University of Manchester
School of Electrical and Electronic Engineering
Project Report May 2016

GPGPU: Acceleration of output of data

Author: Loukas Xanthos

University ID of Author: 9408845

Supervisor: Dr. Fumie Costen

Abstract

GPGPU: Acceleration of output of data

Author: Loukas Xanthos

The aim of this project is to speed up the GPGPU accelerated FD-FDTD computation from the perspective of the output of data.
The data output stage of a piece of software which performs the FD-FDTD computation on a NVIDIA Kepler GPGPU accelerator was optimised for the output of the field data at points on a single line parallel to one side of the 3D cubic FDTD simulation space, for the output of the field data at points on more than one non-consecutive such lines, for the output of the field data at points on a single plane parallel to one of the faces of the 3D cubic FDTD simulation space, for the output of the field data at points on more than one non-consecutive such planes, and for the output of field data at selected non-consecutive discrete points in the 3D cubic FDTD simulation space. The performance improvement for each case is presented quantitatively and qualitatively and discussed.
The introduction of one buffer array into the device memory is proven to be beneficial for all the aforementioned cases, increasing the speed of the output stage of the FD-FDTD software. Also, the introduction of one buffer array into the host main memory further improves the speed of the output stage. These two techniques exploit the performance boost provided by device memory coalescence, elimination of strided host memory access and modern compiler optimisations.
In the conclusions, possible future work for computationally similar projects adopting the methodologies presented throughout this report is outlined.

Supervisor: Dr. Fumie Costen

Acknowledgements

I would like to thank my supervisor, Dr. Fumie Costen for her perpetual support of my academic efforts and encouragement. I consider myself very lucky to have been her tutee and I owe her my sincere gratitude. Also, I would like to thank RIKEN in Saitama, Japan, for providing me with the computational environment which was necessary for my project. Furthermore, I would like to thank our systems administrator Mr. Keith Williams for building and maintaining a robust and safe computer environment that facilitates the secure connection to the RIKEN Login nodes via the servers in Dr. Fumie Costen's office at the University of Manchester. Finally, I would like to thank my family: my mother Amalia, my father Pavlos, and my beloved brother Nikolaos–Georgios for supporting my studies at the United Kingdom and for always being there when I need them.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Introduction

Nowadays there is a growing need for efficient modelling and simulation software, given the increasing complexity of engineering and scientific problems faced by the academia and industry. Hence, modern simulation software attempts to exploit modern hardware at the maximum degree possible, including the use of advanced features of modern Central Processing Units (CPUs) and Graphics Processing Units (GPU). This tendency has made General Purpose GPU processing (GPGPU) a popular solution for the parallel execution of scientific and engineering computations.

This project examines the possible further optimisation of the parallelised version of a piece of software, which implements a modified version of the Finite-Difference Time-Domain (FDTD) [1] method for the solution of the Maxwell equations for the simulation of propagation of electromagnetic (EM) waves in the 3D space. This piece of software is adapted for execution on NVIDIA accelerators from its former version that only made use of the CPU and message-passing interface technologies. Nevertheless, the current implementation of this piece of software for execution on the GPU is found to be computationally expensive at its data output stage and needs to be further optimised by the introduction of ad-hoc data output algorithms.

1.2 Motivation

The main use of the FDTD method is the simulation of propagation of EM waves in the 3D space [2]. However, this method is also employed for the production of computer simulations of computationally similar phenomena, in the scientific fields of biomedical engineering [3] and geophysics [4]. In addition, a computationally similar method is employed for the modelling and solution of Computational Fluid Dynamics (CFD) prob-

lems [5]. Thus a comprehensive examination of the possibilities of speeding up the FDTD computation can be beneficial for a big range of applications.

The author was provided with a parallelised and vectorised Frequency-Depended FDTD ((FD)FDTD) software (hereinafter "the (FD)FDTD software"), which was developed by Dr. Fumie Costen's research group at the University of Manchester, with Dr. Costen's group reporting that more than half the total execution time of the software is spent on the data output stage. At the output stage of the (FD)FDTD software the field data located in the memory of the accelerator device are transferred into the system's main memory. Moreover, there are cases where the (FD)FDTD software user requires the simulation results for points located on just one line or just one plane of the FDTD space, or on more lines or planes of the FDTD space, or merely on some selected points. Therefore not all points in the FDTD space are always of importance to the (FD)FDTD software user and hence, the output of the field data for the whole FDTD space can be inefficient.

In addition, such use cases for GPU accelerated versions of the (FD)FDTD method are not considered in the relevant literature. Hence, an examination of the solution of the problem of accelerating the data output stage of the (FD)FDTD method, even for specific use cases, can provide a speed up to the execution of the (FD)FDTD software and also useful information for computationally similar (memory throughput – limited) projects, not only from the electrical engineering sector.

1.3 The Aims and Objectives of this project

1.3.1 Aims

"The aim of this project is the acceleration of the FD-FDTD computation on the GPU from the view point of the data output" [6].

1.3.2 Objectives

- Become familiar with Linux environments, with the RIKEN HOKUSAI GreatWAVE cluster, with the awk, sed and gnuplot utilities, with Fortran 90 and CUDA Fortran
- Adaptation of the given FD-FDTD software to make it compatible with the RIKEN HOKUSAI GreatWAVE platform
- Implementation of a wall-time measuring mechanism into the given software
- Development of benchmarking and performance data analysis software

- Identification of the main methods used for the acceleration of data transfers between accelerator hardware (GPU) and main memory (CPU)
- Development of algorithms for the acceleration of the output of FDTD field data located on: a single line of the FDTD space, more than one lines of the FDTD space, a single plane of the FDTD space, more than one planes of the FDTD space, selected points of the FDTD space
- Comprehensive examination of the performance of the above algorithms

Chapter 2

Background

2.1 GPGPU

Nowadays, with GPU cards offering a high number of graphics processing cores, GPUs are not only used for the output and processing of graphics; they are also used by scientists and engineers for the acceleration of parallelised and vectorised software. Therefore accelerator hardware accommodating General Purpose GPUs (GPGPUs) is produced by major GPU ventors, which apart from its ability to process graphics data and to output them to displays, it can also be programmed to perform scientific computations, including floating-point arithmetic.

Accelerators are multi-core devices, commonly facilitating thousands of processing cores, which are designed with a Single Instruction Multiple Data (SIMD) architecture. The main difference of accelerators from CPUs lies in the ability of GPGPUs to optimally and quickly process a simple operation performed on thousands of data elements concurrently, whereas CPUs are mainly designed to quickly perform complex operations on single data elements. Therefore, GPUs cannot compete with CPUs in the execution of serial tasks or single-threaded operations, as modern CPUs perform these tasks much faster. Moreover, the memory installed on GPGPU accelerator cards is often of smaller data capacity than that of the main memory of a computer machine.

2.1.1 NVIDIA Kepler K20X - Architecture

This project employs an NVIDIA Kepler K20X accelerator device to run parallelised and vectorised software. NVIDIA Kepler K20X accelerators make use of one NVIDIA Kepler GK110 GPU device (which is referred to as "the device"). The NVIDIA Kepler GK110 GPU has been designed for fast double precision computing performance [7].

The GK110 GPU device which is installed on Kepler K20X accelerators consists of 15 streaming multiprocessors, called "SMX units" [8], and six 64-bit memory controllers [7]. Each of the SMX units has 192 single-precision "CUDA cores", which are equipped with fully pipelined integer arithmetic and IEEE 754-2008 compliant single- and double-precision arithmetic floating-point logic units. An SMX schedules threads waiting for execution in groups of 32 parallel threads. These groups of 32 parallel threads must all execute one common instruction, and form a "warp" [9, pp. 69-70]. For example, a warp could perform the numerical instruction of addition of a constant number to 32 consecutive or non-consecutive GPU memory locations. Warps are scheduled for execution by four warp schedulers residing on the SMX unit, and eight "instruction dispatch units" inside an SMX unit allow the issue and interleaved execution of four warps. However, in GK110 there is a limit of maximum 64 warps per multiprocessor and a limit of 2048 threads per multiprocessor [7].

2.1.2 NVIDIA Kepler K20X - Device memories

The memory of the accelerator hardware is unified. The addressable memory residing on the accelerator hardware is called the "device memory", whereas the main memory of the computer system is called the "host memory". The device memory is organised into different memory units, the access of which is controlled by the device hardware, namely "global memory", "local memory", "shared memory", "constant memory" and "texture and surface memory". In addition, NVIDIA also provides a way of allocating and using a part of the host memory by Direct Memory Access (DMA), called the "pinned" or "page-locked" memory. The device memory can only be accessed only via 32-, 64- or 128-byte memory transactions and these transactions must be physically aligned, *i.e.* only the 32-, 64- or 128-byte segments of device memory whose first address is a multiple of their size can be accessed by device memory transactions (thus, the data segments need to be "naturally aligned") [9, p. 79]. The device memory of the NVIDIA Kepler K20X accelerator has a total capacity of 6GB [10]. The GPU chip also provides a two-level instruction and memory cache hierarchy.

"Global memory" (also referred to as the "device memory" in CUDA programming context) is the largest memory which is visible to the host and can be accessed by it; it resides off the GPU chip and it is cached in L2 cache [11]. The accelerator NVIDIA

Kepler K20X offers up to 6GB of global memory. Every time an instruction that accesses global memory is executed in a warp, the warp coalesces the memory access of the threads within the warp into an optimal number of memory transactions, based on the size of the word accessed by each thread and "the distribution of these memory addresses across each thread" [9, p. 79]. Evidently, the programmer must ensure that instructions which need to access the global memory perform the optimum (smallest) number of coalesced memory transactions. For example, the device may issue a 32-byte transaction (it cannot be of less capacity) for each thread in a warp, if each thread needs to access 8 bytes of global memory; thus, the instruction throughput is divided by $\frac{8}{32} = 4$, as the execution of the instruction shall last 4 times longer than necessary due to the time needed for the completion of the data transactions, which are 4 times bigger than needed by each thread.

In addition, global memory instructions can only read or write words of size 1, 2, 4, 8 or 16 bytes [9, p. 80]. For any command that accesses data residing in global memory to compile to a single data transaction instruction, their data type needs to be 1-, 2-, 4-, 8- or 16-bytes long and "naturally aligned". Otherwise, the command compiles to multiple instructions that perform non-coalesced overlapped data access patterns. The built-in Fortran data types employed in the (FD)FDTD software are at most 8 bytes long, and therefore the access of such data fulfils this alignment requirement.

The maximisation of global memory throughput is a main point of concern of this project, as the 3D arrays which accommodate the data of the electromagnetic fields in the (FD)FDTD software and which are desired to be outputted, reside in the accelerator's global memory. For the maximisation of global memory throughput, the maximisation of the coalescing is important [9, p. 79], which can be achieved by using data types that conform to the alignment requirements, by data padding and/or by patterned (aligned) memory access.

Page-locked memory (also called "pinned host memory") is an allocated memory segment of host memory, which is isolated from the operating system's paging mechanism. This kind of memory achieves the highest theoretical bandwidth between the host and the device. However, pinned host memory is a "scarce resource" [12, p. 72], it is not always available [13, p. 24] and is highly depended on the operating system and the specific application under development [12, p. 72]. As this project tries to achieve its aims using methods that can be employed by similar projects, independently of specific operating system behaviour, the use of this kind of memory is not considered in this project.

2.1.3 NVIDIA CUDA

Compute Unified Device Architecture (CUDA) is an architectural model and framework for general-purpose GPU development which was released by NVIDIA in 2007[14]. The (FD)FDTD software of this project is developed in the CUDA Fortran programming interface, but CUDA also supports other programming interfaces and languages such as C, C++, DirectCompute and OpenACC [9, p. 4].

CUDA introduces the concept of the CUDA "kernel". Kernels are source code segments that accept parameters and behave like C functions. However, a kernel is executed N times concurrently, by N CUDA threads, conversely to C functions that execute in a serial manner [9, p. 9].

In addition, CUDA provides a mechanism of abstractions to CUDA developers. A key abstraction is the hierarchy of thread groups. A multithreaded CUDA program is partitioned into blocks of threads. CUDA threads can be identified by their "thread index", which can be 1D, 2D or 3D. Thus, they can form a 1D, 2D or 3D block of threads, called a "thread block" [9, p. 10]. Blocks are organised into one 1D, 2D or 3D "grid" of thread blocks. The number of thread blocks in a grid can exceed the number of thread processors. The dimensions of thread blocks and of the grid of blocks are specified on each kernel invocation statement of CUDA applications.

Threads within a block can share data through "shared memory" (which is not accessible by the host) and their execution is scheduled for optimal execution by the hardware, according to the memory accesses the threads perform. The execution of threads can also be explicitly synchronised by software, via API calls [9, p. 12]. Conversely, thread blocks do not share access to memory segments and must must be able to execute independently [9, p. 12], *i.e.* no assumption must be made about the order of their execution.

2.2 Factors that delay device-to-host transfers: The PCI bottleneck and strided access of multidimensional arrays

Modern GPU accelerators are attached to hosts using the PCI-Express (PCIe) interface. The PCIe interface features a raw bit rate of 8GT/s and 8Gb/s/lane/link [15]. Therefore, the low PCIe bandwidth becomes a bottleneck in cases of frequent device-to-host or host-to-device data transactions, as threads performing such data transfers suspend until these transfers are completed. Evidently, programmers need to achieve a high device occupancy, *i.e.* to execute a big number of threads, so that the memory transfer operations performed by the device are much fewer than any other kind of processing operations that take place

in the device cores at any moment in time [16][17][18].

In addition, device-to-host and host-to-device transactions can benefit from the CPU cache. In such transactions, data must be transferred to or from the host main memory, with the intervention of the CPU (with the exception of pinned memory transactions, which are outside the scope of this report). The transactions of adjacent main memory addresses result in more cache hits, which means that a higher percentage of the requested data can be found in a CPU cache, and thus they can be fetched into an instruction or be updated by an instruction, with a smaller latency than the corresponding operations on the main memory would require. Then, the contents of the main memory can be updated accordingly without any noticeable latency, due to processor pipelining. The transactions of non adjacent main memory addresses result to a higher cache miss rate, *i.e.* fewer cache hits per transaction and thus, result in a latency equal to the cache miss penalty per transaction that misses the cache. A common example of such transactions is strided array traversal.

The (FD)FDTD software is proven in this report to be a memory throughput–limited piece of software. The same holds for computationally similar software such as other Maxwell equation solvers that use the FDTD method and CFD software [5]. Therefore the two main problems this project faces is the PCI bottleneck in device-to-host transactions and the latency introduced by the output of the 3D FDTD field arrays of the (FD)FDTD software, from the device memory to the host memory.

2.2.1 Related work

There seems to be no published scientific article or treatise focused on the problem of the output of FDTD field data from the device memory to the host memory, or other computationally similar problems. Often, the acceleration of merely the main computation algorithm (FDTD) is considered using GPGPU, rather than the acceleration of the output stage of such applications, such as in [19] and [20], in which nothing is mentioned about the data output to the host memory, and [21], in which the output of data to the host memory is mentioned without details. However, publications [19] and [21] agree with [9], [12] and [13] on the importance of memory coalescing for the minimisation of the latency induced by instructions that perform memory access.

2.3 The (FD)FDTD software

2.3.1 The (FD)FDTD agorithm

The (FD)FDTD software given to the author for the purposes of this project, implements the (FD)FDTD method for the simulation of the propagation of EM waves in the 3D

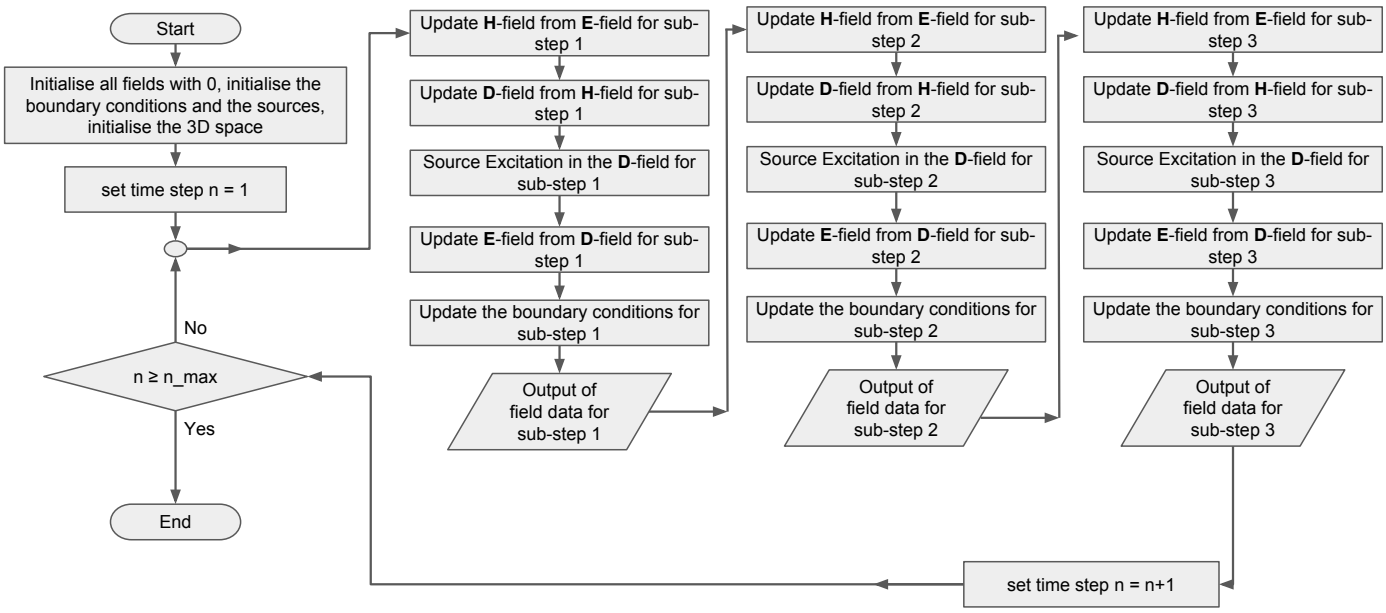


Figure 2.1: Flowchart of the optimised FD-FDTD method implemented by the (FD)FDTD software.

space, and through frequency-depended materials of known electrical properties (Debye parameters). A mathematical description of this method can be found in Section 2.1 of the Progress Report of this project attached in Appendix A of this report.

A flowchart of an abstraction of the (FD)FDTD method is illustrated in Figure 2.1 Specifically, the (FD)FDTD software is optimised by dividing each discrete FDTD time step in three discrete time sub-steps, as described mathematically in Appendix B of the Progress Report (attached in Appendix A of this report). This optimisation provides the compiler the ability to pipeline the processing of some of the processing stages of the (FD)FDTD algorithm in the final executable file.

In addition, the software implements the $\boldsymbol{E}$, $\boldsymbol{H}$ and $\boldsymbol{D}$ fields by using three 3D arrays for each field component, where it stores the scalar value of each of the x, y and z components of each EM field, for every FDTD time sub-step. Thus, the loop depicted in Figure 2.1 is implemented which operates on the field component arrays shown in Table 2.1. These three sub-step stages of the main loop are called the three "main loop parts" of the (FD)FDTD software.

Therefore, the (FD)FDTD software employs 21 different arrays in total, of size (N_x, N_y, N_z) each, for the description of the values of the x, y and z components at each point in the 3D FDTD space, for each discrete FDTD time-step. The (FD)FDTD software initialises these arrays in both the host and device memory, copies any input data from the host memory onto the device memory and begins the FD-FDTD computation in the GPU

Field component arrays updated by main loop part #1	**Field component arrays updated by main loop part #2**	**Field component arrays updated by main loop part #3**
$\mathbf{E_x^{[1]}}(N_x, N_y, N_z)$ $\mathbf{E_y^{[1]}}(N_x, N_y, N_z)$ $\mathbf{E_z^{[1]}}(N_x, N_y, N_z)$	$\mathbf{E_x^{[3]}}(N_x, N_y, N_z)$ $\mathbf{E_y^{[3]}}(N_x, N_y, N_z)$ $\mathbf{E_z^{[3]}}(N_x, N_y, N_z)$	$\mathbf{E_x^{[2]}}(N_x, N_y, N_z)$ $\mathbf{E_y^{[2]}}(N_x, N_y, N_z)$ $\mathbf{E_z^{[2]}}(N_x, N_y, N_z)$
$\mathbf{H_x}(N_x, N_y, N_z)$ $\mathbf{H_y}(N_x, N_y, N_z)$ $\mathbf{H_z}(N_x, N_y, N_z)$	$\mathbf{H_x}(N_x, N_y, N_z)$ $\mathbf{H_y}(N_x, N_y, N_z)$ $\mathbf{H_z}(N_x, N_y, N_z)$	$\mathbf{H_x}(N_x, N_y, N_z)$ $\mathbf{H_y}(N_x, N_y, N_z)$ $\mathbf{H_z}(N_x, N_y, N_z)$
$\mathbf{D_x^{[1]}}(N_x, N_y, N_z)$ $\mathbf{D_y^{[1]}}(N_x, N_y, N_z)$ $\mathbf{D_z^{[1]}}(N_x, N_y, N_z)$	$\mathbf{D_x^{[3]}}(N_x, N_y, N_z)$ $\mathbf{D_y^{[3]}}(N_x, N_y, N_z)$ $\mathbf{D_z^{[3]}}(N_x, N_y, N_z)$	$\mathbf{D_x^{[2]}}(N_x, N_y, N_z)$ $\mathbf{D_y^{[2]}}(N_x, N_y, N_z)$ $\mathbf{D_z^{[2]}}(N_x, N_y, N_z)$

Table 2.1: Field arrays used in each of the three main loop parts of the (FD)FDTD software.

device. No data transaction between the device memory and the host memory takes place before the output stage of each main loop part. At each output stage, the field component arrays shown in Table 2.1 are copied from the device memory into the host main memory, for the purposes of further processing and/or storage to a file or printing to the standard output; because, if the data of the final results are not copied into the host main memory, they cannot be stored or printed.

The parallelised field update routines run in linear time with respect to $\max(N_x, N_y, N_z)$ for each time step, thus they have a computational complexity of $\mathrm{O}(n_{max} \times \max(N_x, N_y, N_z))$ each (n_{max} being the number of time steps of the FDTD simulation), but the software needs to output at least 21 arrays of size (N_x, N_y, N_z), thus having memory complexity $\mathrm{O}(N_x \times N_y \times N_z)$ and the corresponding computational complexity $\Theta(n_{max} \times N_x \times N_y \times N_z)$ for the transfer of each field component array, which is proven in Section 3.1 to have a big impact on the total execution time of the (FD)FDTD software, compared to the time needed for the execution of the routines for the calculation of the field values at any time step. Hence, the (FD)FDTD software is memory-throughput limited, and thus the need for the optimisation of the performance of its output stage emerges.

2.3.2 Software components

The (FD)FDTD software receives input from file "`input_params`", located in the same directory. File "`input_params`" is a text file that has the structure "description: value" per line. This file contains the desired duration of the (FD)FDTD simulation (number

of time steps), the number of EM oscillation sources and their locations, their frequency, the size of the simulation space and other information.

From Figure 2.1 it becomes transparent that the (FD)FDTD software can be perceived as consisting of three main components or stages:

1. `src`: The component which consists of the sub-routines of the main loop that perform source excitation in $\boldsymbol{D}$ field,

2. `fdtd`: The component which consists of the sub-routines of the main loop that perform field updates (excluding any source excitation routines),

3. `output`: The component which consists of the sub-routines of the main loop that perform data output operations).

