Graphics Hardware Accelerated Transmission Line Matrix Procedures

by

Filippo Vincenzo Rossi
B. Eng., University of Victoria, 2008

A Thesis Submitted in Partial Fulfillment
of the Requirements for the Degree of

MASTER OF APPLIED SCIENCE

in the Faculty of Electrical and Computer Engineering

Supervisory Committee

Graphics Hardware Accelerated Transmission Line Matrix Procedures

by

Filippo Vincenzo Rossi
B. Eng., University of Victoria, 2008

Supervisory Committee

Dr. Poman P.M. So (Department of Electrical and Computer Engineering)
Supervisor

Dr. Kin Li (Department of Electrical and Computer Engineering)
Departmental Member

Dr. Yvonne Coady (Department of Computer Science)
Outside Member

Abstract

Supervisory Committee

Dr. Poman P.M. So (Department of Electrical and Computer Engineering)
Supervisor

Dr. Kin Li (Department of Electrical and Computer Engineering)
Departmental Member

Dr. Yvonne Coady (Department of Computer Science)
Outside Member

The past decade has seen a transition of Graphics Processing Units (GPUs) from special purpose graphics processors, to general purpose computational accelerators. GPUs have been investigated to utilize their highly parallel architecture to accelerate the computation of the Transmission Line Matrix (TLM) methods in two and three dimensions. The design utilizes two GPU programming languages, Compute Unified Device Architecture (CUDA) and Open Computing Language (OpenCL), to code the TLM methods for NVIDIA GPUs. The GPU accelerated two-dimensional shunt node TLM method (2D-TLM) achieves 340 million nodes per second (MNodes/sec) of performance which is 25 times faster than a commercially available 2D-TLM solver. Initial attempts to adapt the three-dimensional Symmetrical Condensed Node (3D-SCN) TLM method resulted in a peak performance of 47 MNodes/sec or7 times in speed-up. Further efforts to improve the 3D-SCN TLM algorithm, as well as investigating advanced GPU optimization strategies resulted in performances accelerated to 530 MNodes/sec, or 120 times speed-up compared to a commercially available 3D-SCN TLM solver.

Table of Contents

List of Tables

List of Figures

Glossary

2D — two dimensional, a term used to describe structures/problems that vary only in two spatial directions.

3D — three dimensional, a term used to describe structures/problems that vary in all three spatial directions.

API — application programming interfaces, a collection of functions that enables application programs to interact with system software.

CPU — central processing unit, the hardware of a computer that executes general purpose computer programs.

CUBIN — CUDA binary, binary code that executes on GPUs which is compiled from CUDA (.cu) code.

CUDA — compute unified device architecture, an API for developing programming for execution on NVIDIA GPUs.

Device — in the context of GPU programming the device is the graphics processing unit hardware.

FDTD — finite difference time domain, a numerical method based on the direct discretization of the Maxwell's curl equations using finite difference; it is a popular numerical method in computational electromagnetics.

FLOPS — floating point operations per second, a unit for measuring the numerical performance of computer hardware.

GFLOPS — giga (billions) FLOPS.

Global Memory — bulk memory in the GPU accessible by the GPU multiprocessors as well as the CPU.

GPU — graphics processing unit, a processor specially designed to executed graphics instructions; it is now being used to execute numerical intensive algorithms.

Host — in the context of GPU programming the host is the CPU of the workstation in which GPU hardware is installed.

KCL — Kirchoff's current law

KVL — Kirchoff's voltage law

Link-Lines — transmission lines interconnecting TLM nodes, voltage impulses propagate on these lines.

MFLOPS — Million FLOPS.

MPI — message passing interface, an API enables computers to communicate with one another. It is used in computer clusters and supercomputers to support distributed computing.

Multiprocessor — a group of internal GPU processors.

Mesh — a collection of TLM nodes that form a 2D or 3D grid.

Node — a transmission line junction.

Node-Rate — the number of 3D-SCN nodes processed per second (expressed as million nodes per second, MNodes/sec, in most part of the thesis).

Node-Data-Rate — similar to the Node-Rate except expressed as the amount of memory transferred per second to process a mesh of 3D-SCN nodes.

$$(\text{Node-Data-Rate}_{(\text{Bytes/sec})} = \text{Node-Rate}_{(\text{Nodes/sec})} \times 12_{(\text{floats/node})} \times 4_{(\text{Bytes/float})})$$

NVCC — NVIDIA CUDA Compiler, a compiler translates CUDA code to CUBIN object code for execution on a GPU.

OpenMP — open multi-processing, a cross platform API that supports shared memory multiprocessing programming.

PC — personal computer.

PCI Express — peripheral component interconnect express, a motherboard-level interconnect and an expansion card interface.

PTX — parallel thread execution, an intermediary assembler language for execution on the GPU.

Register Memory — multiprocessor working memory

SDK — software development kit, a collection of APIs and compiler tool for software development.

Shared Memory — on-chip memory residing on each GPU multiprocessor

Speed-Up — used to compare speeds of one application or method with another.

SCN — symmetrical condensed node.

Thread-Block — a collection of threads which defines a multiprocessor working space

TLM — transmission line matrix method, a space and time discretising method for the computation of electromagnetic fields.

Acknowledgments

I would like to express my sincere thanks to gratitude to my supervisor, Dr. Poman P.M. So, Assistant Professor of Electrical and Computer Engineering at the University of Victoria, for the constant encouragement, advice, and support that he provided during the duration of my study.

My heartfelt thanks to my wife, Caroline, for her patience and encouragement during the pursuit of my academic aspirations.

Chapter 1 Introduction

1.1 Motivation

Graphics processing unit (GPU) based parallel computing has been an important topic for the computing industry for over a decade, principally to accelerate compute intensive applications beyond the capacity of high end workstations. Krakiwsky et al. and Inman et al. applied the technique to accelerate the FDTD algorithm [1, 2].Takizawa et al. applied GPU computing to heat transfer simulations [3]. Z. Luo et al. and Harding et al. applied the paradigm to artificial neural network [4] and genetic algorithms [5], respectively. Furthermore, a cluster of GPU based computers can be created to execute grand challenge problems [6]. Researchers at Stanford [7] have been using this technique for years in protein folding computation.

While CPU processor speeds have plateaued in the past 5 years [8], GPU's have gained attention for their co-processing power. The NVIDIA GPU used in this project (Quadro FX 5600) is able to deliver 500 GFLOPS (billion floating point operations per second) [17] of processing speed. At the time of this writing, the latest NVIDIA GPUs (Tesla C1060) are able to perform up to 1000 GFLOPS. Workstations can offload compute intensive tasks to GPUs in order to achieve processing speeds an order of magnitude faster than tasks executed on the motherboard's CPU [9]. The pervasive demand for responsive real-time graphics rendering hardware in the gaming industry as well as graphic design industries [10] have pushed down the price and driven up the performance

of consumer GPUs. Due to these two converging factors, the software industry is looking to adapt compute-intensive applications to the GPU architecture. The types of algorithms that can exploit a GPU's architecture can be challenging to implement because of the constraints imposed by this specialized hardware platform.

The Transmission Line Matrix (TLM) method is a space and time discretising method for the computation of electromagnetic fields [11]. TLM is based on the utilization of a mesh of transmission lines to model the propagation of electromagnetic fields in both 2D [12] and 3D [13] structures. The method is computationally intensive where simulation runs of hours or days are not uncommon. The TLM method would therefore benefit from any form of acceleration. Parallelization efforts based on MPI and Open are effective in increasing the speed of TLM programs running on server class supercomputers [14]. Modern workstations and PCs where the number of processing cores are typically 2, or 4, MPI and OpenMP can increase the speed of the TLM programs modestly .

In 2007 NVIDIA released a programming framework called Compute Unified Device Architecture (CUDA) [18], which provided developers a C like language to program NVDIA's CUDA enabled GPUs. In 2009, in reaction to this attempt by NVIDIA, Apple Inc., developed and released a similar framework called OpenCL (Open Computing Language) [19] which allows execution of programs across heterogeneous platforms comprising of GPUs and CPUs. Hence, OpenCL allows developers to create parallel computer code for execution on a heterogeneous combination of processors (CPUs,

GPUs or other compatible hardware) connected to the computer. OpenCL is now managed by the Kronos Group [20].

We adapted the Transmission Line Matrix algorithms [12-13] to the GPU architecture using both CUDA and OpenCL. To validate the correctness and efficiency of our GPU based implementations, we compared the results obtained with the GPU code to those obtained with a commercially available TLM software:MEFiSTo-3D Pro[16]. The software has both the two-dimensional shunt node TLM and three-dimensional symmetrical condensed node TLM engines.

We recognize that a comparison of the performance between dissimilar hardware platforms (GPUs vs. CPUs) is not a definitive benchmark. The processing power of either hardware platform advances with every new release, but the ever increasing rates of processing power for both are never in lockstep. If the differences in speed between GPUs and CPUs are pronounced enough, then GPUs could be considered a justifiable choice for TLM applications. Establishing a standard threshold for a GPU vs. CPU speed-up to affirm this justification is not easily defined. For example, a 10 times speed-up may prove to be a threshold beyond which it is arguably justified to put forth the effort of porting TLM to a GPU platform. However, a 1.5 time speed-up may not be enough of a payoff for the effort to create a GPU version of an algorithm, especially considering that using Open can easily achieve similar results. By comparing CPU and GPU speeds, we can only make clear statements of the merit of GPUs if the speed-up is so evident that

no argument can be made otherwise. Therefore we proceed with this comparative exploration with the expectation of clearly delineated outcomes.

1.2 Contributions

There are a number of commercially available electromagnetic software packages that have GPU based FDTD engines; CST Microwave Studio[41] and Accelerware Corp's SDK are two of those packages. There are no other known TLM software packages that takes advantage of GPU computing; the packages developed in this thesis are the first CUDA and OpenCL based TLM solvers reported in the literature.

In the process of adapting TLM algorithms to the GPU hardware architecture various advanced GPU optimization techniques were explored and then exploited. Although the software structures that were developed were specific the TLM method, a generalized approach was also created in the form of GPU code templates and coding guidelines. These templates and guidelines can be used as a foundation for future applications to other algorithms that are intended for GPU acceleration.

1.3 Overview of Thesis

Following this introduction, Chapter 2 gives a short description of the two- and three-dimensional Transmission Line Matrix methods. Chapter 3 describes the features of NVIDIA's CUDA enabled GPU; in particular, it's architecture, the programming languages required, and several advanced optimization techniques. Chapter 4 discusses the implementation of the 2D TLM algorithm on a GPU, where chapter 5 details the implementation of the 3D TLM method. Chapter 6 explores various GPU optimization techniques to create a memory optimized approach for GPU algorithms. Chapter 7 applies the previous chapter's memory optimized techniques to revise the design of the 3DTLM method. Chapter 8 reports the performance of two optimized 3D TLM programs, one implemented in CUDA and the other one in OpenCL. And finally, Chapter 9 summarizes important results, reflects on their significance, and presents potential future research direction.

Chapter 2 The Theory of Transmission Line Matrix Method

2.1 Introduction

The Transmission Line Matrix (TLM) method is a space and time discretising method which belongs to a family of differential time domain techniques [11]. Many of the theories and algorithms for the TLM method were developed by P.B. Johns and his colleagues in the early 1970's, where the first 2D TLM method was created [12] and then the 3D Expanded Node method [21] , as well as the Symmetrical Condensed Node in 1987 [13]. Unlike the Finite Difference Time Domain (FDTD) technique [23] which is based on the discretized form of Maxwell's equations, TLM is based on a scattering procedure that describes the behaviour of voltage impulse travelling on a network of transmission lines. TLM grew from network representations of Maxwell's equations where transmission line models were used to simulate microwave applications.

TLM can be presented using Huygens' Principle which models wave propagation using a recursive algorithm [22]. The principle stipulates that each point of an advancing wave front is in fact the center of a new disturbance and hence they are the sources of a new family of waves. The advancing wave, as a whole is then regarded as the sum of all the secondary waves that occurred at an earlier time instance (Figure 2.1).

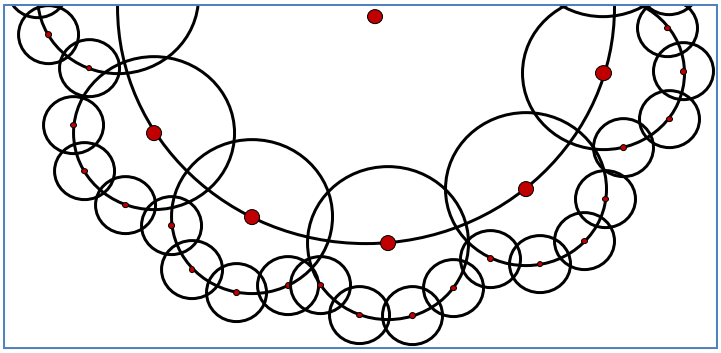


Figure 2.1: Huygens' Principle. Each point of an advancing wave front is the center of a new disturbance which, in turn, is the source of a new of wave fronts.

2.2 Basis of TLM Development

Many problems can be expressed in a more meaningful way by creating analogous representations based on known models[11]. Electrical network analysis is one such example where lumped elements represent physical manifestations of real world phenomena. Inductors and capacitors are devices which can concentrate electric and magnetic energy in space. A network of lumped circuit elements can be used to model a transmission line, Figure 2.2. A network of transmission lines can be used to model the medium that support electromagnetic wave propagation.

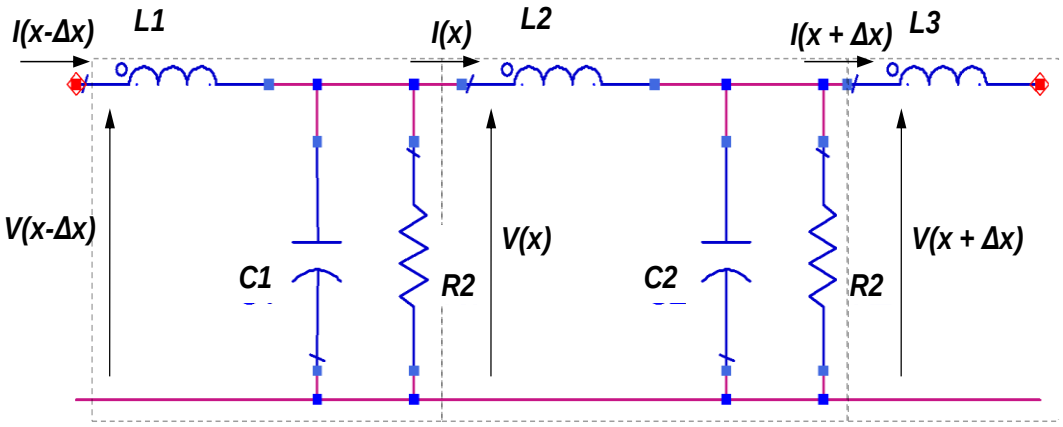


Figure 2.2: Lumped element representation of a segment of transmission line.

A transmission line can be modeled with a cascaded network of lumped element segments such as the one illustrated in Figure 2.2.Each segment, which contains an inductor, a capacitor, and a resistor, can store as well as dissipate electric and magnetic energy. When applying Kirchoff's voltage and current laws (KVL & KCL) to one of the segments we get [11]:

$$-\frac{\partial v(x,t)}{\partial x}\Delta x = L\frac{\partial i(x,t)}{\partial t} \tag{2.1}$$

$$-\frac{\partial i(x,t)}{\partial x}\Delta x = C\frac{\partial v(x,t)}{\partial t} + \frac{v(x,t)}{R} \tag{2.2}$$

Combining the two equations (and eliminating the voltages) gives an equation of the transmission line current that flows through the inductor [11]:

$$\frac{\partial^2 i(x,t)}{\partial x^2} = \frac{LC}{(\Delta x)^2}\frac{\partial^2 i(x,t)}{\partial t^2} + \frac{L}{(\Delta x)^2 R}\frac{\partial i(x,t)}{\partial t} \tag{2.3}$$

Combining equations (2.1) and (2.2) to eliminate the currents gives an equation of the voltage propagating through the transmission line[11]:

$$\frac{\partial^2 v(x,t)}{\partial x^2} = \frac{LC}{(\Delta x)^2}\frac{\partial^2 v(x,t)}{\partial t^2} + \frac{L}{(\Delta x)^2 R}\frac{\partial v(x,t)}{\partial t} \tag{2.4}$$

We now have a system of equations that describes the propagation of current and voltage through a transmission line. We now move on to describe electromagnetic propagation and then draw similarities with that of the aforementioned transmission line equations.

Consider an electromagnetic wave propagating in the *x*-direction. Also, designate the electric field polarity in the *y*-direction (E_y) and the magnetic field polarity in the *z*-direction (H_z). Following the derivations given in [11], Faraday's and Ampere's Laws as described by Maxwell's equations are:

$$\frac{\partial E_y(x,t)}{\partial x} = -\mu\frac{\partial H_z(x,t)}{\partial t} \tag{2.5}$$

$$-\frac{\partial H_z(x,t)}{\partial x} = j_y + \varepsilon\frac{\partial E(x,t)}{\partial t} \tag{2.6}$$

Differentiating (2.5) with respect to *x* gives:

$$\frac{\partial^2 E_y(x,t)}{\partial x^2} = -\mu \frac{\partial^2 H_z(x,t)}{\partial x \partial t} \tag{2.7}$$

Differentiating (2.6) with respect to *t* gives:

$$-\frac{\partial^2 H_z(x,t)}{\partial x \partial t} = \frac{\partial j_y}{\partial t} + \varepsilon \frac{\partial^2 E(x,t)}{\partial t^2} \tag{2.8}$$

Combining (2.7) and(2.8) gives:

$$\frac{\partial^2 E_y(x,t)}{\partial x^2} = \mu \frac{\partial j_y}{\partial t} + \varepsilon\mu \frac{\partial E(x,t)}{\partial t^2} \tag{2.9}$$

Recognizing that:

$$E_y = \frac{j_y}{\sigma} \tag{2.10}$$

and substituting(2.10) into(2.9) gives:

$$\frac{\partial^2 j_y(x,t)}{\partial x^2} = \mu\varepsilon \frac{\partial^2 j_y(x,t)}{\partial t^2} + \mu\sigma \frac{\partial j_y(x,t)}{\partial t} \tag{2.11}$$

Which describes current density where μ is the magnetic permeability, ε is the electric permittivity, and σ is the electric conductivity [11]. We now have enough information to draw similarities between the propagation of electromagnetic fields in space to that of the transmission of electromagnetic signal in the lumped element network.

A comparison between the equation of the circuit(2.3) and that of the electromagnetic field propagating in the *x*-direction (2.11) shows that the two are similar where only the coefficients differ. The velocity of electromagnetic waves in(2.11) is [11]:

$$v = \frac{1}{\sqrt{\mu\varepsilon}} \tag{2.12}$$

By using this velocity equation for the electromagnetic field equation in(2.11) and applying it to the circuit equivalent we get:

$$v = \frac{1}{\sqrt{\frac{L}{\Delta x}\frac{C}{\Delta x}}} = \frac{\Delta x}{\sqrt{L \cdot C}} \tag{2.13}$$

Equation (2.13) gives us a relation between the velocity of a propagating electromagnetic wave and its circuit equivalent, where capacitance, inductance and spatial resolution can be chosen. Therefore, the circuit in Figure 2.2may be used as a basis to represent complex field problems, where extension to two- and three-dimensional models can be made.

The above theoretical discussion is based on the assumption that the resolution (Δx) goes to zero. However, infinitely small resolution cannot be achieved in practical situations, so a finite Δx must be considered. As a rule of thumb, the resolution of a TLM mesh should be sufficiently fine to resolve one tenth of the smallest wavelength in the structure to be modeled [11]:

$$\Delta x \leq \frac{\lambda}{10} \tag{2.14}$$

2.3 2D TLM Shunt Node

Huygens described a mechanism of wave-front propagation where each point on a wave-front acts as an "isotropic spherical radiator" [22]. Consider two intersecting transmission lines (Figure 2.3a) each having the same length, and characteristic impedance. This is termed as a TLM *node*. Each of the four transmission lines radiating in four cardinal directions from the centre of the node are termed *link-lines*. If a one-volt impulse is launched into port 1 (Figure 2.3b), the impulse will travel along the transmission line toward the junction. Once reaching the junction a process called *scattering* occurs. The one-volt impulse sees three transmission lines in parallel and will see an impedance equal to $Z_{Load} = Z_0/3$, where Z_0 is the characteristic impedance of each transmission line. The reflection coefficient seen by the impulse is then [11]:

$$\Gamma = \frac{\left(Z_{Load} - Z_O\right)}{\left(Z_{Load} + Z_O\right)} = \frac{\left(Z/3 - Z\right)}{\left(Z/3 + Z\right)} = -\frac{1}{2} \tag{2.15}$$

Therefore, with a one-volt impulse launched into port 1 the voltage reflected back along port 1 would be $V_1 = -0.5$ V.

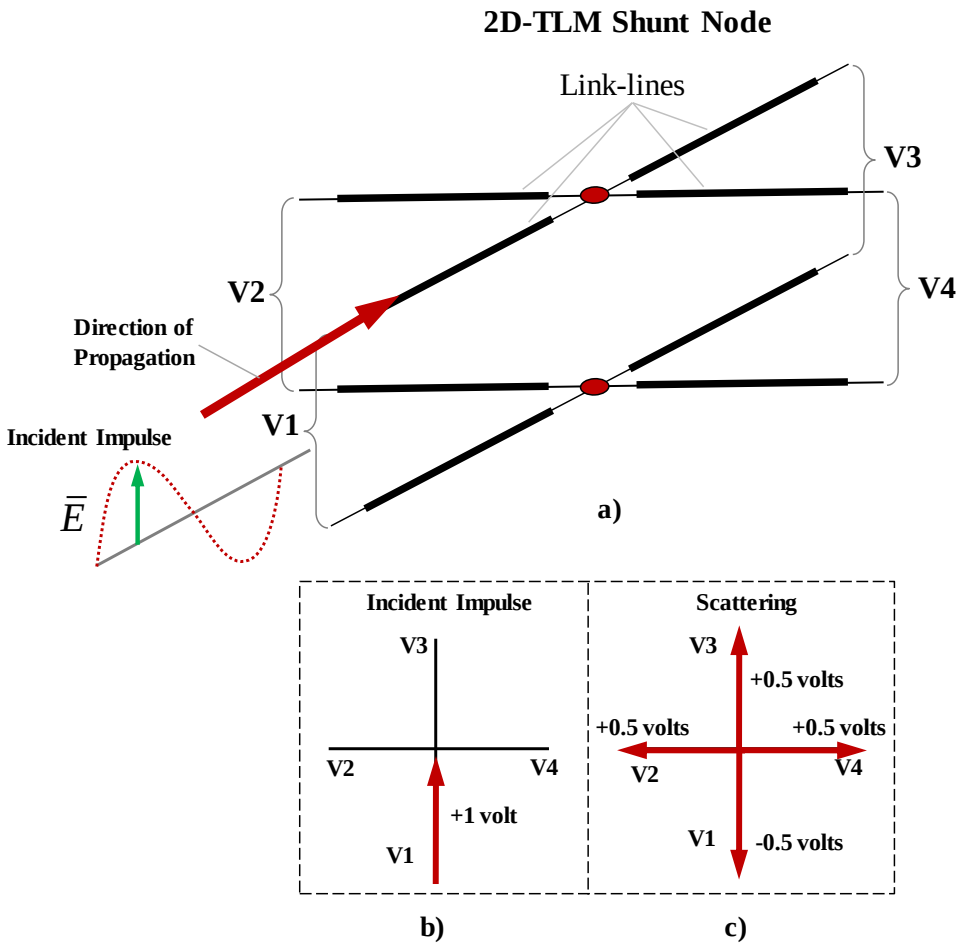


Figure 2.3: 2D-TLM shunt node formed from intersection of two transmission lines(a) three-dimensional view (b) top-view of impending impulse on port V1, (c) scattering result.

The transmission coefficient to each of the three link lines seen by the impulse is [11]:

$$T = 1 + \Gamma = + \frac{1}{2} \tag{2.16}$$

Therefore, the transmitted voltage to each of these three ports would be $V_2=V_3=V_4=$ +0.5V. The original one-volt impulse generates four scattered voltage impulses (Figure 2.3c) radiating in all four directions.

After scattering, the voltage impulses emerging from a node (Figure 2.4a) and propagate to the neighbouring nodes (Figure 2.4b) becoming the incident impulses for the scattering operations at the next time step. The exchange with neighbouring nodes is called the *impulse-interchange* stage of the computation. Figure 2.5 illustrates a *mesh* of nodes and the effect of scattering and impulse-interchange operations over a three time-step period.

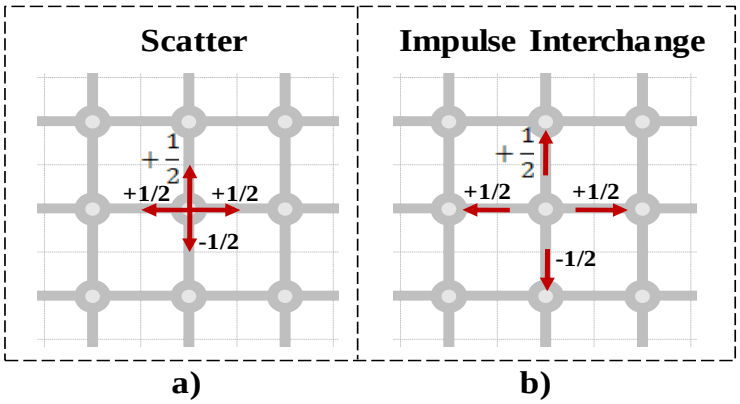


Figure 2.4: Stages of the 2D-TLM Iteration. (a) Scattering calculations are carried out for each node. (b) The impulse-interchange stage exchanges link-line voltages among all nodes.

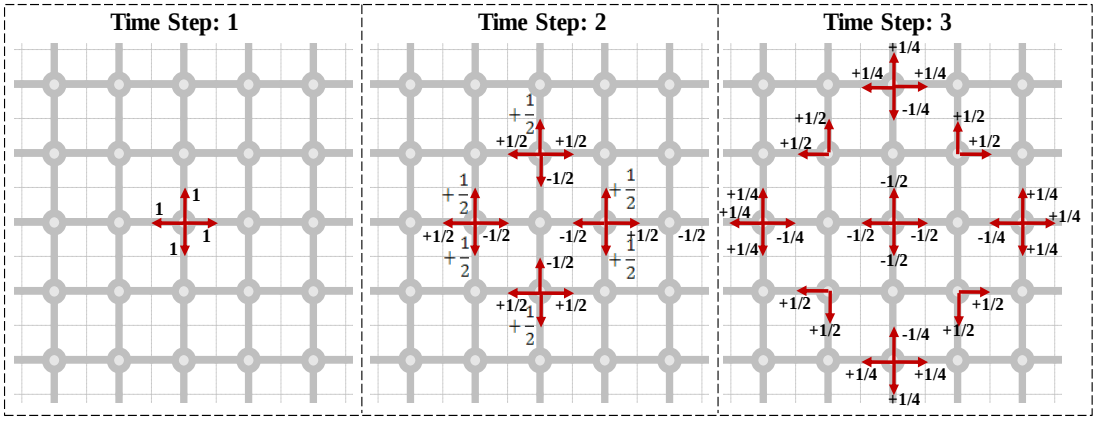


Figure 2.5: Impulse response of 2D mesh of TLM nodes over three time-steps.

2.4 Boundary

A mesh of TLM nodes would model only free space if not for the use of boundaries. Boundaries define structures such as waveguides, filters, antennas and so forth. Boundaries, within a TLM mesh, are placed between nodes (Figure 2.6) and are defined with reflection coefficients (Γ). When a voltage impulse encounters a boundary, it is multiplied by the boundary's reflection coefficient and then reflected back towards the direction in which it came from: $V^r = \Gamma \times V^i$ (Figure 2.6b) [24]. This boundary calculation is dealt with after the scattering stage. The boundary and impulse interchange operations are mutually exclusive hence the order in which they are executed is not important.

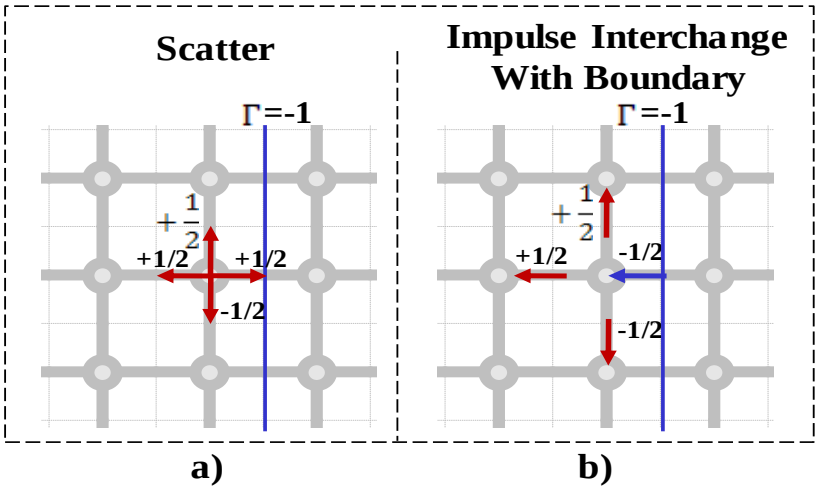


Figure 2.6: Impulse behaviour adjacent to a Perfect Electric Conductor (PEC) Boundary. (a) Scattering adjacent to boundary,(b) impulse interchange and boundary reflection results.

Two types of boundaries are dealt with in this thesis: perfect electric conductor (PEC) and absorbing boundaries [11,24]. The reflection coefficient of a PEC boundary is $\Gamma_{PEC} = -1$, i.e. the magnitude of the reflected voltage is negative of that of the incident voltage (Figure 2.6b). An absorbing boundary does not reflect incident waves but this does not mean the impulse reflection coefficient, Γ_{abs}, is zero. The derivation for Γ_{abs} is given in [24].In general, Γ_{abs} depends on the structure to be modeled as well as on the frequency of operation. For a two-dimensional TLM mesh representing free space,

$$\Gamma_{abs} = \frac{(1-\sqrt{2})}{(1+\sqrt{2})} = -0.17157 \qquad (2.17)$$

For three-dimensional symmetrical condensed node (to be discuss in the next section), $\Gamma_{abs} = 0$.

