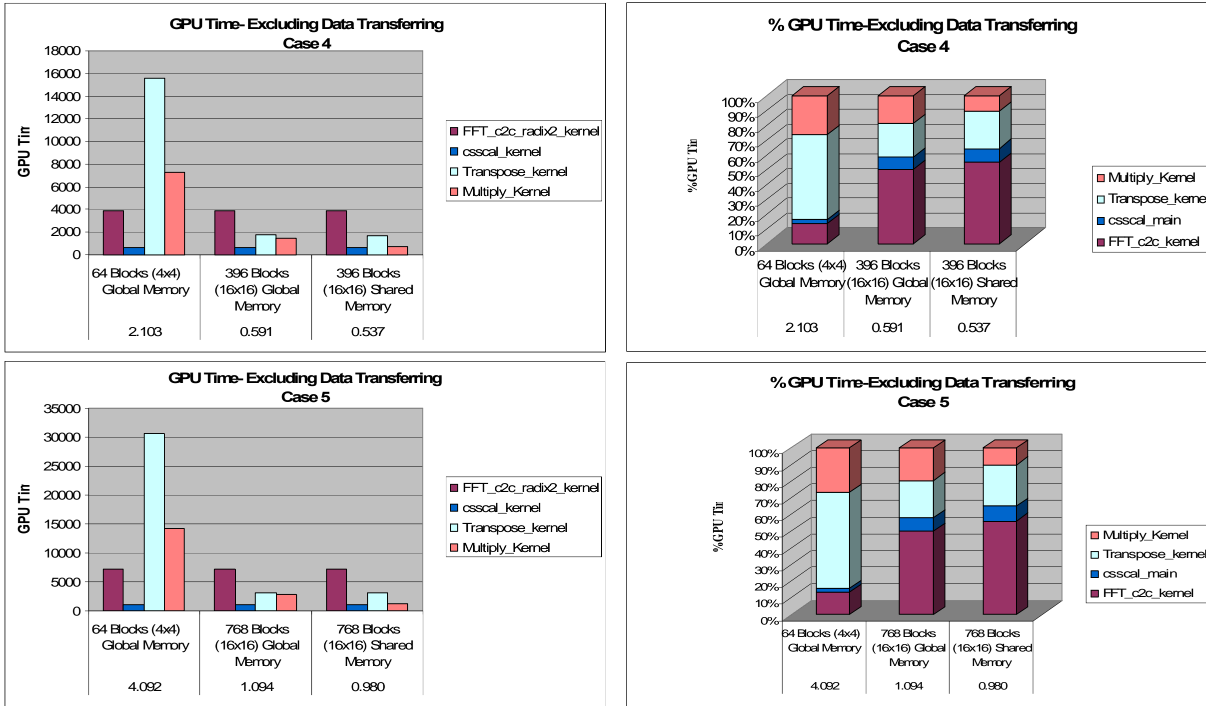


Figure 6.2: Performance Overview of the all problem cases using Algorithm 5.6 with different CUDA execution configurations

It is clear from Figure 6.2 the percentage of the Transpose and Multiply kernels decreases while using more careful execution configuration. With the improved Transpose and Multiply kernels, the FFT becomes the dominant computation. Last column shows the time taken by the GPU when applying the general pattern described in Chapter 5 using the shared memory.

Table 6.2 shows the general overview of the timing results of all implementations for all problem cases, excluding the time taken for transferring the data between the CPU and GPU.

Case	Algorithm 5.2	Algorithm 5.3	Algorithm 5.4	Algorithm 5.5	Algorithm 5.6 3D-Blocks Global-Memory	Algorithm 5.6 General-Pattern Global-Memory	Algorithm 5.6 General-Pattern Shared-Memory
1	4.148	4.141	1.584	0.991	0.141	0.147	0.133
2	6.163	6.097	2.324	1.580	0.188	0.182	0.162
3	8.130	8.115	3.169	2.078	0.233	0.228	0.203
4	8.558	8.561	3.300	2.103	-	0.591	0.537
5	16.838	16.844	6.452	4.092	-	1.094	0.980
6	18.119	18.140	8.385	5.879	-	5.118	-
7	71.120	71.116	45.899	27.823	-	80.208	-

Table 6.2: Timing Results for one Angle of Incidence of different GPU-based Algorithms over a range of Problem Cases; time is msec

From Table 6.2, one can conclude that the performance depends on FFT size (see table 4.2) in a linear way because the per-element computation complexity of the FFT algorithms

grows as the sizes of the Fourier transformation increase.

6.3. Comparison of Single Angle version on GPU versus on one CPU Core

The intended goal during this thesis was to compare how fast a GPU-based approach could be compared to standard CPU solutions. Throughout the research for the thesis, CPU solutions have been used to compare performance and debug and improve the GPU kernels.

In this section, the performance of the different GPU-based implementations described earlier will be compared to their counterpart on CPU. The speedup of these implementations in comparison to the CPU can be found in Table 6.3. The speedup is calculated as follows: CPU Time/GPU Time.

