DOCTOR OF PHILOSOPHY

High performance numerical modeling of ultra-short laser pulse propagation based on multithreaded parallel hardware

Mandana Baregheh

2013

Aston University

Aston University

Some pages of this thesis may have been removed for copyright restrictions.

If you have discovered material in AURA which is unlawful e.g. breaches copyright, (either yours or that of a third party) or any other law, including but not limited to those relating to patent, trademark, confidentiality, data protection, obscenity, defamation, libel, then please read our Takedown Policy and contact the service immediately

High Performance Numerical Modeling of Ultra-short Laser Pulse Propagation based on Multithreaded Parallel Hardware

MANDANA BAREGHEH

Doctor of Philosophy

ASTON UNIVERSITY

December 2012

©Mandana Baregheh, 2012

Mandana Baregheh asserts her moral right to be identified as the author of this thesis

This copy of the thesis has been supplied on condition that anyone who consults it is understood to recognise that its copyright rests with its author and that no quotation from the thesis and no information derived from it may be published without proper acknowledgement.

Aston University

High Performance Numerical Modeling of Ultra-short Laser pulse Propagation based on Multithreaded Parallel Hardware

Mandana Baregheh

Doctor of Philosophy

December 2012

Summary

The focus of this study is development of parallelised version of severely sequential and iterative numerical algorithms based on multi-threaded parallel platform such as a graphics processing unit. This requires design and development of a platform-specific numerical solution that can benefit from the parallel capabilities of the chosen platform.

Graphics processing unit was chosen as a parallel platform for design and development of a numerical solution for a specific physical model in non-linear optics. This problem appears in describing ultra-short pulse propagation in bulk transparent media that has recently been subject to several theoretical and numerical studies. The mathematical model describing this phenomenon is a challenging and complex problem and its numerical modeling limited on current modern workstations.

Numerical modeling of this problem requires a parallelisation of an essentially serial algorithms and elimination of numerical bottlenecks. The main challenge to overcome is parallelisation of the globally non-local mathematical model.

This thesis presents a numerical solution for elimination of numerical bottleneck associated with the non-local nature of the mathematical model. The accuracy and performance of the parallel code is identified by back-to-back testing with a similar serial version.

Keywords: Femtosecond phenomena, Non-linear Schrödinger equation, Parallel numerical simulations

تقدیم به پدر و مادرم،

Acknowledgments

First and foremost, I would like to express my deepest gratitude to my supervisor, Dr. Vladimir Mezentsev, for your invaluable guidance and support throughout my years at Aston. You have been a tremendous mentor for me. Also a special thank you to Prof. Sergei Turitsyn for inviting me and welcoming me to the group.

I owe a great deal of gratitude to all my co-authors, Dr. Holger Schmitz, Janarthanan Rasakanthan and Alexandr Dostovalov, for giving me the chance to work with you. Many thanks to Elena, Pouneh and Mercedes, for making a strictly engineering environment, professional with a shade of pink.

A special thank you to my family. Words cannot express how grateful I am to my mother; you have been always there for me with unconditional love and care. To my beloved sister, I cant thank you enough for your continuous encouragement and support in all means. A special thanks to my husband, for being patient and for standing by me even when I was irritable and depressed.

Finally I thank my God, my Father, for your endless support. You are everything to me.

List of Contents

List of Figures

List of Tables

Abbreviations and units

ALU	Arithmetic Logic Units
CPI	Clock Cycles Per Instruction
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
FFT	Fast Fourier Transform
FFTW	Fast Fourier Transform in the West
FPU	Floating Point Units
fs	femtosecond, 10^{-15} sec
GFLOPS	GIGA Floating Point Operations Per Second
GNLSE	Generalised Non-Linear Schrödinger Equation
GPU	Graphics Processing Units
GVD	Group Velocity Dispersion
HPC	High Performance Computing
MIMD	Multiple Instruction, Multiple Data stream
MISD	Multiple Instruction, Single Data stream
MPA	Multi-Photon Absorption
MPI	Multi-Photon Ionisation
NLSE	Non-Linear Schrödinger Equation
NNLSE	Non-local Non-Linear Schrödinger Equation
OCT	Optical Coherence Tomography
PC	Personal Computer
SDK	Software Development Kit
SIMD	Single Instruction, Multiple Data stream
SM	Streaming Multiprocessor
SSFM	Split-step Fourier Method
SSID	Single Instruction, Single Data stream
SP	Streaming Processor
2D	Two-Dimensional

Not everything that can be counted counts, and not everything that counts can be counted.

- Albert Einstein

Chapter 1

Introduction

Mathematical models describing physical phenomena are widely used in science and engineering applications whenever real world experiments are dangerous, expensive or impossible. Building a realistic simulation model of a physical experiment requires a large number of data inputs and a large number of intricate interactions. This in turn requires a high capacity/bandwidth memory and high computation power. *High Performance Computing* (HPC) makes this possible using parallel hardware to increase the computation power and achieve more accurate results whilst reducing the processing time.

Parallel implementation of numerical models means collaboration of a number of computer resources to solve the problem simultaneously. Most importantly, parallel implementation can lead to elimination of *bottlenecks* which are parts of the program that are slow resulting in limitation of the overall speed of the program. This thesis is focused on effective use of HPC for eliminating bottlenecks and increasing the speed of complex numerical applications. The results of this research have been published and presented in [1–8].

The main focus of this work is design and development of numerical algorithms based on multi-threaded hardware for high performance modeling of a specific physical problem appearing in a wide class of phenomena in non-linear optics. This physical problem appears in describing ultra-short pulse propagation in bulk transparent media. Considering the fact that efficient numerical modeling is highly dependent on efficient usage of the HPC resources, this work includes a description of different HPC platforms and that of the chosen parallel hardware.

The mathematical model for the above mentioned physical problem, presented in Chapter 3, is a challenging and complex extended version of the Non-Linear Schrödinger Equation(NLSE). The extended NLSE coupled to the Drude model of plasma results is the subject of this work, addressed by the Non-local Non-Linear Schrödinger Equation (NNLSE). 3D numerical modeling of such problem is not feasible on modern workstations. However, the high performance computing based on the multithreaded hardware overcomes this limitation.

This work presents a platform specific numerical solution based on the split-step Fourier technique by a problem specific succession of linear and nonlinear operators that approximate the original solution (Chapter 4). The solution of both the linear and the non-linear operators requires parallelisation of some essentially serial algorithms and elimination of the numerical bottlenecks by exploiting the parallel capabilities of the specific hardware. The non-local nature of the non-linear operator makes the parallel implementation challenging. Hence, in Chapter 4 a novel approach is adopted for an efficient parallel solution to the non-local non-linear operator.

In addition, this work also contributes to efforts on exploiting parallel capabilities of the specific hardware for design and development of numerical algorithms as separate modules (Chapter 6). These modules together form the parallel implementation of the numerical model. Proof of efficiency, accuracy and robustness of this implementation is also presented based on by back-to-back testing in a realistic context, reflecting typical experimental regimes of femtosecond laser inscription.

This work also contributes by adopting a novel approach for identification of roughness (poor resolution) in results obtained from a serial CPU-based numerical implementation of the physical problem which was previously developed and implemented in [7,9,10] (Chapter 5). The data obtained from this implementation suffers from insufficient numerical resolution and require refinement of the grids for which identification of roughness is necessary.

1.1 Thesis structure

The thesis is structured according to the following scheme:

Chapter 1 summarises the main contributions of this work.

Chapter 2 introduces high performance computing for development of complicated scientific applications. This chapter describes the limitations of single processors and introduces a number of HPC platforms. CUDA-enabled GPUs are also introduced in this chapter as the chosen parallel hardware for this work.

Chapter 3 presents a mathematical model for ultra-short pulse propagation in bulk transparent media based on non-linear wave equation. In this chapter a description of all the linear and non-linear regimes that effects this process is given.

Chapter 4 describes the numerical scheme used for solution of the non-local non-linear Schrödinger equation using the split-step method.

Chapter 5 introduces an adaptive mesh refinement routine to obtain higher resolution from the data produced by the serial or parallel Schrödinger solver.

Chapter 6 describes the parallel design and implementation of all the numerical modules needed for solution of the non-local non-linear Schrödinger equation.

Chapter 7 verifies the results obtained from the parallel non-local non-linear Schrödinger equation which is obtained by comparing the time efficiency and accuracy with a similar CPU serial implementation.

In Conclusion a summary of the results is provided, as well as a discussion of possible future work.

Chapter 2

Overview and Choice of Computing Platform

2.1 Computer Architecture

The traditional serial computers are simply described by a memory, which is connected to a processor through a data-path. The processing speed of these computers is directly affected by bottlenecks caused by all these three elements [11]. To overcome these bottlenecks different architectural innovations have been implemented during the years, among which clever pipelining and constant increase in the number of transistors can be named.

However, recently the rapid pace of advances in technology has been slowing down as it is getting harder and harder to reduce the heat produced and the power consumption by a large number of transistors on a single chip. In other words, faster uni-processors can be obtained by increasing the clock frequency or the pipelining depth. Increasing the clock frequency requires a very large transistor count which results in a significant increase in energy consumption, heating up the computer chip severely. As for the pipelining depth, problems such as branch prediction prevent addition of more pipelining stages. This means that single processors cannot have massive speed ups anymore and to acquire a faster computer, more and more processors are needed beside each other to form multi-core systems or multiprocessors [12].

Multi-core systems are equipped with a number of processors integrated on a single chip. As for multiprocessors, multiple processors in one single computer or multiple computers are interconnected to share their resource to create a massively parallel system. As it will be explained later, communication between cores is faster in multi-core computing (inter-CPU communication) than that of multiprocessors where the communication is through a network or the motherboard, and hence is slower [12].

2.1.1 Parallel Architecture

Parallelism in computer systems can be in instruction or data stream, and is categorized by Flynn [13] into four different groups:

- *Single Instruction, Single Data stream* (SISD), uni-processors or the traditional Personal Computer(PC) that operates sequentially.
- *Single Instruction, Multiple Data stream* (SIMD), where the same control unit sends a set of instructions to each processing unit to execute them on different data elements simultaneously. SIMD architecture provides *data level parallelism* by executing the same instructions in parallel on structured data elements, such as arrays. The architecture design of the SIMD model is based on the regular structured computations needed for applications like image processing and graphics. It is used in *Graphics*

Aston University

Illustration removed for copyright restrictions

Figure 2.1: SIMD and shared memory MIMD architecture [11]

Processing Units (GPUs) where the same operation is executed on a large number of pixels [11, 12].

- *Multiple Instruction, Single Data stream* (MISD), where independent processing units operate on the same data stream. This model in not used in HPC.
- *Multiple Instruction, Multiple Data stream* (MIMD), comprises many processors that have the capability to execute different set of instructions on different data elements. The MIMD model provides *task level parallelism* which is seen in clusters where nodes are independent and have their own local memory [14].

Fig. 2.1 shows a SIMD model where only one control unit operates and as a result in comparison to the MIMD model, SIMD requires less hardware. In both the SIMD and MIMD parallel models, communication can occur via shared memory access model or distributed memory access. In the shared memory access, the processing elements communicate via

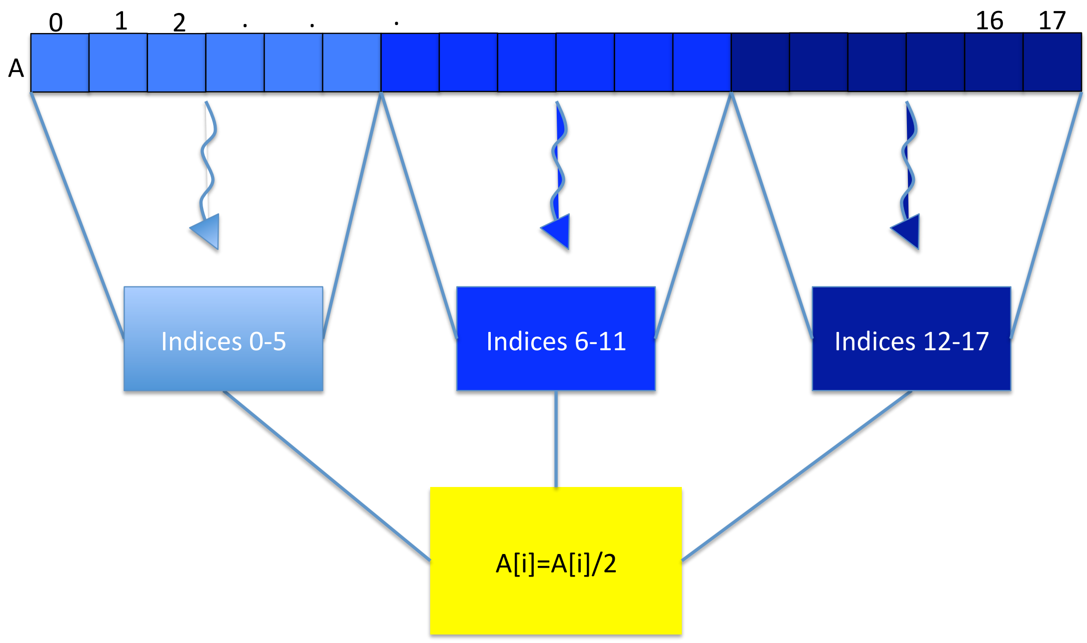


Figure 2.2: Data Level Parallelism in SIMD machines

a common memory, where each processing element is additionally equipped with its own local memory. Cache memory is an instance of processing element's local memory, used to reduce memory latency by bringing data closer to processors. Multiprocessors work with the shared memory access model of the MIMD architecture [11].

In distributed memory access communication occurs by passing messages through a communication network or interconnect. The distributed or message passing model consists of a number of nodes like clusters which are connected through an interconnect, providing means for data transfer between the processing nodes [11].

Considering the memory model of the parallel computers, communication of data between the processes can incur a great cost. Programming model between processors in the distributed memory and shared memory model is described in [14]. As described in this paper, communication between processors in distributed memory model is usually done using the message passing interface and the communication latency varies depending on the

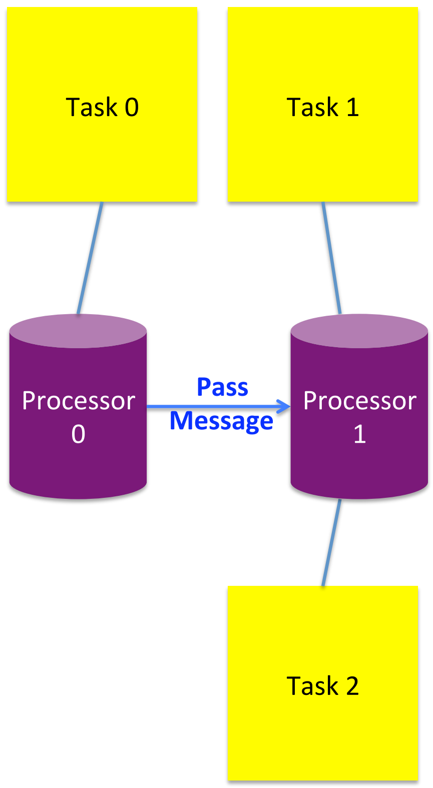


Figure 2.3: Task Level Parallelism in MIMD machines

choice of the interconnect. For the shared memory model, parallelism is provided for example by the OpenMP programming interface, where still accessing a non-local memory has a high overhead depending on the memory layout of the specific system. However, since each processor can execute different set of instructions synchronously, MIMD machines are more flexible and they have the capability of executing other HPC models. They are commonly used for executing parallel applications. Fig. 2.3 illustrates the task level parallelism in distributed memory MIMD machines, where two processors execute in parallel Task0 and Task1. After completion of these tasks, the message passing interface is used to send the result from one machine to the other before execution of Task2.

The SIMD architecture model is more suitable for scientific computing where repetitious calculations are required for structured data elements in parallel. Fig. 2.2 shows the programming model for a SIMD machine, where the same operation (division by 2) is performed on all elements of an array in parallel by a number of threads. A thread is the smallest processing unit, managed by the operating system, which executes a collection of instructions. It should be noted that due to their data parallel programming model, SIMD machines are less flexible and only certain kinds of parallel algorithms work well with them [11, 15].

As mentioned earlier, graphic processing units are designed based on the SIMD architecture model for graphics rendering of a large number of pixels. GPUs can also be used for non-graphics applications, like numerical applications, where the same instruction has to be executed on elements of an array or a matrix in parallel. This is possible by converting a desktop computer to a supercomputer utilizing GPUs. In this way, GPUs will work as co-processors for CPUs to exploit parallelism. It should be noted that the cost of purchase, set up and maintenance of the GPUs is drastically lower than clusters. At this point a decision had to be made on the choice of the parallel hardware between multi-core CPUs, clusters and GPUs. The next section describes all the considerations that had to be taken into account to this effect.

2.2 Choice of Parallel Hardware

Recent advances in CPUs have lead to the use of multi-core systems with hyper-threading technology, where every processor core can run two independent threads at once. For instance, let's consider a multi-threaded single core processor equipped with hardware that can execute multiple threads efficiently. Multi-threading in single core processors is possible through time multiplexing, where the processor switches between different threads. Time multiplexing happens so frequently that it is perceived as if the threads are running simultaneously. In this single core system there is one thread of execution at every instance of time. In a multi-core system, all the CPU resources are replicated into cores. Parallelism is achieved by assignment of threads to these cores to work in parallel. In this way, higher throughput is gained by simultaneous execution of instructions on each core [16].

The Intel Core i7 used in this work has 4 cores with hyper-thread technology. This results in having 8 concurrent threads with the power to calculate 69.23 *GIGA Floating Point Operations Per Second* (GFLOPS). Superior performance is achieved in GPUs by combining the SIMD architecture and the multi-core technology. This is achieved by assigning several *Arithmetic Logic Units* (ALU) for logic and arithmetic operations on integers to every core. This means that a large number of ALUs will perform the same set of instructions on different data elements through the SIMD architecture. GPU technology is capable of running a significantly higher number of active threads, 32 for NVidia and 64 for AMD [16].

To process a large array of scientific data using clusters, initially data should be divided into batches. Subsequently each individual batch should be sent across the interconnect to a processing node where computation will be performed on the data. Frequent communication between the nodes through the network is required, which results in high ratio of communication to processing time, causing communication inefficiencies. Also increasing the number of nodes, increases the performance but at the same time produces more intercommunication overhead.

Even though the cost of a commodity single cluster node is low, this cost increases significantly by increasing the number of nodes. Also the cost of maintaining all these nodes is quite high. In contrast, GPUs provide high performance computing with significantly lower cost. As an instance, NVidia's Tesla technology provides multi-core GPU design by assigning an ALU to every core. The Tesla C1060 is equipped with a total of 30 multiprocessors, each containing 8 cores, these GPUs provide a total of 240 cores at a relatively small cost.

While the lower cost of purchase and maintenance of GPUs is an advantage, one must also consider their arithmetic precision. CPU and cluster node's *Floating Point Units* (FPU)[1] support 80-bit units equivalent to extended double, whereas not all the graphics cards support double precision. Most GPUs use a 32 bit equivalent of single precision floating point. On the other hand while CPUs have only one FPU unit per core, GPUs are equipped with 32 FPU units per multiprocessor. Considering the fact that high-end CPUs are equipped with four cores and GPUs with several hundred multiprocessors, this makes a big difference when using many floating point operations [17].

The lack of double precision in many GPUs is due to the fact that they were initially designed for a graphics pipeline, for which single precision floating point operations are sufficient. For general purpose computing, double precision support is increasing but its speed is still slower than that of single precision floating point operations. Major graphics card manufacturers like NVidia, AMD/ATI have included double precision to their latest products but in a limited capacity (not every streaming processor or core is equipped with a double precision unit). These units are shared between more processors which leads to reduction in the double precision throughput [18].

Taking into account the different parallel hardware described above, considering the focus of this work on elimination of bottlenecks from specific scientific applications, the decision was made to use GPUs as the chosen HPC hardware. This decision was based on the numerical nature of the scientific applications, the SIMD parallel model of GPUs and also the lower cost of purchase and maintenance.

[1]FPUs perform arithmetic operations between two floating-point values

2.2.1 Graphics Processing Units

GPU's throughput-oriented design with multi-core multi-threading and high memory bandwidth has become more and more optimized. In the past decade, GPU's usage has changed from only graphics rendering to general-purpose applications, where their highly parallel capabilities can be put to good use. The challenge in programming GPUs is converting a serial code to a parallel one by adapting different algorithms. Consideration should be given to the limitation on number of threads running on GPUs, which depends on the amount of memory resources needed per thread. In other words, whenever large number of registers (or amount of memory) is needed, the limit on the number of threads reduces accordingly [16].

Not all applications can produce better results on GPUs. Applications that have the following characteristics work well with GPUs:

- Application where the number of threads is far larger than the number of GPU cores. This is due to the fact that GPUs are designed to handle a large number of threads.
- Numerical applications where data is stored in structures like arrays, where the same operation is executed on all the elements of the array using threads. Hence, assigning one thread to every element of array and utilizing a large number of parallel threads (data parallelism).
- Applications where sharing data between threads is only needed for threads in the same block.
- Also applications that use hardware-supported local operations like exponential or square root, which can be executed faster on GPUs in comparison with CPUs.

On the contrary, applications that are serial, have branches (conditional statements), need

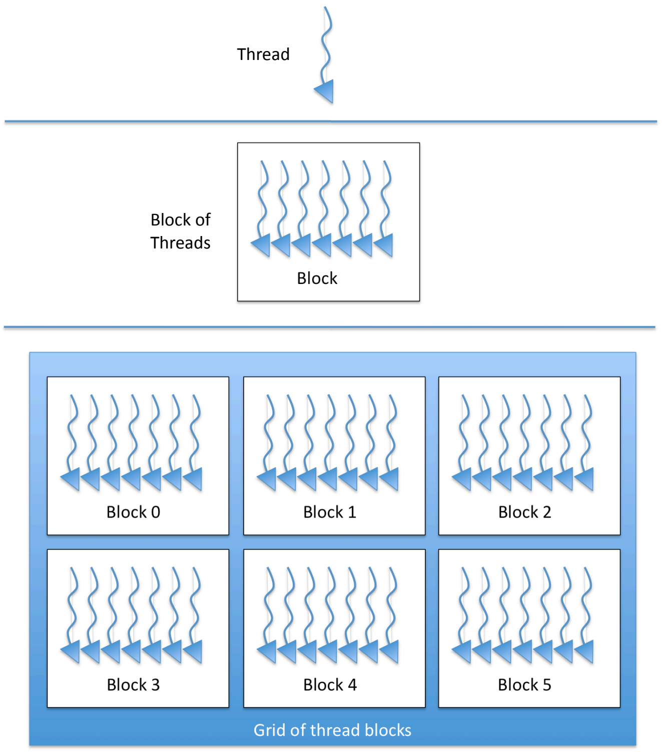


Figure 2.6: Cuda threads, blocks and grids [20]

2.4 Thread Management

As mentioned in the previous chapter, top of the range CPUs (or *hosts*) can support execution of a small number of threads, 8 on quad-core computers, whereas NVidia CUDA GPUs (*devices*) can support up to 768, 1024, 1536 active threads per multiprocessor depending on the GPU capabilities. Apart from the differences in quantities of the active threads on host and device, their entities also differ. CPUs support heavy-weight threads which require slow and expensive *context switching* to switch from one thread to another (i.e., it is costly to assign/stall CPU threads). CUDA threads are lightweight; they operate on small data units and can be scheduled/stalled with little cost. At the same time, a big number of threads can be active at once. This is the exact reason why GPUs are good for data parallelism and CPUs for task parallelism [24].

Both data parallel and task parallel execution are supported in OpenCL [21].

To conclude, it should be noted that commodity numerical software is typically developed for sequential algorithms whereas parallel implementation of numerical operations depends on the choice of hardware platform. For example, to employ clusters the problem should be divided into disjoint domains, where each individual domain will be executed on one cluster serially. Parallel environments deal with the entire domain at the same time, hence the numerical solution should be platform specific. In this study GPUs were chosen considering the data-parallel nature of the scientific applications and specifically CUDA-enabled GPUs were chosen considering their adaptability and programming model.

2.3 The CUDA Computing Platform

CUDA is a commodity co-processor that makes parallel programming possible at a small cost. CUDA-enabled GPUs bring the high computational power and memory bandwidth

Aston University

Illustration removed for copyright restrictions

Figure 2.4: Comparison of Memory Bandwidth in GPUs and CPUs (GB/sec) [22, 23]

Aston University

Illustration removed for copyright restrictions

Figure 2.5: Comparison of Floating Point Operations per Second on GPUs and CPUs (GFLOP/sec) [22, 23]

Figures 2.4 and 2.5 show the superior improvement of GPUs performance compared to CPUs. As illustrated in these figures, the potential floating point speed on GPUs has raised significantly since 2004 (significant increase in the GPU/CPU ratio from 3 to 12). In 2010 this ratio was obtained from comparing NVidia GeForce GTX 480 with 1344.96 GFLOPS and Intel Core i7-980X with 109 GFLOPS. The performance difference shown in these two figures is a result of the sequential performance of CPUs compared to parallel design of GPUs.

In this parallel design, transistors are more involved in data processing procedures and less in data caching and flow control. NVidia's GPUs use the *Single Instruction Multiple Thread* (SIMT) architecture or multiple SIMD, which will execute only one single instruction on multiple data elements. As a result, there will be less need for flow control. In this model, since the same instruction is executed on different data elements, the delay due to memory access can be concealed with computation and there will not be any need for big caches [20].