Thus, for the purposes of this project, time-measuring source code was developed and introduced in the source code of the (FD)FDTD software. The time-measuring source code employs the Fortran function `system_clock` for the measurement of the total execution time (wall-time) [22] of each main software component for each FDTD time sub-step. Attention was also given on handling possible timer wrap-arounds (*i.e.* the overflow of integer variables holding timing information). Hence, output similar to that in Table 2.2 is produced in the standard output stream, at the end of the execution of the (FD)FDTD software:

Timings by loukas (wall time)			
prog	42279568	or	42.27957 s
src	13288758	or	13.28876 s
fdtd	9320457	or	9.320457 s
output	19668385	or	19.66838 s

Table 2.2: Sample timing information output produced by the modified (FD)FDTD software.

The first column of this information indicates what part of the software the timing measurements in the rest of the line correspond to. The string "`prog`" means that information for the elapsed wall-time for the execution of the whole software, excluding the time needed for the allocation and deallocation of arrays, follows in the same line. The string "`src`" means that information for the accumulated elapsed wall-time for the execution of the `src` component of the software follows in the same line. The string "`fdtd`" means that information for the accumulated elapsed wall-time for the execution of the `fdtd` stage of the software follows in the same line, and the string "`output`" means that information for the accumulated elapsed wall-time for the execution of the `output` stage follows in the same line.

The second column of this timing information output is a positive number indicating the accumulated elapsed CPU clock ticks for the stage stated in the first column. The fourth column of this timing information output is the overall elapsed time in seconds, for

the software component mentioned in the first column. Hence, the example in Table 2.2 indicates that the corresponding instant of the (FD)FDTD software needed 42.28 seconds to run - excluding the time needed for the allocation or the deallocation of its arrays, the source update routines needed a total of 13.29 seconds to run, the (FD)FDTD update routines took 9.32 seconds to run, whereas 19.67 seconds elapsed for the output of the fields arrays from the device memory to the host memory, for all FDTD time steps.

For the purposes of this project, only the `output` stage of the (FD)FDTD software is further modified. However, each execution attempt of a specific instant of the (FD)FDTD software results in different timing information being outputted, due to random factors such as the system load at the time of software execution. Nevertheless, it can be safely assumed that the relative time difference of the `output` time (and the elapsed time of any other software stage) to the `prog` time due random factors is negligible. Hence, a useful indicator of how the elapsed total time of the `output` stage varies due to software modifications, can be the ratio of the `output` time to the total elapsed `prog` time, which excludes the time needed for main memory and device memory allocation and deallocation (as the time for memory allocation/deallocation can be affected by random factors too). Thus, this report uses the relative output time, *i.e.* the ratio ξ described by Eq. 2.1 for the assessment of the performance of each ad-hoc algorithm.

$$\text{Relative output time } \xi = \frac{\text{“output” time in seconds}}{\text{“prog” time in seconds}} \tag{2.1}$$

For example, the (FD)FDTD execution instant corresponding to Table 2.2 has a relative output time $\xi = \frac{19.66838}{42.27957} \approx 0.465198203$.

2.4 The computational environment of this project

2.4.1 RIKEN GreatWAVE computer

The supercomputer system HOKUSAI-GreatWAVE in RIKEN, Saitama, Japan was used for the development, debugging and execution of the source codes of this project. RIKEN is Japan's largest pioneering research institution for basic and applied science [23]. In particular, the “Application Computing Server with GPU” (ACSG) [24, pp.2-3] was used for the purposes of this project. Table 2.3 ([24, pp. 2, 3, 5, 6]) contains the specification of the ACSG cluster. The specific configuration of the ACSG cluster used for the execution of the (FD)FDTD software is listed in Table 2.4.

The RIKEN-GreatWAVE system and its subsystems, including the ACSG cluster, are shared between many users. However, the users do not have direct access to the supercomputer clusters. The RIKEN GreatWAVE system employs a back end – front

Nodes	30 nodes of SGI C2108-GP5
CPU	Intel Xeon E5-2670 v3 (2.30GHZ) x 30 units (60CPUs, 720 Cores)
Theoretical peak performance	26.4 TFLOPS (2.3 GHz x 16 floating-point operations x 12 cores x 60 CPUs)
Theoretical peak performance per node	883.2 GFLOPS
Memory capacity	1.8 TB (64GB x 30 units)
Memory bandwidth	68.2 GB/s/CPU and 0.15 Byte/FLOP
Accelerators installed per node	(4 devices/node) x NVIDIA Tesla K20X
Node interconnect interface	InfiniBand FDR
Node link bandwidth	6.8 GB/s x 2 (bidirectional)
Local disk capacity	18 TB ((300GB x 2) x 30 units)
Operating System	Red Hat Enterprise Linux 6 (kernel version 2.6) 64-bit

Table 2.3: Specification of the ACSG cluster.

Nodes	1
Number of CPU cores	8
Number of GPUs	1
Maximum elapsed time	2 hours

Table 2.4: Configuration of the ACSG cluster used for the execution of the (FD)FDTD software.

Operating System	Red Hat Enterprise Linux 6 (Linux kernel version 2.6) 64-bit
CUDA Fortran Compiler	The Portland Group Fortran 90/95 compiler (PGI F90 compiler) v. 15.5-0 64-bit target on x86-64 Linux -tp sandybridge
Native C++ compiler for ACSG	Intel(R) C++ Compiler [ICC] v.16.0.2
Assembler & Binary utilities	GNU Assembler version 2.20.51.0.2-5.36.el6 20100205 configured for target x86_64-redhat-linux and GNU binutils v2.20.51

Table 2.5: Specification of the RIKEN-GreatWAVE front end.

end structure; it provides four login nodes, called the "front end" [24], whereas the supercomputer clusters such as ACSG form the "back end". Users may access the front end servers via a Secure-Shell (SSH) connection for the purposes of software development, tuning, compilation and linking [24]. Users may also use the front end servers for submitting executable software (called "jobs") on to a queue for execution on the back end. Specifically, for each job the users desire to submit for execution on the back end, they have to submit a job shell script, which contains information about the location of the executable which they want to have executed, the cluster they want the job to be executed on, the maximum number of nodes and processes (threads) they want their software to occupy, and the maximum elapsed time for the job execution. If a job exceeds the specified elapsed time, it is "killed" and the user is notified by an e-mail message. If the executable programme of the job produces any output to the standard output or the standard error, this output is stored in a text file in the same directory. The amount of waiting time for a job to be executed on a specific back end varies depending on the total number of jobs submitted to the specific supercomputer, their maximum elapsed time and other factors. Thus, the amount of waiting time for a job to be submitted to the back end can vary from zero seconds (job is processed by the back end immediately) to five days or more.

The software development environment provided in the front end is shown in Table 2.5 ([24]). The provided PGI F90 compiler `pgf90` had to be used for the compilation of CUDA Fortran applications, as currently it is the only CUDA Fortran compiler. The compilation of all CUDA Fortran applications of this project was done using

the `pgf90` compiler arguments `ta=tesla:cc35 -tp=haswell-64 -Mpreprocess -Minfo -Mcuda=kepler+ -fast`. As the RIKEN-GreatWAVE system was built just a few months before the beginning of this project and the GPU development hardware and software equipment was ready just weeks before the project started, the development environment did not work as provided without problems. For example, the compilation of CUDA Fortran source codes first resulted in error messages, which originated due to the GNU Assembler (and thus the `gnu-binutils` package) being of an older version than the compiler required. For this reason, the administrators of RIKEN-GreatWAVE were contacted via e-mail. They suggested the manual installation of a newer version of the package `gnu-binutils` package inside the home directory of our user (the author and postgraduate students in Dr. Fumie Costen's research group share access to the RIKEN-GreatWAVE supercomputer by using the same username to connect to the front end). Hence, the version 2.25.1 of the GNU Binutils package (which includes the GNU Assembler of the same version) was installed in the corresponding home directory on the front end by the author of this report. Also, another problem that arose during the progression of this report was that the version 15.5-0 of the PGI F90 compiler is lagging behind the CUDA Fortran specification. For example, while a `device=device` variable assignment is valid syntax in CUDA Fortran [13, p.30], the PGI F90 compiler stops compilation and outputs an error message when encounters such an assignment statement outside of a CUDA kernel. For such data transfers that take place outside of a CUDA kernel, the corresponding explicit function call method has to be used instead.

2.4.2 Shell script development for batch job execution and final results production

As the RIKEN-GreatWAVE system is shared amongst its users, the the overall execution time (wall-time) of running applications can be affected by the system load. Therefore, for the purposes of this project, each performance test was run three times and the resulting timing counts were averaged, to eliminate the effects of random system load bursts to the performance tests while they were executed. These averaged results are the final results used throughout this report.

For the facilitation of the production process of such results, a number of `bash` shell scripts and `awk` scripts were developed. Namely, script `rawdata2relative.awk` was developed in the `awk` language; this script calculates the relative `output` time ξ from the standard output of one performance test or from the aggregated standard output of more performance tests. Script `gettestcasenames.sed` was coded in the `sed` stream manipulation language; script `gettestcasenames.sed` extracts the name of a performance test out of its filename. The `awk` script "`getdataoutput.awk`" extracts and outputs

one line per performance test (after the output of these tests is processed by script `gettestcasenames.sed`), which contains the name of the performance test, and the corresponding relative `output` time ξ, separated by a single space. Script `grabdata.sh` automatically aggregates in one file the standard output produced by all performance tests inside the current working directory, has the relative `output` time ξ of each performance test calculated by calling the `awk` script "`rawdata2relative.awk`" from the corresponding wall-time information (in seconds), and employs scripts `gettestcasenames.sed` and `getdataoutput.awk` to automatically produce results ready to-be-plotted for every performed performance test.

There have been occasions during the project's progression, where the submission of more than 100 jobs for execution on the back end was required (such as the performance tests of the multiple lines case – Section 3.5, and of the several planes case – Section 4.4). However, cluster ACSG has a limitation on the number of concurrent submitted jobs; only 100 jobs are allowed to be queued at the same time [24, p. 66]. For this reason a `bash` shell script, named "`run_diagonal.sh`", was developed allowing the mass submission of many jobs to the ACSG back end. Script '`run_diagonal.sh` finds all performance tests that reside within a common directory and have the same suffix in their filename and pushes them into a stack structure. Then, while the stack contains at least one element, it checks the number of jobs that are queued for execution by filtering the output of the `pjstat` command; While this number is greater than 80 jobs, the script suspends its execution for 10 minutes, and then the check of the number of queued jobs is repeated. If less than 80 jobs are queued up for execution on the back end, the script pops one element out of the stack and submits it for execution on the back end using the `pjsub` command. The contents of the stack at any time and the output of the `pjsub` command are saved in two files, "`TMPJOBSLEFT`" and "`jobsubd.log`" respectively, to make the processes of logging and job deletion (in case of accidental execution) easier. For the case of accidental mass job submission, the `bash` script "`deletejobs.sh`" was developed, which requests the immediate deletion of the submitted jobs listed in file "`jobsubd.log`" in the current working directory.

Chapter 3

The transfer of single lines of the (FD)FDTD space between the host and device memory

3.1 Device-to-host transfer of the whole 3D space: The NB33 method

The first step of practical progress of this project was the measurement of the time needed for the transfer of the field data for the whole 3D simulation space (or, for a non–buffered '3D array to 3D array' transfer, which is named as 'NB33' transfer, according to the naming convention used in this report), for each (FD)FDTD time step and for each of the $\boldsymbol{E}$, $\boldsymbol{H}$ and $\boldsymbol{D}$ fields, via a direct `host=device` assignment. This is the trivial solution to the data transfer problem in the case of simulating the propagation of EM waves within a cubic 3D space using the (FD)FDTD software. However, this execution time percentage ξ_{NB33} forms the basic term of comparison for assessing the performance of other data transfer methods/algorithms, in case the user of the (FD)FDTD software is interested in a proper subset of the points of the 3D simulation space.

As highlighted in Section 2.3.1, the (FD)FDTD software employs 21 arrays to describe the cartesian components of each of the $\boldsymbol{E}$, $\boldsymbol{H}$ and $\boldsymbol{D}$ fields, at each discrete point in the 3D space, at the current FDTD time step. Each of these arrays is a 3D array of size equal to the (FD)FDTD simulation space, which throughout this project is kept constant and equal to $(N_x, N_y, N_z) = (374, 374, 374)$, as with this size a big amount of global memory is occupied ($\approx$ 5340 MB) and thus there are 699MB free for the introduction of any auxiliary arrays. Also, the number of FDTD time steps used throughout this project was set to 102, to be large enough to allow the assessment of a general behaviour of a specific data transfer method, and small enough for the (FD)FDTD simulation not to

require a large amount execution time.

3.2 NB13: Non-buffered single line to 3D

Sections 3.2, 3.3 and 3.4 examine the case where the Points Of Interest (POI) [1] are located on one or more single lines of the (FD)FDTD space which are parallel to one of the edges of the (FD)FDTD cube (and thus, parallel to the x, y or z axis).

The trivial solution to the single-line POI problem is to pass the data from the accelerator's ("device") memory to the system's main memory with a `host=device` assignment statement. The implementation of this method is straightforward and effortless.

3.2.1 NB13X

The transfer of field data at points in the FDTD space which are parallel to the x axis is considered. These are all the discrete points that are located at coordinates / array indices $(x, y, z) = (i, c_y, c_z), 1 \leq i \leq 374$, where c_y, c_z = constant between 1 and 374 inclusive. The resulting execution time was found to remain invariable with changes in c_y, c_z.

On code level, each data transfer is described as a `host=device` assignment for the transfer of the span of the x dimension of a field array, for each of the $\boldsymbol{E}$, $\boldsymbol{H}$ and $\boldsymbol{D}$ fields (nine 3D arrays in total). Thus, $N_x \times 9 = 374 \times 9 = 3366$ array cells are transferred from the device to host, or 3366×4 bytes = 13464 bytes, as each field component array is of type `real*4`. A pseudocode description of method NB13X can be found in Appendix B.

This method of data output is the trivial solution for the case where the POI lie on a single line which is parallel to the x axis. Thus, this method would be preferred over the non-buffered transfer of the whole FDTD space (the "NB33" method), as it performs a memory transaction with less data (a single line of the 3D space, which forms $\frac{1}{374^2}$ of the data transferred by the "NB33" method), which also happens to be read from consecutive device memory addresses, since FORTRAN is a column-major language [25], thus it is a coalesced memory transaction. Hence, in theory this data transfer method ("NB13X") outperforms the non-buffered transfer of the whole 3D (FD)FDTD space ("NB33").

3.2.2 NB13Y

A modification of the above method ("NB13X") concerns the case where the POI are located on a single line parallel to the y axis. These are all the points which are located

[1] Points Of Interest (POI): The points which are desired to be transferred into the host's main memory for the purpose of further data processing and/or storage

at coordinates / array indices $(x, y, z) = (c_x, j, c_z), 1 \leq j \leq 374$, where c_x, c_z = constant between 1 and 374 inclusive.

On code level, each data transfer is described as a `host=device` assignment for the span of a y dimension of each of the $\boldsymbol{E}$, $\boldsymbol{H}$ and $\boldsymbol{D}$ fields (thus, from nine 3D arrays in total). This method of data output is the trivial solution for the case where the POI are lying on a single line which is parallel to the y axis. A pseudocode description of method NB13Y can be found in Appendix B.

Although this transfer method is very similar to the one above ("NB13X"), it is expected to perform worse. This method requires strided memory access, with a stride of $N_x = 374$ array cells, due to FORTRAN being a column-major language. As each of the arrays representing the electric or magnetic fields is of type `real*4`, the stride length is $N_x \times (4 \text{ bytes}) = 374 \times (4 \text{ bytes}) = 1496 \text{ bytes}$. Nevertheless, the amount of data transferred from the device to host is still $N_y = 374$ array cells, or 374×4 bytes $= 1496$ bytes (for each of the $\boldsymbol{E}$, $\boldsymbol{H}$ and $\boldsymbol{D}$ fields), which is significantly less than transferring the whole 3D FDTD space ($[N_x \times N_y \times N_z \times 4 = 374^3 \times 4]$ bytes = 209.25 MB per field component array). Therefore, this difference in the amount of data to be transferred could result in faster execution than the "NB33" method, when the POI are located on just one line oriented in the y direction. Nevertheless, there are no officially published thorough performance characteristics related to the stride of global memory access by NVIDIA. Hence, a slowdown relative to NB13X could happen with the NB13Y method instead of a speedup. The performance tests of the implementation of this method shall reveal whether or not this method leads to faster execution than method NB33.

3.2.3 NB13Z

This method is the trivial solution of the problem where the POI are located on a single line that is parallel to the z axis. Again, this method is similar to "NB13X" and "NB13Y", but it has a drawback, which is the length of the stride for the z direction. Since FORTRAN is a column-major language, the access of an array in the z direction has a stride length of $N_x \times N_y = 374 \times 374 = 139,876$ array cells. However, the amount of data on a single line in the z direction is $N_z = 374$ array cells, or 1496 bytes (as indicated in the above sections), therefore a faster execution than the "NB33" method could be expected, in the case where the POI are located on a single line oriented in the z direction. Again, as there is no officially published treatise of the device memory access characteristics, a slower execution than method NB33 can also be expected. A pseudocode description of method NB13Z can be found in Appendix B.

3.3 B13: Buffered single line to 3D, buffer in device memory

Methods "NB13Y" and "NB13Z", presented above, access the field arrays in the host system in a strided manner. Because of the stride, device-to-host transfers are not coalesced, thus a decrease in data transfer throughput is expected. Also, strided access increases the host processor's cache miss ratio, resulting in larger execution time of every device-to-host transfer operation for which a cache miss occurs. At the same time, the (FD)FDTD software cannot make any progress with updating the $\boldsymbol{H}$ field again, since the operations of the corresponding CUDA kernel have to wait for the data transfer happening in the default stream to finish its execution[13, ch. 2.10.2, p. 11]. Assigning a different CUDA stream number to either the field update CUDA kernels or the data transfer operations is not a solution, since the field update routines could replace the field data intended to be transferred, before the desired data move takes place. A potential optimisation could be the introduction of line buffers between the device memory and the host memory, as a device-to-device transfer is expected to be completed faster than a device-to-host transfer [26]. There seems to be no quantitative information on how much faster a device-to-device transfer is, as the throughput achieved in any case is application and implementation specific.

The "B13" transfer methods aim to exploit this possible throughput difference, introducing a 1D array in the device memory, eliminating the strided device memory access and its effects during device-to-host transfers, and increasing the coalescence of the data transfer operation between the device and the host memory. Then, a buffering-perfoming kernel is spawn during each transfer, with a grid that contains one block. The block consists of 512 threads. This number of threads per block was chosen empirically, as kernels with 512 threads per block performed better for the following tasks (B13X, B13Y and B13Z) than other multiples of 32 (*i.e.* multiples of the warp size).

The kernel code checks whether or not the id of the current thread is between 1 and $N_x = N_y = N_z = 374$ inclusive. If it is not, the corresponding thread terminates, thus avoiding any out-of-bounds array access. Threads that have an id number between 1 and 374 inclusive copy a value from a cell of a 3D field array which resides in the device memory, into a cell of a 1D array (buffer) also in the device memory. When all the threads of the kernel terminate, the 1D buffer array contains the values of each POI in a line of a 3D field array. A pseudocode description of methods B13X, B13Y and B13Z can be found in Appendix B.

3.3.1 B13X

The B13X method implements a 1D buffer residing in device memory. A kernel is launched before each data transfer, which copies a line parallel to the x axis of the (FD)FDTD space to the 1D buffer, as described in Section 3.3. When the kernel finishes its execution, a `host=device` statement is used to copy the 1D buffer from the device global memory into a field array in the main memory of the host. This procedure is followed for each of the nine field component arrays.

Although a performance improvement of the NB13X method is not supported by the theory using the B13X method, the B13X method is implemented and tested, as there is no proof it may perform worse than the NB33 method, given the high GPU device complexity and its possible undocumented performance characteristics.

3.3.2 B13Y, B13Z

Methods B13Y and B13Z follow the methodology described in 3.3, which is also followed by the B13X method. The B13Y method stores a line parallel to the y axis of the (FD)FDTD space into the 1D buffer, whereas the B13Z method stores a line parallel to the z axis of the (FD)FDTD space into the 1D buffer. Then, both methods copy the buffered line into the 3D field array. A performance improvement is expected with these methods compared to the NB13Y and NB13Z methods respectively, as CUDA FORTRAN is a column-major language and the access of a line in the y or z direction results in strided array accesses, which is eliminated from accelerator's side using the B13Y and B13Z methods.

3.4 B11: Buffered single line to 3D, buffers in both host memory and device memory

Another optimisation of the NB13Y and NB13Z methods, could be the introduction of a buffer in the host memory. Specifically, a 1D array can be placed into the host main memory, eliminating the stride of access of the `host=device` assignment operations. Pseudocode descriptions of the B11 methods (B11X, B11Y and B11Z) can be found in Appendix B.

3.4.1 B11X

The B11X method copies the field component data on a single line of the (FD)FDTD space parallel to its x axis into a 1D buffer in the device memory. he contents of the 1D buffer in the device memory are copied into a 1D buffer in the host main memory. The contents

of the 1D buffer in the host memory can then be further processed and/or stored on user's demand.

Like with the B13X method, a performance improvement of the NB13X method is not supported by the theory using the B11X method. Nevertheless the B11X method is implemented and tested, as there is no proof it will perform worse, given the high GPU and CPU complexity and their possible undocumented performance characteristics, combined with the the complexity of possible compiler optimisations.

3.4.2 B11Y and B11Z

Methods B11Y and B11Z perform the method described in 3.4. Method B11Y stores a line of the (FD)FDTD space, parallel to its y axis into the 1D buffers, whereas the B11Z method stores a line of the (FD)FDTD space that is parallel to its z axis, into the 1D buffers. A performance improvement is expected with these methods compared to methods B13Y and B13Z respectively, as well as NB13Y and NB13Z, as CUDA FORTRAN is a column-major language and the access of a line in the y or z direction results in strided array accesses, which is eliminated from the side of the host, using methods B11Y and B11Z.

3.5 Multiple line case

Section 3.5 concerns the case where the POI lie on multiple lines of the (FD)FDTD space. This case involves multiple lines parallel to just the x, just the y or just the z axis. The investigation of how many single such lines can be transferred from the device memory to the host memory in less time than it takes for the field data of the whole 3D FDTD space to be copied from the device to the host (the NB33 method), would constitute a useful piece of information for the further development of the (FD)FDTD software and computationally similar projects. Methods NB13, B13 and B11 will be used in tests for this purpose.

For the assessment of the performance of these algorithms on the multiple line case, the location of the lines used in the test cases must be taken into consideration. If consecutive lines with respect to one coordinate are selected for a test, *e.g.* all the lines parallel to the x axis, at $y = i, 1 \leq i \leq 374$, then such tests would be pointless, since the lines could be aggregated using one of the plane-transfer algorithms described in Chapter 4, therefore the transfer could be coalesced and be executed more efficiently. Hence, the selection of lines shown in Table 3.1 was made, with $N_b, 1 \leq N_b \leq 374$ being the number of lines to-be-transferred from the device memory to the host main memory. This selection of coordinates aims to eliminate the appearance of multiple lines on a same

plane of the 3D simulation space and at the same time to access the 3D arrays of the field components with big strides in both y and z, or x and z, either x and y, directions (strides of length $\lfloor 374/N_b \rfloor$ array elements).

Line buffer methods	**Coordinates chosen**
NB13X, B13X, B11X	$y = \lfloor i \times N_y/N_b \rfloor = \lfloor i \times 374/N_b \rfloor$ $z = \lfloor i \times N_z/N_b \rfloor = \lfloor i \times 374/N_b \rfloor = y$ $i \in \mathbb{N}, 1 \leq i \leq N_b$
NB13Y, B13Y, B11Y	$x = \lfloor i \times N_x/N_b \rfloor = \lfloor i \times 374/N_b \rfloor$ $z = \lfloor i \times N_z/N_b \rfloor = \lfloor i \times 374/N_b \rfloor = x$ $i \in \mathbb{N}, 1 \leq i \leq N_b$
NB13Z, B13Z, B11Z	$x = \lfloor i \times N_x/N_b \rfloor = \lfloor i \times 374/N_b \rfloor$ $y = \lfloor i \times N_y/N_b \rfloor = \lfloor i \times 374/N_b \rfloor = x$ $i \in \mathbb{N}, 1 \leq i \leq N_b$

Table 3.1: Selection of coordinates for the multiple line buffer tests.