2.5 3D TLM Symmetrical Condensed Node (SCN)

2.5.1 SCN Node Structure

The most widely used three-dimensional TLM node is the Symmetrical Condensed Node (SCN) [13]. This node consists of a network of transmission lines that appears as two orthogonal ports at all six faces of the node (Figure 2.7). The voltages at these orthogonal ports represent the field polarizations. These voltages are named using a three-letter subscript convention proposed in [11] as shown in Figure 2.7. The first letter indicates the direction (*x, y, or z*) of the link-line on which the voltage pulse is propagating along. This second letter denotes the position (*n* or *p*) of the voltage pulse relative to the node center. The last letter specifies the polarization direction of the voltage pulse. For instance, V_{xny} in Figure 2.7 represents the "*y*" polarized voltage pulse located at the left of the node center. To indicate if the voltage is travelling towards or moving away from the node center, a superscript ("*i*": incident or "*r*": reflected) is added to the name. For example, V^{i}_{xny} and V^{r}_{xny} represent the "y" polarized incident and reflected pulses at the "n" (negative) position in the "x" link-line.

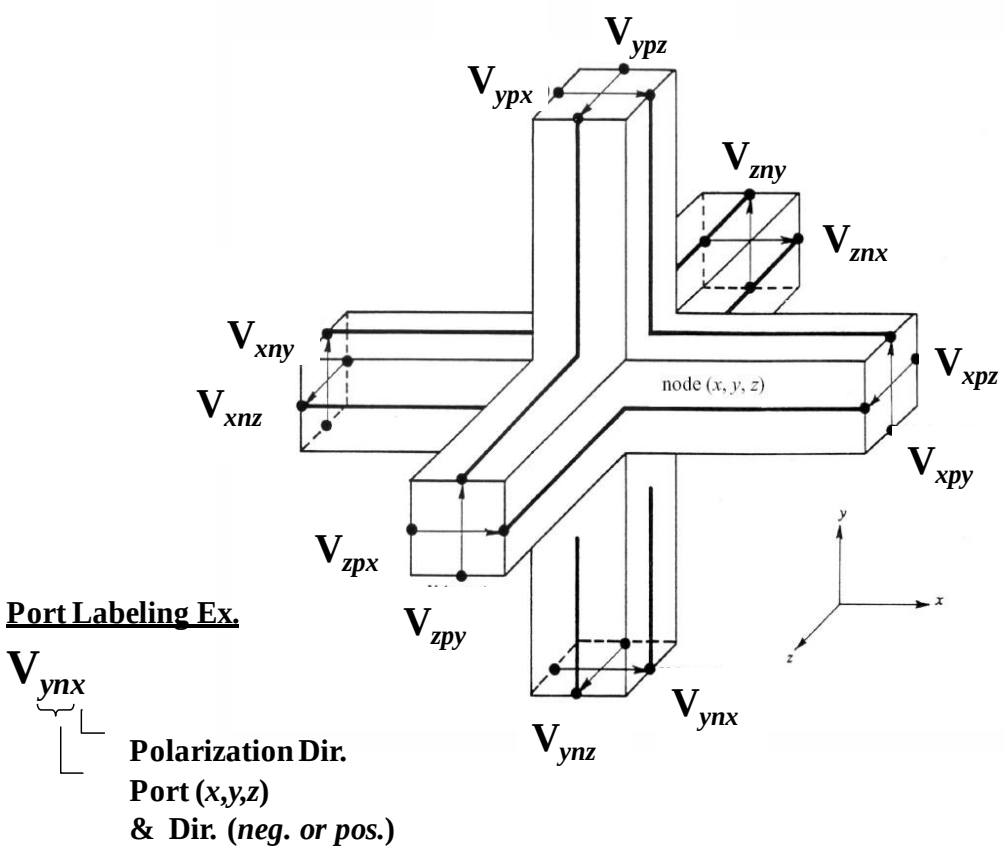


Figure 2.7: 3D TLM SCN Node [13].

2.5.2 SCN Algorithms

A part of the SCN node can be considered in its shunt arrangement (Figure 2.8a) or in its series arrangement (Figure 2.8b). Both of these arrangements are used to develop the scattering matrix for the 3D-SCN node. The essence of the theory presented in [11] is summarized below. In order to develop the scattering procedure, four principles must be enforced [11]:

1. Conservation of electric charge

2. Conservation of magnetic flux
3. Electric field continuity
4. Magnetic field continuity

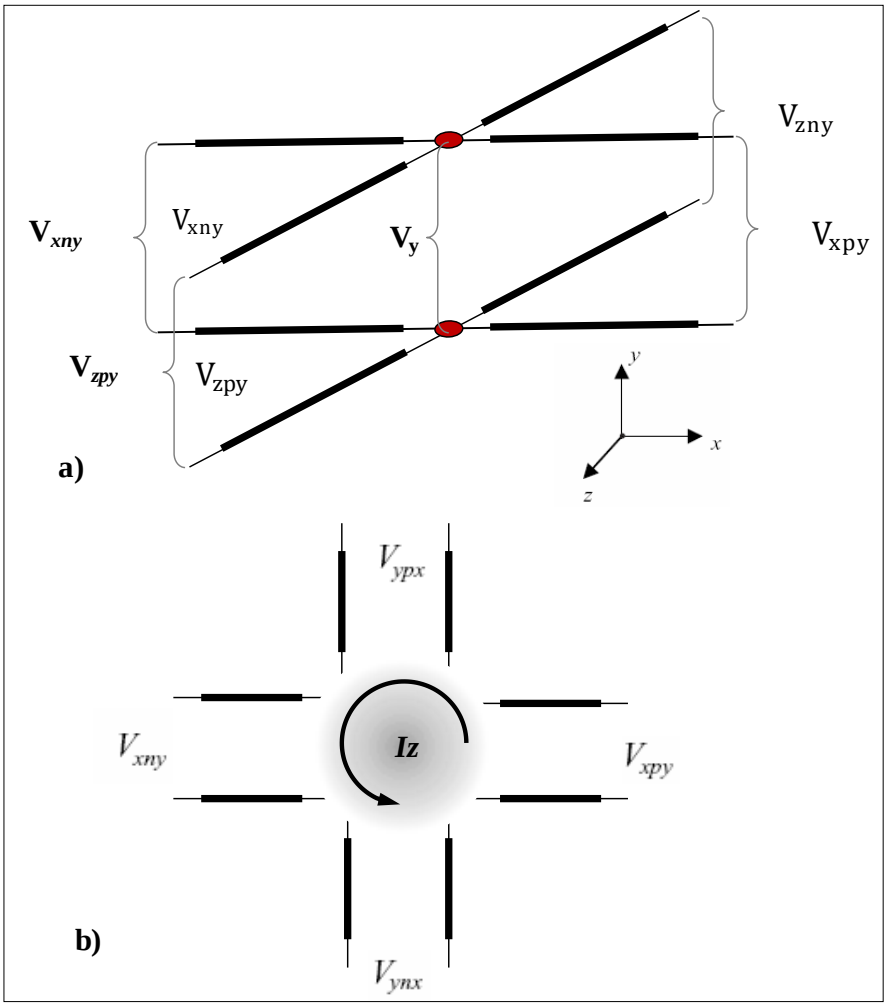


Figure 2.8: (a) SCN Shunt node,(b)SCN series node.

For condition 1, following electric conservation for the y-component of Figure 2.8a, we equate the total incident and reflected charge for the four half link-lines:

$$\frac{C}{2}\left(V_{xny}^{i}+V_{zny}^{i}+V_{xpy}^{i}+V_{zpy}^{i}\right)=\frac{C}{2}\left(V_{xny}^{r}+V_{zny}^{r}+V_{xpy}^{r}+V_{zpy}^{r}\right) \quad (2.18)$$

For condition 2, the case of conserving magnetic flux:

$$\sum_{n}\lambda_{n}=\sum_{n}L_{n}I_{n}=\sum_{n}L_{n}(I_{n}^{i}-I_{n}^{r})=0 \quad (2.19)$$

Expressing equation (2.19) in the z-component as in Figure 2.8b:

$$\frac{L}{2}(-I_{xny}^{i}-I_{ypx}^{i}+I_{xpy}^{i}+I_{ynx}^{i})=\frac{L}{2}(-I_{xny}^{r}-I_{ypx}^{r}+I_{xpy}^{r}+I_{ynx}^{r}) \quad (2.20)$$

Recognizing that $I^{i}=V^{i}/Z$ and $I^{r}=-V^{r}/Z$ equation (2.20) becomes:

$$-V_{xny}^{i}-V_{ypx}^{i}+V_{xpy}^{i}+V_{ynx}^{i}=V_{xny}^{r}+V_{ypx}^{r}-V_{xpy}^{r}-V_{ynx}^{r} \quad (2.21)$$

For condition 3, the continuity of the electric field calculated for the *y*-component in Figure 2.8a, (for either *x*- or *z*- directions) are:

$$V_{zpy}+V_{zny}=V_{xpy}+V_{xny} \quad (2.22)$$

Equation 20 expressed in incident and reflected voltages is:

$$V_{zpy}^{i}+V_{zpy}^{r}+V_{zny}^{i}+V_{zny}^{r}=V_{xpy}^{i}+V_{xpy}^{r}+V_{xny}^{i}+V_{xny}^{r} \quad (2.23)$$

For condition 4 to be satisfied, the continuity of the magnetic field lines implies that the *z*-component must be the same (in either the *x*- or *y*-directions) in Figure 2.8b, where:

$$I_{xpy}-I_{xny}=I_{ynx}-I_{ypx} \quad (2.24)$$

Expressing equation(2.24) in terms of incident and reflected voltages becomes:

$$V_{xpy}^{i}-V_{xpy}^{r}-(V_{xny}^{i}-V_{xny}^{r})=V_{ynx}^{i}-V_{ynx}^{r}-(V_{ypx}^{i}-V_{ypx}^{r}) \quad (2.25)$$

Equations (2.18) to (2.25)express a system of equations for one of three directions. When equations (2.18) to (2.25)are combined, along with their equivalent in the other

two dimensions where reflection voltages calculated, a system of 12 equations are formed [11] to describe 3D-SCN scattering:

$$
\begin{aligned}
V_{ynx}^{r} &= \frac{1}{2}\left(V_{znx}^{i} + V_{zpy}^{i} + V_{xny}^{i} - V_{xpy}^{i}\right) \\
V_{ypx}^{r} &= \frac{1}{2}\left(V_{znx}^{i} + V_{zpx}^{i} + V_{xpy}^{i} - V_{xny}^{i}\right) \\
V_{znx}^{r} &= \frac{1}{2}\left(V_{ynx}^{i} + V_{ypx}^{i} + V_{xnz}^{i} - V_{xpz}^{i}\right) \\
V_{zpx}^{r} &= \frac{1}{2}\left(V_{ynx}^{i} + V_{ypx}^{i} + V_{xpz}^{i} - V_{xnz}^{i}\right) \\
V_{zny}^{r} &= \frac{1}{2}\left(V_{xny}^{i} + V_{xpy}^{i} + V_{ynz}^{i} - V_{ypz}^{i}\right) \\
V_{zpy}^{r} &= \frac{1}{2}\left(V_{xny}^{i} + V_{xpy}^{i} + V_{ypz}^{i} - V_{ynz}^{i}\right) \\
V_{xny}^{r} &= \frac{1}{2}\left(V_{zny}^{i} + V_{zpy}^{i} + V_{ynx}^{i} - V_{ypx}^{i}\right) \\
V_{xpy}^{r} &= \frac{1}{2}\left(V_{zny}^{i} + V_{zpy}^{i} + V_{ypx}^{i} - V_{ynx}^{i}\right) \\
V_{xnz}^{r} &= \frac{1}{2}\left(V_{ynz}^{i} + V_{ypz}^{i} + V_{znx}^{i} - V_{zpx}^{i}\right) \\
V_{xpz}^{r} &= \frac{1}{2}\left(V_{ynz}^{i} + V_{ypz}^{i} + V_{zpx}^{i} - V_{znx}^{i}\right) \\
V_{ynz}^{r} &= \frac{1}{2}\left(V_{xnz}^{i} + V_{xpz}^{i} + V_{zny}^{i} - V_{zpy}^{i}\right) \\
V_{ypz}^{r} &= \frac{1}{2}\left(V_{xnz}^{i} + V_{xpz}^{i} + V_{zpy}^{i} - V_{zny}^{i}\right)
\end{aligned}
\tag{2.26}
$$

It is the system of equations in (2.26) that will be implemented to create a scattering stage of execution for the simulation of a 3D-SCN mesh network.

The impulse-interchange stage of the 3D-SCN mesh is much the same as 2D-TLM. The voltages on the link-lines of each node are exchanged with the link-lines of adjacent nodes.

2.6 CPU Based TLM Implementations

2.6.1 Serial and Parallel CPU Implementations

CPU based TLM algorithms employ nested loops to perform the required scattering and impulse interchange algorithms over a TLM mesh of nodes. The procedure is illustrated in Figure 2.9a, where the scattering procedure is executed over each node; one at a time. When a single pass of scattering computations has executed over all nodes, an additional nested loop performs another pass to effect the impulse-interchange computation, Figure 2.9b. Once scattering and impulse-interchange are executed on all the nodes of a mesh, an iteration of one time-step is considered complete. The next iteration of a time-step can commence where scattering and impulse interchange operations repeat.

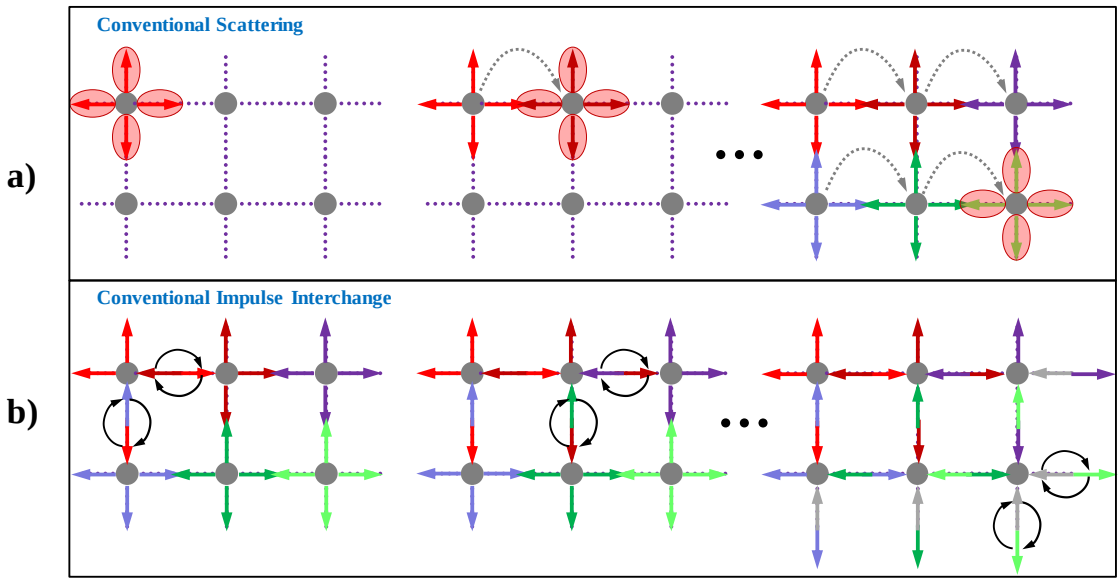


Figure 2.9:(a) Conventional TLM Scattering, and (b) Impulse-Interchange executed serially over a mesh of TLM nodes.

If the workstation contains multiple CPU cores, then parallel programming techniques can be utilized to take advantage of the extra computing power. OpenMP, for example, splits loops of computations into segments [26] and distributes the work equally to all CPU cores. When used for TLM computations, OpenMP utilizes N CPU cores by partitioning a mesh into N segments and allowing each to perform either scattering or impulse-interchange on their respective segment.

2.6.2 Commercial TLM Solver

Commercially available software packages such as Microstripes [15] and MEFiSTo [16] are based on the above mentioned CPU oriented TLM algorithms. Using these software packages one can model electromagnetic structures such as filters,

waveguides, and antennas. TLM algorithms enable the software to compute time domain responses directly; frequency characteristics such as S-parameters are obtained via Fourier transform. Figure 2.10 is a snapshot of the MEFiSTo software depicting the solution of a waveguide filter.

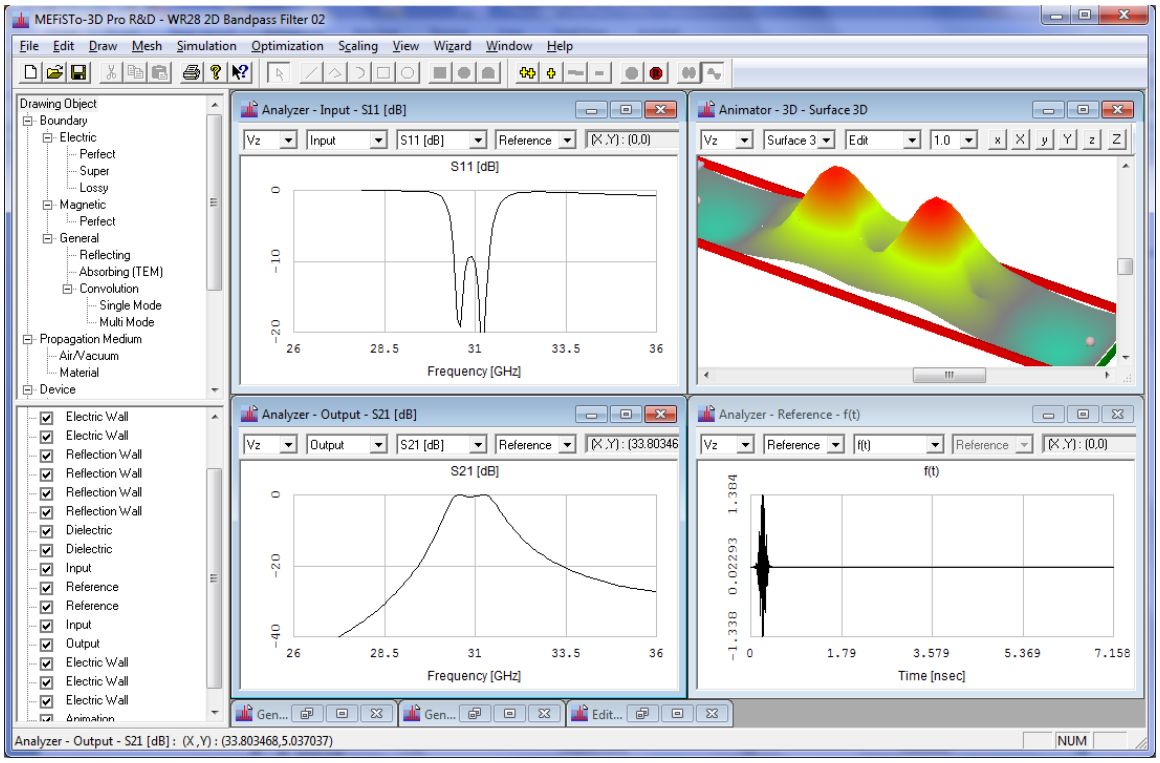


Figure 2.10: Snapshot of MEFiSTo 3D Pro, a commercially available TLM solver.

2.7 Validation Method

Simulation results obtained with MEFiSTo have been used as references to validate the GPU based TLM (GPU-TLM) software developed for this thesis. The purposes of the validation are: (1) to ensure the GPU based implementations do perform TLM computations correctly, and (2) to measure the increase in performance. The structures

modeled by both GPU-TLM and MEFiSTo are identical (resolutions, dimensions and component positioning).

2.7.1 Hardware Configurations

The hardware used to obtain the validation results are summarized below:

Workstation [25]

- brand: HP-xw9400
- clock rate: 2.4 GHz
- CPUs: Two Intel Dual Core Opteron Processors
- Memory: 16 GB DDR2 RAM
- Cache: 1M L2 cache @ 1GHz

GPU [17]

- brand: NVIDIA Quadro FX5600
- clock rate: 600 MHz
- CPUs: 128
- Memory: 1.5 GB GDDR3 RAM
- Interface: PCI Express v2.0 (500 MB/sec)

GPU-TLM speed-up comparisons are made against MEFiSTo utilizing one to four of the workstation's CPU cores.

2.7.2 Validation Structure: WR-28 Band Pass Filter

A validation method used in this project was to simulate a WR-28 band-pass filter (Figure 2.11) in the GPU and in MEFiSTo (2D and 3D versions). This filter was chosen because simulating it encompassed all the components and functions that were desired to be tested, which included:

- Perfect Electric Conductor (PEC) boundaries
- Absorbing boundaries
- Half-Sine excitation plane (using Gaussian shaped excitations)
- Sample probes, in order to perform scattering parameter analysis
- Ease of extending a 2D model to 3D, in both MEFiSTo and GPU-TLM

The WR-28 filter was not exclusively used as other structures were configured and tested, but it was desirable to maintain a consistent structure throughout the stages of algorithm design as a standard for comparison.

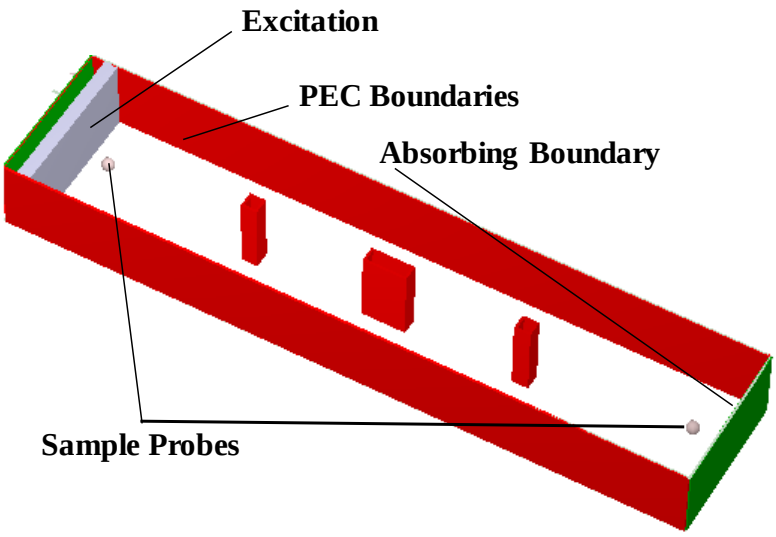


Figure 2.11: WR-28 band pass filter (31 to 32.1 GHz) [16].

2.8 TLM and its Parallel Nature

The TLM method is well adapted to execute on a parallel architecture. The scattering routine requires its calculation to be done for each node in the mesh. The order in which nodes have this calculation carried out is inconsequential. In a multiple processor environment, each processor would be free to process scattering for any node in any order without requiring coordination of the sequence of node execution. In addition, each node's voltages that are stored in memory are dedicated to that node only. Therefore, memory contention is not an issue, where many processors potentially require access to the same memory location. A similar argument can be made for the parallelization of the impulse interchange routine, where relaxed processing order of the TLM computation stages and its inherent memory locality are favourable for parallelization.

Chapter 3 The Graphics Processing Unit

3.1 GPU Introduction

The GPU used for this project contains an architecture that supports the Single Instruction Multiple Data (SIMD) [29] computing paradigm. These GPUs utilize multiple processors to perform identical tasks on large amounts of data. These systems are also scalable by utilizing GPU clusters external to a workstation [31]. Compared to super-computers, these GPUs are relatively inexpensive and are installed on workstations. As a result GPU based high-performance massively parallel processing systems are making their way into the world of scientific computing. We developed and reported our first GPU-TLM engine in 2008 [30].

	Memory	Mem Bandwidth	Processors	Processor Speed
Quadro FX 5600	1.5 GB (GDDR3)	76.8 GB/sec	128	600MHz

Figure 3.1:The NVIDIA Quadro FX6500 [17]Graphics Processing Unit (GPU) used in this project.

The Quadro FX5600 graphics processor GPU used to obtain the validation results for this thesis is shown in Figure 3.1 [17]. This GPU has a total of 128 processors and contains 1.5 GB of GDDR3 memory; the processors runs at 600 MHz. The GPU communicates with the PC via a PCI Express slot (PCI-E v2.0)[32]. The rate of data transfer between the GPU and the PC is8GB/sec. Data transfer between the PC and GPU is a potential bottleneck for GPU based programs that require intensive PC-GPU data transfer. The TLM programs in this thesis require very little PC-to-GPU transfer of data in between time-step iterations, since transfer of mesh data is needed only at the start and at the end of iteration runs.

An important performance metric for GPUs is that of memory transfer rate between the GPU's GDDR3 memory (1.5 GB in this case) and the 128 processors. A peak read/write performance of 76.8 GB/sec has been reported by NVIDIA. The importance of this memory transfer metric is that most algorithms that are executed on GPUs are limited not by their computation speed, but by the efficiency of accessing their internal memory. Memory access speed is, by far, the most difficult to maintain at its peak rate while, at the same time, adhering to the coding required by the TLM algorithms. More detail on this will be given in the following sections.

3.2 Hardware Model

The hardware architecture of the NVIDIA GPU is depicted in Figure 3.2. The 128 processors of the Quadro FX 5600 are grouped into 16 *multiprocessors* (8 processors each). The code that is executed on the GPU is called a *kernel*. A kernel is executed by all multiprocessors concurrently, but each multiprocessor cannot be tasked to execute different kernels, therefore they all work on the same kernel code at the same time. Each multiprocessor contains 8k of *registers* and 16k of on-chip memory called *shared memory*. Shared memory and registers are local to each multiprocessor and access time is in the range of 4-6 clock cycles which is the best of all memory access schemes for the GPU. However, multiprocessors cannot access the registers and shared memory of other multiprocessors (Figure 3.2).

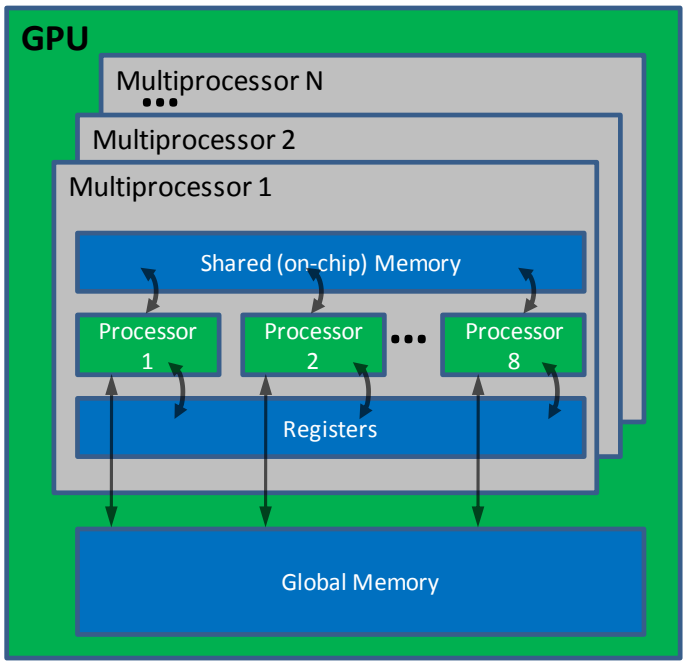


Figure 3.2:NVIDIA GPU Hardware Architecture.

One advantage the NVIDIA GPU architecture has over CPU based systems is that the smallest executable unit of parallelism on a GPU, called a *warp*, comprises 32 threads, as opposed to a maximum of 16 concurrent threads on a CPU. In fact each GPU multiprocessor can support 768 active threads per multiprocessor, for a total of 12,288 active threads for this GPU [17].

The 1.5 GB of GDDR3 memory onboard the Quadro FX5600is called *global memory*. Global memory can be accessed by all multiprocessors, but requires400 to 600 clock cycles for a typical read or write transaction. Although, access to global memory is not as fast as accessing registers and shared-memory, various techniques, such as coalesced memory access, can be utilized to reduce the access time. This will be explained in more detail in the following chapters.

The GPU contains 64K of *constant cache* memory on each GPU which is read-only for the multiprocessors. Constant cache memory can be accessed nearly as fast as shared memory (under special circumstances), but can only be written to by the PC (or Host).It is best utilized when static values are stored by the host and frequently read by kernels as opposed to using slower global memory. A detailed understanding of the characteristics and performance limitations of each of the aforementioned processor and memory resources is crucial when adapting them to best suit intended goals of algorithm adaptations.

3.3 Thread Model

A thread is a sequence of executing instructions that can run independently of other threads. NVIDIA GPUs group threads into thread-blocks (Figure 3.3). Each thread-block can be configured to contain a maximum of 512 threads. Multiple thread-blocks grouped together form a grid (Figure 3.3). Each multiprocessor can execute only one thread-block at a time to cover the grid, where each thread within the thread-block executes each instruction of a kernel [18-19]. The 16 multiprocessors of the GPU are coordinated to execute all the thread-blocks in a grid. The number of thread-blocks that can be defined over a grid is limited only by the GPUs resources (i.e. global memory).

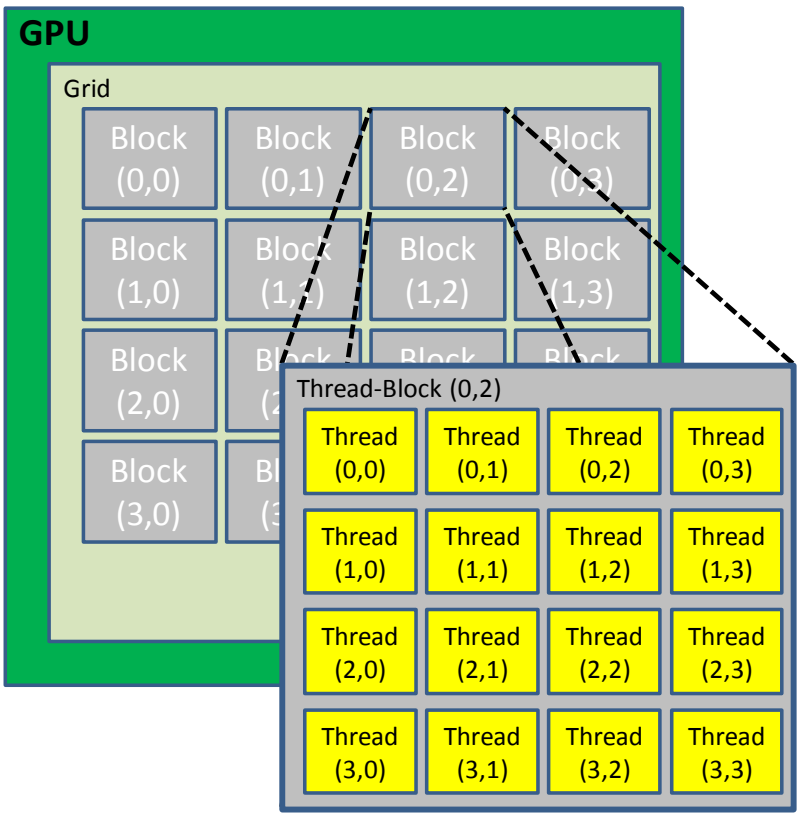


Figure 3.3:Thread-block and grid model

Typically grids of thread-blocks are mapped to access regions in global memory. In the TLM case, for example, each thread could be dedicated to a TLM node, where each thread would be responsible for: copying the node's voltages from global memory, execute TLM scattering, impulse-interchange, and boundary routines, then finally write the results back to global memory. It is thus very important to carefully match a thread-block dimension and grid pattern that accesses data in global memory as fast as possible as well as satisfying the objectives of the algorithm.

