Dispersive Möbius Transform Finite Element Time Domain Method on Graphics Processing Units

David Abraham

Department of Electrical and Computer Engineering

McGill University, Montréal

June 2015

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Electrical Engineering

Acknowledgements

There are many people to whom I owe a great debt of gratitude for their help and support over the past two years, without which the present work could not have taken form. Firstly, I would like to thank my supervisor, Prof. Dennis Giannacopoulos, for his insight, expertise, kindness and guidance throughout the entirety of my time at McGill. His dedication to his students, teaching and passion for our field of research are the embodiment of a great educator and researcher. I would also like to thank Mr. Ali Akbarzadeh-Sharbaf and Dr. Yousef El-Kurdi for their invaluable expertise, patience and suggestions in answering my many questions.

My most heartfelt thanks go to my wonderful family, for their unfailing love and support, for whom I dedicate this thesis. I could not be where I am today without their unwavering faith in me, for which I am eternally grateful. Additionally, I would like to sincerely and truly thank them, as well as Ms. Kirsten Gust, for being my motivation, my inspiration, my encouragement and, at times, my sanity.

I would also like to acknowledge the generous support I received from the Fonds de Recherche du Québec – Nature et Technologies (FRQNT), as well as McGill University's Faculty of Engineering. Some computations in this thesis were made on the supercomputer Guillimin from McGill University, managed by Calcul Québec and Compute Canada. The operation of this supercomputer is funded by the CFI, NanoQuébec, RMGA and the FRQNT, for which I am also grateful.

Abstract

Many naturally occurring materials exhibit non-negligible frequency dependent (or dispersive) behaviour when interacting with electromagnetic fields, necessitating a more accurate and specialized treatment during numerical modelling and simulation. As a result, over the past several years many extensions to standard numerical techniques such as the finite element time domain (FETD) method have been proposed, in order to accommodate and model dispersive phenomena. Among them, those based upon the Möbius z-transform technique are, in general, more accurate and more versatile. However, despite increases in efficiency, dispersive FETD simulations have unfortunately remained slower than their traditional non-dispersive counterparts, owing to the inevitable additional overhead and complexity inherent to these media. For many problems, especially those that are large, complex and contain high order dispersive materials, this additional overhead can prove debilitating.

The goal of this thesis, therefore, is to seek out ways to narrow the performance gap which currently exists between dispersive and traditional FETD computations, by investigating the use of Graphics Processing Units (GPUs) and their massively parallel architectures. By analyzing the z-transform algorithm, sections of the dispersive code amenable to parallelization are identified and adapted to NVIDIA's Compute Unified Device Architecture (CUDA) GPU language. Numerical studies are then undertaken to measure performance increase as a function of simulation parameters, such as number of variables, amount of dispersive material present and order of dispersion.

Results indicate significant performance gain in most cases, with large majoritarily dispersive problems seeing the most improvement.

Résumé

Plusieurs matériaux d'origine naturelle démontrent une dépendance de fréquence de leurs propriétés lorsqu'ils interagissent avec les champs électromagnétiques (dispersion), ce qui nécessite un traitement plus précis et spécialisé lors de la modélisation numérique et de la simulation. En conséquence, au cours des dernières années, de nombreuses extensions à des techniques numériques standards, tels que la méthode des éléments finis dans le domaine temporel (FETD), ont été proposées, afin d'accueillir et de modéliser les phénomènes dispersifs. Parmi eux, ceux basés sur la technique de la transformée en Z Möbius sont, en général, plus précis et polyvalent. Cependant, malgré une augmentation en efficacité, les simulations FETD dispersives restent malheureusement toujours plus lentes que leurs homologues non-dispersifs, à cause de la surcharge et la complexité inhérente à ces médias. Pour plusieurs problèmes, particulièrement ceux qui sont à grand échelle, complexes et contiennent des matériaux dispersifs d'ordre élevé, ces surcharges peuvent s'avérer débilitante.

L'objectif de cette thèse est donc de retrouver une manière à réduire l'écart de performance qui existe présentement entre les calculs FETD dispersifs et traditionnels, en enquêtant l'utilisation des processeurs graphiques (GPUs) et leurs architectures massivement parallélisé. En analysant l'algorithme de la transformée en Z, les sections dispersives du code disposé à la parallélisation sont identifiés et adapté au langage de programmation de NVIDIA, la Compute Unified Device Architecture (CUDA). Des études numériques sont ensuite entreprises pour mesurer l'amélioration de la performance en fonction des paramètres de simulation, comme le nombre d'inconnue, la quantité d'élément dispersif et l'ordre de dispersion.

Les résultats indiquent une amélioration de performance significative dans la majorité des cas, avec les problèmes à grand éhelle et majoritairement dispersifs éprouvant les meilleurs résultats.

Table of Contents

List of Figures

Chapter 1: Introduction

Given the ubiquitous nature of electromagnetic devices in today's world, the need for ever more efficient and accurate design and simulation tools is paramount. With the immense complexity inherent to most problems in electromagnetics, closed form exact solutions are often unrealistic or impossible to obtain, leaving numerical analyses as the only viable option to predict and verify the behaviour of a given physical system. As such, many methods and techniques have been devised over the years in order to treat the myriad of problems and conditions posed by electromagnetic phenomena. Approaches such as the Finite Difference (FD) method attempt to discretize the associated differential operators [1], while Finite Element Methods (FEM) focus upon a discretization of the problem domain [2].

In many cases, simplifying assumptions can be made to reduce the complexity of the differential equations, such as symmetry, time independence or the presence of only a single frequency (time-harmonic form) [3]. For certain problems however, such simplifications may not be possible, or cannot be made without potentially unacceptable losses in accuracy. Fortunately, over the past decades, several methods have been devised which allow for a full treatment of vector electromagnetic wave phenomena in the time domain. One technique in particular, the Finite Element Time Domain (FETD) method, has enjoyed much success for its generality, robustness and accuracy [2]. But while the standard FETD method is quite adept and capable at handling fully time dependent electromagnetics problems, it is unfortunately restricted to non-dispersive materials, that is, materials whose properties do not depend upon the frequency of the electromagnetic fields with which they interact.

Dispersion is a fundamental property inherent to a vast array of materials, with deep ties to essential physical principals such as causality and energy absorption [4]. However, while all real world materials exhibit some kind of dispersion, most often the effect is small enough as to be neglected over a small frequency range of interest, allowing for an excellent treatment via the standard FETD formulation. Nonetheless, such simplifications are not always possible, particularly when dealing with wideband electromagnetic simulations or for problems in which a higher degree of accuracy is required [2]. In such circumstances, a proper dispersive time-domain treatment is required in order to capture the full response of the system. Such methods have far

reaching applications, from the study of dispersive human tissue samples in medical imaging, to dispersive environmental elements in radar applications [5], [6].

Given the clear need for dispersive time domain solvers, several extensions to the FETD method have been proposed, being broadly characterized into three distinct groups: Recursive Convolution (RC), Auxiliary Differential Equation (ADE) and z-transform. As will be demonstrated in Chapter 2, the introduction of dispersion necessitates the appearance of convolutions within the vector wave equation. The RC method attempts to deal with these directly through a clever simplification of the convolution integrals [7], while the ADE method derives a separate defining system of PDEs for the permittivity and permeability [8]. While both methods have achieved notable success, treatment of arbitrarily high order dispersive phenomena can rapidly become intractable. Methods based upon the z-transform, on the other hand, convert material parameters to the z-domain, allowing for the derivation of update equations which remain relatively simple and efficient at arbitrarily high dispersive orders [9], [10].

Regardless of the technique chosen to address dispersion within an electromagnetic system, additional overhead as compared to non-dispersive methods cannot be avoided. As a consequence, dispersive methods are in general much slower and more computationally demanding. The goal of the present work, therefore, is to narrow the performance gap which currently exists between traditional and z-transform dispersive FETD methods, by utilizing the immense computational power afforded by modern Graphics Processing Units (GPUs). Having arisen out of the need to render independent pixels in computer graphics, GPUs are endowed with massively parallel architectures allowing them to perform a tremendous amount of computations concurrently, as opposed to the traditional serial execution of standard processors. By performing an analysis of the z-transform FETD algorithm, computationally intensive overhead amenable to parallelization can be identified and transferred to the GPU to be distributed over multiple workers simultaneously, potentially yielding a significant performance boost. While research has already been conducted into GPU acceleration of dispersive Finite Difference Time Domain (FDTD) methods [11] as well as the base FETD algorithm [12], little if any work has been devoted to addressing the disparity in performance between the latter and dispersive FETD methods. The results obtained herein could therefore further be coupled to these existing parallelizations of the base scheme, for an even greater increase in performance. In this way, parallelization may lead the

way to ever more detailed and complex investigations of frequency dependent phenomena, without the debilitating overhead.

The following document is hereafter divided into six principal chapters, commencing with a review of the standard and dispersive FETD methods. This is followed by an overview of GPUs, as well as an in-depth discussion of parallelization strategy and implementation. Detailed results are then presented and discussed, demonstrating marked improvement in algorithm performance. Lastly, to conclude, a brief summary is presented with emphasis on future work.

Chapter 2: Background

2.1 The Finite Element Time Domain Method

A brief overview of the FETD method based upon the theory presented by Jin in [2] is now presented, as it pertains to non-dispersive, linear, isotropic media. To begin, the second order curl-curl vector wave equation in non-conducting media, defined in some volume V in terms of the electric field strength, $\vec{E}(\vec{r},t)$, is presented as follows:

$$\nabla \times \left[\frac{1}{\mu}\nabla \times \vec{E}(\vec{r},t)\right] + \epsilon \frac{\partial^2 \vec{E}(\vec{r},t)}{\partial t^2} = -\frac{\partial \vec{J}(\vec{r},t)}{\partial t} \quad \vec{r} \in V \tag{2.1}$$

in which μ and ϵ are, respectively, the permeability and the permittivity, and $\vec{J}(\vec{r},t)$ is the source electric current density in space and time. Since the present case is a non-dispersive treatment, the permeability and permittivity are not functions of time, but can vary spatially over the problem domain. In order to obtain a unique solution over this volume, the imposition of boundary conditions along the surface S bounding V are also required. In addition to the possibility of Dirichlet boundaries, a mixed condition on the surface is assumed of the following form:

$$\hat{n} \times \left[\frac{1}{\mu}\nabla \times \vec{E}(\vec{r},t)\right] + Y\frac{\partial}{\partial t}\left[\hat{n} \times \hat{n} \times \vec{E}(\vec{r},t)\right] = \vec{U}(\vec{r},t) \quad \vec{r} \in S \tag{2.2}$$

where $\hat{n}$ is an outward pointing unit vector to the boundary surface, Y is the surface admittance of the boundary and $\vec{U}(\vec{r},t)$ is a known boundary source quantity.

To convert the above wave equation into a finite element problem, equations (2.1) and (2.2) are recast into a weak-form via a scalar product with a set of vector test functions, $\vec{N}_j(\vec{r})$, and integration. After an application of the divergence theorem, the following is obtained:

$$\begin{aligned}\iiint_V \Bigg\{ & \frac{1}{\mu}\left[\nabla \times \vec{N}_i(\vec{r})\right] \cdot \left[\nabla \times \vec{E}(\vec{r},t)\right] + \epsilon \vec{N}_i(\vec{r}) \cdot \frac{\partial^2 \vec{E}(\vec{r},t)}{\partial t^2} \\ & + \vec{N}_i(\vec{r}) \cdot \frac{\partial \vec{J}(\vec{r},t)}{\partial t} \Bigg\} \mathrm{d}V + \iint_S \Big\{ Y\left[\hat{n} \times \vec{N}_i(\vec{r})\right] \\ & \cdot \frac{\partial}{\partial t}\left[\hat{n} \times \vec{E}(\vec{r},t)\right] + \vec{N}_i(\vec{r}) \cdot \vec{U}(\vec{r},t) \Big\} \mathrm{d}S = 0\end{aligned} \tag{2.3}$$

The exact solution to equations (2.1) and (2.2) would satisfy the above for any arbitrary set of functions $\vec{N}_j(\vec{r})$. However, part of solving the problem in a weak-form implies choosing a particular set and satisfying the integrals with respect to that specific choice.

In order to proceed, the problem domain is now subdivided into elements (in this case triangular), each of which is constrained by equation (2.3). The unknown solution within each element is approximately expressed as a combination of known basis functions, selected to ensure adherence to continuity conditions (among others) between elements. Adopting a Galerkin procedure, the testing functions in equation (2.3) are chosen to correspond exactly with these basis functions, yielding the following expansion within each element:

$$\vec{E}(\vec{r},t) = \sum_{j=1}^{N} u_j(t)\vec{N}_j(\vec{r}) \tag{2.4}$$

Given that the basis functions are known, substitution of (2.4) into (2.3) allows for the evaluation of the volume and surface integrals within each element, producing a system of differential equations in the unknown weights $u_j(t)$. In the 1st order linear case, these elemental weights represent the tangential electric field upon the j^{th} edge. The matrices associated with each individual element are then embedded and summed into global matrices according to their connectivity, resulting in the following global system:

$$[T]\frac{d^2\{u\}}{dt^2} + [Q]\frac{d\{u\}}{dt} + [S]\{u\} + \{f\} = \{0\} \tag{2.5}$$

Throughout this text, braces as well as over-arrows denote vector quantities, with the former being generally associated with discretized version of field quantities, and the latter the original continuous fields. Additionally, square brackets denote square matrices and Latin subscripts occurring outside braces and brackets represent vector or matrix components. With this in mind, each of the terms in (2.5) is defined as follows for the k^{th} element:

$$[T_k]_{ij} = \iiint_{V_e} \epsilon_k \vec{N}_i(\vec{r}) \cdot \vec{N}_j(\vec{r}) \mathrm{d}V \tag{2.6}$$

$$[Q_k]_{ij} = \iint_{S_e} Y_k\left[\hat{n} \times \vec{N}_i(\vec{r})\right] \cdot \left[\hat{n} \times \vec{N}_j(\vec{r})\right] \mathrm{d}S \tag{2.7}$$

$$[S_k]_{ij} = \iiint_{V_e} \frac{1}{\mu_k}\left[\nabla \times \vec{N}_i(\vec{r})\right] \cdot \left[\nabla \times \vec{N}_j(\vec{r})\right] \mathrm{d}V \tag{2.8}$$

$$\{f_k\}_i = \iiint_{V_e} \vec{N}_i(\vec{r}) \cdot \frac{\partial \vec{J}_k(\vec{r},t)}{\partial t} \mathrm{d}V + \iint_{S_e} \vec{N}_i(\vec{r}) \cdot \vec{U}_k(\vec{r},t) \mathrm{d}S \tag{2.9}$$

and are related to the global matrices, as mentioned, via summation over all elements:

$$[T] = \sum_{k=0}^{N_{elem}} [T_k] \tag{2.10}$$

With the spatial dependence now encapsulated within the matrices of equation (2.5), the original problem has now been converted into a system of ordinary differential equations in time, which may be solved using any number of standard temporal discretizations. Here, the well-established Newmark-β method [2] is adopted, in which derivative information is approximated through appropriately selected linear combinations of function values in time. Selecting $\beta = \frac{1}{4}$, this yields the following unconditionally stable implicit update equation for the electric field, in which superscripts indicate the discrete temporal step number:

$$\begin{aligned}\left\{\frac{1}{\Delta t^2}[T] + \frac{1}{2\Delta t}[Q] + \frac{1}{4}[S]\right\}\{u\}^{n+1} &= \left\{\frac{2}{\Delta t^2}[T] - \frac{1}{2}[S]\right\}\{u\}^{n} \\ &- \left\{\frac{1}{\Delta t^2}[T] - \frac{1}{2\Delta t}[Q] + \frac{1}{4}[S]\right\}\{u\}^{n-1} \\ &- \left\{\frac{1}{4}\{f\}^{n+1} + \frac{1}{2}\{f\}^{n} + \frac{1}{4}\{f\}^{n-1}\right\}\end{aligned} \tag{2.11}$$

With the above, the solution may be marched forward in time in increments of Δt, solving the requisite matrix problem upon each iteration.

2.2 Effect of Dispersion

Desiring now to include the effects of dispersion in the above formulation, the material parameters μ and ϵ must depend explicitly upon the angular frequency ω. What amounts to a simple product in the frequency domain is now cast into the form of a convolution within the time-

domain, implying a causal temporal dependence of the permeability and permittivity. In other words, the current response of the material depends not only upon its characteristics, but also its past history of interaction with the fields [4]. As such, the governing vector wave equation (2.1) must be altered to include convolutions [10], denoted by $*$:

$$\nabla \times \left[\frac{1}{\mu(t)} * \nabla \times \vec{E}(\vec{r},t)\right] + \epsilon(t) * \frac{\partial^2 \vec{E}(\vec{r},t)}{\partial t^2} = -\frac{\partial \vec{J}(\vec{r},t)}{\partial t} \tag{2.12}$$

and, correspondingly, the weak form is also modified:

$$\begin{aligned} \iiint_V \Bigg\{ &\left[\nabla \times \vec{N}_i(\vec{r})\right] \cdot \left[\frac{1}{\mu(t)} * \nabla \times \vec{E}(\vec{r},t)\right] + \epsilon(t) * \vec{N}_i(\vec{r}) \\ &\cdot \frac{\partial^2 \vec{E}(\vec{r},t)}{\partial t^2} + \vec{N}_i(\vec{r}) \cdot \frac{\partial \vec{J}(\vec{r},t)}{\partial t} \Bigg\} \mathrm{d}V + \iint_S \Big\{ Y\left[\hat{n} \times \vec{N}_i(\vec{r})\right] \\ &\cdot \frac{\partial}{\partial t}\left[\hat{n} \times \vec{E}(\vec{r},t)\right] + \vec{N}_i(\vec{r}) \cdot \vec{U}(\vec{r},t) \Big\} \mathrm{d}S = 0 \end{aligned} \tag{2.13}$$

Following the same procedure as before, this can then be reduced to a system of differential equations in time:

$$[T]\epsilon(t) * \frac{d^2\{u\}}{dt^2} + [Q]\frac{d\{u\}}{dt} + [S]\frac{1}{\mu(t)} * \{u\} + \{f\} = \{0\} \tag{2.14}$$

in which the global matrices are once again built up through the embedding and summation of each local elemental matrix. The expressions defining $[T_k]$, $[Q_k]$, $[S_k]$ and $\{f_k\}$ remain unchanged from (2.6) - (2.9), with the exception that they no longer contain the material parameters μ and ϵ. It is at this stage that a method to accurately compute the required convolutions must be devised.