So far, three possible methods were considered (NB13, B13, B11), for three line cases (the line with the POI being parallel to the x, y and z axis), and the maximum number of tests per line direction is $N_x = N_y = N_z = 374$. Hence, a need emerges for 3 methods $\times$ 3 directions$\times 374 = 3,366$ tests to be produced. Consequently, the process of the test case development and execution had to become automated, to meet the projects' deadlines. For this purpose, the workaround of utilising the (FD)FDTD software's main input text file "`input_params`" was decided, as it was a solution which was quick to implement and it would not make the (FD)FDTD software require more input files to run, thus keeping the inode count of the RIKEN GreatWAVE front-end "user" as low as possible. More specifically, a line is inserted after the existing second line of file "`input_params`", with the string "`nbuff: `" followed by the number of lines to be transferred on the current test-case. Next, a `bash` shell script was developed, which produces all 3,366 tests mentioned above, placing each one in an appropriately named subdirectory that starts with the prefix "`DIAG-`"; for example a sub-directory with name "`DIAG-B11Z-32`" indicates that it contains a B11Z test with 32 lines parallel to the z axis of the FDTD space. Finally, the batch-run and results-grabbing shell script described in Section 2.4.2 were employed to run all 3,366 tests and to retrieve the results in a single file. In addition, a script was produced in the Ruby programming language, which compares the relative `output` time of each test with the `output` time of the NB33 method, and decides, for each line-transfer method, the maximum number of lines which guarantees faster execution than the NB33 method.

3.6 Transfer of single lines – Results & Discussion

3.6.1 NB33: Device-to-host transfer of the whole 3D FDTD space

The relative '`output`' time for the NB33 transfer method was found to be 0.754213. This means that transferring the data for the whole 3D space (which contains 374^3 discrete points) from the global memory of the NVIDIA accelerator to the main memory of the host computer system, for each of the 102 time steps and for each of the $\boldsymbol{E}$, $\boldsymbol{H}$ and $\boldsymbol{D}$ fields, uses the 75.4213% of the total execution time of the (FD)FDTD simulation. This verifies the claim of Dr. Fumie Costen's research group, that more than half of the execution time of their vectorised and parallelised FDTD software is spent on the output of data.

Here it is noteworthy that if the number of time steps is increased, the overall simulation time also increases, but the '`output`' time percentage remains virtually constant, as about 0.75% of the total run time is spent on the '`output`' stage. However, if the size of each edge of the cubic (FD)FDTD space increases, this percentage increases, as more data need to be read from the device memory and be transferred through the PCI-E bus, thus a bigger time delay is introduced at the '`output`' stage. Nevertheless, the number of 102 time steps was chosen for this project, for the reasons highlighted in Section 3.1.

3.6.2 Device-to-host transfers of one line

The resulting average relative `output` times $\bar{\xi}$ of each of the methods NB13X, NB13Y, NB13Z, B13X, B13Y, B13Z, B11X, B11Y, B11Z are shown in Table 3.2 and Figure 3.1.

Method NB13X executes faster than method NB33, with a speedup of $\frac{0.754213}{0.672170} = 1.12$ gained. This happens because method NB13X makes a single coalesced device-to-host transfer of less data than NB33, as highlighted in subsection 3.2.1. Therefore, in theory there is no delay factor acting upon the device-to-host transfer, as no misaligned access happens, and since the data to-be-transferred is of smaller aggregated size in comparison to method NB33, the NB13X transfer happens faster than NB33. The relative `output` time of method NB13Y approaches the relative `output` time of method NB33. It is marginally faster than NB33, but since a stride of 374 array elements is introduced, which corresponds to $1,496$ bytes (as explained in subsection 3.2.2), the memory transactions happen with low coalescence. This leads to a low device multiprocessor occupancy, which results to low effective bandwidth. This effect is more intense in method NB13Z, where the stride of memory access is 139,876 array cells or 559,504 bytes (as explained in subsection 3.2.3), making method NB13Z slower than method NB33 by 0.02%.

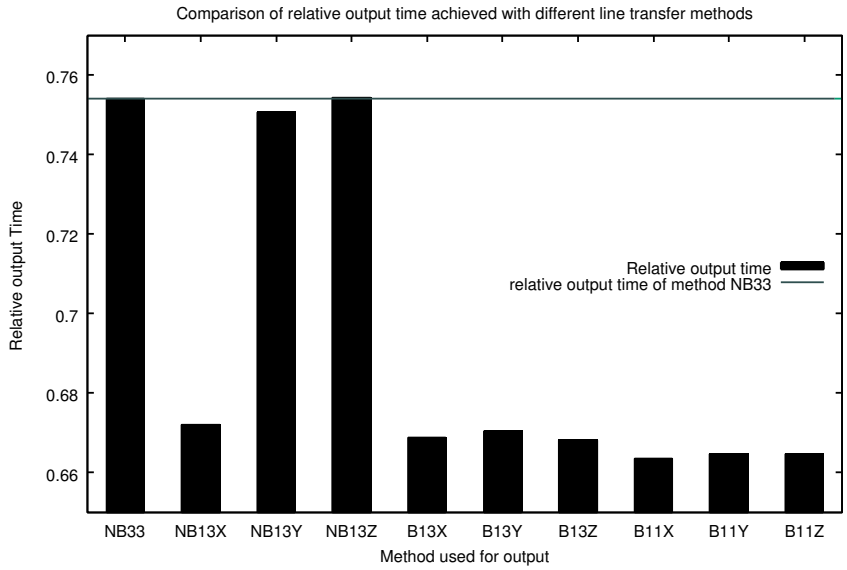


Figure 3.1: Comparison of different line buffer methods, for the transfer of a single line.

Line transfer method	**Relative `output` time**
NB33	0.754213
NB13X	0.672170
NB13Y	0.750759
NB13Z	0.754362
B13X	0.668951
B13Y	0.670431
B13Z	0.668215
B11X	0.663475
B11Y	0.664764
B11Z	0.664784

Table 3.2: Relative "output" timings for the single-line output methods described in chapter 3.

3.6.3 Device-to-host transfers of more than one lines

The process which is described in Section 3.5 was followed. The results of the `Ruby` script are shown in Table 3.3. All methods that concern lines parallel to the x axis, *i.e.* NB13X, B13X, B11X, executed faster than a NB33 transfer. This is due to the fact that iterating over the x dimension of the field arrays does not lead to strided array access in the Fortran programming language (as it is column-major) and thus no delay factors due to non-coalesced device memory access or CPU cache misses are introduced. However, Figure 3.2 reveals that no statement can be made about which of these methods is the fastest for the transfer of multiple non-adjacent lines which are parallel to the x axis of the FDTD space, as the comparative performance of each method depends on the number of non-adjacent lines to-be-transferred from the device memory to the host memory.

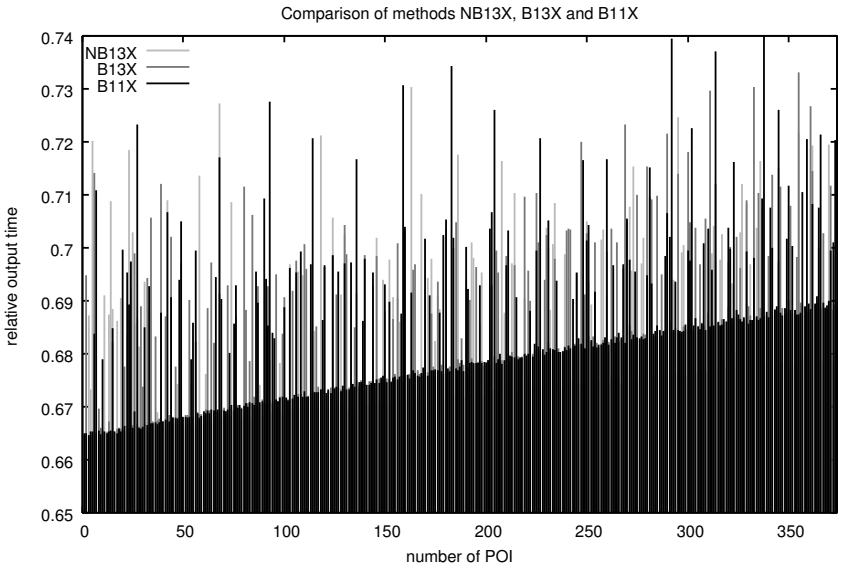


Figure 3.2: Comparison of NB13X, B13X and B11X methods, for the transfer of multiple lines.

Amongst the methods that concern lines which are parallel to the y or the z axis of the FDTD space, by using the double-buffered methods, *i.e.* methods B11Y and B11Z, faster execution than the NB33 method, for 374 non-adjacent lines or fewer, is guaranteed. By the accumulation of the data to-be-transferred in the device-side buffer, the effects of strided device memory access on the data transfer bandwidth are eliminated. Also, by using a host-side line buffer, the effects of strided RAM access are eliminated leading to an increased data transfer bandwidth at the host side.

On the other hand, the faster execution of method NB13Y in comparison to NB33, for more than one line, is not guaranteed. With methods NB13Y and NB13Z, the latency caused by misaligned global memory and RAM access exceeds the latency due to the big volume of data to-be-transferred in the NB33 case. The rate of transfer of lines which are

parallel to the z axis of the FDTD space is improved if a device-side buffer is used, as up to 142 non-adjacent lines can be transferred using the B13Z method, quicker than using the NB33 method. Table 3.3 reveals that method B11Z is the fastest for transferring lines which are parallel to the z axis of the FDTD space, in comparison with methods NB13Z and B13Z, as with method B11Z the faster execution of the output stage than method NB33 is guaranteed for up to 374 non-adjacent lines. For the transfer of lines which are parallel to the y axis, methods B13Y and B11Y guarantee faster execution than method NB33 for 374 non-adjacent lines or fewer. Figure 3.3 reveals that the introduction of the host-side buffer (method B11Y) does not always perform faster memory transactions than method B13Y, as the relative performance of these two methods is depended on the number of POI.

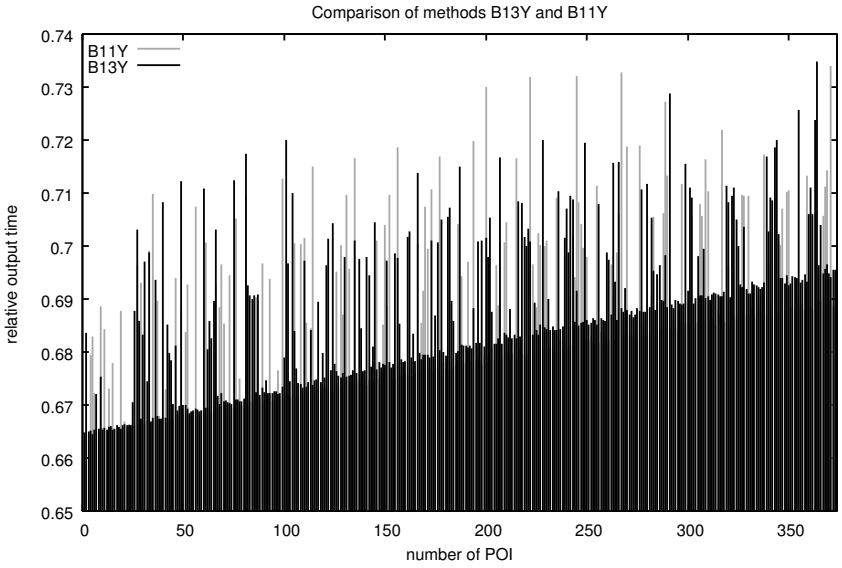


Figure 3.3: Comparison of B13Y and B11Y methods, for the transfer of multiple lines.

Line transfer method	**Number of lines which guarantees faster performance than NB33**
NB13X	374
NB13Y	1
NB13Z	0
B13X	374
B13Y	374
B13Z	142
B11X	374
B11Y	374
B11Z	374

Table 3.3: Relative "output" timings for the single-line output methods described in chapter 3.

Chapter 4

Approach for Plane transfer

Nine methods were considered for transferring a plane of the 3D FDTD space from the accelerator's global memory into the main memory of the host. The approach to the plane-transfer problem is similar to that followed for the transfer of lines in the FDTD space, and thus, the naming convention used for naming the plane transfer methods is congruous with the one used for naming the line transfer methods. The problem of the plane transfer is solved using non-buffered transfer methods (NB23), single-buffered transfer methods (B23, with a device-side buffer) and double-buffered transfer methods (B22, with a device-side and a host-side buffer). These methodologies are applied for planes parallel to the x-y plane, parallel to the y-z plane and parallel to the x-z plane. A pseudocode description of all methods described in Chapter 4 can be found in Appendix B.

4.1 NB23: Non-buffered plane transfer

4.1.1 NB23Z

Method NB23Z concerns the device-to-host transfer of a single plane parallel to the x-y plane of the FDTD space. On code level, these planes, which are named as "X-planes" according to the naming convention used in this report, are defined by the coordinates / array indices $(x, y, z) = (i, j, c)$ with $1 \leq i, j \leq N_y = N_z = 374, \quad i, j \in \mathbb{N}$ and $c = \text{const}$ with $c \in \mathbb{N}, \quad 1 \leq c \leq N_z = 374$. Method NB23Z makes a non-buffered X-plane transfer from the device memory to the host memory. The data for the $N_x \times N_y = 374 \times 374 = 139,876$ points transferred with this method are located in adjacent memory addresses (because Fortran is column-major). Therefore, a high hit ratio is expected from the CPU cache, and the device-to-host data transfers performed by this method are coalesced. Hence, this method is expected to perform faster than the NB33 method.

4.1.2 NB23Y

Method NB23Y concerns the device-to-host transfer of a single plane parallel to the x-z plane of the FDTD space. On code level, these planes, which are named as "Y-planes" according to the naming convention used in this report, are defined by the coordinates / array indices $(x, y, z) = (i, c, j)$ with $1 \leq i, j \leq N_x = N_z = 374, \quad i, j \in \mathbb{N}$ and $c = \text{const}$ with $c \in \mathbb{N}, \quad 1 \leq c \leq N_y = 374$. Method NB23Y makes a non-buffered Y-plane transfer of the data of $N_x \times N_z = 374 \times 374 = 139,876$ points from the device memory to the host memory. As Fortran is a column-major language, this method performs, for each field array, $N_x = 374$ coalesced transfers, followed by a strided memory read and write, which has a stride length of $N_x \times (N_y - 1) = 374 \times (374 - 1) = 139,502$ array elements, which is ensued from $N_x = 374$ coalesced transfers. This pattern is followed foreach field array (for the 21 field arrays described in Section 2.3.1), and $N_z = 374$ times in total, *i.e.* until the transfer of the $N_x \times N_z = 374 \times 374 = 139,876$ array elements is completed. Hence, it is not clear whether or not this method will perform worse than method NB33; performance tests shall reveal which method runs faster.

4.1.3 NB23X

Method NB23X concerns the device-to-host transfer of a single plane parallel to the y-z plane of the FDTD space. On code level, these planes, which are named as "X-planes" according to the naming convention used in this report, are defined by the coordinates / array indices $(x, y, z) = (c, i, j)$ with $1 \leq i, j \leq N_y = N_z = 374, \quad i, j \in \mathbb{N}$ and $c = \text{const}$ with $c \in \mathbb{N}, \quad 1 \leq c \leq N_x = 374$. Method NB23X makes a non-buffered X-plane transfer of the data of $N_y \times N_z = 374 \times 374 = 139,876$ points from the device memory to the host memory, As Fortran is a column-major language, this method performs, for each 3D field array, $N_y = 374$ data transfers with a stride of $N_x = 374$ array elements, followed by a strided memory read and write, which has a stride length of $N_x \times (N_y - 1) = 374 \times (374 - 1) = 139,502$ array elements, which is ensued from $N_x = 374$ coalesced transfers. This pattern is followed foreach field array, and $N_z = 374$ times in total, *i.e.* until the transfer of the $N_x \times N_z = 374 \times 374 = 139,876$ array elements is completed. Hence, a poorer performance than method NB33 is expected with this method. Nevertheless, as there is no officially published documentation of the memory access characteristics of the accelerator device, this method was also examined, as it consists the trivial, most straightforward method for transferring an "X-plane".

4.2 B23: Single buffered plane transfer, buffer in device memory

4.2.1 B23Y and B23X

Methods B23Y and B23X are optimisations of methods NB23Y and NB23X respectively, with the introduction of a device-side buffer. More specifically, with methods B23Y and B23X, a 2D buffer (with dimensions equal to a "Y-plane" or a "X-plane" respectively) is placed into the device memory. On demand for a field data transfer, a buffering-performing kernel is launched; the kernel is spawn for each field, with a 1D grid which contains one 2D block. The block has size $(x, y) = (N_x, N_z) = (374, 374)$, for method B23Y, and size $(x, y) = (N_y, N_z) = (374, 374)$ for method B23X. This dimensions were chosen empirically. The kernel employs a `device=device` statement to copy the contents of a "Y-plane" or an "X-plane" of a 3D field array into the 2D buffer. After the `device=device` statement finishes executing, methods B23Y and B23X copy the contents of the 2D buffer into a plane of the corresponding 3D electromagnetic field array. A faster device-to-host field data output is expected using methods B23Y and B23X instead of NB23Y and NB23X respectively, as the time-consuming misaligned device-to-host data moves are replaced by a sequential and aligned device memory access.

4.2.2 B23Z

With method B23Z, a 2D buffer (with dimensions equal to a "Z-plane") is placed into the device memory. On demand for a field data transfer, a buffering-performing kernel is launched; the kernel is spawn for each field, with a 1D grid which contains one 2D block. The block has size $(x, y) = (N_x, N_y) = (374, 374)$. This dimensions were chosen empirically. The kernel employs a `device=device` statement to copy the contents of a "Z-plane" of a 3D field array into a 2D buffer.After the `device=device` statement finishes executing, method B23Z copies the contents of the 2D buffer into a plane of the corresponding 3D electromagnetic field array. The NB23Z method performs aligned and sequential data transfers, therefore a performance improvement by using method B23Z is not supported by the theory. However, this method was implemented and its computational behaviour was explored, given the undocumented characteristics of the accelerator and CPU devices, in combination with compiler optimisations.

4.3 B22: Double buffered plane transfer, buffer in both device and host memory

4.3.1 B22Y and B22X

A further optimisation of methods B23Y and B23X was considered, which concerns the introduction of a 2D buffer in the host memory. With methods B22Y and B22X, a 2D buffer is placed into the device memory, and a kernel is spawn, exactly as in methods B23Y and B23X respectively. Methods B22Y and B22X further introduce a 2D array in the host memory, acting as a host-side buffer for the device-to-host memory transactions. The introduction of a 2D buffer which constitutes the end-point of field data transfers can eliminate the time delay happening due to strided host memory access, and thus increase the efficiency of the '`output` stage of the (FD)FDTD software for the output of a "Y-plane" or a "X-plane", for methods B22Y and B22X respectively.

4.3.2 B22Z

Method B22Z also employs a 2D buffer in the host memory. The field data are moved from the buffer in the device memory into the buffer in the host memory. Although a speedup of the `output` stage is not supported by the theory, this method was implemented and its computational behaviour was examined, given the lack of official documentation on performance characteristics of memory transfer operations of the CPU or GPU devices and given the complex nature of modern compiler optimisations.

4.4 Several planes case

The multiple planes case concerns the investigation of the transfer of "Z-planes", "Y-planes" and "X-planes". Specifically, the multiple planes performance tests examine how many planes in the FDTD space can be transferred from the device to the host memory, in shorter time than using the method NB33, by using methods NB23X, NB23Y, NB23Z, B23X, B23Y, B23Z, B22X, B22Y, B22Z.

Similarly to Section 3.5, non-adjacent discrete planes were used for the assessment of the performance of these algorithms, because should adjacent discrete planes were chosen for these performance tests, their transfer could be coalesced in a single transfer, and thus be executed more efficiently. Hence, the selection of planes illustrated in Table 4.1 was made, with N_b, $1 \leq N_b \leq 374$ being the number of planes to-be-transferred from the device memory to the host main memory. This selection of plane coordinates leads to strided memory accesses taking place between plane reads/writes,

Line buffer methods	Coordinates of planes chosen
NB23X, B23X, B22X	$x = \lfloor i \times N_x/N_b \rfloor = \lfloor i \times 374/N_b \rfloor$ $i \in \mathbb{N}, 1 \leq i \leq N_b$
NB23Y, B23Y, B22Y	$y = \lfloor i \times N_y/N_b \rfloor = \lfloor i \times 374/N_b \rfloor$ $i \in \mathbb{N}, 1 \leq i \leq N_b$
NB23Z, B23Z, B22Z	$z = \lfloor i \times N_z/N_b \rfloor = \lfloor i \times 374/N_b \rfloor$ $i \in \mathbb{N}, 1 \leq i \leq N_b$

Table 4.1: Selection of plane coordinates, for case of several planes.

with stride length equal to $\lfloor 374/N_b \rfloor$ array elements, per field array. Thus, a number of 3 tests $\times$ 3 discrete plane types $\times$ 374 different cases $= 3,366$ tests had to be produced. Hence, the process was automated using the same software infrastructure as in Section 3.5. Specifically, the modified structure of file "`input_params`", with the introduction of the "`nbuff:` " line was used for specifying the number N_b. A `bash` shell script was produced, which produces all $3,366$ different cases for testing, placing each one inside a subdirectory that starts with the prefix "DIAG-"; for example, a subdirectory with name "`DIAG-B23Y-101`" indicates that it contains a B23Y test-case with 101 discrete planes parallel to the x-z plane. Also, the script in the Ruby programming language, which was produced for the multiple line tests (described in Section 3.5) was used to investigate, for each plane-transfer method, the maximum number of planes which guarantees faster execution of the `output` than the NB33 method.

4.5 Transfer of discrete planes – Results & Discussion

4.5.1 Transfer of a single plane

The average relative `output` times $\bar{\xi}$ of each of the methods NB23X, NB23Y, NB23Z, B23X, B23Y, B23Z, B22X, B22Y, B22Z are presented in Figure **??**.

Method NB23Z executes faster than method NB33, with a speedup of $\frac{0.754213}{0.753033} \approx 1.0016$ gained. According to the theory, the speedup should have been greater, as method NB23Z theoretically makes a single coalesced device-to-host transfer of less data than NB33, as highlighted in Section 4.1.1. However, a speedup of $\frac{0.754213}{0.668391} \approx 1.1284$ is achieved – in comparison to method NB33 – with the single-buffered method B13Z, and a speedup of $\frac{0.754213}{0.665379} \approx 1.1335$ is achieved with the double-buffered method B11Z. According to the theory, a major speed up of method NB13Z was not anticipated using the buffered versions

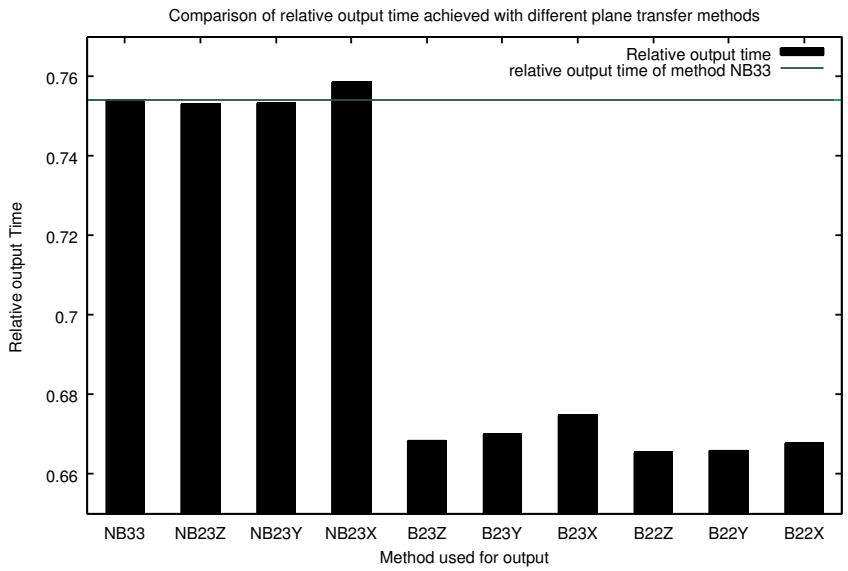


Figure 4.1: Comparison of different methods, for the output of a single plane in the 3D FDTD space.

for "Z-planes", as these methods were expected to perform coalesced and continuous memory transactions, for which a buffer would not be beneficial. Therefore, this low performance gain of method NB23Z relative to method NB33 and the high performance gain of methods B23Z and B22Z could be explained as undocumented device (GPU and CPU) performance behaviour, in combination with the PGI F90 compiler optimisations enabled by the `-fast` compiler argument.