Case	Speedup Algorithm 5.2 vs CPU	Speedup Algorithm 5.3 vs CPU	Speedup Algorithm 5.4 vs CPU	Speedup Algorithm 5.5 vs CPU	Speedup Algorithm 5.6 vs CPU
1	10.37^{-1}	10.35^{-1}	3.96^{-1}	2.47^{-1}	3
2	10.27^{-1}	10.16^{-1}	3.87^{-1}	2.63^{-1}	3.7
3	10.16^{-1}	10.14^{-1}	3.96^{-1}	2.59^{-1}	3.94
4	2.59^{-1}	2.59^{-1}	1	1.56	6.22
5	2.49^{-1}	2.49^{-1}	1.04	1.64	6.88
6	1.76	1.75	4.06	6.4	6.80
7	6.1	6.1	12.1	19.14	19.14

Table 6.3: Speedup of GPU-based Implementations versus CPU-based Implementation; the time is exclusive the data transferring

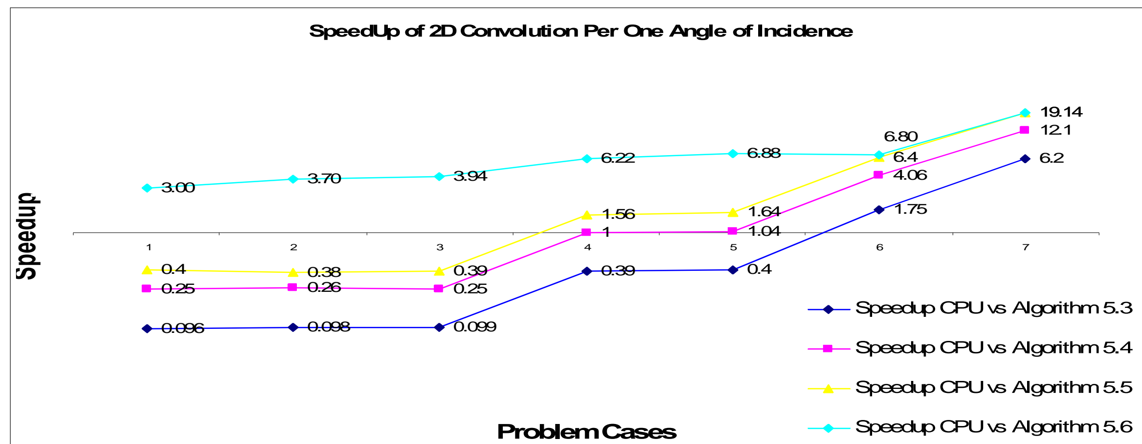


Figure 6.3: Speedup of using different GPU-based algorithms versus their equivalent CPU-based serial algorithm, for the all problem cases; the time is exclusive the data transferring

As we see from Figure 6.3, Algorithm 5.3 shows very poor performance comparing to the CPU-based implementations especially for the small Fourier transform sizes. In spite of using an optimized algorithm, like Algorithm 5.5, to parallelize the FFT computations the CPU outperforms the GPU for the small problem cases (in particular for case 1, 2 and 3), and it is slightly outperformed by the GPU for case 4 and 5. That is because the performance was

limited by the non-optimized execution configurations. But, when more effective configurations were applied to optimize the Transpose/Multiply kernels (Algorithm 5.6), a huge increase in performance has been achieved which outperforms the CPU-based implementations (blue line).

Table 6.4 previews the overall performance and the achieved speedup of the 2D convolution and its estimated influence on the whole EDM application (according to Eq. 4.2) for one angle of incidence. Note that the final shown results consider the best algorithm with optimal configurations of warps per block and blocks per grid only. The speedup is calculated as follows: CPU Time/GPU Time.

	FFT Size (M)	# Matrices (N)	CPU Time (msec)	GPU Time (msec)	SpeedUp Of 2D Convolution	SpeedUp Of whole EDM
1	16x16 = 256	3x(16+1) = 51	0.40	0.133	3.00x	1.5x
2	16x16 = 256	3x(24+1) = 75	0.60	0.162	3.70x	1.57x
3	16x16 = 256	3x(32+1) = 99	0.80	0.203	3.94x	1.59x
4	32x32 = 1024	3x(32+1) = 99	3.30	0.537	6.14x	1.72x
5	32x32 = 1024	3x(64+1) =195	6.75	0.980	6.88x	1.74x
6	64x64 = 4096	3x(64+1) = 195	32.00	4.700	6.8x	1.74x
7	128x128 = 16384	3x(128+1) =387	356.00	18.600	19.14x	1.9x

Table 6.4: General Overview of GPU and CPU execution Times and the Speedup of 2D Convolution Computations on GPU vs CPU and its estimated influence on entire EDM Application