Thread-blocks may be defined as one-dimensional, two-dimensional, or three-dimensional [18]. The choice depends on the nature of the algorithm or data layout required of the algorithm to be parallelized. The choice of dimension is only for readability and has no bearing on the effectiveness of the kernel in the multiprocessors.

3.4 Typical GPU Iteration Cycle

Figure 3.4 illustrates a program cycle and memory partitioning scheme used for the TLM kernels, which is also typical of most GPU programs. Initially the host PC transfers data to global memory and constant cache in the GPU. The host then issues a command to the GPU to execute a kernel. The multiprocessors would then be coordinated by the GPU to execute the kernel code for each thread-block in the grid.

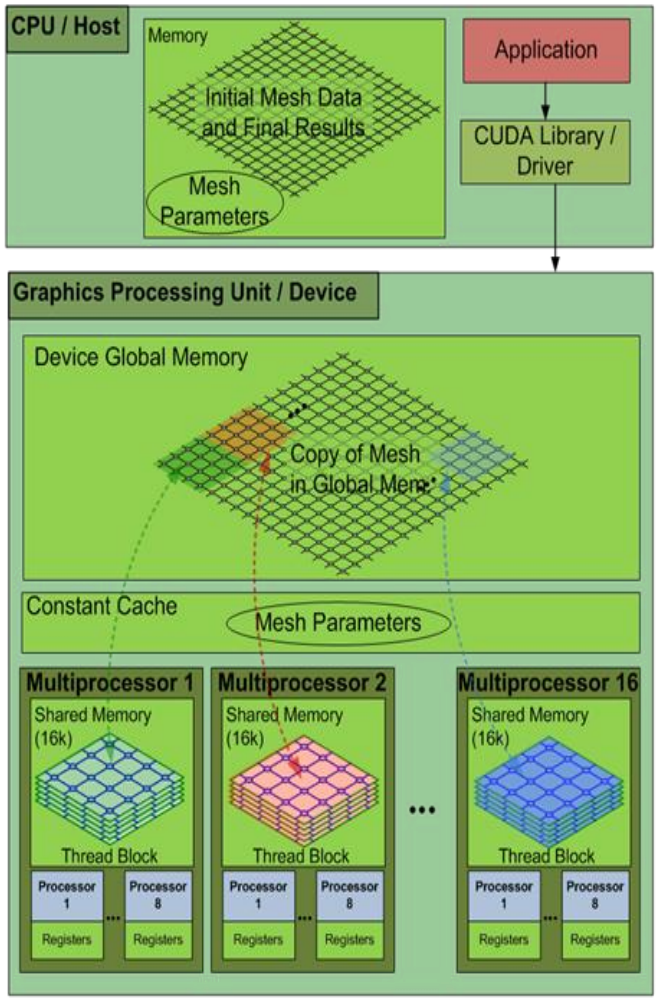


Figure 3.4:Typical program cycle and memory partitioning scheme of a GPU.

The kernel code within each thread-block begins with copying a data-block (partitioned from global memory) into each multiprocessor's shared memory (16k) or register memory (8k) (Figure 3.4). Each multiprocessor proceeds executing kernel code operations for a TLM computation for each thread-block (and for each thread in the thread-block) and finally writes the results back to global memory. The multiprocessor is then freed, at which time the GPU directs it to execute the next available thread-block in

the grid. All multiprocessors work in concert to execute each of the thread-blocks of the grid.

Once the GPU execution of the kernel completes, and all the thread-blocks of a grid have been processed, control is returned to the host. For the TLM case, one time-step iteration would complete after all the threads in the TLM mesh successfully executed the TLM kernel code. The host then issues another command to execute the TLM kernel again for the next time-step iteration. The results in global memory of the previous time-step are used as initial data for the next time-step. When the number of iterations has reached a user defined number, the host then transfers the results from GPU memory to the host for analysis.

3.5 GPU Programming

3.5.1 CUDA

CUDA (Compute Unified Device Architecture) is a parallel computing architecture developed by NVIDIA and was first released in 2007[18]. CUDA code can be either executed on the host (CPU) or on the device (GPU). CUDA code that executed on the host is responsible for initializing the GPU, transferring data to/from the GPU, and calling functions (kernels) in the GPU.

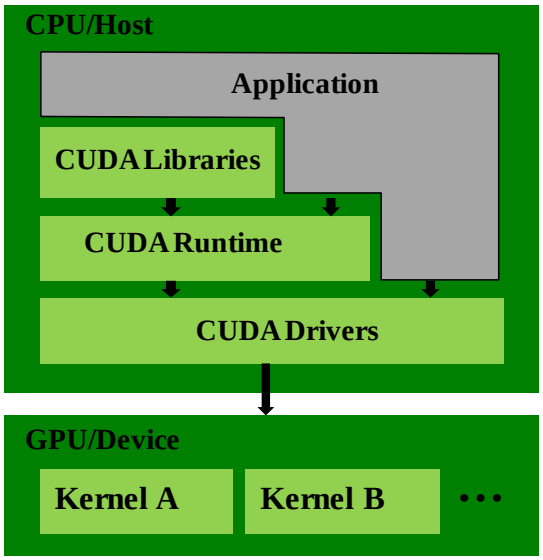


Figure 3.5: Overview of CUDA drivers, libraries, and kernels.

CUDA host code has two forms: C-for-CUDA, and the CUDA Driver API. C-for-CUDA is C like language which provides a simple path for users familiar with the C programming language to write programs to control the device. CUDA Driver API is a low level version of CUDA which is handle-based. Most objects are referenced by handles that may be passed to functions to manipulate CUDA objects. It was C-for-CUDA that was utilized for this project. The CUDA kernel code is compiled by NVIDIA's NVCC (NVIDIA CUDA Compiler) into *cubin* object code and is transferred to the target GPU for execution. Kernel functions are called by host code via the standard C convention.

3.5.2 CUDA Matrix Addition Example

To compare the programming styles of a conventional C program and a parallelized GPU enabled program, an example is given where the sum of two matrices is calculated.

Figure 3.6a depicts a code segment written in C to be executed on the PC. Figure 3.6b shows C-for-CUDA code that is also executed on the PC. The C-for-CUDA code makes calls to the GPU to execute a kernel which facilitates the matrix calculation (Figure 3.6c). The conventional C code (Figure 3.6a) contains a nested for-next loop iterating one index at a time in order to calculate the matrix sum. The C-for-CUDA code (Figure 3.6b) runs on the host (CPU). It launches the matrix addition calculation on the GPU via a call to a kernel Figure 3.6c. A 16×16 two-dimensional thread-block is defined, which is 256 threads. Recall that there is a limit of 512 threads per thread-block. A mapping of one thread per matrix element is established. A grid is defined so that thread-blocks can cover the M×N matrix calculation span(Figure 3.7). The GPU tasks the 16 multiprocessors to execute all the thread-blocks in the grid to facilitate the matrix addition.

Figure 3.6: Programming sample of the matrix operation (C=A+B).(a) sample nested loop algorithm done in C.(b) The same matrix operation implemented in C-for-CUDA which launches (c) the kernel code on the GPU.

It would not be unusual to find some overlap of thread-blocks in order to cover the M×N matrix of the calculation span. This is especially true if the thread-block size is fixed at compile time, but the dimensions of matrices may be user defined and specified at run time. A conditional statement in the CUDA kernel listing in Figure 3.6c, ensures

that thread-blocks that straddle the execution bounds of the matrix execute only threads that fall inside the calculation space. This confines the kernel to the M×N dimension for each matrix so that the dimensions of the matrix can be arbitrary and limited only by the amount of global memory on the GPU.

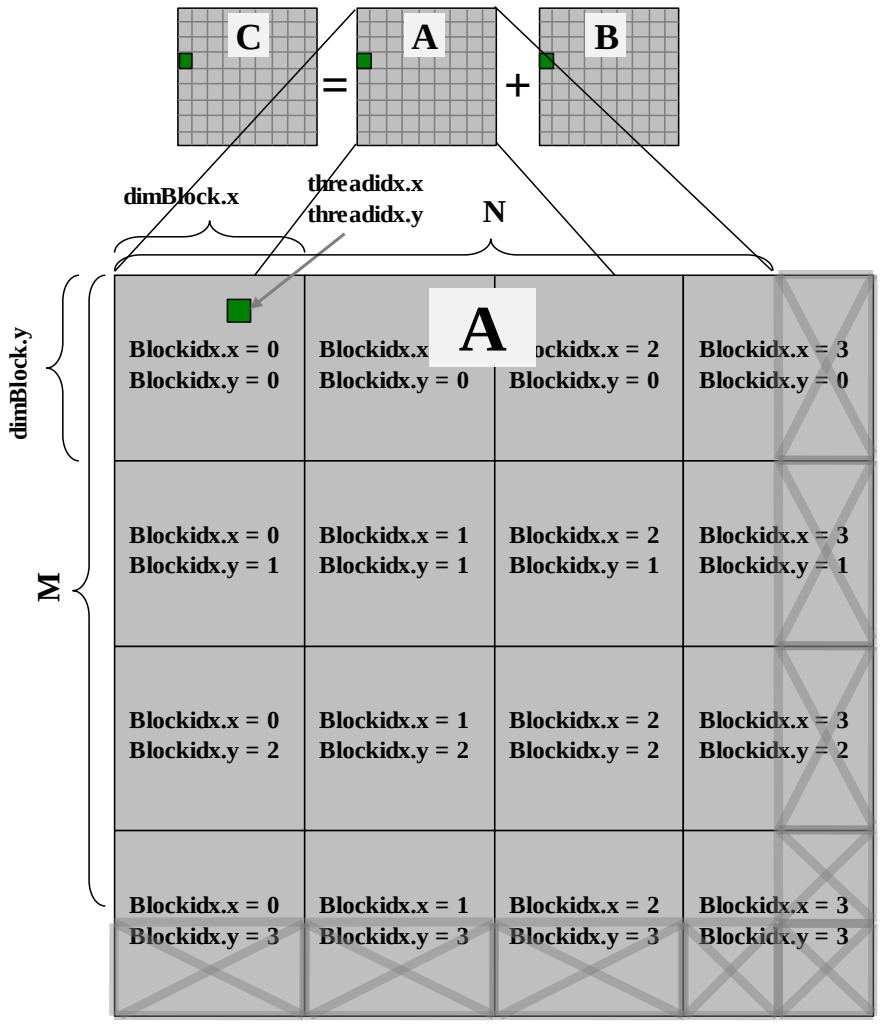


Figure 3.7:Grid layout of an example of a CUDA implemented access method to an M×N matrix using multiple thread-blocks.

Each thread executes kernel code as follows. When a kernel is executed by a multiprocessor, the co-ordinates of a thread-block in the grid and the co-ordinates of the thread within the thread-block are passed to each of the active threads. In Figure 3.6c, and Figure 3.7, the grid coordinate for a thread-block are given by 'blockIdx.x' and 'blockIdx.y', and thread coordinates are given by 'threadIdx.x' and 'threadIdx.y'. Each active thread uses these coordinates to calculate the location in the computation space (co-ordinates of the matrices) in which the thread needs to execute kernel code. Based on that location, a thread is tasked to access a specific area of global memory and copy a 16×16 patch of matrices A and B to the multiprocessor's shared memory. The thread then computes the matrix sum for the 16×16 patch in shared memory. The results of the patch are finally written to global memory. Once all threads of a thread-block have completed execution, the multiprocessor is given the next set of grid and thread co-ordinates for the next available thread-block in the grid.

3.5.3 OpenCL

OpenCL is a framework for writing programs that execute across heterogeneous platforms and operating systems. The most important aspect of OpenCL is that it is an open standard language for parallel programming of processors found in personal computers, servers and handheld/embedded devices, and GPUs. OpenCL was originally created by Apple but is now developed by the Khronos Group [18] with the participation of many industry-leading companies and institutions. The list of institutions includes both NVIDIA and AMD, two of the leading GPU manufactures. Many other hardware

manufactures (Apple, Intel, Toshiba, Motorola) have been collaborating with the Kronos Group to include their hardware products or to contribute their expertise to growing this open standard [20].GPU code in OpenCL is similar to that written in CUDA Driver API. In fact they have virtually identical kernel code syntax[20].

Much control of the GPU hardware can be achieved by both CUDA and OpenCL. There are some very crucial aspects to multiprocessor execution that cannot be controlled, such as inter-thread-block synchronization, as will be discussed next.

3.6 Inter-Thread-Block Synchronization Dilemma

Processors within each multiprocessor can coordinate and synchronize with each other. Unfortunately, multiprocessors cannot communicate nor synchronize with each other [18], and by extension, active thread-blocks cannot synchronize with each other. In addition, CUDA offers no provision to control the sequence in which thread-blocks are executed by multiprocessors over a grid. This is especially challenging when a thread-block is launched and may require results from adjacent thread-blocks.

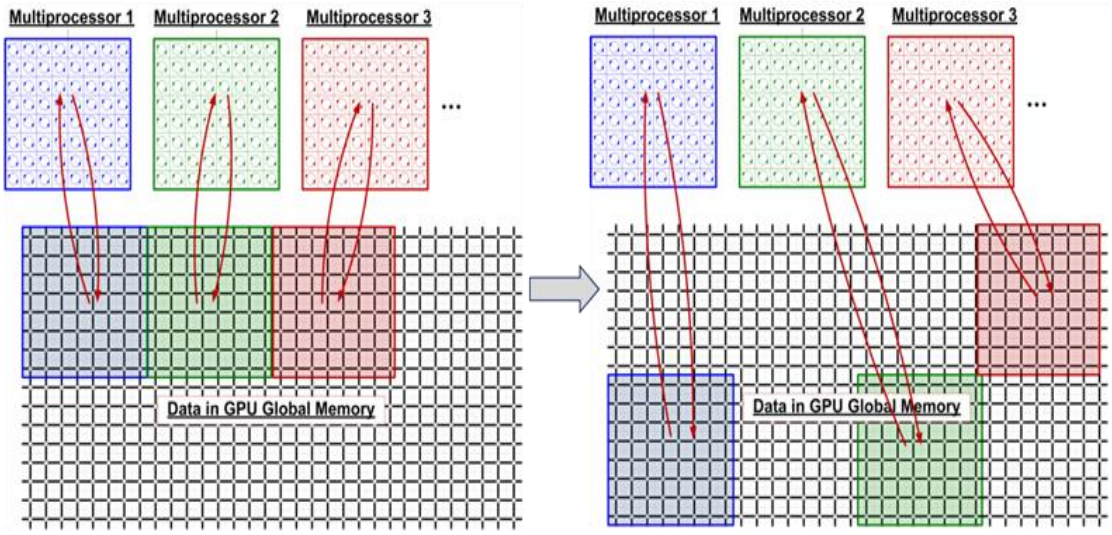


Figure 3.8:Multiprocessor tasking cannot be synchronized, nor coordinated to cover the grid in a predictable order.

An example of an inter-thread-block synchronization problem is implementing the impulse-interchange stage of the TLM algorithm. For example, a kernel may combine both the scattering and the impulse-interchange stages of the TLM method. After a thread-block executes the scattering stage, the impulse-interchange stage must exchange scattering results between all nodes. Nodes insides the thread-block can easily achieve this without difficulty since all threads in a thread-block can coordinate with each other via synchronization commands. Satisfying impulse-interchange between thread-blocks, on the other hand, is a challenge because a thread-block that is currently being executed cannot know if adjacent thread-blocks in the grid have been processed yet or not. Therefore, the kernel must be designed to operate within the synchronization limitations of the GPU without violating the requirements of the desired algorithm. As will be shown, various strategies have been employed for the TLM code with varying success to overcome synchronization limitations.

3.7 Internal Thread-Block Synchronization

Unlike the problem of inter-thread-block synchronization limitations just described, threads within a thread-block can synchronized with each other[17].Figure 3.9 illustrates an example. Five stages of a GPU adapted TLM kernel are depicted with synchronization commands embedded between the stages. The synchronization command forces threads that reach this command to wait until all threads of the thread-block have also reached this command before proceeding.

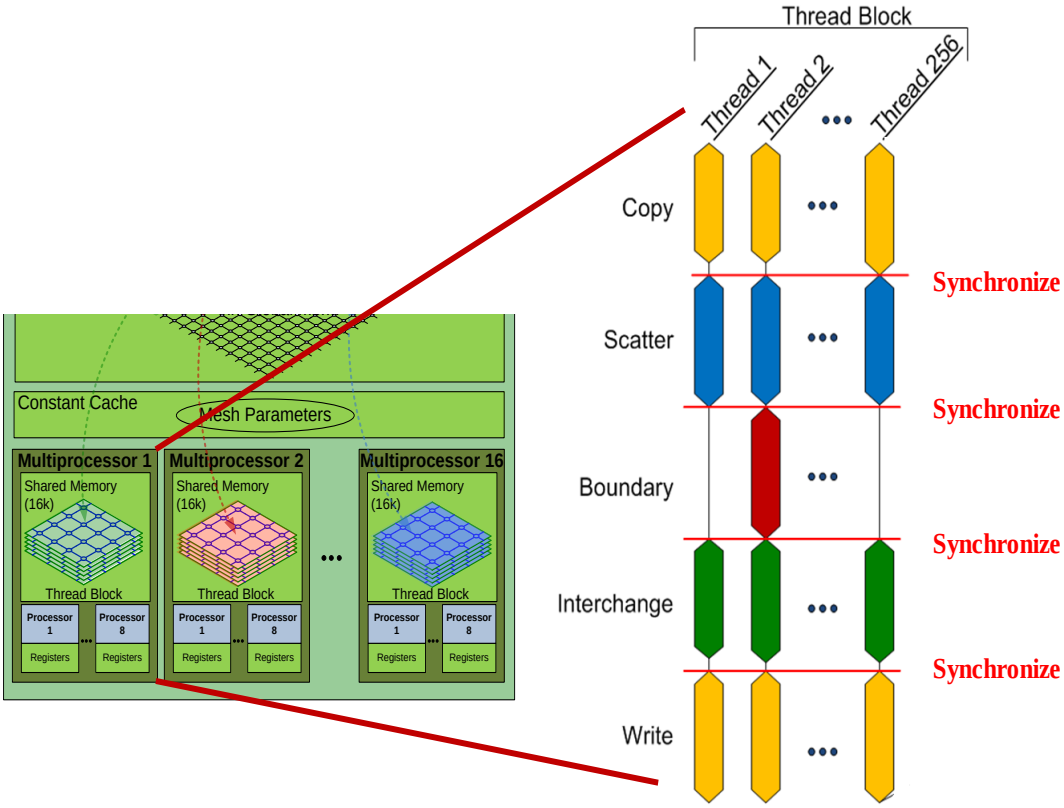


Figure 3.9:Synchronization between threads of a thread-block can be achieved by inserting synchronization commands between stages of computation.

3.8 Memory Coalescing

In the majority of cases, memory access (between global memory and multiprocessors) can be the most prevalent bottleneck for kernel execution time. Therefore, much attention is paid to employing memory optimization techniques to speed up memory access wherever possible. Memory coalescing is one such performance enhancing technique, where access to global memory by a GPU's multiprocessors can be accelerated an order of magnitude faster than standard memory accesses times [33]. In general, access to global memory by any of the multiprocessors requires 400-600 clock cycles of latency for each four-byte data type such as integers or floating-point numbers. As compared to 4-6 clock cycles to access shared memory or registers.

Global memory is arranged physically into banks, 64 bytes in length. The coalescing technique arises from the ability of multiprocessors to access memory of an entire bank at a time. So, floating-point numbers or integers (four bytes in length) can be accessed 16 at a time, all within 400-600 clock cycles. To access a memory bank each multiprocessor uses what are called warps. The details of GPU warps are given in the following section, but for the time being a warp can be understood as a grouping of 32 threads within thread-blocks. The coalescing technique requires three conditions [18]:

1. The starting address of each half-warp (16 threads) falls on a 64 byte interval in global memory
2. Each thread of a half-warp reads/writes 4, 8 or 16 bytes consecutively

3. Threads within each half-warp must be spaced, in consecutive order, at 4, 8 or 16 byte intervals

Figure 3.10a, illustrates an example of a coalesced half-warp (16 threads), where each thread of a half-warp reads/writes 4 bytes at time from global memory (i.e. floating-point values). The three coalescing conditions are satisfied where: (1) the starting thread, t_0, begins at a 64 byte interval, (2) each thread accesses one floating-point or integer at a time (4 bytes), and (3) each thread is spaced 4 bytes apart. Figure 3.10b, also satisfies the coalescing condition despite several threads being blocked from accessing memory. The example illustrated in Figure 3.10b, will be shown as a valuable technique in one of the TLM kernel designs (chapter 8). If the kernel code is programmed in such a way that coalescing is achieved, the GPU hardware detects the coalescing condition automatically and reads/writes entire banks of 64 bytes (16 floats) in one 400-600 clock cycle transaction.

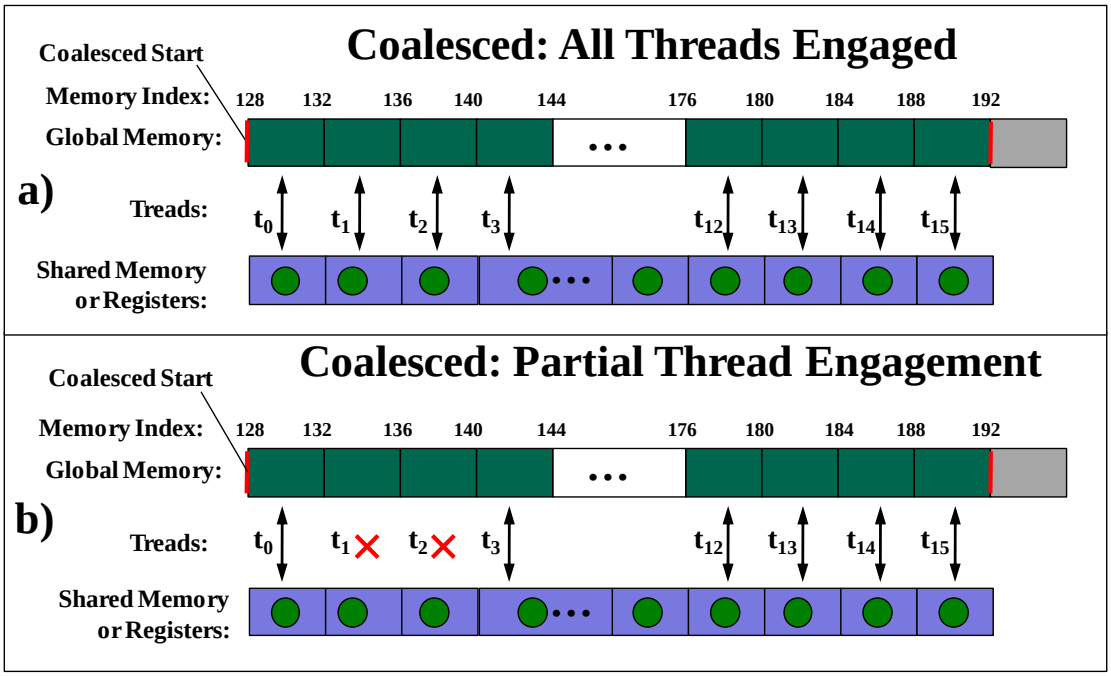


Figure 3.10:Global memory coalesced access examples.(a) all thread of a half-warp (16 threads) engaged in memory access. (b) coalescing is still achieved with partial thread engagement.

Figure 3.11 illustrates two examples of non-coalesced memory accesses. Even when half-warps are aligned, if one or more threads do not align consecutively, as in Figure 3.11a, then coalescing fails. Figure 3.11b, illustrates another non-coalesced circumstance if the starting thread is at a memory address that is not at a 64 byte interval.

Efficient use of multiprocessor resources and arranging global memory transfers in a coalesced approach was helpful to achieve close to the maximum memory transfer rate published by the hardware manufacture. Memory transfers between global memory and the multiprocessors is an important benchmark for GPU performance.

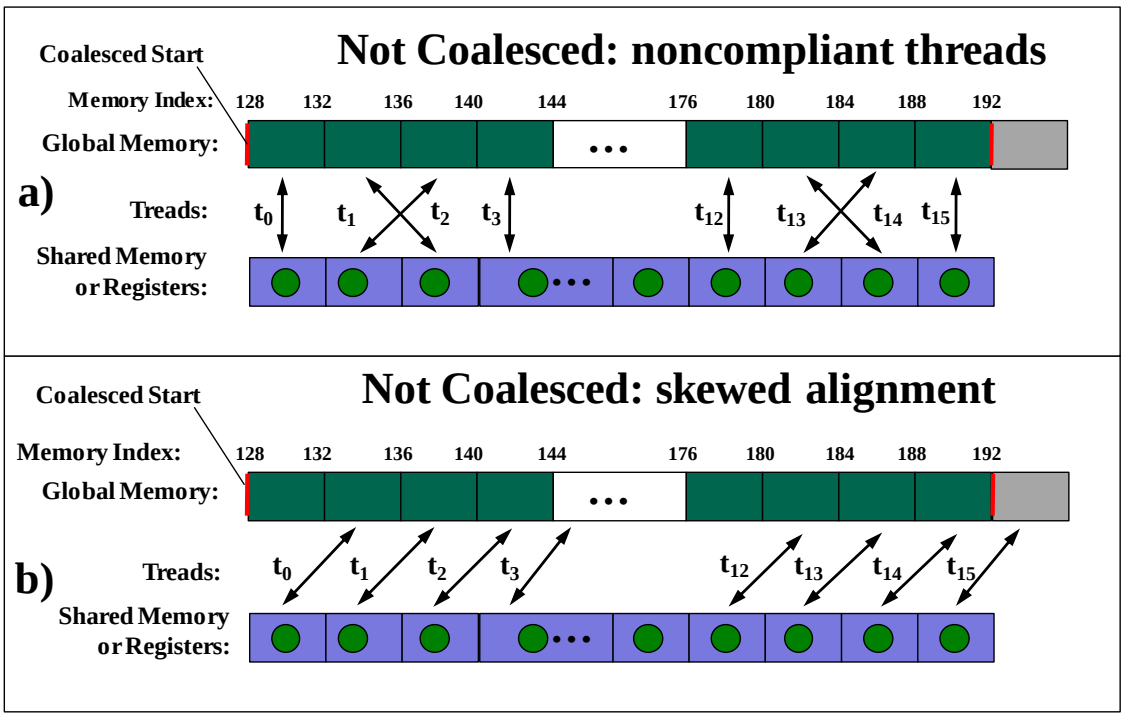


Figure 3.11: Non-Coalesced examples. (a) threads within half-warp are noncompliant to coalescing conditions. (b) half-warp is skewed from the 64 byte interval condition for coalescing.

3.9 Advanced GPU Optimization Techniques

3.9.1 GPU Warp Definition

Many of the following sections that describe advanced optimisation techniques require an understanding of GPU warps. As was mentioned previously, a GPU warp is defined as a grouping of 32 threads [33] and each thread-block may have up to 512 threads defined. To execute all the threads in a thread-block, each multiprocessor must execute one warp (32 threads) at a time; i.e. a multiprocessor partitions thread-blocks into warps and then executes each warp in turn. It is because of the way multiprocessors execute

warps, that thread-block dimensions and resource usage can greatly affect GPU performance.

3.9.2 Occupancy

Occupancy (expressed as a percentage) is a measure of how well a GPU utilizes its resources [33]. The higher the occupancy, the more likely each multiprocessor executes kernel code and not sit idle or be stalled. Occupancy is calculated by determining the number of warps that are able to execute on each multiprocessor, divided by the number of available warps each multiprocessor is designed to process. The Quadro FX5600, for example, is designed to allow 24 warps to be executed on each multiprocessor (Figure 3.12a). These 24 warps (with 32 threads each) are equal to 768 thread-slots. Thread-slots can be thought of as an available resource of possible active threads on each multiprocessor, for which multiple thread-blocks can occupy[18]. For example, if a thread-block is defined as 256 threads (Figure 3.12c), then three thread-blocks can fully occupy a multiprocessor's 768 thread slots (100% occupancy). Another example is if a program defines 512 threads per thread-block, which is the maximum allowed per thread-block, (Figure 3.12b)then only 66.7% of the available warps can be launched (66.7% occupancy) since only one 512 thread thread-block would completely fit into the 768 available thread slots.

The value of occupancy is revealed when one examines how a multiprocessor processes each thread-block. Each multiprocessor loads as many thread-blocks that can

fit in its thread-slot resources(24 warps or 768 threads)It then partitions each thread-block into warps. The multiprocessor executes kernel code, one warp (32 threads) at a time. A warp can stall if it is waiting for a memory transaction to complete, or waiting for other warps of a thread-block to reach a synchronization point. The multiprocessor can avoid being stalled itself by executing the next available warp in the pool occupied warps. The more warps that are occupied (measured by percent occupancy) the more likelihood that a multiprocessor will find and execute a warp that is not stalled.

The example illustrated in Figure 3.12c, defines 256 threads per thread-block which achieves 100% occupancy. Since three thread-blocks can fully occupy each of the multiprocessor's available thread-slots, then the multiprocessor can access one of 24 warps. If some warps have become stalled, the multiprocessor can choose one of 24 warps to execute. The multiprocessor can then interleave warp execution, ensuring that it is executing kernel code as much as possible.