2.3 The Möbius Z-Transform Method

The vast majority of dispersive materials commonly encountered have parameters which can be easily written in a general form as the quotient of ω dependent polynomials in Fourier space [2]. From here, it is relatively straightforward to apply the transformation $s = j\omega$ and convert the expressions within each element into the Laplace domain:

$$\epsilon_k(s) = \frac{\sum_{n=0}^{p} a_{n,k} s^n}{\sum_{n=0}^{p} b_{n,k} s^n} \quad \mu_k^{-1}(s) = \frac{\sum_{n=0}^{p} m_{n,k} s^n}{\sum_{n=0}^{p} h_{n,k} s^n} \tag{2.15}$$

in which p represents the order of dispersion, and $a_{n,k}, b_{n,k}, m_{n,k}$ and $h_{n,k}$ are constants associated with the material's dispersive model. With the material dispersive models now in the Laplace domain, a brief summary of the z-transform FETD algorithm derived by Akbarzadeh-Sharbaf and Giannacopoulos in [10] is now presented. The key step involves the application of a bilinear transformation (a special case of the Möbius transform [13]) of the following form:

$$s \mapsto \frac{2}{\Delta t} \frac{1 - z^{-1}}{1 + z^{-1}} \tag{2.16}$$

in which Δt is the discrete time step used in our Newmark-β scheme. This complex valued transformation thereby converts the expression a second time, mapping the s-domain into the z-domain. Doing so allows for the permittivity and permeability to be expressed in the z-domain as rational functions in z^{-1}:

$$\epsilon_k(z) = \frac{c_{0,k} + c_{1,k} z^{-1} + \cdots + c_{p,k} z^{-p}}{1 + d_{1,k} z^{-1} + \cdots + d_{p,k} z^{-p}} \quad \mu_k^{-1}(z) = \frac{q_{0,k} + q_{1,k} z^{-1} + \cdots + q_{p,k} z^{-p}}{1 + r_{1,k} z^{-1} + \cdots + r_{p,k} z^{-p}} \tag{2.17}$$

The advantages in having transformed these quantities into the z-domain are two-fold. Firstly, in the z-domain, convolutions between functions in time can be simply recast as products between the individual transforms:

$$\epsilon_k(t) * \{u\}(t) \Leftrightarrow \epsilon_k(z)\{u\}(z) \quad \mu_k^{-1}(t) * \{u\}(t) \Leftrightarrow \mu_k^{-1}(z)\{u\}(z) \tag{2.18}$$

and secondly, within the z-domain, multiplication of a transform by z^{-k} corresponds to a discrete shift in time of $k\Delta t$ [14]:

$$z^{-k}\{u\}(z) \Leftrightarrow \{u\}(n-k) = \{u\}^{n-k}. \tag{2.19}$$

In order to use these properties, it is noted that a temporal discretization of the derivatives in (2.14) will lead to an update formula similar to (2.11), requiring in addition however temporal discretizations of terms of the form $\epsilon_k(t) * [T_k]\{u\}(t)$ and $\mu_k^{-1}(t) * [S_k]\{u\}(t)$, denoted by

$\{\mathcal{L}_{\epsilon,k}\}(t)$ and $\{\mathcal{L}_{\mu^{-1},k}\}(t)$ respectively, within each element. Making use of the above, these update equations are now derived for $\{\mathcal{L}_{\epsilon,k}\}(t)$, with those for $\{\mathcal{L}_{\mu^{-1},k}\}(t)$ being obtained similarly.

To begin, all quantities are transformed into the z-domain utilizing (2.18):

$$\{\mathcal{L}_{\epsilon,k}\}(z) = \frac{c_{0,k} + c_{1,k}z^{-1} + \cdots + c_{p,k}z^{-p}}{1 + d_{1,k}z^{-1} + \cdots + d_{p,k}z^{-p}}[T_k]\{u\}(z) \tag{2.20}$$

Both sides are subsequently multiplied by the denominator and the first term isolated:

$$\begin{aligned}\{\mathcal{L}_{\epsilon,k}\} = \left(c_{0,k} + c_{1,k}z^{-1} + \cdots + c_{p,k}z^{-p}\right)[T_k]\{u\} \\ - \{\mathcal{L}_{\epsilon,k}\}\left(d_{1,k}z^{-1} + \cdots + d_{p,k}z^{-p}\right)\end{aligned} \tag{2.21}$$

Lastly, converting back to the time domain via equation (2.19) yields the desired result:

$$\begin{aligned}\{\mathcal{L}_{\epsilon,k}\}^n = [T_k]\left(c_{0,k}\{u\}^n + c_{1,k}\{u\}^{n-1} + \cdots + c_{p,k}\{u\}^{n-p}\right) \\ - d_{1,k}\{\mathcal{L}_{\epsilon,k}\}^{n-1} - \cdots - d_{p,k}\{\mathcal{L}_{\epsilon,k}\}^{n-p}\end{aligned} \tag{2.22}$$

Hence, the next value of the convolution can be computed through knowledge of past field and convolution values, with the amount of history required depending on the order of dispersion.

While the update equation in (2.22) is perfectly valid, more efficient update strategies can be adopted in which previous field and convolution values need not be stored explicitly. Continuing to follow [10], a signal processing technique known as Transposed Direct Form II is used such that values can be accumulated in auxiliary variables upon each iteration. In doing so, (2.22) can be shown to take on the form:

$$\begin{aligned}\{\mathcal{W}_{\alpha,k}\}^n &= c_{\alpha,k}[T_k]\{u\}^n - d_{\alpha,k}\{\mathcal{L}_{\epsilon,k}\}^n + \{\mathcal{W}_{\alpha+1,k}\}^{n-1} \quad && \alpha = 1, 2, \cdots, p-1 \\ \{\mathcal{W}_{\alpha,k}\}^n &= c_{\alpha,k}[T_k]\{u\}^n - d_{\alpha,k}\{\mathcal{L}_{\epsilon,k}\}^n && \alpha = p \\ \{\mathcal{L}_{\epsilon,k}\}^n &= c_{0,k}[T_k]\{u\}^n + \{\mathcal{W}_{1,k}\}^{n-1}\end{aligned} \tag{2.23}$$

in which $\{\mathcal{W}_{\alpha,k}\}$ represents the various auxiliary variables within element k. Similar expressions can likewise be defined for $\{\mathcal{L}_{\mu^{-1},k}\}^n = q_{0,k}[S_k]\{u\}^n + \{\mathcal{G}_{1,k}\}^{n-1}$.

At this point, it is noted that the above formulation is applied on a per element basis, meaning that each element is free to have different dispersive properties. Elements composed of like material can naturally have their auxiliary variables and elemental $[T_k]$ and $[S_k]$ matrices in (2.23) coalesced and treated as a whole rather than individually, similar to the assembly of the global matrices, resulting in one set of auxiliary variables per type of material, ϵ_m:

$$\begin{aligned}
\{\mathcal{W}_{\alpha,\epsilon_m}\}^n &= c_{\alpha,\,\epsilon_m}[T_{\epsilon_m}]\{u\}^n - d_{\alpha,\,\epsilon_m}\{\mathcal{L}_{\epsilon_m}\}^n + \{\mathcal{W}_{\alpha+1,\epsilon_m}\}^{n-1} \quad \alpha = 1,2,\cdots,p-1 \\
\{\mathcal{W}_{\alpha,\epsilon_m}\}^n &= c_{\alpha,\,\epsilon_m}[T_{\epsilon_m}]\{u\}^n - d_{\alpha,\,\epsilon_m}\{\mathcal{L}_{\epsilon_m}\}^n \quad \alpha = p \\
\{\mathcal{L}_{\epsilon_m}\}^n &= c_{0,\,\epsilon_m}[T_{\epsilon_m}]\{u\}^n + \{\mathcal{W}_{1,\epsilon_m}\}^{n-1} \\
[T_{\epsilon_m}] &= \sum_{k=1}^{N_{\epsilon_m}} [T_k]
\end{aligned} \tag{2.24}$$

With these provisions in mind, at long last the results of (2.24) and (2.14) may be combined into a single update equation for the electric field in doubly dispersive media as follows:

$$\begin{aligned}
\left\{\frac{1}{\Delta t^2}[\tilde{T}] + \frac{1}{2\Delta t}[Q] + \frac{1}{4}[\tilde{S}]\right\}\{u\}^{n+1} &= \left\{\frac{2}{\Delta t^2}[\tilde{T}] - \frac{1}{2}[\tilde{S}]\right\}\{u\}^n \\
&- \left\{\frac{1}{\Delta t^2}[\tilde{T}] - \frac{1}{2\Delta t}[Q] + \frac{1}{4}[\tilde{S}]\right\}\{u\}^{n-1} - \frac{1}{\Delta t^2}\left\{\{\widetilde{\mathcal{W}}\}^n\right. \\
&\left. - 2\{\widetilde{\mathcal{W}}\}^{n-1} + \{\widetilde{\mathcal{W}}\}^{n-2}\right\} - \frac{1}{4}\left\{\{\tilde{\mathcal{G}}\}^n + 2\{\tilde{\mathcal{G}}\}^{n-1} + \{\tilde{\mathcal{G}}\}^{n-2}\right\} \\
&- \left\{\frac{1}{4}\{f\}^{n+1} + \frac{1}{2}\{f\}^n + \frac{1}{4}\{f\}^{n-1}\right\}
\end{aligned} \tag{2.25}$$

where the modified global matrices and vectors are given by:

$$\begin{aligned}
[\tilde{S}] &= \sum_{k=1}^{N_{elem}} q_{0,k}[S_k] \quad \{\widetilde{\mathcal{W}}\}^n = \sum_{m=1}^{N_\epsilon} \{\mathcal{W}_{1,\epsilon_m}\}^n \\
[\tilde{T}] &= \sum_{k=1}^{N_{elem}} c_{0,k}[T_k] \quad \{\tilde{\mathcal{G}}\}^n = \sum_{m=1}^{N_\epsilon} \{\mathcal{G}_{1,\epsilon_m}\}^n
\end{aligned} \tag{2.26}$$

and by which doubly dispersive indicates that both the permittivity and permeability are frequency dependent. Singly dispersive materials (either electrically or magnetically) are easily handled by setting either $\{\mathcal{W}\}$ or $\{\mathcal{G}\}$ to zero. These last two equations when coupled to the unchanged

quantities from (2.6) through (2.9) contain all required information to update all variables upon each iteration. The process has the following basic steps:

1. Advance $\{u\}^n$ to $\{u\}^{n+1}$ using (2.25) and all previously known quantities.
2. Advance $\left\{\mathcal{L}_{\epsilon_m}\right\}^n$ and $\left\{\mathcal{L}_{\mu_m^{-1}}\right\}^n$ to $(n+1)$ using $\{u\}^{n+1}$, $\left\{\mathcal{W}_{1,\epsilon_m}\right\}^n$ and $\left\{\mathcal{G}_{1,\mu_m^{-1}}\right\}^n$.
3. Advance each $\left\{\mathcal{W}_{\alpha,\epsilon_m}\right\}^n$ and $\left\{\mathcal{G}_{\alpha,\mu_m^{-1}}\right\}^n$ in turn to $(n+1)$ using (2.24).
4. Repeat until desired end time.

With an understanding of the inner workings of both non-dispersive and dispersive FETD methods, attention is now shifted toward analysis of the method as it pertains to parallelization.

Chapter 3: Parallelization Strategy

3.1 The Graphics Processing Unit

Dedicated graphics hardware within computer workstations have been required since the early days of computing, with machines communicating with the user often through visual means. In their chapter "History of GPU Computing", Kirk & Hwu [15] mention that while some of the earliest of graphics processing units (GPUs) were non-programmable pipelines for the display of simple wire-frame drawings, the modern day GPU has evolved into an immensely configurable, complex, and powerful computational engine. Indeed, they note that today's GPUs have enjoyed tremendous growth in computational power over the past several years, propelled by advances in miniaturization as well as an ever increasingly high demand for consumer level graphics cards capable of real-time high-definition rendering. Quality video playback or computer games, for example, can require the rendering of 1920×1080 resolution images at 60 frames per second, necessitating the computation and update of over 124 million pixel elements every second; a substantial computational workload.

It is no surprise, then, that graphics processing hardware rapidly began to evolve to exploit the nature and structure of graphics computations, in order to cope with the ever increasing computational demand. In the vast majority of cases, the 3D modelling, rendering and display operations associated with computer graphics today requires the same computation be independently repeated thousands of times on different data sets. Texture mapping, Kirk & Hwu give as an example, requires each pixel within an object independently undergo coordinate transformations, transferring a 2D image to a 3D surface. As such, the advantages of concurrent processing were rapidly identified and GPUs progressively developed ever more parallelized architectures: with many units mapping the textures at once without the need to communicate, the computational throughput of the entire system increases drastically. Such was the driving force behind ever more realistic and intensive graphics that GPUs developed a multitude of very intricate parallel pipelines, each capable of handling the complexity of three dimensional models and scenes [16].

Propelled by the need for high computational and memory throughput, GPUs have quickly surpassed and eclipsed their central processing unit (CPU) counterparts in terms of sheer number

of floating point operations per second and memory bandwidth. Fig. 1 illustrates the performance increase of both GPUs and CPUs over the past 12 years, in terms of floating point operations per second, demonstrating an escalating imbalance in performance.

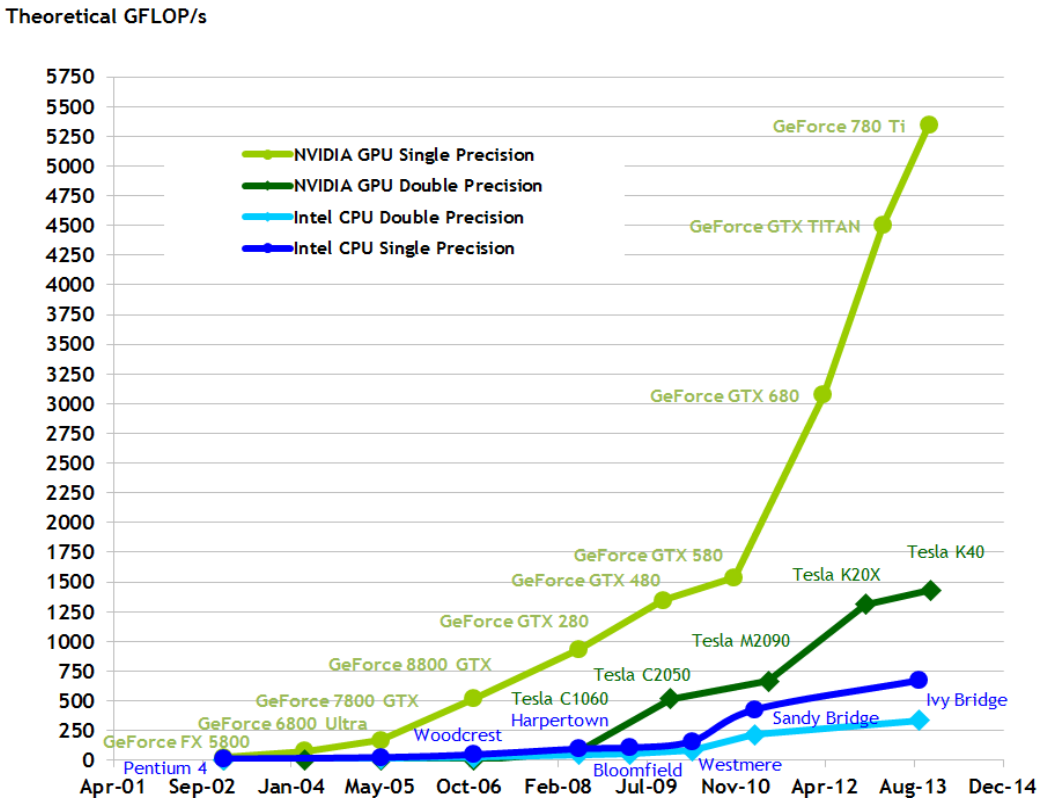

Fig. 1 Evolution of floating point operations per second for both GPUs and CPUs (reproduced from [16] with permission).

The contrast in performance between GPUs and CPUs in terms of floating point operations and memory bandwidth arises from the evolution of fundamentally different design philosophies, argue Kirk & Hwu. They go on to note that CPUs have been streamlined over the years to excel at traditional sequential computations, and as such have been equipped with sophisticated caching and control structures. GPUs on the other hand have focused on parallelization and arithmetic, resulting in huge amounts of data processing structures at the expense of far less cache and control [16], as evidenced in Fig. 2. Here, orange regions represent the various memory spaces available, with the extremely rapid cache used to reduce dependence on slower DRAM, which itself acts as general storage. The green regions highlight Arithmetic Logic Units (ALUs), which are responsible for executing arithmetic and logical operations, while yellow areas outline control regions whose job it is to coordinate execution. Since most applications do not fall entirely within

the sequential or parallel categorizations, this difference in design and specialization motivates joint CPU/GPU algorithms, in order to get the best of both worlds.

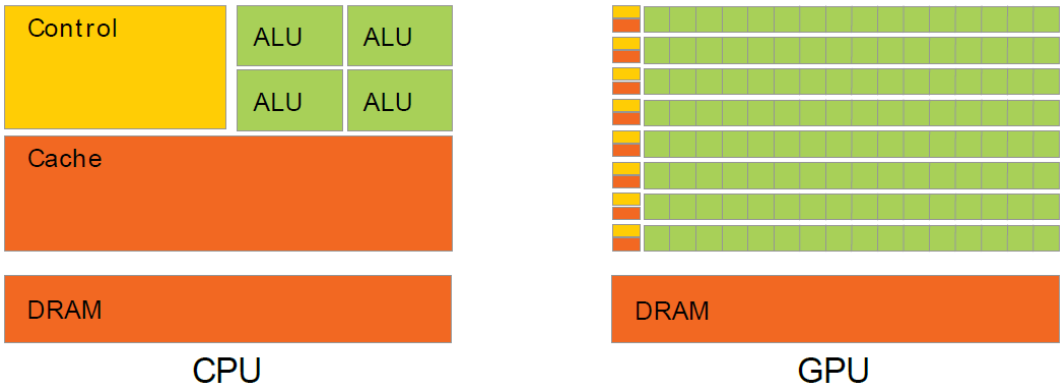

Fig. 2 CPU vs. GPU architecture, demonstrating fundamentally different design philosophies (reproduced from [16] with permission).

It is only in relatively recent times however that GPUs have begun to be utilized for a wide array of scientific and engineering problems in addition to graphics processing. Prior to 2006, explain Kirk & Hwu, programming a GPU for scientific computing was a complex and difficult procedure, requiring the use of specific graphics application programming interfaces (APIs). Even with the use of high-level languages, programmers were severely limited in the applicability and generality of their applications, due to constraints within the existing APIs themselves. With such a narrow range of programming functionality, they conclude that GPUs were too unwieldy to see widespread use in scientific computing.

3.2 Compute Unified Device Architecture

As detailed in the CUDA C Programming Guide [16], the NVIDIA Corporation introduced the Compute Unified Device Architecture (CUDA) language for its GPUs in November of 2006. Being described by the guide as a general purpose parallel computing language, CUDA enabled programmers and developers simpler and less restrictive access to NVIDIA hardware. In doing so, the immense computational power of the GPU became accessible to a far wider and less specialized audience. Furthermore, being principally programmed in popular high-level environments such as C and C++, with a simple set of language extensions to accommodate GPU functionality, a basic familiarity with the aforementioned languages enabled the new CUDA user to enjoy a comparatively gentle learning curve [17]. An application written in CUDA, therefore, could have some portions written in C/C++, meant for execution on the CPU, and other sections

making extensive use of the language extensions made for execution on the GPU. CUDA thereby allowed the user the freedom to easily use both serial and parallel computations within a single application, an important and extremely useful feature.

In addition to the relative ease of use, CUDA enjoyed another significant benefit, that of scalability. With a vast assortment of GPU products available with wildly differing specifications, scalability ensured that codes designed for less powerful systems were able to take full advantage of additional resources when run on more modern or powerful machines [16]. Applications written in CUDA could therefore experience performance boosts simply by migrating to more modern hardware, standing the test of time. In this way, CUDA has grown since then into a widely used and established language and it is for these reasons that the present investigation is carried out entirely on NVIDIA GPUs, using the CUDA language.