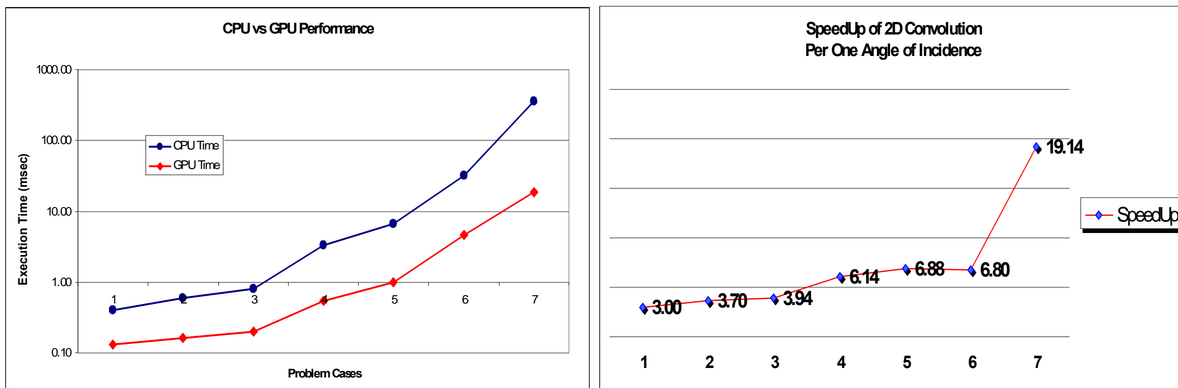


Figure 6.4: Overall Performance of 2D Convolution Computations of single angle version on GPU versus on one CPU core

As we see from the graph in Figure 6.4, the execution time increases proportionally to the number of elements in the convolution. Almost linear speedup has been observed compared to the serial implementation of the algorithm running on one CPU core. The extrapolation of Figure 6.4 indicates that larger Fourier transform presents more parallelism and faster performance on GPUs due to the fact that it executes sequentially on the CPU. From that, one can conclude that more convolutions become more advantageous in terms of performance. Therefore, it was proposed, to process multiple versions of the application for different angles of incidence in parallel, as explained in Section 5.4. Consequently, a cluster of CPUs can be replaced by a smaller system of GPUs; however, the question at hand is whether a cluster of

GPU's will comply with the specification for heat generation and available space while achieving significant performance gain. Next section shows the experimental results and explains this behavior in more details.

6.4. Comparison of Multiple Angles of Incidence version on GPU vs on a Cluster of CPUs

Another performance comparison was carried out between running multiple versions of the application in parallel on the GPU (Algorithm 5.7) with running them on a multi-core CPUs. Figure 6.5 shows that executing more angles yields execution time per an angle being faster than processing one angle at a time. To limit the number of computations, only the first four problem cases were considered.

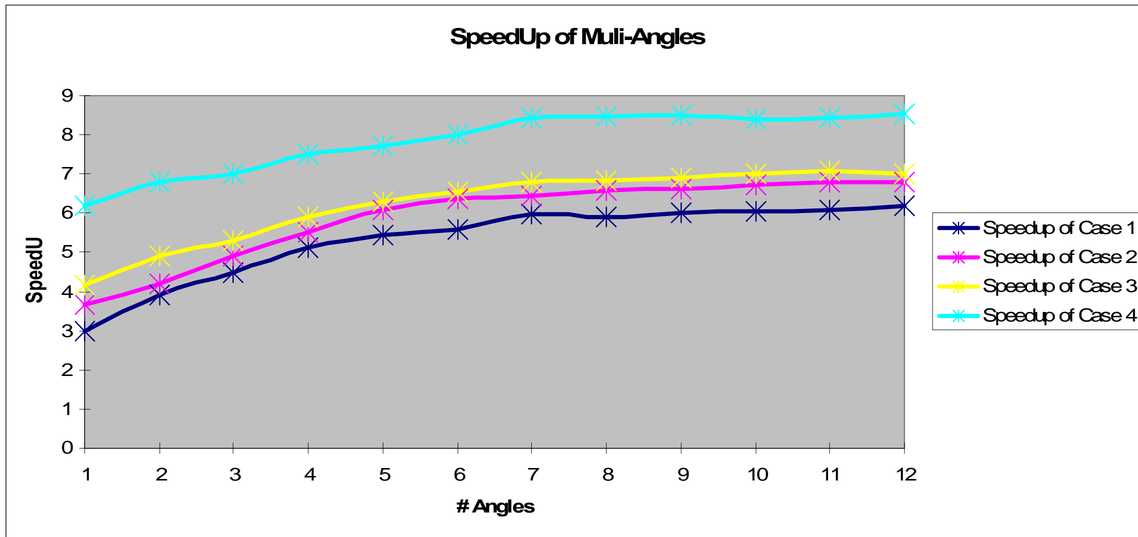


Figure 6.5: Speedup of 2D Convolution Computations of Multiple Angles version per Single Angle on GPU versus on one Isolated CPU

As has been seen, the achieved performance results are so encouraging to consider computations on a cluster of GPUs instead of a cluster of CPUs. However, besides the performance gain, other metrics should be taken into account before, like power consumption, price and space. See Table 6.6.

The power consumption is an important design parameter, since more power consuming results in an increase of the surrounding temperature, which means extra costs for cooling the system. The NVIDIA Tesla C1060 GPU consumes at peak 225 W, which means that for a typical power specification of ~ 4KW, up to 16 GPU Tesla GPUs can be used with one or two hosts. Consequently, to outperform a cluster of state-of-the-art CPU's (Intel Xeon Quad Cores) generating a comparable heatload, each single GPU must minimally have the performance of 7 CPU cores.

Table 6.5 shows an overview about the speedup of the 2D convolution, when running one angle on GPU at a time and running 7 angles on GPU at the same time, in comparison with one angle on the CPU.