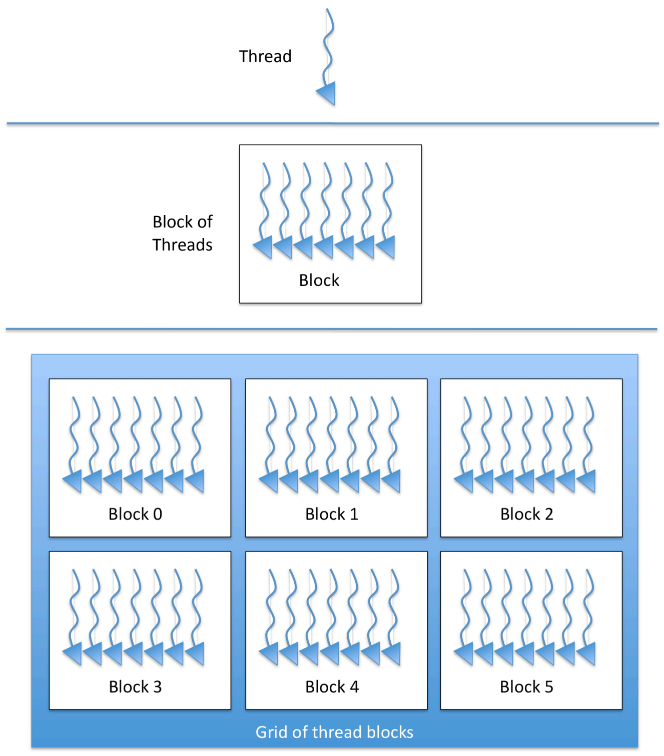


Figure 2.6: Cuda threads, blocks and grids [20]

2.4 Thread Management

As mentioned in the previous chapter, top of the range CPUs (or *hosts*) can support execution of a small number of threads, 8 on quad-core computers, whereas NVidia CUDA GPUs (*devices*) can support up to 768, 1024, 1536 active threads per multiprocessor depending on the GPU capabilities. Apart from the differences in quantities of the active threads on host and device, their entities also differ. CPUs support heavy-weight threads which require slow and expensive *context switching* to switch from one thread to another (i.e., it is costly to assign/stall CPU threads). CUDA threads are lightweight; they operate on small data units and can be scheduled/stalled with little cost. At the same time, a big number of threads can be active at once. This is the exact reason why GPUs are good for data parallelism and CPUs for task parallelism [24].

In CUDA's parallel design, a large number of pixels are mapped to parallel threads for graphics rendering. For numerical applications, data elements (e.g. elements of an array) are mapped to threads for parallel execution of numerical algorithms. Figure 2.6 illustrates a number of threads that are grouped together to form a *block*, and these blocks are structured into a *grid*, where all the threads in a grid execute the same set of instructions. The CUDA device defines the maximum number of threads and blocks in each grid. However, the developer should specify the actual number of threads used in each application. Consequently, the actual number of blocks is usually calculated by the problem size and the number of processors in the system [20].

GPU	**Compute Capability**	**Max Threads/Block**	**Max Block Dimensions**	**Max Grid Dimensions**	**Max Active threads/MP**
Quadro FX 570	1.1	512	$512 \times 512 \times 64$	65535×65535	768
Tesla C 1060	1.3	512	$512 \times 512 \times 64$	65535×65535	1024
Tesla M 2050	2.0	1024	$1024 \times 1024 \times 64$	$65535 \times 65535 \times 65535$	1536

Table 2.1: Maximum number of threads and maximum block and grid dimension on several CUDA-enabled GPUs (MP stands for multiprocessor) [20, 25, 26]

Table 2.1 shows the maximum number of threads and blocks for three different CUDA-enabled GPUs. In this table, *Compute Capability* describes the hardware specifications of CUDA cores and consists of two parts: the major revision number which is the whole number part and the minor revision number which is the number after the decimal point. Devices with the same major revision number are of the same architecture and minor revision number describes the minor changes and improvements to the cores [20].

Using CUDA as the programming interface, the GPU acts as the computation and together with a traditional CPU will run the application. The host will send the highly parallel computational instructions to the device and the threads on the device will perform them in parallel. The part of the application that needs to be executed many times on different data elements can be defined as a function, which is called the *kernel*. Once a kernel is launched, a grid of thread blocks will be created comprising of a large number of lightweight threads to execute this kernel on different data elements [20].

CUDA provides a software environment that enables programming using a high level languages similar to C. This software environment is a combination of a set of extensions to the C language and a runtime library. These extensions are used by the developer to write the kernel as a C function and also set the number of threads and blocks for kernel execution. The runtime library provides means for device initialisation, memory and context management using C functions executed on the host. Compilation of the kernel code to the binary code is done by NVidia's own C compiler driver called *NVCC* for execution on the device [20].

2.5 Hardware Model

CUDA's hardware consists of an array of multi-threaded *Streaming Multiprocessors*(SM) and each SM contains a number of computation engines called *Streaming Processor*(SP) which are equipped with ALUs. The numbers of SMs and SPs for three different CUDA-enabled GPUs are shown in Table 2.2. The fourth column of this table indicates the number of CUDA cores, or in other words, the total number of SPs on the graphics card. Each CUDA core for devices of compute capability 2.x has a pipelined floating point unit in addition to the pipelined integer unit supporting both single and double precision arithmetic. Tesla M2050 supports 515 GFLOPS for double and 1030 GFLOPS for single precision with its advanced new Fermi architecture [26, 27]. It should be mentioned that the number of SPs in each SM determines the instruction throughput. The higher the number of SPs, the more instructions are processed at the same time [20].

GPU	Compute Capability	No. of Streaming Multiprocessor	No. of Streaming Processor	No. of CUDA cores
Quadro FX 570	1.1	4	8	32
Tesla C 1060	1.3	30	8	240
Tesla M 2050	2.0	14	32	448

Table 2.2: Number of SMs, SPs and CUDA cores [20,25,26]

When the host launches a kernel grid, multiple thread blocks will be assigned to SMs based on the resource usage. For example in Tesla C 1060, maximum 8 blocks can be assigned to each SM. Threads in each block perform simultaneously and different block can work in parallel on each SM. Every time a block terminates, new blocks will be invoked to finish the execution of the kernel [28].

Threads in each multiprocessor are scheduled to operate in groups of size 32 parallel threads which is called a *warp*. Once a warp stalls, the SM can switch to another resident warp with zero overhead, this helps to hide the memory and arithmetic latency. It should be noted that assigning the number of threads to a multiple of 32 (warp size), results in a more efficient thread management [28].

CUDA needs one of NVidia's CUDA-enabled GPUs that include GEFORCE 8, 9 and 200 series, TESLA or QUADRO architecture. On devices with compute capability of 2.0 and higher, more than one kernel can be executed at any time instance. This in-turn improves CUDA's flexibility significantly. The SIMT architecture used in CUDA provides data level parallelism through coordinated threads, as well as thread level parallelism through in-

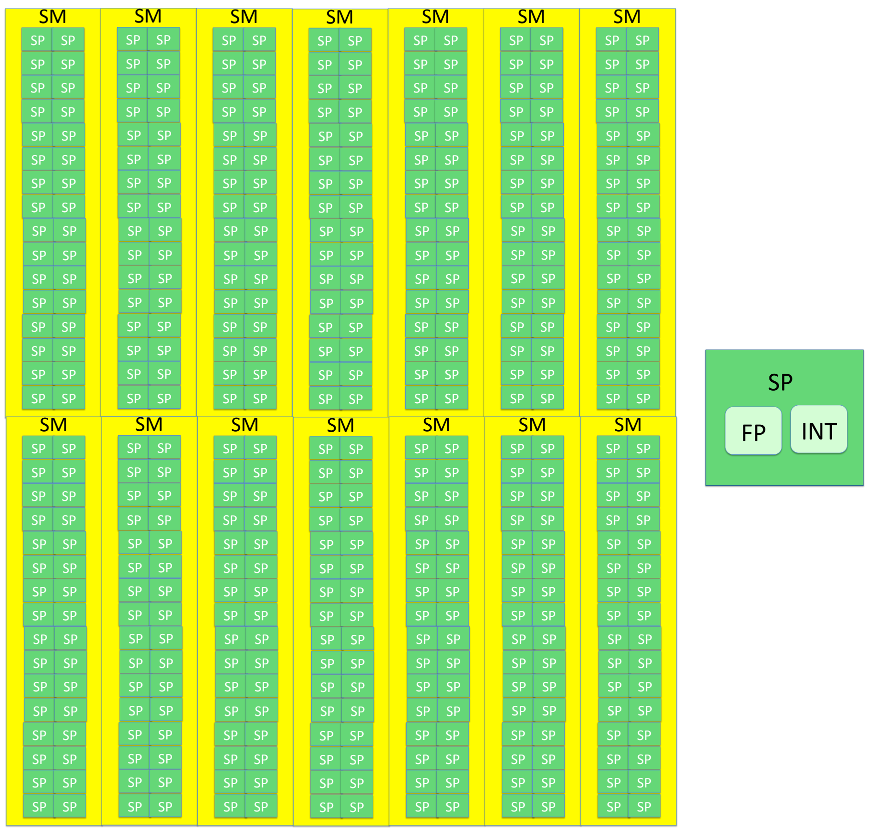


Figure 2.7: Fermi based hardware architecture of Tesla M 2050, with 14 SMs containing 32 SPs where each core contains a floating point unit(FP) and an Integer unit(INT) [27]

dependent threads. The CUDA toolkit contains a CUDA version of the BLAS library called CUBLAS for linear algebra computations and a CUDA version of FFT called CUFFT for Fourier transform. Also a set of very efficient numerical algorithms is available in the CUDA Software Development Kit (SDK).

2.6 Memory Model

Memory management and optimization is one of the more complicated aspects of CUDA. Developers need to fully comprehend different features of all the memory spaces on the device. CUDA devices are equipped with different memory spaces with different features, which make them suitable for specific usage. These memories are shown in Figure 2.8 and are described below:

- *Global memory*: off-chip DRAM memory where all the threads in the grid have access to. The host also has read/write access to this memory. For this reason, Global memory is used for communication between host and device. Global memory has larger capacity and higher access latency.
- Local memory: off-chip memory and private to each thread. Accessing local memory takes as much time as accessing global memory since its not cached. It is used whenever the available register space is not enough to hold the required variables;
- *Shared memory*: on-chip memory where all the threads in each block have access to. This memory is faster than local and global memory;
- *Constant memory and Texture memory*: read-only cache memory, all the threads in the grid can read these two memories;
- *Registers*: each thread in every block has access to its own registers. As there is only very limited number of registers available to each thread, if more registers is needed for a program, the data will be saved to local memory. Accessing registers might be delayed when reading/writing the same address at once [24].

Table 2.3 shows memory capacities for three different CUDA-enabled GPU cards. As shown in this table, global memory is the largest memory space and the only memory that can be accessed from CPU. Hence, using the C runtime for CUDA, developers can allocate, deallocate, copy data to the global memory, as well as transferring data back to host memory. Limited amount of available registers per thread in CUDA, limits the number of threads per

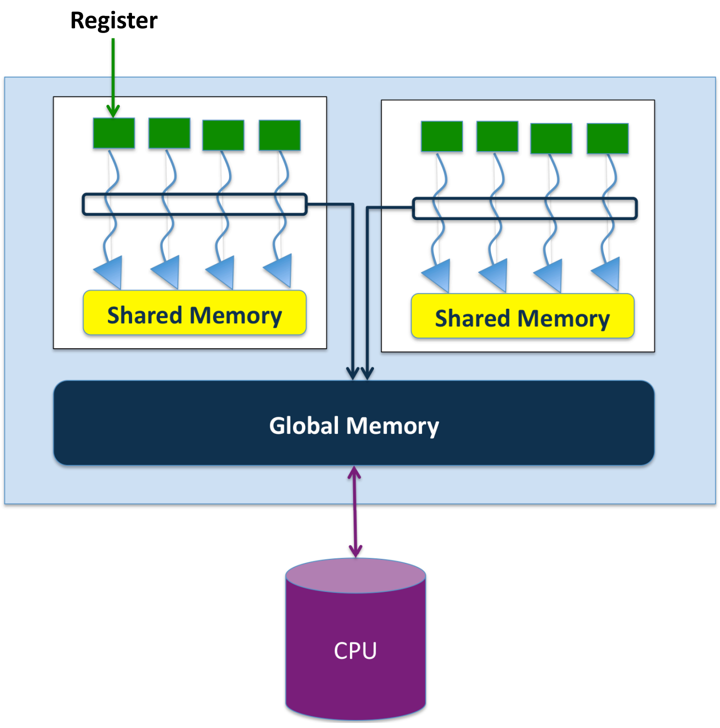


Figure 2.8: Cuda memory access [20]

multiprocessor. For instance, the number of 32 bit registers per streaming multiprocessor in Tesla C 1060 is 16K, which adds up to 128K register for the device. This means if invoking 768 threads in an SM, 20 registers will be available to each thread. Increasing the amount of registers available to each thread, means decreasing the number of threads per SM. The amount of shared memory assigned to each block also limits the number of threads [29].

In the CUDA programming interface, it is desired to limit the host and device data transfer to absolute minimum due to the low bandwidth between host and device memory. This in turn might lead to execution of parts of the application on GPU that will not speedup the process, but still the cost of data transfer back to CPU for performance will be higher. On this platform, developers are highly encouraged to make use of the faster shared memory instead of the global memory inside the kernel. Since shared memory is smaller than the global memory, this can be done by dividing data into subsets that can fit into shared memory and can be processed independently [24, 29].

GPU	**32-bit Registers (per SM)**	**Shared Memory (per SM)**	**Global Memory**	**Constant Memory**
Quadro FX 570	8K	16KB	256MB	64KB
Tesla C 1060	16K	16KB	4GB	64KB
Tesla M 2050	32K	48KB	3GB	64KB

Table 2.3: Comparison of memory capacities on different GPUs [20, 25, 26]

It should be noted that the highest memory bandwidth while accessing device's global memory can be reached using *coalesced memory access*. This can be achieved by arranging the threads to access many consecutive memory locations together. This way all the threads that belong to the same warp, can access consecutive memory locations using a minimal number of transactions [29].

With regards to the shared memory, it is important to know that it is divided into memory modules of equal size called *banks*. The shared memory banks can be accessed simultaneously by different threads, but a request for the same memory bank by two different threads cannot be processed at once which will lead to serial access of the memory [24].

2.7 Application Development on CUDA

Finally, it should be mentioned that certain parts of applications can run on CUDA-enabled GPUs efficiently and the following should be considered when making a decision on which parts to run on a GPU [20, 24, 29, 30]:

- *Data-parallel* applications, which perform arithmetic operations using a large number of lightweight threads on large data sets, hide global memory latency and work well on CUDA.

- Better use of on-chip registers and shared memory reduces bandwidth usage. It should be noted that shared memory and registers have both limited capacity and are divided between thread blocks.

- Control flow divergence within warps slows down performance. It is better to avoid divergence by reorganizing threads.

- Coalesced memory access pattern leads to coherent memory management for CUDA applications.

- Data transfer between host memory and CUDA's global memory is costly. As a result, the cost of executing the operations on CPU should be higher than the cost of CPU-GPU data transfer. In other words, executing a small number of operations using small number of threads wont provide any performance advantages.

- Those parts of the code that are serial and cannot be parallelised should remain on the CPU and only execution of parallelised algorithms should be transferred to GPU.

A group of Berkeley researchers specified the numerical domain in which parallel architectures should perform well in comparison to serial architectures [31]. They presented 13 classes of algorithmic methods called *dwarves*, where each class applies to similar computation and communication of numerical data. Currently there is ongoing work on most classes of the dwarves among which Fast Fourier Transform is covered in [32], N-Body in [33] and Monte Carlo in [34].

A different subset of dwarves is also studied by Che et al in [35] where he has performed a quantitative comparison of GPU/CPU performance on structured grid, unstructured grid, combinational logic, dynamic programming and dense linear algebra. This comparison is made on an NVidia GeForce GPU and a dual-core hyper-threaded CPU. The first comparison for structured grid is made on SRAD method, which is used for noise reduction on ultrasound images based on partial differential equations. The solution includes a reduction, followed by an update of each data elements using its neighboring elements and then update of each element using its north and west neighbors on a 2D matrix. It is reported in this paper that for a 2048×2048 data size, the CUDA implementation is 17 times faster than the single-thread CPU version and 5 times faster than the hyper-threaded CPU version.

Another comparison is focused on combinational logic and includes an encryption and decryption algorithm called DES, which involves a number of bit-level permutations. The sequential version of the algorithm includes conditional statements which leads to control flow divergence in the CUDA version and slows down the execution significantly. This problem has been overcome by using look-up tables. Che et al in [35], reports speed-up of $12\times$ over the hyper-threaded CPU implementation and $37\times$ over a single threaded CPU implementation for a problem of size 2^{18}. Overall, six different data parallel algorithms are addressed in [35]. Implementation of all these algorithms shows impressive speed-up in comparison to both single threaded and hyper threaded CPU implementations.

Chapter 3

Femtosecond Laser Pulse in Transparent Materials

This chapter presents the fundamental background of the physical problem that is numerically solved using the GPU hardware platform. In this chapter, all the linear and non-linear effects that are present during ultra-short pulse propagation in bulk transparent material are explained and moreover a mathematical model is presented to describe these effects.

3.0.1 Ultra-short Pulse Lasers

There has been extensive advances in ultra-short pulse lasers starting with De Maria's work in [36], for electromagnetic pulses with time duration in picosecond range. Since then, the peak intensity of the ultra-short pulses has increased due to the concentration of energy in a short time frame, leading to better performance of ultra-fast lasers in femtoseconds and attoseconds range. These lasers are used for various applications according to their unique features that are described below:

- **The ultra short duration** of these lasers provides fast temporal resolution, making it possible to capture extremely fast processes or fast moving objects such as electrons and molecules accurately [37–39];
- **The high repetition rate** alongside **the high average power** of ultrashort lasers is used in Optical clocks to achieve high clock rates on microprocessors [37];
- The **ultra broad spectrum** in ultrashort lasers provides spatial resolution for *Optical Coherence Tomography*(OCT), where the short coherence length of these lasers allows cross sectional 3D imaging of soft tissues.
- **The ultra high peak intensity** of these lasers is very effective for fabricating micrometer-structures or micro-machining in solid materials through *ablation*. Here ablation causes the direct change of material from solid state to gas, which is achieved without increase in the temperature. The non-thermal phase transition from solid to gas using ultra-short lasers is proven to be useful in various fields(e.g. medical and surgical applications) [37,40].

Non-linear propagation of these ultra-short lasers in materials and also the laser-matter interactions have been subject to several studies [41–44]. Generally, light passing through a transparent material does not result in changes in the material or the light itself. However, when a high intensity femtosecond laser pulse passes through transparent material, it results in changes both in the material and the light interaction. In other words, when material is exposed to the high power and intensity of the ultra-short pulse lasers, it responds in a non-linear way. As a result, several fundamental non-linear mechanisms effect the interaction of the light with the material [39].

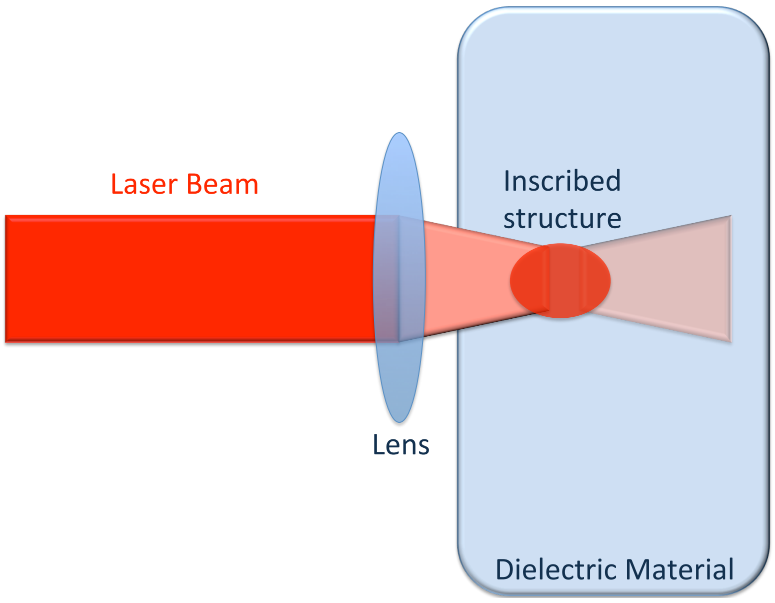


Figure 3.1: Micro-fabrication of dielectric material, where laser pulse enters from left to right

In this process both laser and material parameters affect the energy absorption and the result of the micro-machining (machining micro structures on surface or bulk of materials). These parameters include energy and duration of the pulse, pulse repetition rate and wavelength, the band-gap and thermal properties of the material. The high intensity of the pulse causes non-linear absorption in the material and the short duration of the pulse causes the energy to be absorbed by the electrons before any thermal effect [45].

In the world of photonics, with the technical advances in ultra-short pulse lasers [41, 42, 44], study of pulse propagation in different materials is proven to be essential. It should be mentioned here that laser light in this context is a laser beam in space and a laser pulse in time. Hence, it is limited to space and time and its called a bullet. When femtosecond bullets, with high peak intensity, propagate through transparent media (e.g. glass), self-focusing increases the intensity. This in turn, leads to plasma generation and creation of permanent micro-fabrication of solid materials. Typical applications of this phenomenon are femtosecond (fs) laser inscription for micro-fabrication and micro-machining.

The purpose of this chapter is to describe the linear and nonlinear phenomena which effect micro-machining of solid materials using ultra-short lasers. This chapter also presents a mathematical model for these phenomena based on the wave equation. The main focus is on the use of femtosecond pulse to induce microstructures and to fabricate bulk transparent material where a non-linear absorption regime transfers laser energy into the material and starts the process of electron ionization.

Fabrication of transparent materials using femtoseconds laser has been studied extensively for over a decade. These studies include both fabrication of waveguides inside glass and modification of refractive index inside transparent solids using laser systems. Fig. 3.1 shows micro-fabrication of transparent material (by focusing the laser beam into the material using a microscope object) [45].

3.1 Pulse Propagation

The narrow bandwidth wave propagation in envelope approximation can be described using the non-linear wave equation in the form of the Generalised Non-Linear Schrödinger Equation (GNLSE) which is a generic mathematical model. This section is focused on a numerical solution for a specific model, which describes femtosecond laser pulse propagation in transparent media. In this model, GNLSE is coupled to the Drude model of plasma resulting from multi-photon and avalanche ionization processes [9,10,46]. However this approach can be extended to similar models.

Ultra-short laser pulse propagation in medium is best described by the derivation of electromagnetic wave equation from the system of Maxwell equations:

$$\nabla \times \mathcal{E} = -\frac{\partial \mathcal{B}}{\partial t} \quad , \tag{3.1}$$

$$\nabla \times \mathcal{H} = \frac{\partial \mathcal{D}}{\partial t} + \mathcal{J} \quad , \tag{3.2}$$

$$\nabla \cdot \mathcal{D} = \rho \quad , \tag{3.3}$$

$$\nabla \cdot \mathcal{B} = 0 \quad . \tag{3.4}$$

In this system, $\mathcal{E}$ and $\mathcal{H}$ represent the electric and magnetic field vectors. Subsequently, $\mathcal{D}$ and $\mathcal{B}$ are the electric and magnetic flux densities. Source of the electromagnetic field in this system is presented by the electron current density vector $\mathcal{J}$ and the electric concentration ρ [47].

Propagation of the electric and magnetic fields in the medium creates densities $\mathcal{D}$ and $\mathcal{B}$ which can be stated through the following equations:

$$\mathcal{D} = \varepsilon_0 \mathcal{E} + \mathcal{P} \quad , \tag{3.5}$$

$$\mathcal{B} = \mu_0 \mathcal{H} + \mathcal{M} \quad , \tag{3.6}$$

where ε_0 and μ_0 are the vacuum permittivity and permeability respectively and $\mathcal{P}$ and $\mathcal{M}$ stand for the induced electric and magnetic polarisation (for dielectric materials which are nonmagnetic, $\mathcal{M} = 0$).

To obtain the wave equation that describes ultra-short laser pulse propagation, Eq. 3.6 is substituted in Eq. 3.1. Then the rotation or curl of the result is taken as shown below:

$$\nabla \times \nabla \times \mathcal{E} = -\mu_0 \frac{\partial \nabla \times \mathcal{H}}{\partial t} \quad . \tag{3.7}$$

Subsequently substituting Eqs. 3.2 and 3.5 in Eq. 3.7, results in the following, where c is the speed of light in the vacuum:

$$\triangledown \times \triangledown \times \mathcal{E} = -\frac{1}{c^2}\frac{\partial^2 \mathcal{E}}{\partial t^2} - \mu_0 \left(\frac{\partial^2 \mathcal{P}}{\partial t^2} + \frac{\partial \mathcal{J}}{\partial t} \right), \quad \mu_0 \varepsilon_0 = \frac{1}{c^2} \quad . \tag{3.8}$$

Here, relation $\triangledown \times \triangledown \times \mathcal{E} \equiv \triangledown(\triangledown.\mathcal{E}) - \triangledown^2 \mathcal{E}$ can be used, where $\triangledown.\mathcal{E} = 0$. This is due to the fact that in the charge neutral media there is equal amount of free electron density ρ_e and electron hole density ρ_h ($\triangledown.\mathcal{D} = \rho = \rho_e - \rho_h = 0$). As a result, Eq. 3.8 can be written as the following:

$$\triangledown^2 \mathcal{E} - \frac{1}{c^2}\frac{\partial^2 \mathcal{E}}{\partial t^2} = \mu_0 \left(\frac{\partial^2 \mathcal{P}}{\partial t^2} + \frac{\partial \mathcal{J}}{\partial t} \right), \quad \mu_0 \varepsilon_0 = \frac{1}{c^2} \quad . \tag{3.9}$$

Here the Laplacian operator $\triangledown^2 = \triangle = \partial^2/\partial x^2 + \partial^2/\partial y^2 + \partial^2/\partial z^2$ is used to expand the equation in x, y and z direction. However, since electric wave propagates in one direction in space and time, only the z divergence is considered and the x and y divergence are substituted with $\Delta_\perp$, where $\Delta_\perp = \partial^2/\partial x^2 + \partial^2/\partial y^2$. This will result in the following equation in which $\Delta_\perp$ represents transverse diffraction of the laser pulse,

$$\frac{\partial^2}{\partial z^2}\mathcal{E} + \Delta_\perp \mathcal{E} - \frac{1}{c^2}\frac{\partial^2 \mathcal{E}}{\partial t^2} = \mu_0 \left(\frac{\partial^2 \mathcal{P}}{\partial t^2} + \frac{\partial \mathcal{J}}{\partial t} \right), \quad \mu_0 \varepsilon_0 = \frac{1}{c^2} \quad . \tag{3.10}$$

The next step will be writing polarisation $\mathcal{P} = \mathcal{P}_{\mathcal{L}} + \mathcal{P}_{\mathcal{NL}}$ in terms of the electric field $\mathcal{E}$. Here polarisation is decomposed into a linear ($\mathcal{P}_{\mathcal{L}}$) and non-linear ($\mathcal{P}_{\mathcal{NL}}$) part. Polarisation for isotropic, homogenous mediums can be written as the following power series:

$$\begin{aligned}\hat{\mathcal{P}} &= \hat{\mathcal{P}}^{(1)}(r,\omega)+\hat{\mathcal{P}}^{(3)}(r,\omega)+\dots \quad , \\ &= \varepsilon_0\chi^{(1)}(\omega)\hat{\mathcal{E}}(r,\omega)+\varepsilon_0\chi^{(3)}(\omega)|\hat{\mathcal{E}}(r,\omega)|^2\hat{\mathcal{E}}(r,\omega)+\dots \quad , \qquad (3.11)\end{aligned}$$

where ω is the angular frequency of light and χ is the electric susceptibility. This equations shows that polarization at position r is directly related to electric field at point r. It should be noted that glass is amorphous and has inversion symmetry which results in elimination of all the even terms from the polarisation equation above.