3.3 The GPU Architecture

As outlined by Wilt in [18], the modern NVIDIA GPU is composed of a complex scalable array of multithreaded Streaming Multiprocessors (SMs), each capable of executing hundreds of concurrent threads and floating point operations at once. He further details how each SM can be thought of as essentially a miniature processor, having at its disposal eight or more CUDA cores (principally used for arithmetic operations), two or more units dedicated to special transcendental functions, and schedulers. He also mentions and describes how the SM is home to an assortment of dedicated memory spaces: registers, shared memory and constant memory. Registers are specifically assigned as working memory to each executing thread and constitute the fastest type. Shared memory, as the name implies, is shared between concurrently executing threads, allowing them to trade information at high speed. Lastly, constant memory (technically not part of the SM and located “off chip”) can store a variety of invariant values often used by executing threads, with a relatively high read speed thanks to additional hardware.

Despite their quickness, the aforementioned on-chip memories are limited in size for each SM, restricting the volume of data that can be rapidly accessed. In such cases Farber [19] details how each SM also has access to the GPU global memory, a vastly larger amount of storage present on the card, but off-chip. Owing to this, global memory accesses are far and away slower than their on-chip counterparts, taking on average many, many more clock cycles to execute. While often times unavoidable, he mentions the method by which a program chooses to access global memory

can play a crucial role in determining performance. Furthermore, Farber notes that since GPU global memory residing on the device is physically separate from the host machine, any data required by the SMs during the course of their execution, which is not already in global memory, must be transferred from the host. This transfer can result in significant overhead, and so code is often written in order to minimize the amount of data transfer. Even so, in the worst case scenarios the overhead of data transfer can overshadow any performance boost gained from the parallelization, resulting in very little and, in some cases, negative performance increases.

The execution of threads on the SMs is performed according to specific groupings, rather than all threads being scheduled individually, as explained in the CUDA C Programming Guide [16]. The guide details how each SM is assigned batches of threads known as blocks, ensuring that all threads within a specified block have access to the SM's resources (enabling exchange via shared memory for example). Multiple blocks can be assigned to an SM, as long as it has sufficient resources to accept it. Additionally, the guide explains how blocks are further subdivided into collections of 32 threads called warps, with each warp member thread executing in step with each other. This execution style is described as Single Instruction Multiple Thread (SIMT), in which every thread within the warp is executing identical instructions at the same time, albeit not necessarily on the same data (SIMT does, however, also allow for each thread to branch independently of each other, if needed). It is here that the strength of the GPU in parallel processing is seen; tasks in which identical operations must be repeated many times on different data sets can be handled concurrently and with ease. This complex organization of blocks, threads and memory is depicted schematically in Fig. 3.

Since the exact order in which blocks will be run cannot be guaranteed, there can be no sequential interdependence of threads within different blocks, i.e. a thread cannot easily depend on the result of another [16]. Hence, to take full advantage of the GPU's power, the required operations must not only follow the SIMT principle, but be fully independent of each other. It is these criteria that one must keep in mind when analyzing a new algorithm for parallelization, as will be done in the next section. Failing to adhere to either requirement would result in either meaningless results, or the serialization of thread execution on the GPU, negating any benefit.

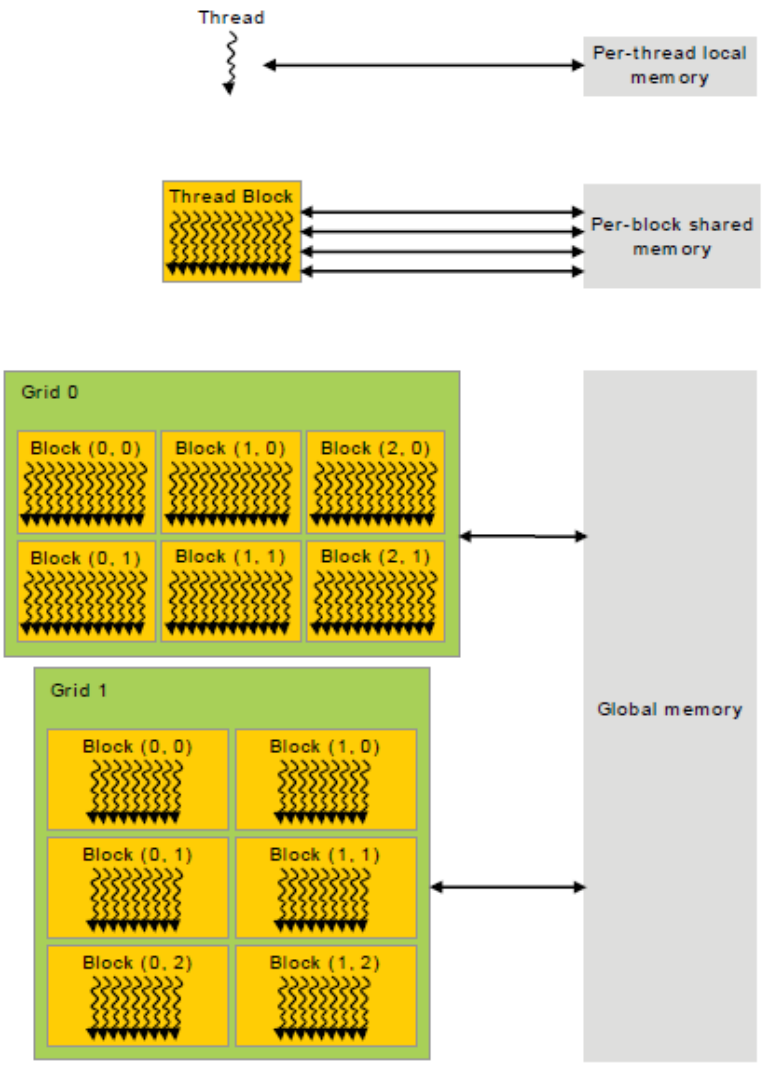

Fig. 3 Overview of a CUDA device's hardware organization (reproduced from [16] with permission).

3.4 Analyzing the Dispersive FETD Algorithm

With a comprehensive understanding of the FETD algorithm, dispersion, the GPU architecture and its programming model, an analysis of the algorithm presented in Section 2.3 may now be performed. With the main goal being the isolation and acceleration of dispersive overhead within the FETD formulation, the update equations (2.11) and (2.25) are reproduced below in order to proceed with a direct comparison.

$$\begin{aligned}
\left\{\frac{1}{\Delta t^2}[T]+\frac{1}{2\Delta t}[Q]+\frac{1}{4}[S]\right\}\{u\}^{n+1} &= \left\{\frac{2}{\Delta t^2}[T]-\frac{1}{2}[S]\right\}\{u\}^{n} \\
&-\left\{\frac{1}{\Delta t^2}[T]-\frac{1}{2\Delta t}[Q]+\frac{1}{4}[S]\right\}\{u\}^{n-1} \\
&-\left\{\frac{1}{4}\{f\}^{n+1}+\frac{1}{2}\{f\}^{n}+\frac{1}{4}\{f\}^{n-1}\right\} \\
\left\{\frac{1}{\Delta t^2}[\tilde{T}]+\frac{1}{2\Delta t}[Q]+\frac{1}{4}[\tilde{S}]\right\}\{u\}^{n+1} &= \left\{\frac{2}{\Delta t^2}[\tilde{T}]-\frac{1}{2}[\tilde{S}]\right\}\{u\}^{n} \\
&-\left\{\frac{1}{\Delta t^2}[\tilde{T}]-\frac{1}{2\Delta t}[Q]+\frac{1}{4}[\tilde{S}]\right\}\{u\}^{n-1}-\frac{1}{\Delta t^2}\left\{\{\widetilde{\mathcal{W}}\}^{n}\right. \\
&\left.-2\{\widetilde{\mathcal{W}}\}^{n-1}+\{\widetilde{\mathcal{W}}\}^{n-2}\right\}-\frac{1}{4}\left\{\{\tilde{\mathcal{G}}\}^{n}+2\{\tilde{\mathcal{G}}\}^{n-1}+\{\tilde{\mathcal{G}}\}^{n-2}\right\} \\
&-\left\{\frac{1}{4}\{f\}^{n+1}+\frac{1}{2}\{f\}^{n}+\frac{1}{4}\{f\}^{n-1}\right\}
\end{aligned} \tag{2.25}$$

In observing the above equations, their similarities and differences become readily apparent. Recalling that the difference between $[\tilde{S}], [\tilde{T}]$ and $[S], [T]$ is merely a scaling of the elemental sub-matrices, they will have the exact same sparsity pattern and therefore require approximately the same amount of computational effort to multiply or solve. Since the remaining quantities $[Q]$ and $\{f\}$ are identical in both formulations, it is immediately evident that the two schemes differ essentially in the presence of the auxiliary variables, $\{\widetilde{\mathcal{W}}\}$ and $\{\tilde{\mathcal{G}}\}$. The inclusion of dispersion in the FETD method is therefore tantamount to evaluating the auxiliary variable update equations upon each iteration:

$$\begin{aligned}
\{\mathcal{W}_{\alpha,\epsilon_m}\}^{n} &= c_{\alpha,\,\epsilon_m}[T_{\epsilon_m}]\{u\}^{n}-d_{\alpha,\,\epsilon_m}\{\mathcal{L}_{\epsilon_m}\}^{n}+\{\mathcal{W}_{\alpha+1,\epsilon_m}\}^{n-1} \qquad \alpha=1,2,\cdots,p-1 \\
\{\mathcal{W}_{\alpha,\epsilon_m}\}^{n} &= c_{\alpha,\,\epsilon_m}[T_{\epsilon_m}]\{u\}^{n}-d_{\alpha,\,\epsilon_m}\{\mathcal{L}_{\epsilon_m}\}^{n} \qquad \alpha=p \\
\{\mathcal{L}_{\epsilon_m}\}^{n} &= c_{0,\,\epsilon_m}[T_{\epsilon_m}]\{u\}^{n}+\{\mathcal{W}_{1,\epsilon_m}\}^{n-1}
\end{aligned} \tag{2.24}$$

Attention has now shifted therefore toward an analysis of these update equations, in order to determine whether they are compatible with the SIMT and independence principles put forth in the previous section. Looking closely at equation (2.24), it is noted that the updates are composed of three basic linear algebra operations: matrix-vector product, vector-scalar product and vector-vector addition. As a first step, the matrix-vector product between $[T_{\epsilon_m}]$ and $\{u\}^{n+1}$ is expressed, by definition, as a summation:

$$\left\{\left[T_{\epsilon_m}\right]\{u\}^{n+1}\right\}_i = \sum_{j=1}^{n}\left[T_{\epsilon_m}\right]_{i,j}\{u\}_j^{n+1} \tag{3.1}$$

From here it becomes clear that each i (i.e. each entry of the product vector) is completely independent of the others. Indeed, a knowledge of the i^{th} row of $\left[T_{\epsilon_m}\right]$ and $\{u\}^{n+1}$ is all that is required to compute the i^{th} entry of the product, $\left[T_{\epsilon_m}\right]\{u\}^{n+1}$. Furthermore, it is evident that the calculation of each entry of the product involves the exact same operations, simply performed on

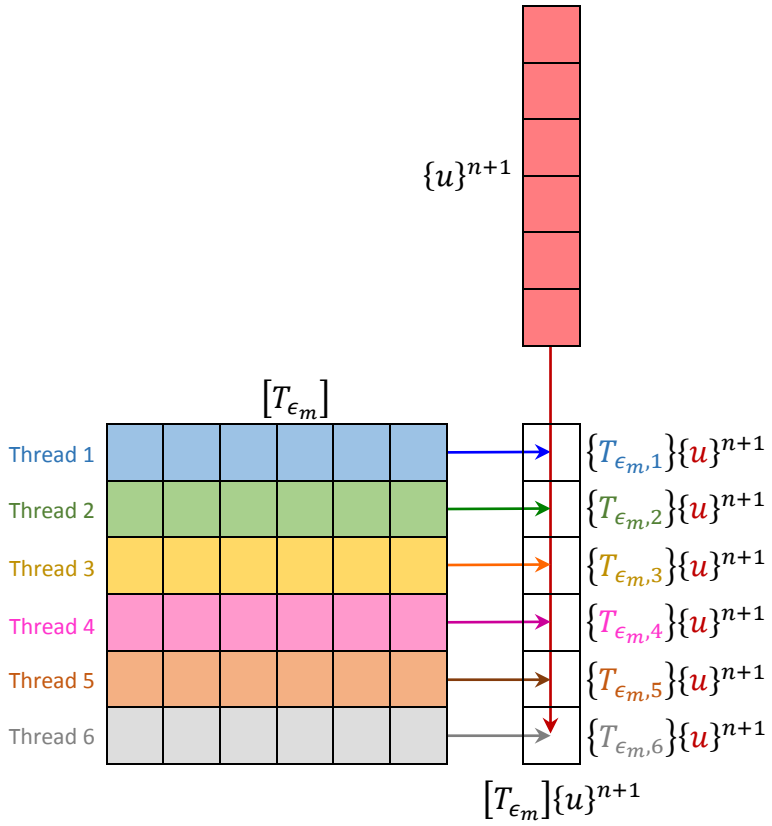

Fig. 4 Parallelized matrix-vector product, demonstrating each thread computing an entry of the product vector.

different datasets. This suggests that each entry of the product vector can be calculated by a distinct thread, allowing for each entry to be computed simultaneously. This is depicted pictorially in Fig. 4 above.

In each of the update equations (2.24), the matrix vector product need only be calculated once, then re-used for each subsequent auxiliary variable. Once the product vector is known, progress may be made one auxiliary variable at a time, taking the required linear combination of

vectors. Here now however, it is also evident that vector addition and scaling are equally carried out in a componentwise independent manner, with each entry computed via identical operations:

$$\{\mathcal{W}_{\alpha,\epsilon_m}\}_i^n = c_{\alpha,\epsilon_m}\left\{[T_{\epsilon_m}]\{u\}\right\}_i^n - d_{\alpha,\epsilon_m}\{\mathcal{L}_{\epsilon_m}\}_i^n + \{\mathcal{W}_{\alpha+1,\epsilon_m}\}_i^{n-1} \tag{3.2}$$

Since the scaling and addition are dependent upon the solution of the matrix-vector product however, these two operations cannot be performed concurrently. This in turn suggests that each thread should be responsible for computing one entry of the product and the subsequent additions and scaling. In the case of a singly (electrically) dispersive material, this parallelized version of the update equations would then have the form as seen in Fig. 5, for each running thread. A magnetically or doubly dispersive material would also naturally include the parallelization of $[S_{\epsilon_m}]\{u\}^{n+1}$ and $\{\mathcal{G}\}$.

id = thread number;
// Compute the idth matrix-vector product entry.
for (j = 0; j < n; ++j)
{
$\{r\}_{\mathrm{id}} \mathrel{+}= [T_{\epsilon_m}]_{id,j}\{u\}_j^{n+1}$;
}
// Update the idth entry of each auxiliary variable.
$\{\mathcal{L}_{\epsilon_m}\}_{id}^{n+1} = c_{0,\epsilon_m}\{r\}_{id} + \{\mathcal{W}_{1,\epsilon_m}\}_{id}^{n}$;
$\{\mathcal{W}_{1,\epsilon_m}\}_{id}^{n+1} = c_{1,\epsilon_m}\{r\}_{\mathrm{id}} - d_{1,\epsilon_m}\{\mathcal{L}_{\epsilon_m}\}_{id}^{n+1} + \{\mathcal{W}_{2,\epsilon_m}\}_{id}^{n}$;
(…)
$\{\mathcal{W}_{p,\epsilon_m}\}_{id}^{n+1} = c_{p,\epsilon_m}\{r\}_{\mathrm{id}} - d_{p,\epsilon_m}\{\mathcal{L}_{\epsilon_m}\}_{id}^{n+1}$;

Fig. 5 Parallelized pseudocode to update auxiliary variables for an electrically dispersive medium.

As exemplified by the above analyses, it is concluded that the overhead imposed by the inclusion of dispersion within the FETD equations is an excellent candidate for parallelization. The parallelized algorithm will, however, still necessitate intimate cooperation between the CPU and GPU (including memory transfers, as mentioned in Section 3.3), with those sections common between the serial and parallel methods (and therefore also the non-dispersive method) still requiring treatment by the CPU. For example, from Fig. 5, knowledge of $\{u\}^{n+1}$ is naturally required in order to update all of the $\{\mathcal{W}_{\alpha,\epsilon_m}\}$ to $n+1$. However, $\{u\}^{n+1}$ is obtained by the CPU via solution of the matrix problem in (2.25), and must therefore be transferred from the CPU to

the GPU, since they do not share the same memory. Likewise, once the updates are complete, the GPU must then transfer the result back to the CPU, for use in the next matrix solve. For the purpose of this thesis, the overall parallelized algorithm therefore takes on the form as seen in Fig. 6, whereby it becomes clear that any speed boost afforded by the GPU in updating the auxiliary variables will cause the dispersive method's execution to asymptotically approach that of the non-dispersive method.

```
(…)                          // Compute [A] and {b}.
{u}^{n+1} = [A]^{-1}{b}      // Compute {u}^{n+1}.
cudaMemcpy()                 // Transfer {u}^{n+1} from CPU to GPU.
<<< Kernel >>>               // Update auxiliary variables on GPU.
cudaMemcpy()                 // Transfer variables back to CPU.
(…)                          // Repeat the process.
```

Fig. 6 Main program loop pseudocode for parallelized dispersive FETD method.

Chapter 4: Implementation

4.1 Overview

Having laid the theoretical foundations required to understand, analyze and parallelize the dispersive FETD method, a detailed discussion of the actual algorithm as it was developed and implemented in this project is now presented. In order to verify any performance gain afforded by parallelization and the above analyses, an experimental procedure was devised whereby the dispersive FETD algorithm was written both in a traditional serial execution style on a CPU (acting as a control), as well as a parallel style on a GPU (Fig. 6), in which the only difference was in the updating of auxiliary variables. The two pieces of code then had detailed execution time information for a variety of problems recorded and compared, with any performance changes reported in terms of this data.

Execution time was selected as the performance metric of choice in performing these comparisons, owing to both its simplicity and immediately relatable nature. Any performance increase found to be granted by parallelization thereby directly translated to a narrowing of the gap between dispersive and non-dispersive methods. With direct comparison between two different implementation strategies having been the ultimate goal, numerous subtleties arose in how the algorithms were implemented such that the comparison was meaningful and fair. These issues are further addressed within the following sections.

4.2 Algorithm Design

Prior to commencing any writing of code, it was necessary to establish a framework within which the various aspects of this project were to be implemented. Of chief concern was the selection of an appropriate FETD solver and related libraries (such as linear algebra, etc.), since such pieces of code would fundamentally form the basis of all of the above discussed dispersive methods. While choosing to use existing software packages and libraries was tempting, in the present work the decision was made to use majoritarily custom routines written specifically for this thesis. The decision to “start from scratch” rather than implement existing free or commercial software and libraries stemmed from the requirement to be able to not only understand but control every aspect of the solution process, to ensure an accurate and fair comparison in performance.