From this table, we see that running multiple angles in parallel could speed up the small problem cases with noticeable factor, whereas it does not affect much the bigger cases, where FFT size is 32x32 and higher, because the more threads are run sequentially.

Cases	Problem Cases						
	1	2	3	4	5	6	7
Execution Time of 2D Convolution on one CPU Core	400	600	800	3300	6750	32000	356000
Execution Time of 2D Convolution of one angle	133	162	203	537	980	4700	18600
Speedup of 2D Convolution of one angle	3x	3.7x	3.94x	6.14x	6.88x	6.8x	19.14x
Total Execution Time of 2D Convolution of 7 angles	467	654	826	2735	5281	31600	129670
Execution Time of 2D Convolution per one angle	67	94	118	391	755	4514	17238
Speedup of 2D Convolution per one angle	5.97x	6.39x	6.78x	8.44x	8.94x	7.09x	20.65x
Speedup of 2D Convolution at 7 Angles on GPU vs 1 Angle on CPU	0.87	0.92	0.97	1.2	1.28	1.01	2.74

Table 6.5: The speedup of 2D convolution on GPU in comparison to one Isolated CPU when running one and 7 angles of incidence in parallel.

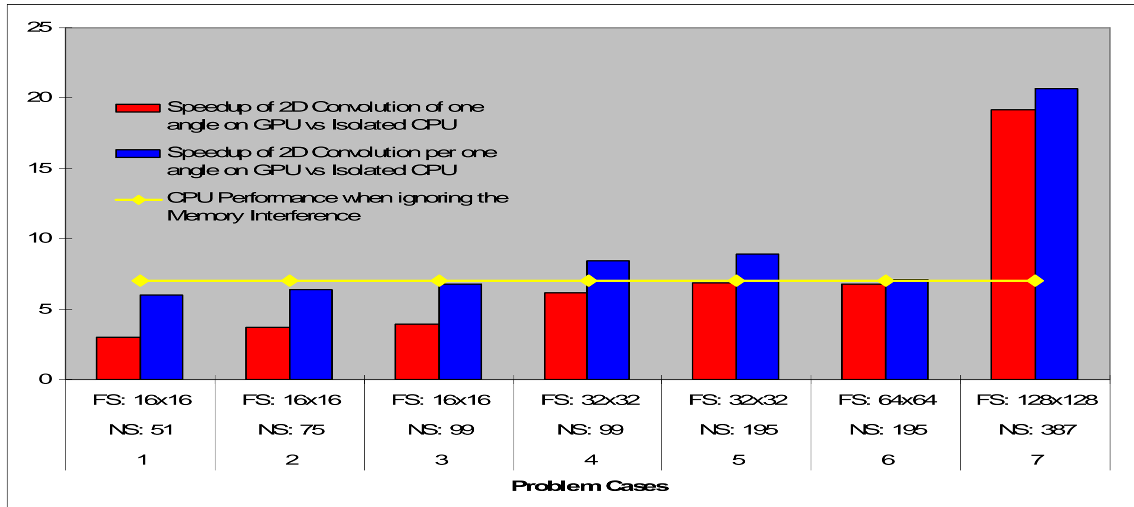


Figure 6.6: Speedup of 2D Convolution Computations of Single Angle on GPU when running one Angle versus 7 Angles of Incidence in Parallel in comparison to Isolated CPU, FS: FFT Size, NS: Number of Slices

However, as we notice the total execution time of 2D convolution of running 7 angles compared to the CPU does not lead to a speedup for the small problem sizes, where FFT size 16x16. Besides, it is clear from the graph depicted in Figure 6.6, for bigger cases except Case 7 (the largest) the required minimum speedup factor of 7 is not surpassed with large margins. For instance, case 7 shows a factor of 2.74 times faster for 2D convolution, which (according to Eq. 4.2) leads to a factor of 1.47 times faster regarding the whole EDM application.

Nevertheless, it must be noticed that all these comparisons were done against one separated CPU core run @3.0GHz, while in an actual system this CPU core, which runs @2.6GHz, is not isolated but works within 8 cores architecture. Recent investigations at ASML suggest that the Intel dual quad core CPU works at 5/8 efficiency for this application because

of memory interference. Taking this into account, the speedup of the 2D convolution will be increased by factor of 1.6. If we take a look at Figure 6.7, we see that when running multiple angles the speedup per one angle for all the cases surpasses the CPU performance with noticeable margins (See the intersection of the yellow line with the blue column).