Here the following two expressions are used to define the Fourier transform and the inverse Fourier transform,

$$\hat{E}(r,\omega) = \int_{-\infty}^{+\infty} E(r,t)e^{i\omega t}dt \quad , \qquad (3.12)$$

$$E(r,t) = \frac{1}{2\pi}\int_{-\infty}^{+\infty} \hat{E}(r,\omega)e^{-i\omega t}d\omega \quad . \qquad (3.13)$$

In this relation, the linear polarisation ($\mathcal{P}_L \equiv \mathcal{P}^{(1)}$) for linear materials is quantified by the first order electric susceptibility ($\chi^{(1)}$). For non-linear materials, the non-linear polarisation is quantified by higher orders of electric susceptibility. Also, the frequency dependent dielectric constant is quantified in terms of the electric susceptibility by the following relation [48],

$$\varepsilon(\omega) = \chi^{(1)}(\omega)+1 \quad . \qquad (3.14)$$

By Fourier transformation, Eq. 3.10, is written as follows:

$$\left(\partial_z^2 + \varepsilon(\omega)\frac{\omega^2}{c^2} + \Delta_\perp\right)\hat{\mathcal{E}}(r,\omega) = -\mu_0\omega^2\hat{\mathcal{F}}_{NL} \quad , \tag{3.15}$$

where $\hat{\mathcal{F}}_{NL} \equiv \hat{\mathcal{P}}_{NL} + i\hat{\mathcal{J}}/\omega$ presents the non-linear effects caused due to non-linear polarisation and electron current density $\mathcal{J}$.

Propagation of ultra-short pulse in transparent media is effected by a number of linear and non-linear effects. Theses effects consist of dispersion, Kerr non-linearity, plasma absorption and defocusing, multi-photon absorption, avalanche and plasma ionization. It should be noted that multi-photon absorption results in creation of plasma by multi-photon and avalanche ionization. This, in turn, leads to resistive absorption of laser light and it's defocusing. These effects together with Kerr effect, diffraction and dispersion make the pulse propagation a very complex phenomenon developing over a range of scales in space and time. Detailed description of these effects is given in the following sections.

3.1.1 Dispersion

The dispersion effect describes the dependence of the phase velocity of light on frequency. Phase velocity of the wave in a medium is calculated by $v = c/n$, c being the speed of light in vacuum and n the refractive index of the medium. Phase velocity is the speed at which phase of different frequencies of light will travel. This phenomenon is caused due to wavelength dependence of the refractive index. The propagation rate of the changes in the amplitude is called the *Group Velocity*, which can be written in terms of frequency:

$$v_g = k'^{-1} = \left(\frac{\partial k(\omega)}{\partial\omega}\right)^{-1} \quad \text{where} \quad k = n(\omega)\frac{\omega}{c} \quad , \tag{3.16}$$

where k presents the wave vector. Frequency dependence of this function means that different frequencies of light will travel with different speed. This phenomenon is called the *Group Velocity Dispersion* (GVD) and quantified by $d = \partial^2 k(\omega)/\partial\omega^2$.

To simplify deriving the dispersion effect, only the linear terms on the left hand side of Eq. 3.15 are considered. In this equation the dielectric constant ($\varepsilon(\omega)$) is a complex value, where its real part is related to the refractive index $n(\omega)$ and its imaginary part is related to absorption coefficient. For materials with low optical losses, the imaginary part is negligible and the dielectric constant $\varepsilon(\omega)$ can be replaced by $n^2(\omega)$ [47]:

$$\begin{aligned}&\left(\frac{\partial^2 \hat{\mathcal{E}}}{\partial z^2} + n^2(\omega)\frac{\omega^2}{c^2} + \Delta_\perp\right)\hat{\mathcal{E}}(r,\omega)\\ &= \left(\frac{\partial^2 \hat{\mathcal{E}}}{\partial z^2} + k^2(\omega) + \Delta_\perp\right)\hat{\mathcal{E}}(r,\omega) \quad . \qquad (3.17)\end{aligned}$$

This equation, currently only showing a trivial transformation of Eq. 3.15, shows that for linear propagation of the laser pulse, the refractive index, wave vector and frequency are connected. Here the $k^2(\omega)$ can be expanded using the Taylor series into the following:

$$\begin{aligned}k(\omega) &= k(\omega_0) + \left.\frac{\partial k}{\partial\omega}\right|_{\omega_0}(\omega-\omega_0) + \frac{1}{2}\left.\frac{\partial^2 k}{\partial\omega^2}\right|_{\omega_0}(\omega-\omega_0)^2 + \ldots \quad ,\\ k^2(\omega) &= k^2(\omega_0) + \left.\frac{\partial k^2}{\partial\omega}\right|_{\omega_0}(\omega-\omega_0) + \frac{1}{2}\left.\frac{\partial^2 k^2}{\partial\omega^2}\right|_{\omega_0}(\omega-\omega_0)^2 + \ldots \quad . \qquad (3.18)\end{aligned}$$

The next step is to transfer Eq. 3.17 back to time domain, where $-i(\omega-\omega_0) \to \partial/\partial t$ and $-(\omega-\omega_0)^2 \to \partial^2/\partial t^2$, as shown below:

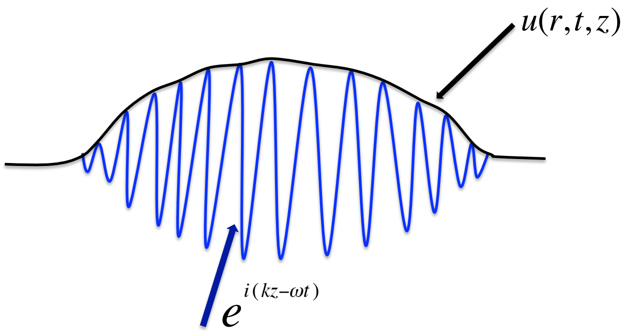


Figure 3.2: Slowly varying amplitude of the electric field

$$\frac{\partial^2 \mathcal{E}}{\partial z^2} + \Delta_\perp \mathcal{E} + k_0^2 \mathcal{E} - 2ik_0 v_g^{-1} \frac{\partial}{\partial t} \mathcal{E} - \frac{1}{2} d \frac{\partial^2}{\partial t^2} \hat{\mathcal{E}} = 0 \quad , \tag{3.19}$$

where Group Velocity is defined by $v_g = k'^{-1} = \partial\omega/\partial k$ and Group Velocity Dispersion by $d = k'' = \partial^2 k/\partial\omega^2$.

On the other hand, E can be described in terms of the envelope amplitude of electric field (u) (shown in Fig. 3.2):

$$\mathcal{E} = u(r,t,z) e^{i(kz-\omega t)} \quad , \tag{3.20}$$

where $e^{i(kz-\omega t)}$ describes quick oscillation with angular frequency ω and the wave vector k, where $\partial/\partial z = ik + \partial/\partial z$ and $\partial/\partial t = i\omega + \partial/\partial t$. Second derivative of Equation 3.20 ($\partial^2 E/\partial z^2 = [\partial^2 u/\partial z^2 + 2ik_0(\partial u/\partial z) - k_0^2 u] e^{i(kz-\omega t)}$) is substituted in Eq. 3.19, resulting in the following:

$$\frac{\partial^2 u}{\partial z^2} + 2ik_0 \left(\frac{\partial u}{\partial z} \right) + \Delta_\perp u - 2ik_0 v_g^{-1} \frac{\partial}{\partial t} u - \frac{1}{2} d \partial_{tt} u = 0 \quad . \tag{3.21}$$

In this equation, u presents the slowly varying envelope of the electric field moving with group velocity v_g, in the moving frame of coordinate shown below:

$$t \rightarrow t' - z'/v_g \quad , \tag{3.22}$$

$$z \rightarrow z' \quad . \tag{3.23}$$

Fig. 3.2 illustrates the small rate of change in the amplitude of the electric field where $|\partial u/\partial z| \ll |ikzu|$. This results in a negligible value for $\partial^2 u/\partial z^2$. Furthermore the following expansion can be made based on 3.23:

$$\begin{aligned} \frac{\partial u}{\partial z} &= \frac{\partial u}{\partial z'}\frac{\partial z'}{\partial z} + \frac{\partial u}{\partial t'}\frac{\partial t'}{\partial z} = \frac{\partial u}{\partial z'} + \frac{1}{v_g}\frac{\partial u}{\partial t'} \quad , \\ \frac{\partial u}{\partial t} &= \frac{\partial u}{\partial t'}\frac{\partial t'}{\partial t} + \frac{\partial u}{\partial z'}\frac{\partial z'}{\partial z'} = \frac{\partial u}{\partial t'} \quad . \end{aligned}$$

This in turn further simplifies Eq. 3.21 into the following:

$$2ik_0\left(\frac{\partial u}{\partial z}\right) - \frac{1}{2}d\partial_{tt}u + \Delta_{\perp}u = 0 \quad . \tag{3.24}$$

As mentioned earlier, d is the GVD parameter which will calculate dispersion's sign and magnitude.

3.1.2 Optical Kerr Effect

This effect describes the increase in the local refractive index of the material caused by the high intensity of the laser pulse. This effect is defined by $n = n_0 + n_2 \mid u \mid^2$, where n_2 is the non-linear index which changes the refractive index of the material proportional to intensity ($I = | u |^2$) [48,49].

The Kerr effect is also responsible for self-focusing, which is limited by plasma and multi-photon absorption. Positive value of n_2 for most materials means that the local refractive index increases with intensity variations. This means that the refractive index will have a bigger value at the center of the laser beam, acting as a lens to focus the beam. It should be noted that while the change in refractive index is intensity dependent, the effect of self-focusing is dependent on the peak power of the pulse [10,39].

The strength of self-focusing increases with the increase in the power of the laser pulse. However, when this value exceeds the critical power of self-focusing, the laser beam will collapse. The critical power (P_{cr}) of the laser pulse is calculated by the relation given below, where λ represents the wavelength of the laser [39]:

$$P_{cr} = \frac{3.77\lambda^2}{8\pi n_0 n_2} \quad . \tag{3.25}$$

Another effect that changes refractive index is defocusing. This effect causes a negative change of the refractive index, prevents more self-focusing and causes the catastrophic collapse of the beam. A more detailed description of this regime is given in the next section.

The optical Kerr effect is described by the third order non-linear polarisation ($\mathcal{P}_{\mathcal{NL}} = \mathcal{P}^{(3)}$),

$$\frac{\partial u}{\partial z} = ikn_2|u|^2u \text{ where } n_2 = \frac{\omega\chi^{(3)}}{2cn_0k} \quad . \tag{3.26}$$

As a result of this relation, the refractive index will change with the non-linear rate $n_2|u|^2$.

3.1.3 Plasma Absorption and Defocusing

Propagation of a femtosecond pulse in bulk transparent material will form an electron concentration. As this concentration grows, the electron density will become plasma. This plasma density will have two effects on the laser pulse, rapid absorption of energy and defocusing of the pulse. The first effect is due to the absorption of the laser energy and the latter is caused by the negative effect of the free electrons on refractive index. As refractive index of the plasma is lower than the surrounding material, it causes a defocusing effect on the pulse. As mentioned earlier in section 3.1.2, this effect prevents catastrophic collapse of the beam caused by self-focusing [39].

Plasma absorption and defocusing is derived using the Drude model of plasma, which describes acceleration of electrons in an electromagnetic field:

$$m_e \frac{\partial v_e}{\partial t} = -e\mathcal{E} - \frac{m_e}{\tau_c} v_e \quad , \tag{3.27}$$

where v_e is electron velocity, e is electron charge and finally m_e is electron mass. In this equation, collision causes changes in electron momentum which is described by $(m_e v_e)/\tau_c$. Here τ_c is the time taken between collisions for electrons to relax. Since collision causes reduction in electron momentum, it should be subtracted from equation of motion as shown above. Moreover, a linear relationship between the free electrons and the induced current density $\mathcal{J}$ exists as shown below:

$$\mathcal{J} = -e\rho v_e \quad . \tag{3.28}$$

This relation is used for solution of $\partial J/\partial t$ in Eq. 3.10, to derive plasma absorption and defocusing effect. This is achieved by writing Eq. 3.27 to oscillate at $-i\omega$ to achieve the following:

$$v_e = -\frac{e\mathcal{E}}{m_e}\tau_c(1 - i\omega\tau_c)^{-1} \quad . \tag{3.29}$$

Substituting Eq. 3.29 into 3.28 will provide current density J in terms of the electric field:

$$\mathcal{J} = -e^2\frac{\rho\mathcal{E}}{m_e}\tau_c\frac{(1 + i\omega\tau_c)}{(1 + \omega^2\tau_c^2)} \quad . \tag{3.30}$$

This in turn will lead to derivation of the plasma absorption and defocusing as shown below:

$$\frac{\partial u}{\partial z} = \frac{\sigma}{2}(1 + i\omega\tau)\rho u \text{ where } \sigma = \frac{\mu_0 e^2}{m_e k(1 + \omega^2\tau^2)} \quad . \tag{3.31}$$

3.1.4 Multi-photon Absorption

Individual photons lack enough energy for electron excitation in transparent material. However, absorption of multiple photons at once can cause excitation of the electrons as illustrated in Fig. 3.3. Multi-Photon Absorption (MPA) describes the probability of absorption of a number (K) of photons simultaneously. Here the total energy of the K photons matches the energy needed to excite electrons from the valence band to the conduction band [10,46].

The MPA process results in changes in the number of electron density and as a result this term is placed on the right hand side of current density equation. Fenq et al in [49], presents the following relation to describe MPA:

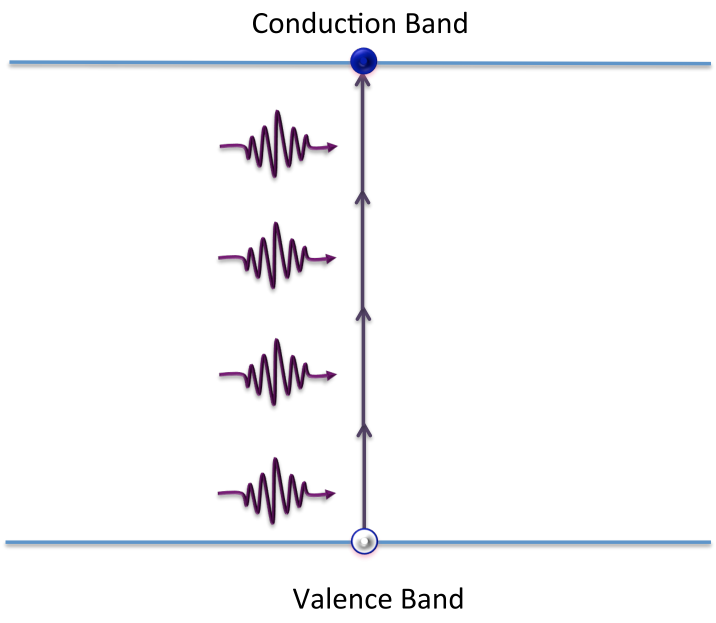


Figure 3.3: Multi-photon absorption regime where an electron is excited from valence band to the conduction band.

$$\frac{\partial u}{\partial z} = i\frac{\beta^{(K)}}{2}|E|^{2(K-1)}E \quad , \tag{3.32}$$

where $\beta^{(K)}$ controls the K-photon absorption. This equation shows that the rate of K-photon absorption is proportional to K. The order of photon absorption is simply estimated by $K = (\mathrm{E_g}/\bar{\mathrm{h}}\omega)$, which indicates the smallest number of photons with energy $\bar{h}\omega$ that is required to overcome E_g. Here, E_g is the energy to free an electron [10,49].

3.1.5 Multi-photon and Avalanche Ionisation

Multi-photon Ionisation

This effect happens due to ionization of the electrons by high intensity laser field. MPI can happen through two different mechanisms based on the frequency and intensity of the laser: *multi-photon ionisation* (MPI) and *tunneling ionisation*. Illustration of these two regimes is

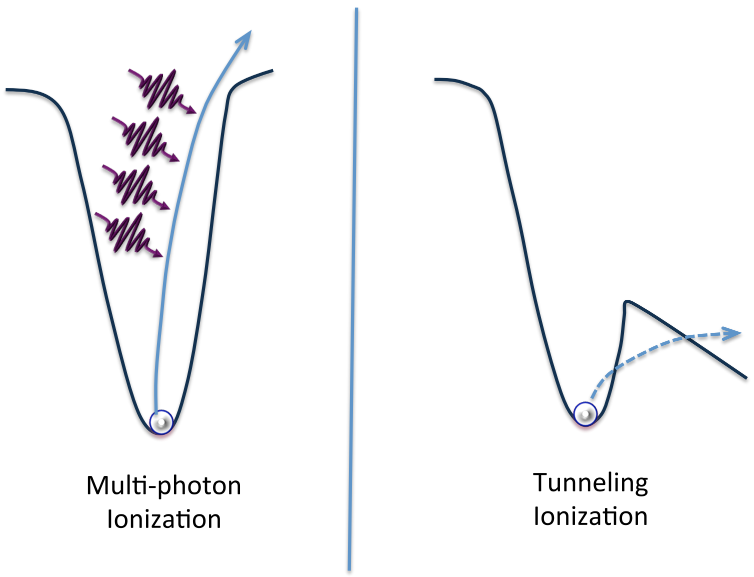


Figure 3.4: Photon Ionization for different values of the Keldysh parameter γ; Multi-photon ionization for $\gamma > 1.5$ and tunneling ionization for smaller values of γ

given in Fig. 3.4.

For low frequency and high intensity lasers, the tunneling mechanism causes the electron ionization. In this regime, the strong electric field causes the electrons to tunnel their way through the atom barrier and become free. For high frequency lasers, multi-photon ionization happens when a bound electron is hit by a number of photons simultaneously. MPI occurs when the right number of photons is absorbed for successful ionization. This means that the number of absorbed photons multiplied by the photon's energy should be bigger than the band gap [39].

Multi-photon and tunneling ionization can be distinguished by the Keldysh parameter $\gamma = \omega/e(mcn\varepsilon_0 E_g/I)^{1/2}$ [50]. For $\gamma > 1.5$ multi-photon ionization happens and for smaller values of γ, tunneling will happen. The value for this parameter in this thesis is greater than 1.8, hence multi-photon ionization is the dominant regime.

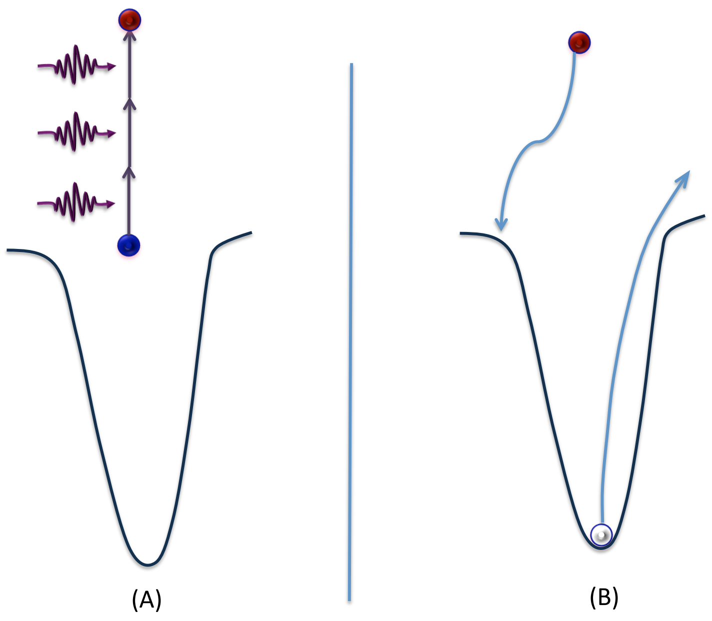


Figure 3.5: Avalanche ionization, (A) an initially free electron absorbing energy. (B) this electron impact ionizing another electron by collisional energy transfer.

Avalanche Ionization

Avalanche ionization describes the collisional energy transfer from free electrons high in the conduction band to bound electrons as illustrated in Fig. 3.5. In this regime, the free electrons (or seeds) continue to absorb energy and free bound electrons by impact ionization. This in turn will result in having two electrons in the minimum conduction band that will subsequently act as seeds. These seeds absorb multiple photons one after another and promote to higher energies in the conduction band, which will result in further impact ionization of electrons. It should be mentioned that first multi-photon ionization causes generation of free seed electrons and then avalanche ionization starts the impact ionization regime. Fig. 3.5 illustrates this process [39].

The Drude model of plasma (Eq. 3.27 and Eq. 3.28) is employed to describe the evolution of plasma in bulk material. For the purpose of this work, the following model presented in [49] is used. This model describes multi-photon and avalanche ionization as the main sources of plasma in electromagnetic field ($\partial\rho/\partial t$),

$$i\frac{\partial\rho}{\partial t} = \frac{1}{n^2}\frac{\sigma_{bs}}{E_g}\rho|u|^2 + \frac{\beta^{(K)}}{k\hbar\omega}|u|^{2K} \quad , \tag{3.33}$$

where σ_{bs} is the cross-section for the electron neutral inverse Bremsstrahlung, E_g is the ionization energy and $\beta^{(K)}$ controls the K-photon absorption.

3.2 Non-local Non-Linear Schrödinger Equation

As mentioned in the previous sections, propagation of ultra-short pulse in transparent media is effected by a number of linear and non-linear effects including dispersion, Kerr non-linearity, plasma absorption and defocusing and multi-photon absorption. The combination of all these effects form the extended non-linear Schrödinger equation (Eq. 3.34) which is presented based on the mathematical model used by Fenq et al in [49]. This model was originally studied by Feit and Fleck in [51] and later on generalised in [49] to add the GVD and the MPA regimes,

$$\partial_z u + \frac{1}{2k}\Delta_\perp u - \frac{1}{2}d\partial_{tt}u + k_0 n_2|u|^2 u = -\frac{i\sigma}{2}(1+i\omega\tau)\rho u - i\frac{\beta}{2}|u|^{2(K-1)}u \quad . \tag{3.34}$$

To simplify this analysis, Eqs. 3.34 and 3.33 are rescaled to a dimensionless format. Hence, the following dimensionless variables are introduced,

$$t \to t_p t' \ , \ \rho \to \rho_{BD}\rho' \ , \tag{3.35}$$

where ρ_{BD} represents plasma breakdown intensity and t_p presents the pulse width. Furthermore, intensity threshold of multi-photon ionization (I_{MPA}) is used to normalise the electric field. This is due to the fact that only intensities higher than the MPA intensity threshold result in seeding electrons. Since it is the generation of plasma which results in micro-machining of material, the numerical modeling is set to start when $I \sim I_{MPA}$.

To achieve a dimensionless model normalised with I_{MPA} the following is derived from Eq. 3.33:

$$\begin{aligned}
\frac{\rho_{BD}}{t_p}\frac{\partial\rho'}{\partial t'} &= \frac{\sigma_{bs}\rho_{BD}}{n_b^2 E_g}\rho'|u|^2 + \frac{\beta^{(K)}}{k\hbar\omega}|u|^{2K} \quad , \\
\frac{\partial\rho'}{\partial t'} &= \frac{\sigma_{bs}\rho_{BD}t_p}{n_b^2 E_g \rho_{BD}} I_{MPA}\rho' \frac{|u|^2}{I_{MPA}} + \frac{\beta^{(K)} I_{MPA}^K t_p}{k\hbar\omega\rho_{BD}} \frac{|u|^{2K}}{I_{MPA}^K} \quad .
\end{aligned}$$

It is desired to set the intensity of the electric field at the point where plasma ionization rate becomes steep. To achieve this MPA intensity threshold is set to the following:

$$I_{MPA} = \left(\frac{k\hbar\omega\rho_{BD}}{\beta^{(K)}t_p}\right)^{\frac{1}{K}} \quad . \tag{3.36}$$

As a result, a dimensionless model is obtained where $u' = u/\sqrt{I_{MPA}}$ and $u_0 = \sqrt{I_{MPA}}$:

$$\boxed{\partial_z u - i\kappa\Delta_\perp u - i\sigma|u|^2 u = -id\partial_{tt}u - \gamma(1+i\omega\tau)\rho u - \mu|u|^{2(K-1)}u \quad ,} \tag{3.37}$$

$$\boxed{\frac{\partial\rho}{\partial t} = \nu\rho \mid u \mid^2 + \left(\frac{\mid u \mid}{u_0}\right)^{2K} \quad .} \tag{3.38}$$

Eq. 3.37 describes an envelope amplitude u of the laser wave coupled to an equation describing concentration of plasma carriers ρ along the z-axis. These two equation together form the Non-local Non-linear Schrödinger Equation, called here after the NNLSE and are employed in the following chapters to present a numerical scheme for the above mentioned physical phenomena.

The terms in the left hand-side of Eq. 3.37 describe the effects of beam diffraction and Kerr self-refraction. These terms alone form the well-known GNLSE. Kerr effect leads to a singular catastrophic self-focusing which is ultimately arrested by effects of dispersion, multi-photon absorption, plasma absorption and defocusing. The first term in the right hand-side of Eq. 3.37 describes dispersion. The second term on the right hand side $\gamma(1+i\omega\tau)\rho u$ comprises effects of plasma absorption and defocusing and the final term in this equation describes MPA [10].

As for the Eq. 3.38, the first term in the right hand side describes avalanche ionization and the second term describes multi-photon ionization.

Chapter 4

Numerical Scheme for Modeling Femtosecond Pulse Propagation

The high intensity of the laser field used in this research results in complex processes of material ionization and evolution of plasma that comprises cascaded multi-photon absorption, plasma absorption, and defocusing. As mentioned in Chapter 3, a combination of these non-linear effects results in an extremely complex mathematical model that is an extended version of the Generalised Non-linear Schrödinger Equation. This extended model, called the Non-local Non-linear Schrödinger Equation, is derived from coupling between the extended NLSE and the particle balance equation for plasma resulting from multi-photon and avalanche ionization processes.