For instance, the introduction of dispersive media or parallelization into an existing FETD solver may require modification of fundamental pieces of the source code not readily accessible to the user. Additionally, and more importantly, since a direct comparison was to be made between parallelized and serial versions of the algorithm, it was important that the two versions differed only in terms of the analysis presented in Section 3.4. For example, had a software package in the CPU version of the auxiliary update equations used a vector multiplication algorithm or sparse storage format that differed from that used on the GPU (owing to language differences or other constraints), a performance gain or loss may have been recorded simply due to one algorithm or format being fundamentally more efficient than the other (or even the same algorithm being written two different ways), resulting in confounding variables. By custom writing the majority of the routines used in both versions, it could be assured that the only independent variable linked to performance was the utilization of the GPU.

While the latter justification pertained to the updating of the auxiliary variables, there nonetheless remained the portions of the serial and parallel algorithms which were identical (the portions of pseudocode in Fig. 6 not in blue). Since these portions (mostly composed of linear algebra operations, including matrix solving) were to be run by the CPU in both cases, they could have in principle both made use of existing linear algebra packages (such as Eigen [20]) without fear of altering the comparison. However, it was nonetheless decided to once again use custom written subroutines and sparse storage formats for these portions of the algorithm, owing to the ability to exert more control and increase efficiency. Since these custom routines were written specifically for the problem at hand, they were capable of exploiting the symmetry and structure of the resulting datasets, combining multiple methods to achieve ideal performance over some generic libraries. Additional information pertaining to these custom routines and sparse matrix formats is found in subsequent sections.

4.3 Programming Language and Chronometry

With the above requirements in place, the task of actually implementing the algorithm was begun, commencing with the necessity of picking a programming language. Of primary importance in selecting a language for the serial CPU version of the code was similarity to CUDA. In like manner to the last section, non-essential differences between the serial and parallel versions needed to be kept to a minimum so as to yield a fair comparison. In this case, comparing a serial

piece of JAVA code, for example, to a parallelized piece of C code may not have been a fair test, since differences in language efficiency and structure could have obfuscated the desired correlation. As was mentioned in Chapter 3 however, CUDA is primarily an extension of the C and C++ programming languages, and therefore these constituted the most logical choices. Ultimately the C++ programming language was selected as the main workhorse for this project, given its more modern object oriented nature. To facilitate the development and writing of the C++ dispersive FETD code, Microsoft Visual Studio Ultimate 2013 (update 3) was used as an integrated development environment for most of the project. Likewise, the NVIDIA Nsight Visual Studio Edition (version 4.1) plugin was used, allowing for the compiling, debugging and profiling of GPU executed CUDA code within Visual Studio.

As previously mentioned, the main statistics having been collected in the present work were execution times of the various regular and parallelized versions of the dispersive algorithm, in order to compare performance. Consequently, the need to perform accurate measurements of execution time was of primary importance. Initially, program execution was measured using the available Visual Studio and NVIDIA profiling applications. After testing, this was eventually deemed unsatisfactory, since the exact method used for measuring elapsed time may differ between the two profilers, and because the use of profilers sometimes requires the insertion of extra code snippets into the algorithms under test [21]. Additionally, despite NVIDIA providing a set of "CUDA Event Timers" with high resolution, through experimentation it was found that such timers measured time spent strictly on the GPU, without regard for overhead, some CUDA API calls and other potentially significant delays. These issues lead to the eventual investigation and use of the host machine's high-resolution performance counter, accessed via either the QueryPerformanceCounter function (Windows) or the clock_gettime() function (Unix) [22]. Allowing for approximately microsecond resolution, the performance counter proved to be the optimal choice for measuring execution times on both the CPU and GPU.

4.4 Serial CPU Code Overview

A more in depth look at the actual machinery and implementation of the C++ CPU version of the dispersive FETD algorithm, which was used in the next chapter as a control, is now undertaken. In commencing, several simplifying assumptions were made in order to reduce the complexity and size of the final versions of the code. Firstly, rather than coding the functions

required to generate the problem meshing (triangular spatial discretization) and global matrices of equation (2.25) in C++, it was decided to have a custom written MATLAB routine generate this data and save it to binary files. The C++ and CUDA code would then commence by reading the pre-formed matrices (and other requisite constants) from file and storing them to working memory. In doing so, existing MATLAB functions could be utilized to mesh the domain and perform other special functions required in the preparatory stages, alleviating some of the programming burden. The use of existing libraries was adopted here contrary to the previous section since the creation of the global matrices is a one-time fixed cost upon commencing the algorithm, in contrast to the update equations which must be executed during each and every time step. For a suitably large number of time steps, this initial cost becomes negligible. Additionally, since the present work was not concerned with augmenting or parallelizing the global matrix construction, it would therefore be identical in both versions of the code, and so serves little purpose to include.

Secondly, any and all custom subroutines which were shared between the CPU and GPU implementations (such as matrix solving) were compiled into dynamically linked libraries, rather than being duplicated in each of the serial and parallel source files. Doing so allowed for the elimination of any variability between the two in compilation, since they both reference the same pre-compiled library, ensuring yet again a fair comparison in execution times. The use of dynamically linked libraries equally allowed for increased program efficiency, as the overall size of the requisite executables was decreased.

As was briefly mentioned in the previous section, the selection of a preferred sparse storage format had the ability to strongly impact software performance, having been required in order to boost efficiency and eliminate the need to manipulate massive, mostly empty, matrices. From the theoretical treatment in Chapter 2, it was apparent that each edge weight $\{u\}_i(t)$ could only belong to a maximum of two elements (since space was discretized in terms of triangles) and so could only couple to a maximum of 4 other edges and itself. This implied that regardless of the number of elements, each row of the global matrices had at most five entries, an extremely sparse problem. With this sparsity pattern pervasive in the global matrices, it was determined that the best sparse storage scheme for this problem was the ELLPACK format. This format is generally much faster than the alternatives for unstructured FEM meshes, as it exploits the fact that each row of the matrices has a precisely known number of non-zeroes [23]. For an $n \times n$ sparse matrix with m

non-zeroes per row, the ELLPACK format converts the sparse matrix into two $n \times m$ dense matrices, reducing the required memory from n^2 elements to $2mn$. For the current investigation, n was generally well over 10,000 while m was only 5; a substantial savings. To convert to the ELLPACK format, a sparse matrix essentially has all zeroes squeezed out of each row, resulting in the so-called "data" matrix, while the original entry's column information is stored into a second "index" matrix. Therefore, the row number of an entry in the data matrix corresponds exactly to the row number in the original sparse matrix, while the column number can be found by reading the corresponding value within the index matrix. This is exemplified in Fig. 7, for which $n = 5$ and $m = 2$.

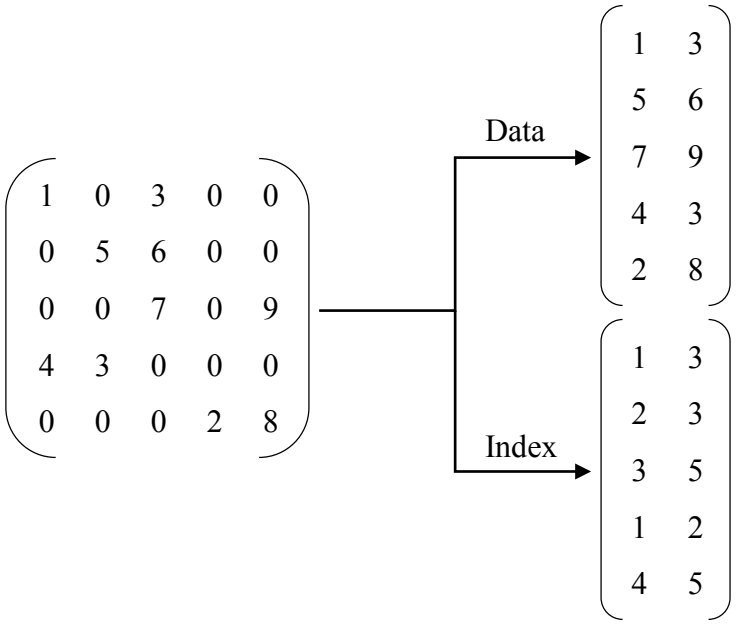

Fig. 7 ELLPACK sparse matrix storage format, ideal for unstructured FETD meshes.

If each of the matrix quantities $[\tilde{S}], [\tilde{T}], [Q], [S_{\epsilon_m}], [T_{\epsilon_m}]$ had been stored individually, time would then have to have been spent merging some of them into the requisite combinations given in equation (2.25). Instead, the decision was made to pre-combine the above into pre-assembled matrices $[M], [M_1]$ and $[M_2]$, defined as follows:

$$[M] = \frac{1}{\Delta t^2}[\tilde{T}] + \frac{1}{2\Delta t}[Q] + \frac{1}{4}[\tilde{S}]$$
$$[M_1] = \frac{2}{\Delta t^2}[\tilde{T}] - \frac{1}{2}[\tilde{S}] \tag{4.1}$$
$$[M_2] = \frac{1}{\Delta t^2}[\tilde{T}] - \frac{1}{2\Delta t}[Q] + \frac{1}{4}[\tilde{S}]$$

for which the update equations were then simply recast into the following form:

$$[M]\{u\}^{n+1} = [M_1]\{u\}^n - [M_2]\{u\}^{n-1} + \{\mathcal{K}\}$$
$$-\left\{\frac{1}{4}\{f\}^{n+1} + \frac{1}{2}\{f\}^n + \frac{1}{4}\{f\}^{n-1}\right\} \tag{4.2}$$

wherein the substitution

$$\{\mathcal{K}\} = -\frac{1}{\Delta t^2}\left\{\{\tilde{\mathcal{W}}\}^n - 2\{\tilde{\mathcal{W}}\}^{n-1} + \{\tilde{\mathcal{W}}\}^{n-2}\right\}$$
$$-\frac{1}{4}\left\{\{\tilde{\mathcal{G}}\}^n + 2\{\tilde{\mathcal{G}}\}^{n-1} + \{\tilde{\mathcal{G}}\}^{n-2}\right\} \tag{4.3}$$

has equally been made. In this way, the matrices $[M_1]$ and $[M_2]$ were stored in the ELLPACK format and were ready for immediate use by the programs at run time (after being loaded from binaries). The $[M]$, $[T_{\epsilon_m}]$ and $[S_{\epsilon_m}]$ matrices were not, however, stored in the ELLPACK format. The former received additional treatment (detailed below), since it appears on the left hand side of (4.2). The latter two, we recall from Chapter 2, only involve those elements with permittivity ϵ_m, and as such could in fact contain rows of all zeroes. While this could have been represented in the ELLPACK scheme above (since non-zero rows nevertheless contained on average 5 entries), it did not represent the most efficient form. Instead, a modified ELLPACK (MELLPACK) approach was devised in which an additional column was added to the index matrix, whereby the first column then designated row number, and all-zero rows were omitted. This is demonstrated in Fig. 8, for which once again $n = 5$ and $m = 2$.

Lastly, attention was turned toward the $[M]$ matrix. Given that it appears on the left hand side of the update equations, the CPU was forced to solve the system $[M]\{u\}^{n+1} = \{b\}$ upon each iteration. In order to render this process more efficient, the symmetric positive definite nature of matrix $[M]$ was exploited (as well as the fact that it is invariant over the course of the computation)

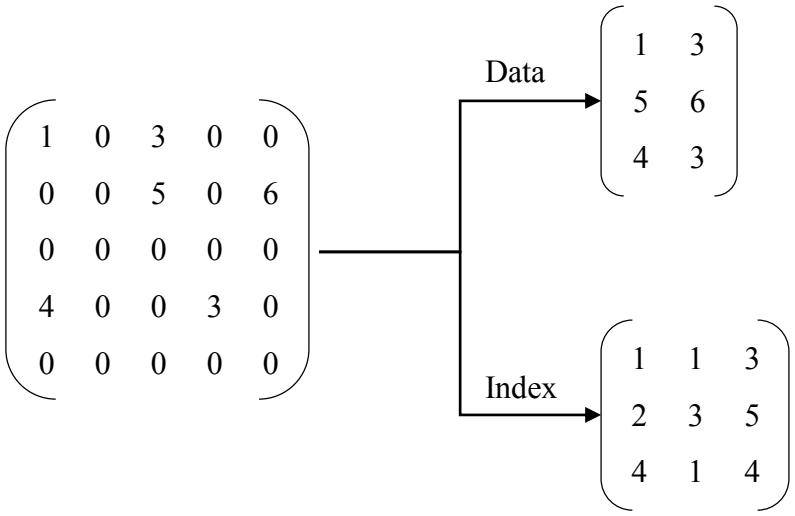

Fig. 8 MELLPACK sparse matrix storage format, with added column in the "Index" matrix.

by using a Cholesky factorization with a fill-in reducing permutation [24]. As the name suggests, a permutation matrix $[S]$ was first applied to $[M]$ in order to reduce the appearance of non-zeroes after factorization to a minimum:

$$[L][L]^T = [S]^T[M][S]. \tag{4.4}$$

This altered the solution process as follows:

$$\begin{aligned} [L]\{y\} &= [S]^T\{b\} \\ [L]^T\{x\} &= \{y\} \\ [S]^T\{u\}^{n+1} &= \{x\}. \end{aligned} \tag{4.5}$$

In this way, the solving process was rendered very efficient (in essence sparse forward/backward substitution), and required only the storage of the $[L]$ matrix and the permutation matrix $[S]$. However, the Cholesky factor $[L]$ required a very different storage pattern from the above. Indeed, by design, $[L]$ was a lower triangular matrix and was a terrible choice for the ELLPACK format, as each row had a different number of elements. To counter this, the $[L]$ matrix was stored in the Compressed Sparse Row (CSR) format [23]. Here, three vectors were used to represent the sparse data. The first "data" vector contained all non-zero entries of the matrix of interest concatenated by row, while the second "column" vector contained their respective column indices. The i^{th} entry of the third "row" vector, meanwhile, yielded the location of the start of the i^{th} row within the

"data" vector. Thus, an $n \times n$ matrix with m non-zeros is reduced from n^2 elements down to $2m + n$, as depicted in Fig. 9, for which $n = 5$ and $m = 9$.

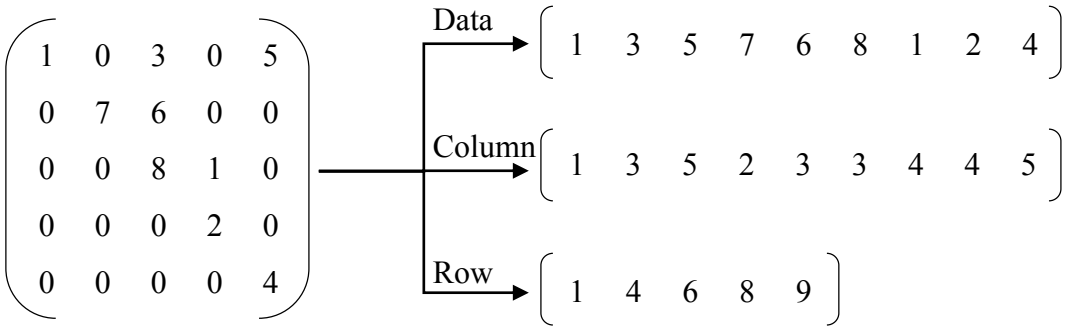

Fig. 9 Compressed Sparse Row (CSR) storage format, good for upper or lower triangular matrices.

With the above data structures, numerical methods and design choices having been made, the serial CPU dispersive algorithm and all the required shared libraries were written in C++ and thoroughly tested in order to assure proper functionality. From here, the task of writing the parallelized CUDA version was undertaken, being somewhat easier due to much of the overlap with the serial version.

4.5 Parallel GPU Code Overview

Having established much of the required machinery, libraries and techniques in the formulation of the serial algorithm, the creation of the parallel CUDA algorithm proved somewhat easier, as only the auxiliary variable updating required re-writing. Nonetheless, much care had to be taken in putting together the CUDA segments of the code to ensure respectable performance on the part of the GPU. In order to obtain good results, the details of GPU hardware and organization presented in Chapter 3 must be taken into account when composing CUDA code. Consequently, while the methods of parallelization for the algorithm were amply addressed in Section 3.4, focus is now made on the incorporation of the GPU hardware layout into the software design.

As was detailed previously, the GPU has many different layers of memory at its disposal, with varying levels of size and speed. The slowest memory operations occur in moving data to and from the GPU and CPU (given that they are physically separate devices), and is necessary within each iteration to transfer $\{u\}^{n+1}$ to the GPU prior to updating, and $\{\mathcal{K}\}$ from the GPU back to the CPU afterwards. In the present work these operations were unavoidable given the focus on only

the dispersive overhead aspect of the simulation, although they do have the potential to be alleviated in future work (see Section 6.2).

Requirements for the size of the working arrays, dictated by the problem dimensions, unfortunately negated the ability of using the extremely fast, yet limited, on-chip shared memory. While it was possible to attempt to speed up matrix-vector multiplications by pre-loading small sections of the vector of interest into shared memory (essentially a form of caching), this was not employed in this project due to the sparse nature of the matrices. This owing to the fact that in the sparse case the number of zero elements drastically reduces the effectiveness of caching the data, since multiplications by zero are skipped. The increase in efficiency afforded was therefore not expected to be large, however it could nonetheless be implemented in future versions of the code, or for problems with better suited matrices.

These constraints thereby necessitated the use of the larger, yet much slower, global memory banks on the GPU card. Nevertheless, optimizations in how data is stored in global memory could be made in order to reduce latency to a minimum. If each thread executing concurrently within a warp were to request a read from different sections of global memory, the result would be very slow, as 32 separate memory fetches must be performed. However, should each of the threads access consecutive memory locations, such that a single contiguous block is needed, CUDA allows for the whole chunk to be fetched in one call [25]. It was therefore of great importance from an efficiency standpoint to ensure that all global memory accesses were coalesced in this way. For the current problem, in which each thread was simultaneously performing matrix-vector multiplication, among other operations, this translated into the need to store each MELLPACK matrix in a column major format (matrices stored in an array by concatenating the columns), rather than row major [23], as exemplified in Fig. 10.

However, not all of the data used during updating was too large to be contained in the more rapid memory layers. The material parameters of equation (2.24), for example, represent generally only a few bytes of data, yet are read many times be each thread. When coupled with their static unchanging values (they are never written to), they were found to be excellent candidates for storage in the GPU constant memory. Given that the constant memory is not only quicker than global memory, but also cached, storing the material parameters here alleviated many thousand global memory calls, improving performance.