The use of method NB23Y leads to a speedup of $\frac{0.754213}{0.753283} \approx 1.0012$ in comparison to method NB33. The relative `output` time of 0.753283 achieved by method NB23Y suggests the strided data access taking place with this method affects much its execution time, yet not as much as to constitute method NB23Y slower than method NB33 for the transfer of a "Y-plane". A speedup of $\frac{0.754213}{0.670004} \approx 1.1257$ is achieved by using method B23Y in relation to method NB33, and thus $\frac{0.753283}{0.670004} \approx 1.1243$ relative to method NB23Y. Hence, the device-side buffer provided a greater speed boost to the `output` stage of the (FD)FDTD software, by coalescing the data transfers into one single transfer. Method B22Y runs faster than method B23Y, with relative `output` time $\xi = 0.665822$, thus achieving a speedup of $\frac{0.754213}{0.665822} \approx 1.1328$ relative to method NB33 and of $\frac{0.753283}{0.665822} \approx 1.1314$ relative to method NB23Y. Hence, the elimination of strided array access patterns in the host main memory by the introduction of the 2D buffer in the host-side memory, resulted in faster memory-transfer operations, as expected by the theory (Section 4.2.1).

Method NB23X has relative `output` time $\xi = 0.758424$, thus it is slower than method NB33 by a factor of $\frac{0.758424}{0.754213} \approx 1.0066$. Therefore the strided access pattern described in Section 4.1.3 in addition to the misaligned device memory access constitutes method NB23X inefficient for a transferring the field data corresponding a plane in the FDTD

space from the device memory into the host memory. However, method B23X is of increased efficiency in comparison to method NB23X, as it achieves a relative `output` time $\xi = 0.674913$, and thus a speedup of $\frac{0.754213}{0.674913} \approx 1.1175$ relative to method NB33, due to the elimination of misaligned device memory access by the introduction of the device-side 2D buffer array. The double buffered method B22X achieves a relative `output` time $\xi = 0.6678$, and therefore a speedup of $\frac{0.754213}{0.6678} \approx 1.1294$ relative to method NB33 and a speedup of $\frac{0.674913}{0.6678} \approx 1.011$ relative to method NB23X. This figure means that the elimination of strided host memory access due to the introduction of the 2D host-side buffer decreases the relative `output` time – as suggested in Section 4.3.1, and thus decreases the overall execution time of the (FD)FDTD software.

4.5.2 Transfer of several planes

The algorithms described in Section 4.4 were implemented and their performance was assessed. The results of the Ruby script are shown in Table 4.2. Evidently, since method NB23X is slower than method NB33, the number of discrete "X-planes" that can be transferred from the device memory into the host memory using method NB23X, in a shorter time than if method NB33 is used, is zero. The relative "`output`" time ξ of methods NB23Y and NB23Z does not differ much from the time ξ of method NB33, therefore only one discrete plane can be transferred from the device memory to the host memory using methods NB23Y and NB23Z quicker than if method NB33 is used.

The introduction of a buffer in the device memory is beneficial for methods NB23X and NB23Y, yet for method NB23Z, as it increases the number of planes that can be transferred from the device memory to the host memory quicker than if method NB33 is used, from 1 (method NB23Z) to 71 (method B23Z). Also, method B23Y can transfer 37 more planes than method NB23Y, and method B23X can transfer 44 more planes than method NB23X, faster than method NB33 can.

The most efficient methods for transferring planes from the device memory to the host memory is the doubled-buffered methods. Doubled-buffered method B22X can transfer 49 planes in shorter time than method NB33 needs, which is 4 more planes than the corresponding single-buffered method B23X. Method B22Y can transfer 41 more planes than the corresponding single-buffered version B23Y. The host-side buffer is beneficial yet for method B22Z, as compiler optimisations make possible the quicker transfer of 114 planes than if method NB33 is used, which is 43 more planes than the corresponding single-buffered method, the NB23Z method, can transfer in the same time interval.

Plane transfer method	Number of planes which guarantees faster performance than NB33
NB23X	0
NB23Y	1
NB23Z	1
B23X	45
B23Y	38
B23Z	71
B22X	49
B22Y	79
B22Z	114

Table 4.2: Relative "output" timings for the single-line output methods described in Chapter 4.

Chapter 5

Approach for non-consecutive points

Another use case for the output of the (FD)FDTD software is the case where the POI are discrete points at various locations in the FDTD space, which are not necessarily consecutive in the x, y or z direction. Hence, this report examines the performance and efficiency of three ways of copying the data of such POI from the device memory to the host memory, namely: "RND-DIRECT", "RND-MAP" and "RND-LST" (the names of these methods originated during the development of this project). A pseudocode description of methods RND-DIRECT, RND-MAP and RND-LST can be found in Appendix B. For the assessment of the performance of methods 'RND-DIRECT", "RND-MAP" and "RND-LST", the location of each point amongst the POI must be specified into the (FD)FDTD software and each of these methods must be executed for an increasing number of POI. Trivially, the condition $1 \leq$ [Number of POI] $\leq N_x \times N_y \times N_z = 374 \times 374 \times 374 = 52,313,624$ must hold. Evidently, the process of the performance assessment of methods 'RND-

DIRECT", "RND-MAP" and "RND-LST" must be automated – at least to the extend of the specification of the locations of the POI.

The selection of unique pseudo-random 3D Cartesian coordinates for the POI is sufficient for the purposes of the investigation of the performance of methods "RND-DIRECT", "RND-MAP" and "RND-LST". Thus, a piece of software was developed for the generation of a specified number of such coordinates, and is described in Section 5.1. In addition, the assessment of the performance of methods "RND-DIRECT", "RND-MAP" and "RND-LST" could be quicker if the test-cases of different numbers of POI can be quickly –and therefore, automatically– produced. Hence, shell scripts were developed and used for the production of the test-cases. The pseudo-random 3D Cartesian coordinates generator can accept an argument indicating the number of POI to output, for the test-case producing shell script to be developed faster and to communicate easily with the pseudo-random 3D Cartesian coordinates generator software; thus, the production of test cases can take place inside a "for" loop of the shell script and the counter of the loop can be passed to the coordinates generator software as an argument in one line of code.

5.1 Pseudo-random 3D cartesian coordinates generator in C++11

For the automated selection of the POI, a pseudo-random 3D Cartesian coordinates generator was developed in C++. Its source code can be found in Appendix C. This piece of software receives an argument from the command line, namely "`M_poi`", which defines the number of 3D Cartesian coordinates that this software shall generate. Therefore, the condition $1 \leq$ `M_poi` $\leq N_x \times N_y \times N_z = 374 \times 374 \times 374 = 52,313,624$ must hold for the argument `M_poi`. Hence, `M_poi` must be of type `unsigned long long int`. The Standard Templates Library (STL) [27, p. 3] of the C++11 version of the C++ standard provides a way of converting a string (the command line arguments are passed into C++ programmes as strings of type `char*`) to type `unsigned long long int` in one line of code: the `std::stoull` function template [27, p. 713], which this piece of software employs for this type conversion, as it saves time from the development of the programme.

The source code of the pseudo-random 3D cartesian coordinates generator software defines a class named "Coordinate", which has three public members of type integer (`int`), variables x, y and z, representing Cartesian coordinate components. The class overloads the "less than" operator `<` to compare two objects of type "Coordinate". The comparison returns true if the x component of the left operand is less than the x component of the

right operand, and false in the opposite case. If the x components of the two operands are equal, the comparison returns true if the y component of the left operand is less than the y component of the right operand, and false in the opposite case. If the x components of the operands are equal and the y components of the operands are equal, the comparison returns true if the z component of the left operand is less than the z component of the right operand and false otherwise.

Having defined the order of objects of type "Coordinate" by overloading the "`<`" operator, the software also declares and instantiates a set container (C++ STL `std::set` template) of element type "Coordinate". C++ STL sets are containers that hold objects of a specified type by the programmer [27, p. 314], and can have objects of that type inserted into them in logarithmic time [27, p. 324], remove objects and they allow their search for objects in logarithmic time [27, p. 315]. In addition, C++ STL sets automatically keep their elements in a sorted order [27, p.314] (and hence the need to define the overloaded version of operator `<` with objects of type "Coordinate") and do not allow duplicate objects to be inserted. Their property of rejection of duplicates is utilised in the pseudo-random 3D Cartesian coordinates generator software as the infrastructure for omitting duplicate coordinates, should such be generated by the pseudo-random coordinates generator software.

The coordinates generator software begins its execution with converting the first argument passed on to its main function (*i.e.* the first command line argument) from type `char*` to type `std::string` and then from type `std::string` to type `unsinged long long int` by using the `std::stoull` function template, and its value is stored in variable `M_poi`. Next, a seed is initialised from the system clock and it enters a "for" loop, with its loop counter variable set to zero. In each iteration of the loop, three random numbers are generated, by invoking the pseudo-random number generator function `rand()` of the C standard library three times and storing its results to variables `rndx`, `rndy` and `rndz`. The modulo operation is performed on the results of the `rand()` and the number one is added to them, to keep its results bounded between 1 and $N_x = N_y = N_z = 374$. Then, an object of type Coordinate is instantiated, with x= `rndx`, y= `rndy`, and y= `rndy`, and a search operation is carried out on the set, to see if this ordered triple already exists in the set. If it exists, the current iteration of the "for"-loop is repeated, with the generation of a new pseudo-random ordered triple. Else, the ordered triple is inserted in the set and the "for"-loop proceeds to its next iteration, until its counter reaches the value `M_poi`.

The exact performance of the execution of this piece of software cannot be predicted, as it depends on the seed value for the pseudo-random number generator function. However, it was run eleven times (compiled with `-O3` compiler optimisations) for the purposes of this project and each time it could generate 10,000,000 unique 3D Cartesian coordinates within 2 hours on the RIKEN GreatWAVE front-end, and the same amount of

unique coordinates in less than 8 seconds on the RIKEN GreatWAVE "ACSG" cluster back-end, using the same configuration as the (FD)FDTD software.

5.2 Modification of the structure of the (FD)FDTD software

For the assessment of methods "RND-DIRECT", "RND-MAP" and "RND-LST", the input stage of the (FD)FDTD software was extended. An additional input file with filename "`output_points`" is read by the (FD)FDTD software, before file "`input_params`" is read. The first line of file "`output_points`" contains a number `M_poi`, with $1 \leq$ `M_poi` $\leq N_x \times N_y \times N_z = 374 \times 374 \times 374 = 52,313,624$. The first line of file "`output_points`" is followed by `M_poi` lines containing the 3D Cartesian coordinates of the POI as integers separated by single spaces. In addition, a 3D array, named `poi_lst` is declared, which contains the `M_poi` coordinates of the POI read from file "`output_points`".

5.3 RND-DIRECT: Direct device-to-host transfers

Method RND-DIRECT is the trivial solution to the non-consecutive points problem. With this method, for each 3D field array, the `output` stage of the (FD)FDTD software executes a do-loop, for `M_poi` total iterations, *i.e.* for each point amongst the POI. The body of each do-loop, makes a direct device-to-host memory transfer of the field values of the point, with a `host=device` assignment statement.

This method is expected to result in poor device occupancy and in strided host memory access. Therefore, this method is expected to perform worse than methods RND-LST and RND-MAP even for few POI and worse than method NB33 for many POI. Compiler optimisations might be beneficial to some extend, which the performance tests of the implementation of this method shall reveal.

5.4 RND-LST: Kernel invocation on a POI list, direct buffering, lookup in host

The concept behind method RND-LST is that a copy of the sorted list of POI is kept inside the device memory. For each field array, a CUDA kernel is launched across each POI. Thus, each thread should correspond to a specific point in the list of POI. Each thread copies the field value of the corresponding POI for the corresponding field array from the 3D array in device memory, into a 1D buffer array of length `M_poi` also in

the device memory. Then, the (FD)FDTD software performs a device-to-host transfer of the 1D buffer from the device memory, into a 1D buffer in the host main memory, by a `host=device` assignment statement. The data from the host-side buffer are then associated to the corresponding 3D Cartesian coordinates on the host side, and can be saved and/or be further processed at programmer's will, which is outside the scope of this project.

The CUDA kernel is launched with a grid containing one 2D block. The block has size $(x, y) = (\lfloor \frac{M\_poi}{1024} \rfloor + 1,\ 1024)$. The value 1024 was chosen because it allows the biggest number of `M_poi`, *i.e.* number of points for output, which is equal to $N_x \times N_y \times N_z = 374 \times 374 \times 374 = 52,313,624$. Therefore, the id of each thread and block can designate which point in the POI list must be transferred from the device memory to the host memory. If the id of a thread and block does not correspond to a point in the POI list, the thread terminates immediately, before it attempts to perform any data transfer operations.

The introduction of a device-side buffer is expected to increase the coalescence of the device-to-host data copy operation, and thus increase the data copy bandwidth. Also, the introduction of the host-side buffer and the elimination of strided host memory access is expected to increase the efficiency of the data copy operation. Hence, method RND-LST is expected to perform faster than method NB33 for up to a few tens of thousands of POI. It is also expected to be faster than method RND-DIRECT for several POI (from hundreds to tens of thousands of points) when the CUDA kernel can achieve high occupancy, but a worse performance than method RND-DIRECT is expected for a few tens of points, as the occupancy achieved by the CUDA kernel would be too low, due to most threads in a block being inactive.

5.5 RND-MAP: Kernel invocation on a POI list, buffering, lookup in device

Method RND-MAP also performs double-buffered device-to-host data transfers, as it places a buffer in both the device memory and in the host main memory, which provide the same benefits as for method RND-LST. Method RND-MAP introduces a 3D array of dimensions $(x, y, z) = N_x \times N_y \times N_z = 374 \times 374 \times 374$ (named as "the map"), which associates each point in the 3D FDTD space to a specific location (array index) in the POI buffer arrays (location i in the device-side buffer corresponds only to the same location i in the host-side buffer). If point $(x, y, z) = (i_0, j_0, k_0)$ in the FDTD space does not constitute a Point Of Interest, then the map associates this location to the value zero, *i.e.* $\text{map}(i_0, j_0, k_0) = 0$. For this reason, the host and device buffers have indices

starting from 0, instead of 1, and thus both buffers have length (`M_poi` + 1) each. Thus, a CUDA kernel is spawn across all points in the FDTD space, with a 2D grid of dimensions $(x, y) = (N_y, N_z) = (374, 374)$, which contains a 1D block of dimension $x = N_x$. These sizes were chosen empirically. Therefore the id of a thread designate a unique point in the 3D FDTD space. Each thread in the CUDA kernel translates the combination of its id number (i) and the id numbers (j, k) of its block, to a point $(x, y, z) = (i, j, k)$ in the FDTD space; if it corresponds to a point located out of the bounds of the FDTD space, it terminates immediately. Otherwise, the following operation takes place:

```
device_POI_buffer( device_POI_map(i,j,k) ) = device_3D_field_array(i, j, k)
```

and the thread terminates. Thus, the map is used to look-up, on the device-side, which index in the device POI buffer the data of each point in the FDTD space must be copied into. Then, the corresponding `device=device` assignment operation takes place, and finally, when the CUDA kernel terminates, the contents of the device-side buffer are copied into the host-side buffer via a `host=device` assignment. Then, the data from the host-side buffer are associated to the corresponding 3D Cartesian coordinates on the host side, and can be saved and/or be further processed.

The introduction of the two buffers should provide the same benefits mentioned in Section 5.4. High occupancy is also expected with this selection of grid and block dimensions for the CUDA kernel employed by method RND-MAP, thus this method is expected to perform well - better than method NB33 for up to hundreds of thousands of points and better for method RND-DIRECT for more than a few tens of points, for the same reasons mentioned in Section 5.4. A comparison of methods RND-MAP and RND-LST reveals that warps in method RND-MAP must be executed in two steps, which are the extraction of the value of `device_POI_map(i,j,k)` and the `device=device` assignment of the field value into the device-side buffer; whereas warps of CUDA kernel of method RND-LST perform the copy to the device-side buffer in one move. This might lead to a slower performance of method RND-MAP relative to method RND-LST.

5.6 Results & Discussion

Performance tests for methods RND-DIRECT, RND-MAP and RND-LST were created and run using shell scripts and the pseudo-random coordinates generator software which is described in Section 5.1 was utilised in the shell scripts for the automation of the test-case creation process. Table 5.1 illustrates the number of POI used for the performance test of each non-consecutive points method. The relative `output` times ξ for every number of POI of each method were aggregated, and were stored in three files, one for each method. The results were plotted using the GNU tool `gnuplot`, and are presented in Figure 5.1.

The curves shown in Figure 5.1 are Bezier curves formed by the data points, of degree 23 for method RND-DIRECT, of degree 31 for method RND-MAP and of degree 31 for method RND-LST. This method was chosen because it produces smooth curves which clearly indicate the trends followed by methods RND-DIRECT, RND-MAP and RND-LST for increasing numbers of POI.

Non-consecutive points method	**Number of POI used for performance assessment**
RND-DIRECT	1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 2500, 3000, 3500, 4000, 4500, 5000, 10E3, 20E3, 50E3, 100E3, 200E3, 500E3
RND-MAP	1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10E3, 20E3, 50E3, 100E3, 200E3, 500E3, 1E6, 1.1E6, 1.2E6, 1.3E6, 1.4E6, 1.5E6, 1.6E6, 1.7E6, 1.8E6, 1.9E6, 2E6, 5E6, 10E6
RND-LST	1, 2, 5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10E3, 20E3, 50E3, 100E3, 200E3, 500E3, 1E6, 1.1E6, 1.2E6, 1.3E6, 1.4E6, 1.5E6, 1.6E6, 1.7E6, 1.8E6, 1.9E6, 2E6, 5E6 , 10E6

Table 5.1: Number of POI used for performance assessment of methods RND-DIRECT, RND-MAP and RND-LST.

Figure 5.1 reveals that methods RND-LST and RND-DIRECT perform equally fast for up to 100 POI, and they are both faster than methods RND-MAP and NB33. This suggests that the time for the initialisation and set up of the CUDA kernel of method RND-LST acts as a bandwidth bottleneck, limiting its effectiveness for the transfer of fewer than 100 POI. Method RND-DIRECT remains faster than method NB33 for the transfer of 2500 POI or fewer. Method RND-DIRECT achieves this by transferring a smaller overall data size in the expense of performing misaligned memory access in both device and host memory. This misaligned memory access becomes more computationally expensive than they aligned copy of the field data of the whole FDTD space when the method attempts to transfer the field data of more than 2500 POI from the device memory to the host memory. Here it is noteworthy that method RND-DIRECT reaches the relative output time $\xi \approx 1$ for the transfer of more than $5,000,000$ POI, which means that almost the whole of the software execution time is spent on the data output stage. This is a clear indication of the throughput limitations induced by misaligned device memory read operations and strided host memory access.

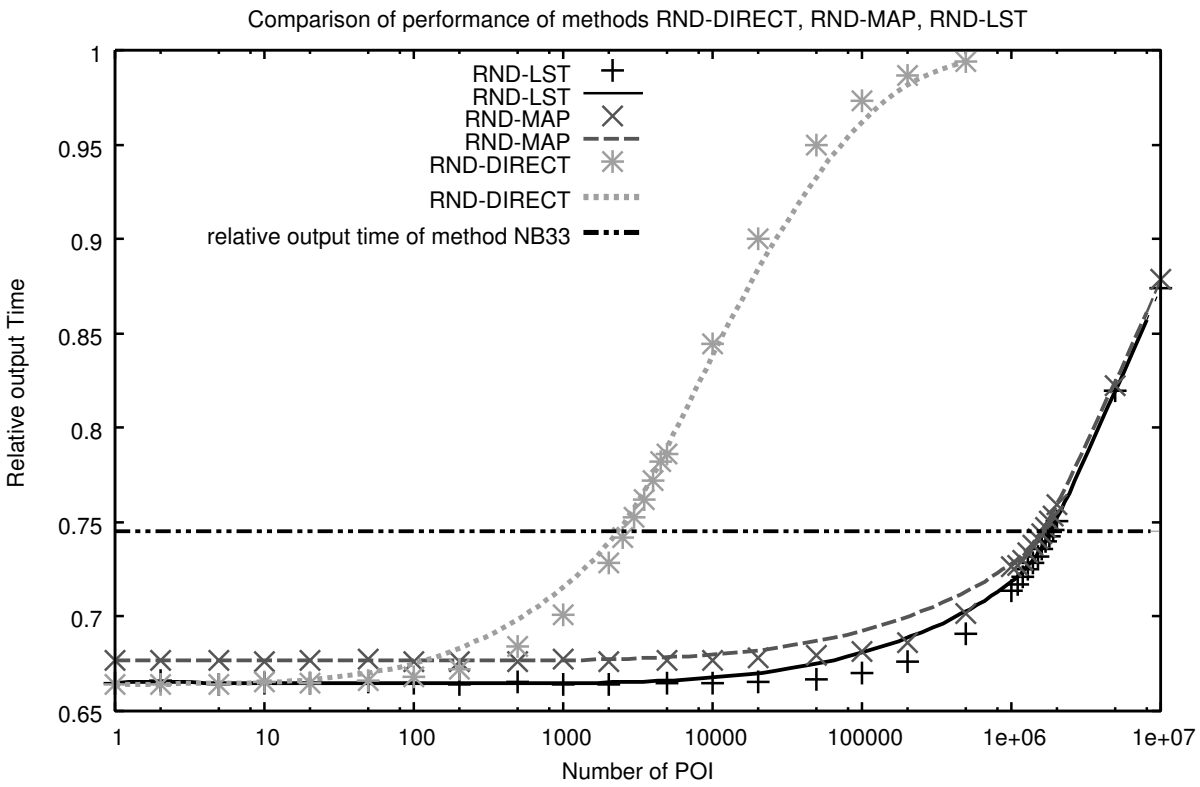


Figure 5.1: Comparison of performance of methods RND-DIRECT, RND-MAP and RND-LST.

Method RND-MAP results in faster execution than method RND-DIRECT for more than 100 POI and it remains faster than method NB33 for the transfer of up to $1,900,000$ POI. This suggests that the invocation of the CUDA kernel of method RND-MAP for the buffering of the value of electromagnetic fields at the POI and the introduction of two buffers has resulted in coalesced, aligned data transfers of fewer data and in the elimination of strided host memory access, thus significantly improving the performance of the `output` stage. Therefore method RND-MAP provides the ability to transfer 750 times the number of points method NB33 can transfer in the same amount of time.

The fastest method for data output of non-consecutive POI, amongst methods RND-DIRECT, RND-MAP and RND-LST, is proven to be RND-LST. Method RND-LST remains faster than methods RND-DIRECT, RND-MAP, RND-LST and NB33 for the transfer of up to $2,000,000$ from the device memory to the host memory. This is half a time more POI than method RND-MAP can transfer in the same amount of time. Hence, the additional implicit device memory read performed by method RND-MAP and described in Section 5.5 results in a significant latency for the transfer of up to $5,000,000$ POI.