NNLSE in Chapter 3 is presented in the form of Eqs. 3.37 and 3.38 and is a system of non-linear partial differential equations which cannot be solved analytically. Two main numerical approaches have been used for solution of GNLSE type equations, known as *Finite Difference* and *Pseudo-spectral* methods. The non-local interaction of wave and plasma makes it impossible to use a straightforward parallel implementation, typical for explicit finite difference methods. The pseudo-spectral methods are proven to be faster and in that category *Spilt-Step Fourier Method* (SSFM) is known to be the most efficient. It should be mentioned that the Finite Difference method uses paraxial approximation in time domain to solve the problem, whereas the SSFM uses spectral methods for a faster solution [47, 52].

In [53, 54], Taha and Ablowitz compared different numerical solutions for GNLSE including the SSFM and finite difference methods and stated the superior performance of the SSFM on CPUs. In [55], SSFM and the fourth-order Runge-Kutta method were implemented for parallel simulation of GNLSE using GPUs which indicated similar computation time for both methods.

The Split-Step Fourier method can be easily modified for different differential operators (e.g a more complex dispersion term) and it is universal regarding independent linear and non-linear operators. Due to the versatility of the SSFM, this method was chosen similar to that in [9, 49] to reduce NNLSE into a succession of small linear and non-linear steps by dividing the propagation.

Parallel solution of GNLSE-like systems have been discussed previously by Zoldi et al in [56]. In 2009, Hellerband et al used the same approach to implement a parallel solution for GNLSE on GPUs in [57, 58]. Both Zoldi and Hellerband used SSFM for large scale problems, where parallelisation of the Fast Fourier Transform (FFT) is addressed.

Operator splitting techniques address the numerical solution of GNLSE-type problems (i.e. partial differential equations comprising linear differential operator and local non-linear terms). This technique solves the GNLSE by a problem-specific succession of linear and non-linear operators that approximate the original equation [47, 52].

Solution of the linear problem, referred to as the *linear step*, is usually efficiently performed using spectral methods which often have efficient parallel numerical implementations such as FFT. Efficiency of the spectral methods results from reducing the solution to a discretised linear partial differential equation as a matrix inversion problem into a set of trivial uncoupled equations in the corresponding spectral domain. Similarly, the non-linear problem, or the *nonlinear step*, is usually already local i.e. already represents a set of uncoupled equations. Thus operator splitting makes possible efficient parallel implementations for equations with local non-linearities as shown e.g. in [56].

In this research we consider a more complicated model which describes ultra-short laser pulse propagation in transparent media (see e.g. [48]). A fundamental difference between relevant mathematical models and the NNLSE, is in the nature of the non-linear term which is non-local in time due to peculiar interaction of laser light with plasma.

This chapter addresses the problem of eliminating a numerical bottleneck associated with non-local nature of the non-linear terms in the wave equation. A conventional operator splitting technique is exploited to efficiently deal with the linear operator, while special attention is paid to constructing an efficient parallel method for the non-linear operator. Since the typical problem size of $10^4 \times 10^4$ fits to the GPU memory, well-developed existing numerical implementations are used such as an optimised off-the-shelf FFT.

4.1 Split-Step Method

The split-step method derives an approximate discretised solution by considering effects of linearity and non-linearity independently by dividing the problem into two sequential steps where in the first step non-linearity acts alone and the second step is for independent linear effect. These two steps are derived from Eq. 3.37 and denoted by L and N which schematically denote the linear and non-linear operators correspondingly [47]:

$$\partial_z u = Lu = -i\kappa\Delta_\perp u + id\partial_{tt}u \quad , \tag{4.1}$$

$$\partial_z u = Nu = -i\gamma(1+i\omega\tau)\rho u - \sigma|u|^2 u - i\mu|u|^{2(K-1)}u \, , \tag{4.2}$$

where the linear term L includes the dispersive and diffraction effects while the non-linear term N includes plasma absorption and defocusing, non-linear Kerr effect and MPA. The consecutive application of the linear and the non-linear operators after finite increment of the evolution variable z approximates the original system of differential equations:

$$u(z+\Delta z) = \exp\left(\frac{N}{2}\Delta z + L\Delta z + \frac{N}{2}\Delta z\right) u(z) \tag{4.3}$$

As stated in [47], here a symmetrised SSFM technique is used where the nonlinear effect is included in the middle of the step to reduce the leading error to third order in step size z. Using this technique, pulse propagates along the z-axis from z to $z+\Delta z$ by taking a small step $\Delta z/2$ to progress the non-linear term in the field. After that the linear term makes a full step of size Δz to bring the linear term further in the field. Consequently another half step $\Delta z/2$ is taken to advance the non-linear term. This process is repeated to propagate the laser pulse in the spatial coordinate. This mathematical approach is based on the fact that both linear and non-linear effects act concurrently and using a very small step size Δz, they can interplay as two separate linear and non-linear effect [49, 57].

The solving of the linear term can be preformed in the frequency domain where an efficient implementation of FFT is available. This approach implies a periodic boundary condition. It should be noted that since the laser pulse vanishes close to the boundaries while propagating in time, these boundaries do not cause any limitations on the problem. Meanwhile, the non-linear term is implemented in the time domain. Choosing the right value for the step size is important to produce reliable results [57, 59]. The method used in this research for calculation of the step size is described in detailed in section 4.3.

A detailed description of the solving method for the linear and non-linear term is given in the next sections.

4.1.1 Linear term

This section provides a parallel solving method for the linear term (Eq. 4.4) of the Split-Step Fourier method,

$$i\partial_z u + \kappa\Delta_\perp u - d\partial_{tt} u = 0 \quad . \tag{4.4}$$

Here the initial pulse is stored in a matrix of size $N_r \times N_t$ with N_t pointing along the time domain and N_r pointing along the radial domain as shown in Fig. 4.1.

Fast Fourier Transform is applied to the amplitude of the electric field (u) to solve Eq. 4.4 in the frequency domain. The transformation is applied to batches of size N_t for every radial position that will result in the following:

$$i\partial_z \tilde{u} + \kappa\Delta_\perp \tilde{u} + d\omega^2 \tilde{u} = 0 \quad \text{where} \quad \tilde{u} = F \cdot u \quad . \tag{4.5}$$

Subsequently, matrix transposition is used to rearrange the $N_r \times N_t$ matrix into a $N_t \times N_r$.

Eq. 4.5 is to be solved with the initial condition $u_0 = u(z = z_0)$. To solve this equation, the Laplacian $\Delta_\perp$ is written in polar coordinates as a radial diffraction [60]:

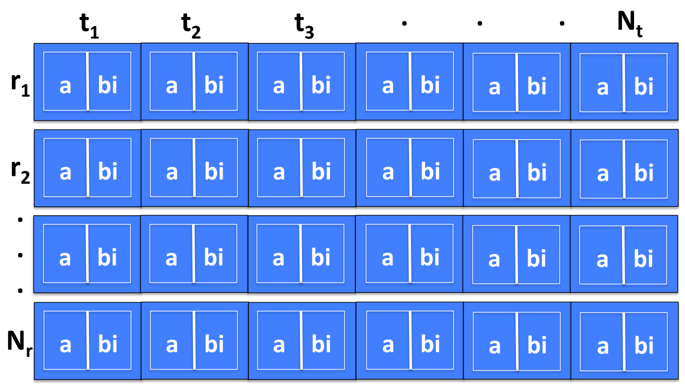


Figure 4.1: Initial data saved in a Matrix of size $N_r \times N_t$

.

$$\Delta_\perp u = \frac{\partial^2 u}{\partial r^2} + \frac{1}{r}\frac{\partial u}{\partial r} \quad . \tag{4.6}$$

Here Eq. 4.6 is discretized using the central difference scheme where $r_n = n \times \Delta r$ and expanded into the following equation:

$$\Delta_\perp u = \frac{u_{n+1} - 2u_n + u_{n-1}}{\Delta r^2} + \frac{1}{r_n}\frac{u_{n+1} - u_{n-1}}{2\Delta r} \quad . \tag{4.7}$$

The second term in Eq. 4.6 can result in a division by Zero problem, whenever $r = 0$. To overcome this problem, the L'Hopital rule is used to differentiate $\partial u / r \partial r$ as shown below [60]:

$$\frac{\frac{\partial}{\partial r}(\frac{\partial u}{\partial r})}{\frac{\partial}{\partial r} r} = \frac{\partial^2 u}{\partial r^2}|_{r=0} \quad . \tag{4.8}$$

This will result in the following for the laplacian $\Delta_\perp$:

$$\Delta_\perp u = \begin{cases} 2\dfrac{\partial^2 u}{\partial^2 r} = 4\left(\dfrac{u_1 - u_0}{\Delta r^2}\right) & \text{if } r = 0 \quad , \\ \dfrac{u_{n+1} - 2u_n + u_{n-1}}{\Delta r^2} + \dfrac{1}{r_n}\dfrac{u_{n+1} - u_{n-1}}{2\Delta r} & \text{otherwise} \quad . \end{cases} \tag{4.9}$$

To simplify the third term in the Eq. 4.4, variable $v = e^{id\omega^2 z}\hat{u}$ is defined. Substituting v in this equation results in the ω independent relation shown below:

$$i\partial_z v + \kappa\Delta_\perp v = 0 \quad . \tag{4.10}$$

The next step will be discretisation over z, which is achieved using the Crank Nicolson method, to derive the following equation [61]:

$$i\frac{v^{m+1} - v^m}{\Delta z} + \frac{1}{2}\kappa\Delta_\perp\left[v^{m+1} + v^m\right] = 0 \quad . \tag{4.11}$$

As described in [61], since the linear term is essentially a time-dependent equation, the Crank Nicolson method provides both stability and second order accuracy in space and time and results in a complex tridiagonal system by the following rearrangement,

$$\left[i + \frac{\Delta z}{2}\,\kappa\Delta_\perp\right]v^{m+1} = \left[i - \frac{\Delta z}{2}\,\kappa\Delta_\perp\right]v^m \quad , \tag{4.12}$$

where the transverse diffraction operator and the dispersion operators are diagonal and a tridiagonal solver can be used to calculate v^{m+1}. Eq. 4.13 shows that for very small values of step size Δz, there will not be a significant change in the amplitude of the electric field described by the linear effects ($v_{m+1} \longrightarrow v_m$ for $\Delta z \longrightarrow \varepsilon$):

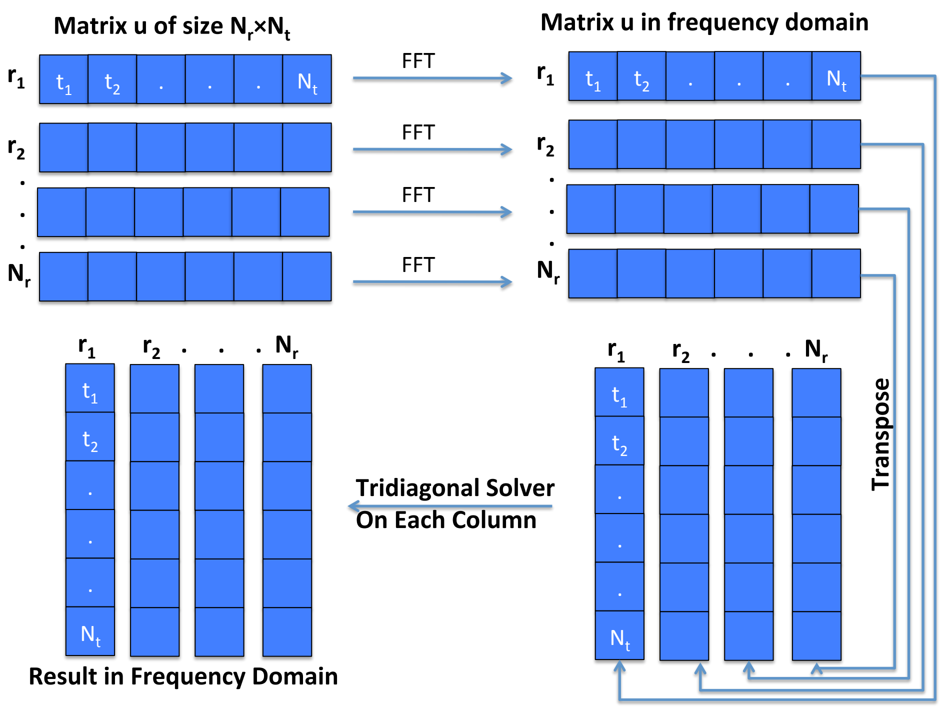


Figure 4.2: Solution of the Linear term in Frequency domain

$$v^{m+1} = \left[1 - i\,\frac{\Delta z}{2}\,D_v\right]^{-1}\left[1 + i\,\frac{\Delta z}{2}D_v\right]v^m \quad . \tag{4.13}$$

Fig. 4.2 shows all the steps to be taken for the solution of the linear term. It should be noted that to obtain the final result in time domain, a transpose operation to inverse the result back to $N_r \times N_t$ dimension and an inverse fast Fourier transform is necessary (these steps are shown in Fig. 4.3).

4.1.2 Non-Linear Term

This section presents a novel approach for numerical modeling of the non-linear part of the extended non-local non-linear Schrödinger equation in parallel (Eq. 4.14). While routines like Fast Fourier Transform and tridiagonal solver provide a satisfactory scalable parallel solution for numerical integration of the linear operator, the non-linear step is a convoluted

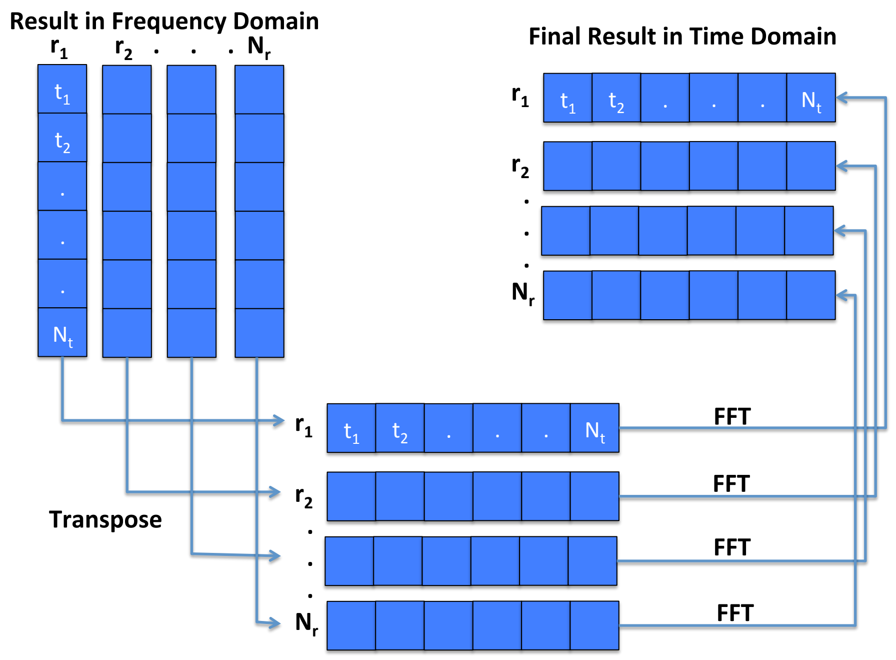


Figure 4.3: Final Solution of the Linear term in time domain

non-local operator. The coupling between Eqs. 4.14 and 4.15 makes the solution non-local, which results in impossible straightforward parallel implementation.

Straightforward solutions to coupled differential equations are easily obtained when dealing with local coupling where interaction between the equations is based on the same space/time point. In some cases, interaction between the equations happens over an interval of space/time leading to non-local coupling. In these cases, the behavior of different effects should be calculated over a ”region” in time or space rather than a single point [62]. This is illustrated in Fig. 4.4 which shows the dependence of ρ on u as an integral over time resulting in non-local coupling of Eqs. 4.14 and 4.15:

$$\partial_z u = -i\gamma(1+i\omega\tau)\rho u - \sigma|u|^2 u - i\mu|u|^{2(K-1)}u \quad , \tag{4.14}$$

$$\partial_t \rho = \nu\rho|u|^2 + |u|^{2K} \quad . \tag{4.15}$$

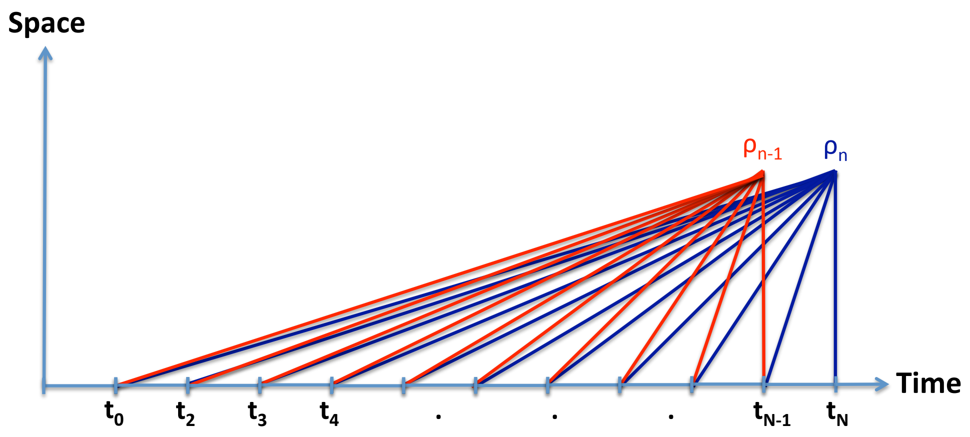


Figure 4.4: Non-local dependence of ρ on u in time domain

To solve the non-linear term, Eq. 4.14 is split into an amplitude and phase as shown below:

$$f(\rho, I) = \partial_z I = -2(\gamma\rho I + \mu I^K) \quad , \tag{4.16}$$

and

$$\partial_z \phi = \sigma I - \gamma\omega\tau\rho \quad , \tag{4.17}$$

where $u = \sqrt{I}e^{i\phi}$ and I presents the amplitude of the electric field.

First the amplitude relation (Eq. 4.16) is solved using a leapfrog integration over z step for every t [61],

$$I_{k+\frac{1}{2}} = I_k + \frac{1}{2} f(\rho_k, I_k)\Delta z \text{ and } I_{k+1} = I_k + f(\rho_{k+1/2}, I_{k+1/2})\Delta z \text{ where } z \to k. \tag{4.18}$$

To calculate $f(\rho, I)$, Eq. 4.15 is usually solved using one of the methods for numerical integration for first order ordinary differential equations, which is essentially a serial procedure. This work contributes to a novel parallel solution by analytical integration of ρ over time which is valid since, for a given intensity $|u|^2$, Eq. 4.15 is linear for ρ.

Here, considering the choice of parallel hardware and also considering the easier parallel implementation of integration, the second method is used. This method parallelises the solution by splitting the problem as an integral equation in time domain. This method is used to solve Eq. 4.15 and then the result is used for the solution of the non-linear term.

This solution starts by writing Eq. 4.15, which describes plasma evolution at the rate of multi-photon and avalanche ionizations, in terms of intensity, where $I = |u|^2$:

$$\partial_t \rho = \nu \rho I + I^K \quad . \tag{4.19}$$

This equation is solved using the variation of constants method. Using this method, first the homogenous part of the equation ($\partial_t \rho = \nu \rho I$) is divided by ρ:

$$\frac{\partial_t \rho}{\rho} - \nu I = 0 \quad . \tag{4.20}$$

Subsequently integration will result in the following:

$$ln|\rho| - \nu \int_{-\infty}^{t} I dt = c \quad , \tag{4.21}$$

where c is the integration constant. Taking the exponential of both sides of Eq. 4.21 will result in the equation shown below:

$$\rho e^{-\nu \int_{-\infty}^{t} I dt} = e^{c} \quad ,$$
$$\rho = C e^{\nu \int_{-\infty}^{t} I \,\mathrm{d}t} \quad \text{where } C = e^{c} \quad . \tag{4.22}$$

The following term can be used to further simplify the plasma equation:

$$\phi = \nu \int_{-\infty}^{t} I \,\mathrm{d}t \quad \text{and} \quad \phi' = \nu I \quad . \tag{4.23}$$

This will yield to the equation shown below derived from Eqs. 4.22 and 4.23:

$$\rho = C e^{\phi} \quad . \tag{4.24}$$

Now it is time to address the inhomogeneous part of Eq. 4.19. Here, due to variation of constants, constant C will be replaced with function C_t and yield to the following:

$$C_t e^{\phi} = I^{K} \quad ,$$
$$C_t = I^{K} e^{-\phi} \quad , \tag{4.25}$$

where integration over time will yield to the following:

$$C = \int_{-\infty}^{t} I^{K} e^{-\phi} dt \quad . \tag{4.26}$$

Finally the plasma ρ can be derived from Eqs. 4.26 and 4.24 as shown below:

$$\rho(t) = e^{\phi} \int_{-\infty}^{t} I^K e^{-\phi} \, \mathrm{d}t \quad \text{where} \quad \phi = \int_{-\infty}^{t} I \, \mathrm{d}t \quad . \tag{4.27}$$

This term is true from $t_0 = -\infty$ since $\rho(t = -\infty) = 0$. The proof of this solution can be achieved by taking derivative of Eq. 4.27, as shown below:

$$\begin{aligned} \rho'(t) &= (e^{\phi})' \int_{-\infty}^{t} I^K e^{-\phi} \, \mathrm{d}t + e^{\phi} I^K e^{-\phi} \\ &= e^{\phi}\phi' \int_{-\infty}^{t} I^K e^{-\phi} \, \mathrm{d}t + I^K \quad . \end{aligned} \tag{4.28}$$

Here considering $\phi' = \nu I$, we can derive the following:

$$\begin{aligned} \rho'(t) &= \nu I e^{\phi} \int_{-\infty}^{t} I^k K e^{-\phi} \, \mathrm{d}t + I^k K \\ &= \nu I \rho + I^K \quad . \end{aligned}$$

Finally, after calculating $f(\rho, I)$ of the amplitude term in 4.18, the phase Eq. 4.17 is solved using the following second order scheme:

$$\frac{\phi^{n+1} - \phi^n}{\partial_z} = \frac{\sigma}{2}(I_k + I_{k+1}) - \frac{\gamma\omega\tau}{2}(\rho_k + \rho_{k+1}) \quad . \tag{4.29}$$

4.2 Initial Condition

The initial electric field is considered to have geometry of a Gaussian pulse which is focused through a lens into the bulk transparent media equivalent to that used in [49]. The radially symmetric initial conditions with Gaussian shape in radial and temporal dimension is presented below,

$$\mathcal{E}(r,z=0,t) = \mathcal{E}_0 exp\left(-\frac{r^2}{\omega_0^2} - \frac{ikr^2}{2f} - \frac{t^2}{t_p^2}\right) \quad , \tag{4.30}$$

where ω_0 and r are the waist and radius of the incident beam accordingly. In this equation, f presents the focal length of the lens and t_p presents the pulse-width,

4.3 Adaptive Step Size

In order to obtain reliable results for numerical modeling of pulse propagation along the z-axis, the value of step size (Δz) should be chosen carefully. Choosing a large step size, will produce unreliable results. On the other hand, choosing a small value leads into implementation of a large number of steps [57].

To calculate a suitable step size and obtain relative smoothness, the electric field $u(r,t)$ should change slowly. Here $u(r,t)$ is a discretisation of the continuous electric field stored in a complex matrix. This smooth change is obtained when both the amplitude and the phase of the complex data changes slowly. For this reason $\Delta\phi$ and $\Delta|u|$ are chosen to be less than a prescribed limit (1%).

To ensure a proper approximation of the discrete numerical scheme, the numerical step in evolution variable z is varied. By calculating the individual linear and non-linear effects separately, critical step size associated with each effect is obtained. Ultimately the minimum step size that presents the most dangerous or dominant effect is chosen. In other words, Δz is calculated by testing all the possible limitations that are enforced by different linear and non-linear effects.

In this thesis, numerical discretisation of the extended NLSE is used for estimation of the correct step size. This approach is only used in the set step routine to prevent unreliable estimation of the step size.

$$\frac{du}{dz} \rightarrow \frac{\Delta u}{\Delta z} = i\kappa\Delta_{\perp}u + i\sigma|u|^2u - id\partial_{tt}u - \gamma(1+i\omega\tau)\rho u - \mu|u|^{2(K-1)}u \quad . \tag{4.31}$$

This equation can be written as the following:

$$\Delta u = (i\kappa\Delta_{\perp}u + i\sigma|u|^2u - id\partial_{tt}u - \gamma(1+i\omega\tau)\rho u - \mu|u|^{2(K-1)}u)\Delta z \quad . \tag{4.32}$$

It can be seen in the equation above that the maximum Kerr self-refraction effect and multi-photon absorption (the second and last term on the right hand-side) depends on maximum intensity $I_{max} = |u_{max}|^2$. Consequently, for the plasma absorption and defocusing (the forth term on the right hand-side), the maximum effect depends on the maximum plasma ρ_{max}.

The following steps is taken to calculate the step size for the maximum optical Kerr effect:

$$\frac{du}{dz} = i\sigma|u|^2u \quad ,$$
$$u^*\frac{du}{dz} = i\sigma|u|^2|u|^2 \quad ,$$
$$u^*u_z = -i\sigma|u|^4 \quad ,$$
$$R(e^{i\phi})_z = -i\sigma R^2Re^{i\phi} \text{ where } u = Re^{i\phi} \quad ,$$
$$e^{i\phi}i\phi_z = -i\sigma R^2e^{i\phi} \quad ,$$
$$\phi_z = \sigma R^2 \quad .$$

Finally step size for the Kerr effect can be calculated from $\Delta\phi = \sigma|u|^2\Delta z$. This equation shows the direct dependence of Δz on $|u|^2$. Here Δz_{Kerr} addresses the step size obtained from the optical Kerr effect:

$$\Delta z_{Kerr} = \frac{\Delta\phi}{\sigma|u|^2} \quad . \tag{4.33}$$

The same solution can be applied to the MPA term, which leads to calculation of the step size for the MPA effect based on $|u|^2$:

$$\Delta z_{MPA} = \frac{\Delta\phi}{\mu|u|^{2(K-1)}} \quad . \tag{4.34}$$

Maximum step size for the plasma absorption (Δz_{abs}) and defocusing (Δz_{defoc}) effects is presented by,

$$\Delta z_{abs} = \frac{\Delta\phi}{\gamma\rho} \quad \text{and} \quad \Delta z_{defoc} = \frac{\Delta\phi}{\gamma\omega\tau\rho} \quad . \tag{4.35}$$

As for the solution of the linear terms (dispersion and beam diffraction), a matrix transpose will be needed. Maximum dispersion will be calculated by taking the discrete derivative of the dispersion term, $u_{tt} = d(u_{n+1} - 2u_n + u_{n-1})/\Delta t^2$. For this effect, the step size Δz depends on the time step size Δt.