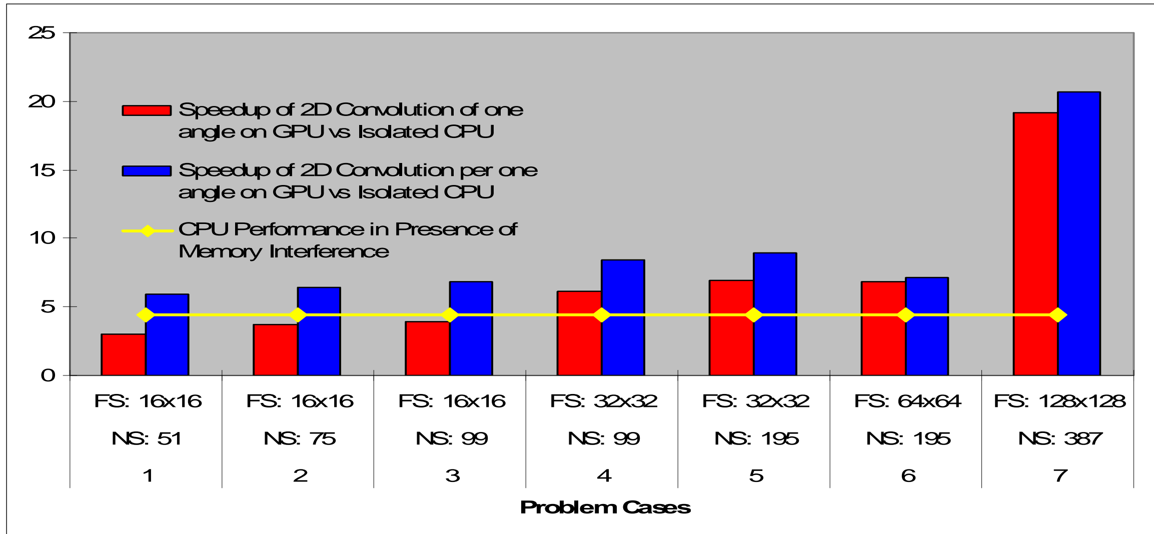


Figure 6.7: Speedup of 2D Convolution Computations of Single Angle on GPU when running one Angle versus 7 Angles of Incidence in Parallel in comparison to CPU in Presence of Memory Interference, FS: FFT Size, NS: Number of Slices

Table 6.6 shows a comparison of a cluster of CPUs against a cluster of CPU in terms of a few specification metrics, such as: number of cards, price, power consumption and peak theoretical performance.

	Metrics	CPU (Intel Xeons Dual Quad Cores)	Tesla C1060	Tesla S1070
1	# of Cards	14 cards	16 cards	4 cards
2	# of cores	8 cores per card, each core @2.6 GHz	240 cores per card, each core @1.3 GHz	960 cores per card, each core @1.5 GHz
3	Price	~$21,000	~$19,824	~$28,000
4	Max Power	~2700 W	3600 W	3200 W
5	Peak Performance	(SP) ~2.6 Tera Flops (DP) ~1.3 Tera Flops	(SP) 14.928 Tera Flops (DP) 1.248 Tera Flops	(SP) 16.56 Tera Flops (DP)1.384 Tera Flops
6	GFlops/Watt	~1.0 (SP)	4.1 (SP)	5.1 (SP)

Table 6.6: Comparison of a cluster of CPUs versus GPUs (Theoretically)

6.5. Summary

Many parallelization techniques have been exploited to show key features that may use the GPU as to accelerate the performance of the 2D convolution, which consumes about 50% of the total CPU time of EDM algorithm. This chapter evaluated the performance of the GPU-based 2D convolution implementations.

The first part has shown that the 2D convolution can run extremely fast on GPUs but

several aspects must be carefully taken into account, such as:

- Minimizing the time taken for uploading and downloading the data arrays between the CPU-GPU.
- Maximizing independent parallelism in the algorithm.
- Using effective execution configurations.

Algorithm 5.5 performs very well compared to the other algorithms, especially for big problem sizes, since it performs the FFT computations for batch of matrices in parallel. However, it does not show any speedup for the small problem sizes, where the FFT size is 16x16 and the number of layers is 32 and lower, because the matrix size is too small and it does not batch Transpose/Multiply kernels. On other hand, Algorithm 5.6 has considered all these parallelization aspects and it, thus, shows the best performance of the 2D convolution on the GPU.

Another delay was observed in accessing the data located in GPU's memory, which could be accelerated using the shared memory. However, it seems that the optimization using shared memory was not so valuable, if we compare the performance gain against the programming efforts. The timing measurements show that using the shared memory could optimize the execution time by at most 9%. Beside, it was not sufficient for the large problem sizes. This is due to the fact that, the importance of the shared memory appears only as a communication media between threads within a block, and not as a temporary storage media.

In addition, using 3D blocks technique was also not-so-rewarding trick, since it is not easy to program and there was a lack of proper documentation provided by NVIDIA. On the other hand, the general pattern, using 2D blocks, shows a slightly better performance with less programming efforts.

The second part of this chapter has compared the GPU-based implementations of the 2D convolution algorithm for single angle of incidence workload (Algorithm 5.6) to their counterpart that runs sequentially on the Isolated CPU. The timing results indicate a speedup factor of 3-19 times faster over the CPU-based implementation for a range of problem sizes.

Finally, the last part has evaluated the performance of the 2D convolution algorithm for multiple angles workloads on a GPU (Algorithm 5.7) comparing with multiple separated angles workloads on a multi-core CPU. It conducts that in same power budget a single GPU can work as 7 separated CPU cores. The results of running the 2D convolution of 7 workloads of EDM introduce a significant performance improvement per one workload of EDM. It exhibits a factor of ~6-20 times faster than the CPU-based implementation excluding the memory interference delay of CPU Dual Quad Cores. However, by ignoring this delay, the performance of accelerating only the 2D convolution on GPU may outperform the equivalent version that run serially on CPU, but only when the FFT size is 32x32 or higher and number of layers is 32 or higher. Taking this delay into account, leads to considerable speedup for all the cases, where the FFT size ranging from 16x16 to 128x128 and number of layers varying from 16 to 128 (See Figure 6.8).). However, the overall performance of the 2D convolution in this case is not feasible for the small problem sizes.