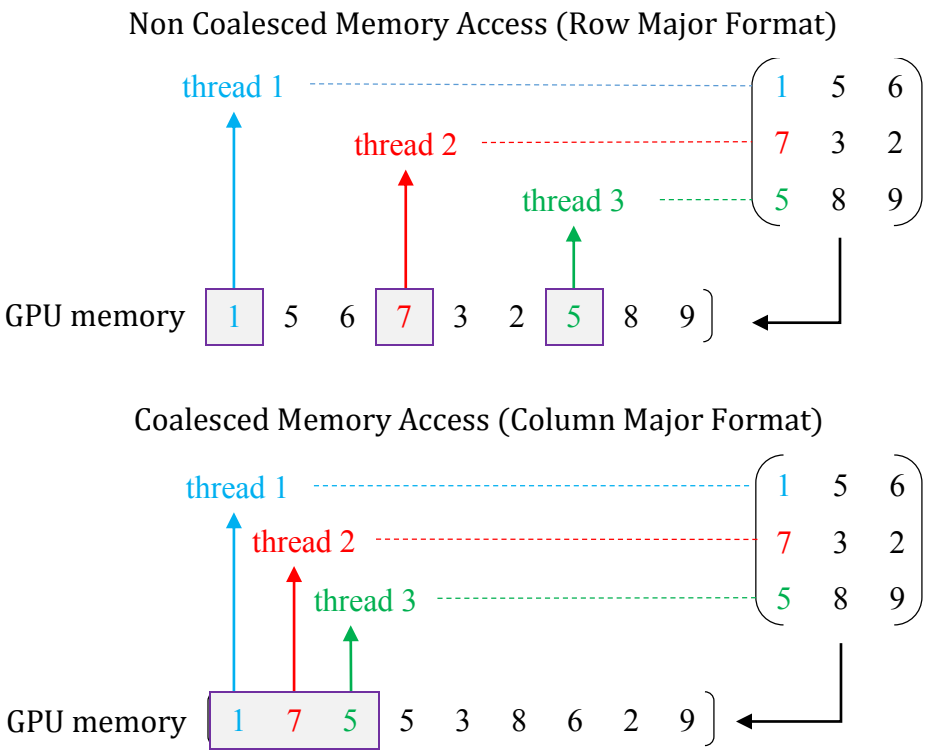

Fig. 10 GPU global memory coalescing dependence on data structure storage format.

The final optimization of interest in moving the dispersive algorithm into a parallel GPU setting was that of GPU SM occupancy. As seen previously in Chapter 3, the structure in which threads are actually executed on the GPU is a hierarchy involving various sub-structures known as grids (collections of blocks), blocks and warps. The most fundamental of these, the warps, execute groups of 32 threads roughly in lock step with one another, according to the SIMT principle. While the number of threads per warp is, in general, not configurable by the programmer, the number of threads contained within a block and likewise number of blocks within a grid, are. Owing to hardware constraints and execution factors, certain configurations of blocks and threads are more efficient than others [25]. For instance, suppose a block is defined to contain 150 threads. When the block is distributed to the various SMs of the GPU, its member threads will be parceled into warps of exactly 32 threads to be executed concurrently. Problematically, however, 150 is not equally divisible by 32, resulting in 4 complete warps of 32 and one partially occupied warp of 22. Given that the number of threads per warp is non-negotiable, this implies that 10 threads are "wasted" during execution, preventing other blocks, warps and threads from doing useful work in their stead.

In the above rather simplistic example, the GPU would be said to be experiencing sub-optimal occupancy, a condition which may be caused by a number of factors in addition to block configuration, such as memory and other resource availabilities [25]. In order to determine if a given program has less than 100% occupancy, NVIDIA has available a CUDA Occupancy Calculator [26], in which the various aspects of a program may be analyzed to yield an occupancy estimate. While 100% occupancy does not necessarily imply ideal or increased efficiency in all cases [19], it is nonetheless a good statistic to strive for, ensuring that the GPU SMs are idle as little as possible.

Having kept all of the above factors in mind, the parallel GPU version of the algorithm was written using a combination of C++ and CUDA, and was methodically tested to ensure proper functioning. With the majority of the programming complete, time trials were then begun to verify the performance of each version, the results of which are thoroughly explored in the following chapter. A final global overview of all the software written for this thesis, and their various interdependencies, is given in Fig. 11.

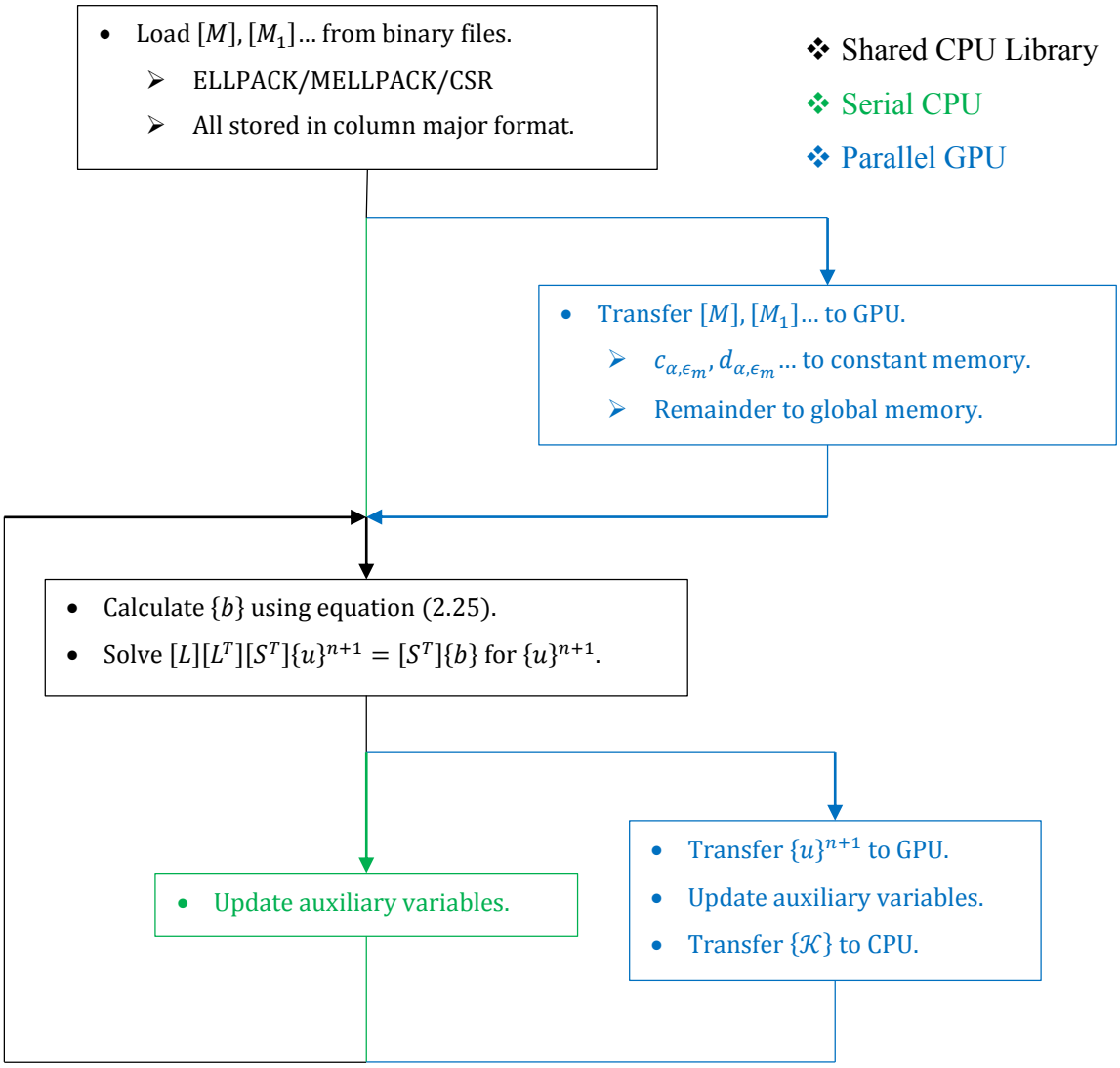

Fig. 11 Global overview of serial CPU and parallel GPU dispersive FETD code.

Chapter 5: Results

5.1 Hardware Specifications and Setup

In order to proceed with the investigation of GPU acceleration in dispersive FETD methods, a test problem needed to be devised to which all of the above software could be applied. Owing to its simplicity, a two dimensional parallel plate waveguide was selected measuring 20 cm by 4 cm, in which the upper and lower plates constituted perfect electric conductors (PEC) and each end of the waveguide was fitted with a first order absorbing boundary condition. The problem domain was meshed with 1st order triangular elements and excited by a differentiated Blackman-Harris pulse at the leftmost edge. All simulations were run for 6000 time steps, with a temporal spacing of $3.34 \times 10^{-12}\ s$.

With the stated goal of the present work being the reduction in overhead execution time required in the treatment of dispersive media, it is important to note that the quantity of overhead is dependent upon several factors specific to the problem under consideration. Rather than fixing these factors and obtaining a single set of results, it was decided to perform a more comprehensive investigation by which the effects of different independent variables could be observed on the performance increase or decrease provided by the GPU. Firstly, it is evident from equation (2.24) that the amount of work required to update the auxiliary variables is dependent on the dispersive order of the material under consideration, p. Consequently, two different materials of different orders were selected, so as to observe any effect on performance. Specifically, a medium with two pairs of Lorentz poles was selected as a 4th order material, whose permittivity's frequency dependence is expressed in Fourier space as

$$\epsilon(\omega) = \epsilon_\infty + G_{e_1}\frac{(\epsilon_s - \epsilon_\infty)\omega_{e_1}^2}{-\omega^2 + 2\delta_{e_1} j\omega + \omega_{e_1}^2} + G_{e_2}\frac{(\epsilon_s - \epsilon_\infty)\omega_{e_2}^2}{-\omega^2 + 2\delta_{e_2} j\omega + \omega_{e_2}^2} \tag{5.1}$$

where the material parameters were selected as follows in the electrically dispersive case

$$\begin{gathered} G_{e_1} = 0.2, G_{e_2} = 0.4, \omega_{e_1} = 3.1\pi \times 10^9, \omega_{e_2} = 2.2\pi \times 10^9 \\ \delta_{e_1} = \frac{0.05}{2}\omega_{e_1}, \delta_{e_2} = \frac{0.02}{2}\omega_{e_2}, \epsilon_s = 5.2\epsilon_0, \epsilon_\infty = 3.1\epsilon_0. \end{gathered} \tag{5.2}$$

For the magnetically dispersive case, the permeability's frequency dependence was identical to that in (5.1), but with the following parameters used instead

$$G_{m_1} = 0.9, G_{m_2} = 0.5, \omega_{m_1} = 3.3\pi \times 10^9, \omega_{m_2} = 4.2\pi \times 10^9$$
$$\delta_{m_1} = \frac{0.06}{2}\omega_{m_1}, \delta_{m_2} = \frac{0.03}{2}\omega_{m_2}, \mu_s = 3.7\mu_0, \mu_\infty = 1.8\mu_0. \tag{5.3}$$

This was compared to a media with a single Debye pole, exhibiting first order dispersion governed by:

$$\epsilon(\omega) = \epsilon_\infty + \frac{\Delta\epsilon}{1 + \tau_e j\omega} \tag{5.4}$$

where once again the material parameters for the electrically dispersive case are

$$\epsilon_\infty = 2.4\epsilon_0, \Delta\epsilon = 4.89317\epsilon_0, \tau_e = 10^{-11} \tag{5.5}$$

and for the magnetically dispersive case

$$\mu_\infty = 1.10423\mu_0, \Delta\mu = 3.2\mu_0, \tau_m = 6 \times 10^{-12}. \tag{5.6}$$

In both the 1st and 4th order cases, the material parameters were adopted from [10] and while theoretically possible, do not necessarily represent known real world materials. In addition to the order of dispersion, whether the material is doubly or singly dispersive was also taken into account, since the latter requires approximately twice the computational workload of the former.

Secondly, the amount of dispersive material present within a given domain equally alters the workload required, since from Section 3.4 it is noted that the sizes of the $[T_{\epsilon_m}]$ and $[S_{\epsilon_m}]$ matrices depends on the number of dispersive elements present. Here, only one kind of the above dispersive materials was used at a time, but the amount varied between 25% and 90% of the total volume of the domain with the remainder being free space, as depicted in Fig. 12. While the material distribution within the geometry of Fig. 12 is very simple, more complex distributions would not be hypothesized to alter the workload significantly, and so were not considered. Indeed, geometry would be expected to impact performance only insomuch as two disjoint elements represents six degrees of freedom, whereas two conjoined elements have five. However, unless

dealing with odd distributions in which a majority of individual elements are disjoint, the effect is expected to be minimal.

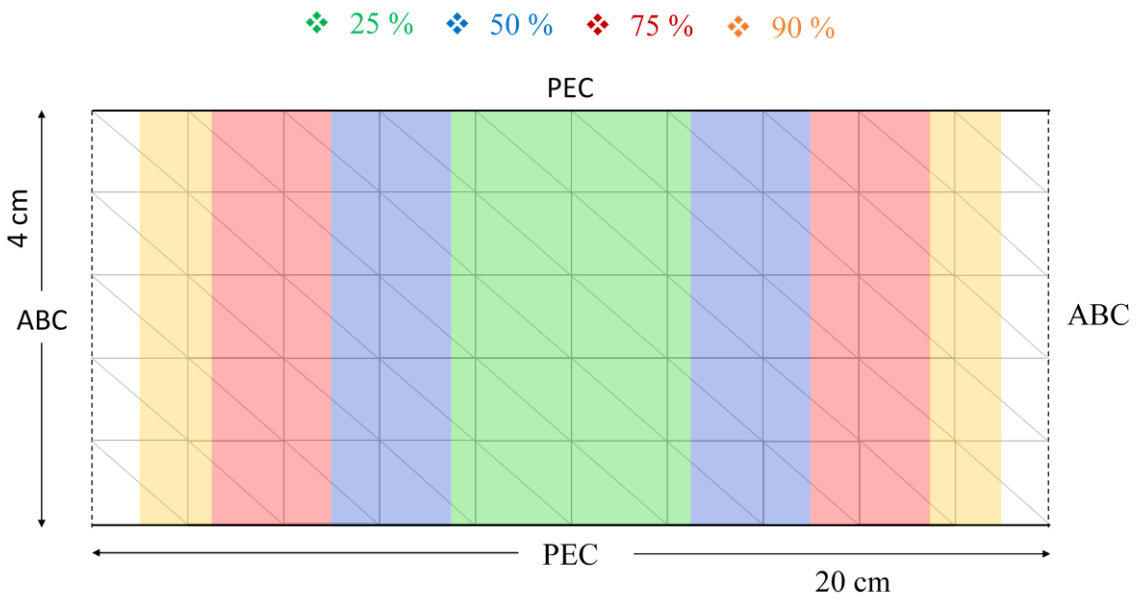

Fig. 12 2D parallel plate waveguide filled with varying amounts of dispersive media (not to scale).

Thirdly, the various operations performed in the course of solving do not all scale identically with number of variables, and as a result the association between GPU performance and number of degrees of freedom (D.o.F, i.e. the granularity of the discretization) was also considered. Lastly, as indicated in Fig. 1, there is an imbalance in performance when considering single versus double precision floating point arithmetic, for both CPUs and GPUs. Furthermore, it was discovered during verification of the parallel GPU code that, depending on the length and size of the problem at hand, floating point errors may accumulate too quickly to obtain a meaningful answer if single precision arithmetic is employed, despite the potential increase in performance. For these reasons, it was decided to also measure and compare the performance of the GPU as a function of the floating point precision used.

Having selected the above five parameters to act as independent variables within the following time trials, the last factor to consider was the hardware upon which the above software would run. Rather than selecting a single computer with which to perform all trials, here two different machines were used in order to investigate any additional effects on performance and behaviour. The first was a typical consumer device, a Toshiba laptop equipped with an Intel Core

i7 CPU clocked at 1.73 GHz, with 4 GB RAM and running 64 bit Windows 7 Home Premium Edition (Service Pack 1). The GPU for this device was the main mobile display driver, an NVIDIA GeForce 310M graphics card clocked at 1.468 GHz with 512 MB dedicated video memory, 16 CUDA cores, and CUDA compute capability 1.2.

The second piece of hardware used was the McGill High Performance Computing (HPC) cluster, also known as "Guillimin" [27]. This project's serial code was executed on one of Guillimin's 216 SW2 nodes equipped with dual Intel Sandy Bridge EP E5-2670 processors, clocked at 2.6 GHz with 4 GB of RAM per core (8 cores per processor). The parallel code was run on one of Guillimin's AW accelerator nodes, containing the same Intel processors, as well as dedicated dual NVIDIA K20 graphics cards clocked at 706 MHz, with 5 GB dedicated video memory, 2496 CUDA cores and CUDA compute capability 3.5. Contrary to the personal computer, Guillimin runs CentOS 6.5 (a Linux distribution) on each of its login nodes (nodes with which the user interacts with directly), however the worker nodes upon which the code was run are fully dedicated to computation.

Due to time constraints (as well in some cases hardware constraints), not every possible combination of the previously identified independent variables was run together and on each of the above pieces of hardware. The combinations which were tested and recorded are presented in the following section.

5.2 Measuring the Dispersive Overhead

All data collected during the course of testing is now presented and discussed in detail, in order to determine the impact of GPU utilization in dispersive media, and to elucidate areas in which further improvement may be made.

As a starting point, the importance of including dispersive effects in numerical models is reiterated and demonstrated in Fig. 13, for which a comparison of results obtained by the solver can be seen for both dispersive and non-dispersive media. These traces were produced using the above sample waveguide problem run for 1000 time steps, with a 5 cm dielectric, and field values being recorded at the leftmost absorbing boundary. In the non-dispersive case, for which $\epsilon = 4.6\epsilon_0$ and $\mu = 1.8\mu_0$ the initial pulse is clearly seen, as well as two clean and mostly undistorted reflections from the first and second interfaces. In the dispersive case however, in which the 4th

order doubly dispersive model of (5.1) has been used, the pulse is drastically distorted upon reflection, as each frequency component propagates at different speeds within the medium, resulting in a substantially different field measurement. The importance of incorporating dispersion (and therefore also the performance of dispersive methods) should therefore not be underestimated.

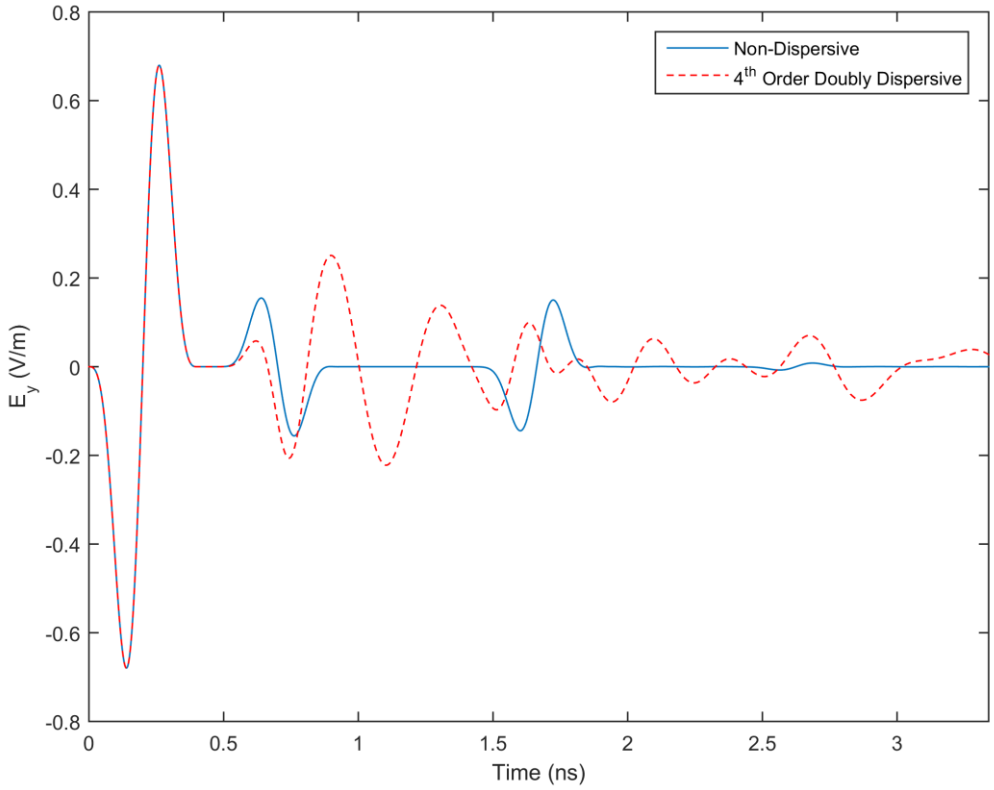

Fig. 13 Differences in reflected pulse resulting from dispersion, highlighting the importance of accurately modelling frequency dependent materials.