Subsequently maximum diffraction will be calculated by $\Delta_{\perp} u = K(u_{rr} + u_r/r)$, where step size Δz, is calculated from radial step size Δr. This approach chooses the minimal step size associated with the most dominant effect for the steps size of the combined numerical scheme.

Chapter 5

Numerical Modeling of Ultra-short Pulse Propagation on CPU

A serial and CPU-based numerical solution to the NNLSE is used in [7, 9, 10] for numerical modeling of femtosecond laser pulse propagation in bulk transparent material. The data obtained from this implementation suffers from insufficient numerical resolution whenever sudden changes in the intensity of the electric field occur. This is due to the multi-scale nature of the pulse propagation that causes intrinsic stiffness in the mathematical problem. To obtain a finer grid, a hierarchy of adaptive refinement is used, where the grids are dynamically refined in both the spatial and temporal features to generate adequate numerical resolution during the evolution along the z-axis as described in [9].

To refine the data obtained from the serial NNLSE solver, after a certain number of steps are taken along the z-axis, the local resolution of the grids should be checked to ensure sufficient resolution of the grid. Whenever poor numerical resolution is identified, the refinement routine should be started to produce a finer grid. This is achieved by increasing the number

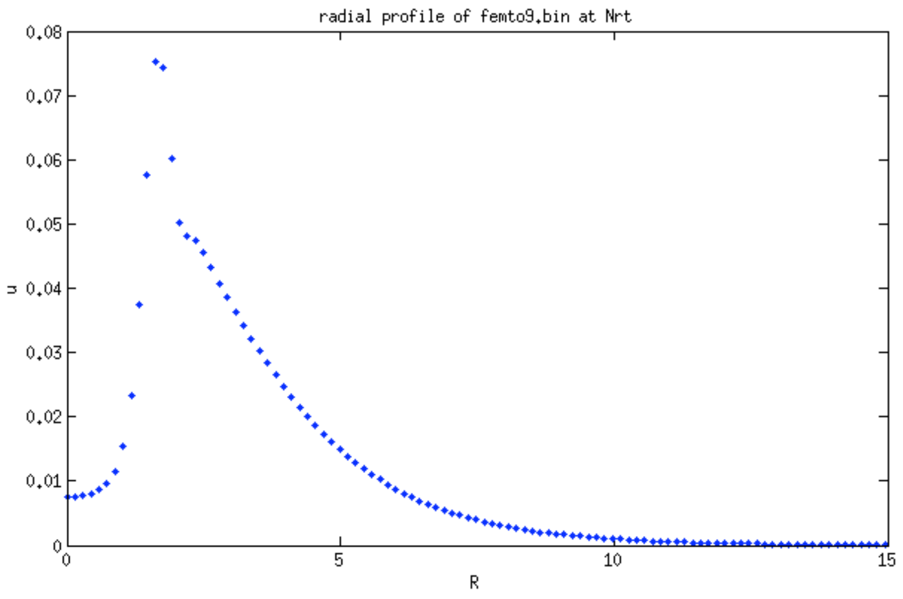


Figure 5.1: Radial profile of the electric field at specific point along the z-axis

of mesh points and refining the under-resolved areas with grids of higher resolution similar to that used in [63]. This procedure is repeated whenever inadequate resolution in the local grid is detected.

In this chapter, the adaptive mesh refinement routine is facilitated by the design and implementation of a numerical routine for detection of the approximate point with poor resolution in the grid. A sample of poor resolution of the data profile is shown in Fig. 5.1. This figure shows an under-resolved profile, which makes identification of the maximum intensity unreliable.

5.1 Detection of Poor Resolution

This study presents a new method to detect and locate the approximate point with poor resolution by calculation of the amplitude's rate of change in time and radial domain using the second discrete derivation. This is approximated by the 3-point centered second difference [64]:

$$D1 = \frac{u_{n+1} - 2u_n + u_{n-1}}{h^2} \quad , \tag{5.1}$$

and

$$D2 = \frac{u_{n+2} - 2u_n + u_{n-2}}{4h^2} \quad , \tag{5.2}$$

where h presents the step size and u_n the amplitude of the electric field. Here, u is defined as a 2-D Complex matrix of size $N_r \times N_t$ where every row contains $u(t)$ for a fixed radial position. Note that here the discrete second second derivative should provide close results for a properly resolved mesh function since second derivatives are the highest differential terms involved in our model.

These two equations describe the discretised curvature of the field u_n. This means sufficiently close values of $D1$ and $D2$ indicates proper discrete resolution while significant discrepancy between the two indicates poor approximation resulting in poor resolution. It should be noted that for small values of u, $D1$ and $D2$ will be minor and this can lead to irregularities. Therefore, peripheral stability is achieved by focusing the refinement on areas where the amplitude of the beam is greater than a small percentage of the maximum amplitude defined by $u_{threshold} = 10\% \times Max(|u_{r,t}|)$.

Ultimately the maximum roughness in radial or time domain can be calculated and normalised by the maximum amplitude in the time/radial profile for every z step accordingly,

$$roughness = \frac{|D1 - D2|}{Max(|u_{Local}|)}, \tag{5.3}$$

where the results obtained from the detection routine will be used for adaptive mesh refinement.

The serial NNLSE solver together with the adaptive mesh refinement routine was used in [7] for numerical modeling of femtosecond pulse propagation. In this work, the focus is on energy deposition for typical regimes of femtosecond inscription in fused silica by fundamental harmonics at 1030 nm and second harmonics at 515 nm. In this work, absorption of energy is analysed which is defined by $E_{in} - E_{out}$, where $E_{in} = E(z = 0)$ and E_{out} is computed by the following:

$$E(z) = \int_{-\infty}^{\infty} dt \int_{0}^{\infty} rdr|u|^2 \tag{5.4}$$

This paper shows that different absorption effects dominate in different regimes. As a result, for lower energy regions, MPA is the dominating effect while the plasma absorption is the dominating effect for the higher energies. Distribution of the plasma density for both fundamental and second harmonics wavelength produced by the serial NNLSE solver and refined by the adaptive mesh routine is shown in Fig. 5.2 [7].

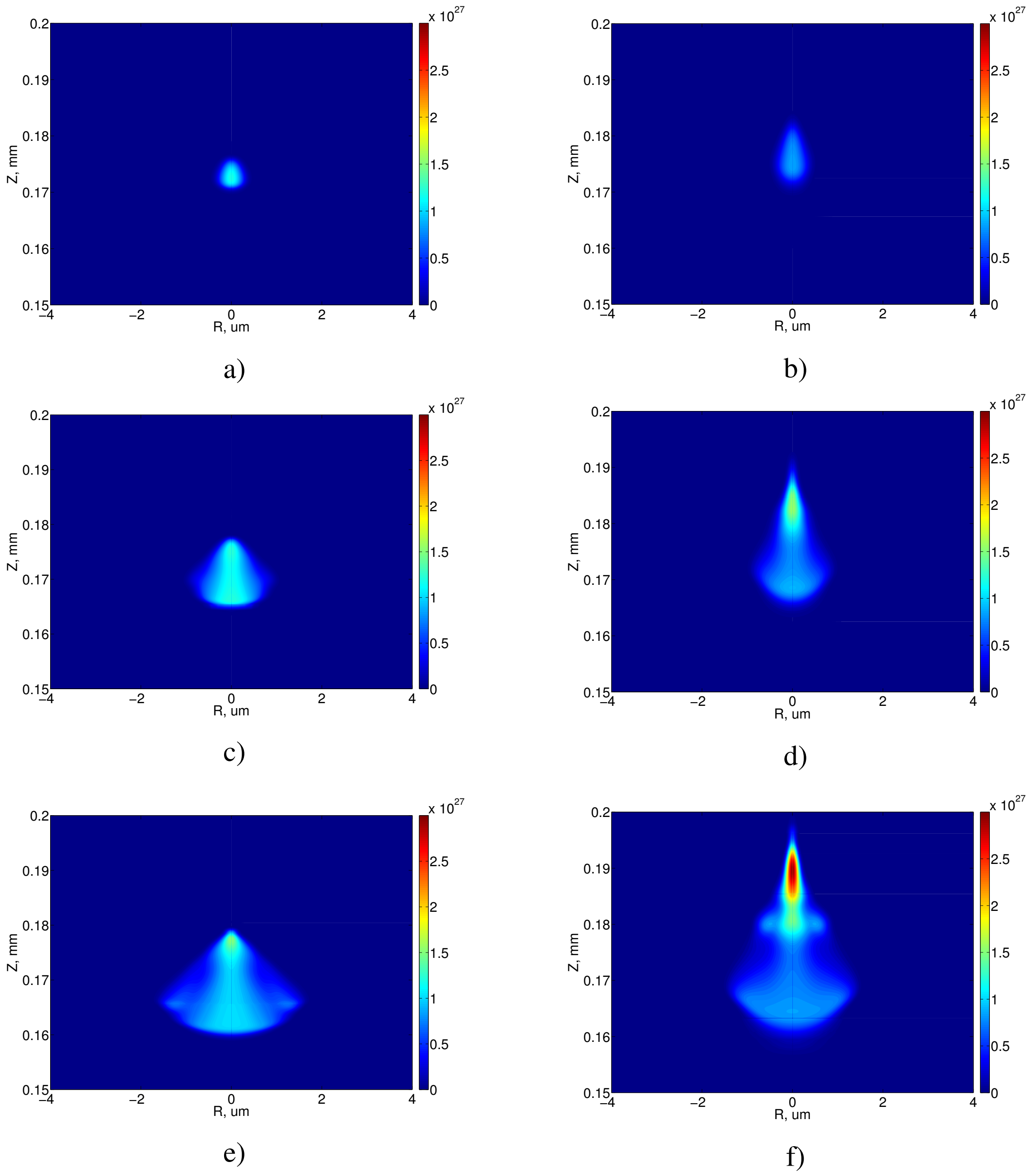


Figure 5.2: Distribution of the plasma density for wavelength λ = 1030 nm (left column), λ = 515 nm (right column) with different initial pulse energies: a),b)-41 nJ; c),d)-132 nJ; e),f)-335 nJ. [7]

Chapter 6

Parallel Implementation of Non-local Non-linear Schrödinger Equation on GPUs

This chapter describes the parallel implementation of a specific numerical model that simulates femtosecond laser pulse propagation in transparent media. This implementation was developed based on NVidia's CUDA parallel hardware. Performance of the mentioned multi-threaded parallel code will be compared with a serial CPU version similar to that described in [9, 10]. In this chapter, the CPU implementation will be addressed by *serial NNLSE solver* and the parallel GPU-based implementation will be addressed by *NNLSE solver*.

Development of the mentioned code started by profiling the existing serial CPU version to identify the numerical bottlenecks associated with different modules in this code. The results obtained from profiling the serial NNLSE solver can help identifying opportunities to eliminate bottlenecks and gain speed by a parallel implementation.

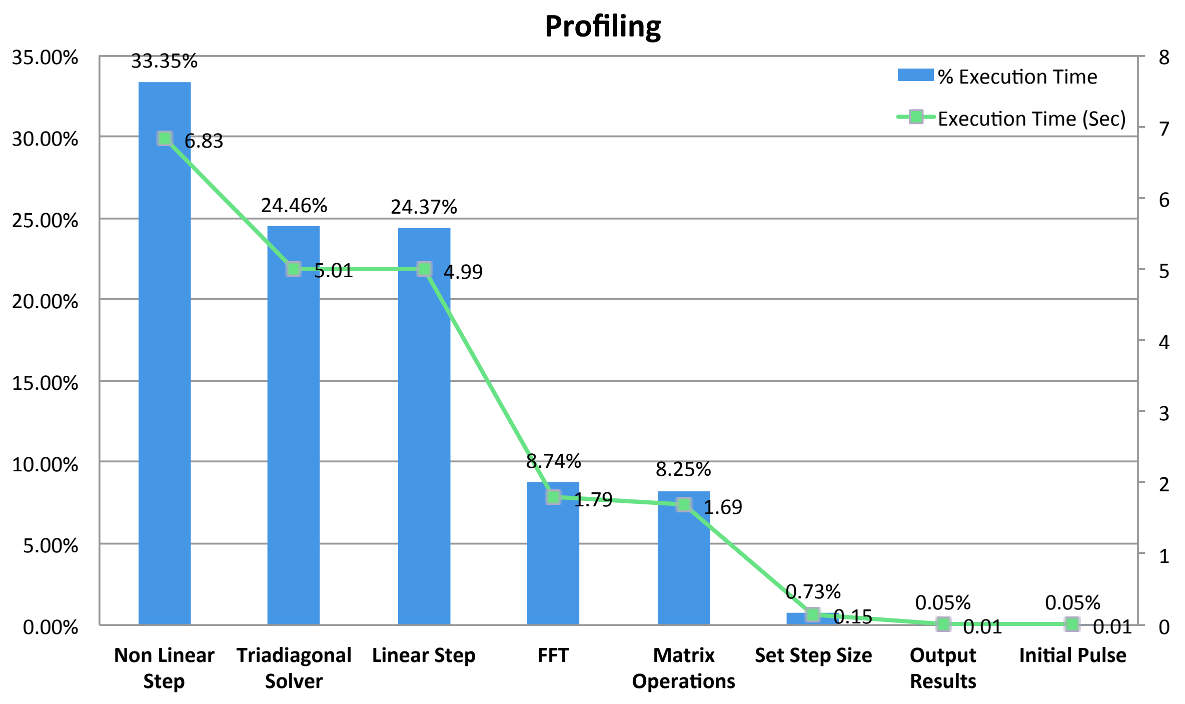


Figure 6.1: Results obtained from profiling the existing serial NNLSE solver

The profiling output of the serial NNLSE solver is illustrated in Fig. 6.1. This figure shows that 33.35% of the running time (6.83s) belongs to the NonLinearStep method, responsible for solving the non-linear term. This method includes a number of standard mathematical functions that execute faster on parallel hardware. The second method with the highest running time is a tridiagonal solver that solves a tridiagonal matrix serially using the LU decomposition method. Fig. 6.1 shows that this method takes 24.46% of the running time (5.01s). Solving a tridiagonal matrix using a parallel algorithm in CUDA should reduce this method's running time by assigning concurrent threads to elements of the tridiagonal matrix. It should be noted that the LinearStep routine, responsible for solution of the linear term, includes execution of the Fourier transform and the tridiagonal solver module. However, the time shown in Fig. 6.1 indicates only the time spent in the Linear step routine and not any of the modules called by the Linear Step routine. As it can be seen from the profiling

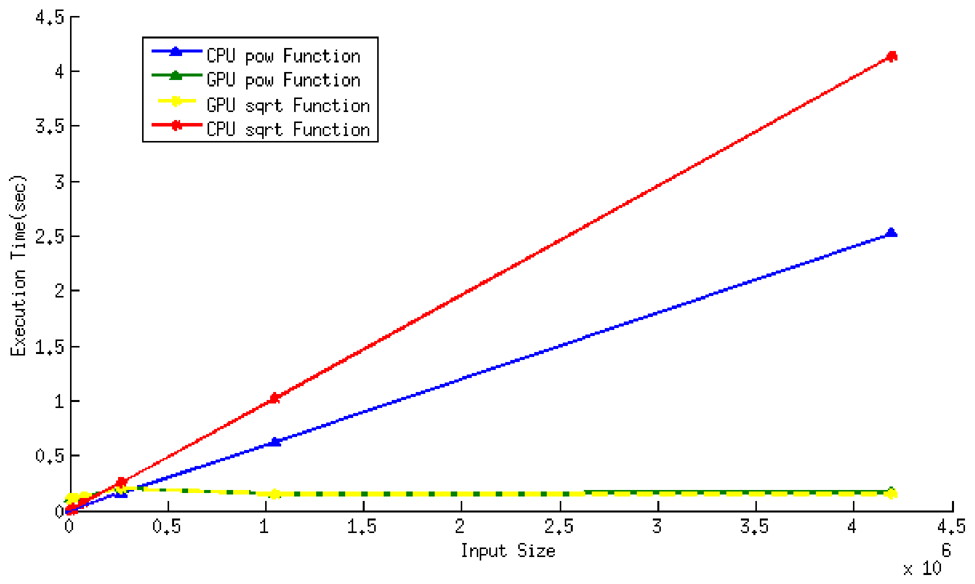


Figure 6.2: Average run time obtained from implementation of standard math functions(Power and Square root) on CPU and GPU

report, there are a number of bottlenecks in the existing serial NNLSE Solver. This chapter aims to remove these bottlenecks by using the highly parallel capabilities of the NVidia's CUDA .

Solving the NNLSE includes the execution of a number of standard mathematical operations such as square root or power. To examine the time it takes to run these functions in both CPU and GPU, a test program was developed to execute the pre-defined math functions *pow* and *sqrt*. These functions are used to calculate power and square root of all the elements of an array of type floating point. The average run time of these two implementations for input size from 2^{12} up to 2^{22} is illustrated in Fig. 6.2. This figure shows the GPU's run time of the pow and sqrt function is identical and is significantly faster that the CPUs serial implementation.

As mentioned in the previous chapter, the split-step method is used for solution of the NNLSE. Using this method data is first initialised with radially symmetric Gaussian pulse

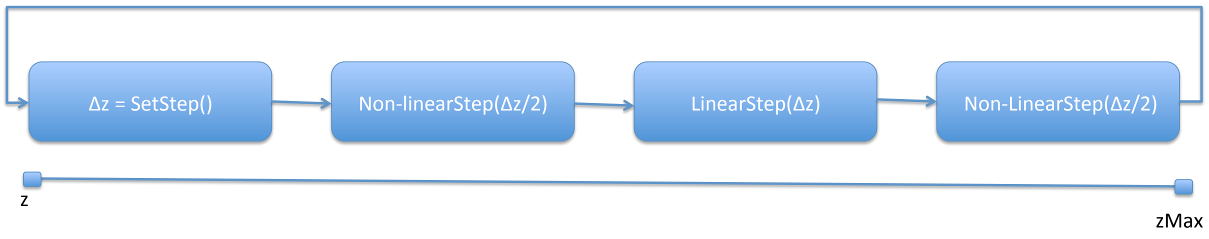


Figure 6.3: Order of linear and nonlinear split steps taken along the z-axis

and then the step size Δz is computed based on routines described in sections 4.2 and 4.3 in the SetStep routine. Subsequently, the non-linear step will be executed over step size of $\Delta z/2$ in NonlinearStep routine, after which the signal is transformed to Fourier domain for implementation of the linear term(LinearStep routine). After execution of the inverse Fourier transform, another non-linear step is taken with step size of length $\Delta z/2$. As a result the implementation of the NNLSE solver will consist of three main routines for calculation of the linear term, the non-linear term and the step size as illustrated in Fig. 6.3. Each of these routines will make use of parallel modules including tridiagonal solver, FFT and etc. A parallel implementation of these two routines and the modules is described in sections 6.1 and 6.2.

It should be noted that the initial pulse is stored in a complex matrix $u(m,n)$ which is a discretisation of the continuous field $u(r,t)$. To boost performance, this matrix is physically stored as a 1D array of size $N_r \times N_t$ where N_r and N_t stand for the number of grid points in radial and time direction accordingly. Every row in this matrix contains $u(t)$ for a fixed radial position. The array is periodically reorganised using parallel routines to perform matrix transposition in order to provide continuous storage for coordinate-wise parallel operations such as Fourier transform and integration of the radial operator using the Crank-Nicolson scheme [61]. The numerical step in the evolution variable z is varied to ensure a proper

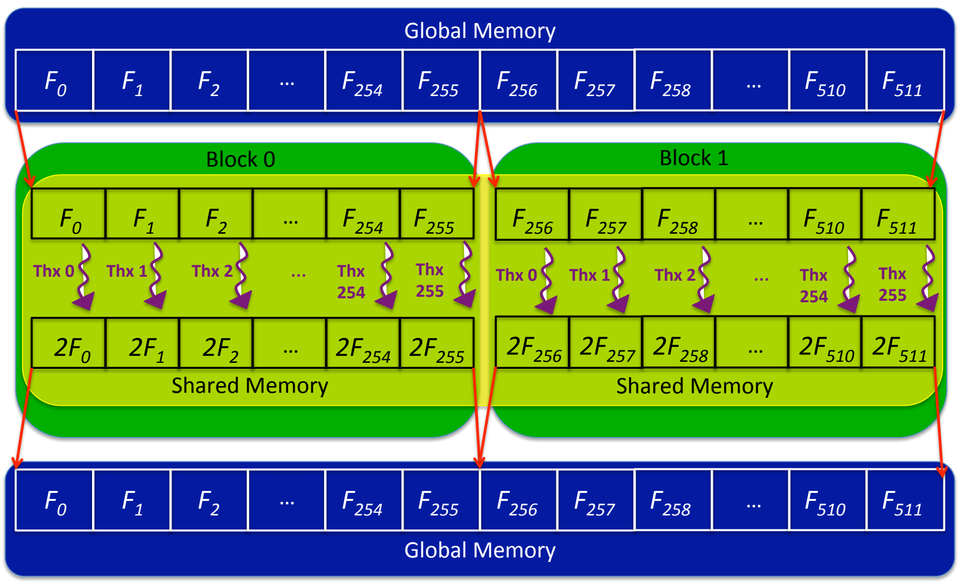

Global Memory
F_0 F_1 F_2 ... F_{254} F_{255} F_{256} F_{257} F_{258} ... F_{510} F_{511}
Block 0
Block 1
Thx 0
Thx 1
Thx 2
Thx 254
Thx 255
$2F_0$ $2F_1$ $2F_2$ ... $2F_{254}$ $2F_{255}$ $2F_{256}$ $2F_{257}$ $2F_{258}$ ... $2F_{510}$ $2F_{511}$
Shared Memory
Shared Memory
Global Memory

Figure 6.4: Execution of independent operation on data stored in 1-D array in CUDA

approximation of the discrete numerical scheme.

Both the linear and the non-linear routines include execution of independent operations on data input (e.g. multiplication of all the elements of the data array by a constant). Implementation of such operations is straightforward as there is no memory dependency. Fig.6.4 illustrates an execution of a single independent operation on all the elements of the input array. To execute such operations efficiently, initially data is loaded into the low-latency shared memory and assigned to thread blocks. This data is accessed by CUDA thread blocks through *thread ID* and *block ID*, which are distinctive coordinates assigned by the CUDA runtime. Thread ID and block ID can be used to perform various operations on elements of an ordered data input like arrays and matrices. Using these IDs, parallel implementation is achieved by assigning one thread to each data element as shown in Fig. 6.4. As it can be seen in this figure, every 256 threads form a thread block, where the threads within each block have read/write access to the same shared memory [20, 29].

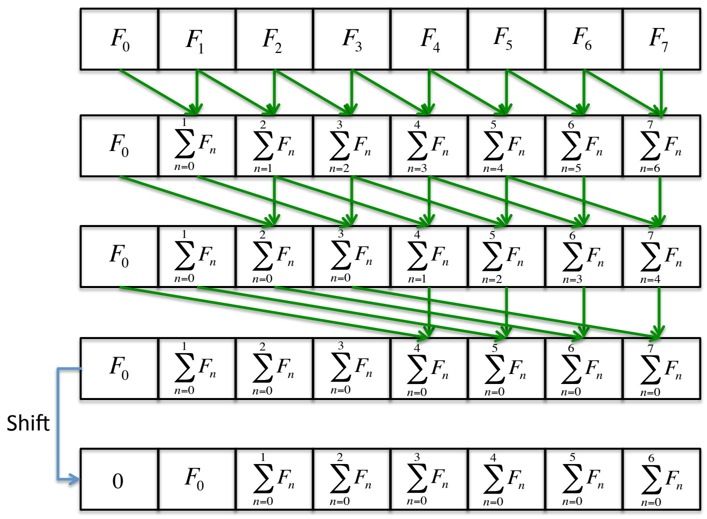


Figure 6.5: Naive algorithm for implementation of exclusive scan [65]

6.1 Non-linear Step

The non-linear routine is implemented for execution of the non-linear term in the NNLSE solver. As mentioned in part 4.1.2, parallelisation of the solution of the non-local non-linear term is done by splitting the problem as an integral equation in time domain. The parallel implementation of this routine requires the modules described in the following subsections.

6.1.1 Scan Sum

This module performs the *all-prefix-sum* operation on all the elements of the input array as shown below:

Ordered data input,

$$[F_0, F_1, ..., F_{N-1}] \quad . \tag{6.1}$$

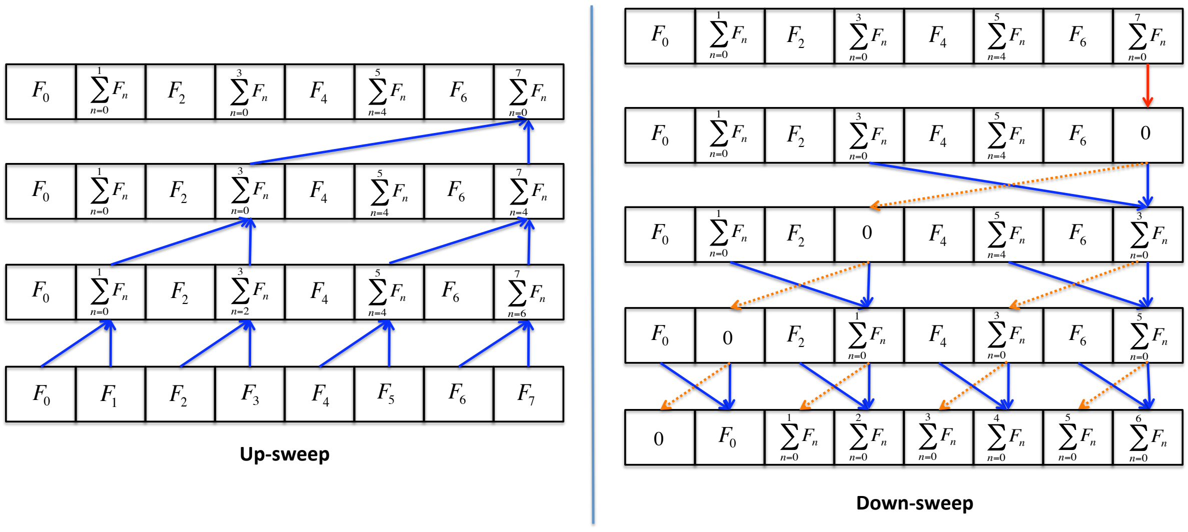


Figure 6.6: Work efficient algorithm to implement parallel exclusive scan [65]

Ordered data output,

$$[0,\ F_0,\ F_0+F_1,\ \ldots,\ F_0+F_1+\ldots+F_{N-1}]\quad . \tag{6.2}$$

This operation takes an array of size N as input and produces an array of the same size where each element is equal to the sum of all the previous elements which is also referred as exclusive scan sum [66, 67].