Chapter 6

Conclusions

6.1 Achievements

The objectives of this project were completed and the aim of this project was successfully fulfilled.

Three use cases for the device-to-host transfer of a single line in the FDTD space were considered. For these use-cases, methods NB13X, NB13Y, NB13Z, B13X, B13Y, B13Z, B11X, B11Y, B11Z were developed. All these methods apart from method NB13Z are faster than method NB33 which the (FD)FDTD employed prior to the development of this project, and which is the method implied to be used in the relevant literature. Method NB13Z performs slower than method NB33 by 0.02%, whereas method NB13Y offers the smallest possible speed up of 1.0047 . Method NB13X offers a speedup of 1.12. Methods B13X and B13Z perform similarly, by speeding up the output stage by a factor of $0.75/0.67 = 1.12$. However methods B11X, B11Y, B11Z were proven to be the fastest methods for transferring the data of a single line in the FDTD space from the device memory to the host memory as they offer an approximate speedup of $0.75/0.66 = 1.14$ relative to method NB33.

The use case of the POI being laid on more than one, consecutive lines was also considered. The performance of methods NB13X, NB13Y, NB13Z, B13X, B13Y, B13Z, B11X, B11Y, B11Z was tested for this use case. The non-buffered method NB13X, single-buffered methods B13X, B13Y, and double-buffered methods B11X, B11Y and B11Z can transfer 374 non-consecutive lines in shorter time than method NB33 requires. Their comparative performance varies for different numbers of POI.

Three use cases for the device-to-host transfer of a single X-plane, Y-plane and Z-plane in the FDTD space were considered. For these cases, non-buffered methods NB23X, NB23Y, NB23Z; single-buffered methods B23X, B23Y, B23Z; and double-buffered methods B22X, B22Y, B22Z were developed. Methods NB23Z and NB23Y had a better performance than method NB33, featuring speedups of factors 1.0016 and 0.10012 cor-

respondingly. Non-buffered method NB23X was proven to be slower than method NB33 by a factor of 1.0066, due to the latency introduced by non-coalesced device memory transactions and strided host memory access. Single-buffered methods B23Z, B23Y and B23X are faster than method NB33 and also faster than the corresponding non-buffered versions, by offering speedups of factors 1.1266, 0.1243 and 1.1237 relative to the NB23Z, NB23Y and NB23X correspondingly. In particular, method B23X is faster than NB33 by a factor of 1.1175 . Double-buffered methods offer the biggest speed-ups relative to method NB33, with B22Z, B22Y and B22X being faster by a factor of 1.0045, 1.0063 and 1.0107 relative to methods B23Z, B23Y and B23Z.

Methods NB23X, NB23Y, NB23Z, B23X, B23Y, B23Z, B22X, B22Y, B22Z were assessed for the device-to-host transfer of several non-consecutive X- Y- or Z-planes. Method NB23X cannot move any planes faster than method NB33. Single-buffered method NB23X achieves the transfer of up to 45 non-consecutive X-planes and double-buffered method B22X achieves the transfer of a further four planes from the device memory to the host memory. As Y-planes are concerned, non-buffered method NB23Y can transfer one Y-plane from the device memory to the host memory in shorter time than method NB33 needs. The single-buffered method NB23Y can transfer 38 non-consecutive planes, whereas the double-buffered method B22Y can transfer 79 non-consecutive planes from the device memory to the host memory. For the transfer of Z-planes, method NB23Z can transfer only one plane in a shorter amount of time than method NB33. The single-buffered method B23Z achieves the transfer of 71 non-consecutive X-planes, whereas the double-buffered method B22Z can transfer up to 114 non-consecutive X-planes from the device memory to the host main memory in a shorter amount of time than method NB33.

Finally the use case where the POI are selected non-consecutive points was tackled by the development of methods RND-DIRECT, RND-MAP and RND-LST. Method RND-DIRECT was found to perform equally well with method RND-LST for fewer than 100 POI. For up to 2500 POI, method RND-DIRECT achieves a faster performance than method NB33, but slower performance than method RND-LST. For more than 100 POI, method RND-DIRECT performs worse than both method RND-MAP and method RND-LST. Method RND-MAP was found to perform worse than method RND-LST at all times. Method RND-MAP can transfer up to $1,900,000$ POI from the device memory to the host memory in a shorter amount of time than method NB33. Method RND-LST is the most effective method in comparison to RND-DIRECT and RND-MAP for this use case, as it can transfer up to $2,000,000$, quicker than method NB33.

From the above results it is evident that the introduction of a buffer in both the device and host memory and its exploitation, when possible, for the creation of aligned and coalesced device-to-host data transfers is much beneficial for parallelised and vectorised memory throughput–limited applications, such as those implementing the (FD)FDTD method,

and can fairly increase their performance. The above results depend on the compute capability of the NVIDIA accelerator device used (compute capability 3.5), and might produce different results if they are directly applied to older devices (with compute capability ≤ 3.5) without adjustments. However, equal or better performance advantages achieved with these methods are expected on all current and future devices with the same or bigger memory capacity.

Little future work is left for computationally similar projects which may implement the aforementioned methods. This may involve experimentation with the block and grid sizes of the buffering kernels, for the achievement of optimum performance in a specific computational environment, and/or the exploration of the performance of pinned memory in a specific computation environment. However, these activities can be completed within a few days or weeks, using the official NVIDIA profiling software `nvprof`. Finally, the possibility of exploitation of more than one GPGPU accelerators can be examined.

6.2 Self-reflection

The author has found this project to be a fair challenge. As the time management is concerned, the Gantt Chart which was produced in the first semester, was closely followed without much deviation. However, the author could had used the time of holidays more efficiently for even quicker completion of the final report. From the technical point of view, the author, for the purposes of this project, had to get familiar with concepts of concurrent GPGPU programming in time shorter than one academic semester. In the same amount of time, he also had to learn the programming languages Fortran 90 and CUDA Fortran to a usable level, and to get acquainted with the Linux supercomputer cluster environment of the RIKEN GreatWAVE supercomputer cluster. The author indeed succeeded to learn many programming languages of different programming paradigms such as Fortran 90, CUDA Fortran, Ruby, bash shell, awk and sed to a level that allowed the use of these languages by him for the purposes of the completion of this project, and familiarised himself with fundamental and advanced elements of GPGPU programming and computer architecture theory in a short amount of time. In addition, the author managed to comprehend the basic mathematical background of the (FD)FDTD method and to understand the long (FD)FDTD software he was provided with, which was written in Fortran 90 – a language he was not familiar with by that time– in a week. He also learnt how to operate the provided shared supercomputer cluster efficiently and used it in a professional manner. Despite the demanding nature of this project, the author managed to achieve the objectives and the aims of this project in the specified amount of time. Ultimately, the author enjoyed this project much and this project has increased the author's interest in the field of computational electromagnetics, a field which he would like to explore further.

References

[1] A. Taflove, "Application of the finite-difference time-domain method to sinusoidal steady-state electromagnetic-penetration problems," *Electromagnetic Compatibility, IEEE Transactions on*, no. 3, pp. 191–202, 1980.

[2] A. Taflove, "Advances in computational electrodynamics: the finite-difference time-domain method," 1998.

[3] M. I. Hallaj and O. R. Cleveland, "FDTD simulation of finite-amplitude pressure and temperature fields for biomedical ultrasound," *The Journal of the Acoustical Society of America*, vol. 105, no. 5, 1999.

[4] J. J. Simpson, "An established numerical method applied to geophysics," *Eos, Transactions American Geophysical Union*, vol. 93, no. 29, pp. 265–266, 2012.

[5] J. Beggs and W. Briley, "An implicit LU/AF FDTD method," 2001. NASA Langley Technical Report Server.

[6] L. Xanthos, "Progress Report of 3rd year Individal Project," The University of Manchester, February 2016.

[7] NVIDIA® Corporation, "NVIDIA's Next Generation CUDA Compute Architecture: Kepler™ GK110/210," tech. rep., 2014. Accessible at `http://international.download.nvidia.com/pdf/kepler/NVIDIA-Kepler-GK110-GK210-Architecture-Whitepaper.pdf` [Online; accessed 1 February 2016].

[8] A. Ruiz and M. Ujaldon, "Exploiting kepler capabilities on zernike moments," *Annals of Multicore and GPU Programming*, vol. 1, no. 1, pp. 27–37, 2014.

[9] NVIDIA®, "CUDA C Programming Guide," tech. rep., v.7.5, September 2015.

[10] NVIDIA® Corporation, "NVIDIA® Kepler® K Series Datasheet," tech. rep., October 2013.

[11] NVIDIA® Corporation, "Tuning CUDA Applications for Kepler," tech. rep., October 2013.

[12] NVIDIA®, "CUDA C Best Practices Guide," tech. rep., September 2015.

[13] The Portland Group – PGI®, "CUDA Fortran Programming Guide and Reference," tech. rep., 2014.

[14] A. Gregerson, "Implementing Fast MRI Gridding on GPUs via CUDA," NVIDIA® Corporation, 2013. Accessible at `http://www.nvidia.com/docs/IO/47905/ECE757_Project_Report_Gregerson.pdf` [Online; accessed 06 March 2015].

[15] PCI Express® , "PCI Express® base specification revision 3.0," tech. rep., November 2010, pp. 2,40.

[16] G. C. de Verdière, "Introduction to gpgpu, a hardware and software background," *Comptes Rendus Mécanique*, vol. 339, no. 23, pp. 78 – 89, 2011. High Performance Computing / Le Calcul Intensif.

[17] C. Wooley, "GPU Optimization Fundamentals," NVIDIA® Corporation, 2013. Accessible at `https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf` [Online; accessed 06 October 2015].

[18] A. Magni, C. Dubach, and M. O'Boyle, *Exploiting GPU Hardware Saturation for Fast Compiler Optimization*, pp. 99–106. ACM, 2014.

[19] M. Livesey and F. Costen and T. Nanri and N. Nakashima and S. Fujino and others, "Development of a CUDA Implementation of the 3D FDTD Method," *IEEE Antennas & Propagation Magazine*, vol. 54, no. 5, pp. 186–195, 2012.

[20] Y. Inoue and M. Unno and S. Aono and H. Asai, "GPGPU-based ADE-FDTD method for fast electromagnetic field simulation and its estimation," in *Asia-Pacific Microwave Conference 2011*, pp. 733–736, Dec 2011.

[21] J. Sheaffer and M. van Walstijn and F. Bruno, "Physical and numerical constraints in source modeling for finite difference simulation of room acoustics," *Journal of the Acoustical Society of America*, vol. 135, pp. 251–261, 1 2014.

[22] ISO/IEC, *ISO/IEC 1539:1991. Information technology – Programming languages – FORTRAN.* ISO/IEC, 1991.

[23] RIKEN, "Official Website of RIKEN." `http://www.riken.jp/en/about/intro`. [Online; accessed 10 February 2016].

[24] Advanced Center for Computing and Communication, RIKEN, *HOKUSAI-GreatWave User's Guide*, 3 ed., September 2015.

[25] D. B. Loveman, "High performance fortran," *Parallel & Distributed Technology: Systems & Applications, IEEE*, vol. 1, no. 1.

[26] "Advanced CUDA Webinar – Memory Optimizations," NVIDIA® Corporation, p. 14, 2009. Accessible at `http://on-demand.gputechconf.com/gtc-express/2011/presentations/NVIDIA_GPU_Computing_Webinars_CUDA_Memory_Optimization.pdf` [Online; accessed 15 February 2016].

[27] N. Josuttis, *The C++ Standard Library: A Tutorial and Reference*. Addison-Wesley, 2 ed., 2012.

Appendix A

The 1st semester Progress Report

The University of Manchester

SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

Progress Report

GPGPU: Acceleration of output of data

Loukas Xanthos
9408845

Supervised by:
Dr. Fumie Costen

Third Year Individual Project 2015/16

Contents

Chapter 1

Introduction

1.1 Introduction

As engineering problems get more complex, the need for their modelling and simulation becomes intense. The academia and industry are putting effort in developing and deploying mathematical models and computer software able to simulate and model real phenomena and their interactions accurately. Nevertheless, the increased problem size and model accuracy cannot be completely handled even with the modern processors.

Hence, software engineers try to find techniques and tools that help accelerating the execution of their software. There is a number of optimisations which are carried out in execution thread level, to ensure no wasting of computational resources. Also, developers achieve utilising modern hardware by parallelising their software, as this can boost the execution performance (execution can become tens of times faster). Nowadays, available Central Processing Units (CPUs) may have clock rates of less than 10 GHz and as many as tens of processing cores, whereas a single modern Graphics Processing Unit (GPU) can provide thousands of cores, which can be used by software and support floating point arithmetic. This ability for high parallelisation GPUs provide has made General Purpose GPU processing (GPGPU) popular. Graphic card manufacturers now promote GPU accelerated computing and they design accelerator products that target scientists and engineers.

This project investigates the improvement of the parallelisation of a software that performs Finite-Difference Time-Domain (FDTD) [1] simulation of propagation of electromagnetic waves in the 3D space. This piece of software is ported to run on NVIDIA accelerators from its former CPU version. However, its current GPGPU implementation is found to be inefficient and needs to be improved by introducing ad-hoc algorithms and extending its design with other software design techniques.

1.2 Motivation

In addition to the usage of the FDTD method for simulations of propagation of Electro-Magnetic (EM) waves and their interactions in the electrical and electronic sectors, the FDTD method is used to model computationally similar phenomena to the propagation of the EM waves in the biomedical [2] and geophysics [3] sectors. Besides, the same FDTD method, with minor modifications, is used to model and solve Computational Fluid Dynamics (CFD) problems [4]. Thus, a thorough investigation of the possibilities of accelerating the major components of FDTD software can benefit all these sectors.

A parallelised and vectorised Frequency-Depended FDTD (FD-FDTD) software has been developed by Dr. Fumie Costen's research group at the University of Manchester. Nevertheless, the group's findings indicate that more than half the total run time is spent on the data output stage. At this stage, field data located in the accelerator memory must be moved into the system's main memory. Moreover, there are cases when the

FDTD software user is interested on the field values of points located on just one line of the FDTD space, or on just one plane. They could also be interested in two or more lines or planes, or merely on a few selected points. In the group's current implementation of the FDTD EM simulation software this is not possible; the user has to wait until they get all the field data for the whole 3D space instead.

The implementation described above is inefficient, as computational power is not utilised at all while the software user has to wait until they get the results and no calculations of field values are being done. This project investigates how this issue can be solved. This investigation will be performed in two stages:

- First, techniques for speeding up the output stage will be considered.
- Next, the output stage will be attempted to become concurrent with the other main software components.

1.3 The aims and objectives of this project

1.3.1 Aims

The aim of this project is the acceleration of FD-FDTD computation on the GPU from the view point of the data output.

1.3.2 Objectives

- Become familiar with Linux environments, with the RIKEN HOKUSAI GreatWAVE cluster, with the awk, sed and gnuplot utilities, with Fortran 90 and CUDA Fortran
- Modifications of the given FD-FDTD software to make it compatible with the RIKEN HOKUSAI GreatWAVE platform
- Implementation of time measuring source code into the given software
- Development of benchmarking and performance data analysis scripts
- Identification of main techniques currently used for speeding up data transfers between accelerators (GPU) and main memory (CPU)
- Design of algorithms for speeding up the output of FDTD field data located on: a single line of the FDTD space, more than one line of the FDTD space, a single plane of the FDTD space, more than one planes of the FDTD space, selected points of the FDTD space
- Thorough investigation of the performance of the above algorithms

Chapter 2

Supporting Theory

2.1 Analysis of the Finite-Difference Time-Domain Method

2.1.1 The Maxwell equations

The time-domain Maxwell equations are:

$$\nabla \times \mathbf{H} = \frac{\partial \mathbf{D}}{\partial t} + \mathbf{J} \qquad \text{Ampere's Law} \tag{2.1}$$

$$\nabla \times \mathbf{E} = -\frac{\partial \mathbf{B}}{\partial t} \qquad \text{Faraday's Law} \tag{2.2}$$

$$\nabla \cdot \mathbf{D} = \rho_v \qquad \text{Gauss' Law for the electric field} \tag{2.3}$$

$$\nabla \cdot \mathbf{B} = 0 \qquad \text{Gauss' Law for the magnetic field} \tag{2.4}$$

, in MKS units, where $\mathbf{E}$ is the electric field intensity (in $\frac{\text{Volt}}{\text{Metre}}$), $\mathbf{D}$ is the electric displacement field (in $\frac{\text{Coulomb}}{\text{Metre}^2}$), $\mathbf{B}$ is the magnetic flux density (in Tesla), $\mathbf{H}$ is the magnetic field strength (in $\frac{\text{Ampere}}{\text{Metre}}$), $\mathbf{J}$ is the conduction current density (in $\frac{\text{Ampere}}{\text{Metre}^2}$), ρ_v is the electric charge density (in $\frac{\text{Coulomb}}{\text{Metre}^3}$) .

And using the microscopic form of Ohm's Law ($\mathbf{J} = \sigma\mathbf{E}$), eq. (2.1) becomes:

$$\nabla \times \mathbf{H} = \frac{\partial \mathbf{D}}{\partial t} + \sigma\mathbf{E} \tag{2.5}$$

For a homogeneous medium, which has material depended permittivity ϵ (in $\frac{\text{Farad}}{\text{Metre}}$) and permeability μ (in $\frac{\text{Henry}}{\text{Metre}}$), ϵ_0 ($\frac{\text{F}}{m}$) is the permittivity of free space, ϵ_r ($\frac{\text{F}}{m}$) is the relative permittivity of the material, μ_0 ($= 4\pi/10^7 \frac{\text{H}}{\text{m}}$) is the permeability of free space and μ_r ($\frac{\text{F}}{\text{m}}$) is the permeability of free space and the flux densities are linked to the field intensities with eq. (2.6) and (2.7):

$$\mathbf{D} = \epsilon\mathbf{E} = \epsilon_r\epsilon_0\mathbf{E} \tag{2.6}$$

$$\mathbf{B} = \mu\mathbf{H} = \mu_r\mu_0\mathbf{H} \tag{2.7}$$

2.1.2 Frequency dependency of material parameters

For fields varying with frequency $f = \frac{\omega}{2\pi}$, equations (2.5) and (2.6) combine to [5]:

$$\nabla \times \boldsymbol{H} = \frac{\partial \boldsymbol{D}}{\partial t} + \sigma \boldsymbol{E} = \frac{\partial \epsilon \boldsymbol{E}}{\partial t} + \sigma \boldsymbol{E} = \left(\sigma + \epsilon \frac{\partial}{\partial t}\right) \boldsymbol{E} \tag{2.8}$$
$$= (\sigma + \jmath\omega\epsilon_0\epsilon_r)\, \boldsymbol{E} = \jmath\omega\epsilon_0 \left(\epsilon_r + \frac{\sigma}{\jmath\omega\epsilon_0}\right) \boldsymbol{E} = \frac{\partial \epsilon \boldsymbol{E}}{\partial t} = \frac{\partial \boldsymbol{D}}{\partial t}$$

where

$$\epsilon = \epsilon_r\epsilon_0 - \jmath\frac{\sigma}{\omega} = \epsilon_0 \left(\epsilon_r - \jmath\frac{\sigma}{\omega\epsilon_0}\right). \tag{2.9}$$

The first-order Debye model can be used to model dispersive materials. Thus, the relative permittivity can be expressed as [5]:

$$\epsilon_r = \epsilon_\infty + \frac{\epsilon_\mathrm{S} - \epsilon_\infty}{\jmath\omega\tau_\mathrm{D} + 1} \tag{2.10}$$

, where ϵ_∞ is the infinite relative dielectric constant (optical permittivity), ϵ_S is the relative dielectric constant at lower frequencies (much lower than 4×10^{14} Hz), ω is the angular frequency of the varying fields and τ_D is the characteristic relaxation time of the medium.

The **E**-field is now related to the **D**-field using (2.6) and (2.10) [5]:

$$\boldsymbol{D} = (\epsilon_0\epsilon_\infty + \frac{\epsilon_0\epsilon_\mathrm{S} - \epsilon_0\epsilon_\infty}{\jmath\omega\tau_\mathrm{D} + 1} - \jmath\frac{\sigma}{\omega})\boldsymbol{E} \tag{2.11}$$
$$= (\epsilon_0\epsilon_\infty + \frac{\epsilon_0\epsilon_\mathrm{S} - \epsilon_0\epsilon_\infty}{\jmath\omega\tau_\mathrm{D} + 1} + \frac{\sigma}{\jmath\omega})\boldsymbol{E}$$
$$= \frac{(\jmath\omega)^2\epsilon_0\epsilon_\infty\tau_\mathrm{D} + \jmath\omega(\epsilon_0\epsilon_\mathrm{S} + \sigma\tau_\mathrm{D}) + \sigma}{\jmath\omega(\jmath\omega\tau_\mathrm{D} + 1)}\boldsymbol{E}.$$

2.1.3 Kane Yee's algorithm

The FDTD algorithm was first proposed by Kane Yee in 1966 [6][1]. His method solves Maxwell's curl equations using a "leapfrog time-stepping process". The method works as follows:

First, the 3D space is discretised in 3D cells. Each cell edge has the size of a unit of distance and contains values of the electric or magnetic field inside it. Cells that represent the magnetic field are placed $\frac{1}{2}$ unit of distance apart. This is called the Yee Grid. The Yee Grid is the basis for FDTD method's simplicity, as Maxwell's divergence equations become:

$$\nabla \cdot \mathbf{D} = 0 \tag{2.12}$$

$$\nabla \cdot \mathbf{B} = 0 \tag{2.13}$$

Hence, equations (2.12) and (2.13) do not need to participate in the solution process.

Yee's algorithm is the following:

1. Discretise Maxwell's curl equations (i.e. Ampere's Law and Faraday's Law) in space and time. All derivatives should employ central differences. Solving these

finite difference equations yields the "update equations", which describe the field values of future time steps based on the field values of past time-steps.

2. Calculate the value of **H**-field of the next time-step from the present **E**-field values.
3. Calculate the value of **E**-field of the next time-step from the present **H**-field values.
4. Repeat from step 2, until the specified time-step limit by the user is exhausted.

The equations used for the calculation of the field values are presented in Appendix B

2.1.4 Frequency Depended (FD-)FDTD

Equation 2.11 yields a relation between the **D**-field, the **E**-field and the frequency-depended parameters of the media. Hence, this relation is exploited when dispersive media are present in the simulation. With the assumption that the environment is source-free, the scalar value of the x,y and z components of the **D**-field of the next iteration $(n+1)$ can be obtained using the **E**-field data from the current iteration (n) [5]. Equations for obtention of the values of the **D**-field components can be found in Appendix B (equations B.7, B.8, B.9). The FD-FDTD algorithm is summarised with a flow-chart in figure 2.1.

2.2 General Purpose GPU computing – merits and demerits

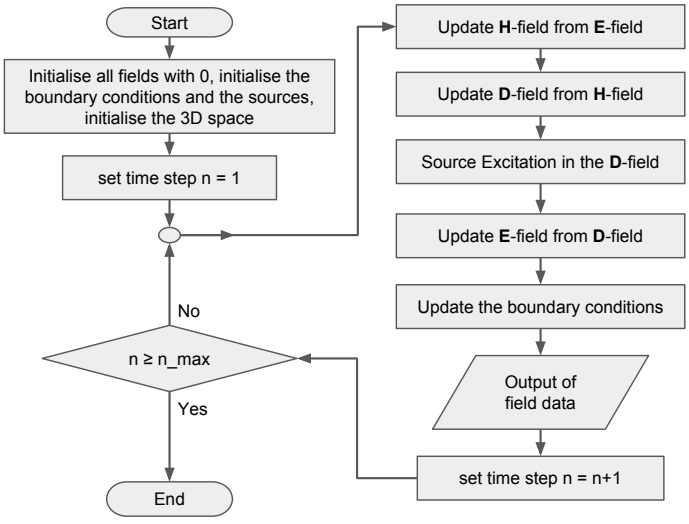


Figure 2.1: Flowchart of the FD-FDTD method

Nowadays, GPU vendors sell programmable graphic cards accommodating General Purpose GPUs (GPGPUs). GPGPUs, besides their standard use as graphics processors and for outputting graphics to displays, can also be programmed to accelerate scientific computations. They are multi-core devices featuring a single instruction multiple data (SIMD) architecture.