In order to meet the coalescing conditions mentioned earlier, and attain high occupancy, thread-block dimensions must meet two conditions: the thread-block must be multiples of 16, and it must divide into 768 evenly. There is a lower limit of 8 thread-blocks imposed by the hardware to fit inside the available 768 thread slots. This means that the smallest thread-block dimension that can achieve 100% occupancy as well as meet coalescing conditions is 96.

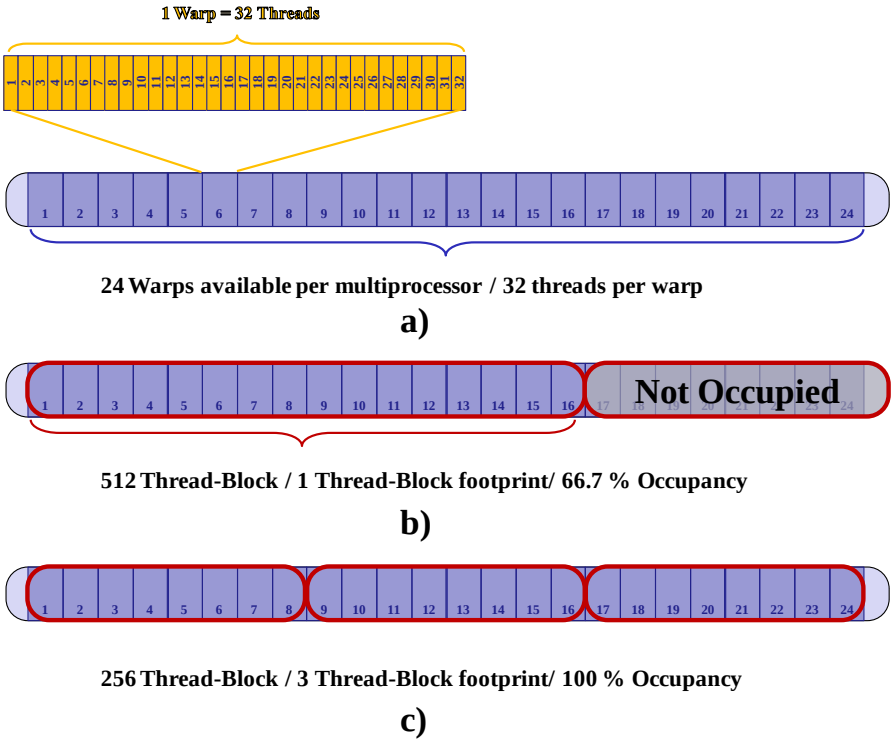


Figure 3.12: Thread-Block Occupancy Examples. (a) 24 Warps/multiprocessor.(b) 512 thread per thread-block = 66.7% occupancy.(c) 256 threads/thread-block = 100% occupancy (or all 768 thread are active).

3.9.3 Occupancy and GPU Resources

Thread-block dimensions are not the only parameter that influences occupancy. GPU memory resources also must be managed appropriately to ensure high occupancy. Resources that impact occupancy are: registers and shared-memory. As was discussed in the previous section, the number of available thread-slots depends on the GPU hardware (768 in our case). Register usage and shared memory usage must be distributed across all threads that occupy the 768 thread-slots. The complexity of the kernel code determines the amount of shared memory or registers required per thread. Figure 3.13a, illustrates a thread-block of 256 threads in which its kernel code requires 3.5k of shared memory per thread-block. The thread-block dimension can achieve 100% occupancy because three of

these thread-blocks are able to fit into the 768 thread-slot footprint evenly, and the total amount of shared memory used (3×3.5k = 11.5k) is less than the 16k available. Figure 3.13b, shows the consequence using more share-memory (6k) than the previous example. Over usage of shared memory resources, in this case, drops occupancy to 66.7%, reducing the number of available warps from 24 to 16, and consequently diminishing performance. The same argument can be made for the usage of register memory (8k), which as a resource, must be distributed across all threads that occupy a multiprocessor's thread-slot resources. Carefully coding a kernel to make the most efficient use of thread resources as well as memory resources can satisfy all the aforementioned occupancy requirements. In combination with coalesced memory access, a kernel can achieve the maximum theoretical memory transfer rate.

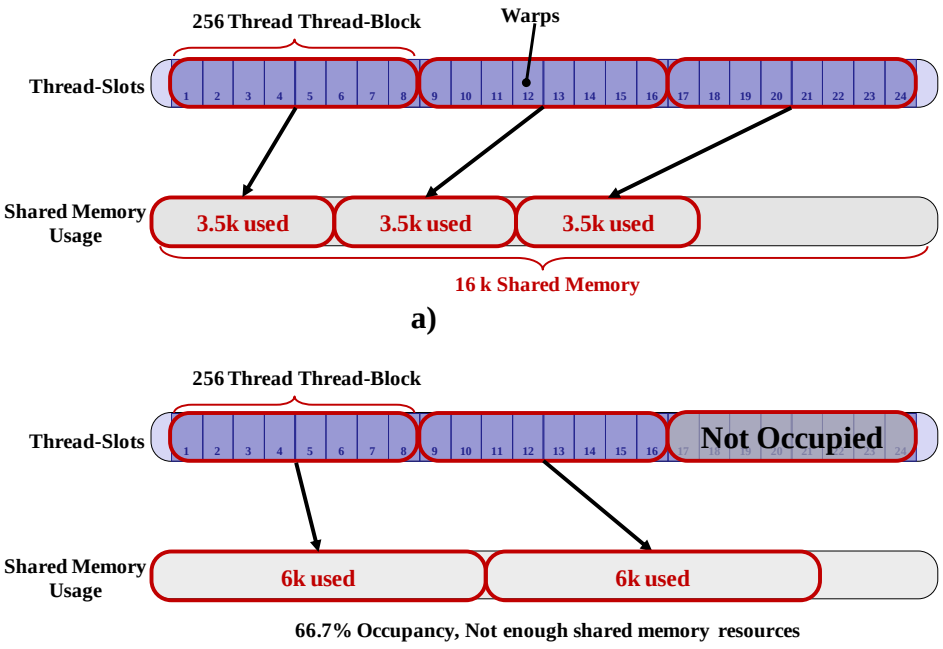


Figure 3.13: Impact of share memory (16k) on occupancy. (a) 100% occupancy as there is enough shared memory to span thread-slots. (b) 66.7% occupancy since there is not enough shared-memory to span all thread-slots.

3.9.4 Memory Pre-fetching

Memory pre-fetching is a technique that employs partial memory latency-hiding by designing the kernel such that memory access commands are issued at strategic points. Figure 3.14a, shows an example of what is usually done to compute two computations (‘A’ and ‘B’) that must be executed consecutively. The first computation requires two values from global memory (‘A1’ and ‘A2’). A synchronization command must be issued before calculating ‘A_result’ to ensure the global memory read transactions are completed. The next set of computations (‘B_result’) has the same order (read,

synchronize, calculate), but ‘A_result’ must be used in the computation of ‘B_result’. In this case the computation of ‘A_result’ must be completed before ‘B_result’.

Figure 3.14b, show the same routine, but rearranged to employ pre-fetching. When the multiprocessor completes reading ‘A1’, and ‘A2’ (and just after a synchronization command), it immediately issues a command to read ‘B1’, and ‘B2’just before calculating A_result. Note that a synchronization command does not exist between the global memory read and the calculation. Without the synchronization command, control is immediately returned to the executing threads while a global memory read transaction takes place. The multiprocessor commences the calculation of ‘A_result’ at the same time as ‘B1’ and ‘B2’ are being read from global memory [34].This simultaneous action hides the time it takes to access global memory. The strategy of employing pre-fetching is to anticipate global memory reads by issuing global memory read command just prior to a preceding computation. A similar process can be done for global memory write transactions.

Calculation A & B

```
// First calculation
float resultA,resultB = 0;
float A1 = global_data_dev[address_A1 + thread_id];
float A2 = global_data_dev[address_A2 + thread_id];
__syncthreads();

result_A = A1 + A2;

__syncthreads();

// Second Calculation
float B1 = global_data_dev[address_B1 + thread_id];
float B2 = global_data_dev[address_B2 + thread_id];
__syncthreads();

result_B = result_A + B1 + B2;
__syncthreads();
```

Calculation A & B with Pre-fetching

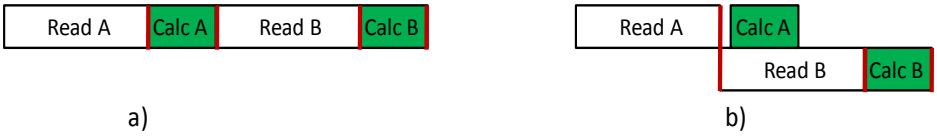
```
// First calculation
float resultA,resultB = 0;
float A1 = global_data_dev[address_A1 + thread_id];
float A2 = global_data_dev[address_A2 + thread_id];
__syncthreads();

float B1 = global_data_dev[address_B1 + thread_id];
float B2 = global_data_dev[address_B2 + thread_id];

result_A = A1 + A2;

__syncthreads();

// Second Calculation
result_B = result_A + B1 + B2;
__syncthreads();
```



a)

b)

Figure 3.14: (a) computation of A & B by sequential operations: copying from global memory, synchronization, calculation A, copy, etc... (b) the same calculation but rearranged to pre-fetchB data while simultaneously calculating A.

Chapter 4 2D TLM Implementation

This chapter reports the design of a two-dimensional TLM program using C-for-CUDA. The discussion will first centre on the C-for-CUDA code that is run on the host. It then focuses on details of the kernel design that handles inter-thread-block synchronization for the 2D-TLM algorithm.

4.1 C-for-CUDA Host Program

Before discussing the design of the 2D-TLM kernel, it is helpful to be aware of the context of this kernel execution. A short description will be given on the host program which supports initialization calls to the GPU, as well as launching kernels. Figure 4.1 shows a pseudo-code listing of the 2D-TLM host program. The host program contains code that instantiates a 2D-Mesh within the CPU's memory in global memory of the GPU. The program then transfers the 2D mesh data from the CPU to the GPU. After the host sets up excitation and sampling parameters it starts a timer just before executing the main loop. The loop contains calls to the 2D-TLM kernel, and excitation kernel and iterates N time-steps. When the iterations of the loop complete, the timer stops, and the speed is calculated. Speed is measured a as *node-rate*, which is the number of 2D-TLM nodes processed per second (usually expressed as million nodes / second, or equivalently MNodes/sec).

```
// Pseudocode of typical 2D-TLM Host program
Main (){
    set_Mesh_Dimensions(x,y,z);
    numIter = N; // set the number of iterations

    // Setup of Exciation
    setup_Excitation(...);
    setup_Excitation_Waveform(GaussianSine, impulse...);

    // Setup of Samplint
    specify_Sampling_Probe(coordinates...);

    // Specify Boundaries
    set_Boundary(coordinates...);
    set_Boundary(coordinates...);
    //...

    StartTimer(...)
    for (idx = 0 ; idx < numIter ; numIter++){
        execute_Excitation(...);
        execute_2DTLM_Kernel(...);
    }
    StopTimer(...)
    Calculate_Speed(...);// in Meganodes/sec

    extract_Mesh();
    extract_Samples();
}
```

Figure 4.1: Pseudo-code for a typical 2DTLM HOST program.

4.2 2D-TLM Kernel Code Design

The 2D-TLM node consists of four link-lines, each has its associated voltage value (Figure 4.2a).A one-to-one relationship is established between thread-block threads and TLM nodes in this design (Figure 4.2b). Other thread mappings can be considered (one thread to multiple nodes), but a one-to-one mapping is simple and straight forward, and will be used in this kernel design. Thread-block dimension is defined as 16×16 or 256 threads (Figure 4.2c). This dimension achieves full GPU occupancy, where three thread-

blocks are able to fit within the 768 available thread-slots. It will be shown that shared-memory usage limits thread-occupancy for this kernel. Figure 4.2c demonstrates how this design copies patches of 2D-TLM nodes, via thread-blocks, to shared memory in the multiprocessors. Since shared memory is much faster to work with than global memory, it is used as a workspace for thread-block execution.

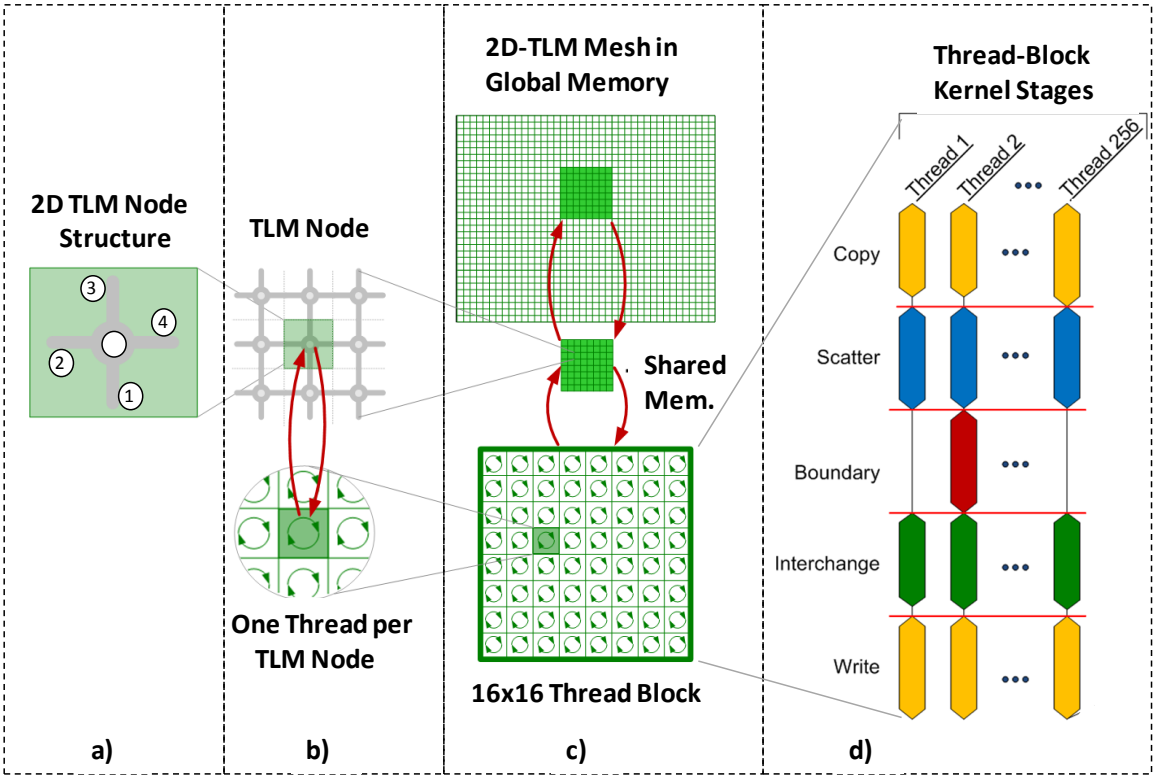


Figure 4.2: 2D-TLM mapping to GPU:(a) 2D-TLM node structure, (b) each 2D-TLM node is mapped to a GPU thread, (c) 16×16 thread-blocks executing the 2D TLM GPU kernel over a mesh of TLM nodes (shared memory use as working memory).

Five stages of execution for the 2D TLM kernel are (Figure 4.2d):

- Copy a patch of node data from the initial mesh in global memory to on-chip shared memory
- Scattering calculations

- Boundary calculations
- Impulse Interchange
- Write results from shared memory to results mesh in global memory

A grid scheme is used to ensure the entire 2D-TLM mesh is covered by the 16×16 thread-blocks (similar to the matrix addition example in chapter 3).When a thread-block in the grid is processed by a multiprocessor, *blockID* and *threadID* parameters are passed to each executing thread. Each thread uses this information to calculate the location within the 2D-TLM mesh in which the thread-block processes (Figure 4.3). Each time a multiprocessor is tasked with a specific patch of the 2D TLM mesh, it reads the four voltages of each node in the 16×16 node patch, performs the TLM calculation for each node, then writes the results to global memory [30].

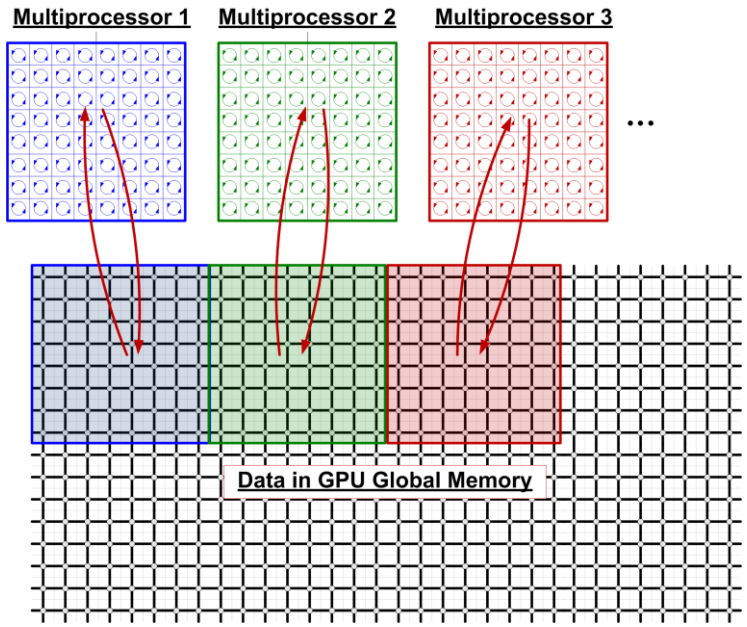


Figure 4.3: Multiprocessors tasked to access global memory using their thread-blocks. The GPU automatically schedules the sequence of the partitioning.

Within each thread-block, scattering, boundary and impulse interchange operations are synchronized. Synchronization between multiprocessors (thread-blocks), on the other hand, is handled indirectly by algorithm specific logic. Figure 4.4depicts a solution for the 2D TLM algorithm. A 16×16 node patch is copied from the mesh in global memory to the on-chip shared memory, labelled VS_1 (Figure 4.4a). Scattering is then executed on all 256 (or 16×16) nodes where scattering results are written back to VS_1 (Figure 4.4b).

Impulse-interchange is processed next where the results are transferred from VS_1 to VS_2 (Figure 4.4c). The impulse-interchange steps shown in Figure 4.4c and e, show that voltages from one node, in VS_1, are copied to its neighbour's nodes in VS_2. This *radiative-node-method* solves two synchronization problems. First, it does not matter in

which order the nodes of a thread-block execute this method. Second, the radiative-node-method solves the problem of synchronization between thread-blocks, which will be explained next.

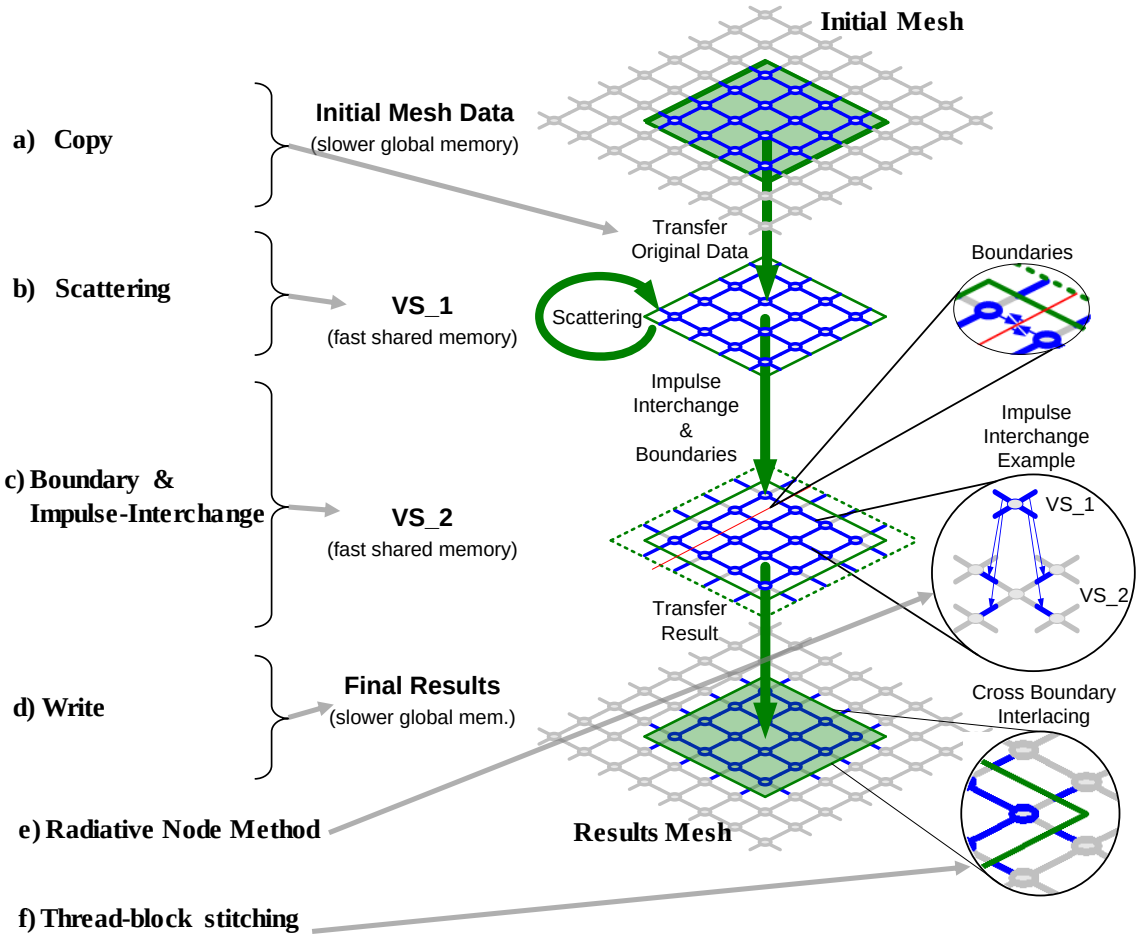


Figure 4.4: 2D-TLM Kernel execution method. (a) A 2D patch is read into shared memory (VS_1).(b) Scattering is calculated and stored back to VS_1. (c) Impulse-interchange and boundary calculations transfer to VS_2. (d) Results written back to global memory.(e) Radiative node method of impulse-interchange. (f) Thread-block to thread-block stitching.

Note that in Figure 4.4c, that VS_2 is larger than VS_1 by one dimension (18×18) in all directions. This is done for two reasons. First, the impulse-interchange stage, which uses the radiative-node-method, requires that nodes in VS_1 that are on the edges of the 16×16 thread-block, write some of their results out beyond the 16×16 confines of VS_1. The second reason involves synchronization between all GPU thread-blocks. The footprint of writing the results of VS_2 to the results-mesh in global memory is depicted in Figure 4.4f, as well as Figure 4.5a. Only voltages indicated in blue are written to global memory. Note that the voltage link-lines that are just inside the boundaries of the 16×16 block (grey) are not written to global memory. This allows for thread-block to thread-block interlacing. One important advantage seen by this technique is that all blocks over the entire TLM mesh self-stitch, (Figure 4.5b), thereby effecting inherent thread-block to thread-block synchronization. The overhead of this self-stitching radiative-node-method is lighter than compared to post kernel stitching approaches, where a second kernel would need to be created to stitch all the grid-blocks of data together.

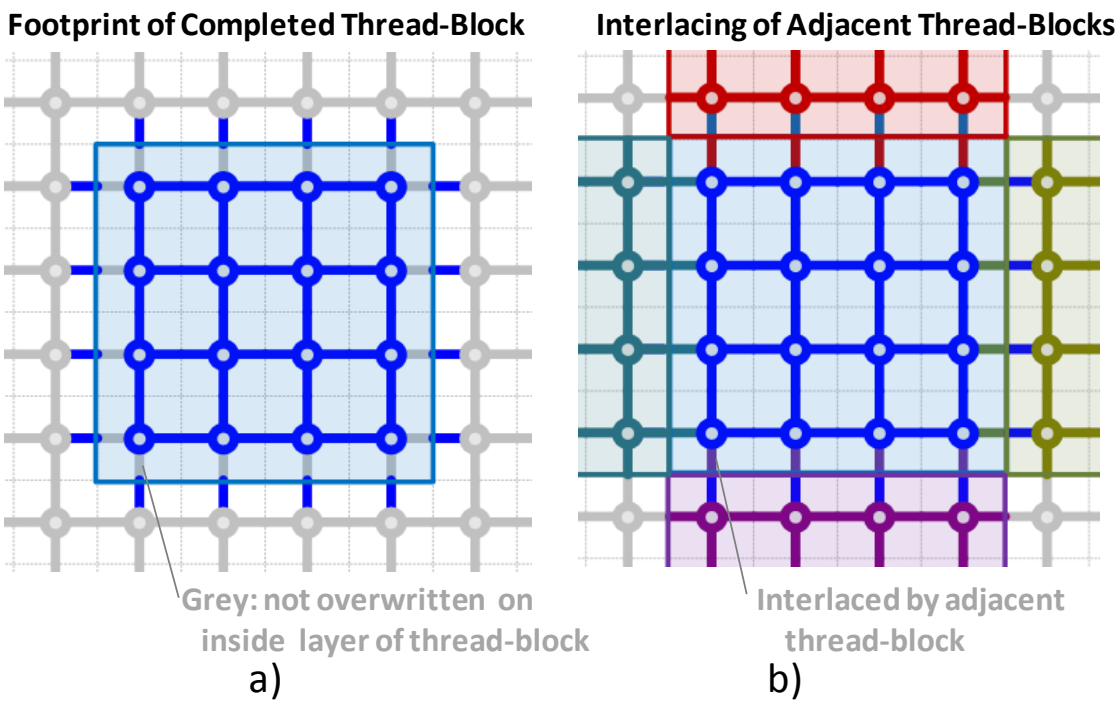


Figure 4.5:(a) Footprint of 2D TLM Results written to global memory (blue). (b) Interlacing of adjacent thread-blocks thereby achieving intrinsic synchronization between thread-blocks

A disadvantage with this method is that the results of each thread-block are not written back to the original global memory space (where the original TLM mesh was). The results, instead, are written to a blank area in global memory that is equal in size to the original TLM mesh. In other words double the memory is required. The start of the next iteration would require the kernel to rearrange pointers to swap the roles of the two memory locations.

4.3 Comparison of 2D-TLM CPU and GPU Algorithm Speeds

The performance of a 2D-TLM GPU kernel is measured by timing the execution of increasingly larger arrays of TLM nodes (Figure 4.6). As was mentioned previously, the node-rate is the number of 2D-TLM nodes processed every second. It is calculated as:

$$Node-Rate_{2D-TLM} = \frac{MeshSize \times (\#\ of\ \ Iterations)}{T_{Execution}} \tag{4.1}$$

where MeshSize is the size of the mesh in nodes, $T_{Execution}$ is the execution time in seconds, and the number of iterations executed in the host program.

The speed of the GPU version of the 2D-TLM method, shown in Figure 4.6**,**indicate that at a lower number of nodes, the code has no advantage over the CPU code. This can be attributed to the time spent by the host on overhead every iteration. A fixed amount of time is spent in the host making kernel calls as well as transferring control back and forth between the host and GPU. As the mesh size increases, more time is spent in the kernel, thereby reducing the amount of time spent in the host per iteration. The GPU, therefore, offers little benefit when the TLM mesh sizes are small, where overhead of the GPU kernels is high enough to render the GPU inefficient when compared to the CPU. As the mesh increases in size, substantial increase in performance is observed.

The speed result of an OpenMP version of the 2D-TLM algorithm is also shown (Figure 4.6) where 4 CPUs are engaged in parallel. Although there's a notable performance boost compared to the serial 2D-TLM version, the GPU version is faster

still. The GPU algorithm performs 7 times faster than the serial version, and 2.6 times faster than the OpenMP version.

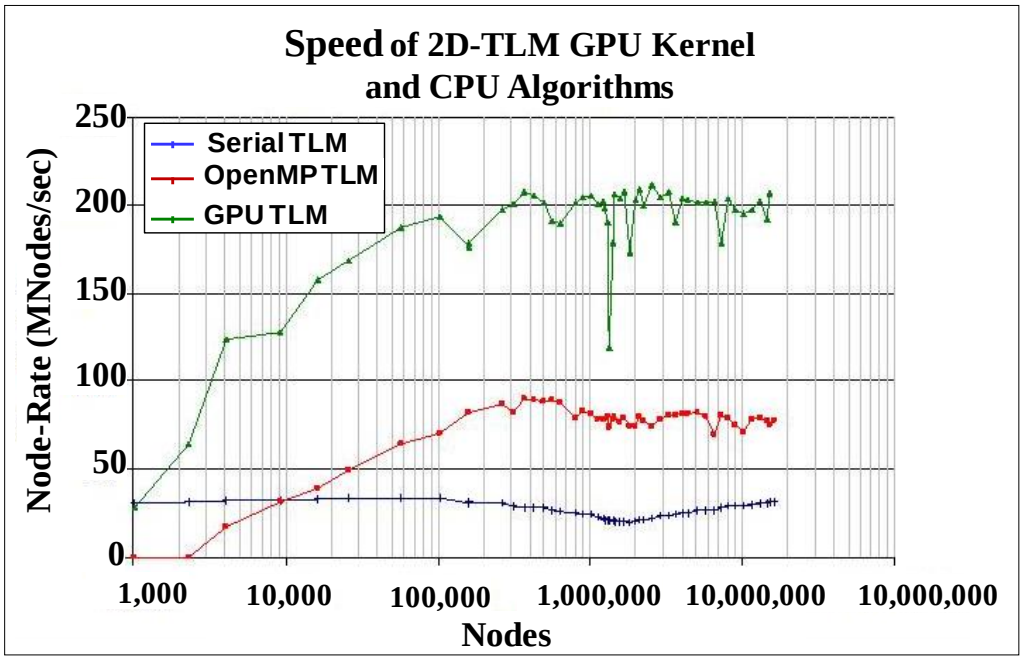


Figure 4.6:Speed of 2D-TLM algorithms executed using a single CPU (serial), four CPU cores (OpenMP), and on a GPU.

4.4 Validation of Implementation

To validate the GPU code, a simulation of a WR28 waveguide band-pass filter was modeled using the 2D-TLM CUDA code as well as using MEFiSTo. The S_{11} and S_{21} parameters of the simulated filter were measured for both programs and are shown in Figure 4.7. As can be seen, the GPU results (green) closely follow the MEFiSTo

results(red). The differences between results are also shown in Figure 4.7, where minor discrepancies are exhibited.