Blelloch in [66,67] stated the possibility of efficient parallel implementation of this essentially serial operation. In [65], Harris described an efficient parallel algorithm to implement exclusive scan on CUDA architecture. He started by describing a naive algorithm shown in Fig. 6.5, where numerous addition operations result in inefficient parallel implementation of this module. Subsequently, an efficient scan algorithm is described in [65], which uses the *balanced tree* approach. This efficient algorithm works by creating a balance tree on the input data and calculating the exclusive scan by sweeping up and down this tree as shown in Fig. 6.6.

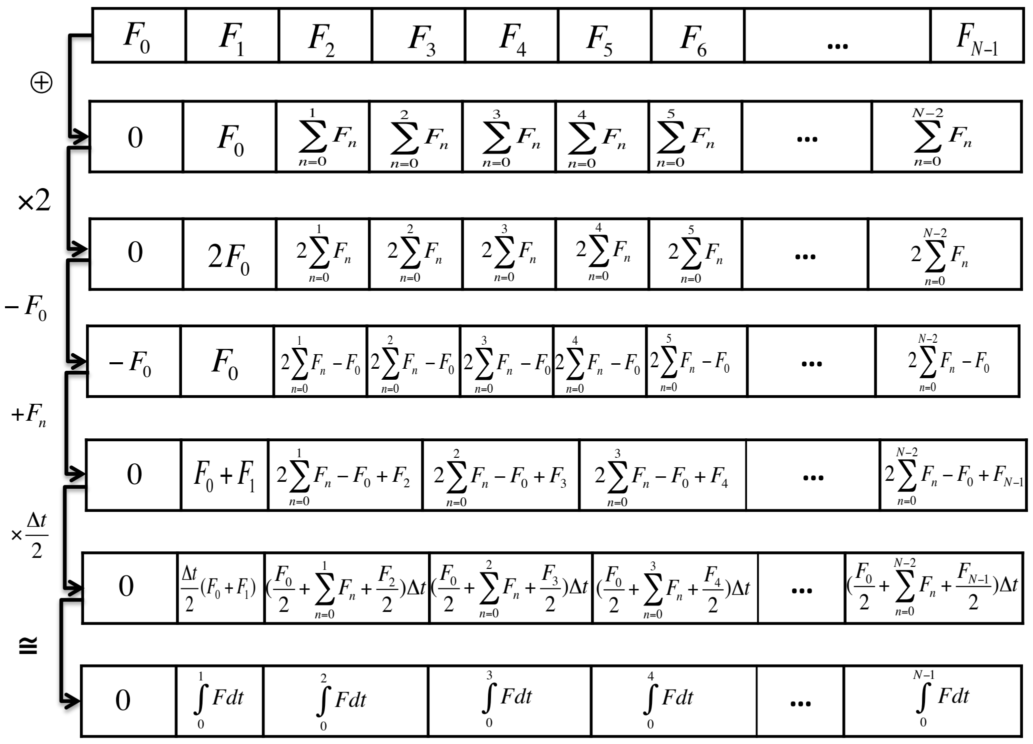


Figure 6.7: Implementation of integration using trapezoidal rule

The parallel prefix sum code described in [65] and provided in the CUDA SDK [68] is used for implementation of the scan sum operation in this work.

6.1.2 Integration

As mentioned in section 4.1.2, partial numerical integration is required to approximate evolution of plasma over time. This is achieved by delegation of the trapezoidal rule to concurrently running threads [61].

$$\rho(t) = e^{\phi} \int_{-\infty}^{t} I^k e^{-\phi}\,\mathrm{d}t \quad \text{where} \quad \phi = \int_{-\infty}^{t} I\,\mathrm{d}t \quad . \tag{6.3}$$

To implement this code, a stand-alone module was initially implemented to calculate $cosh(x)$ by integration for the following equation:

$$\int_{-1}^{1} \left(\frac{1}{2}(e^x + e^{-x}) \right) dx \quad . \tag{6.4}$$

Integration was implemented using the trapezoidal rule to approximate the area under the graph of the function $f(x) = \int_{-1}^{1} \left(\frac{1}{2}(e^x + e^{-x}) \right) dx$ for the interval a = -1 to b = 1 for n subintervals.

$$\int_a^b f(x)dx = \frac{\Delta t}{2}(f(1) + 2\sum_{k=1}^{n-1} f(k) + f(n)) \quad \text{where} \quad \Delta t = \frac{b-a}{n} \quad . \tag{6.5}$$

Parallel implementation of the integral module is achieved by executing the steps shown in Fig. 6.7. The first step shown in this figure is implementation of the *all-prefix-sum* operation which was described in the previous section as a separate module. Following the scan sum, a number of independent operations are executed in the same manner shown in Fig. 6.4.

Accuracy of the integration module is tested by comparing the results obtained from implementation of Eq. 6.4 with $\sinh(x)$. Based on relation 6.6, comparing the results obtained from calculation of these two terms, shows the accuracy of the integration.

$$\int \left(\frac{1}{2}(e^x + e^{-x}) \right) dx = \sinh(x) \quad . \tag{6.6}$$

The integration module was developed both in the C programming language for a CPU platform and through the CUDA interface for a GPU platform. Comparison of the results obtained from implementation of Eq. 6.6 is performed by calculation of the relative error for each implementation,

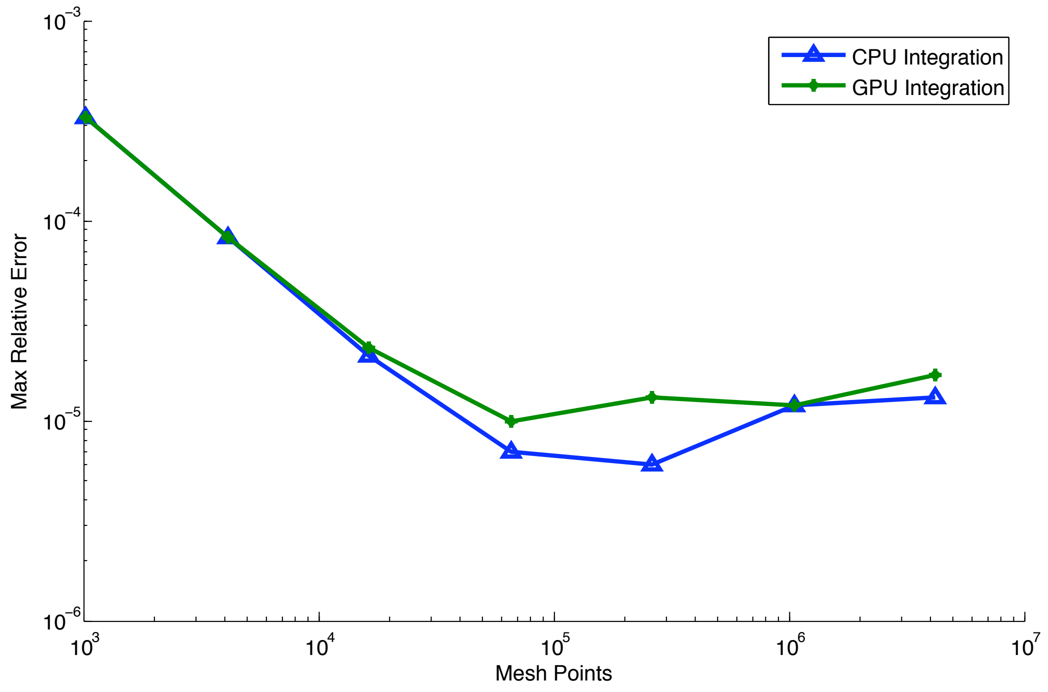


Figure 6.8: The average accuracy of the integration for the CPU and the GPU, measured using Eq. 6.7

$$\frac{\sinh(x) - \int \left(\frac{1}{2}(e^{x} + e^{-x})\right) dx}{\max(\sinh(\mathrm{x}))} \quad . \tag{6.7}$$

Figure 6.8 shows the accuracy of the integration routine both on the GPU and CPU. As shown in this figure, when the number of elements n grows from 32 to 1024 the error reduces from 10^{-4} to 10^{-6}. This reduction in the relative error is better shown in Fig 6.9 which plots relative error in the GPU from $n = 4$ to 1000. This figure shows error reducing from 10^{-2} to 10^{-6}, resulting in four orders of magnitude reduction in the error, providing floating point precision for mesh points greater than 100.

In addition to accuracy, the run time of the two implementation are shown in Fig. 6.10, which indicates that GPUs give a superior performance as expected.

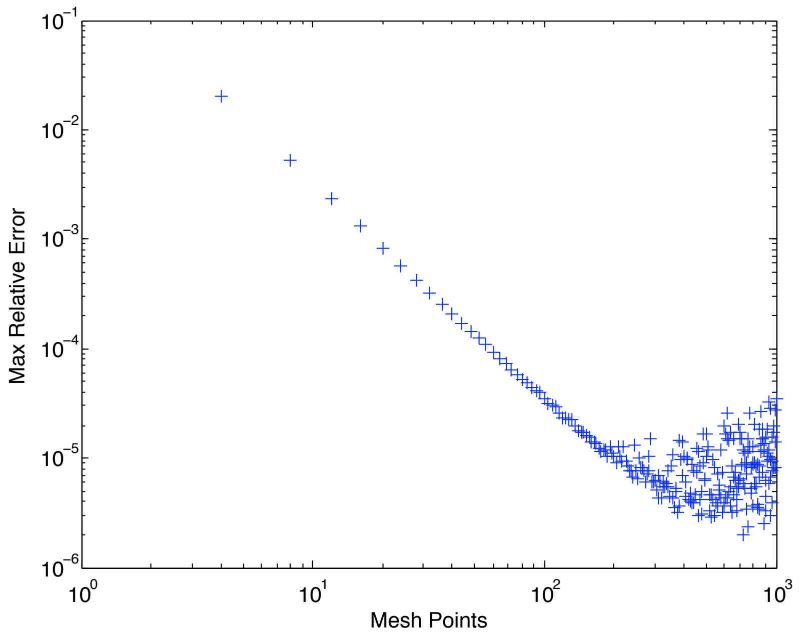


Figure 6.9: The accuracy of the GPU implementation of the integration routine measured by Eq. 6.7

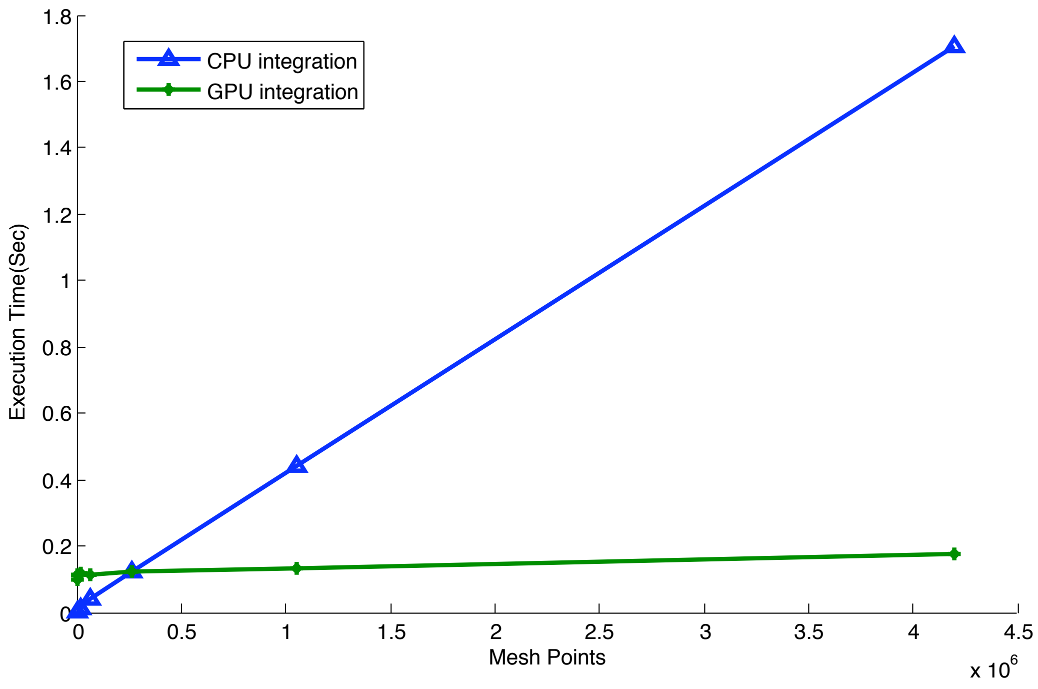


Figure 6.10: CPU vs GPU comparison of the average run time of the integration routine for mesh size 32x32 to 1024x1024

6.2 Linear Step

The linear method is implemented for execution of the linear term in the extended non-linear Schrödinger equation. As mentioned in part 4.1.1, parallelisation of the solution of the linear term includes parallel implementation of a tridiagonal solver and a Fourier transform. These modules are described in detail in following sections.

6.2.1 Tridiagonal Solver

This module provides a solution for large tridiagonal matrices, which contain nonzero elements only on the diagonal, upper diagonal and lower diagonal. The tridiagonal solver module is used frequently in numerical applications. Therefore, it is trivial to find a fast optimised solution for these matrices.

A tridiagonal matrix is usually solved using the LU decomposition method. This method presents the matrix as a product of an upper and a lower triangular matrix. Then it uses forward substitution and backward substitution to get the unknown vector x. This method, which is also known as the Thomas algorithm, was first presented in [61,69].

The matrix equation is:

$$\mathbf{A} \cdot \mathbf{x} = \mathbf{b} \quad , \tag{6.8}$$

where **A** is a tridiagonal matrix, **b** is a known column vector on the right-hand side and **x** is an unknown column vector.

LU decomposition on A is:

$$\mathbf{L} \cdot \mathbf{U} = \mathbf{A} \quad , \tag{6.9}$$

where L is the lower triangular, U is the upper triangular. The original equation can be written as $b = (L.U).x$, which can be presented as b = L.(U.x). Here an unknown vector y can be used to solve $b = L.y$ by forward substitution which in turn leads to solution of the the original problem by backward substitution $y = U.x$. Hence, the original problem can be decomposed into the following steps:

$$\mathbf{L} \cdot \mathbf{y} = \mathbf{b} \quad , \tag{6.10}$$

$$\mathbf{U} \cdot \mathbf{x} = \mathbf{y} \quad . \tag{6.11}$$

This recursive method is well suited when there is a single processor, single thread system and the matrix is solved serially. For large matrices with a large number of equations a parallel solution can speed up the computations. The first parallel algorithm for solution of a tridiagonal matrix called *The Cyclic Reduction* was developed by R.W. Hockney in [70]. Following that, Stone in [71] presented a recursive double algorithm for parallel solution of the tridiagonal equation.

In [72], Stone compared different algorithms for solution of a tridiagonal matrix including the odd-even cyclic reduction algorithm by Buzbee et al. [73], the Buneman algorithm [74] and his own double recursive algorithm. Subsequently he stated the cyclic odd-even reduction method as the most efficient algorithm.

The cyclic odd-even reduction method was chosen for parallel implementation of a tridiagonal solver on CUDA. Fig. 6.11 illustrates the cyclic odd-even reduction method where reduction is performed by eliminating the odd entries in parallel. This is done by combining the equations in groups of three and eliminating the even ones in succession [75].

For example, the first three equations in the matrix shown in Fig. 6.11 are presented below:

$$
\begin{aligned}
b_0x_0 + c_0x_1 &= f_0 \quad , \\
a_1x_0 + b_1x_1 + c_1x_2 &= f_2 \quad , \\
a_2x_1 + b_2x_2 + c_2x_3 &= f_3 \quad .
\end{aligned}
\tag{6.12}
$$

To perform cyclic reduction, three new variables α, β and γ are defined as shown below:

$$
\alpha = \frac{-a_1}{b_0}, \;\; \beta = 1, \;\; \gamma = \frac{-c_1}{b_2} \quad . \tag{6.13}
$$

The following relationship is true for these variables:

$$
\alpha b_0 + \beta a_1 = 0 \;\text{ and }\; \beta c_1 + \gamma b_2 = 0 \quad . \tag{6.14}
$$

Subsequently, the Eqs. 6.12 are multiplied by these variables respectively:

$$
\begin{aligned}
\alpha b_0x_0 + \alpha c_0x_1 &= \alpha f_0 \quad , \\
\beta a_1x_0 + \beta b_1x_1 + \beta c_1x_2 &= \gamma f_2 \quad , \\
\gamma a_2x_1 + \gamma b_2x_2 + \gamma c_2x_3 &= \gamma f_3 \quad .
\end{aligned}
\tag{6.15}
$$

At this stage, combining these three equations, will result in the following single equation in:

$$
\begin{aligned}
(\alpha c_0 + \beta b_1 + \gamma a_2)x_1 + (\gamma c_2)x_3 &= \alpha f_0 + \beta f_1 + \gamma f_2 \quad , \\
b_1'x_1 + c_1'x_3 &= f_1' \quad .
\end{aligned}
\tag{6.16}
$$

As it can be seen above, the even unknowns are eliminated in the final equation and only odd

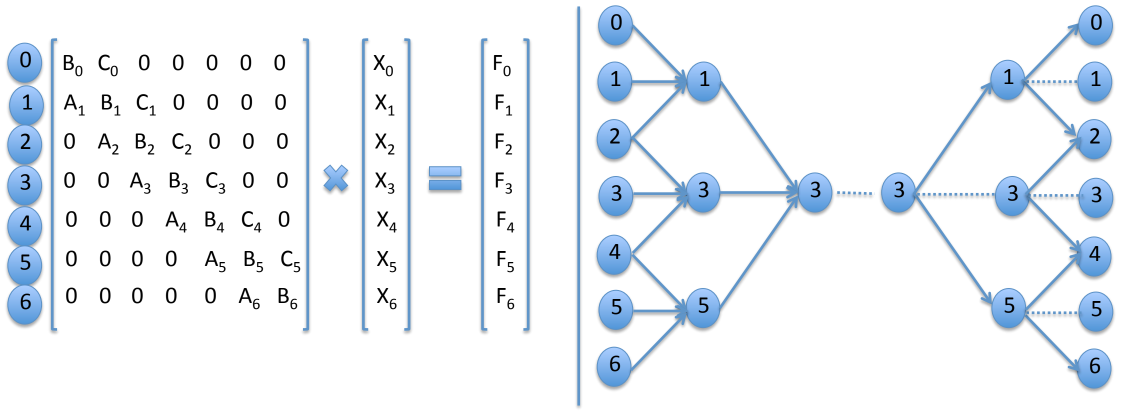


Figure 6.11: solution of a tridiagonal matrix using the cyclic reduction method [75]

unknowns are left. This process is performed in parallel and successively until the system is reduced to only one equation with one unknown. After calculating the value of this unknown, back substitution is used to calculate the value of all the other unknowns [75]. This process is illustrated in Fig. 6.11.

Implementation of the cyclic reduction routine requires specification of the level of parallelism considering the choice of hardware. This has been analysed in several papers for different systems, Kershaw has designed a system for Cray in [76], Lambiotee et al in [77] for CDC STAR-100 (a vector supercomputer) and Cox in [78] for an FPS-T20 (a parallel processing system with vector processors in a hypercube topology). A derivation of the divide and conquer strategy that was presented in [78] is used in this work as a solution of the tridiagonal solver in CUDA.

In this system, a tridiagonal matrix of size $N \times N$ and a platform with P processors is considered, where $N = 2^n - 1$ and $P = 2^p - 1$. An execution of $k = n - p$ consecutive cycles are required to reduce the system. The number of equations in each local processor will be $N_p = 2^{k+1} - 1$ and the number of common unknowns in each processor will be $C_n = 2^k - 1$. This is illustrated in Fig. 6.12 which shows the cyclic reduction of a 15×15 matrix in a sys-

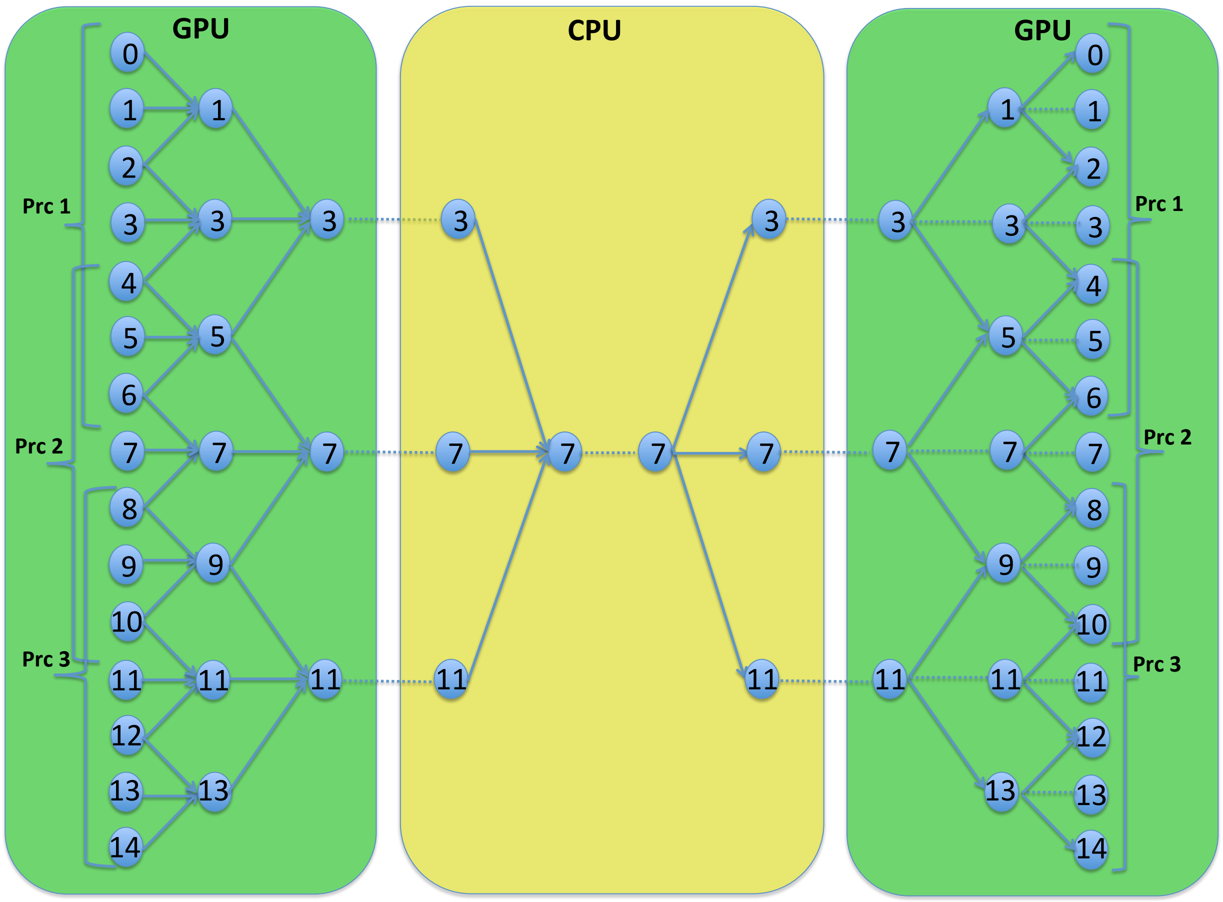


Figure 6.12: solution of a tridiagonal matrix using the cyclic reduction method on CUDA

tem with three processors, where $n = 4, p = 2$. In this system a collection of seven equations is assigned to each processor and each processor will reduce the number of equations in two cycles ($k = 2$). Subsequently, the final reduced equation in each processor is send back to CPU. The CPUs host code performs further reduction to narrow down the whole matrix to one single equation that leads to straightforward calculation of all the unknowns.

It should be noted that here parallelism is achieved by processing a very large array with multiple thread-blocks, where each thread-blocks reduces a portion of the array. Here final reduction of the results obtained in each thread-blocks requires synchronized communication between thread-blocks. However, CUDA does not support global synchronization which is the reason why the partial results are communicated back to CPU for the final reduction.

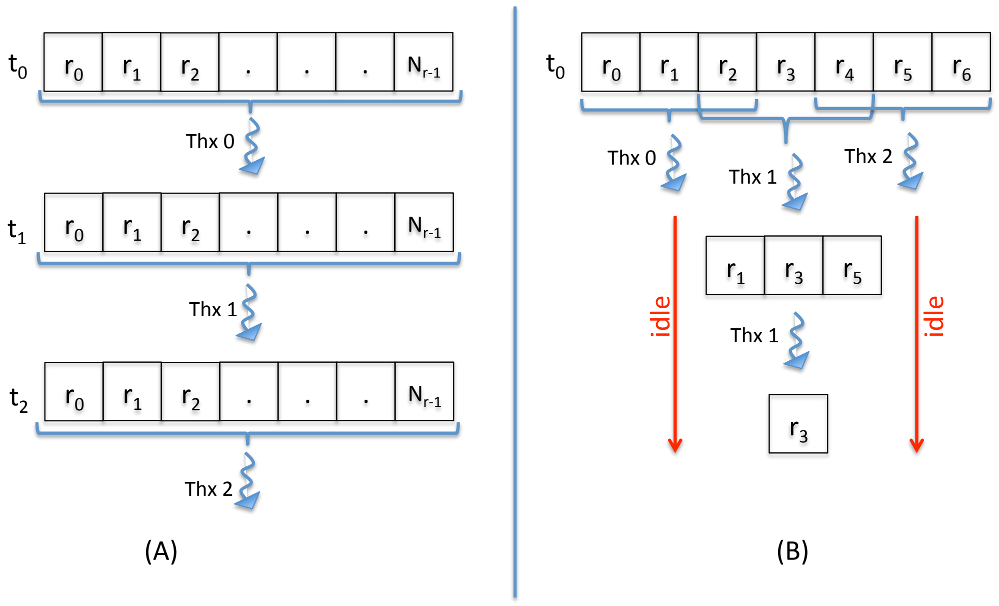

t_0
r_0 r_1 r_2 . . . N_{r-1}
Thx 0
t_1
r_0 r_1 r_2 . . . N_{r-1}
Thx 1
t_2
r_0 r_1 r_2 . . . N_{r-1}
Thx 2
(A)
t_0
r_0 r_1 r_2 r_3 r_4 r_5 r_6
Thx 0
Thx 1
Thx 2
idle
r_1 r_3 r_5
Thx 1
r_3
idle
(B)

Figure 6.13: Assignment of threads for implementation of (A) the cyclic reduction routine (B) the LU decomposition method for solution of a tridiagonal matrix.