Modern processors like the Intel Xeon E5 v3 family have tens of cores and can offer a performance class of 1010 GigaFLOPS [7] , whereas accelerators such as the NVIDIA Kepler K20X have thousands of cores and can deliver over 1 TeraFLOPS [8]. This difference is merely due to the number of processing cores each device has, and thus due to the number of threads each device can execute concurrently.

However, GPUs are not as good as CPUs in executing single-threaded tasks, as GPUs are highly optimised mainly for parallel execution of thousands of simple tasks and for performance per watt [9], while CPUs are mainly optimised for executing serial tasks. Multi-core CPUs are also optimised for parallel execution and feature some SIMD instructions, but they are not as energy efficient as GPUs and cannot compete with them in executing thousands of parallel tasks. Therefore, given an algorithm that can be fully parallelised and vectorised, GPUs are more efficient, since the increased number of cores

can lead to a very small execution time and CPUs might require ten times the amount of the energy GPUs use to achieve the same results [9].

Moreover, the memory on a GPU accelerator card often has less capacity than the system's main memory. In addition, GPU accelerators (called "devices") are attached to systems (called "hosts") using the PCI-Express (PCIe) interface. PCIe has a raw bit rate of 8 GT/s [10], and 8Gb/s per lane per link [10]. Thus, the low PCIe data rates become a bottleneck when frequent data transfers between the device and host memory are required. To achieve satisfying performance, programmes need to "saturate" the hardware, i.e. to execute enough threads to fully use all of the device's resources, so that memory operations are fewer than any other processing happening on the device [11][12][13].

2.3 The computational environment of this project

This project is going to be developed, debugged and run on the HOKUSAI GreatWAVE supercomputer[14] of the Advanced Centre for Computing and Communication, RIKEN, located in Saitama, Japan.

More specifically, the Application Computing Server with GPUs (ACSG) cluster will be used, which is equipped with 20 Intel Xeon E5–2670 v3 CPUs, which are clocked at 2.3 GHz, providing a theoretical peak performance of 36.8 GFLOPS per core per CPU. In the ASCG cluster, there are 64GB of main memory installed per unit (or "node"). The memory bandwidth is 68.2 GB/s/CPU and 0.15 Byte/FLOP.

The ASCG cluster is also equipped with Kepler 4 K20Xm accelerators. The GPU clock of these accelerator has an operating frequency of 732 MHz. Every device has 6GB of global memory, and a bandwidth of 250 GB/s. The peak performance of the GPGPU in single precision is 3950 GFLOPS, and 1310 GFLOPS in double precision [15].

This project will use 8 CPU cores and 1 accelerator, thus 6GB of accelerator memory.

All clusters of the HOKUSAI GreatWAVE supercomputer run on a Linux operating system that is supplied with the GNU C compiler, the PGI Fortran compiler, bash shell, and popular linux tools such as grep, sed and awk.

The RIKEN HOKUSAI GreatWAVE supercomputer cluster operates on a Linux platform (Linux kernel version 2.6) and features a front-end — back-end system [14]. GreatWAVE users are expected to connect to the front-end and use it for software development (source code editing, compilation and link procedures), file transfers between the RIKEN cluster and their own computers, basic software debugging and tuning and job management. Jobs run on the back-end [14]. A diagram of the sub-systems of the HOKUSAI GreatWAVE supercomputer and their connections can be found in Appendix C.

Chapter 3

Practical progress so far

3.1 Source code modifications for compatibility

The group's FD-FDTD software was examined and its various components were identified. However, the software was intended to run on an older platform and to be compiled using an older version of the PGI Fortran compiler. Hence, the source code given to be

modified was found to be incompatible with the new platform (i.e. the RIKEN HOKUSAI GreatWAVE system), and major modifications were applied to make it compatible. These modifications included the change of the name of a variable (from '`sum`' to '`sum2`'), because the old name is a Fortran 90 intrinsic function and that function is overloaded by the '`cudafor`' module. Also, the PGI Fortran compiler required that the extension of all CUDA Fortran source filenames to be changed from `.f90` to `.cuf`. As a consequence, any '`include`' statements in the Fortran source code raise compilation errors because the referenced `.f90` files cannot be found. Thus, they need to be updated to reference the corresponding `.cuf` files. The use of sed scripts for text filtering helped the author save time and make no mistakes during the process of fixing all the 'include' statements in all the affected Fortran source code files.

3.2 Set-up of the computational and developmental environment at the front-end

The group's FD-FDTD programme is executed on the Application Computer Server (ACS) with GPU ("ACSG") cluster, where NVIDIA Kepler K20Xm accelerators and Intel Xeon E5–2670 v3 CPUs are installed. Thus, the compilation process must target the 64-bit Intel Haswell architecture and NVIDIA Kepler accelerators with 3.5 compute capability. The first try for compiling the given FDTD software (on the front-end) targeted to the 64-bit Intel Haswell architecture with `-O2` level compiler optimisations resulted in error messages from the GNU assembler, which indicated that the assembler could not understand the mnemonics of some instructions used by modern 64-bit Haswell Intel Xeon processors [14]. The HOKUSAI GreatWAVE administrators were contacted via email about the issue. They suggested that the newest version of GNU binutils (and hence, the GNU assembler) should be installed into our Linux user's 'home' directory, as our research group has not got global system access permissions. The author undertook the work of downloading and installing the newest version of GNU binutils on a local directory. This indeed solved the issue.

However, to bypass the system-wide installed version of GNU binutils and to build the software using the newest version of the assembler located in our 'home' directory, required updating the `$PATH` variable after logging in to the front-end. To eliminate this time-consuming requirement, 'export' statement was added in the `.bashrc` file , on the HOKUSAI GreatWAVE front-end. Now, the update of the `$PATH` variable occurs any time we connect to the HOKUSAI GreatWAVE front-end, allowing anyone in our research group to use the newest version of GNU binutils without having to manually update the value of the `$PATH` variable.

3.3 Identification of main performance indicators

Numerous changes are being applied to the given software, to find the optimum way to accelerate the transfer of data from the accelerator memory to the host memory. After any changes are applied, two main parameters must be measured, to indicate

- how much faster the new version is and
- to make the processes of debugging and further development easier.

These are the time to execute each discrete software component, and the size of any data transfers between the host and the accelerator – as the execution of the same operations on different data sizes might result in bigger or smaller execution times. The measurement of the size of data transfer between the GPU and the host memory can be achieved using NVIDIA's profiler, nvprof, with the 'memtransfersize' argument.

The given GPGPU FD-FDTD software used consists of the following main components:

1. `src`: The component that performs the update of the field values of the points where the sources reside

2. `fdtd`: the calculation of the field values in each cell of the Yee Grid.

3. `output`: the output of the field data to the standard output.

As this project is concerned with the speed up of the output of data of a pre-existing FD-FDTD implementation, the execution time of the pre-existing components must be measured. The main point of concern is the total amount of time the FD-FDTD software user has to wait until they get the results, hence wall-clock time is measured; not cpu time or user time. This is the time needed for the various FD-FDTD software components to finish, including any system calls, memory allocation/deallocation or waiting for system or input/output resources to become available.

Thus, the wall-time of the execution of these components is measured individually, as well as the execution time of the application from start to finish, without including the time needed for memory allocation in the beginning of the execution, and without including the time needed at the end of the programme's execution for memory deallocation. For the purposes of this project, this is referred to as the '`prog`' time.

By performing these measurements, various optimisations can be applied to the software given. As this project focuses on the data output stage, any optimisations attempted must focus on making the time of the '`output`' stage shorter, i.e. making the '`output`' wall-time to be the shortest possible percentage of the '`prog`' wall-time.

After verifying that the algorithm, which is considered to be the optimum, performs better than any previous ideas, an attempt to overlap its execution with other software components will be made, thus to "hide" the execution time of the '`output`' component using a pipeline. At this development stage, the total wall-time of the '`output`' stage ceases to be an significant measure for further development. Instead, the time required for the part of the '`output`' stage that cannot be hidden (i.e. cannot be interleaved with other software components) must be measured.

3.4 Procedure currently followed for the measurement of programme execution time

Currently, all software components have been parallelised and vectorised, but the execution of each discrete component does not concur with the execution of any other component. The execution wall-time of the various FD-FDTD software components described in section 3.3 is measured by taking the difference between two clock times, one at the beginning of the component execution and one at the end. Clock time is obtained by calling the Fortran 90 `system_clock` function [16].

Two variables are needed for each component: One for the beginning of the wall-time measurement, and one for the end. Both obtain a value by calling the `system_clock` function. In addition, the given FD-FDTD software implements an "unrolled" version of the FD-FDTD algorithm, in which all software components occur three times per time step. Thus, three source code instances correspond to each component. This results in the need of having six variables for each component, three for storing the beginning time of the component execution and three for their end of execution. Moreover, one variable for each component is used, to hold the total execution time, by accumulating the execution time of each of the three loops. A list of all the variables used for the purpose of wall-time measurements can be found in Appendix A.

However, the `system_clock` function returns the number of CPU clock counts passed since an undefined time. Therefore, it is possible that it will 'roll-over' at an undefined time. To resolve this situation, if-else statements are used as a sanity check (a component's execution cannot terminate before this component begins its execution) and the returned value from the `system_clock` function is adjusted accordingly.

The output of the FD-FDTD software after the completion of its execution generates timing information similar to the following:

```
Timings by loukas (wall time)
prog     42279568 or    42.27957     s
src     13288758 or    13.28876     s
fdtd      9320457 or    9.320457     s
output     19668385 or    19.66838     s
```

The first column describes the component for which measurements will follow on the same line. Next, the total number of CPU clock counts is shown, and after two columns the total number of seconds elapsed on this component is printed.

Since many instances of the same source code will be generated for the intents of this project, a shell script that appends the time-measuring source code to the FD-FDTD software source code given was also produced. The execution time of individual CUDA Kernels, can be observed using NVIDIA's profiler (`nvprof`) via the command line.

3.5 The current data analysis process

Many possible solutions are being considered for speeding up the output of field data from the accelerator memory to the system's main memory. When a new concept is implemented into the source code of the FD-FDTD software, the experiment is run at least three times, to eliminate the effect the system's state might have on the final results. Then, the average of the resulting wall-time measurements is calculated. The maximum absolute deviation of each arithmetic mean time (i.e. for '`prog`', '`src`', '`fdtd`' or '`output`') from the corresponding individual measurements is also calculated, as a confirmation that the experiment results do not change dramatically (i.e. the absolute deviation of any percentage time should not be more than two percentage points higher or lower the corresponding mean percentage time). In addition, awk scripts have been developed for the purposes of automatic calculation of the information described above, and for generating experiment result data which are ready to be plotted using the gnuplot plotting utility.

Chapter 4

Future work and conclusions

4.1 Planned future work – possible technical risks

The author has already begun to design an algorithm for accelerating the given FD-FDTD software's output for points located on a single line of the 3D space, following his time plan[1] without deviations so far. Future work includes the design and testing of algorithms for accelerating the output of data located on specific locations of the 3D space. As explained in the introduction of this report, three cases for the location of these points

[1]The project's plan can be found in Appendix D

will be considered. Two weeks are assigned for the development of each algorithm and one week is assigned for performance measurements, thus enough time margin is provided for algorithm re-considerations. No work is assigned during the Christmas holidays, and no practical work is assigned during the Easter holidays, so that any deviations from the time plan can be compensated during those days.

An effort for analysing this project technical risks[2] has been made. Hard disk failures leading to loss of source code or experimental results may be catastrophic for the this project's development, as it is based entirely on these. For this reason, daily backups are kept in the group's computer cluster and any important documents are uploaded on online cloud storage services (including the University's online file storage service for students). Also, valuable time will be lost in case the ACSG cluster is busy or goes through unplanned maintenance, or for any other reason it cannot execute any requested experiments. As these situations are beyond the author's control, any practical project work is carried out before any documentation work and experimental versions of the software are submitted in ACSG's execution queue as early as possible, to leave ample time in case of mishaps. Ultimately, the design of algorithms for the speed up of the data output stage of the given FD-FDTD software might turn out to be impossible, or at least, the author's knowledge and experience might not suffice to achieve it by the time this project must be completed, even if more literature is reviewed. In that case, a meticulous analysis of the steps followed and the reasons for their failure will be carried out.

4.2 Conclusions

In conclusion, this project investigates the acceleration of the output of data of a pre-existing FD-FDTD electromagnetic wave propagation simulation software, which uses general-purpose GPUs to accelerate its major components. However, because the limited bandwidth of the PCI-Express interface, used to connect the accelerators to the main system, is not taken into consideration in the current implementation of the software, the software does not utilise the GPU accelerator in its full potential. The system where all software experiments will be attempted is the ACSG cluster of the RIKEN's HOKUSAI-GreatWAVE supercomputer, which is equipped with NVIDIA Kepler K20Xm accelerators. The environment of this system was modified to support the objectives of this project. In addition, as the author gained experience with the GNU/Linux Operating System and the Fortran 90 programming language, the FDTD software under examination was modified to output timing information of its main components. This is going to be useful for any further modification work targeting to reduce the execution time of the '`output`' stage. Also, several scripts were coded, to render the development and testing phases partially automated, thus easier and quicker. A health and safety risk assessment can be found in Appendix F.

[2] A table containing possible technical risks and any precautions taken can be found in Appendix E.

References

[1] A. Taflove, "Application of the finite-difference time-domain method to sinusoidal steady-state electromagnetic-penetration problems," *Electromagnetic Compatibility, IEEE Transactions on*, no. 3, pp. 191–202, 1980.

[2] M. I. Hallaj and O. R. Cleveland, "FDTD simulation of finite-amplitude pressure and temperature fields for biomedical ultrasound," *The Journal of the Acoustical Society of America*, vol. 105, no. 5, 1999.

[3] J. J. Simpson, "An established numerical method applied to geophysics," *Eos, Transactions American Geophysical Union*, vol. 93, no. 29, pp. 265–266, 2012.

[4] J. Beggs and W. Briley, "An implicit LU/AF FDTD method," 2001. NASA Langley Technical Report Server.

[5] F. Costen, *High Speed Computational Modeling in the Applicaton of UWB Signals*. PhD thesis, PhD thesis, University of Manchester, 2005.

[6] K. Yee, "Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media," *IEEE Trans. Antennas Propagat.*, 1966.

[7] G. Lento, "Intel® advanced vector extensions," tech. rep., Intel® Corporation, September 2014.

[8] NVIDIA® Corporation, "NVIDIA® Tesla® K Series Datasheet," tech. rep., October 2013.

[9] NVIDIA® Corporation, "Doing more with less of a scarce resource." `http://www.nvidia.com/object/gcr-energy-efficiency.html`. [Online; accessed 01 October 2015].

[10] PCI Express® , "PCI Express® base specification revision 3.0," tech. rep., November 2010, pp. 2,40.

[11] G. C. de Verdière, "Introduction to gpgpu, a hardware and software background," *Comptes Rendus Mécanique*, vol. 339, no. 23, pp. 78 – 89, 2011. High Performance Computing / Le Calcul Intensif.

[12] C. Wooley, "GPU Optimization Fundamentals," NVIDIA® Corporation, 2013. Accessible at `https://www.olcf.ornl.gov/wp-content/uploads/2013/02/GPU_Opt_Fund-CW1.pdf` [Online; accessed 06 October 2015].

[13] A. Magni, C. Dubach, and M. O'Boyle, *Exploiting GPU Hardware Saturation for Fast Compiler Optimization*, pp. 99–106. ACM, 2014.

[14] Advanced Center for Computing and Communication, RIKEN, *HOKUSAI-GreatWave User's Guide*, 3 ed., September 2015.

[15] NVIDIA® Corporation, "NVIDIA Tesla®K20-K20X GPU Accelerators Benchmarks," tech. rep., November 2012. Accessible at `http://www.nvidia.com/docs/IO/122874/K20-and-K20X-application-performance-technical-brief.pdf` [Online; accessed 06 October 2015].

[16] ISO/IEC, *ISO/IEC 1539:1991. Information technology – Programming languages – FORTRAN*. ISO/IEC, 1991.

Appendix A

Variables declared for the purpose of wall-time measurements

The following variables have been declared and used for the purpose of wall-time measurement:

```
!-----------------------------------------------------------------
!    Variables for timing
!-----------------------------------------------------------------
       integer :: count_startprog, count_endprog, count_prog
       integer :: count_startsource1, count_startsource2, count_startsource3
       integer :: count_startfdtd1, count_startfdtd2, count_startfdtd3
       integer :: count_endsource1, count_endsource2, count_endsource3
       integer :: count_endfdtd1, count_endfdtd2, count_endfdtd3
       integer :: count_startoutput1, count_startoutput2, count_startoutput3
       integer :: count_endoutput1, count_endoutput2, count_endoutput3
       integer :: count_source, count_fdtd, count_output
```

Appendix B

Finite Difference (FD) Equations used in the FDTD method

B.1 Obtention of the H^n-field from the E^n-field

[1] From Equation 2.2 and Equation 2.7, the values of the xyz components of the **H**-field can be obtained using central finite differences of the components of the **E**-field: (Equations from [5])

For the x component:

$$H_x^n{}_{(i,j,k)} = \frac{\mu^{n-1}{}_{(i,j,k)} H_x^{n-1}{}_{(i,j,k)}}{\mu^n{}_{(i,j,k)}} - \frac{\Delta t}{\mu^n{}_{(i,j,k)}} \left[\frac{E_z^n{}_{(i,j,k)} - E_z^n{}_{(i,j-1,k)}}{\Delta y} - \frac{E_y^n{}_{(i,j,k)} - E_y^n{}_{(i,j,k-1)}}{\Delta z} \right] \quad [i_{min}+1 \le i \le i_{max}-1, j_{min}+1 \le j \le j_{max}, k_{min}+1 \le k \le k_{max}]. \tag{B.1}$$

For the y component:

$$H_y^n{}_{(i,j,k)} = \frac{\mu^{n-1}{}_{(i,j,k)} H_y^{n-1}{}_{(i,j,k)}}{\mu^n{}_{(i,j,k)}} - \frac{\Delta t}{\mu^n{}_{(i,j,k)}} \left[\frac{E_x^n{}_{(i,j,k)} - E_x^n{}_{(i,j,k-1)}}{\Delta z} - \frac{E_z^n{}_{(i,j,k)} - E_z^n{}_{(i-1,j,k)}}{\Delta x} \right] \quad [i_{min}+1 \le i \le i_{max}, j_{min}+1 \le j \le j_{max}-1, k_{min}+1 \le k \le k_{max}]. \tag{B.2}$$

For the z component:

$$H_z^n{}_{(i,j,k)} = \frac{\mu^{n-1}{}_{(i,j,k)} H_z^{n-1}{}_{(i,j,k)}}{\mu^n{}_{(i,j,k)}} - \frac{\Delta t}{\mu^n{}_{(i,j,k)}} \left[\frac{E_y^n{}_{(i,j,k)} - E_y^n{}_{(i-1,j,k)}}{\Delta x} - \frac{E_x^n{}_{(i,j,k)} - E_x^n{}_{(i,j-1,k)}}{\Delta y} \right] \quad [i_{min}+1 \le i \le i_{max}, j_{min}+1 \le j \le j_{max}, k_{min}+1 \le k \le k_{max}-1]. \tag{B.3}$$

[1] The exponent n means that the field values refer to the current time step. The exponent $n+1$ means that the field values refer to the next time step.

B.2 Obtention of the $\mathbf{E}^{n+1}$-field from the $\mathbf{H}^{n}$-field

From Equation 2.8 using central differences, the values of the xyz components of the **E**-field in the next time step can be obtained from the values of the xyz components of the **H**-field in the current time step. (Equations from [5])

For the x component:

$$E_x^{n+1}{}_{(i,j,k)} = \frac{\epsilon^n{}_{(i,j,k)} E_x^n{}_{(i,j,k)}}{\epsilon^{n+1}{}_{(i,j,k)}} + \frac{\Delta t}{\epsilon^{n+1}{}_{(i,j,k)}} \left[\frac{H_z^n{}_{(i,j+1,k)} - H_z^n{}_{(i,j,k)}}{\Delta y} - \frac{H_y^n{}_{(i,j,k+1)} - H_y^n{}_{(i,j,k)}}{\Delta z} \right] \quad [i_{min}+1 \leq i \leq i_{max}, j_{min} \leq j \leq j_{max}, k_{min} \leq k \leq k_{max}]. \tag{B.4}$$

For the y component:

$$E_y^{n+1}{}_{(i,j,k)} = \frac{\epsilon^n{}_{(i,j,k)} E_y^n{}_{(i,j,k)}}{\epsilon^{n+1}{}_{(i,j,k)}} + \frac{\Delta t}{\epsilon^{n+1}{}_{(i,j,k)}} \left[\frac{H_x^n{}_{(i,j,k+1)} - H_x^n{}_{(i,j,k)}}{\Delta z} - \frac{H_z^n{}_{(i+1,j,k)} - H_z^n{}_{(i,j,k)}}{\Delta x} \right] \quad [i_{min} \leq i \leq i_{max}, j_{min}+1 \leq j \leq j_{max}, k_{min} \leq k \leq k_{max}]. \tag{B.5}$$

For the z component:

$$E_z^{n+1}{}_{(i,j,k)} = \frac{\epsilon^n{}_{(i,j,k)} E_z^n{}_{(i,j,k)}}{\epsilon^{n+1}{}_{(i,j,k)}} + \frac{\Delta t}{\epsilon^{n+1}{}_{(i,j,k)}} \left[\frac{H_y^n{}_{(i+1,j,k)} - H_y^n{}_{(i,j,k)}}{\Delta x} - \frac{H_x^n{}_{(i,j+1,k)} - H_x^n{}_{(i,j,k)}}{\Delta y} \right] \quad [i_{min} \leq i \leq i_{max}, j_{min} \leq j \leq j_{max}, k_{min}+1 \leq k \leq k_{max}] \tag{B.6}$$

B.3 Obtention of the $\mathbf{E}^{n}$-field components from the $\mathbf{D}^{n}$-field components

The **D**-field is used in the computational procedure of the FD-FDTD. With the assumption that the environment is source-free, the scalar value of the x,y and z components of the **D**-field of the next iteration $(n+1)$ can be described with eq. B.7, B.8, B.9 using the data from the current iteration (n) (equations from [5]):

$$D_x^{n+1}{}_{(i,j,k)} = \Delta t \left[\frac{H_z^n{}_{(i,j+1,k)} - H_z^n{}_{(i,j,k)}}{\Delta y} - \frac{H_y^n{}_{(i,j,k+1)} - H_y^n{}_{(i,j,k)}}{\Delta z} \right] + D_x^n{}_{(i,j,k)} \quad [i_{min}+1 \leq i \leq i_{max}, j_{min} \leq j \leq j_{max}, k_{min} \leq k \leq k_{max}] \tag{B.7}$$

$$D_y^{n+1}{}_{(i,j,k)} = \Delta t \left[\frac{H_x^n{}_{(i,j,k+1)} - H_x^n{}_{(i,j,k)}}{\Delta z} - \frac{H_z^n{}_{(i+1,j,k)} - H_z^n{}_{(i,j,k)}}{\Delta x} \right] + D_y^n{}_{(i,j,k)} \quad [i_{min} \leq i \leq i_{max}, j_{min}+1 \leq j \leq j_{max}, k_{min} \leq k \leq k_{max}] \tag{B.8}$$

$$D_z^{n+1}{}_{(i,j,k)} = \tag{B.9}$$
$$\Delta t \left[\frac{H_y^n{}_{(i+1,j,k)} - H_y^n{}_{(i,j,k)}}{\Delta x} - \frac{H_x^n{}_{(i,j+1,k)} - H_x^n{}_{(i,j,k)}}{\Delta y} \right] + D_z^n{}_{(i,j,k)}$$
$$[i_{min} \leq i \leq i_{max}, j_{min} \leq j \leq j_{max}, k_{min} + 1 \leq k \leq k_{max}]$$