Special attention must be paid that all the measurements were done in single precision. Thus, since the current GPUs introduce much lower performance w.r.t. double precision computations compared to single precision, these results will probably be interpreted differently if double precision computations will be considered later, unless the new CUDA architecture generation “Fermi” will be used.

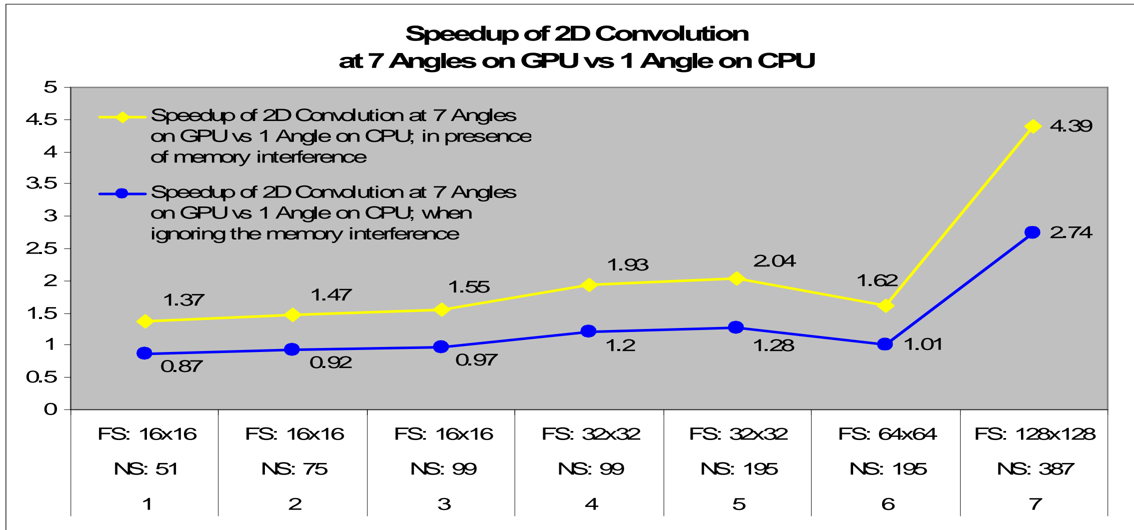


Figure 6.8: Speedup of 2D Convolution Computations at running 7 Angles of Incidence in parallel on GPU versus 1 Angle on CPU; FS: FFT Size, NS: Number of Slices

7 Conclusion and Future Work

This thesis was motivated by the framework of wafer metrology within ASML. It is aimed at measuring line profile on a wafer in order to obtain an optimum focus and exposure control with sub nanometer precision. This was done through scatterometry followed by a reconstruction process by means of electromagnetic diffraction model (EDM) in multiple angles of incidence. EDM computes the electromagnetic diffraction of the incident field by a grating structure. The time complexity of EDM is mainly determined by computing 2D convolutions. The latter algorithm involves computing direct/inverse FFTs and element-wise products in each layer in z-direction and for each of the field components.

The computationally intensive nature of EDM combined with the exploding growth of electromagnetic diffraction data sets, for an accurate solution, creates a continuously increasing demand for processing capability. Multi-core architectures like GPU form a new development trend in computer architecture, as a mechanism to avoid the power, memory and frequency scaling constraints of traditional microprocessor design, in an attempt to ensure the continuity of processing power increases.

At the outset of this project two research questions were posed, the first one is related to validation the feasibility of EDM application on the GPGPU. Second, how much performance gain can be achieved for one and multiple angles of incidence. Therefore, this thesis contributes by examining and evaluating the behavior of EDM application on the NVIDIA Tesla GPU architecture, a state-of-the-art heterogeneous many-core processor. The implementation of entire EDM on the GPU was outside the scope of this thesis. So, the GPGPU was used as a co-processor to accelerate certain parts of this algorithm for various problem sizes.

From quantitative analysis, we found out that EDM application can not entirely be parallelized because of the data-dependent nature of some of its calculations. From a hardware perspective, the GPU appears well-suited to some of its parallelizable associated algorithms (e.g. 2D convolution), as there is an abundance of parallelism available. The 2D convolution algorithm itself maps well to the GPU's multiprocessors and their SIMD-like nature. Thus, throughout this thesis, we investigated:

- The essential key points that must be taken into account when implementing such algorithm on CUDA GPU architecture.
- The performance of the 2D convolution of one EDM workload for single angle of incidence on a GPU compared to one CPU core.
- The performance of the 2D convolution of multi-EDM workloads—for multiple angles of incidence— per a single angle on a GPU compared to one CPU core.
- The performance of the 2D convolution of multi-EDM workloads for multiple angles of incidence on a GPU compared to multi-core CPUs.
- Hence, comparing the new CPU-GPU system against a cluster of CPUs in terms of a few metrics, such as: performance, power, space, and price.