In this thesis, two different methods were implemented for solution of the tridiagonal matrix; the first one uses the cyclic reduction method as described above and the second one uses the LU decomposition method. As mentioned earlier, the LU decomposition is a recursive and serial procedure more suitable for single processors. Here considering the nature of the problem, where it is required to find solution of a tridiagonal solution for every row of a 2D matrix, LU decomposition can be parallelised by assigning one thread to every row. The thread assignment for both implementation is illustrated in Fig. 6.13. As it is shown in this figure, one thread is assigned to every row for parallel LU decomposition of individual rows. Comparison of the run time of the two different methods shows a superior performance of the serial LU decomposition method. As stated in [79] by Zhang et al, the cyclic reduction method suffers from bad thread utilisation. This is shown clearly in Fig. 6.13, where lower stages of the reduction and substitution leaves most of the threads idle. In this method also coalesced memory access is impossible since every thread needs to access three equations at

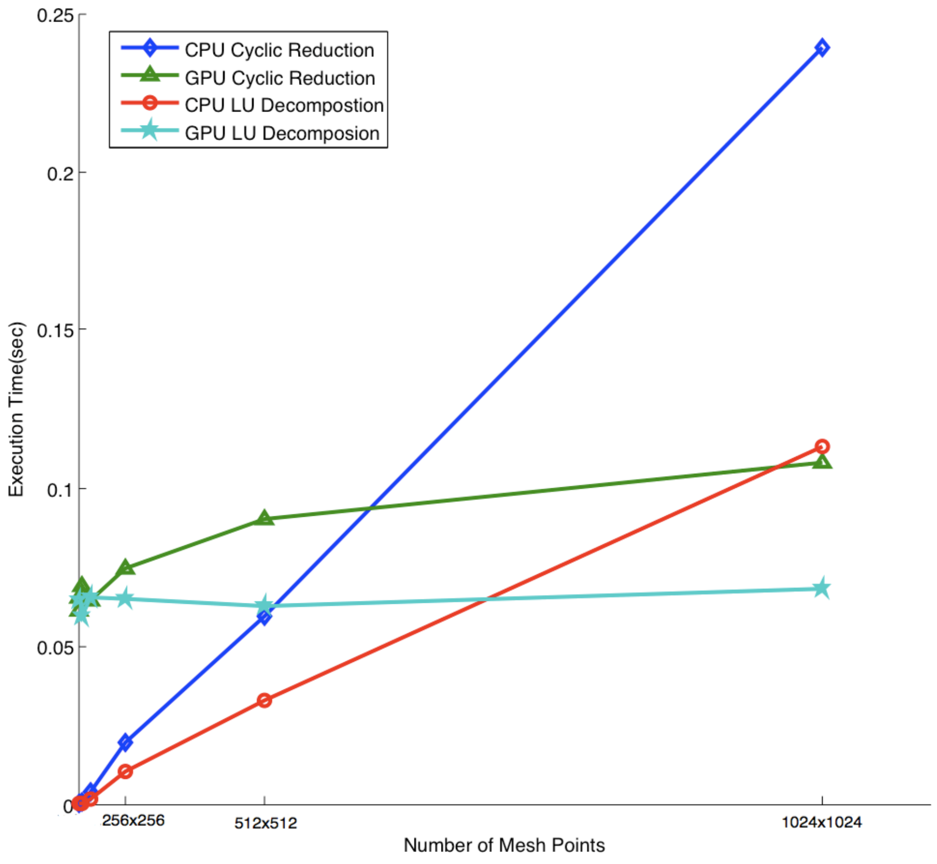


Figure 6.14: Average run time of the cyclic reduction and the LU decomposition method for both the CPU and GPU implementation

once. The timeline for execution of these two methods in the CPU and the GPU is presented in Fig. 6.14.

6.2.2 Fast Fourier Transform

This section describes an early investigation of the CUDA version of the Fast Fourier Transform called *CUFFT* [32]. Here CUDA's CUFFT library is compared with one of the most popular FFT libraries called *Fast Fourier Transform in the West* (FFTW) for CPUs [80]. This comparison has been investigated previously in [81–84], where [82] and [83] stated the superior performance of the CUFFT library for data sets of size 2^n where $n \geq 13$. Also [84] stated that the time needed to transfer data to GPU's global memory dominates in this process. These papers reported up to 10$\times$ speedup over FFTW.

Implementation of FFT for the NNLSE solver requires batched FFT where a Complex to Complex Fast Fourier Transform has to be implemented on every row of the input matrix. The CUFFT library provides a batched FFT function for 1-Dimensional data input which is suitable for the NNLSE solver [32].

To transform data to Fourier domain using both FFTW and CUFFT, two functions should be called. The first one creates a *Plan* to store all the information needed for the transformation and the second one *Executes* this plan on the input array. Using the CUFFT library means that in addition to these two functions, CUDA's *memcpy* function should be called to transfer data between CPU and GPU. Fig. 6.15 presents two plots, showing the run time of these two functions on CPU and GPU separately. The left plot presents the run time of FFT using the CUFFT library, where the data transfer time (blue bar) is significantly larger than run time. However, for NNLSE solver the initial data is produced in the GPU and kept in GPU's global memory for execution of a number of linear and non-linear steps. As a result no data transfer is needed for executing the Fourier transform. As for the plan creation, both the FFTW and the CUFFT implementation create a single plan once and then use it repeatedly for each step along the z-axis.

As a result, when comparing batched FFTW and CUFFT, only the run time of these two libraries is considered. This is shown in Fig. 6.16.

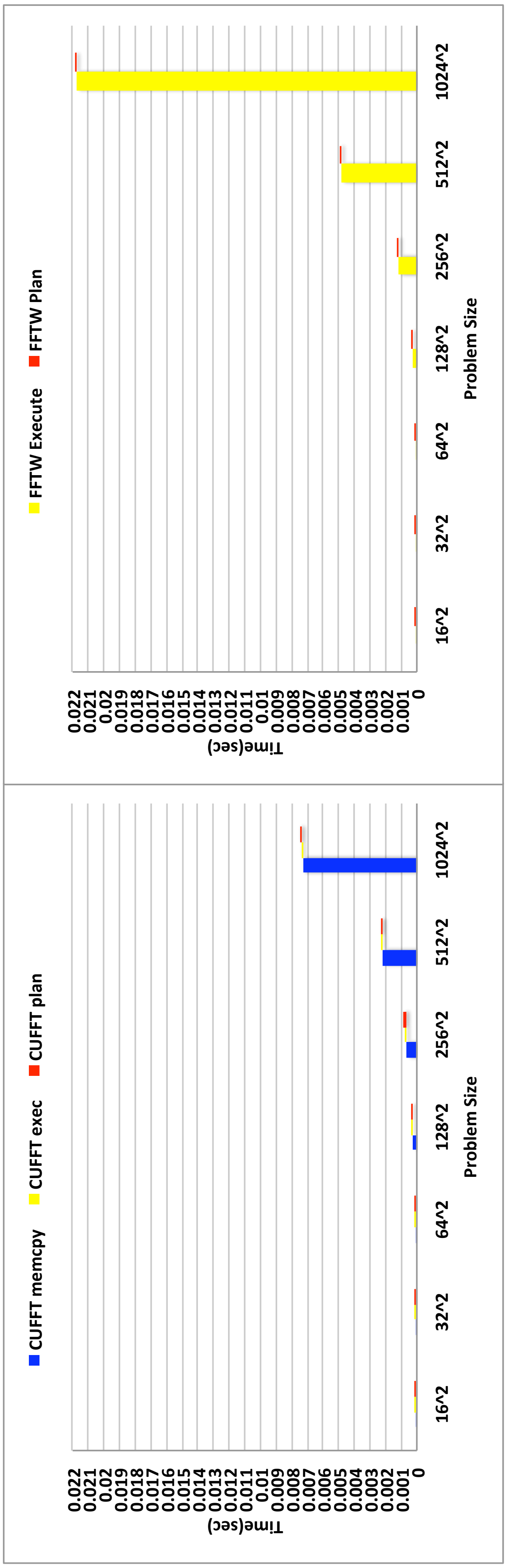


Figure 6.15: Left: Run time of all the components of a CUFFT module for various data sizes. Right: Run time of all the components of a FFTW module for various data sizes.

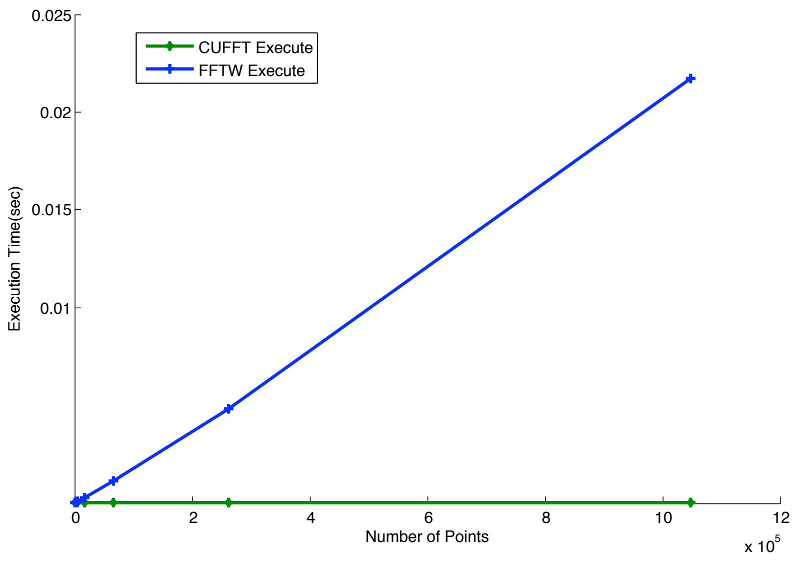


Figure 6.16: Average run time of the batched Fourier Transform employing CUFFT and FFTW library. This figure illustrates only the time taken to execute fast Fourier transform using the two different libraries and excludes data transfer and plan time.

6.3 Set Step

The Set Step method is implemented for calculation of the step size. As mentioned in section 4.3, femtosecond pulse propagates along the z-axis by step size Δz. This step size is computed by calculation of the maximum step size (Δz_{max}) obtained from all the linear and non-linear effects independently. The set step routine method requires implementation of a parallel module to find the maximum value in a given array. This is implemented using a parallel *Reduction* algorithm, to be discussed in the following section.

6.3.1 Reduction

A reduction algorithm is used whenever it is required to combine an input data set in some way to acquire a single answer. On a traditional computer this can be achieved by using a loop to go through all the elements serially and combine the data. This approach is not ideal

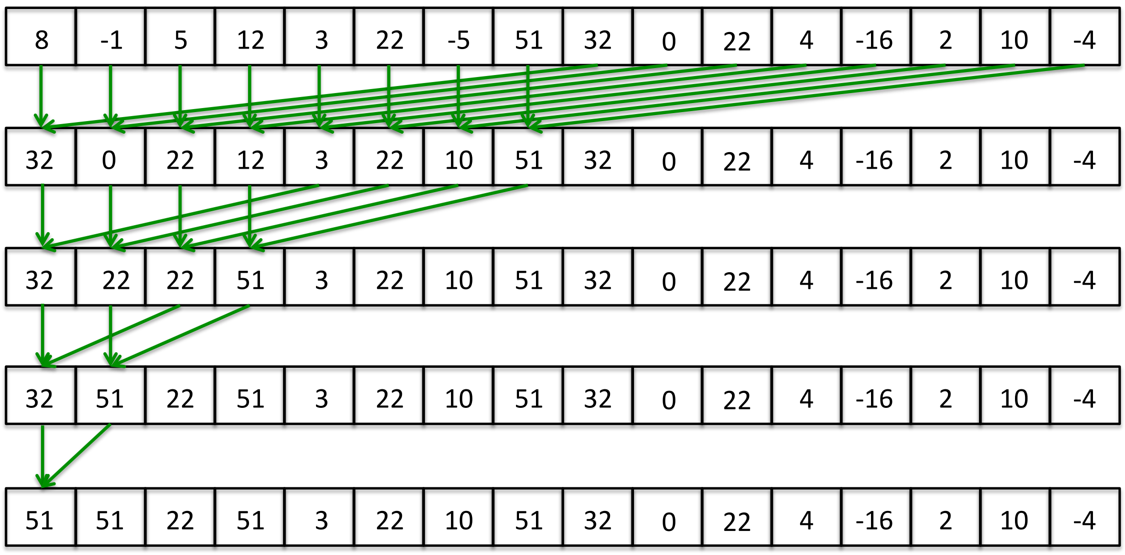


Figure 6.17: Execution of an efficient parallel reduction algorithm

on parallel hardware considering the essentially serial nature of these loops. To achieve a better performance on parallel hardware, a parallel reduction algorithm is used where each thread is responsible for combining a pair of data input. This will halve the number of data to be processed in a tree like approach. The pairwise combination will be executed step by step until a single result is obtained [85].

The CUDA SDK provides an optimised parallel reduction that calculates the sum of all the values in an array in parallel. This sample code uses an efficient tree-like approach in each thread block to reduce the input data. It divides the code into a number of kernels to provide global synchronisation between all the thread blocks and uses sequential addressing to avoid shared memory bank conflicts [86, 87].

As mentioned in section 2.6, GPU's shared memory is divided into 32-bit banks and each bank can only be accessed by one thread per instruction cycle. As a result, it is beneficiary to arrange the data such that consecutive data elements are stored in different banks of shared memory. This will result in spacing the data out so that none of them lie in the same bank.

Considering the fact that addresses $0, 16, 32, \ldots$ belong to bank 0, addresses $1, 17, 33, \ldots$ belong to bank 1 and so on, conflict-free shared memory access is guaranteed in this code using sequential addressing as illustrated in Fig. 6.17 [20, 87].

The NNLSE solver uses a version of the reduction algorithm where the pair of data are compared and the greater value is chosen. This will mean that at the end of the reduction, the maximum value stored in the array is obtained. For this, a modified version of the reduction code provided in the CUDA SDK was used [86].

After development and testing of all the modules required for implementation of the NNLSE solver in CUDA, these modules were put together to form the main two routines for calculation of the linear and non-linear terms. Comparison of these two implementations was performed in two different ways: accuracy and time efficiency. This comparison is described in detail in the next chapter.

Chapter 7

Comparison of Serial and Parallel GPU-based Implementation of the Non-linear Schrödinger Solver

In this chapter, the performance of the GPU-based parallel NNLSE solver is compared with a serial CPU version similar to that described in [9, 10]. These two implementations are compared based on their run time and accuracy.

Two platforms were used for this comparison:

- A GPU platform with a NVidia Tesla C1060, NVidia driver 195.36 and CUDA version 3.0;
- A CPU platform with Intel Core i7, 2.67 GHz, 6 GB RAM.

As mentioned in chapter 2, the NVidia card has 30 streaming multiprocessors (SM), where each multi-processor is comprised of 8 streaming processors (SP), providing a total of 240

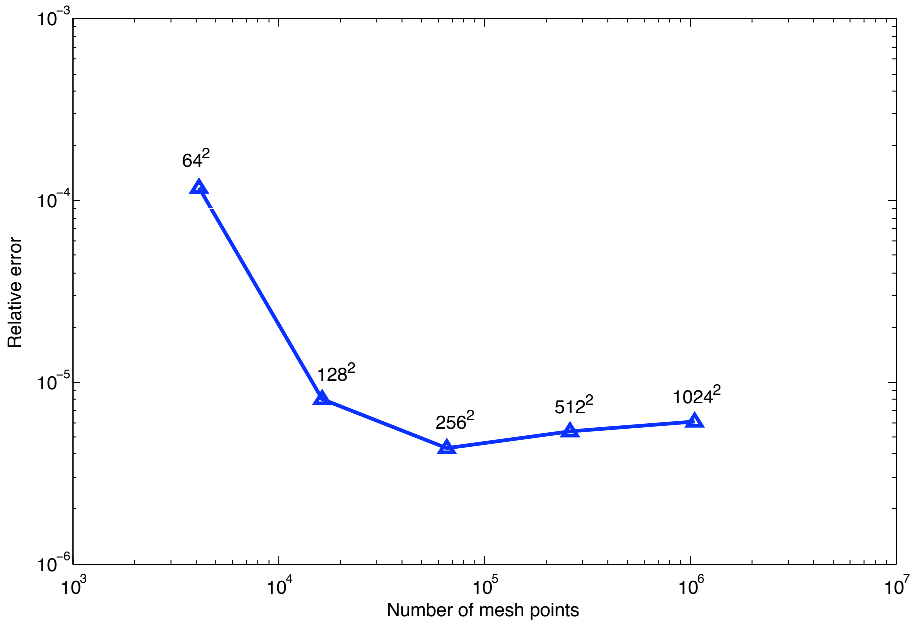


Figure 7.1: Relative error derived from comparison of results produced by CPU and GPU for different number of mesh points starting from $N_r \times N_t = 64 \times 64$ up to $N_r \times N_t = 1024 \times 1024$.

cores. A 16KB shared memory is assigned to each SM, providing a fast low-latency data cache. Additionally, there is 4GB of device memory available on the GPU. The results obtained from implementation of the NNLSE solver on these two platforms are compared here for their accuracy and run time.

In this research, the results obtained from the CPU implementation are considered to be exact and the parallel implementation approximates these results. As a result, the accuracy of the results is calculated from the difference between the results obtained from the CPU and the GPU implementation. This difference is computed by calculation of the absolute error defined by $|u_{cpu} - u_{gpu}|$, where indices cpu and gpu present results obtained from the serial NNLSE solver on CPU and the parallel NNLSE solver on GPU accordingly. When dealing with very small values of u_{cpu} and u_{gpu}, these points might cross zero which leads to inaccuracies in calculation of the absolute error. To overcome this problem and also obtain a unit-less error, the absolute error is normalised. Here relative error is obtained by normalisa-

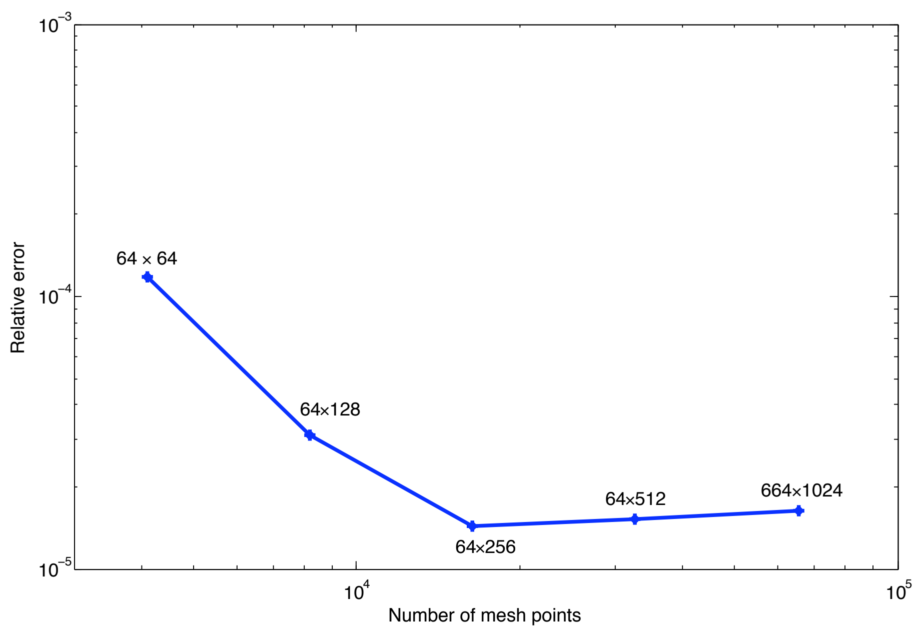


Figure 7.2: Relative error derived from comparison of results produced by CPU and GPU for different number of mesh points starting from $N_r \times N_t = 64 \times 64$ up to $N_r \times N_t = 64 \times 1024$

tion where the absolute error is divided by the peak intensity($|u_{cpu} - u_{gpu}|/\max|u_{cpu}|$) [88]. Figures 7.1 and 7.2, plot the maximum relative error calculated from the result of the NNLSE solver provided by CPU and GPU for square and non-square problems. Both the CPU and the GPU results are obtained from completion of ten split steps combining both the linear and non-linear routines in the NNLSE solver.

Fig. 7.1 illustrates the value of 10^{-3} for relative error of a very coarse resolution of 32^2. This figure shows that as the mesh density increases, the maximum error becomes very small in the range of 10^{-6} which is the arithmetic precision limitation. Since the implementation of this code differs in CPU and GPU due to the peculiar parallelisation of the GPU code, this figure proves that both methods converge with sufficiently high accuracy at finer resolutions. The same conclusion can be made for the non-square data shown in Fig. 7.2.

The robustness and accuracy of the parallel NNLSE solver is analysed by increasing the

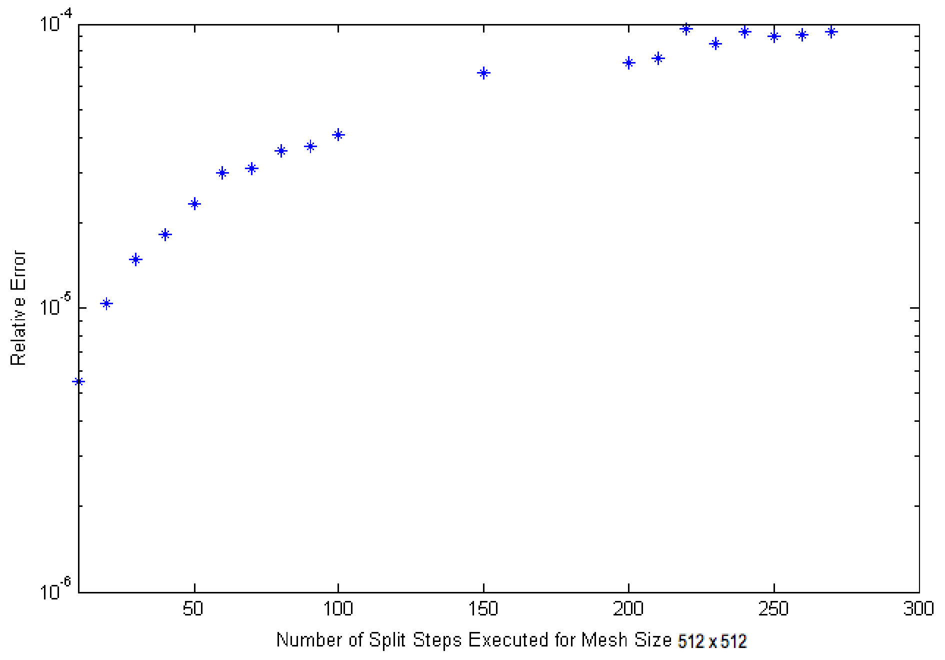


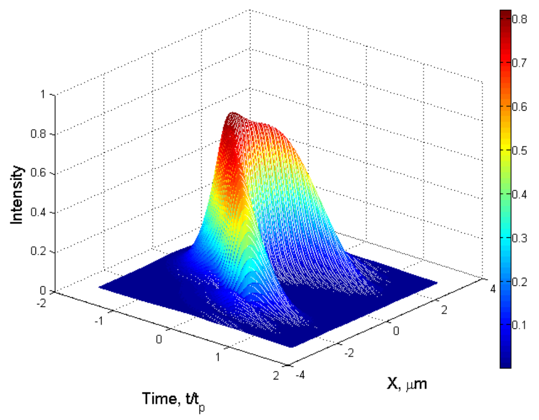
Figure 7.3: Relative error derived from comparison of results produced by CPU and GPU for increasing number split steps for mesh size 512^2

number of split steps along the z-axis as illustrated in Fig. 7.3. This figure shows that when increasing the number of steps, the difference between the results obtained from the CPU and GPU increases slightly. This is due to the difference in implementation of the numerical method for the non-linear step as well as the internal precision of CPU's floating point operations. This means that while preforming 32-bit single precision operations, the CPUs hardware will round all the 32-bit floats to its internal 80-bit long double. Then, again after execution of the operation, the number will be rounded back from 80-bit long double to 32-bit float. As a result, due to rounding the number twice, there will be a difference between the results obtained from the two different platforms. Using the newer series of CUDA-enabled GPUs, with double precision support, will provide better accuracy for the parallel NNLSE solver [89].

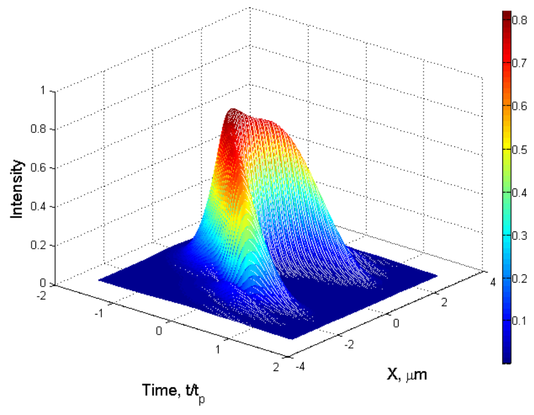
The results obtained from the parallel and serial NNLSE solver is verified by back to back testing of these two implementations as shown in figure 7.4. These figures illustrate normalised intensity dynamics of a femtosecond pulse at different points along the propagation axis z produced by the CPU and the GPU implementation. As it can be seen, these results illustrate the identical behavior of the femtosecond pulse intensity obtained from these two implementations. This set of figures show that as the pulse propagates in the transparent material, focusing causes the pulse to become narrow and sharp. Eventually the absorption happens right in the middle of the pulse, producing crescent like structure which merges again into a single pulse.

The run time of these two implementations is calculated by taking the wall clock time on both CPU and GPU. On the GPU platform this time includes the initial set-up and the data transfer time between CPU's host memory and GPU's global memory. Data transfer time is mainly for transferring the calculated result back to the CPU after completion of every split step.