Appendix C

HOKUSAI GreatWAVE system diagram & connection from Manchester

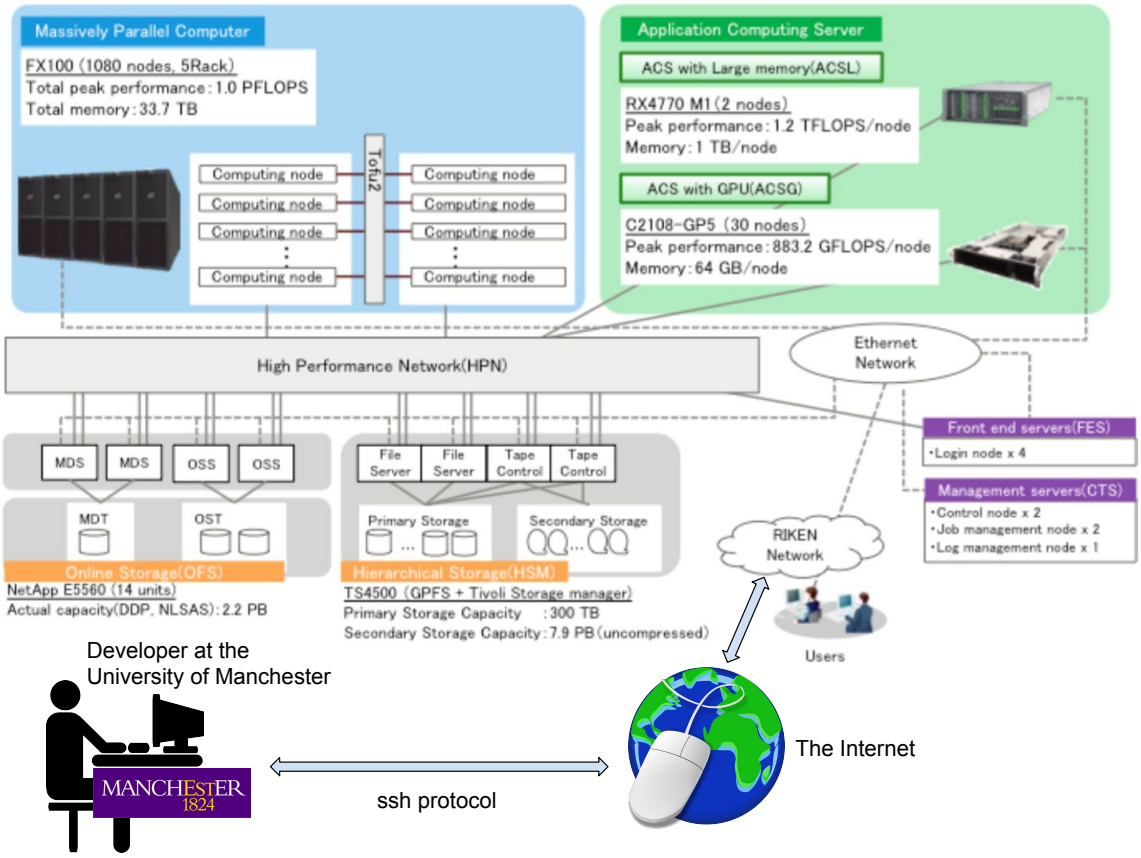


Figure C.1: The RIKEN HOKUSAI GreatWave System diagram (from [14]) and connection from the UK

Appendix D

Project Plan

D.1 Milestones

The major deadlines for this project are:

1. The hand-in of the Progress Report on Friday 6 November 2015, 12pm.
2. The hand-in of the Final Report on 29 April 2016, 12pm
3. The beginning of the oral examination period on 2 May 2016

This means that all practical project work must have finished well before 29 April 2016. The final report shall be finished at least one week before the 29 April 2016, to allow time for proof-reading. All preparation work for the "viva voce", must be finished by 2 May 2016. All these deadlines are represented by a diamond symbol ("milestone") on the Gantt Chart.

D.2 'Float' period

The period 18 December 2015 - 18 January 2016 is a float period. No work is assigned during this period, which coincides with the school's Christmas break. In case there are deviations from the project's plan on 18 December 2015, they can get compensated during the float period that follows.

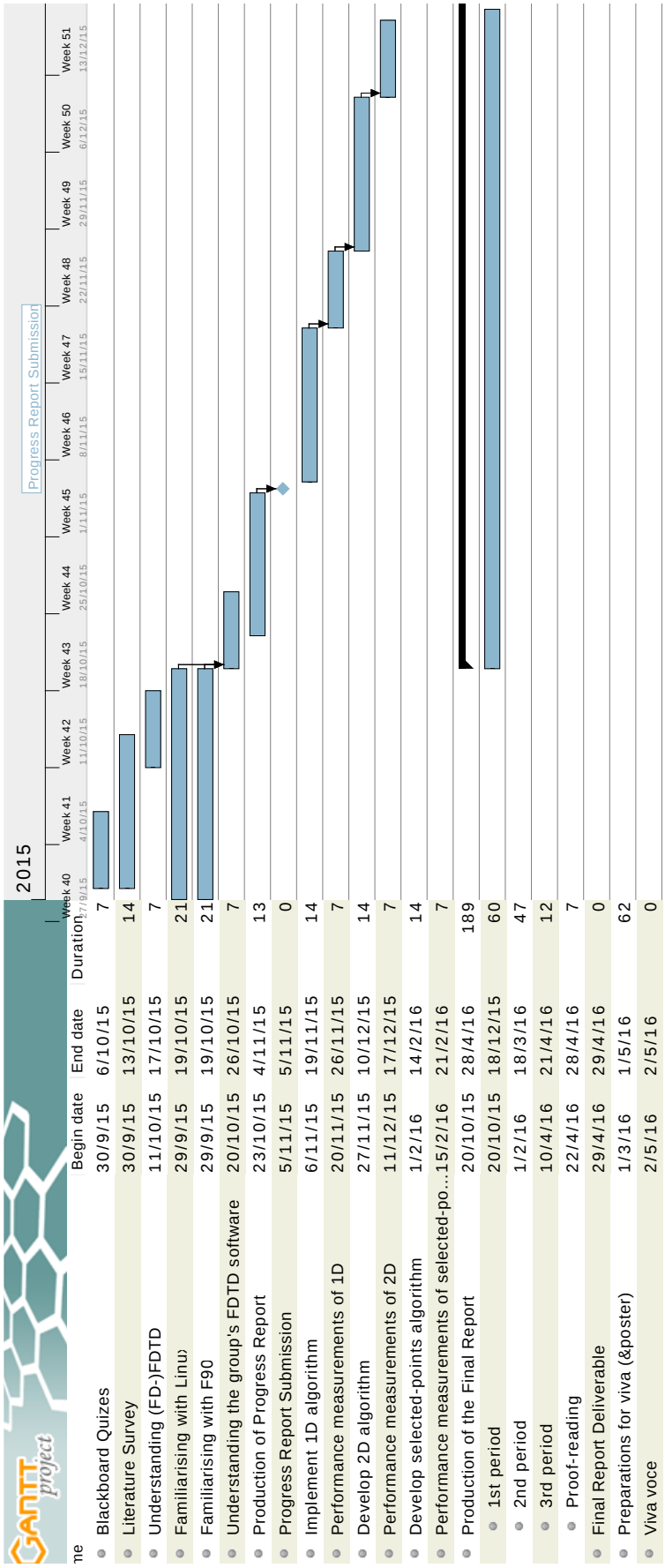
D.3 Notes

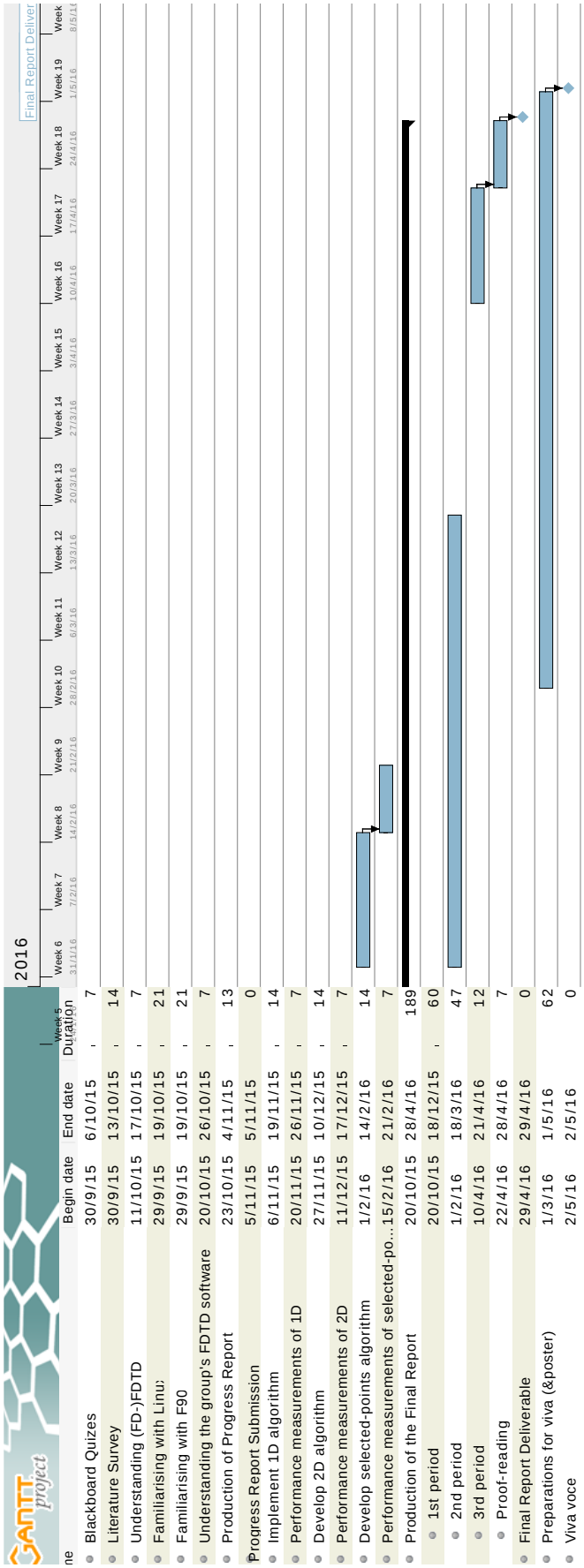
1D algorithm: The algorithm for the speed up of the output of field values of points lying on a single line of the 3D FDTD space.
2D algorithm: The algorithm for the speed up of the output of field values of points lying on a single plane of the 3D FDTD space.
Selected-points algorithm: The algorithm for the speed up of the output of field values of selected points of the 3D FDTD space.

D.4 The list of tasks and the Gantt chart

The 'duration' is measured in days.

Name	Begin date	End date	Duration
Blackboard Quizes	30/9/15	6/10/15	7
Literature Survey	30/9/15	13/10/15	14
Understanding (FD-)FDTD	11/10/15	17/10/15	7
Familiarising with Linux	29/9/15	19/10/15	21
Familiarising with F90	29/9/15	19/10/15	21
Understanding the group's FDTD software	20/10/15	26/10/15	7
Production of Progress Report	23/10/15	4/11/15	13
Progress Report Submission	5/11/15	5/11/15	0
Implement 1D algorithm	6/11/15	19/11/15	14
Performance measurements of 1D	20/11/15	26/11/15	7
Develop 2D algorithm	27/11/15	10/12/15	14
Performance measurements of 2D	11/12/15	17/12/15	7
Develop selected-points algorithm	1/2/16	14/2/16	14
Performance measurements of selected-points algorithm	15/2/16	21/2/16	7
Production of the Final Report	20/10/15	28/4/16	189
1st period	20/10/15	18/12/15	60
2nd period	1/2/16	18/3/16	47
3rd period	10/4/16	21/4/16	12
Proof-reading	22/4/16	28/4/16	7
Final Report Deliverable	29/4/16	29/4/16	0
Preparations for viva (&poster)	1/3/16	1/5/16	62
Viva voce	2/5/16	2/5/16	0


GANTT project
2015
Progress Report Submission
Week 40 7/9/15
Week 41 4/10/15
Week 42 11/10/15
Week 43 18/10/15
Week 44 25/10/15
Week 45 1/11/15
Week 46 8/11/15
Week 47 15/11/15
Week 48 22/11/15
Week 49 29/11/15
Week 50 6/12/15
Week 51 13/12/15
me
Begin date
End date
Duration
Blackboard Quizes 30/9/15 6/10/15 7
Literature Survey 30/9/15 13/10/15 14
Understanding (FD-)FDTD 11/10/15 17/10/15 7
Familiarising with Linux 29/9/15 19/10/15 21
Familiarising with F90 29/9/15 19/10/15 21
Understanding the group's FDTD software 20/10/15 26/10/15 7
Production of Progress Report 23/10/15 4/11/15 13
Progress Report Submission 5/11/15 5/11/15 0
Implement 1D algorithm 6/11/15 19/11/15 14
Performance measurements of 1D 20/11/15 26/11/15 7
Develop 2D algorithm 27/11/15 10/12/15 14
Performance measurements of 2D 11/12/15 17/12/15 7
Develop selected-points algorithm 1/2/16 14/2/16 14
Performance measurements of selected-po... 15/2/16 21/2/16 7
Production of the Final Report 20/10/15 28/4/16 189
1st period 20/10/15 18/12/15 60
2nd period 1/2/16 18/3/16 47
3rd period 10/4/16 21/4/16 12
Proof-reading 22/4/16 28/4/16 7
Final Report Deliverable 29/4/16 29/4/16 0
Preparations for viva (&poster) 1/3/16 1/5/16 62
Viva voce 2/5/16 2/5/16 0


GANTT project
2016
Final Report Deliver
ne
Begin date
End date
Duration
Week 6 31/1/16
Week 7 7/2/16
Week 8 14/2/16
Week 9 21/2/16
Week 10 28/2/16
Week 11 6/3/16
Week 12 13/3/16
Week 13 20/3/16
Week 14 27/3/16
Week 15 3/4/16
Week 16 10/4/16
Week 17 17/4/16
Week 18 24/4/16
Week 19 1/5/16
Blackboard Quizes 30/9/15 6/10/15 7
Literature Survey 30/9/15 13/10/15 14
Understanding (FD-)FDTD 11/10/15 17/10/15 7
Familiarising with Linu: 29/9/15 19/10/15 21
Familiarising with F90 29/9/15 19/10/15 21
Understanding the group's FDTD software 20/10/15 26/10/15 7
Production of Progress Report 23/10/15 4/11/15 13
Progress Report Submission 5/11/15 5/11/15 0
Implement 1D algorithm 6/11/15 19/11/15 14
Performance measurements of 1D 20/11/15 26/11/15 7
Develop 2D algorithm 27/11/15 10/12/15 14
Performance measurements of 2D 11/12/15 17/12/15 7
Develop selected-points algorithm 1/2/16 14/2/16 14
Performance measurements of selected-po... 15/2/16 21/2/16 7
Production of the Final Report 20/10/15 28/4/16 189
1st period 20/10/15 18/12/15 60
2nd period 1/2/16 18/3/16 47
3rd period 10/4/16 21/4/16 12
Proof-reading 22/4/16 28/4/16 7
Final Report Deliverable 29/4/16 29/4/16 0
Preparations for viva (&poster) 1/3/16 1/5/16 62
Viva voce 2/5/16 2/5/16 0

Appendix E

Technical risk analysis

Possible situation	Possible effects	Severity	Solutions applied
Hard disk failure	Loss of data, Loss of project time, Loss of recorded progress	Severe	Daily backups are kept in the group's cluster HDDs and personal backups of important documents are kept on cloud services.
The HOKUSAI GreatWAVE supercomputer cannot accept a job soon because of big job queues	Loss of project time	Moderate	Experimental versions of the project's software are submitted for execution as soon as possible. If multiple ideas are under consideration, all of them are implemented and then submitted even if a change of mind happens afterwards.
The HOKUSAI GreatWAVE supercomputer goes through unplanned maintenance	Loss of project time	Moderate	Any practical project work is carried out before any documentation work
The aims are impossible to achieve at all or in the time available for this project	The aims of the project are never achieved	Severe	In such a case, a thorough analysis of all steps followed and reasons of failure will be carried out. A thorough and meticulous literature review has been carried out, which suggest it is feasible for the project aims to be met in the given amount of time.

Table E.1: Technical Risks

Appendix F

Risk Assessment

MANCHESTER 1824

The University of Manchester

SCHOOL OF E&EE RISK ASSESSMENT

WORK ACTIVITY/ WORKPLACE (WHAT PART OF THE ACTIVITY POSES RISK OF INJURY OR ILLNESS)	HAZARD (S) (SOMETHING THAT COULD CAUSE HARM, ILLNESS OR INJURY)	LIKELY CONSEQUENCES (WHAT WOULD BE THE RESULT OF THE HAZARD)	WHO OR WHAT IS AT RISK (INCLUDE NUMBERS AND GROUPS)	EXISTING CONTROL MEASURES IN USE (WHAT PROTECTS PEOPLE FROM THESE HAZARDS)	WITH EXISTING CONTROLS				MEASURE REQUIRED TO PREVENT OR REDUCE RISK (WHAT NEEDS TO BE DONE TO MAKE THE ACTIVITY AS SAFE AS POSSIBLE)	PERSON RESPONSIBLE FOR ACTIONS AND AGREED TIMESCALES TO ACHIEVE THEM	WITH NEW CONTROLS			
					SEVERITY	LIKELIHOOD	RISK RATING	RISK ACCEPTABLE			SEVERITY	LIKELIHOOD	RISK RATING	RISK ACCEPTABLE
Spend prolonged periods sitting before computer/laptop and dragging a mouse	Repetitive Strain Injury (RSI) and Work Related Upper Limb Disorder (WRULD)	Pain, Numbness, Aching, Tiredness and Tingling in muscles, nerves, tendons and other soft body tissues of the hand, wrist, arm, shoulder and spine	Students who need to program and use computer and laptop in the project	Students are to be provided with copies of the documents Health and Safety Executive publication "INDG36 - Working with VDUs".	2	3	6	NO	Take frequent breaks from the computer Do simple stretching and shaking exercises whilst taking a break from the computer.	Students themselves and supervisor	2	1	2	YES
Focusing the eyes on a computer display for protracted, uninterrupted periods of time	Computer vision syndrome (CVS)	Headaches, Blurred vision, Neck pain, Fatigue, eye strain, Dry, Irritated eyes, Double vision, Polyopia, and Difficulty refocusing the eyes	Students who need to program and use computer and laptop in the project	Approved seating, desks and lighting are provided in laboratory SSB/A 19 Students are to be provided with copies of the following documents: (1) The University's Code of Practice and Guidance Note on "Use of Display Screen Equipment", (2) The University's Guidance Note "Display Screen Equipment / Workstation Set Up"	3	3	9	NO	Take frequent breaks from the computer Blink the eyes consciously every now and then (this helps replenish the tear film) and to look out the window to a distant object or to the sky	Students themselves and supervisor	2	1	2	YES

Activity Location: Barnes Wallis computer cluster or Home

MANAGER/SUPERVISOR	NAME: Dr. Fumie Costen	SIGNED:	DATE: 27/10/2015
STUDENT	NAME: Mr. Loukas Xanthos	SIGNED: Ξανθός Λουκάς	DATE: 27/10/2015

MANCHESTER 1824

The University of Manchester

SCHOOL OF E&EE RISK ASSESSMENT

IF THE ANSWERS TO ANY OF THE QUESTIONS BELOW IS YES THEN ADDITIONAL SPECIFIC RISK ASSESSMENTS MAY BE REQUIRED.

IS THERE A RISK OF FIRE?	Y/N	DOES THE ACTIVITY REQUIRE ANY HOME WORKING?	Y/N
ARE SUBSTANCES THAT ARE HAZARDOUS TO HEALTH USED?	Y/N	ARE THE EMPLOYEES REQUIRED TO WORK ALONE	Y/N
IS THERE MANUAL HANDLING INVOLVED?	Y/N	DOES THE ACTIVITY INVOLVE DRIVING	Y/N
IS PPE WORN OR REQUIRED TO BE WORN?	Y/N	DOES THE ACTIVITY REQUIRE WORK AT HEIGHT	Y/N
ARE DISPLAY SCREENS USED?	Y/N	DOES THE ACTIVITY INVOLVE FOREIGN TRAVEL	Y/N
IS THERE A SIGNIFICANT RISK TO YOUNG PERSONS?	Y/N	IS THERE A SIGNIFICANT RISK TO NEW / PREGNANT MOTHERS?	Y/N

***Severity value* = potential consequence of an incident/injury**

5	Very High	Death / permanent incapacity / widespread loss
4	High	Major Injury (Reportable Category) / Severe Incapacity / Serious Loss
3	Moderate	Injury / illness of 3 days or more absence (reportable category) / Moderate loss
2	Slight	Minor injury / illness – immediate First Aid only / slight loss
1	Negligible	No injury or trivial injury / illness / loss

***Likelihood value* = what is the potential of an incident or injury occurring**

5	Almost certain to occur
4	Likely to occur
3	Quite possible to occur
2	Possible in current situation
1	Not likely to occur

Risk rating* = *severity value* × *likelihood value

***Risk ratings* are classified as *low* (1 – 5), *medium* (6 – 9) and *high* (10 – 25)**

Risk Classification and Actions:

Rating	Classification	Action
1 – 5	Low	Tolerable risk - Monitor and Manage
6 – 9	Medium	Review and introduce additional controls to mitigate to "As Low As Reasonably Practicable" (ALARP)
10 – 25	High	Stop work immediately and introduce further control measures

LIKELIHOOD \ SEVERITY	1	2	3	4	5
1	Low	Low	Low	Low	Low
2	Low	Low	Medium	Medium	High
3	Low	Medium	Medium	High	High
4	Low	Medium	High	High	High
5	Low	High	High	High	High

Appendix B

Pseudocode of algorithms described in this report

The code developed for the project is based on the long and complicated (FD)FDTD software, therefore the author's code work cannot be easily perceived when isolated from the (FD)FDTD software.

Hence, this Appendix contains a pseudocode description of the main methods listed in this report. The segments of pseudocode that concern the application of each method on the output stage of the (FD)FDTD software, must be repeated for the transfer of the data of each field array, at the end of every time sub-step of the main loop of the FD-FDTD method.