In first place, this thesis was able to successfully translate the given code of the linear solver application to C and it could approve the stability of the algorithm in single as well as in single-double precisions. It indicates that the single precision version can work safely at faster performance than the equivalent versions that run in double and mixed precisions while it keeps almost same precision as the mixed precision calculations.

This study contributes with statistical data and benchmark of specific parts of the 2D convolution algorithm, showing its bottlenecks and pointing out interesting places to optimize the performance, which turned into finding other strategies of parallelizing.

Implementation of the 2D convolution framework resulted in two stages: parallelization of the FFT computations and the scalar-products for many disjointed small matrices. Parallelizing the computations of each matrix separately performs poorly due to the fact that a GPU needs to be assigned a massive amount of independent threads, so that each scalar core can do other useful work while the threads are waiting on the data or to synchronize. Besides, the number of calculations per fetched element from global memory should be large. Therefore, alternative solution was adopted to maximize data parallelism by streaming all the computations. It, basically, groups all the matrices in one vector as to perform the 2D FFTs on a batch of matrices on the GPU concurrently. The available NVIDIA CUDA FFT API supports batch execution for doing multiple 1D transforms in parallel, but it does not provide this feature for higher-dimensional transforms (2D and 3D). To utilize this optimal feature we have proposed an algorithm that performs a 2D FFT using a 1D transforms and doing matrix transposes. The latter approach delivers intense performance accelerations in comparing to the currently available un-batched 2D FFT solution from NVIDIA. Later, this algorithm was further optimized by combining the custom forward/inverse 2D FFTs with point-wise multiplications, which resulted in leaving out two matrix transposes. The second stage was parallelizing the execution of the scalar-products and matrix transposes. Thus, throughout this thesis several optimization schemes appeared in order to processing these two separated kernels for multiple streams in parallel by merging the threads of the multiple girds in one then optimally spreading these threads on the multiprocessors (e.g. specifying an optimal size of grids as 1D or 2D arrays of thread blocks and identifying each block as 2D or 3D arrays of threads, and using shared memory).

This demonstrates that the GPU does not introduce speedup unless a few important parameters are taken into consideration, such as:

- Maximizing the data parallelism.
- Reducing memory accesses.
- Spreading the workloads efficiently on the GPU's cores by choosing optimal execution configurations.

This highlights an important essence that not all algorithms are suitable to map on a GPU. CUDA programming model is easy to understand for C programmers, but it is more difficult to get an optimal mapping for the GPU. This already raises the question whether there is an optimal solution. Before it is implemented, it is hard to estimate the performance gain due to the aforementioned reasons.

This thesis has proved that the 2D convolution for a batch of matrices can be efficiently parallelized on GPU by using custom 2D FFT implementation. Real performance benefits can be seen only when performing 2D convolution of large FFT size on a big number of matrices in parallel. The speedup increases proportionally to the number of elements in the convolution. This is restricted in EDM application since the convolution is done for small computational

problem sizes, where the number of convolutions and elements in a convolution are very limited. The timing results of running single angle of incidence workload indicate a noticeable performance gain over its counterpart implementation that runs serially on the CPU, particularly when the FFT size is 32x32 or higher, and the number of layers is 32 or higher. Therefore, that motivates to maximize the workload to increase the parallelism, especially for the small problem sizes, where FFT size is 16x16. That could be achieved by running multiple versions of EDM for different angles incidence in parallel. This is useful as it allows more FFT computations. The timing results per an angle show a significant speedup factor of ~6-20 times over the serial CPU-based implementation (for FFT sizes varying from 16x16 to 128x128). Thus, we conclude that the multi-angle application is a must to employ the GPU more efficiently.

In the realistic ASML application, the performance of a cluster of GPUs will be compared to a cluster of CPUs, where the number of each is determined by a specification of maximum power consumption and generated heatload. Thus, in terms of this metric the one-angle version is not feasible. For the multi-angles version at the defined heatload, it turns out that for accelerating the 2D convolutions "only" for FFT size under consideration, the performance gain is critical since it is beyond the estimated factor. However, recent investigations at ASML for this application have shown that the performance of an Intel Dual Quad Core is reduced by a factor of 5/8 due to effects of memory interference. Taking this performance reduction into account, the GPU shows a considerable speedup. Besides, it must be noted that GPU-based system is slightly cheaper and occupies smaller space.

Our recommendations besides accelerating 2D convolution are to offload more computations of EDM to the GPU in order to increase the overall performance for the entire application. For instance, Matrix-Vector product algorithm is computationally intensive algorithm and consumes about 80% of the total execution time of EDM algorithm (including 2D convolutions), so parallelizing the remained 30% may lead to considerable overall speedup for the application according to Amdahl's law. Thus, implementing this part of the algorithm and comparing the resultant behavior with expectations set in this thesis would be of interest.

Another attractive future work would be interesting to investigate the application's behavior on the NVIDIA's new generation of CUDA computing architecture "Fermi", since it shows plenty of hardware optimizations. Thus, this may accelerate the computations by a significant factor especially for double floating point computations since, it promises 8x faster performance compared to the current GPUs.

Furthermore, since this study investigates all the computations in single floating point precision, while a higher precision is still preferred for this application, it would be also useful to validate the application's results for higher precision at better performance. For instance, extending the work of mixed floating point precision computations by adopting the approach presented in [36] for EDM application.