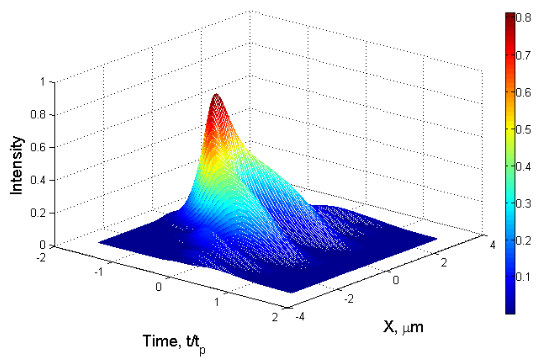
Timing of both the CPU and the GPU implementation was measured using the *gettimeofday* function declared in *sys/time.h* library in Linux. This function provides high-resolution timing with 1 micro second resolution. Fig. 7.5 illustrates the run time of the CPU and GPU implementation after completion of 10 split steps for various number of mesh points [90,91]. Fig. 7.5 shows that as the problem size increases, the GPU computing speed is notably increased providing $21\times$ speed-up for mesh size 1024^2 after execution of 10 steps. Here the speed-up is calculated by dividing the run time on CPU by the acquired run time on GPU. Fig. 7.6 shows the run time of the NNLSE solver on CPU and GPU for mesh size 256^2 for various number of steps starting from 10, up to 500 split steps. It can be concluded from that as the number of steps increases, the GPU/CPU speed up increases. This shows that after initial set up of the code (e.g. reading the initial parameter from file, transferring parameters to CUDA's constant parameter), increasing the number of steps will not increase GPU's run time significantly.



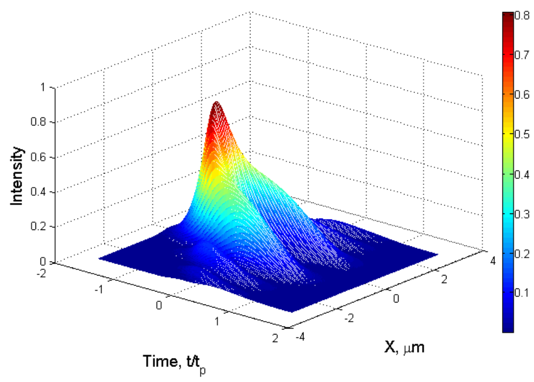
a) CPU, z = 0.1695mm



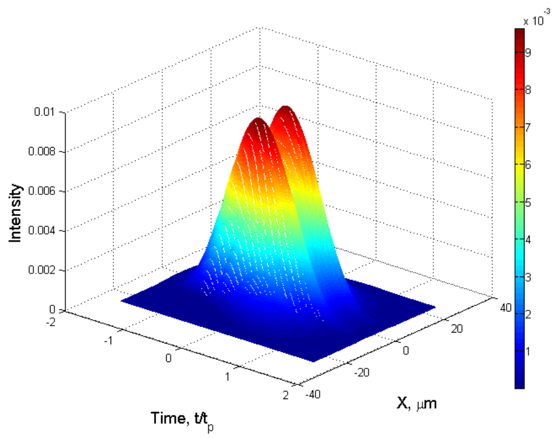
b) GPU, z = 0.1695mm



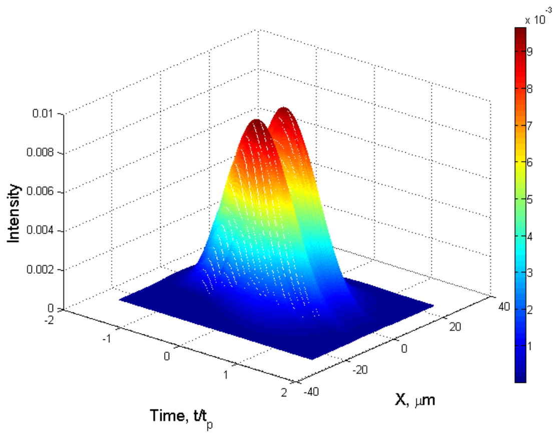
c) CPU, z = 0.1747mm



d) GPU, z = 0.1747mm



c) CPU, z = 0.2030mm



d) GPU, z = 0.2030mm

Figure 7.4: Evolution of the light intensity pattern along the z-axis. These results are obtained from the CPU and the GPU implementation of the NNLSE solver.

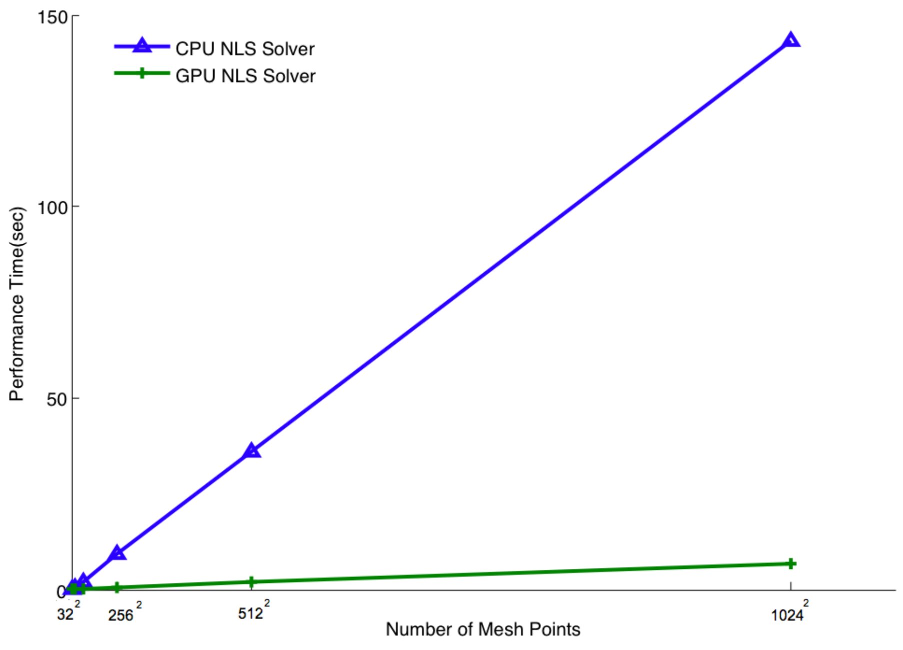


Figure 7.5: Run time of NNLSE Solver on CPU and GPU for 10 split steps

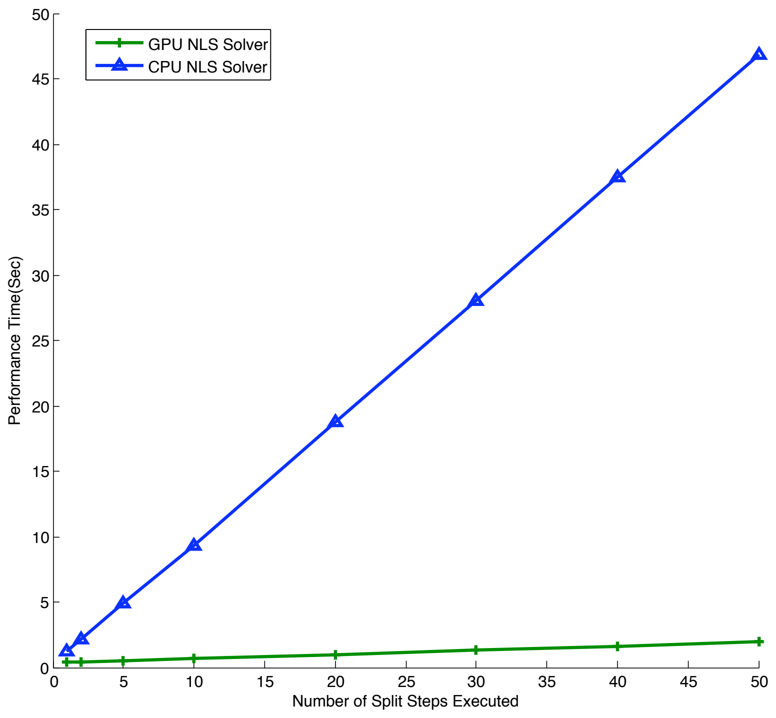


Figure 7.6: Run time of NNLSE Solver on CPU and GPU for various number of steps on mesh size 256^2

Chapter 8

Conclusions and future work

The main objective of this study is design and development of numerical algorithms based on multi-threaded parallel hardware. After extensive study of high performance computing platforms, NVidia's CUDA-enabled graphics card was chosen for parallel implementation of the specific problems addressed in this research.

In this study a platform-specific numerical solution for a physical problem appearing in a wide class of phenomena in non-linear optics is presented. This problem has recently been subject to several theoretical and numerical studies and appears in describing ultra-short pulse propagation in transparent media. The mathematical model describing this phenomenon, is a challenging and complex extended version of the non-linear Schrödinger equation. 3D numerical modeling of such problems is not feasible on modern workstations. However, the high performance computing based on the GPU multithreaded overcomes this limitation.

The numerical solution presented uses the split-step Fourier technique to solve the model by a problem specific succession of linear and nonlinear operators that approximate the original solution. The solution of both the linear and the non-linear operators requires parallelisation of some essentially serial algorithms and elimination of the numerical bottlenecks by exploiting the parallel capabilities of the specific hardware. The parallel solution of the linear problem is efficiently performed using spectral methods which employ efficient parallel numerical implementations of the fast Fourier transform. The non-local nature of the non-linear operator makes the parallel implementation challenging. Hence, special attention is paid to constructing an efficient parallel solution for the non-local non-linear operator.

The efficiency, accuracy and robustness of the above mentioned implementation is evaluated by back-to-back testing in a realistic context, reflecting typical experimental regimes of femtosecond laser inscription. All the work is proof principle on single precision floating point platform which has limit of 10^{-6} due to arithmetic truncation. This figure scales down on newer versions of the hardware platform supporting double precision. The run time of the new GPU implementation showed $25\times$ speed-up for mesh size 512^2 for the propagation of a femtosecond pulse along the z-axis.

Future work could include execution of the specific problem on double precision hardware for higher accuracy or modification of the numerical implementation for execution on multi-GPUs/ GPU clusters to achieve faster run time. Moreover, the approach developed in this study can be applied to numerical modeling of a wide range of physical problems. This model could be applicable to numerical modeling of dispersion-managed solitons subject to [92] which is based on non-linear Schrödinger type equation. Also numerical simulations similar to that described in [93] could benefit from this model for numerical simulation of self-focusing lasers in silica.

References

[1] M. Baregheh, V. Mezentsev, and H. Schmitz, "Parallel simulation of non-local non-linear schroedinger systems using multithreaded graphical processing unit." The International Workshop on Optical Waveguide Theory and Numerical Modelling, 2010.

[2] M. Baregheh, V. Mezentsev, and H. Schmitz, "Multi-threaded parallel simulation of non-local non-linear problems in ultrashort laser pulse propagation in the presence of plasma," *Proc. SPIE* **8071**, p. 807115, 2011.

[3] A. Dotovalov, S. Babin, M. Baregheh, M. Dubov, and V. Mezentsev, "Comparative numerical study of efficiency of energy deposition in femtosecond microfabrication with fundamental and second harmonics of yb-doped fiber laser," *Proc. SPIE* **7914**, p. 791432, 2011.

[4] M. Baregheh, V. Mezentsev, and H. Schmitz, "Multi-threaded parallel numerical modellung of femtosecond pulse propagation in laser machining," in *CLEO/Europe and EQEC 2011 Conference Digest, OSA Technical Digest (CD) (Optical Society of America)*, 2011.

[5] A. Dotovalov, S. Babin, M. Baregheh, M. Dubov, and V. Mezentsev, "Efficiency of energy deposition by fundamental and second harmonics in femtosecond laser inscription," in *CLEO/Europe and EQEC 2011 Conference Digest, OSA Technical Digest (CD) (Optical Society of America)*, 2011.

[6] M. Baregheh, V. Mezentsev, and H. Schmitz, "Multi-threaded parallel simulation of non-local non-linear pulse propagation on gpus." International Supercomputing Conference, 2011.

[7] A. Dotovalov, S. Babin, M. Dubov, M. Baregheh, and V. Mezentsev, "Comparative numerical study of energy deposition in femtosecond laser microfabrication with fundamental and second harmonics of yb-doped laser," in *Laser Physics*, 2011. Accepted.

[8] M. Baregheh, V. Mezentsev, and H. Schmitz, "High performance numerical modeling of ultra-short laser pulse propagation based on multithreaded parallel hardware." Journal paper, draft in Progress, 2011.

[9] V. Mezentsev, J. Petrovic, J. Dreher, and R. Grauer, "Adaptive modeling of the femtosecond inscription in silica," *Proc. SPIE, Laser-based Micropackaging* **6107**, 2006.

[10] V. Mezentsev, J. Petrovic, M. Dubov, I. Bennion, J. Dreher, H. Schmitz, and R. Grauer, "Femtosecond laser microfabrication of subwavelength structures in photonics," *Proc. SPIE, Laser-based Micropackaging* **6459**, p. 64590B, 2007.

[11] A. Grama, A. Gupta, G. Karypis, and V. Kumar, *Introduction to Parallel Computing*, Pearson Education Limited, Edinburgh Gate Harlow, Essex CM20 2JE, England, second ed., 2003.

[12] V. E. with Edmond Chow, "Introduction to high-performance scientific computing." public draft, 2011. URL: http://tacc-web.austin.utexas.edu/staff/home/veijkhout/public_html/istc/istc.html.

[13] M. J. Flynn and K. Rudd, "Parallel architectures," *ACM Computing Surveys* **28(1)**, pp. 67–70, 1996.

[14] C. Quammen, "Introduction to programming shared-memory and distibuted memory parallel computers." Technical report, 2005.

[15] A. J. van der Steen, "hpc architecture." URL : http://www.phys.uu.nl/ steen/web08/architecture.html.

[16] K. Fatahalian and M. Houston, "A closer look at gpus," *Communications of the ACM* **51(10)**, pp. 50–57, October 2008.

[17] A. Vajda, *Programming Many-Core Chips*, Springer Science Business Media, 2011.

[18] "Nvidia's next generation cuda compute architecture fermi," 2009. URL:http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Archite

[19] J. Layton, "Gpu programming for the rest of us," 2009. URL : http://www.clustermonkey.net/content/view/248/33/1/0/.

[20] "Nvidia cuda programming guide." Nvidia Corporation. URL : http://developer.nvidia.com/.

[21] "opencl." URL : http://www.khronos.org/opencl/.

[22] "Geforce 8 series." URL : http://en.wikipedia.org/wiki/GeForce_8i_Series.

[23] "Processors, intel microprocessor export compliance metrics." URL : http://www.intel.com/support/processors/sb/cs-023143.htm.

[24] "Cuda c best practice guide," 2010. http://www.nvidia.co.uk/page/home.html.

[25] "Nvidia tesla computing processor solving tomorrow's problems today," 2010. URL: http://www.nvidia.com/docs/IO/43395/NV_DS_Tesla_C1060_US_Jan10_lores_r1.pdf.

[26] "Tesla m-class gpu computing modules accelerating science," 2011. URL:http://www.nvidia.com/docs/IO/105880/DS-Tesla-M-Class-Aug11.pdf.

[27] "Looking beyond graphics," 2009. URL : http://www.nvidia.com/content/PDF/fermi_white_papers

[28] W. W. Hwu, C. Rodrigues, S. Ryoo, and J. Stratton, "Compute unified device architecture application suitability," *Computing in Science and Engineering* **11(3)**, pp. 16–26, 2009.

[29] D. Kirk and W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Morgan Kaufmann, Burlington, MA 01803, USA, 2010.

[30] S. Ryoo, C. I. Rodrigues, S. S. Baghsorkhi, S. S. StoneE, D. B. Kirk, and W. W. Hwu, "Optimization principles and application performance evaluation of a multithreaded gpu using cuda," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 73–82, 2008.

[31] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick, *The landscape of parallel computing research: A view from Berkeley*, Technical Report No.UCB/EECS-2006-183, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, 2006.

[32] "Cuda cufft library." Nvidia Corporation. URL : http://developer.nvidia.com/.

[33] L. Nyland, M. Harris, and J. Prins, *Fast N-Body simulation with CUDA, GPU Gems 3*, Addison Wesley, 2007.

[34] "Monte-carlo option pricing." Nvidia Corporation. URL : http://developer.download.nvidia.com/compute/cuda/sdk/website/projects/MonteCarlo/doc/Monte

[35] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, and K. Skadron, "A performance study of general-purpose applications on graphics processors using cuda," *Journal of Parallel and Distributed Computing* **68(10)**, pp. 1370–1380, 2008.

[36] A. J. D. Maria, D. Stester, and H. Heynau, "Self mode-locking of lasers with saturable absorbers," *Appl. Phys. Lett.* **8**, pp. 174–176, 1966.

[37] U. Keller, "Recent developments in compact ultrafast lasers," *Nature* **424**, pp. 831–838, 2003.

[38] K. Ueda and K. L. Ishikawa, "Attoclocks play devils advocate," *Nature Physics* **7**, pp. 371–372, 2011.

[39] C. B. Shaffer, *Interaction of Femtosecond Laser Pulses with Transparent Materials*. PhD thesis, Hardvard University, Cambridge, Massachusetts, 2001.

[40] "Lectures on ultra-short-pulse laser technology and applications." URL: http://info.tuwien.ac.at/tubiomed/de/aktuell/.

[41] N. Bloembergen, "Laser induced electric breakdown in solids," *IEEE journal of Quantum Electronics* **10(3)**, pp. 357–386, 1974.

[42] J. H. Marburger, "Self focusing theory," *Progress in Quantum Electronics* **4**, pp. 35–110, 1974.

[43] D. Arnold and E. Cartier, “Theory of laser induced free electron heating and im pact ionization in wide band gap solids,” *Physical Review B (Condensed Matter)* **64(23)**, pp. 15102–15, 1992.

[44] W. Smith, J. H. Bechtel, and N. Bloembergen, “Picosecond laser induced breakdown at 5321 and 3547 observation of frequency dependent behavior,” *Physical Review B (Solid State)* **15(8)**, pp. 4039–55, 1977.

[45] L. Cerami, E. Mazur, S. Nolte, and C. B. Schaffer, “Femtosecond laser micromachining,” 2007. URL: http://mazur.harvard.edu/publications.php?function=latest&ty=pre.

[46] R. R. Gattass and E. Mazur, “Femtosecond laser micromachining in transparent materials,” *Nature Photonics* **2(4)**, pp. 219–225, 2008.

[47] G. P. Agrawal, *Nonlinear Fiber Optics*, Academic Press, Burlington, MA ; London, 1951.

[48] L. Bergé, S. Skupin, R. Nuter, J. Kasparian, and J.-P. Wolf, “Ultrashort filaments of light in weakly-ionized, optically-transparent media,” *Reports on Progress in Physics* **70**, pp. 1633–1713, 2007.

[49] Q. Feng, J. V. Moloney, A. Newell, E. Wright, K. Cook, P. Kennedy, D. Hammer, B. Rockwell, and C. Thompson, “Theory and simulation on the threshold of water breakdown induced by focused ultrashort laser pulses,” *IEEE Journal of Quantum Electronics* **33, No. 2**, pp. 127–137, 1997.

[50] L. V. Keldysh, “Gpu accelerated fully space and time resolved numerical simulations of self-focusing laser beams in sbs-active media,” *Journal of Computational Physics* **20**, p. 1307, 1945 (1964).

[51] M. D. Feit and J. A. Fleck, “Effect of refraction on spot size dependence of laser induced breakdown,” *Appl. Phys. Lett.* **24**, pp. 169–172, 1974.

[52] T. R. Taha and M. J. Ablowitz, “Analytical and numerical aspects of certain nonlinear evolution equations. ll. numerical, nonlinear schroedinger equation,” *Journal Of Computational Physics* **55**, p. 203, 1984.

[53] T. R. Taha and M. J. Ablowitz, “Analytical and numerical aspects of certain nonlinear evolution equations. I. analytical,” *J. Comput. Phys.* **55**, 1984.

[54] T. R. Taha and M. J. Ablowitz, “Analytical and numerical aspects of certain nonlinear evolution equations. ii. numerical, nonlinear schroedinger equation,” *J. Comput. Phys.* **55**, 1984.

[55] J. M. AlcarazPelegrina and P. RodriguezGarcia, “Simulations of pulse propagation in optical fibers using graphics processor units,” *Computer Physics Communications* **182**, pp. 1414–1420, 2011.

[56] S. Zoldi, V. Ruban, A. Zenchuk, and S. Burtsev, “Parallel implementations of the split-step fourier method for solving nonlinear schroedinger systems,” in *SIAM News*, pp. 8–9, 1999.

[57] S. Hellerbrand and N. Hanik, “Acceleration of the split-step fourier method by using a graphics processing unit(gpu),” 2009. URL : http://www.lnt.ei.tum.de/mitarbeiter/hellerbrand/media/HellerbrandS_ SplitStep-FourierMethodGPU_ TUMReport2009.pdf.

[58] S. Hellerbrand and N. Hanik, “Fast implementation of the split-step fourier method using a graphics processing unit,” *Proc. of OFC/NFOEC* **182**, pp. 1–3, 2010.

[59] K. Germaschewski, R. Grauer, L. Bergé, V. K. Mezentsev, and J. J. Rasmussen, “Splittings, coalescence, bunch and snake patterns in the 3d nonlinear schroedinger equation with anisotropic dispersion,” *Physica D* **151**, pp. 175–198, 2001.

[60] K. F. Riley, M. P. Hobson, and S. J. Bence, *Mathematical methods for physics and engineering*, Cambridge University Press, third ed., 2006.

[61] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannerye, *Numerical Recipes in C++, The Art of Scientific Computing*, Cambridge University Press, Cambridge, CB2 2RU, UK, second ed., 2003.

[62] F. Abdullaev and V. V. Konotop, *Nonlinear waves: classical and quantum aspects*, Kluwer Academic Publishers, p.O. Box 17, 330 aa Dordrecht, The Netherlands, 2004.

[63] H. Friedel, R. Grauer, and C. Marliani, “Adaptive mesh renement for singular current sheets in incompressible magnetohydrodynamic ows,” *J. Comp. Phys.* **134**, pp. 190–198, 1997.

[64] B. S. W. Schroder, *A Workbook for Differential Equations*, John Wiley and Sons, Inc.,, Hoboken, New Jersey, 2010.

[65] M. Harris, “Parallel prex sum (scan) with cuda,” 2007. Technical Report,NVIDIA Corporation.

[66] G. E. Blelloch, “Prefix sums and their applications,” 1990. Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University.

[67] G. E. Blelloch, “Scans as primitive parallel operations,” *IEEE Transactions on Computers* **38(11)**, pp. 1562–1538, 1989.

[68] “Cuda parallel prefix sum.” URL:http://developer.nvidia.com/cuda-cc-sdk-code-samples#scan.

[69] L. H. Thomas, “Elliptic problems in linear different equations over a network,” 1949. Watson Sci.Comput. Lab. Rept.

[70] R. W. Hockney, “A fast direct solution of poissons equation using fourier analysis,” *J. ACM* **12**, pp. 95–113, 1965.

[71] H. S. Stone, “An effcient parallel algorithm for the solution of a tridiagonal linear system of equations,” *J. ACM* **20**, pp. 27–38, 1973.

[72] H. S. Stone, “Parallel tridiagonal equation solvers,” *ACM Transactions on Mathematical Software (TOMS)* **1(4)**, pp. 289–307, 1975.

[73] B. Buzbee, G. Golub, and G. Nielsen, "On direct method for solving poissons eqations," *SIAM J. Numer. Anal.* **7**, pp. 627–655, 1970.

[74] O. Buneman, *A compact Non-iterative Poisson Solver*, Rep. 294, Inst. for Plasma Res., Stanford, California, 1969.

[75] G. E. Karniadakis and R. M. Kirby, *Parallel scientic computing in C++ and MPI*, Cambridge University Press, The Pitt building, Trumpington Street, Cambridge, United Kingdom, 2003.

[76] D. Kershaw, *Solution of a single tridiagonal linear systems and vectorization of the iccg algorithm on the cray-1. Parallel Computations*, G. Rodrigue, Ed., Academic Press, Inc., New York, 1982.

[77] J. Lambiotte and R. Voigt, "The solution of tradiagonal linear systems on the cdc star-100 computer," *ACM Trans., Math Software* **1(4)**, pp. 291–312, 1975.

[78] C. L. Cox, "Implementation of a divide and conquer cyclic reduction algorithm on the fps t-20 hypercube," in *In Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, pp. 1532–1538, 1988.

[79] Y. Zhang, J. Cohen, and J. D. Owens, "Fast tridiagonal solvers on the gpu," pp. 127–136, 2010.

[80] F. M. Johnson, "Fftw." URL: http://www.fftw.org.

[81] H. Tomono, M. Aoki, and T. Tsumuraya, "Gpu based acceleration of first principles calculation," *Journal of Physics: Conference series* **215 012121**, 2010.

[82] "Cufft1.1/2.0 vs fftw 3.1.2(x86-64) vs fftw 3.2 (cell) comparion." URL : http://sharcnet.ca/ merz/CUDA_benchFFT/.

[83] K. Despain, "Fast fourier transforms(ffts) and graphical processing units(gpus)." URL :.

[84] "Gpu benchmarking," 2007. URL : http://www.cv.nrao.edu/ pdemores/gpu.

[85] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with cuda," *Queue* **6(2)**, p. 4053, 2008.

[86] "Cuda parallel reduction." URL: http://developer.nvidia.com/cuda-cc-sdk-code-samples#reduction.

[87] M. Harris, "Optimizing parallel reduction on cuda." URL: http://developer.download.nvidia.com/compute/cuda/1_1/Website/projects/reduction/doc/reductio

[88] C. L. Dym and E. Ivey, *Principles of mathematical modeling*, Elsevier Academic Press, Burlington, USA, 2003.

[89] H. Huan, "Note on floating-point number numerics," 2010. URL:http://home.uchicago.edu/h̃huan/floating.pdf.

[90] "High-resolution calender." URL: http://www.delorie.com/gnu/docs/glibc/libc_434.html.

[91] D. Kalev, "Using high-resolution timers." URL : http://www.devx.com/cplus/Article/35375/1954.

[92] S. K. Turitsyn, B. G. Bale, and M. P. Fedoruk, "Dispersion-managed solitons in fibre systems and lasers," *Elsevier* **521**, pp. 135–203, 2012.

[93] S. Mauger, G. de Verdire, and S. S. L. Berg, "Dispersion-managed solitons in fibre systems and lasers," 2012. In Press, Accepted Manuscript.