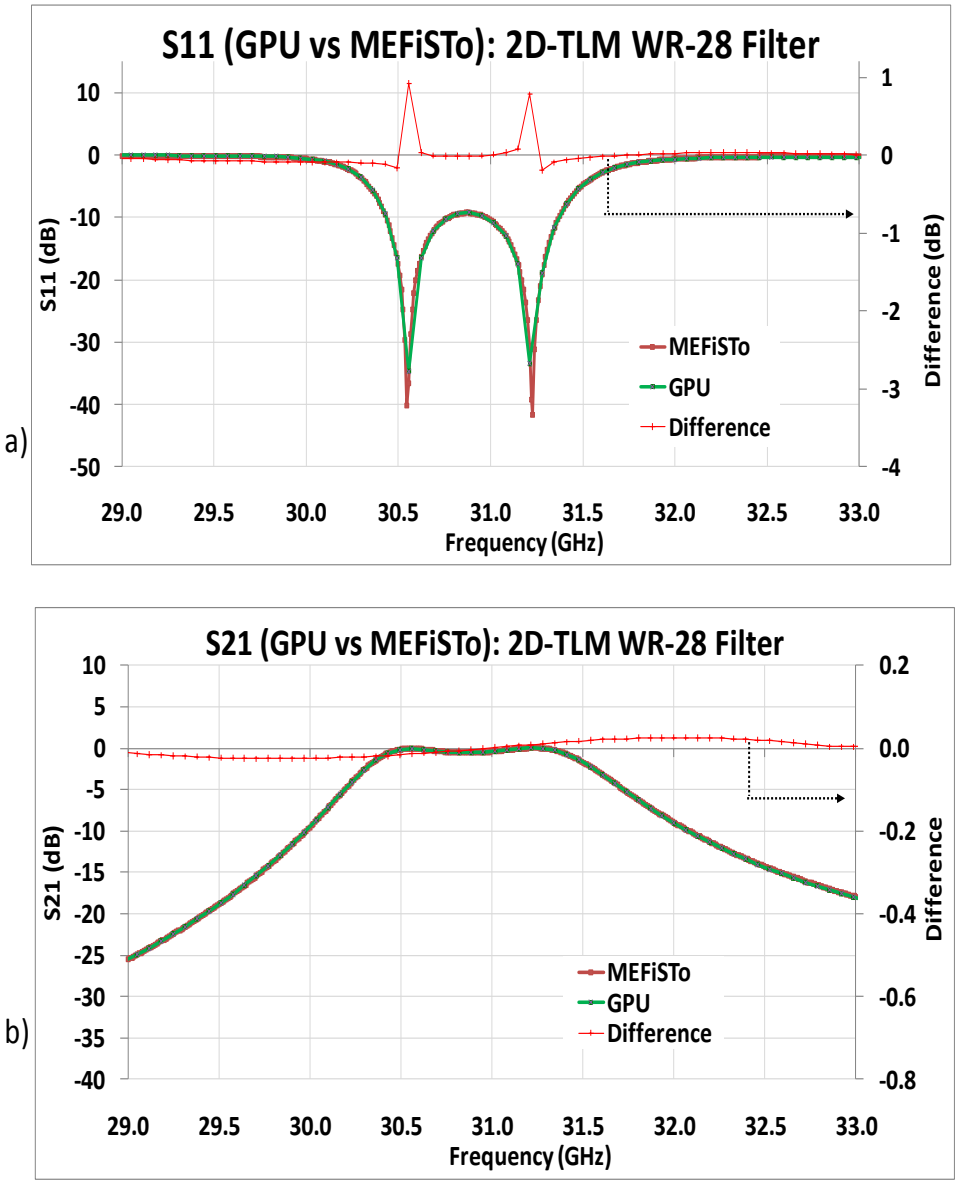


Figure 4.7:Comparison of S-parameter measurements of a WR-28 filter implemented in MEFiSTo and the 2D-TLM GPU Kernel: (a) Return loss (S11)and (b) Insertion loss (S21).

A 'C' version of the 2D-TLM algorithm was coded to execute on the CPU. OpenMP was then used to convert this serial version of the 2D-TLM algorithm to a parallel version, which utilized the 4 CPU cores available on the workstation. Figure 4.8 shows the peak node-rates for of all four versions of the 2D-TLM programs when simulating a WR-28 filter. The node-rate of the CUDA program simulating the filter is 153.9 MNodes/sec, whichiss lower than an empty mesh (210 MNodes/sec in Figure 4.6; a 27% drop. It was found that the boundary stage of execution impacted the performance of the 2D-TLM kernel.

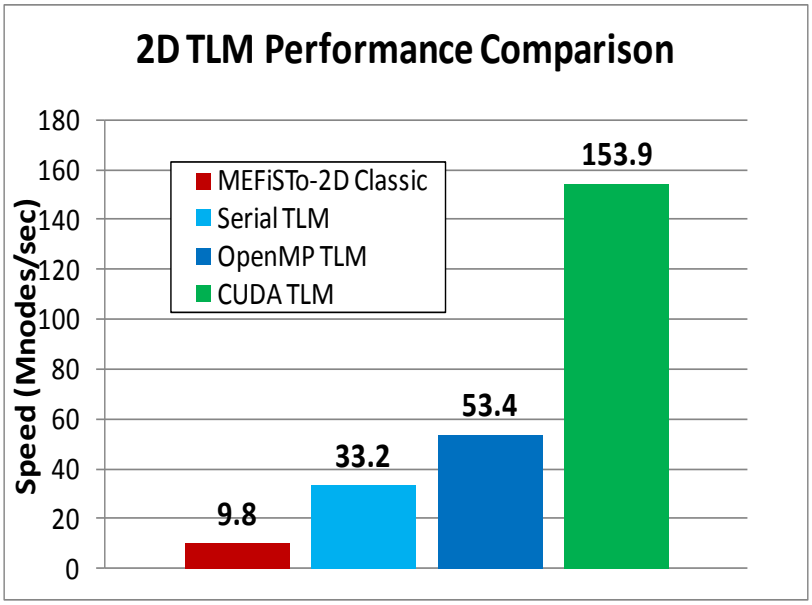


Figure 4.8: Speed comparisons of various CPU and GPU implementations of a WR-28 filters.

The boundary routine in the 2D-TLM kernel code uses a nested series of conditional statements and lookup tables with the purpose of deciding if the nodes being operated on are adjacent to a boundary. If so, the boundary reflection stage of computations executes. The node-rate plots in Figure 4.6are speed measurements of mesh structures with only

4boundaries (encompassing the mesh structure). As the number of boundaries increases, performance drops considerably. In the case of the WR-28 filter, 12 additional boundaries are necessary, which impacts speed of the kernel significantly. It was determined that conditional statements in the GPU code unravelled part of the parallelism, which is discussed further in the following sections.

4.5 Improvement to 2D TLM Contiguous Memory Model

This first implementation of the 2D-TLM GPU kernel did not include an efficient memory layout. Hence, an improvement was made by organizing memory in a contiguous manner. Figure 4.9 shows a representation of the memory model for the original kernel implementation. The placement of voltage values in global memory is not localized to each thread-block nor to each node. As well, the consecutive arrangement of voltages span across the grid. Each of the four voltage values of the TLM nodes are layered at dissimilar addresses. Therefore, using a 16×16 thread-block, a multiprocessor executing this kernel needs to retrieve data from 64 different locations in global memory.

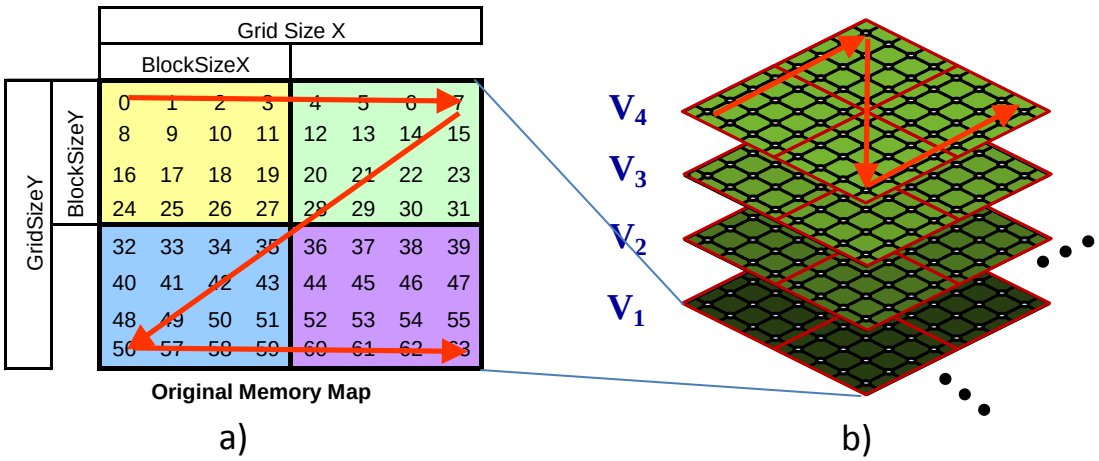


Figure 4.9: (a) An example of a2×2 grid of thread-blocks mapped to global memory. The sequence of memory spans accross the grid. (b) Each voltage of the 2D-TLM link-lines stored in stacks resulting in a non-contiguous memory orgainaization

The memory model was revised so that all the voltages of a thread-block (16×16) were stored in a contiguous and local manner. When a thread-block of a multiprocessor accesses a patch of TLM nodes in global memory using this new scheme, it is done in one contiguous read or write. Figure 4.10a and b shows an example of 2×2 grid of global memory which is organized in a contiguous manner. Figure 4.11 shows the performance results of the revised 2D-TLMimplementation where a peak performance of 340 MNodes/sec can be observed. This is a 62% improvement in speed compared to the previous implementation (210MNodes/sec).

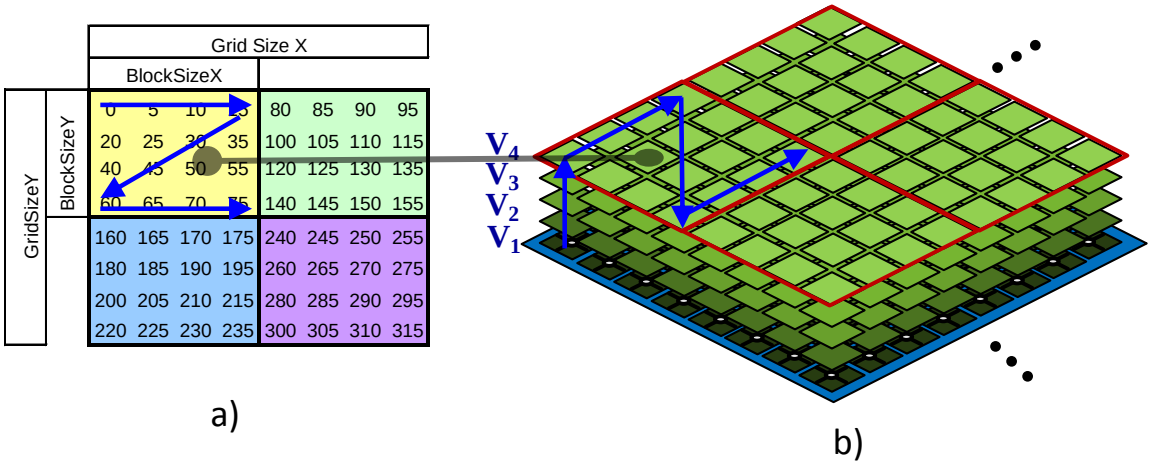


Figure 4.10:Contiguous memory organization. (a) An example of a 2×2 grid of thread-blocks mapped to global memory, where the memory sequences are confined to the mapped thread-block footprints in global memory. (b) Each voltage of the 2D-TLM link-lines stored in consecutive order.

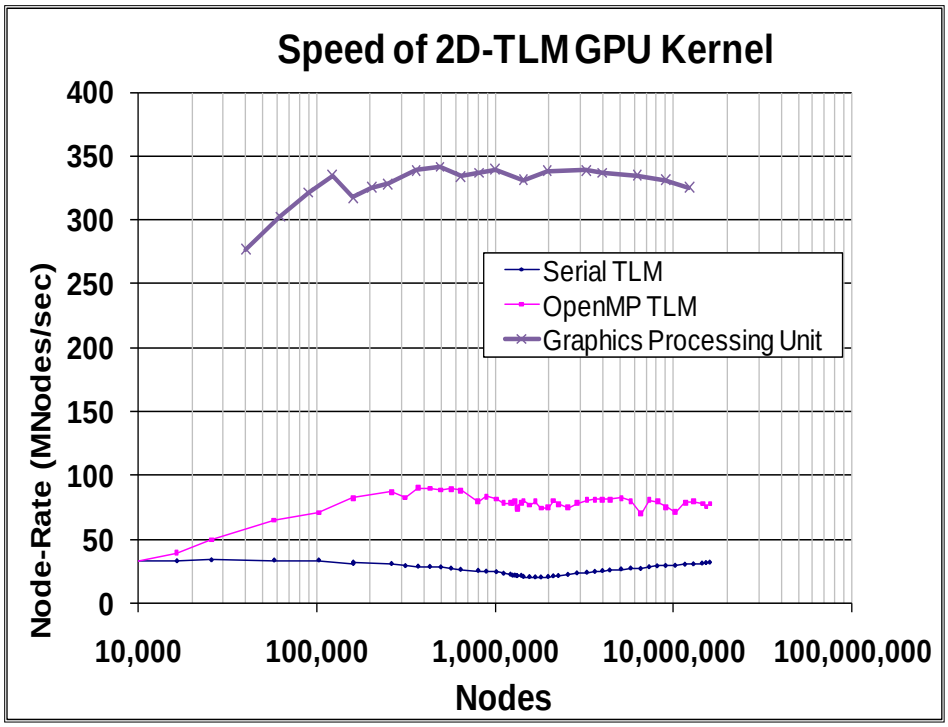


Figure 4.11: Speed results of revised 2D-TLM kernel with memory organized in a contiguous manner.

When the revised kernel code was used to simulate a WR-28 filter, the node-rate increased to 246 MNodes/sec from 153.9MNodes/sec (Figure 4.12). The speed-up of the revised implementation compared to MEFiSTo-2D Classic is 25. When compared to the OpenMP version, the new kernel is 4.6 times faster.

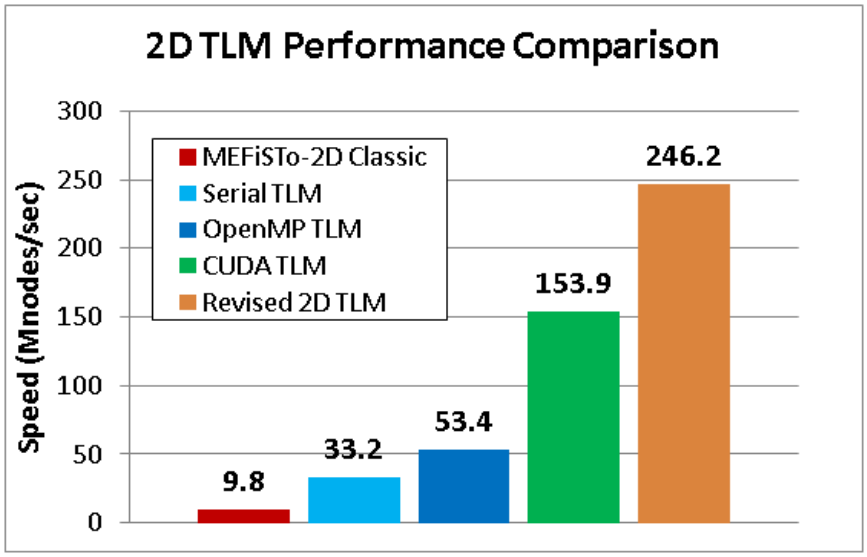


Figure 4.12: Speed results simulating a filter using the revised implementation of the 2D-TLM kernel, compared with its first version, and other implementation types (Serial TLM, OpenMP, MEFiSTo)

4.6 Warp Serialization and the Boundary Stage of Execution

As was discussed earlier, conditional statements in the boundary stage of the 2D-TLM CUDA kernel code reduces its node-rate performance significantly when even a modest number of boundaries are added to a mesh structure. The cause of this diminished performance is due to what is called *warp serialization*. This condition is caused by conditional statements in a kernel which splits execution paths of threads within a thread-block. If a conditional statement in the kernel causes one part of a warp to follow a different execution path than other parts of a warp, then the warp will become divergent and will serialize into consecutively executing threads. Some of the threads stall and wait while others complete. For the 2D-TLM kernel, conditional statements test each node to determine if it is adjacent to a predefined boundary. Once a boundary is detected it also tests for boundary direction (up, down, left, right), as well as a search for the appropriate

reflection coefficient in a look-up table. These nested conditional statements compounds the problem of warp serialization when any part of a boundary falls within a warp, and a cascade of serializations occur that unravel parallelism of the thread-block and reduces the performance of the kernel. This condition contributed to the 27% decrease in speed reported earlier (figures 4.5 and 4.7).A solution to overcome warp serialization is presented in the following chapter.

4.7 Occupancy Analysis

The 2D-TLM kernel design uses 256 threads per thread-block with the intent of achieving 100% occupancy. But, the occupancy that was measured was actually 33%. This low result is attributed to the amount of shared memory each thread-block of this kernel design requires. The shared memory variable VS_1 of the 2D-TLM kernel design (see Figure 4.3) consumes 4096 bytes and VS_2 consumes 4624 bytes, with a total of 8720 bytes of shared memory used. This is under the 16k of shared memory available, but it reduces occupancy such that only one thread-block is able to occupy each multiprocessor's thread-slice resources (Figure 4.13). This results in a 33% occupancy. As was demonstrated in the node-rate results of the 2D-TLM kernels, even with less than optimal occupancy, the execution speed of this kernel is much faster than the CPU based versions.

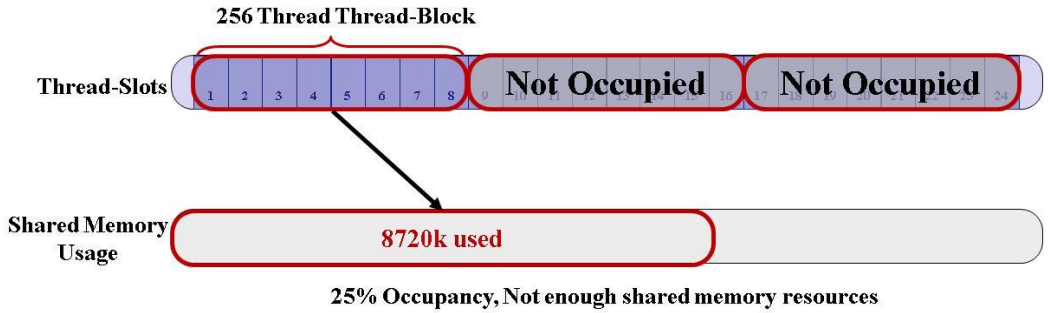


Figure 4.13: The occupancy of the 2D-TLM kernel is reduced to 33% from 100% because shared memory uses more than half of a multiprocessor's resources, which excludes 2/3 of the available warps to be utilized.

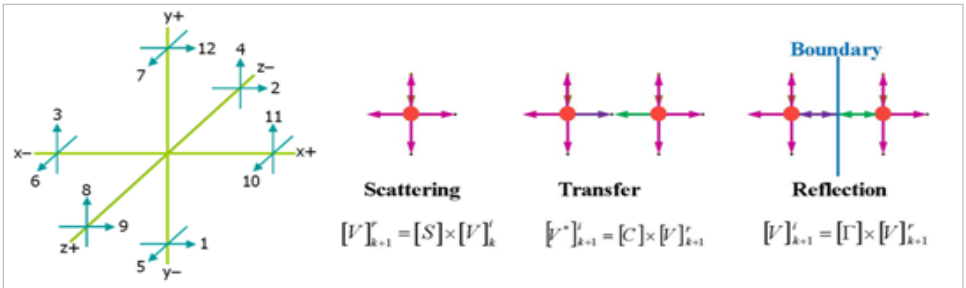
The 2D-TLM kernel was the first attempt at adapting the TLM algorithm to the GPU paradigm; hence the program engaged none of the optimization techniques outlined in chapter 3. These techniques are employed to implement the GPU based 3D-TLM program discussed in the following chapter.

Chapter 5 3D-SCN Implementation: First Design

The experience obtained from designing the 2D-TLM kernel became useful when porting a CPU based 3D-SCN program to the GPU environment. The memory resource for each three-dimensional node is three times that of a 2D-TLM node (12 vs. 4 voltage link-lines). Since the 2D-TLM kernel design consumes over 50% of shared memory (and resulted in reduced occupancy), it is clear that resource management is a central focus for any future kernel design. Therefore, a different approach is discussed in the following sections to diminish memory resource usage of a 3D-SCN kernel.

5.1 3D-SCN to GPU Adaptation

The stages of calculation for the 3D-SCN method are fundamentally the same as the 2D case (scattering, impulse-interchange, and boundary reflections). Figure 5.1lists 12 3D-SCN scattering equations for 12 of the voltage link-lines of a single node. Similar to the 2D-TLM kernel, a one-to-one relationship is established between 3D-SCN nodes and threads. Therefore, the method requires that a mesh of nodes be partitioned into 3D cubes of the same dimension as the 3D thread-block. Each thread within the 3D thread-block are mapped to a node in the partitioned cube, where each thread would carry out all stages of 3D-SCN calculations.




```
SCN_Scatter(){
    V( 1,X,Y,Z) = 0.5 * ( V02 + V03 + V09 - V11 )
    V( 2,X,Y,Z) = 0.5 * ( V01 + V06 - V10 + V12 )
    V( 3,X,Y,Z) = 0.5 * ( V01 + V04 + V08 - V12 )
    V( 4,X,Y,Z) = 0.5 * ( V03 + V05 - V07 + V11 )
    V( 5,X,Y,Z) = 0.5 * ( V04 + V06 - V08 + V10 )
    V( 6,X,Y,Z) = 0.5 * ( V02 + V05 + V07 - V09 )
    V( 7,X,Y,Z) = 0.5 * (-V04 + V06 + V08 + V10 )
    V( 8,X,Y,Z) = 0.5 * ( V03 - V05 + V07 + V11 )
    V( 9,X,Y,Z) = 0.5 * ( V01 - V06 + V10 + V12 )
    V(10,X,Y,Z) = 0.5 * (-V02 + V05 + V07 + V09 )
    V(11,X,Y,Z) = 0.5 * (-V01 + V04 + V08 + V12 )
    V(12,X,Y,Z) = 0.5 * ( V02 - V03 + V09 + V11 )
}
```

Figure 5.1:Sampleof 3D-SCN scattering calculations.

Since the memory requirements for the 3D-SCN method is three times that of the 2D-TLM method, it is thus important to find the largest thread-block volume allowed by the hardware. The larger the thread-block volume, the fewer thread-blocks needed to cover the grid. As a result, there are fewer thread-block interfaces. This would minimize the number of nodes adjacent to thread-block surfaces. The fewer nodes adjacent to thread-block surfaces, the fewer inter-thread-block data exchange operations needed during the impulse-interchange stage.

Since a maximum of 512 threads are allowed per thread-block for the GPU, the maximum allowable thread-block dimension is 8×8×8, where the percentage of nodes adjacent to this cube surface is 58 % ((512-6×6×6)/512=296/512). The amount of memory a N×N×N cube of SCN node would require is:

$$MemCube_{N\times N\times N} = (4_{bytes/\ float})(12_{floats/node})(N^3)(2_{sharedMemFootprint}) \quad (5.1)$$

An8×8×8 cube, therefore, consumes49,152 bytes of memory. Unfortunately, when a thread-block copies the cube of voltages from global memory to shared memory for local processing, 49,152 bytes exceeds the 16k available in shared memory.

Another possible dimension is 5×5×5 which would need 12,000 bytes of memory. The percentage of nodes adjacent to this cube's surface is 78.4%, which is higher than the 8×8×8 cube, but is a necessary compromise. Although the 125 thread thread-block cube (5×5×5) would fit into the available shared-memory, it consumes more than half of the available shared memory resources. The occupancy calculation for 5×5×5 cube is17%. This low occupancy was a concern in the initial design, but it was thought at the time that it was a necessary compromise in order to maximize the volume per thread-block cube.

5.2 Post Kernel Stitching:

A disadvantage to the self stitching method used in the 2D-TLM kernel is that twice the amount of memory is required to store TLM mesh values in the global memory. It also requires two temporary storage areas in the shared memory (VS_1 and VS_2). If one 5×5×5 cube of 3D-SCN nodes requires 12,000 bytes of shared memory, then more than twice this amount must be set aside for shared-memory usage, which exceeds the 16k available. Consequently, it is not feasible to have a 3D self-stitching kernel based on the

design of the previous 2D kernel. Reducing the cube dimensions further (4×4×4) would step further away from the design goals set at the outset. We have thus developed a two-kernel design: one to perform the 3D-SCN stages of calculations internal to each cube, and a second kernel to stitch all the cubes together. The two kernels work in the following manner. Kernel_1 performs scattering, boundary operations and, impulse-interchange internal to the 5x5x5 node cube. Kernel_2 performs inter-cube impulse exchanges (Figure 5.2 and Figure 5.3)[36-39].

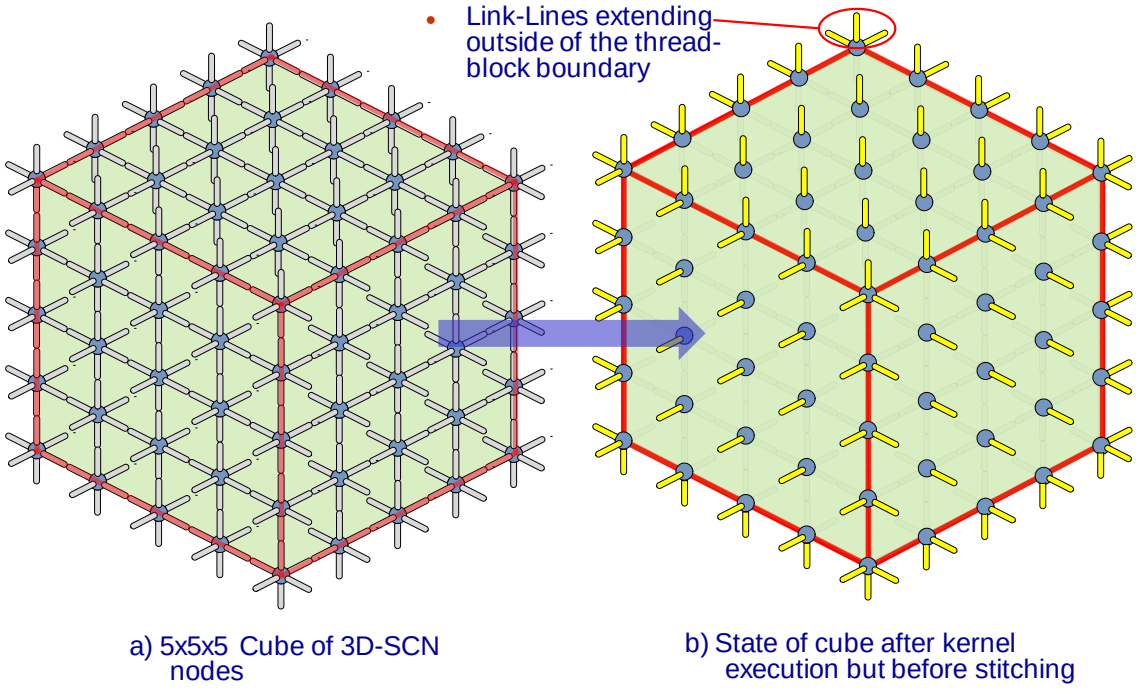


Figure 5.2:(a) A 5×5×5 cube of 3D-SCN nodes occupying shared-memory prior to being processed by the 3D-SCN kernel. (b) After kernel execution all link-lines within the cube contain their proper kernel values. But the link-lines outside the cube surface require an exchange with neighbouring nodes.

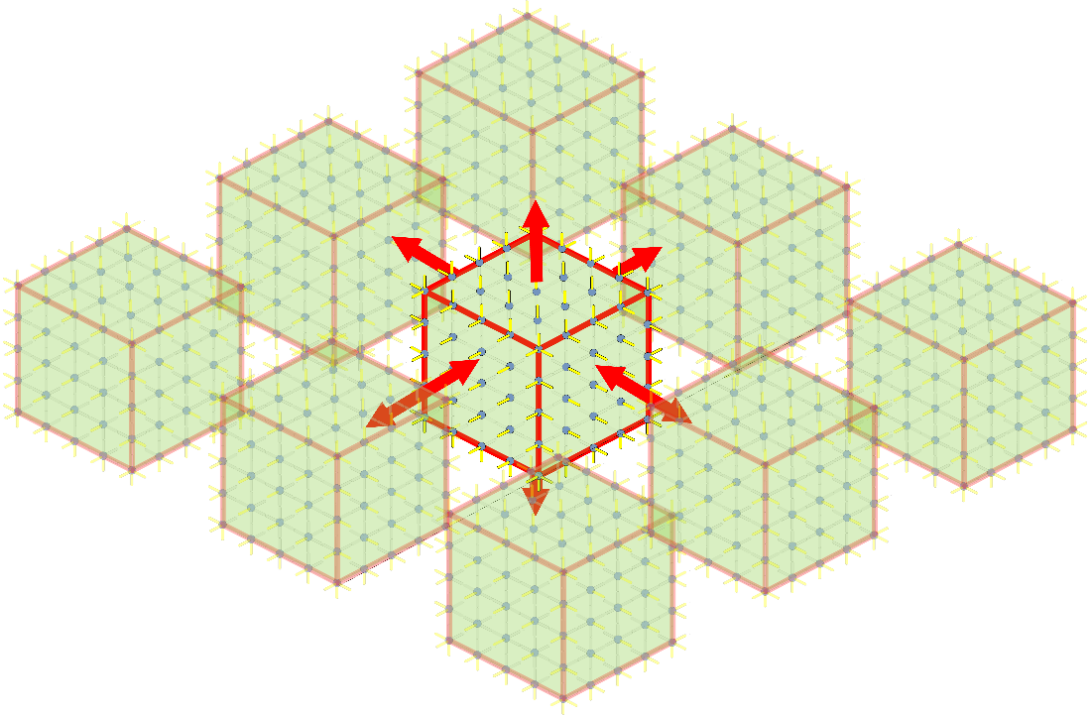


Figure 5.3:After all cubes of a mesh have had the 3D-SCN kernel operate on them, the Stitching kernel performs an inter-cube exchange of link-line voltages.