Having now further motivated the need for efficient dispersive solvers, it was important to also establish exactly how costly the inclusion of dispersion is within an FETD simulation. If a miniscule proportion of computation time is actually spent updating the auxiliary variables, then even the largest speed up on the part of the GPU will lead to marginal improvement in overall performance. In other words, there was not a large difference in workload between the dispersive and non-dispersive treatments to begin with. As such, attention is first turned to Fig. 14, in which the proportion of time spent in the various stages of the serial CPU algorithm for a 4^{th} order, doubly dispersive, single precision simulation run on the laptop computer is shown. Each simulation was run ten times for each of the investigated combinations of variables and the presented data was computed using the averages of each set. Additionally, a breakdown is also presented in Fig. 15

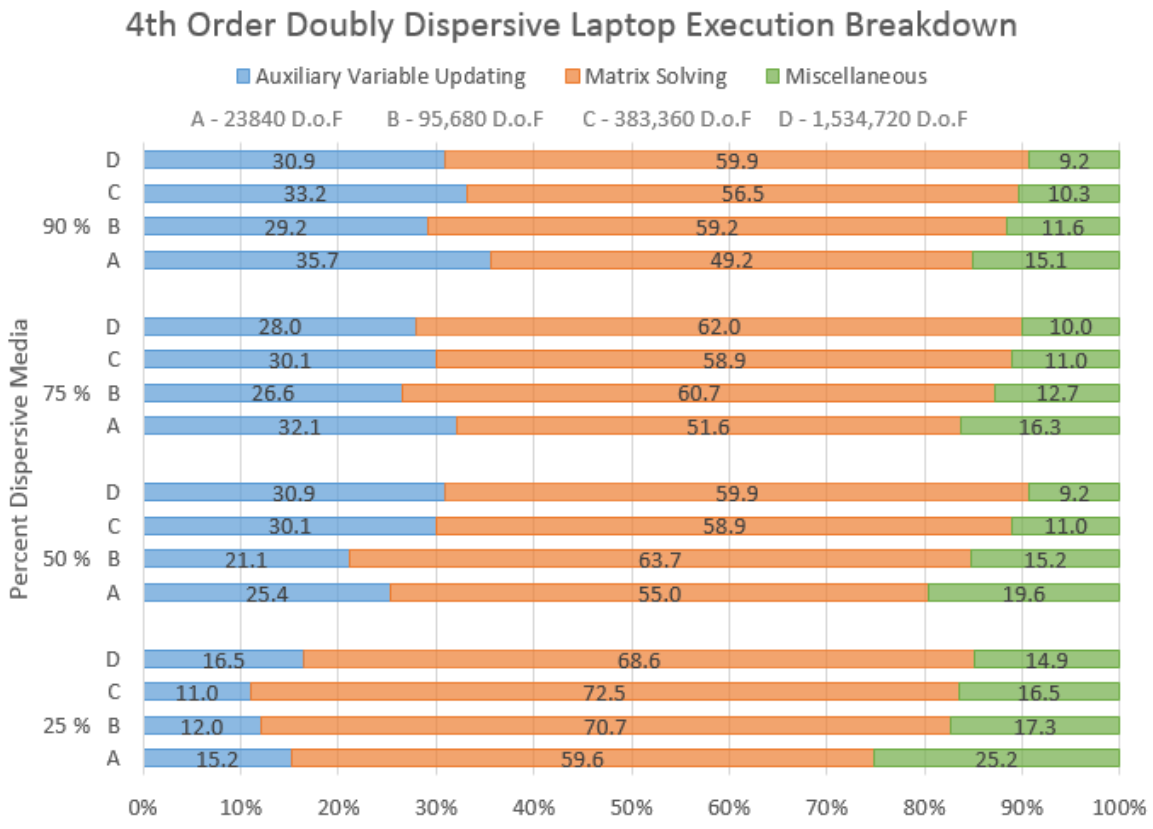

Fig. 14 Single precision serial CPU computation analysis for a 4th order doubly dispersive medium.

for a serial CPU, 1st order, singly dispersive, single precision simulation run on the laptop computer. It was found that the breakdowns for double precision simulations mirror those for single precision, while those for other combinations of independent variables lie between the extremes of Fig. 14 and Fig. 15, and so have been excluded for brevity.

Observing these breakdowns, many interesting conclusions may be drawn about the inclusion of dispersion within FETD simulations. Firstly, there is a strong dependence of the overhead upon the type of dispersive material (1st vs 4th order) under investigation and, secondly, there is also a general correlation between the fraction of the volume occupied by the dispersive material and the amount of time spent updating auxiliary variables. Both of these results are to be expected, given that in either case it is recalled that material parameters and proportion of dispersive material influence the amount of overhead required. If comparing across problems with the same number of degrees of freedom, the cost of matrix solving thereby remains approximately fixed while the cost of updating increases, resulting in the above trends. When comparing solely across degrees of freedom, a discernable trend is less apparent, however it is notable that the

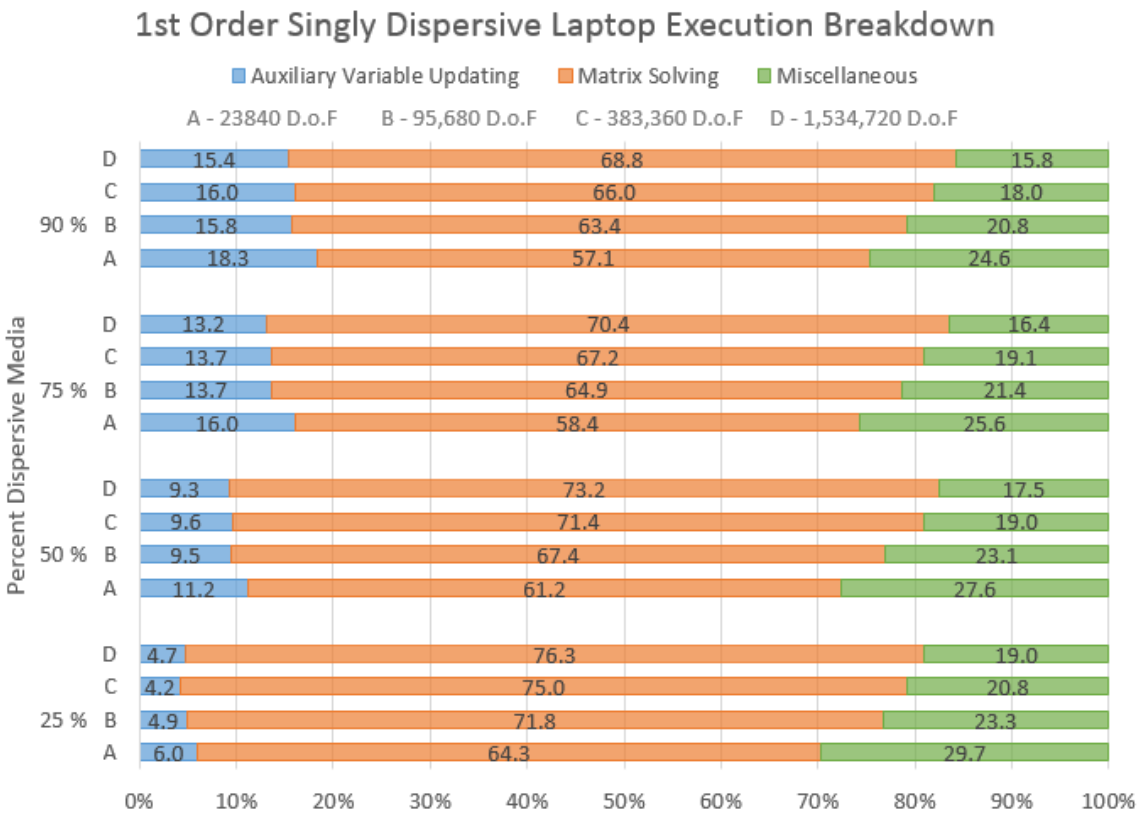

Fig. 15 Single precision serial CPU computation analysis for 1st order singly dispersive medium.

variance observed between different sized problems in 4th order doubly dispersive media is larger than in the 1st order singly dispersive case. This is possibly attributed to the former incurring greater variability during execution due to its longer run time, since the laptop used was running an operating system at the time. Indeed, when comparing the same data collected on Guillimin, in which the worker node is not running other processes, the spread is less pronounced.

From these analyses, it is clear that the greatest benefit to be gained via parallelization is in high order, doubly dispersive problems for which a significant amount of dispersive material is present. However, it is important to note that in all cases the maximum possible overall performance increase is approximately bounded by Amdahl's law [19]. Since only a fraction of the overall algorithm has been parallelized, the maximum achievable overall performance must asymptotically approach a limit. In the present case, for example, those portions of the algorithm shared by both the dispersive and non-dispersive algorithms will always take the same fixed amount of time, as they are not parallelized, and so performance must plateau. If P represents the fraction of the algorithm that is being parallelized and S_p represents the speed up given to this

portion by the GPU (i.e. if the GPU runs this section twice as fast as the CPU, $S_p = 2$) then Amdahl's law may be mathematically expressed as

$$S_{tot} = \frac{1}{(1-P) + \frac{P}{S_p}} \tag{5.7}$$

in which S_{tot} is the speedup given to the whole program. The highest value for P taken from the above is approximately 35.7 % and the lowest value is about 4.2 %. In order to calculate the asymptotic best case overall speed-up (i.e. performance on par with non-dispersive methods) the limit as S_p tends to infinity is taken, resulting in 1.56 and 1.04 times speedups. Again, these numbers serve to illustrate that in some cases the dispersive overhead can be significant while in others it is less of a burden. Whether the forthcoming results are worthwhile to implement depends partly, therefore, on the specific problem under consideration, as problems containing little dispersive material with relatively simple frequency dependence do not differ greatly in performance from non-dispersive media.

5.3 GPU Speedup Data and Discussion

Having motivated, in some cases, the need for parallelization in dispersive FETD methods through a comparison of time spent in overhead operations, attention is now shifted toward the main results of the present work. Having noted that the overall performance increase is fundamentally limited by the nature of the current investigation, the metric of choice for comparing the GPU and CPU performance in updating the auxiliary variables was selected to be the ratio S_p, defined as follows:

$$S_p = \frac{\text{Time Updating Auxiliary Variables on CPU}}{\text{Total Time on GPU}} \tag{5.8}$$

Here, "Total Time on GPU" includes the overhead required to transfer data to and from the GPU as well as time spent updating. Each simulation of interest was run ten times and the results averaged, with error being calculated as plus or minus one standard deviation from the mean. As a result, Fig. 16 is presented below in which a doubly dispersive, single precision problem has been executed on the laptop computer, with dispersive order, degrees of freedom and fraction of dispersive material serving as independent variables. Additionally, Fig. 17 demonstrates

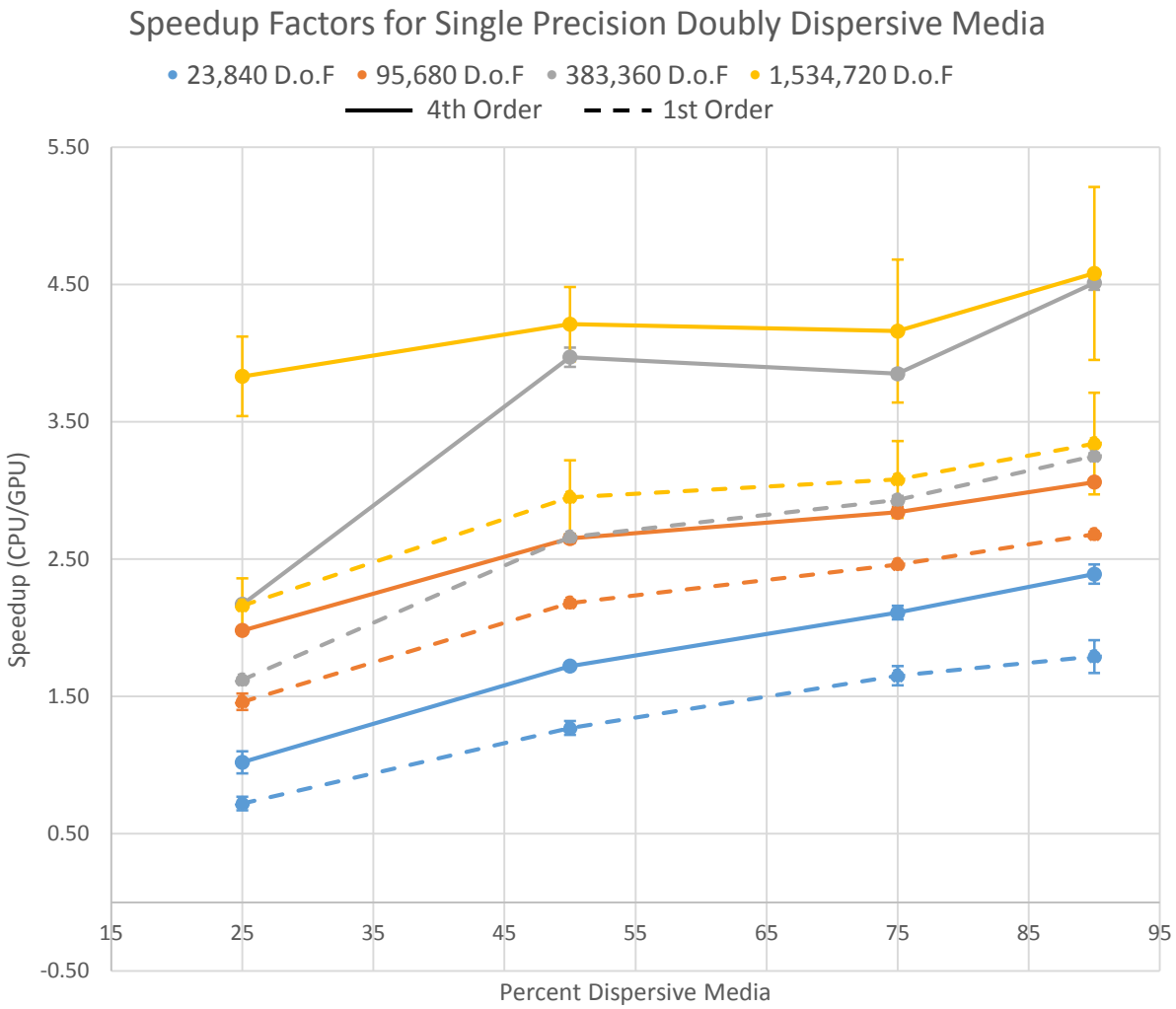

Fig. 16 GPU speedup as a function of problem size, dispersive order and amount of dispersive material, for a doubly dispersive problem executed in single precision on a laptop computer.

performance as a function of the same parameters, however with singly dispersive media. Lastly, Fig. 18 represents results obtained for a doubly dispersive, 4^{th} order problem executed on Guillimin, for which floating point precision, degrees of freedom and fraction of dispersive media serve as independent variables.

At first glance, the overall performance of the GPU is good, with a peak performance on the laptop of nearly 4.87 times faster, and with Guillimin's peak performance understandably better at about 9.66 times faster. Despite the majority of data points demonstrating improved performance, there are nonetheless several instances of speedups less than unity, notably concentrated around smaller problems with relatively little dispersive media. Upon further

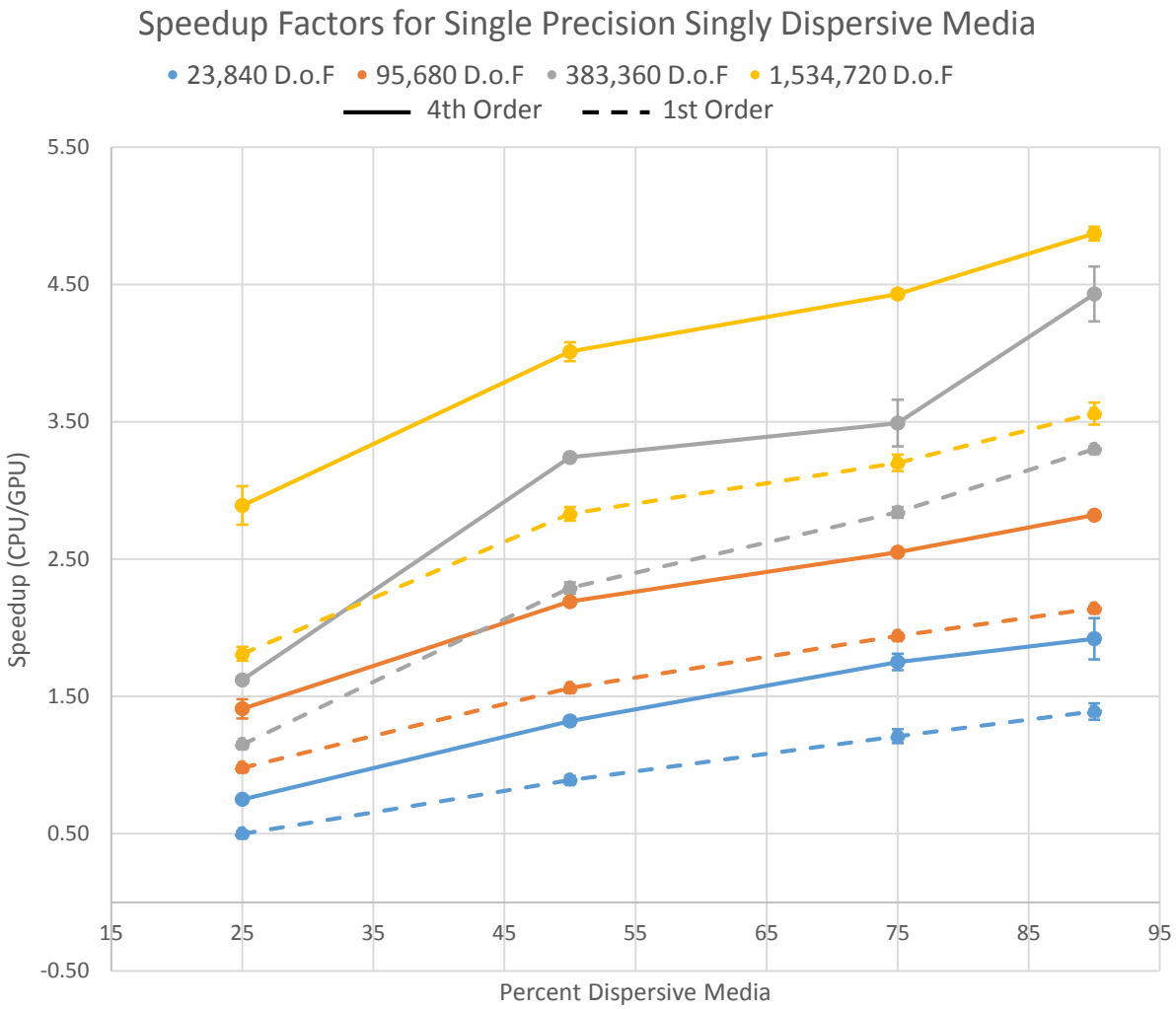

Fig. 17 GPU speedup as a function of problem size, dispersive order and amount of dispersive material, for a singly dispersive problem executed in single precision on a laptop computer.

inspection, several interesting trends can also be found in the presented datasets. Comparing all three graphs at once, it is clear that as the proportion of dispersive material within the problem volume increases, so too does the effectiveness of the GPU. Additionally, in general it is evident that GPU performance also increases with the number of degrees of freedom. Comparing specifically Fig. 16 and Fig. 17, the fact that materials with higher dispersive orders tend to experience better performance is also observed, as well as doubly dispersive media somewhat out performing singly dispersive in general. Lastly, in Fig. 18, problems being executed with single floating point precision tend to perform better than their double precision counterparts.