Bibliography

---

[1] NVIDIA Corporation, "Tesla C1060 Computing Processor Board", NVIDIA, 2008.

[2] NVIDIA Corporation, "NVIDIA CUDA, Programming Guide Version 1.1", July 2009.

[3] NVIDIA Corporation, "NVIDIA CUDA: Compute Unified Device Architecture, Reference Manual", June 2008.

[4] NVIDIA Corporation, "CUDA CUFFT Library", October 2008.

[5] NVIDIA Corporation, "CUDA CUBLAS Library", March 2008.

[6] NVIDIA Corporation, "NVIDIA Compute PTX: Parallel Thread Execution", July 2008.

[7] NVIDIA Corporation, http://www.nvidia.com

[8] AMD Corporation, http://www.amd.com/uk/Pages/AMDHomePage.aspx

[9] M. Ekman, F. Warg, J. Nilsson, "An In-Depth Look at Computer Performance Growth", ACM SIGARCH Computer Architecture News, Volume 33, Issue 1, March 2005.

[10] NVIDIA Corporation, "Optimizing Matrix Transpose in CUDA", 2009.

[11] Chen T., Raghavan R., Dale J., and Iwata E., "Cell broadband engine architecture and its first implementation– a performance view". IBM Journal of Research and Development, v. 51, n. 5, 2007. http://www.ibm.com/developerworks/power/library/pa-cellperf/

[12] Buatois L., Caumon G., Lévy B, "Concurrent number cruncher: An efficient sparse linear solver on the GPU", 2007.

[13] Krijger J., Weserman R., "Linear Algebra Operations for GPU Implementation of Numerical Algorithms", 2003.

[14] Introduction, GPGPU history, ATI Stream and CUDA overviews http://www.pcper.com/article.php?aid=745

[15] Ikonas Graphics Systems http://www.virhistory.com/ikonas/ikonas.html

[16] Wikipedia Amdahl's law, http://en.wikipedia.org/wiki/Amdahl%27s_law

[17] Wolfram Mathworld. “Convolution Theorem” http://mathworld.wolfram.com/ConvolutionTheorem.html

[18] Wolfram Mathworld. “Fast Fourier Transform” http://mathworld.wolfram.com/FastFourierTransform.html

[19] Kozin I. N., “Overview of novel architectures“, MEW19 presentation, Dec 2008.

[20] Lippert A., “NVIDIA GPU Architecture for General Purpose Computing“, NVIDIA presentation, April, 2009.

[21] NVIDIA Corporation, “NVIDIA TESLA S1070 DATASHEET“.

[22] NVIDIA Corporation, “NVIDIA TESLA C1060 DATASHEET“.

[23] Wikipedia, http://en.wikipedia.org/wiki/Multi-core_(computing)

[24] Wikipedia, http://en.wikipedia.org/wiki/AMD_FireStream

[25] http://reviews.cnet.com/graphics-cards/nvidia-geforce-8800-gts/4505-8902_7-32143069.html

[26] NVIDIA Corporation, CUDA Technology; http://www.nvidia.com/CUDA.

[27] http://cui.unige.ch/~chopard/GPGPU/3-more-on-cuda.pdf

[28] NVIDIA Corporation, NVIDIA CUDA Programming Guide 2.0, 2008: http://developer.download.nvidia.com/compute/cuda/2_0/docs/NVIDIA_CUDA_Programming_Guide_2.0.pdf

[29] NVIDIA Corporation, NVIDIA SDK samples, TransposeNew, 2009.

[30] Walsh R., Conway S., Joseph E. C., and Wu J., “Powerxcell“, 2008.

[31] W. B. Langdon CREST centre, King’s College, “A CUDA SIMT Interpreter for Genetic Programming“, 2009.

[32] Li J., Martinez J. F., “Dynamic power-performance adaptation of parallel computation on chip multiprocessors“, 12th International Symposium on High-Performance Computer Architecture, HPCA-12 2006, Austin, Texas, February 2006.

[33] Govindaraju N. K., Lloyd B. Dotsenko Y., Smith B., and Manferdelli J., “High Performance Discrete Fourier Transforms on Graphics Processors“, Microsoft

Corporation, SC '08: Proceedings of the 2008 ACM/IEEE conference on Supercomputing, 2008.

[34] M. Gschwind, H. P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, T. Yamazaki, " Synergistic Processing in Cell's Multicore Architecture ", Micro, IEEE Volume 26, Issue 2, March-April 2006

[35] NVIDIA Corporation, NVIDIA optimization guide, 2009.

[36] Goddeke D, Strzodka R., Turek S., “Accelerating Double Precision FEM Simulations with GPUs“, Sep. 2005.

[37] Glaskowsky P. N., “NVIDIA’s Fermi: The First Complete GPU Computing Architecture“, A white paper, September 2009.

[38] NVIDIA Corporation, “NVIDIA® CUDA™ Architecture Introduction & Overview”, March 2009.

[39] NVIDIA Corporation, “The CUDA Compiler Driver NVCC”, 2008.

[40] NVIDIA Corporation, NVIDIA SDK samples, FFT-based 2D convolution, 2009.