5.3 SCN Node Memory Structure

The 3D-SCN method requires 12 floating point numbers (48 bytes in total) per node. A simple array of nodes with each node occupying 48 bytes of memory would violate the memory coalesce conditions outlined in chapter 3. The first condition states that starting addresses must be at N×64 byte intervals. To satisfy the first coalescing condition, it is necessary to patch 16 extra bytes of data to the SCN node structure so that the starting address of each node would fall on a 64 byte interval (Figure 5.4). This is not a fully memory coalesced method since conditions 2 and 3 are not yet satisfied (i.e. one thread paired to 4,8, or 16 bytes, not 64). Instead, this memory padding scheme provides a modest acceleration of memory access, because partial coalescing is achieved. The node

structure in Figure 5.4is larger than 16 bytes and thus violates the second global memory coalescing condition. But it can be aligned to the nearest 64 byte address; under this circumstance an aggregation of read/writes can take place to reduce the number of memory transactions. This partial coalescing method enables the multiprocessor to generate fewer read/write transactions. In the SCN case, instead of 12 read/write transactions to process 12 voltage values, the structure shown in Figure 5.4requires only 4 aggregate read/write transactions [36-39].

```
struct __align__(16) {
        float V1;
        float V2;
        …
        float V12;
        // 16 bytes of unused memory per node
};
```

Figure 5.4:A partially coalesced memory structure is aligned to the nearest 64 byte address decreases the number of memory access transactions.

A consequence of this technique, as can be seen in Figure 5.4, is that each node structure occupies more memory than is required by a 3D-SCN node. Since the 3D-SCN node only requires 48 bytes of memory for voltage values, 16 bytes of unused memory are available for other purposes such as embedding boundary information into the node structure.

5.4 Boundary Embedding

The 2D-TLM implementation discussed in chapter 4 suffers a performance reduction of 27% when a modest number of boundaries are included for a mesh structure. For more complex structures with many boundaries, the speed decreases by 50% or more. The 3D-SCN kernel design addresses this performance problem by embedding boundary information directly into the unused area of the aligned node structure (Figure 5.5). By embedding boundary information into each node the issue of warp serialization mentioned in chapter 4 is mitigated [36-39]. An advantage of being able to embedding boundary information into each node is that irregular boundary shapes can be created. (i.e. cylinders, spheres, cones...) It must be recognized that the discrete nature of the mesh will impact the behaviour of the model.

Figure 5.5 illustrates how boundary information is embedded into the node structure. A number of restrictions are imposed by using this structure to embed boundaries. A floating-point number must be used instead of an integer because the GPU requires that all elements of an aligned structure be the same type in order to employ partial coalescing. In addition, the GPU cannot cast floating-point numbers to an integer type; therefore the embedded boundary information must be handled as a floating-point value when accessed by the GPU. A consequence of this is that only the mantissa (22 bits) of a floating point number can be utilized. The exponent section of this floating-point number is left blank in order to avoid 'Not-a-Number' (NaN) error conditions. It was found through trial and error that if a bit combination in the exponent sets a 'Not-a-Number' condition, the GPU immediately returns an error to the host.

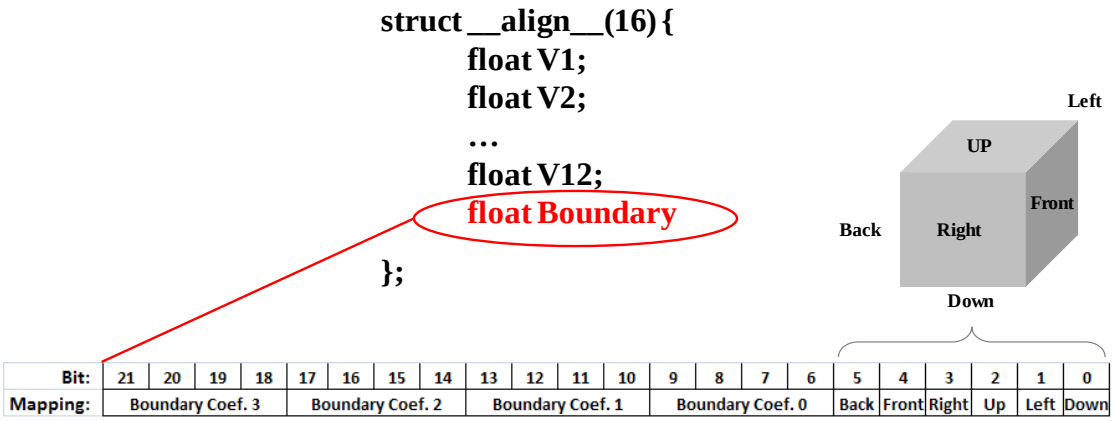


Figure 5.5:Node structure with embedded boundary information. Note that, the structure has only 13 floating-point numbers, there are still twelve bytes of used memory available for other purposes.

Speed tests revealed that utilizing the thirteenth floating-point of the node structure slowed the GPU kernel down modestly (~ 10%). Utilizing a fourteenth floating-point number slowed the kernel down further. It was decided to limit boundary embedding to just one of the floating-point values available in the node structure as a compromise to speed.

The first 6 bits of the embedded boundary information are used to define the direction in which boundaries are positioned relative to the node (i.e. up, down, left, right, forward, backward). The embedded field also contains four indexes (4 bits each) that are used to look-up the boundary coefficients (a float) in a look-up table. This boundary coefficient look-up table is created by the host at initial setup and stored in constant cache. Each boundary coefficient index could reference up to 16 coefficients (four bit addressing). 3D-SCN structures may contain perfectly electric walls (Γ=-1)perfectly magnetic walls(Γ

=1), and absorbing boundaries(Γ depends on the modes of propagation). In most practically modeling situations, the number of distinct reflection coefficients are less than 16.

A maximum of four boundary indices can fit in the embedded boundary floating-point number. This limit means that each node can be configured to be adjacent to a maximum of four boundaries. Therefore, structures are limited to ones in which all nodes must be adjacent to four boundaries or less. This limitation does not necessarily limit the complexity of structures. Provided that the defined resolution of the mesh is made sufficiently high, most structures can be modeled. Figure 5.6, illustrates a 2D case for two dissimilar resolutions. Figure 5.6 shows that for higher resolutions, no 2D nodes are adjacent to more than two boundaries. Figure 5.6b illustrates that as the resolution decreases, at some point it will be low enough that some 2D nodes must be adjacent to three boundaries or more. The 2D example illustrated in Figure 5.6can be extended to the 3D case. If the resolution of a 3D structure is low enough then a node may require 5 boundaries surrounding it. Only in structures that contain parallel boundaries one node apart will 5 boundaries be required. But if the resolution is made sufficiently high, then four boundaries are sufficient.

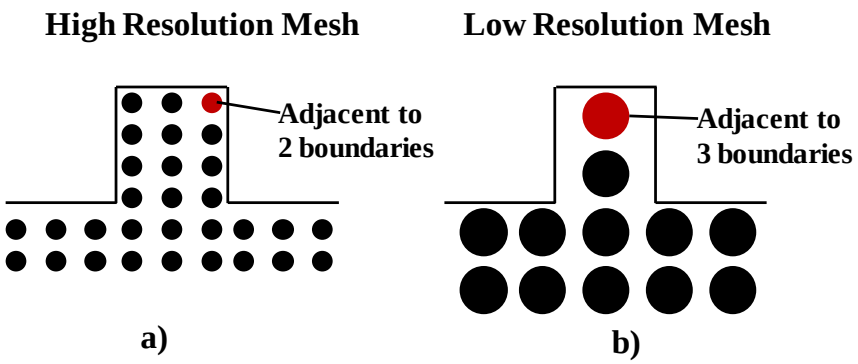


Figure 5.6: 2D example of the impact of high and low resolution TLM meshes on the number of boundaries that can surround nodes. (a) High resolution mesh with a maximum of two boundaries adjacent to any node. (b) Low resolution mesh require 3 or more boundaries adjacent to any one node.

5.5 Effect of Boundaries on Computation Speed

The boundary embedding scheme discussed in the previous section has been implemented in our 3D-TLM kernel. The code segment responsible for decoding the boundary information and execute the corresponding boundary operations is shown in Figure 5.7.

```
dir_code = (int)V_struct[tz][ty][tx].Bndry_Dir;

if(  dir_code >0  ){
     dir_down  = (dir_code & 0x1); // z = 0 , xy plane
     dir_left  = (dir_code & 0x2); // x = 0, yz plane
     dir_up    = (dir_code & 0x4); // z = const , xy-plane
     dir_right = (dir_code & 0x8); // x = const,  xy-plane
     dir_front = (dir_code & 0x10); // y=const ,  xz-plane
     dir_back  = (dir_code & 0x20); // y=0 , xz-plane
     ...
```

Figure 5.7:Sample of boundary kernel code with single conditional statement to detect boundary proximity.

As was mentioned previously, warp serialization occurs when conditional statements impose divergent computation paths within a warp. Warp serialization, however, can be diminished if all conditions within a warp are assessed at the same time for all threads. In the case of our embedded boundary scheme, each thread would read boundary information and test for a non-zero value. A non-zero value would indicate that the node is adjacent to a boundary (Figure 5.7). Once detected, the embedded boundary information would be extracted. The advantage of the embedded boundary routine is that only one condition statement is needed to test for boundary proximity, rather than testing for multiple boundaries as in the case for the 2D-TLM implementation.

The performance of the 3D-SCN GPU kernel is shown in Figure 5.8. The two plots in Figure 5.8show the best and worst case boundary scenarios. The best case scenario occurs when no boundaries are defined. The worst case is when all nodes in a mesh have boundaries defined. Under these two circumstances, the 3D-TLM kernel has a peak performance of 47 and 43.5 MNodes/sec, respectively. As can be seen from the figure the worst case scenario reduces the 3D-TLM kernel's performance only by 5%. It is informative to note that, similar to the 2D-TLM case presented in Chapter 4, the 3D-TLM kernel has better performance when the mesh size is large because the overhead of calling kernel function does not increase with mesh dimensions.

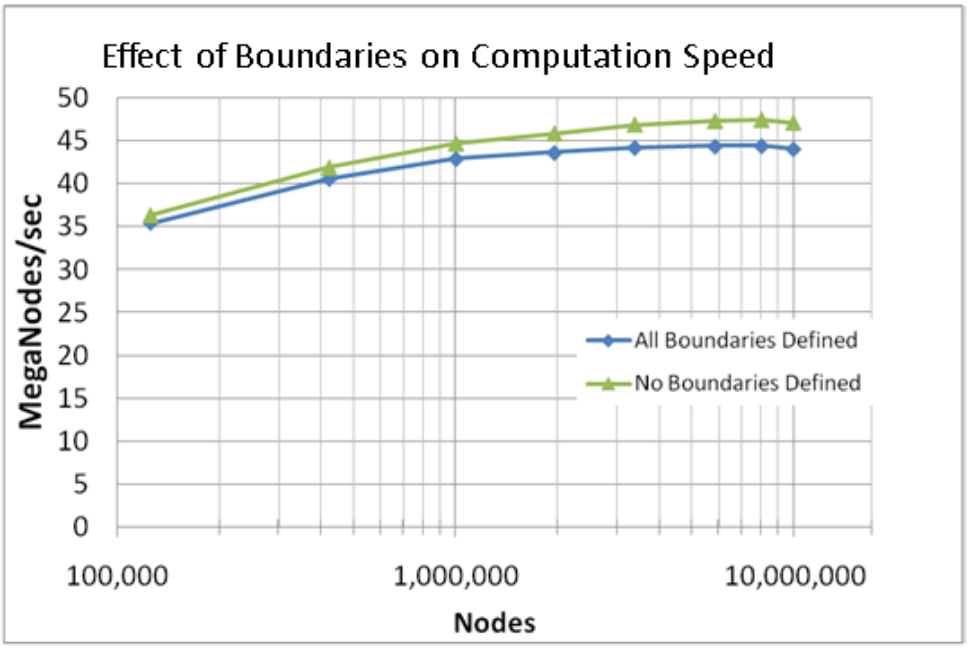


Figure 5.8:depicts two curves that show the performance of the algorithm under two extreme situations: (1) a mesh has no boundary, and (2) a mesh fully loaded with boundaries, i.e. every node in the mesh has all boundaries defined.

5.6 Validation and Speed Comparison

A C-for-CUDA application was coded that used the 3D-SCN kernel. Similar to the process of validating the 2D-TLM kernel designs, a WR-28 band-pass filter was implemented using both MEFiSTo-3D Pro as well as the 3D-SCN GPU kernel code. The S-parameters obtained with the two programs are shown in Figure 5.9a and b. Also shown in the figure are the differences between the GPU and MEFiSTo traces, where the differences are found to be negligible. These tests confirm the 3D-SCN method implemented on a GPU is correctly implemented.

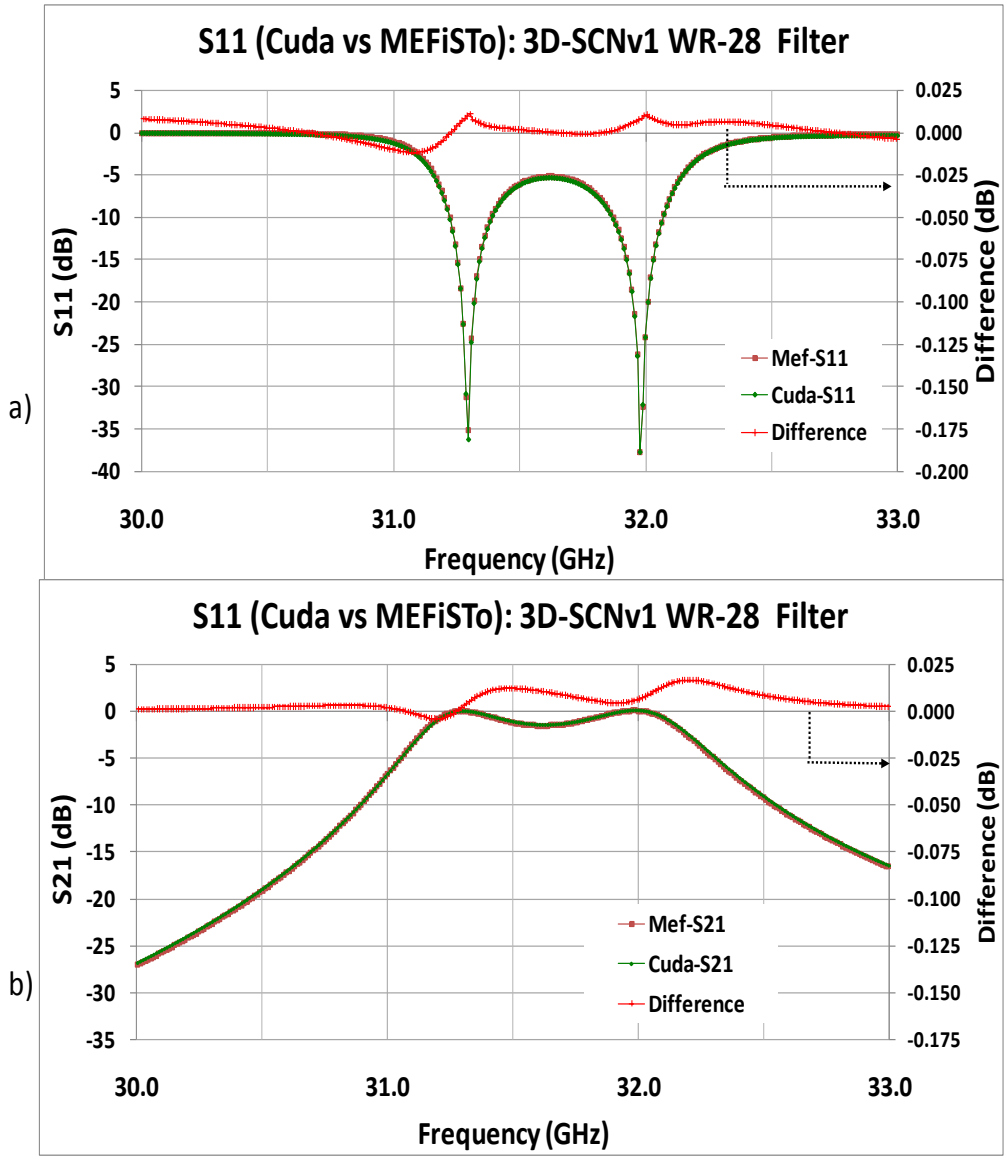
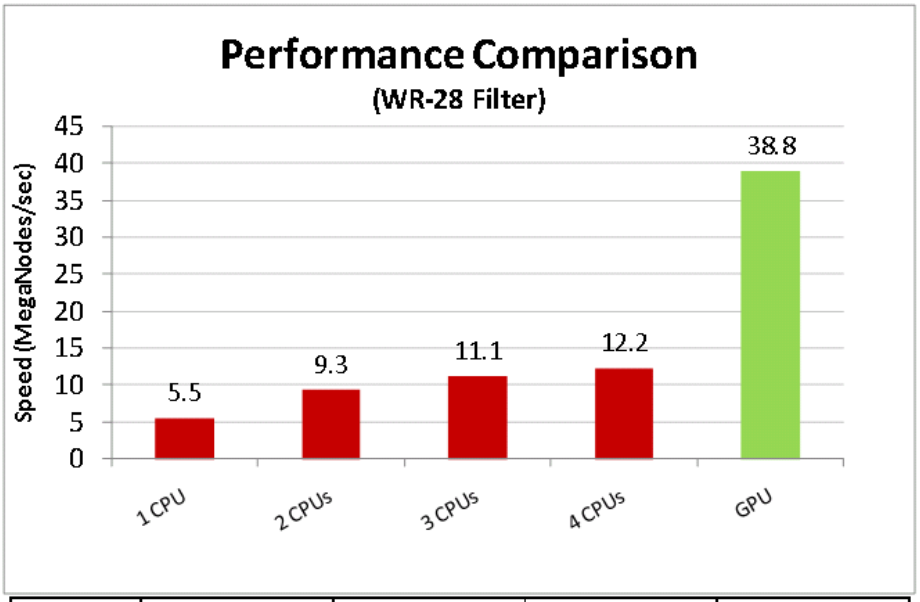


Figure 5.9: Comparison of S-parameter measurements of a WR-28 filter implemented in MEFiSTo and the first version of the 3D-SCN GPU kernel: (a) Return loss (S11) and (b) Insertion loss (S21)

The speed of the GPU version of the WR-28 filter is shown in Figure 5.10. The figure also shows the performance of MEFiSTo-3D configured to simulate the same filter. MEFiSTo-3D can engage 1,2,3 and 4 CPUs. The workstation that was used was an HP-xw9400 Workstation, 2x Dual Core Opteron Processors, 2.4 GHz, 16 GB DDR2 RAM,

1M L2 cache @ 1GHz. Compared to the filter running on 1 CPU the GPU version achieved 7.1 times speed-up. Even when MEFiSTo is running on 4 CPUs, the GPU has a3.2 times speed-up.



GPU vs.	Mefisto (1CPU)	Mefisto (2CPU)	Mefisto (3CPU)	Mefisto (4CPU)
Speed-up:	7.1	4.2	3.5	3.2

Figure 5.10: Performance of MEFiSTo and GPU when modeling the WR-28 Filter

5.7 Speed of 3D-SCN Kernel Memory Model

When the3D-SCN kernel is stripped of the TLM stages of computation, it is left with only the copy and writes stages. This is done to measure the upper bound of the algorithm based on the memory model of the kernel. The yellow trace in Figure 5.11 shows that the upper bound of the memory model for the 3D-SCN kernel peaks at approximately 60 MNodes/sec.

Node-Data-Rate is a similar speed measurement benchmark as node-rate except expressed as the number of bytes transferred per second to process a mesh of 3D-SCN nodes. It was found that the node-data-rate of the 3D-SCN kernel is 7.7 GB/sec. This is, much less than the theoretical maximum published by NVIDIA of 76.8 GB/sec (by an order of magnitude). Clearly this memory model did not take full advantage of the capabilities of the GPU. Even so, its performance was superior to that of a CPU based TLM solver.

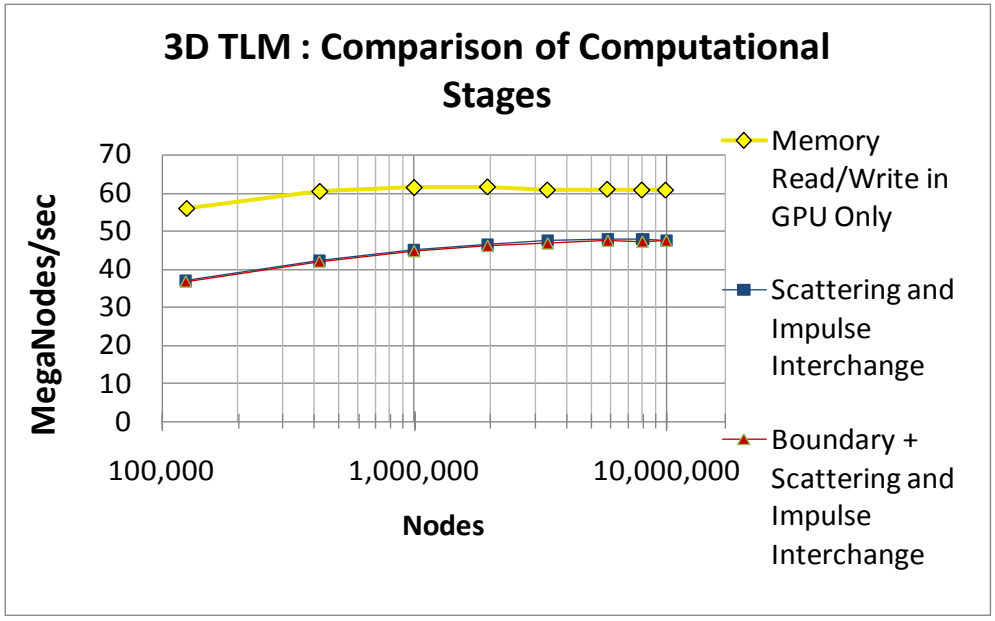


Figure 5.11:A comparison of the speeds of the 3D-SCN kernel with that of the memory model of the 3D-SCN design.

Chapter 6 Exploration of Advanced GPU Optimization Techniques

The full speed potential of GPUs are not realizable without taking full advantage of various speed-up techniques, including optimized memory access methods. The 3D-SCN kernel, described in the previous chapter, achieves only a fraction of the maximum theoretical memory access speed (one tenth) because its memory access model is not optimally designed. The goal of this chapter is to explore various memory access techniques in order to develop a framework of memory-centric kernels which approach speeds as close to the GPUs theoretical maximum as possible. The chapter after this one describes the use of this memory centric design to revise the 3D-SCN kernel to achieve much faster execution speed than the first version of the 3D-SCN kernel.

6.1 Global Memory Access Speed

The maximum theoretical bandwidth of memory transfer between a GPUs global memory and its multiprocessors can be calculated using the hardware specifications available in the product literature where [33]:

$$MaxBandwidth = (ClockRate) \times (\frac{InterfaceWidth}{8bits/byte}) \times (2directions) \qquad (6.1)$$

The clock rate of the QuadroFX 5600 is 600 MHz, its interface width is 512, and the transfer is bidirectional, therefore its maximum bandwidth is calculated to be 76.8 GB/sec.

The theoretical maximum memory transfer rate of 76.8 GB/sec for the QuadroFX 5600 reflects simultaneous copy and write transactions. In general, a 3D-SCN kernel first executes a global memory copy stage, next a calculation stage is processed, and finally a global memory write stage is done. Therefore, the copy and write stages found within the 3D-SCN kernel are not simultaneous but separate. It is more convenient to express global memory speed measurements as *effective memory bandwidth*, which measures a round trip memory transfer rate: copy + write. Therefore, the measurements made in the following sections are compared against a maximum effective memory bandwidth of the Quadro FX5600, which is 38.4 GB/sec (76.8/2 GB/sec) [33].

6.2 Memory-Centric Kernel Setup

A memory-centric test kernel was designed to determine conditions which allow the fastest possible memory transfer between global memory and GPU multiprocessors (Figure 6.1).This kernel's main function is to read and write data between global memory and GPU multiprocessors via varying thread-block sizes. The rate in which the kernel processes global memory can be measured and compared with the maximum effective memory bandwidth of the GPU. It also includes a simple calculation, x=x+1, inserted between the read and write operations. A comparison of the kernels speed can be made when including and then omitting the calculation. The kernel is listed in Figure 6.1.

The memory-centric kernel transfers data, back and forth between the GPUs global memory and its multiprocessors. The amount of data that transfers each time the kernel launches is set to the maximum amount of global memory (1.5 GB) available on the GPU. To measure the design's performance, the kernel should be called multiple times by the host in a loop, where elapsed times are then measured. By transferring the maximum memory available, the effect of host's overhead (to launch the kernel)can be rendered insignificant.

```
////////////////////////////////////////
//  Coalesced Memory Access Test Kernel //
////////////////////////////////////////
__global__ void
mem_speed_test_kernel_7( float* data_dev, int array_size )
{
    int dimBlocksPerGrid_x = gridDim.x; // # of blocks in x-dir of grid
    int blockID_x = blockIdx.x; // current block ID#
    int threadsPerBlock_x = blockDim.x; // block dimensions(threads per thread block)
    unsigned int tid_x = threadIdx.x;
    extern __device__ __shared__ float data_shrd[];

    // Set the number of floats accessed by each thread-block
    int floats_per_thread = 1;
    const int floatsPerBlock = threadsPerBlock_x*floats_per_thread;

    // Read 'threadsPerBlock_x' floats from global memroy
    //   and store into shared memory
    data_shrd[tid_x]     = data_dev[blockID_x*floatsPerBlock + tid_x ];
    __syncthreads();

    // Simple calculation
    data_shrd[tid_x] = data_shrd[tid_x] + 1;
    __syncthreads();


    // Write 'threadsPerBlock_x' floats from shared memory
    //   to global memory
    data_dev[blockID_x*floatsPerBlock + tid_x ] = data_shrd[tid_x];
    __syncthreads();

}
```

Figure 6.1: Listing of memory-centric test kernel.

6.3 Memory Coalesced Kernel Design

The kernel maps one thread to one floating-point value (4 bytes per thread) when accessing global memory. Thread-block dimensions and memory addressing schemes that meet coalescing conditions should result in much higher transfer rates than non-coalesced configurations [33-35].Figure 6.2 shows the impact of varying thread-block dimensions on the effective memory bandwidth (GB/sec). At thread-block sizes that both meet the coalescing condition (starting at 96 threads per thread-block) as well as achieving high occupancy, the measured rate of memory transfers reached well above the non-coalesced configurations. At one of the peak performing thread-dimensions (96 threads per thread-block) the kernel achieves 25.4 GBytes/sec, which is 66% of the maximum effective memory bandwidth. Whereas only 7.3% of the maximum effective memory bandwidth can be achieved by using dimensions that do not conform to coalescing conditions(i.e. 97 threads per thread-block). This is nearly an order of magnitude difference in performance between these two extremes. Therefore the impact of memory optimization is an advantage that cannot be ignored when designing kernels.

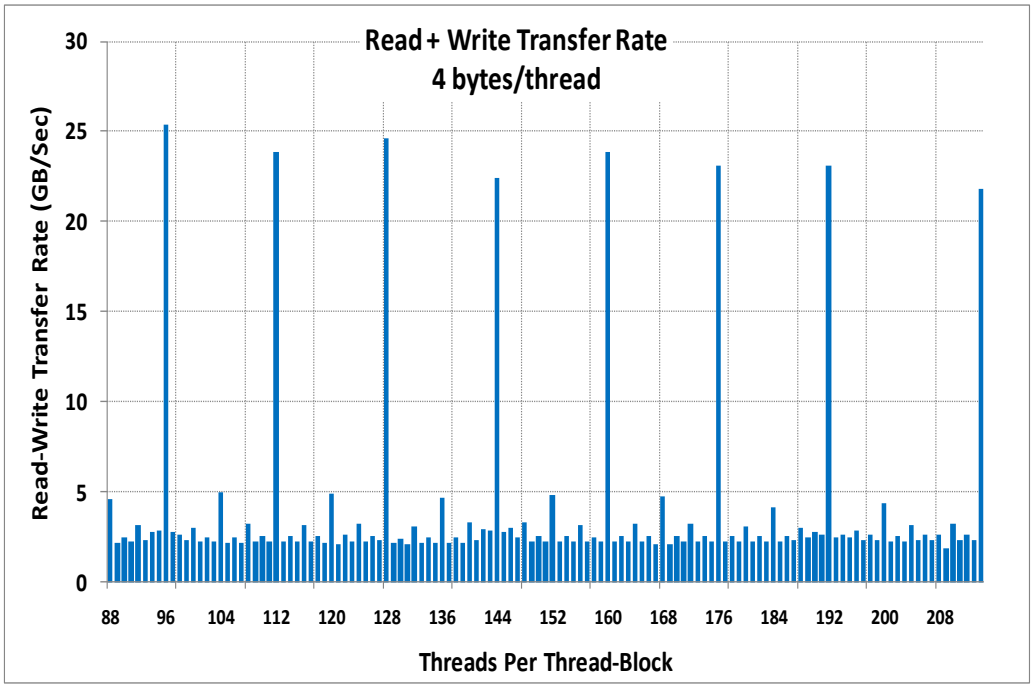


Figure 6.2: Read-Write transfer rates between global memory and the multiprocessors when varying thread-block dimensions.

The plot in Figure 6.2 also reveals that at certain thread dimensions (i.e. threads = 104, 120, 136...) minor jumps in speed occur, (5 GB/sec).Although these dimensions do not fully adhere to the coalescing conditions, partial coalescing can occur at intervals where some of the active warps happen to align with memory appropriately during kernel execution.

6.4 Varying Thread-to-Memory Ratio

Figure 6.3 shows results from three cases of thread-to-memory mapping scenarios: 4bytes/thread, 8 bytes/thread, and 16 bytes/thread. Only thread-block dimensions that adhere to the coalescing conditions are shown, where the non-coalesced thread-block

dimensions are omitted. The simple calculation within the kernel (x=x+1) is removed from the kernel code to measure memory transfer rates only.