B.1 Pseudocode for method NB13X

B.1.1 Application of the method on the output stage

```
host_3D_field_array(1:nx,j0,k0) = device_3D_feld_array(1:nx,j0,
    ↪ k0)
```

B.2 Pseudocode for method NB13Y

B.2.1 Application of the method on the output stage

```
host_3D_field_array(i0,1:ny, k0) = device_3D_feld_array(i0,1:ny,
    ↪ k0)
```

B.3 Pseudocode for method NB13Z

B.3.1 Application of the method on the output stage

```
host_3D_field_array(i0, j0, 1:nz) = device_3D_feld_array(i0, j0,
    ↪ 1:nz)
```

B.4 Pseudocode for method B13X

B.4.1 CUDA Kernel

```
KERNEL x_buffering
   PARAMETERS:[
        INTEGERS: j0, k0, nx,ny,nz,
        REAL arrays in Device memory:
            device_3D_field(1:nx,1:ny,1:nz),
            device_1D_buffer(1:nx) ]

BEGIN

Integer variables i, j, k

  i = id_of_thread.x + (id_of_block.x * 512)
  j = j0
  k = k0

if 1 <= i <= nx
  device_1D_buffer(i) = device_3D_field(i,j,k)
endif

END
```

B.4.2 Application of the method on the output stage

```
launch kernel x_buffering [j0, k0, nx, ny,nz, device_field_array
    ↪ , device_1D_buffer,
blocks:nx/512+1, threads per block: 512 ]
```

```
host_3D_field_array(1:nx,j0,k0) = device_1D_buffer(1:nx)
```

B.5 Pseudocode for method B13Y

B.5.1 CUDA Kernel

```
KERNEL y_buffering
   PARAMETERS:[
        INTEGERS: i0, k0, nx,ny,nz
        REAL arrays in Device memory:
            device_3D_field(1:nx,1:ny,1:nz),
            device_1D_buffer(1:ny) ]

BEGIN

Integer variables i, j, k

  i=i0
  j = id_of_thread.x + (id_of_block.x * 512)
  k = k0

if 1 <= j <= ny
  device_1D_buffer(j) = device_3D_field(i,j,k)
endif

END
```

B.5.2 Application of the method on the output stage

```
launch kernel y_buffering [i0, k0, nx, ny,nz, device_field_array
   ↪ , device_1D_buffer,
blocks:ny/512+1, threads per block: 512 ]

host_3D_field_array(i0, 1:ny, k0) = device_1D_buffer(1:ny)
```

B.6 Pseudocode for method B13Z

B.6.1 CUDA Kernel

```
KERNEL z_buffering
   PARAMETERS:[
        INTEGERS: i0, j0, nx,ny,nz
        REAL arrays in Device memory:
            device_3D_field(1:nx,1:ny,1:nz),
            device_1D_buffer(1:nz) ]

BEGIN

Integer variables i, j, k

  i = i0
  j = j0
 k = id_of_thread.x + (id_of_block.x * 512)

if 1 <= k <= nz
  device_1D_buffer(k) = device_3D_field(i,j,k)
endif

END
```

B.6.2 Application of the method on the output stage

```
launch kernel z_buffering [i0, j0, nx, ny,nz, device_field_array
   ↪ , device_1D_buffer,
blocks: nz/512+1, threads per block: 512 ]

host_3D_field_array(i0, 1:nz, k0) = device_1D_buffer(1:nz)
```

B.7 Pseudocode for method B11X

B.7.1 CUDA Kernel

```
KERNEL x_buffering
   PARAMETERS:[
        INTEGERS: j0, k0, nx,ny,nz
        REAL arrays in Device memory:
            device_3D_field(1:nx,1:ny,1:nz),
            device_1D_buffer(1:nx) ]

BEGIN

Integer variables i, j, k

  i = id_of_thread.x + (id_of_block.x * 512)
  j = j0
  k = k0

if 1 <= i <= nx
  device_1D_buffer(i) = device_3D_field(i,j,k)
endif

END
```

B.7.2 Application of the method on the output stage

```
launch kernel x_buffering [j0, k0, nx, ny,nz, device_field_array
   ↪ , device_1D_buffer,
blocks:nx/512+1, threads per block: 512 ]

host_1D_buffer_array(1:nx) = device_1D_buffer(1:nx)
```

B.8 Pseudocode for method B11Y

B.8.1 CUDA Kernel

```
KERNEL y_buffering
   PARAMETERS:[
        INTEGERS: i0, k0, nx, ny, nz
```

```
        REAL arrays in Device memory:
            device_3D_field(1:nx,1:ny,1:nz),
            device_1D_buffer(1:ny) ]

BEGIN

Integer variables i, j, k

  i = i0
  j = id_of_thread.x + (id_of_block.x * 512)
  k = k0

if 1 <= j <= ny
  device_1D_buffer(j) = device_3D_field(i, j, k)
endif

END
```

B.8.2 Application of the method on the output stage

```
launch kernel y_buffering [i0, k0, nx, ny,nz, device_field_array
  ↪ , device_1D_buffer,
blocks:ny/512+1, threads per block: 512 ]

host_1D_buffer_array(1:ny) = device_1D_buffer(1:ny)
```

B.9 Pseudocode for method B11Z

B.9.1 CUDA Kernel

```
KERNEL z_buffering
   PARAMETERS:[
        INTEGERS: i0, j0, nx,ny,nz
        REAL arrays in Device memory:
            device_3D_field(1:nx,1:ny,1:nz),
            device_1D_buffer(1:nz) ]
```

```
BEGIN

Integer variables i, j, k

  i=i0
  j=j0
  k= id_of_thread.x + (id_of_block.x * 512)

if 1 <= k <= nz
  device_1D_buffer(k) = device_3D_field(i,j,k)
endif

END
```

B.9.2 Application of the method on the output stage

```
launch kernel z_buffering [i0, j0, nx, ny,nz, device_field_array
  ↪ , device_1D_buffer,
blocks:nz/512+1, threads per block: 512 ]

host_1D_buffer_array(1:nz) = device_1D_buffer(1:nz);
```

B.10 Pseudocode for method NB23X

B.10.1 Application of the method on the output stage

```
host_3D_field_array(i0,1:ny, 1:nz) = device_3D_feld_array(i0,1:
  ↪ ny, 1:nz)
```

B.11 Pseudocode for method NB23Y

B.11.1 Application of the method on the output stage

```
host_3D_field_array(1:nx, j0, 1:nz) = device_3D_feld_array(1:nx,
  ↪  j0, 1:nz)
```

B.12 Pseudocode for method NB23Z

B.12.1 Application of the method on the output stage

```
host_3D_field_array(1:nx, 1:ny, k0) = device_3D_feld_array(1:nx,
    ↪ 1:nz, k0)
```

B.13 Pseudocode for method B23X

B.13.1 CUDA Kernel

```
KERNEL Xplane_buffering [
   PARAMETERS:
        INTEGERS: i0, nx, ny, nz
        REAL arrays in Device memory:
            device_3D_field(1:nx,1:ny,1:nz),
            device_2D_buffer(1:nx, 1:ny)]

BEGIN

Integer variables i, j, k

  i = i0
  j = id_of_thread.x
  k = id_of_block.x

if 1 <= j <= ny and 1 <= k <= nz
  device_2D_buffer(i, j) = device_3D_field(i, j, k)
endif

END
```

B.13.2 Application of the method on the output stage

```
launch kernel Zplane_buffering [ k0, nx, ny,nz,
    ↪ device_field_array, device_2D_buffer,
        block dimension y: nz
```

```
        block dimension x: ny]

host_3D_field_array(i0, 1:ny, 1:nz) = device_2D_buffer(1:nx, 1:
    ↪ nz);
```

B.14 Pseudocode for method B23Y

B.14.1 CUDA Kernel

```
KERNEL Yplane_buffering [
    PARAMETERS:
        INTEGERS: j0, nx, ny, nz
        REAL arrays in Device memory:
            device_3D_field(1:nx,1:ny,1:nz),
            device_2D_buffer(1:nx, 1:nz)]

BEGIN

Integer variables i, j, k

  i = id_of_thread.x
  j = j0
  k = id_of_block.x

if 1 <= i <= nx and 1 <= j <= ny
  device_2D_buffer(i, j) = device_3D_field(i, j, k)
endif

END
```

B.14.2 Application of the method on the output stage

```
launch kernel Yplane_buffering [ j0, nx, ny,nz,
    ↪ device_field_array, device_2D_buffer,
        block dimension y: nz ,
        block dimension x: nx ]
```

```
host_3D_field_array(1:nx, j0, 1:nz) = device_2D_buffer(1:nx, 1:
    ↪ nz);
```

B.15 Pseudocode for method B23Z

B.15.1 CUDA Kernel

```
KERNEL Zplane_buffering [
    PARAMETERS:
        INTEGERS:k0, nx,ny,nz
        REAL arrays in Device memory:
            device_3D_field(1:nx,1:ny,1:nz),
            device_2D_buffer(1:nx, 1:ny)]

BEGIN

Integer variables i, j, k

  i = id_of_thread.x
  j = id_of_block.x
  k = k0

if 1 <= i <= nx and 1 <= j <= ny
  device_2D_buffer(i, j) = device_3D_field(i, j, k)
endif

END
```

B.15.2 Application of the method on the output stage

```
launch kernel Zplane_buffering [ k0, nx, ny,nz,
    ↪ device_field_array, device_2D_buffer,
        block dimension y: ny
        block dimension x: nx ]

host_3D_field_array(1:nx, 1:ny, k0) = device_2D_buffer(1:nx, 1:
    ↪ ny);
```

B.16 Pseudocode for method B22X

B.16.1 CUDA Kernel

```
KERNEL Xplane_buffering [
   PARAMETERS:
        INTEGERS: i0, nx, ny, nz
        REAL arrays in Device memory:
            device_3D_field(1:nx,1:ny,1:nz),
            device_2D_buffer(1:nx, 1:ny)]

BEGIN

Integer variables i, j, k

  i = i0
  j = id_of_thread.x
  k = id_of_block.x

if 1 <= j <= ny and 1 <= k <= nz
  device_2D_buffer(i, j) = device_3D_field(i, j, k)
endif

END
```

B.16.2 Application of the method on the output stage

```
launch kernel Zplane_buffering [ k0, nx, ny,nz,
   ↪ device_field_array, device_2D_buffer,
        block dimension y: nz
        block dimension x: ny]

host_2D_buffer_array(1:ny, 1:nz) = device_2D_buffer(1:nx, 1:nz);
```

B.17 Pseudocode for method B22Y

B.17.1 CUDA Kernel

```
KERNEL Yplane_buffering [
   PARAMETERS:
        INTEGERS: j0, nx, ny, nz
        REAL arrays in Device memory:
            device_3D_field(1:nx,1:ny,1:nz),
            device_2D_buffer(1:nx, 1:nz)]

BEGIN

Integer variables i, j, k

  i = id_of_thread.x
  j = j0
  k = id_of_block.x

if 1 <= i <= nx and 1 <= j <= ny
  device_2D_buffer(i, j) = device_3D_field(i, j, k)
endif

END
```

B.17.2 Application of the method on the output stage

```
launch kernel Yplane_buffering [ j0, nx, ny,nz,
   ↪ device_field_array, device_2D_buffer,
        block dimension y: nz ,
        block dimension x: nx ]

host_2D_buffer_array(1:nx, 1:nz) = device_2D_buffer(1:nx, 1:nz);
```

B.18 Pseudocode for method B22Z

B.18.1 CUDA Kernel

```
KERNEL Zplane_buffering [
   PARAMETERS:
```

```
        INTEGERS:k0, nx,ny,nz
        REAL arrays in Device memory:
            device_3D_field(1:nx,1:ny,1:nz),
            device_2D_buffer(1:nx, 1:ny)]

BEGIN

Integer variables i, j, k

  i = id_of_thread.x
  j = id_of_block.x
  k = k0

if 1 <= i <= nx and 1 <= j <= ny
  device_2D_buffer_array(i, j) = device_3D_field(i, j, k)
endif

END
```

B.18.2 Application of the method on the output stage

```
launch kernel Zplane_buffering [ k0, nx, ny,nz,
   ↪ device_field_array, device_2D_buffer,
        block dimension y: ny
        block dimension x: nx ]

host_1D_buffer_array(1:nx, 1:ny) = device_2D_buffer(1:nx, 1:ny);
```

B.19 Pseudocode for method RND-DIRECT

B.19.1 Application of the method on the output stage

```
//Integer M_poi
//Integer poi_lst(1:M_poi, 1:3) resides in host memory and
   ↪ contains the list of coordinates of POI

for i=1,M_poi
```

```
  this_x = poi_lst(i, 1)
  this_k = poi_lst(i, 2)
  this_z = poi_lst(i, 3)
  host_3D_field_array(this_x, this_y, this_z) =
    ↪ device_3D_field_array(this_x, this_y, this_z)
end
```

B.20 Pseudocode for method RND-MAP

B.20.1 CUDA Kernel

```
kernel RNDMAP [
PARAMETERS:
INTEGER: M_poi, nx, ny,nz,
REAL ARRAYS in device: device_poi_map(nx:ny:nz),
  ↪ device_1D_buffer(0:M_poi), device_3D_field_array(1:nx,1:ny
  ↪ ,1:nz) ]
BEGIN
INTEGER variables: i,j,k
REAL arrays in device: device_poi_buffer(0:M_poi)

i = id_of_thread.x
j = id_of_block.x
k = id_of_block.y

if 1 <= k <= nz and 1 <= y <= ny and 1 <= x <= nx
then
   device_poi_buffer( d_poi_buffer(i,j,k) ) =
     ↪ device_3D_field_array(i,j,k)
endif

END
```

B.20.2 Application of the method on the output stage

```
launch kernel RNDMAP [M_poi, nx, ny,nz, device_poi_map,
  ↪ device_1D_buffer, device_3D_field_array
```

```
        grid dimension: nx,
        bock dimension y:nz,
        block dimension x: nx ]

    host_1D_buffer(0:M_poi) = device_1D_buffer(0:M_poi)

/*
//this commented-out segment of pseudocode describes
//how the buffer can be used for the printing
//of the field data. Evidently,  buffer access is performed with
//complexity O(M_poi).
for i=1, M_poi
    x_extracted = poi_lst(i,1)
    y_extracted = poi_lst(i,2)
    z_extracted = poi_lst(i,3)
    print "Point at (" + x_extracted + ", " + y_extracted + ", "
      ↪ + z_extracted + ") has value  + host_1D_buffer(i)
end
*/
```

B.21 Pseudocode for method RND-LST

B.21.1 CUDA Kernel

```
kernel RNDLST [
PARAMETERS:
INTEGER: M_poi, nx, ny,nz,
REAL ARRAYS in device: device_poi_list(1:M_poi),
  ↪ device_1D_buffer(1:M_poi),
          device_3D_field_array(1:nx,1:ny,1:nz) ]
BEGIN
INTEGER variables: i,j,k
REAL arrays in device: device_poi_buffer(0:M_poi)

i = (id_of_block.x-1)*dimension_of_block.x + id_of_thread.x

if 1 <= i <= M_poi
```

```
then
    x_l = device_poi_lst(i,1)
    y_l = device_poi_lst(i,2)
    z_l = device_poi_lst(i,3)
   device_poi_buffer(i) = device_3D_field_array(x_l, y_l, z_l)
endif

END
```

B.21.2 Application of the method on the output stage

```
launch kernel LST [M_poi, nx, ny,nz, device_poi_list,
   ↪ device_1D_buffer, device_3D_field_array
       grid dimension: nx,
       bock dimension y:nz,
       block dimension x: nx ]

    host_1D_buffer(1:M_poi) = device_1D_buffer(1:M_poi)

/*
//this commented-out segment of pseudocode describes
//how the buffer can be used for the printing
//of the field data. Evidently, buffer access is performed with
//complexity O(M_poi).
for i=1, M_poi
    x_extracted = poi_lst(i,1)
    y_extracted = poi_lst(i,2)
    z_extracted = poi_lst(i,3)
    print "Point at (" + x_extracted + ", " + y_extracted + ", "
       ↪ + z_extracted + ") has value  + host_1D_buffer(i)
end
*/
```

Appendix C

The source code of the pseudo-random 3D cartesian coordinates generator in C++11

```
/* AUTHOR: Loukas Xanthos
 * The University of Manchester
 */

#include <iostream>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string>
#include <set>

using namespace std;

#define MAXN 374

typedef unsigned long long int ullint;

class Coordinate {
    public:
        Coordinate(int mx, int my, int mz) {
            x=mx; y=my; z=mz;
        }
```

```
        void get() { cout << x << ' '<< y << ' ' << z << '\n'; }
    private:
        int x,y,z;

    friend bool operator<(const Coordinate& cleft, const
      ↪ Coordinate& cright);
};

bool operator<(const Coordinate& cleft, const Coordinate& cright
  ↪ ) {
    if (cleft.x < cright.x) return true;
    if (cleft.x > cright.x) return false;

    if (cleft.y < cright.y) return true;
    if (cleft.y > cright.y ) return false;

    if (cleft.z < cright.z) return true;

    return false;
}

typedef std::set<Coordinate> setOfCoords;

bool CheckForDuplicate(const Coordinate& tstcoord) {
    static setOfCoords usedsetOfCoords;

    // Found in the set?
    if (usedsetOfCoords.find(tstcoord) != usedsetOfCoords.end())
      ↪ {
        return false; //yes-fail.
    }

    // no, it is unique
    usedsetOfCoords.insert(coordinate);

    return true;

}
```

```
int main (int argc, char *argv[] )
{
    int rndx, rndy, rndz;
    std::string str;
    std::string::size_type sz = 0;
    ullint M_poi;

    if ( argc != 2 ) {// argc should be 2 for correct execution
            // We print argv[0] assuming it is the program name
        cout<<"usage: "<< argv[0] <<" <M_poi>\n";
        exit(0);
    }
    str = argv[1];
    M_poi = stoull(str,&sz);

    /* initialize random seed: */
    srand (time(NULL));

    for(ullint i=0; i<M_poi; i++){
        do{
        rndx = rand() % MAXN + 1;
        rndy = rand() % MAXN + 1;
        rndz = rand() % MAXN + 1;
    } while(!CheckForDuplicate(Coordinate(rndx,rndy,rndz)));
        cout << rndx << ' ' << rndy << ' ' << rndz << '\n';
    }

    return 0;
}
```

Appendix D

Technical risk analysis

Possible situation	Possible effects	Severity	Solutions applied
Hard disk failure	Loss of data, Loss of project time, Loss of recorded progress	Severe	Daily backups are kept in the group's cluster HDDs and personal backups of important documents are kept on cloud services.
The HOKUSAI GreatWAVE supercomputer cannot accept a job soon because of big job queues	Loss of project time	Moderate	Experimental versions of the project's software are submitted for execution as soon as possible. If multiple ideas are under consideration, all of them are implemented and then submitted even if a change of mind happens afterwards.
The HOKUSAI GreatWAVE supercomputer goes through unplanned maintenance	Loss of project time	Moderate	Any practical project work is carried out before any documentation work
The aims are impossible to achieve at all or in the time available for this project	The aims of the project are never achieved	Severe	In such a case, a thorough analysis of all steps followed and reasons of failure will be carried out. A thorough and meticulous literature review has been carried out, which suggest it is feasible for the project aims to be met in the given amount of time.

Table D.1: Technical Risks

Appendix E

Risk Assessment

MANCHESTER 1824

The University of Manchester

SCHOOL OF E&EE RISK ASSESSMENT

WORK ACTIVITY/ WORKPLACE (WHAT PART OF THE ACTIVITY POSES RISK OF INJURY OR ILLNESS)	HAZARD (S) (SOMETHING THAT COULD CAUSE HARM, ILLNESS OR INJURY)	LIKELY CONSEQUENCES (WHAT WOULD BE THE RESULT OF THE HAZARD)	WHO OR WHAT IS AT RISK (INCLUDE NUMBERS AND GROUPS)	EXISTING CONTROL MEASURES IN USE (WHAT PROTECTS PEOPLE FROM THESE HAZARDS)	WITH EXISTING CONTROLS				MEASURE REQUIRED TO PREVENT OR REDUCE RISK (WHAT NEEDS TO BE DONE TO MAKE THE ACTIVITY AS SAFE AS POSSIBLE)	PERSON RESPONSIBLE FOR ACTIONS AND AGREED TIMESCALES TO ACHIEVE THEM	WITH NEW CONTROLS			
					SEVERITY	LIKELIHOOD	RISK RATING	RISK ACCEPTABLE			SEVERITY	LIKELIHOOD	RISK RATING	RISK ACCEPTABLE
Spend prolonged periods sitting before computer/laptop and dragging a mouse	Repetitive Strain Injury (RSI) and Work Related Upper Limb Disorder (WRULD)	Pain, Numbness, Aching, Tiredness and Tingling in muscles, nerves, tendons and other soft body tissues of the hand, wrist, arm, shoulder and spine	Students who need to program and use computer and laptop in the project	Students are to be provided with copies of the documents Health and Safety Executive publication "INDG36 - Working with VDUs".	2	3	6	NO	Take frequent breaks from the computer Do simple stretching and shaking exercises whilst taking a break from the computer.	Students themselves and supervisor	2	1	2	YES
Focusing the eyes on a computer display for protracted, uninterrupted periods of time	Computer vision syndrome (CVS)	Headaches, Blurred vision, Neck pain, Fatigue, eye strain, Dry, Irritated eyes, Double vision, Polyopia, and Difficulty refocusing the eyes	Students who need to program and use computer and laptop in the project	Approved seating, desks and lighting are provided in laboratory SSB/A 19 Students are to be provided with copies of the following documents: (1) The University's Code of Practice and Guidance Note on "Use of Display Screen Equipment", (2) The University's Guidance Note "Display Screen Equipment / Workstation Set Up"	3	3	9	NO	Take frequent breaks from the computer Blink the eyes consciously every now and then (this helps replenish the tear film) and to look out the window to a distant object or to the sky	Students themselves and supervisor	2	1	2	YES

Activity Location: Barnes Wallis computer cluster or Home

MANAGER/SUPERVISOR	NAME: Dr. Fumie Costen	SIGNED:	DATE: 27/10/2015
STUDENT	NAME: Mr. Loukas Xanthos	SIGNED: Ξανθός Λουκάς	DATE: 27/10/2015

MANCHESTER 1824

The University of Manchester

SCHOOL OF E&EE RISK ASSESSMENT

IF THE ANSWERS TO ANY OF THE QUESTIONS BELOW IS YES THEN ADDITIONAL SPECIFIC RISK ASSESSMENTS MAY BE REQUIRED.

IS THERE A RISK OF FIRE?	Y/N	DOES THE ACTIVITY REQUIRE ANY HOME WORKING?	Y/N
ARE SUBSTANCES THAT ARE HAZARDOUS TO HEALTH USED?	Y/N	ARE THE EMPLOYEES REQUIRED TO WORK ALONE	Y/N
IS THERE MANUAL HANDLING INVOLVED?	Y/N	DOES THE ACTIVITY INVOLVE DRIVING	Y/N
IS PPE WORN OR REQUIRED TO BE WORN?	Y/N	DOES THE ACTIVITY REQUIRE WORK AT HEIGHT	Y/N
ARE DISPLAY SCREENS USED?	Y/N	DOES THE ACTIVITY INVOLVE FOREIGN TRAVEL	Y/N
IS THERE A SIGNIFICANT RISK TO YOUNG PERSONS?	Y/N	IS THERE A SIGNIFICANT RISK TO NEW / PREGNANT MOTHERS?	Y/N

***Severity value* = potential consequence of an incident/injury**

5	Very High	Death / permanent incapacity / widespread loss
4	High	Major Injury (Reportable Category) / Severe Incapacity / Serious Loss
3	Moderate	Injury / illness of 3 days or more absence (reportable category) / Moderate loss
2	Slight	Minor injury / illness – immediate First Aid only / slight loss
1	Negligible	No injury or trivial injury / illness / loss

***Likelihood value* = what is the potential of an incident or injury occurring**

5	Almost certain to occur
4	Likely to occur
3	Quite possible to occur
2	Possible in current situation
1	Not likely to occur

Risk rating* = *severity value* × *likelihood value

***Risk ratings* are classified as *low* (1 – 5), *medium* (6 – 9) and *high* (10 – 25)**

Risk Classification and Actions:

Rating	Classification	Action
1 – 5	Low	Tolerable risk - Monitor and Manage
6 – 9	Medium	Review and introduce additional controls to mitigate to "As Low As Reasonably Practicable" (ALARP)
10 – 25	High	Stop work immediately and introduce further control measures

LIKELIHOOD \ SEVERITY	1	2	3	4	5
1	Low	Low	Low	Low	Low
2	Low	Low	Medium	Medium	High
3	Low	Medium	Medium	High	High
4	Low	Medium	High	High	High
5	Low	High	High	High	High