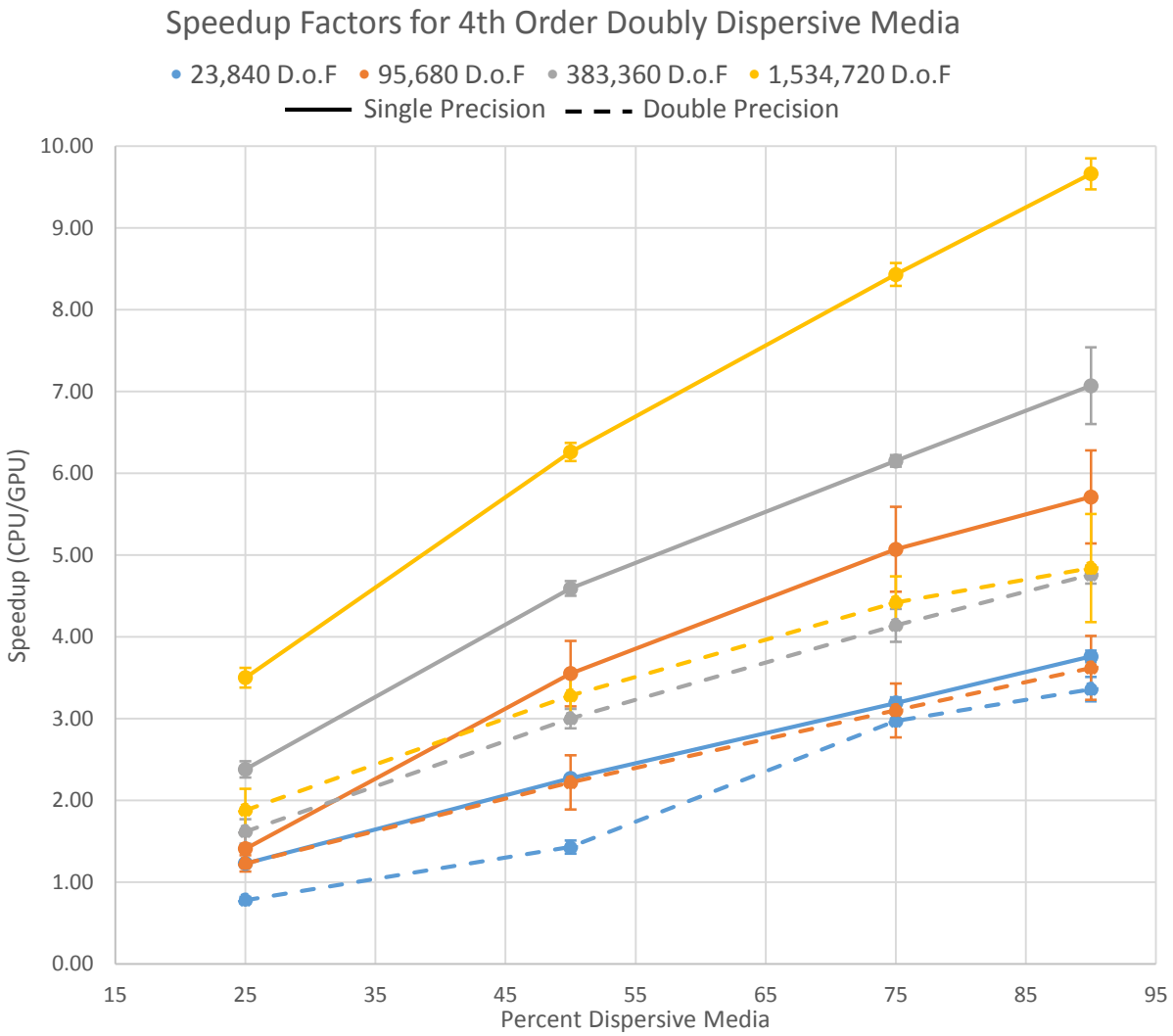

Fig. 18 GPU speedup as a function of problem size, floating point precision and amount of dispersive material, for a doubly dispersive 4th order problem on Guillimin.

There exists, in essence, two fundamental hypotheses giving rise to these many observed trends, either shared or unique, between each of the traces in Fig. 16 through Fig. 18. The first is that of workload. The effectiveness of the GPU is strongly tied, of course, to the parallelizability of the problem at hand. If the number of threads required is very small, the GPU will be hard pressed to outperform its CPU counterpart. However, as the number of required threads grows, the true strength of the GPU may begin to be utilized. With each additional new thread running concurrently, the computational throughput of the whole system is increased. It is therefore expected that as the number of threads needed grows, so too does performance, up until the maximum number of running threads is reached. Moreover, it is also extremely favourable to have

more threads so that the GPU may better hide latencies associated with certain operations [25]. For example, while one thread waits for data to be fetched from main memory, another thread may be swapped in and run, thereby potentially increasing performance even if the aforementioned plateau has been attained. Lastly, the amount of work each thread is given may also have an impact on performance. If a thread accomplishes very little during execution, the overhead required within the GPU for scheduling and resource allocation may reduce performance, in addition to the CPU simply being able to run through many short serial calculations quicker. Furthermore, having more independent operations within each individual thread has the possibility to increase instruction-level parallelism (ILP). Similar to hiding the latency associated with memory access, ILP allows the GPU to cover the latency associated with more fine-grained instructions. For example, when computing the product $(a + b) \times (c + d)$, each of the summations may be computed simultaneously before the final product is formed, further saving time [18].

The second is that of GPU memory transfer overheads. Recalling that the CPU and GPU are forced to transfer data back and forth in the present implementation, the effectiveness of the GPU can be heavily influenced by the volume of data being moved. If the time taken to transfer $\{u\}^{n+1}$ and $\{\mathcal{K}\}$ is longer than the time saved via parallelization, the GPU performance will naturally suffer under the burden of this overhead and will experience speedups less than one (slow downs). The fraction of time spent updating versus transferring can therefore play a critical role, as minimizing time wasted to GPU transfer overhead will naturally lead to increased performance.

With these mechanisms in mind, plausible explanations for the behaviours previously identified in Fig. 16 through Fig. 18 are readily available. As the proportion of dispersive media is increased, both mechanisms take effect. In the first case, more dispersive material results in larger matrices and vector operations, increasing the number of threads and overall workload. Secondly, the sizes of the $\{u\}^{n+1}$ and $\{\mathcal{K}\}$ vectors are dictated only by the degrees of freedom and as a result, with increasing amounts of dispersive material more time is spent performing useful work while the transferring overhead remains fixed, improving efficiency.

Similar arguments can be made for the singly/doubly dispersive, degrees of freedom and dispersive order variables. Doubly dispersive materials require twice the auxiliary variables of singly dispersive ones, resulting in more work per thread, but the same amount of transfer overhead. Likewise, higher order dispersive elements equally require more auxiliary variables and

work, for the same amount of data transfer as their lower order counterparts. Furthermore, as the number of degrees of freedom is increased the number of required threads and workload also increases. This may also result in an improved ratio of useful work to transfer overhead, as the computational workload scales with the number of variables faster than memory transfers do (the latter being approximately linear). Lastly, given the data presented in Fig. 1 as well as the fact that double precision floating point numbers require approximately twice as long to transfer as single precision numbers, the higher speedup factors for single precision simulations is not surprising.

It is worth noting that while the serial and parallel algorithms both made use of the same shared libraries, variations were observed in the duration of these common subroutines (primarily on the laptop) depending on whether or not the GPU was used. That is to say that despite executing the exact same code, operations such as matrix solving took slightly different amounts of time depending on whether or not the update equations were run on the CPU or GPU. On the laptop, the discrepancy was found to be approximately 5.49% on average while on Guillimin the effect is essentially non-existent, at 0.67%.

Ideally, since both versions are performing the exact same operations outside of auxiliary variable updating, the use of a GPU versus CPU should have no bearing on performance of the matrix solving subroutine. In the case of the laptop however, the 5.49% difference may cause some simulations to slow down, with longer solving times eclipsing GPU speedup. This observation is not, however, of concern as far as the present investigation is concerned. The discrepancy is most likely attributable to the specific hardware/software setup that was used on the laptop, since Guillimin did not suffer from this issue. These differences could therefore potentially be alleviated in the case of the laptop by tweaking the circumstances under which the code is executed and/or software/hardware alterations. Moreover, it is not certain that the above tweaks would have any effect on the auxiliary variable updating, meaning the above results would not change in either case. Lastly, even if there were to be an effect, most cases saw the CPU matrix solve run faster than the GPU, indicating that the data presented above would represent lower bounds on performance increase, demonstrating viability of the method nonetheless.

5.4 GPU Transfer Overhead

A pivotal factor in explaining some of the results of Fig. 16 through Fig. 18 relied upon the need to transfer information between the CPU and GPU upon each iteration of problem solving.

Indeed, in many cases this extra overhead of data transfer required to utilize the GPU can itself prove debilitating, as evidenced in the above results in which the speedup was less than unity. In such cases, as mentioned, the cost of having to transfer between host and GPU memory actually exceeded the benefit gained through parallelization. This of course is intimately tied to the discussion presented in Section 5.2, as small problems with relatively little dispersive materials not only stand little to gain, but may actually perform worse.

With the burden of memory transfers hypothesized to play a key role in the performance trends found above, a more detailed breakdown of time spent on the GPU is now presented in order to both substantiate some of the above analyses, and propose ways to improve efficiency in the future. As a starting point, Fig. 19 demonstrates the breakdown of time spent on the GPU performing computations (the kernel) versus transferring data (cudaMemCpy), as a function of the percent of dispersive media present in the volume (corresponding to the solid orange trace of Fig. 16).

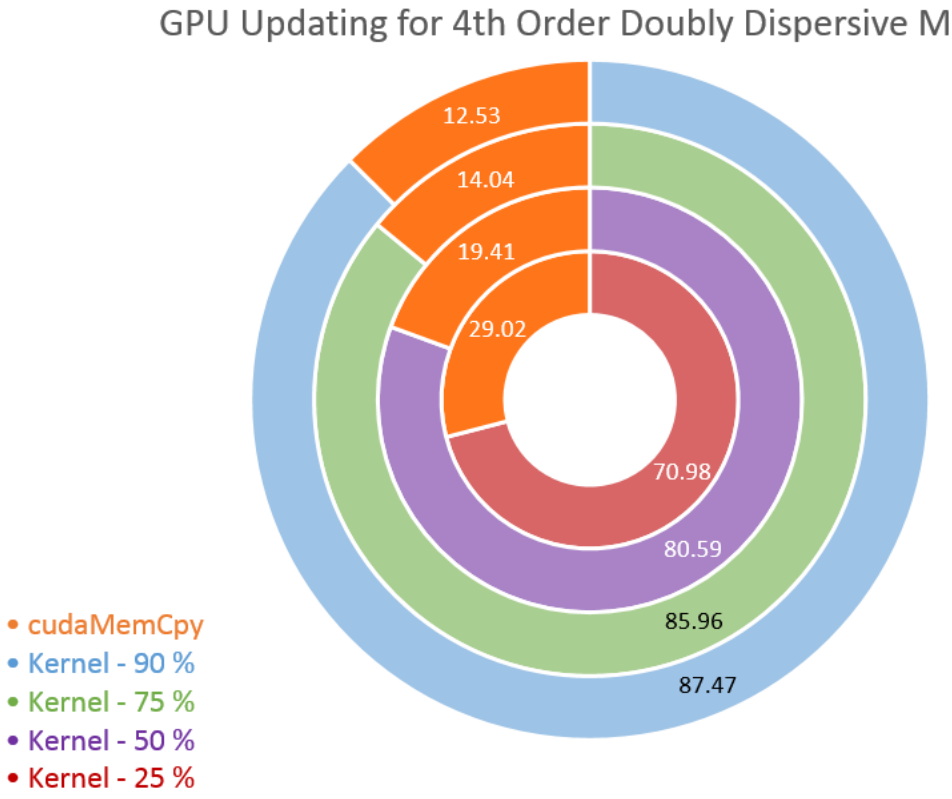

Fig. 19 Breakdown of time spent on the GPU as a function of the amount of dispersive material present, for a doubly dispersive, 4th order problem with 95,680 D.o.F executed in single precision on the laptop.

Here, the mechanisms proposed previously gain support, as a clear decrease in the amount of time spent performing GPU overhead is observed with increasing dispersive presence. Furthermore, Fig. 20 is presented below in which the same breakdown is presented, but this time as a function of degrees of freedom (corresponding to the first point of each solid trace of Fig. 16). Once again an overall decreasing trend in the amount of time spent transferring data to and from the GPU is found, with the notable exception of the outermost ring. This outlier is possibly explained by noting that the assumed one-to-one scaling of memory transfer time to degrees of freedom is only approximate.

Lastly, attention is turned toward a breakdown of GPU time for those simulations run on Guillimin. Fig. 21 demonstrates the partitioning of GPU time as a function of the amount of dispersive material present, for a 4^{th} order doubly dispersive single precision computation (corresponding to the solid orange line of Fig. 18). Here now a drastic deviation from the previous two figures is observed. Of primary importance is the observation that a very large majority of GPU time is now spent transferring data, as opposed to actually dealing with the dispersive

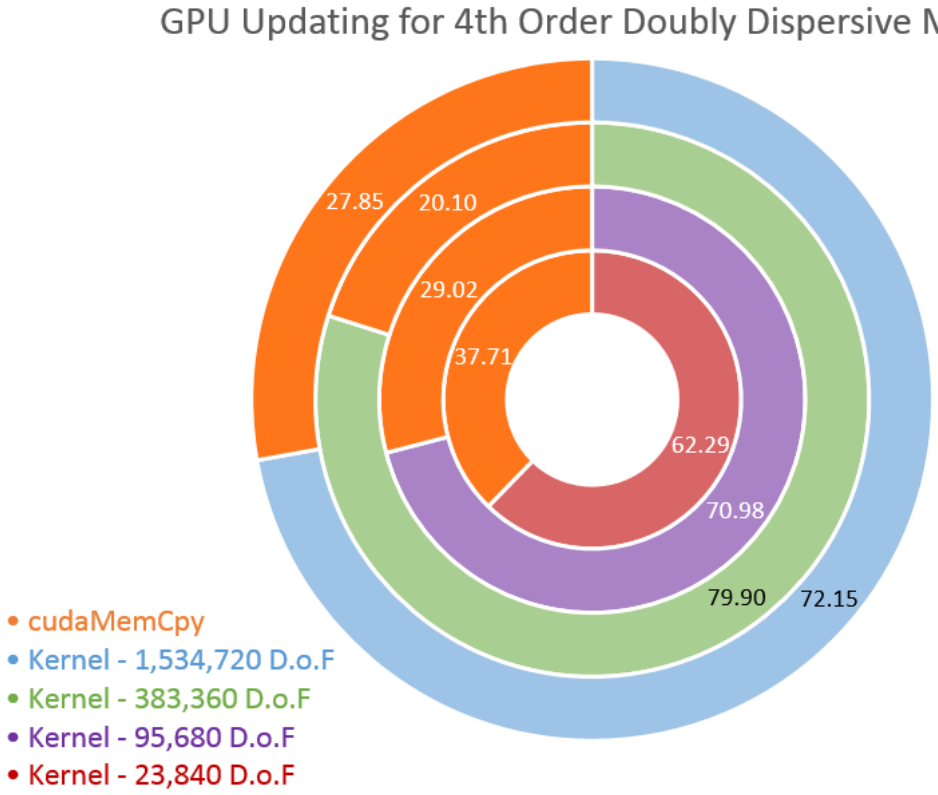

Fig. 20 Breakdown of time spent on the GPU as a function of the number of degrees of freedom, for a doubly dispersive, 4^{th} order problem filled to 25% capacity, executed in single precision on the laptop.

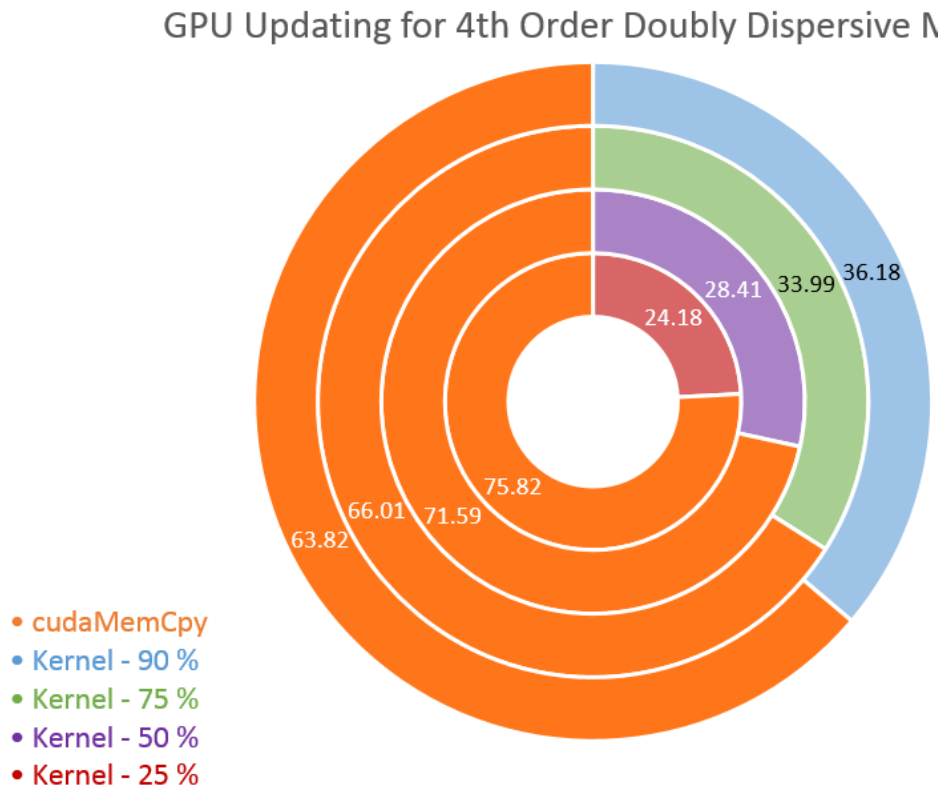

Fig. 21 Breakdown of time spent on the GPU as a function of dispersive material present, for a doubly dispersive, 4th order problem with 95,680 D.o.F executed in single precision on Guillimin.

overhead. In other words, the high powered GPUs available on Guillimin execute the main kernel so quickly that more time is spent in data transfers than actual computations. The overhead does however still follow the previously noted trend, decreasing with increased dispersive presence, yet even in the 90% case it represents more than half of computation time. To further demonstrate the significance of data transfers in the previous results, Fig. 16 and Fig. 18 have been recreated below in Fig. 22 and Fig. 23, respectively, but have had the speedup metric altered so as to exclude time spent transferring data back and forth between the CPU and GPU. The resulting plots hence measure strictly differences in computation time.