The plot in Figure 6.3 shows that the best performing memory model is 8 bytes/thread followed by 4bytes/thread, and then by 16 bytes/thread. The peak performance of the best performing kernel (8bytes/thread) is 33 GB/sec which is 86% of the maximum effective memory bandwidth (38.4 GB/sec). The 4 bytes/thread model peaks at 28.4 GB/sec which is 73.9% of the theoretical maximum. Both peaks occur when the thread-block is defined at 96 threads.

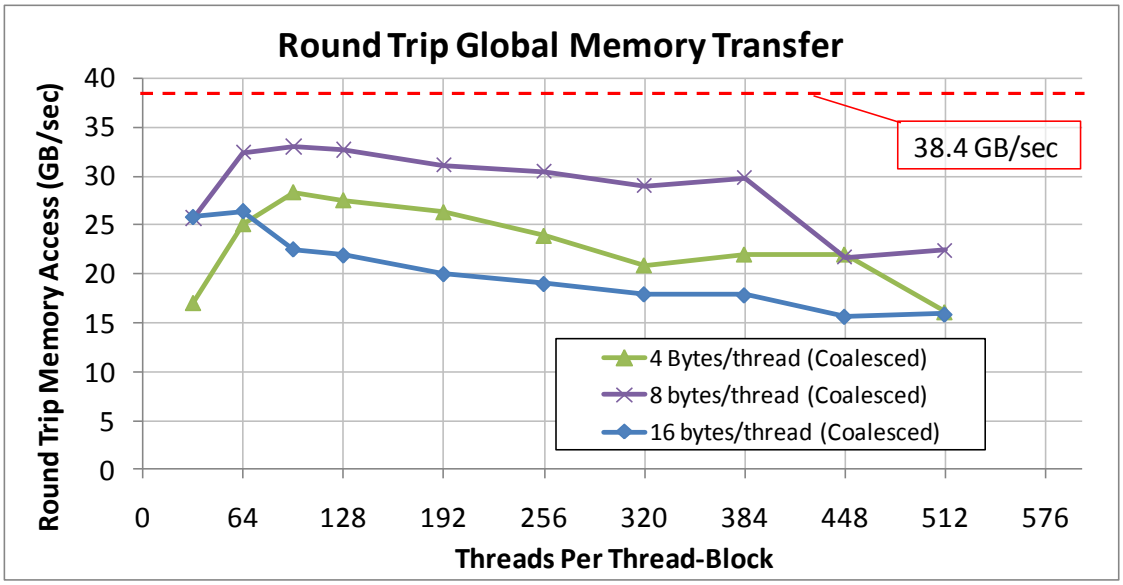


Figure 6.3: Read-Write transfer rates of a memory centric test kernel: 4 bytes/thread, 8bytes/thread, and 16 bytes/thread.

Figure 6.4 shows the occupancies for each thread-block dimensions tested in Figure 6.3. Not all of the thread-block dimensions shown in Figure 6.4 achieve 100% occupancy. In some cases the thread-block dimensions do not divide into the 768 thread-

slots evenly, which results in occupancies that are less than 100% (i.e. 32, 64, 320, 448, and 512). This results in a decrease in performance at these points, which correlates with the plot in Figure 6.3. It should be noted that this kernel uses very little shared memory and register resources, so that they did not influence occupancy. Once the behaviour of optimized memory access was assessed, the 3D-SCN algorithm was redesigned to adopt these memory-centric techniques, and to accelerate performance.

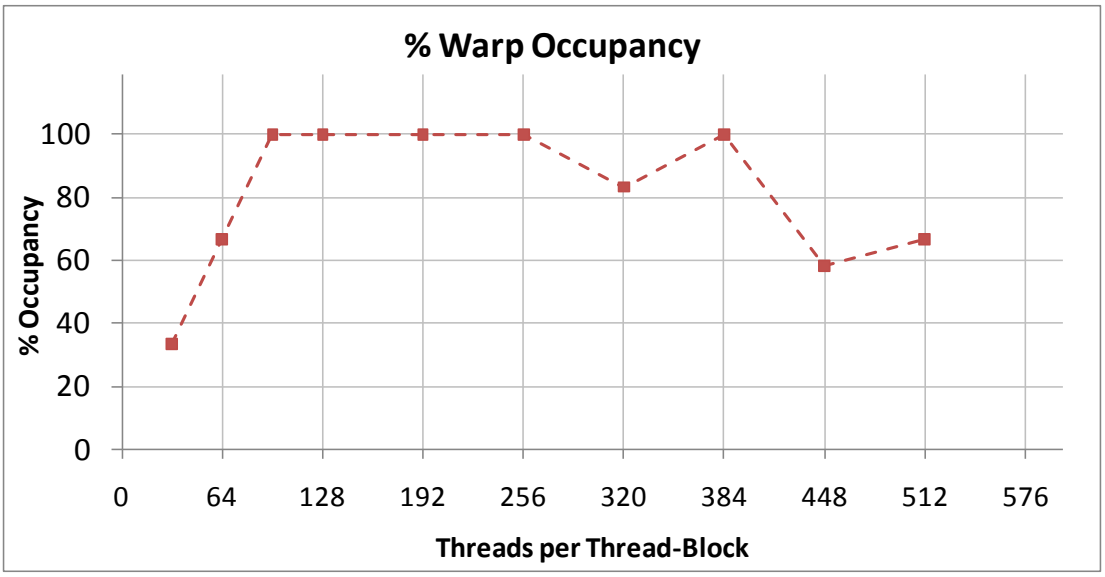


Figure 6.4: Occupancies of only thread-blocks that adhere to global memory coalescing. (8 bytes/thread)

Chapter 7 3D-SCN Scattering Kernel: Second Design

7.1 Memory Organization

The previous chapter explored kernel design techniques which optimized GPU memory access. This chapter covers the application of these techniques to create a new 3D-SCN kernel that is accelerated by a memory centric kernel design. The chapter following this reports that this revised design achieves an order of magnitude faster performance than the first 3D-SCN design.

In order to take advantage of memory coalescing for the second 3D-SCN kernel design, the arrangement of SCN link-line voltages stored in global memory must be different from the original design. Figure 7.1, illustrates the new memory plan for a mesh of 3D-SCN voltage values for one of the twelve link-line voltages for 3D-SCN nodes. The resolution of the memory plan for the *y*- and *z*- dimensions are each one node wide, but the *x*- dimension is partitioned into 64-node segments. The addressing is contiguous, first in the *x*-direction, then the *y*-direction and finally the *z*-direction. A thread-block is designed to be one dimensional, containing 64 threads. The kernel uses the thread-block to read 64 voltage values at a time in the *x*-direction.

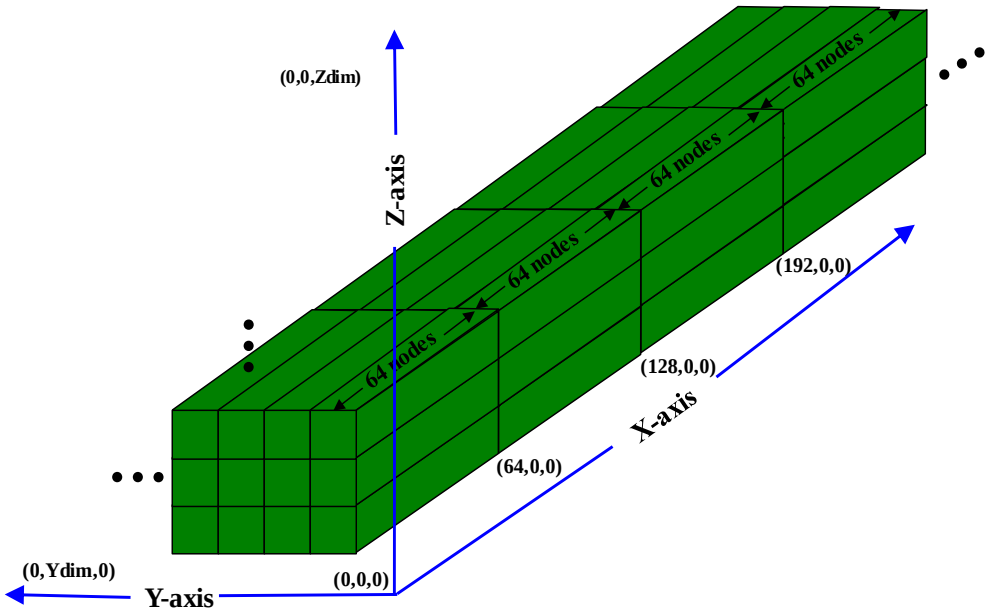


Figure 7.1:Global memory organization of link-line voltages of a 3D-SCN mesh of nodes.

The coordinate structure defined in Figure 7.1 stores values of just one of the 3D-SCN link-lines. Figure 7.2 illustrates how all 12 link-line voltages, for all nodes of a mesh, are stored in global memory. Each link-line voltage of the SCN node (V_1 to V_{12})is assigned to a contiguous region of global memory. During the copy stage of a kernel execution, the thread-block copies 64 values of $V_1$for a 64-node segment. The thread-block is then tasked to read 64 values of V_2, and so on, until all 12 link-line voltages are read for the target 64-node segment. The kernel then commands each thread of the thread-block to compute scattering for the respective nodes. Finally the scattering results are written back to global memory in the same way as they were copied.

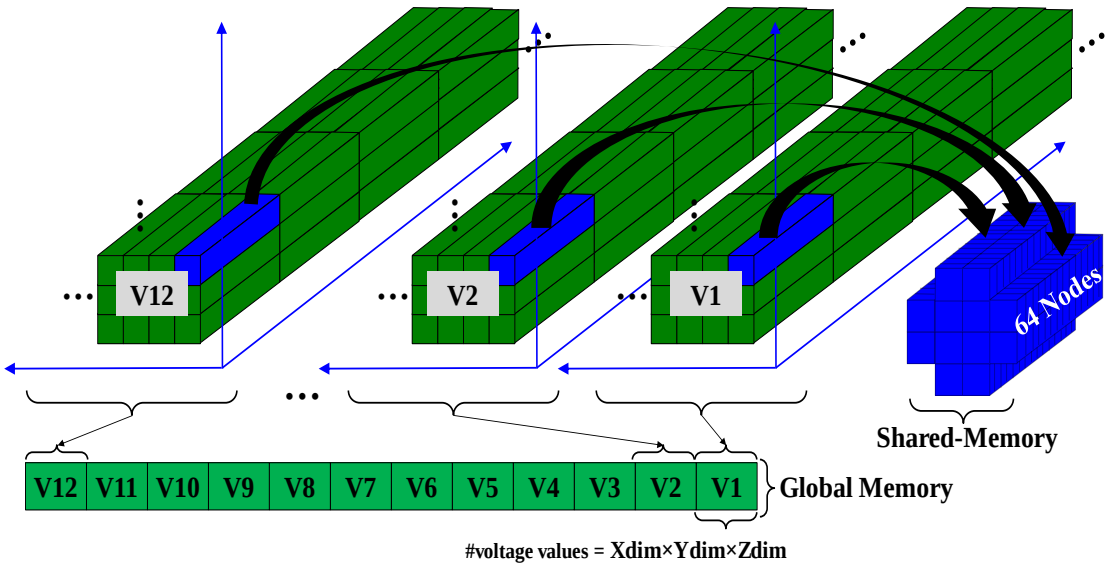
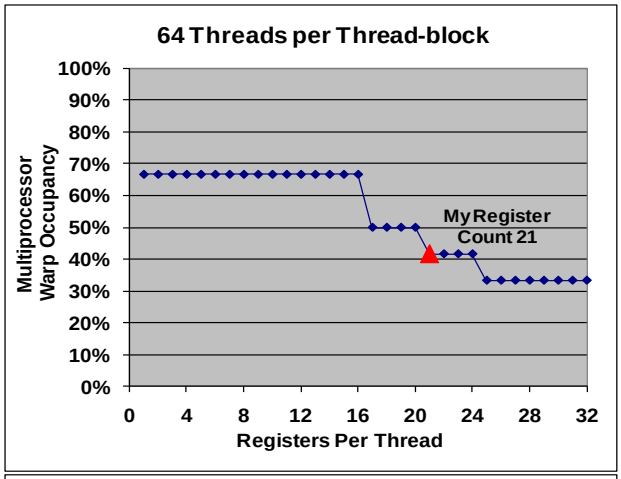


Figure 7.2: The mapping of 12 link-line voltages from global memory to a multiprocessor's shared memory space during a read stage of kernel execution.

The choice of using 64 threads per thread-block is to balance available multiprocessor resources with occupancy. A 64-thread thread-block can achieve 66.7% occupancy. The 3D-SCN kernel needs 21 registers per thread; with this resource usage the occupancy drops to 42% (Figure 7.3a).

a)



b)

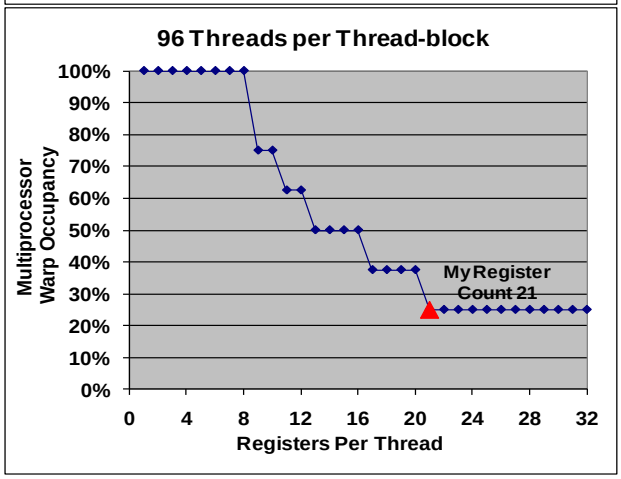


Figure 7.3: Occupancy sensitivity to varying register usage. a) 64 threads per thread-block achieving 42% occupancy when using 21 registers. b) 96 threads per thread-block achieving 25% occupancy when using 21 registers.

A 96-thread thread-block could also be considered because it can achieve 100% occupancy when the thread-block does not utilize more than 8 registers per thread (Figure 7.3b). Since the 3D-SCN kernel requires 21 registers, the occupancy would drop to 25% (Figure 7.3b). Instead, a 64-thread thread-block is the better alternative, achieving 42% occupancy, based on register usage.

7.2 Scattering and Impulse Interchange Kernels

The kernel design requires that the scattering and impulse-interchange stages be separated into individual kernels because including scattering and impulse-interchange operations in the same kernel is not possible with the current version of CUDA due to the lack of inter-thread-block synchronization capability. The two kernels would therefore be executed in sequence: scattering, then impulse-interchange each iteration (Figure 7.4b and c).

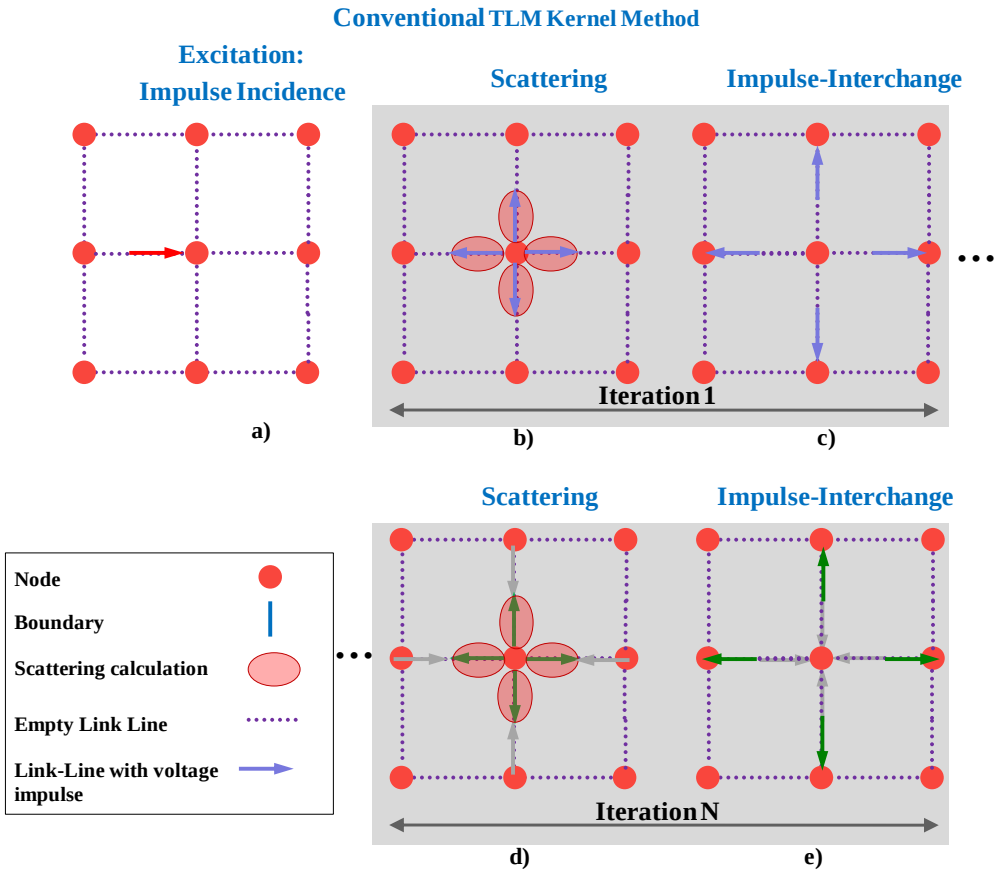


Figure 7.4:A 2D example of the operation of the Scattering kernel and Impulse-Interchange kernel.

Global memory transactions consume over 80% of the scattering kernel's execution time, where less than 20% of its time is spent on scattering calculation. The entire purpose of the impulse-interchange kernel is to move voltages around in global memory. This kernel spends nearly 100% of its execution time on global memory transactions. Clearly, the time spent by these kernels on memory transactions dominate each time-step iteration. Techniques that reduce the number of memory transactions in either of these kernels would help cut execution time.

7.3 Alternating Scattering Kernels

A new approach to executing 3D-SCN operations has been designed to reduce the number of memory transactions per iteration. The aforementioned scattering and impulse-interchange kernels are replaced with two slightly different scattering kernels: Scattering_Stage_1, and Scattering_Stage_2 (Figure 7.5b and c). The process of impulse-interchange is effectively absorbed into the design of these two new scattering kernels. Instead of executing the scattering and impulse-interchange kernels for each time-step, the new scattering kernels are executed in an alternating fashion, Scattering_Stage_1 in the odd time steps and Scattering_Stage_2 in the even time steps, over the course of the iteration process. This reduces the number of memory transactions by executing only one of these scattering kernels per iteration, instead of two kernels (scattering plus impulse-interchange).

The design of the first kernel, Scattering_Stage_1, uses the conventional 3D-SCN scattering kernel that is described in section 7.2 (Figure 7.5b). After the first scattering kernel completes its execution, all link-lines within all nodes of a mesh contain the intermediate results. Under the conventional implementation, an impulse-interchange procedure is used to exchange link-line voltages among neighbouring nodes.

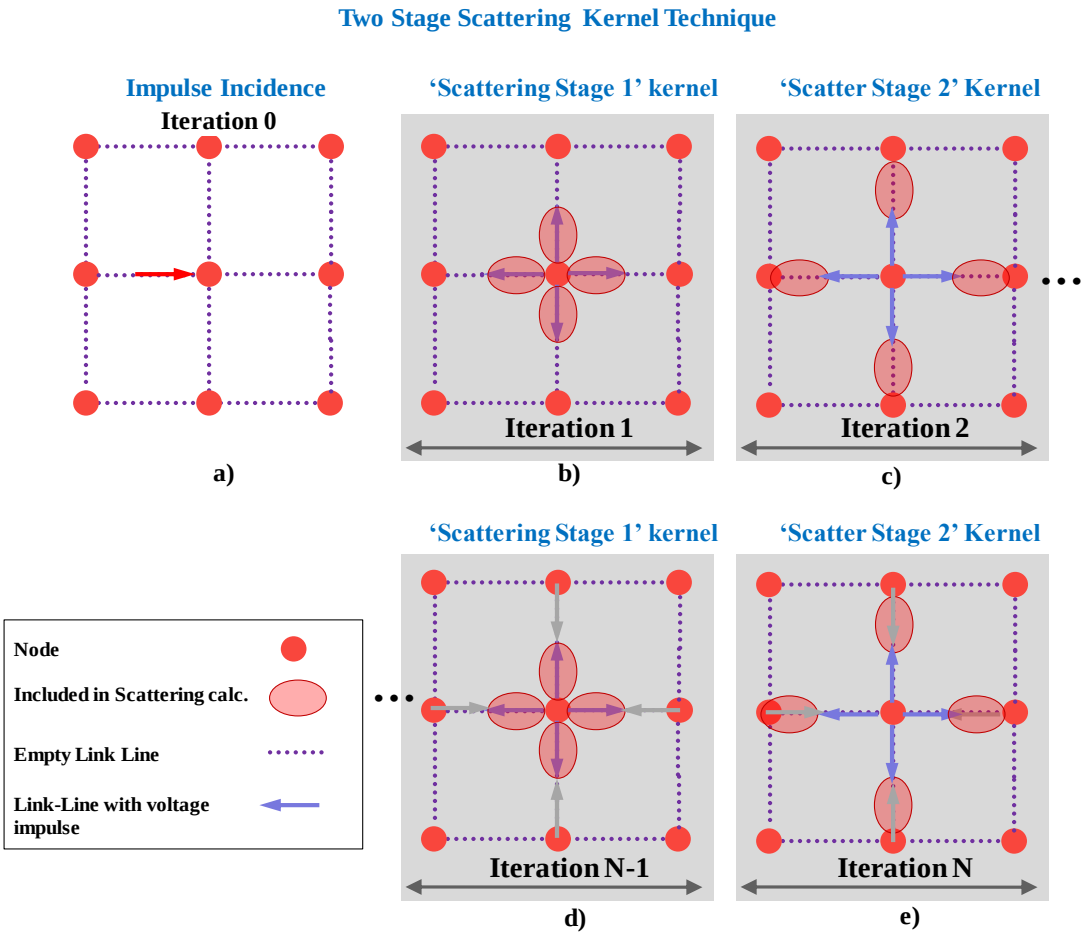
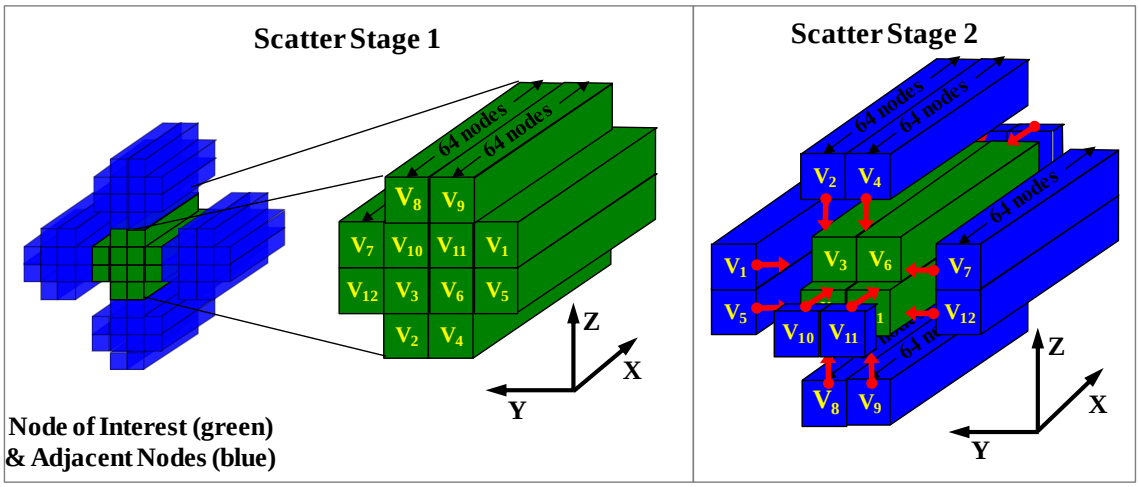


Figure 7.5: 2D Representation of the Alternating Scatter-Interchange Technique. a) An impulse is incident on the centre node. b) Scattering_Stage_1 executes a conventional scattering operation. c) Scattering_Stage_2 executes scattering in the link-lines of adjacent nodes. d) and e) Scattering_Stage_1 and Scattering_Stage_2 executed several iterations later.

After some scrutiny it is recognised that after the Scattering_Stage_1 kernel completes its scattering operations that the scattering results can remain where they are. During the next iteration, the scattering procedure at a node would need to access the voltage values in neighbouring nodes. The operation of the second scattering kernel, Scattering_Stage_2, is shown in (Figure 7.5c). The new kernel tasks each node to reach out into the voltage link-lines of adjacent nodes to read their voltage values, and then perform the scattering operation (Figure 7.5c) as if they just accessed their own link-lines. The scattering results are then written back to the same voltage link-lines of the neighbouring nodes from which they were originally read. The calls to Scattering_Stage_1 and Scattering_Stage_2 are invoked alternately every iteration. This technique absorbs the 3D-SCN impulse-interchange operation into a modified scattering procedure and therefore increases the overall execution speed of the GPU based 3D-SCN method.

7.4 3D-SCN Alternating Scattering Kernel Design

For illustrative convenience, a virtual memory mapping scheme for 3D-SCN nodes is shown in Figure 7.6. The link-line voltages that are accessed during Scattering_Stage_1 is shown in Figure 7.6a. All 12 link-line voltages in each 64-node segment are read into a multiprocessor's registers in a coalesced manner mainly because the starting address of each 64-node segment starts at an N×64 byte address. Once scattering completes, the results are written back to the same global memory regions, again in a coalesced manner.



a) **b)**

Figure 7.6: 3D-SCN memory mapping of a 64-node segment for, a) the Scattering_Stage_1 kernel which accesses link-line voltages within its node and, b) the Scattering_Stage_2 kernel which accesses link-line voltage values out into its neighbouring nodes.

The design of Scattering_Stage_2 differs from Scattering_Stage_1; the starting addresses of each of the 12 voltage segments in Scattering_Stage_2 are modified in order to read link-line voltages of adjacent nodes (Figure 7.6b). A consequence of altering the starting addresses is violating memory coalescing requirements.

7.4.1 Non-Coalesced Consequence of "Scatter Stage 2"

Unlike Scattering_Stage_1, where all accesses to link-line voltages in global memory are coalesced, the design of Scattering_Stage_2 cannot do the same. Of the 12 voltage segments that must be accessed, 8 can be read or written to in a coalesced manner. Memory accesses to the other four voltages segments, which are in the $\pm x$-directions, violate coalescing conditions (Figure 7.6b). These accesses violate coalescing because their starting addresses fall outside the N×64 index interval. Figure 7.7 illustrates this

non-coalesced circumstance where the starting address is skewed from a 64-byte interval. Each individual voltage of a node segment then requires 400-600 clock cycles to read. This is up to 12 times slower than a coalesce read of an entire node segment. The non-coalesced write transaction suffers the same inefficiency.

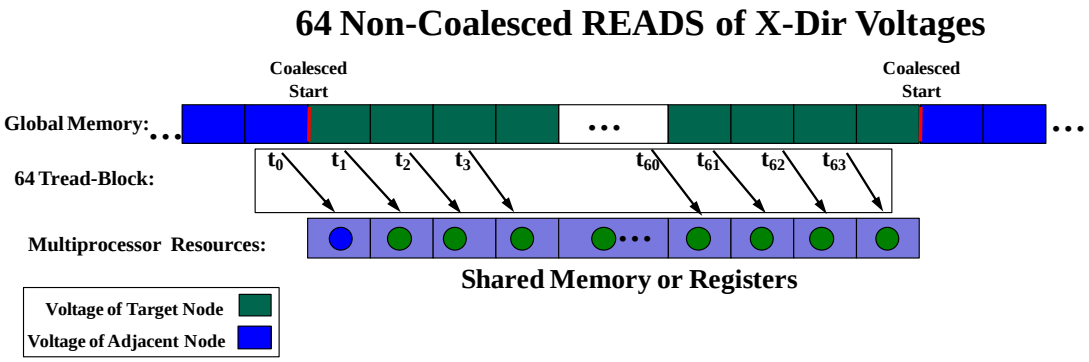


Figure 7.7: Illustration of non-coalesced reading of x-direction voltages within the Scattering_Stage_2 kernel.

7.4.2 Partial Coalesced Solution for Scattering_Stage_2

The design of the Scattering_Stage_2 kernel is improved by devising a method to induce partially coalesced reads and writes of the four ±*x*-direction voltages. This method accesses 63 of the 64 voltages of node segments in a coalesced manner (Figure 7.8a), where the 64th is read as non-coalesced (Figure 7.8b). The process of the coalesced reading of 63 voltages can be observed in Figure 7.8a. Threads t_0 to t_{63} read voltages from global memory, and then transfer them to shared memory, but shifted by one index. This process fulfills the kernel's objective where each node reads the link-line voltages of adjacent nodes for 63 of the 64 node voltages in the *x*-direction. Since t_0 starts at a coalesced interval, a coalesced read of 64 voltages are read within 400-600 clock cycles. The last voltage that is read (t_{63}) is not required, and is discarded.

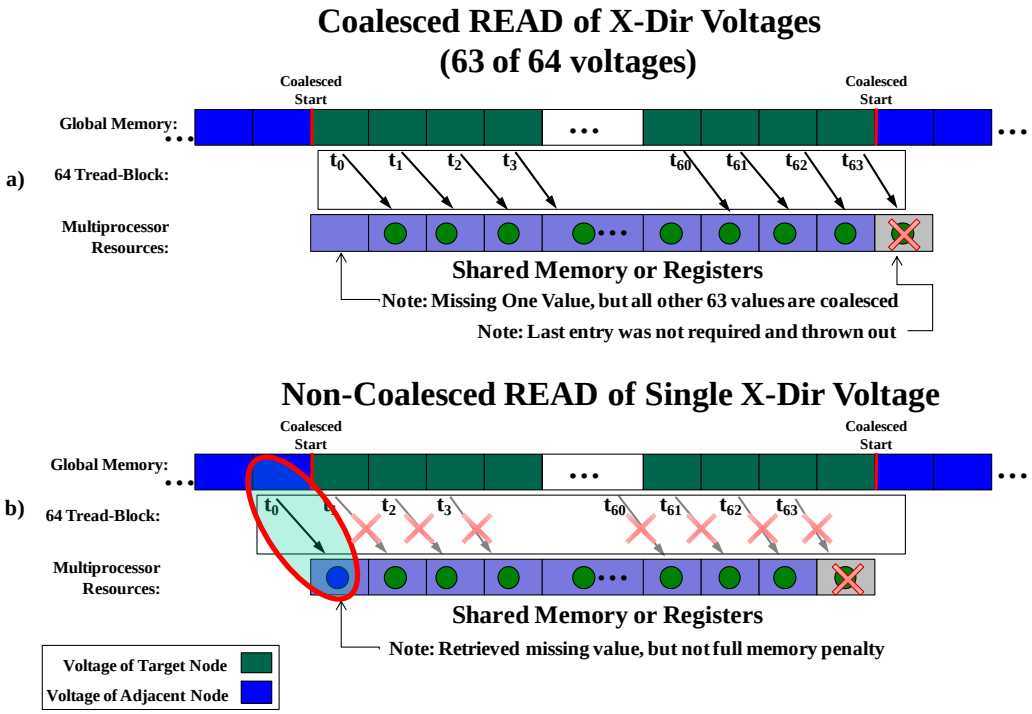


Figure 7.8: Reading of *x*-direction voltages in Scattering_Stage_2. a) 63 of 64 voltages are read in a coalesced manner, b) last of the 64 voltages is read in a non-coalesced manner.