As a result, both figures experience bumps in performance, with exceptional increases being observed in the case of the Guillimin data. These results are of course expected given the presented breakdowns, with transfer overheads being particularly debilitating in the case of the HPC. While the increase in performance experienced by the laptop executed code is non-negligible, the almost three fold improvement seen in the peak values of the Guillimin results strongly suggest searching for an implementation in which data transfers are further minimized or altogether omitted. With the removal of transfer overheads, many of the previously identified

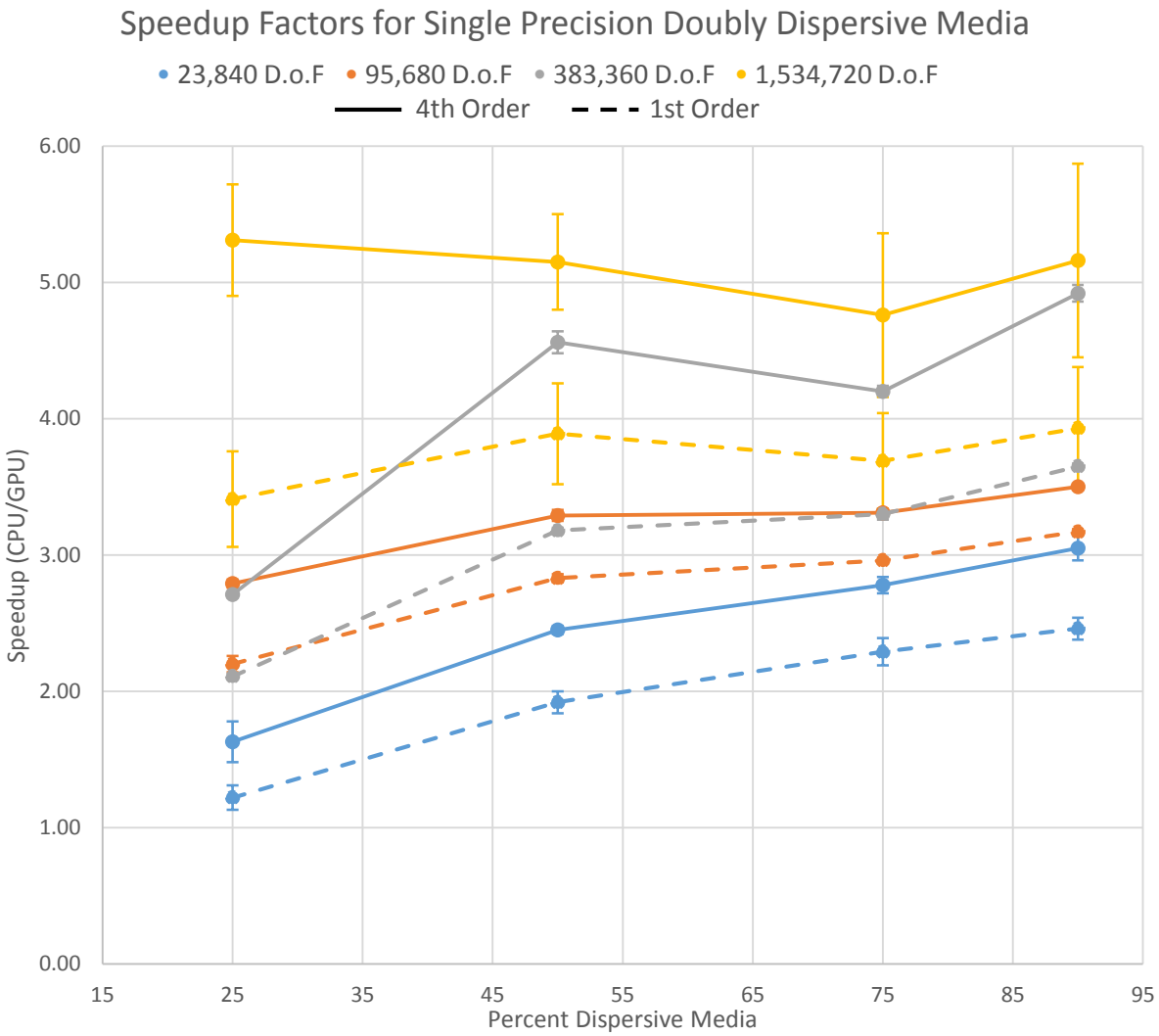

Fig. 22 Recreation of Fig. 16 in which the performance metric has been altered to exclude memory transfer times.

trends remain, albeit in a less pronounced fashion, being chiefly motivated now by GPU utilization and latency hiding.

Since the update equations for the auxiliary variables cannot fundamentally be performed without knowledge of $\{u\}^{n+1}$, which is of fixed size, a reduction in transfer times using the present method is unlikely. While certain software/hardware optimizations may be possible (such as the use of special pinned memory on the host [25]), these techniques would be hard pressed to deliver the same performance boost as observed in Fig. 22 and Fig. 23. Given these factors, the optimal choice of course is the elimination of GPU data transfers altogether.

To accomplish this, there cannot be any disconnect between the memory in which $\{u\}^{n+1}$ is computed and that in which the auxiliary variables are updated. The most straightforward and

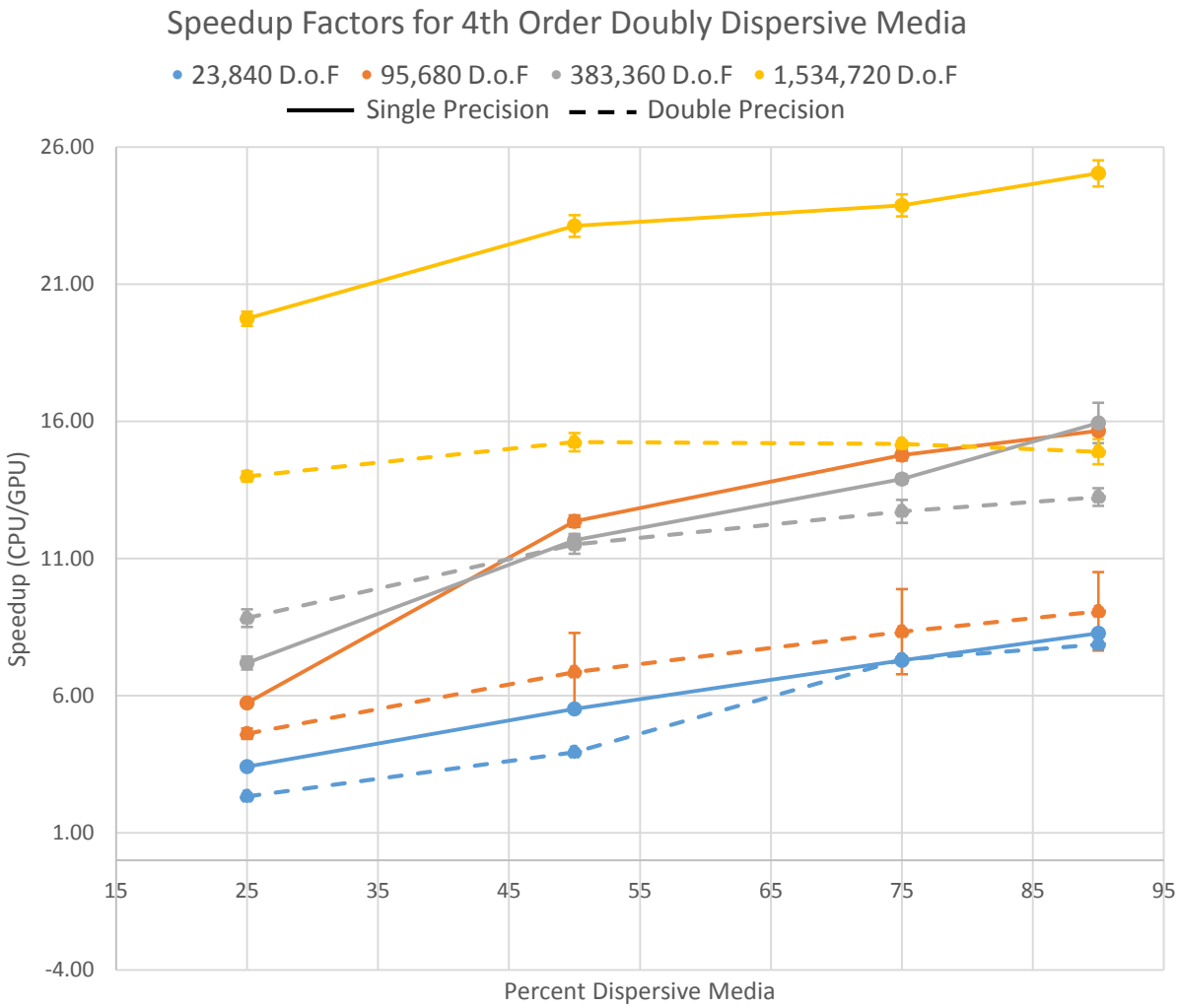

Fig. 23 Recreation of Fig. 18 in which the performance metric has been altered to exclude memory transfer times.

efficient way to accomplish this is to combine the present treatment with an existing parallelization of the base (non-dispersive) FETD algorithm. In doing so, matrix solving and the remaining miscellaneous operations are treated by the GPU, and so all quantities of interest reside in GPU global memory and stay there. The execution of dispersive overhead on the GPU would thereby no longer require any memory transfers from the host (except for those required to initialize the problem), resulting in the peak performances reported above, narrowing the performance gap between dispersive and non-dispersive methods to a minimum.

Chapter 6: Conclusion

6.1 Summary

A thorough investigation into the acceleration of dispersive electromagnetics simulations has been presented. While the use of the more versatile z-transform FETD method has led to some improved efficiency, the overhead associated with dispersive media was nonetheless found to still constitute, in many cases, a significant burden as compared to non-dispersive problems. This thesis was therefore primarily concerned with accelerating this overhead in the hopes of narrowing the performance gap between dispersive and non-dispersive FETD methods, through the use of GPUs and mass parallelization.

The GPU was introduced as a massively parallel computational engine, adhering to a fundamentally different design philosophy from the CPU. With a complex memory and execution hierarchy, the GPU excels at performing many thousands of floating point operations at once, with best performance found to occur when these operations obeyed the SIMT and independence principles. Additionally, the exploitation of the GPU hardware was best found to be via the CUDA programming language, alleviating much of the tedium and restrictions associated with lower level GPU programming.

The exact nature of the dispersive overhead was found to consist of a series of update equations for a set of auxiliary variables, composed entirely of elementary linear algebra operations. These operations were found to be excellent candidates for parallelization, owing to their adherence to the SIMT and independence principles put forth previously.

The impact of using a GPU in dealing with the dispersive overhead was measured through the use of two separate versions of code, identical in all respects except for their handling of the auxiliary variable update equations. The control version made use of the CPU throughout, while the version under test made use of the CPU and GPU. CUDA and C++ were the programming languages of choice, with many custom routines and shared libraries being used to ensure a fair comparison in execution times. Many optimizations in CUDA were also discussed and implemented in order to get good performance on the part of the GPU.

Measuring the speedup afforded to the overhead by the GPU as a function of several simulation parameters yielded excellent results, with the GPU showing good performance boosts in most cases. The peak speedup (including memory transfer overhead associated with the use of the GPU) seen was 4.87 times for a personal computer and 9.66 times for a high performance cluster. Excluding transfer overheads, these improvements were found to jump to 5.31 and 25.04 times faster, respectively, demonstrating the burden these transfers pose in the solution process.

6.2 Future Work

The results presented herein represent a good initial foray into parallelization of dispersive FETD z-transform methods. Nevertheless, additional optimizations and techniques may be applied in the future to further boost the performance of the GPU.

Some of the optimizations which were discussed in the above work were not implemented within this project, either due to hardware constraints, time constraints or the hypothesis that they would not yield sufficiently large boosts in performance to be worthwhile. However, each of these methods in the future may be fully implemented and fine-tuned to verify this assumption and yield truly optimal performance. For example, despite the sparse data structures used, caching data in shared memory during matrix-vector multiplication might nonetheless alleviate some global memory requests. Using pinned or specially allocated memory on the host machine might improve performance when transferring data to and from the GPU. Lastly, a more careful arrangement of the CUDA code (such as loop unrolling) may result in improved performance and utilization through instruction-level parallelism.

Small performance tweaks aside however, three major additional avenues of exploration exist for further boosting performance. The first, as previously discussed, is the parallelization of not just the dispersive overhead, but the whole FETD algorithm. In doing so, the need to transfer data back and forth between GPU and CPU upon each iteration is alleviated to yield significant speed boosts. Secondly, the current investigation made use of only one CPU and one GPU at a time, however many resources exist today with clusters of each. Guillimin, for example, is a high performance cluster and has many computational nodes each with two high-powered GPUs. Rather than restricting execution to one GPU, it may be possible to further subdivide the workload such that two GPUs work on the problem concurrently. This would yield an additional tier of parallelism and a massive boost in computational throughput, with even higher speedup factors expected.

Lastly, if a “fair” comparison is no longer the goal but rather sheer speed, the use of existing libraries coded specifically for CUDA capable GPUs (such as cuBLAS for linear algebra [28]) have the potential to further increase performance, having been fine-tuned for optimality.

References

[1] A. Taflove and S. C. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, 3rd ed., Norwood, MA: Artech House, 2005.

[2] J. Jin, *The Finite Element Method in Electromagnetics*, 2nd ed., New York, NY: John Wiley & Sons. Inc. & IEEE, 2002.

[3] C. T. Johnk, *Engineering Electromagnetic Fields and Waves*, 2nd ed., New York, NY: John Wiley & Sons, 1988.

[4] J. D. Jackson, "Plane Electromagnetic Waves and Wave Propagation," in *Classical Electrodynamics*, 3rd ed., New York, NY: John Wiley & Sons, 1999, pp. 295-352.

[5] C. Rappaport, "A Dispersive Microwave Model for Human Breast Tissue Suitable for FDTD Computation," *IEEE Antennas Wireless Propag. Lett.,* vol. 6, pp. 179-181, Apr. 2007.

[6] F. L. Teixeira, W. C. Chew, M. Straka, M. L. Oristaglio and T. Wang, "Finite-Difference Time-Domain Simulation of Ground Penetrating Radar on Dispersive, Inhomogeneous, and Conductive Soils," *IEEE Trans. Geosci. Remote Sens.,* vol. 36, no. 6, pp. 1928-1937, Nov. 1998.

[7] D. Jiao and J.-M. Jin, "Time-Domain Finite-Element Modeling of Dispersive Media," *IEEE Trans. Microw. Wireless Compon. Lett.,* vol. 11, no. 5, pp. 220-222, May 2001.

[8] F. L. Teixeira, "Time-Domain Finite-Difference and Finite-Element Methods for Maxwell Equations in Complex Media," *IEEE Trans. Antennas Propag.,* vol. 56, no. 8, pp. 2150-2166, Aug. 2008.

[9] D. M. Sullivan, "Frequency-Dependent FDTD Methods Using Z Transforms," *IEEE Trans. Antennas Propag.,* vol. 40, no. 10, pp. 1223-1230, Oct. 1992.

[10] A. Akbarzadeh-Sharbaf and D. D. Giannacopoulos, "A Stable and Efficient Direct Time Integration of the Vector Wave Equation in the Finite-Element Time-Domain Method for Dispersive Media," *IEEE Trans. Antennas Propag.,* vol. 63, no. 1, pp. 314-321, Jan. 2015.

[11] M. R. Zunoubi, J. Payne and W. P. Roach, "CUDA Implementation of TEz-FDTD Solution of Maxwell's Equations in Dispersive Media," *IEEE Antennas Wireless Propag. Lett.,* vol. 9, pp. 756-759, Jul. 2010.

[12] H.-T. Meng, B.-L. Nie, S. Wong, C. Macon and J.-M. Jin, "GPU Accelerated Finite-Element Computation for Electromagnetic Analysis," *IEEE Antennas Propagat. Mag.,* vol. 56, no. 2, pp. 39-62, Apr. 2014.

[13] A. D. Wunsch, "The Bilinear Transformation," in *Complex Variables with Applications*, 1st ed., Reading, MA: Addison-Wesley, 1983, pp. 362-378.

[14] A. V. Oppenheim and A. S. Willsky, "The Z-Transform," in *Signals and Systems*, 2nd ed., Upper Saddle River, NJ: Prentice Hall, 1996, pp. 741-816.

[15] D. B. Kirk and W.-M. W. Hwu, *Programming Massively Parallel Processors: A Hands-on Approach*, Burlington, MA: Morgan Kaufmann Publishers, 2010, pp. 21-39.

[16] NVIDIA Corporation, "CUDA C Programming Guide," Santa Clara, CA. Rep. PG-02829-001_v6.5, Aug. 2014.

[17] W.-M. W. Hwu, "Introduction & GPU-Accelerated Computation and Interactive Display of Molecular Orbitals," in *GPU Computing Gems*, Emerald Ed. ed., Burlington, MA: Morgan Kaufmann, 2011, pp. xix-19.

[18] N. Wilt, *The CUDA Handbook*, Upper Saddle River, NJ: Addison-Wesley, 2013.

[19] R. Farber, *CUDA Application Design and Development*, Waltham, MA: Morgan Kaufmann, 2011.

[20] B. Jacob and G. Guennebaud, "Eigen C++," 18 March 2015. [Online]. Available: http://eigen.tuxfamily.org/index.php?title=Main_Page. [Accessed 11 May 2015].

[21] Microsoft Corporation, "Collecting Detailed Timing Data by Using Instrumentation," 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/dd264993(v=vs.120).aspx. [Accessed 11 May 2015].

[22] Microsoft Corporation, "Acquiring high-resolution time stamps," 2015. [Online]. Available: https://msdn.microsoft.com/en-us/library/windows/desktop/dn553408(v=vs.85).aspx. [Accessed 11 May 2015].

[23] N. Bell and M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA," NVIDIA Corporation, Santa Clara, CA. Rep. NVR-2008-004, Dec. 2008.

[24] The Mathworks, Inc., "Cholesky Factorization," 2015. [Online]. Available: http://www.mathworks.com/help/matlab/ref/chol.html. [Accessed 11 May 2015].

[25] NVIDIA Corporation, "CUDA C Best Practices Guide," Santa Clara, CA. Rep. DG-05603-001_v7.0, Mar. 2015.

[26] NVIDIA Corporation, "CUDA Toolkit 3.1 Downloads," 2015. [Online]. Available: https://developer.nvidia.com/cuda-toolkit-31-downloads. [Accessed 11 May 2015].

[27] McGill HPC, "Guillimin Hardware Information," 2013. [Online]. Available: http://www.hpc.mcgill.ca/index.php/starthere/81-doc-pages/215-guillimin-hardware. [Accessed 11 May 2015].

[28] NVIDIA Corporation, "cuBLAS," 2015. [Online]. Available: https://developer.nvidia.com/cuBLAS. [Accessed 11 May 2015].