After reading 63 of the 64 voltages described above, one more voltage needs to be read into the first position in shared memory (Figure 7.8b). This final voltage is read by invoking a thread-block read, but only allowing thread t_0 to execute the transfer. Although, this last value is read in a non-coalesced manner, the penalty is paid by only one thread and not all 64. Therefore the combination of both read transactions described in (Figure 7.8a and b) reduces the number of non-coalesced reads to only one.

The writing operation shown in Figure 7.9a and b follows a similar process as the read operation previously described. The only difference with this method can be seen in

thread t_{63} in Figure 7.9a. Thread t_{63} is prevented from writing a junk value from shared memory to global memory. Coalescing still occurs regardless if one or more threads of a thread-block are prevented from accessing global memory. Figure 7.9b shows a similar process as the read case where only one of the 64 threads write in a non-coalesced manner.

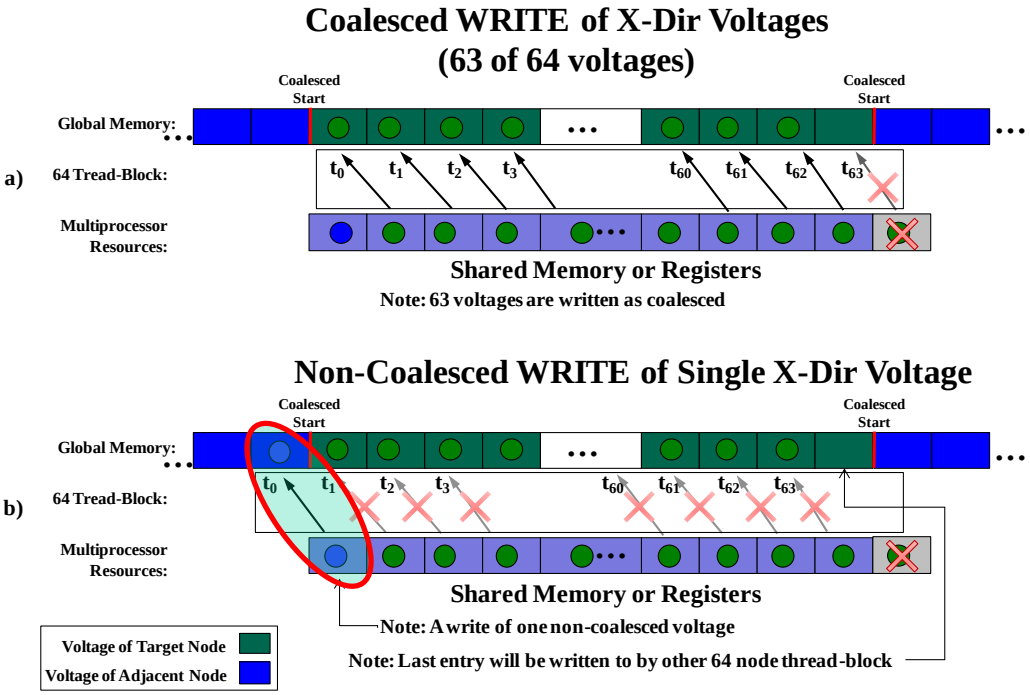


Figure 7.9: Writing of X-Direction voltages in Scattering_Stage_2. a) 63 of 64 voltages are written in a coalesced manner, b) last of the 64 voltages is written in a non-coalesced manner.

7.5 Pseudo-code Listing of Scattering Kernel

A pseudo-code listing of the Scattering_Stage_1 kernel is shown in Figure 7.10. When a thread-block launches, each of its 64 threads execute this kernel code. Segments of this code are highlighted to distinguish sections separated by synchronization commands.

Several optimization strategies are utilized by this kernel, which include: global memory pre-fetching, register conservation, and coalesced memory access.

The first section Figure 7.10a calculates the starting address of each voltage. Since 64 threads execute this kernel code, each thread calculates a different address for each voltage. The first thread (t_0) calculates the addresses that start at N×64 byte intervals, which adhere to coalescing conditions. The code segment in Figure 7.10b, copies 6 link-line voltages from global memory in anticipation of their use in the next code section(Figure 7.10c). The code segment in Figure 7.10c uses global memory pre-fetching to hide the computation of the two scattering calculations while at the same time reading two more link-line voltages (v6 and v10). These two voltages are read in anticipation of the next section. The pre-fetching strategy is accomplished by placing the read instructions just before the calculation instructions. This partially hides the time in which to access global memory where the computation of the calculations is done simultaneously(section 3.9.4). The code segment in Figure 7.10d also uses global memory pre-fetching, calculates the next two scattering computations, and writes the results back to global memory. The final code section (Figure 7.10e) calculates the rest of the scattering computations and writes the final results back to global memory.

A strategy of conserving registers is used in this kernel code by declaring the floating-point variables for voltages only when needed. The compiler can automatically detect if registers become out of scope between synchronization sections. It then reuses these registers in the following code segments.

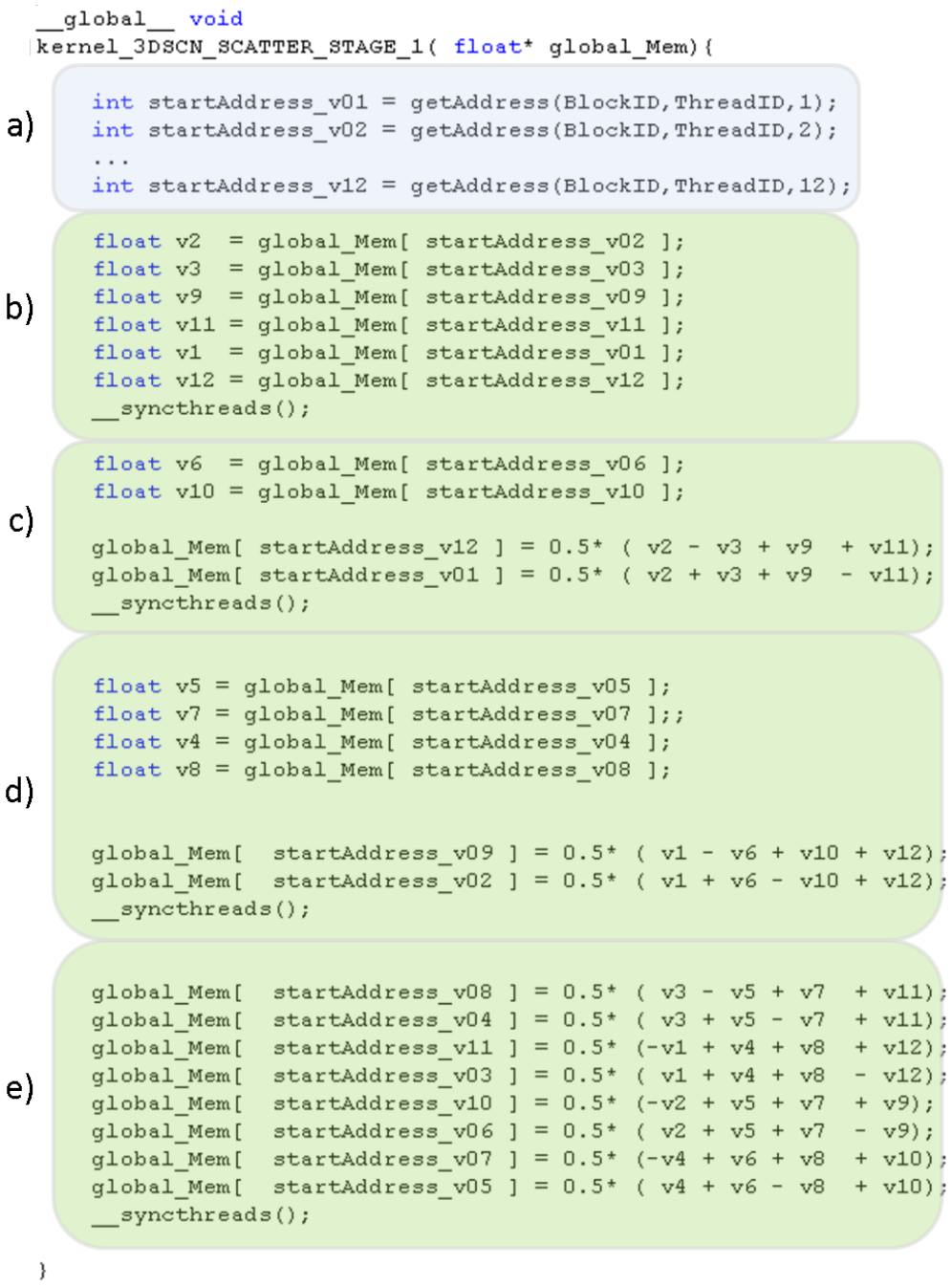


Figure 7.10Pseudo-code listing of the Scattering_Stage_1 kernel.(a) Starting addresses of each node is calculated. (b) Six voltages are read in anticipation of the first calculation. (c) Pre-fetching is used here by issuing global memory copy commands before the first set of scattering calculations. (d) Pre-fetching is also used here and more calculations are done. (e) The rest of the scattering calculations are completed.

7.6 Sampling Kernel

The process of sampling simulates the insertion of a probe into a structure, such as a filter, to measure voltages and sample at discrete time intervals. Analysis of the sampled voltage data is then conducted to calculate its spectrum. Multiple probes can be used to calculate the scattering parameters of a structure.

The sampling kernel design reads link-line voltages of a user defined node and stores the results into an array in global memory. The sampling kernel needs to be launched by the host after each TLM iteration where the array of samples would be transferred to the host after the total numbers of iterations are completed.

The alternating scattering kernel operation, described in section 7.3, causes a minor problem in the design of a sampling kernel. Since the sampling kernel must be executed after one of the scattering kernels, the sample kernel must monitor which scattering kernel is last executed. When sampling is to be done after Scattering_Stage_2 ,the sample kernel reads and stores the voltages of the target node; on the other hand, when sampling is executed after Scattering_Stage_1, the sample kernel reads the link-line voltages in its neighbouring nodes.

7.7 Boundary Kernels

The same boundary method used for conventional boundary operations can be used with the alternating scattering kernel design. The boundary kernel design accepts the coordinates of a rectangular boundary plane, as well as a reflection coefficient. It performs a boundary calculation wherever link-lines are adjacent to the defined boundary.

Figure 7.11 shows a 2D-TLM representation of how the boundary method operates. After either of the scattering kernels complete their execution, the boundary kernel multiplies the link-line voltages adjacent to boundaries by a reflection coefficient (Γ). It then swaps the link-line voltages on either side of the boundary (Figure 7.11c and e). The reason for swapping becomes evident when reviewing the operation of both the two scattering kernels (Figure 7.11b and d). The swapped link-line that contains the reflected voltage is positioned for the next scattering kernel to operate correctly.

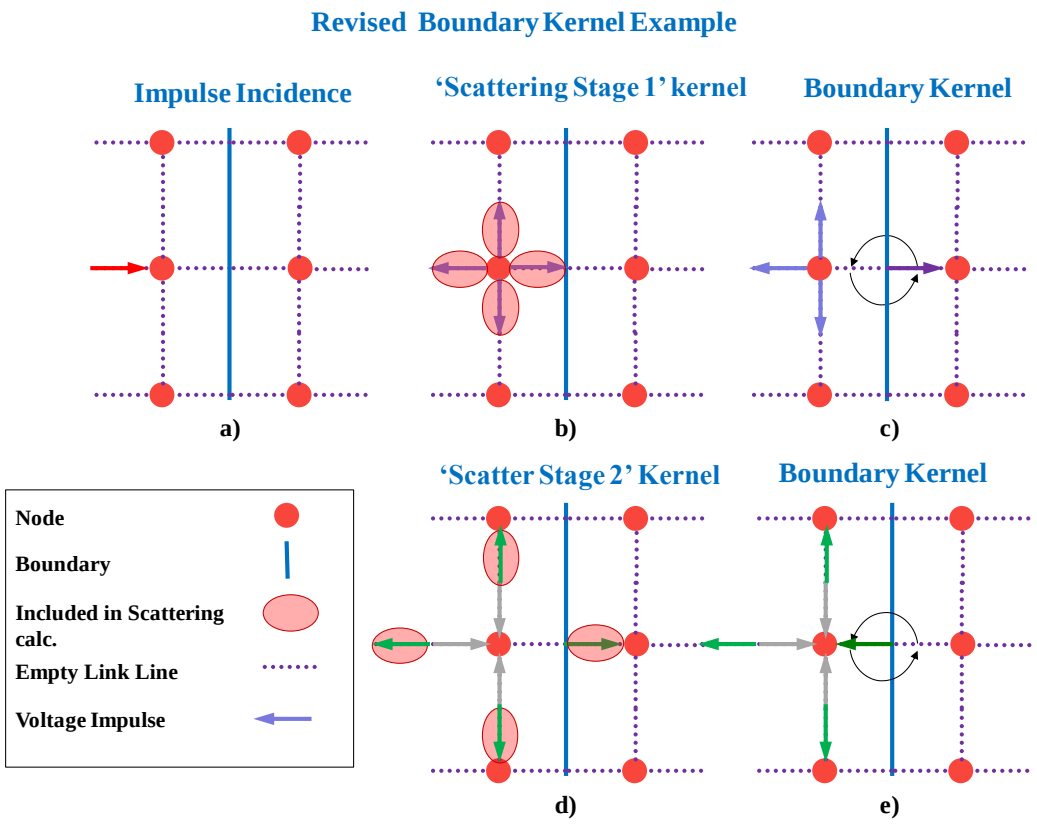


Figure 7.11: 2D TLM representation of two iterations which use the revised boundary kernel. a) An impulse is incident on a node, b) Scattering_Stage_1 executes, and c) the boundary kernel launches. The next iteration begins with d) Scattering_Stage_2, where voltages are in their correct position, and then e) the same boundary kernel is executed.

One disadvantage with this boundary method is that it is only able to handle simple plane structures; planes that can only be parallel with the *xy*-, *xz*-, or *yz*- planes. Conversely, the boundary embedding technique described in chapter 5 allows more complex structures to be defined. The boundary method described here is implemented in a separate kernel for keeping the scattering kernels' resource usage low at the expense of reducing structure complexity. A future version of the boundary kernel may embed boundary information into a global memory structure similar to that of one of the link-line voltages described in chapter 7.1. Compared to the current implementation, this method would consume more memory but improve the speed of execution.

7.7.1 Extra Node Layer Requirement at Mesh Extents

A consequence of using the alternating scattering kernels is that nodes must exist on either side of all boundaries. Many structures, such as waveguides and resonant cavities, require that the extents of the mesh be surrounded by boundaries. Since every boundary must have nodes on either side when using the new scattering kernels, then at least one extra layer of nodes must be defined that surround a structure. This translates into using additional memory, and extra computing of scattering for an added number of nodes.

7.8 Excitation Kernels

Three types of excitation kernels that are used for this design are: single point excitation, plane excitation, and half-sine plane excitation (Figure 7.12a-c). The single point excitation kernel requires a single node coordinate, as well as the direction of polarization of the impending impulses. Both the plane excitation (Figure 7.12b) and the half-sine plane excitation (Figure 7.12c) require the:

- specification of either the *xy*- plane, *xz*-plane or the *yz*-plane
- coordinates of the rectangular plane which defined the bounds of excitation
- direction of polarization of the impending voltages on the link lines

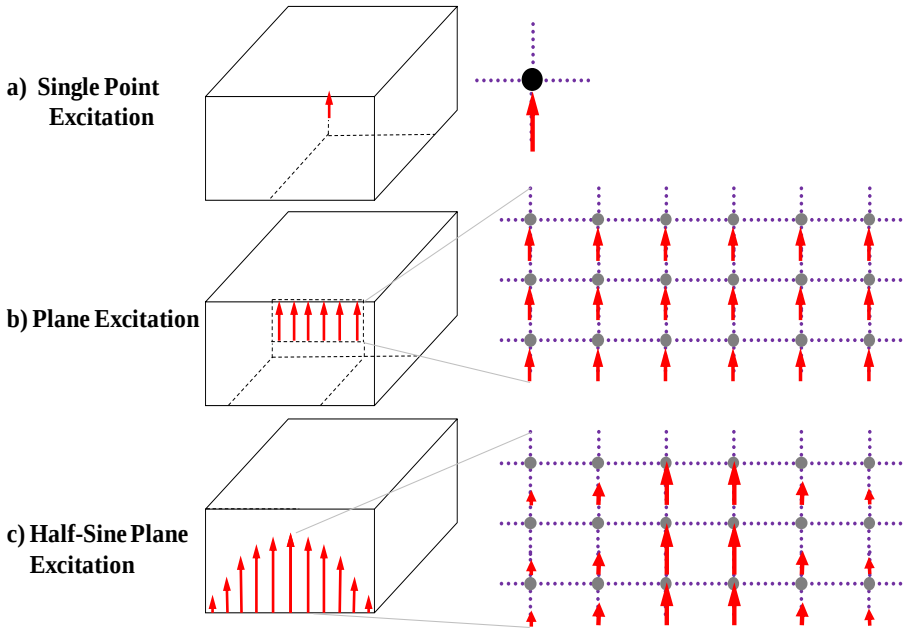


Figure 7.12: Three excitation kernels: a) single point excitation, b) plane excitation, and c) half-sine plane excitation.

Each of the aforementioned excitation types need to function with both of the scattering kernels. The defined node coordinates of an excitation structure are static. The excitation kernels identify which link-line to impose an excitation voltage depending on which of the two stages of scattering kernels was executed previously.

Figure 7.13 illustrates two iterations of a single point excitation for a two dimensional TLM mesh. A single link-line of a target node is shown with an imposed excitation voltage (Figure 7.13a). Scattering_Stage_1 occurs next where the results from scattering can be seen in the target node's link-lines (Figure 7.13b). The next iteration begins with the launch of the excitation kernel, (Figure 7.13c), which imposes an excitation voltage on the link-line of the adjacent node. This is done so that when Scattering_Stage_2

launches, the link-lines that are targeted for scattering includes the one that is intended for excitation (Figure 7.13d).

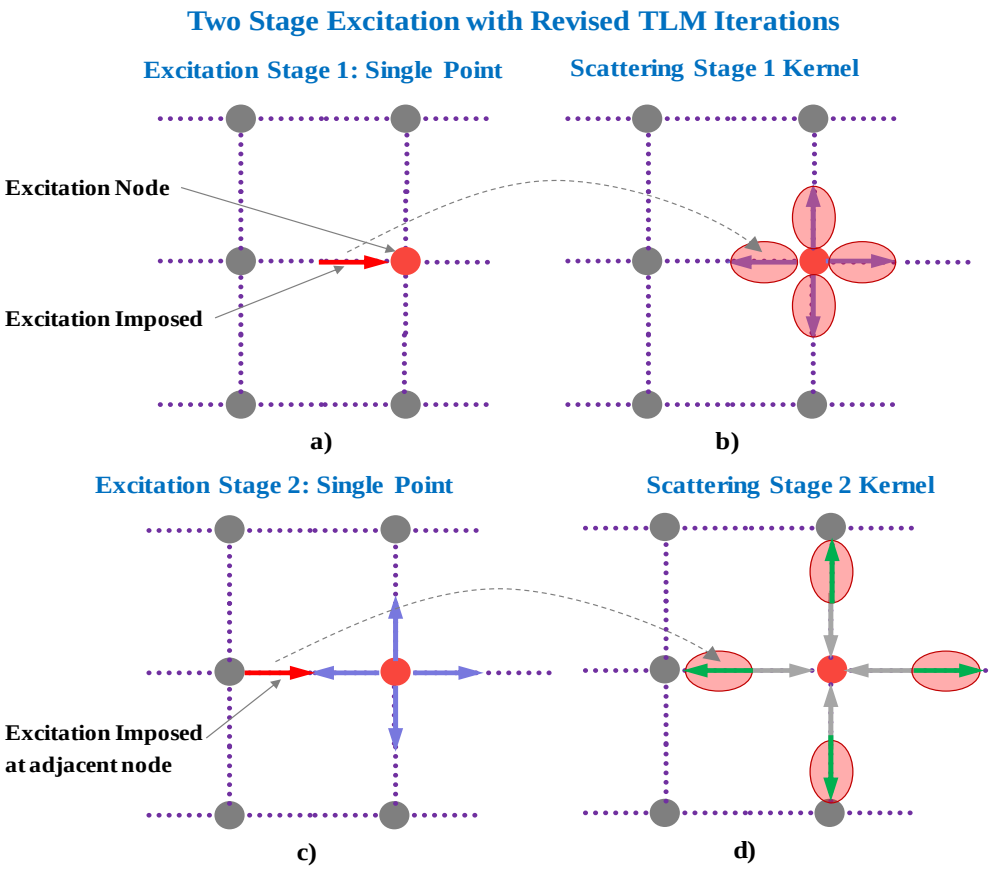


Figure 7.13:Stages of a single point excitation. a) Stage 1 excitation of single link-line at excitation node. b) Scattering_Stage_1 results of excitation (highlighted). c) Excitation for next iteration occurs at link-line of adjacent node. d) Scattering_Stage_2 results of excitation (highlighted).

Both the plane excitation and half-sine plane excitation kernels function similarly to that of the singe point excitation kernel, where alternate iterations requires excitation of alternating locations of link-lines.

7.9 Pseudo-Code Listing of 3D-SCN Host Program

Figure 7.14shows a pseudo-code listing of the 3D-SCN host program. This program flow is similar to that of the 2D-TLM listing in Figure 4.1. The program is coded in both CUDA and OpenCL; the two versions are nearly the same except for minor syntax differences. The main loop of the listing shows how the two scattering stages are executed alternatingly inside the iteration loop. The main loop contains calls to supporting kernels such as sampling, excitation and boundary execution.

```
// Pseudocode of typical 3D-SCN Host program
Main (){
    instantiate_GPU_object(...);
    set_Mesh_Dimensions(x,y,z);
    numIter = N; // set the number of iterations

    // Setup of Exciation
    set_Excitation_Structure(point or plane or halfSinePlane);
    set_Excitation_Waveform(GaussianSine, impulse...);

    // Setup of Samplint
    set_Sampling_Probe(coordinates...);

    // Create Boundaries
    set_Boundary(coordinates...);
    set_Boundary(coordinates...);
    //...

    Start_Timer(...)
    for (idx = 0 ; idx < numIter ; numIter++){
        execute_Excitation(...);

        if(idx%2 = 0){ // If iteration is even
            execute_Scattering_Stage_1(...);
        }
        else{ // if iteration is odd
            execute_Scattering_Stage_2(...);
        }

        execute_Boundary_Kernel(...);

        execute_Sampling_Kernel(...);
    }
    Stop_Timer(...)
    Calculate_Speed(...);// in Meganodes/sec

    extract_Mesh();
    extract_Samples();
}
```

Figure 7.14: Pseudo-code listing of 3D-SCN host program.

7.10 Validation with Waveguide Band-pass Filters

Similar to the process of validating the previous GPU-TLM designs, a WR-28 band-pass filter (Figure 7.15a) is used to demonstrate the correctness of this second design of the 3D-SCN GPU kernels. Since CUDA and OpenCL kernel codes are very similar, only the results obtained with our CUDA code are reported here. The results are compared with that obtained with MEFiSTo-3D pro. The S-parameters obtained with the two programs are shown in Figure 7.15b and c. Also shown in the figure are the differences between the GPU and MEFiSTo traces, where the differences are found to be negligible. The validation results for two additional rectangular waveguide band-pass filters are shown in Figure 7.16 and Figure 7.17. These tests confirmed the GPU-TLM code has been correctly implemented to perform 3D-SCN calculations.

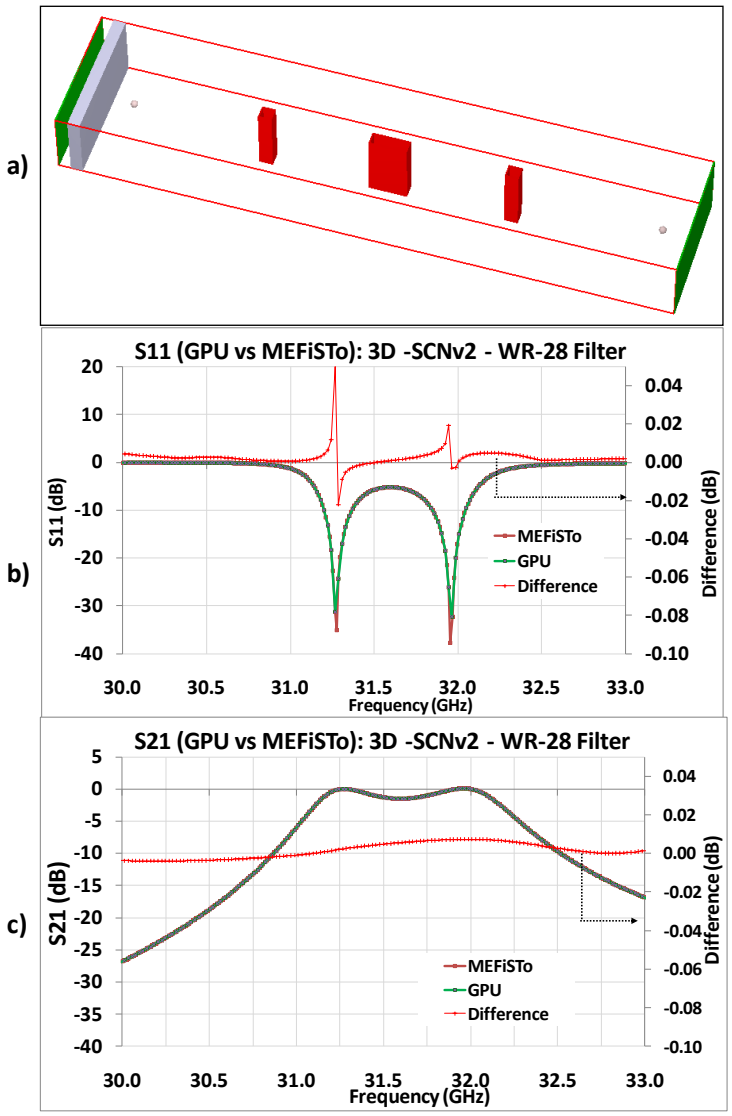


Figure 7.15Comparison of the S-parameters of a WR-28 waveguide band-pass filter obtained with MEFiSTo and with the second version of the 3D-SCN GPU kernel:(a) WR-28 Band-pass filter. (b) Return loss (S11) and (c) Insertion loss (S21).

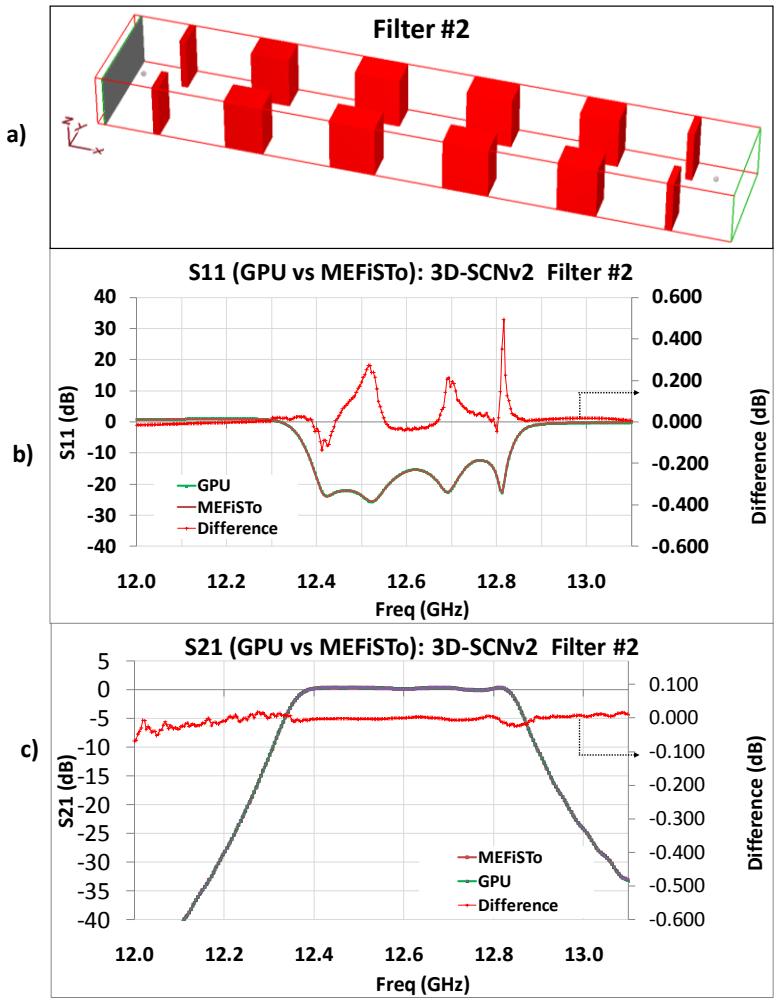


Figure 7.16 Comparison of the S-parameters of a waveguide band-pass filter obtained with MEFiSTo and with the second version of the 3D-SCN GPU kernel: (a) An iris coupled rectangular waveguide band-pass filter. (b) Return Loss. (c) Insertion Loss.

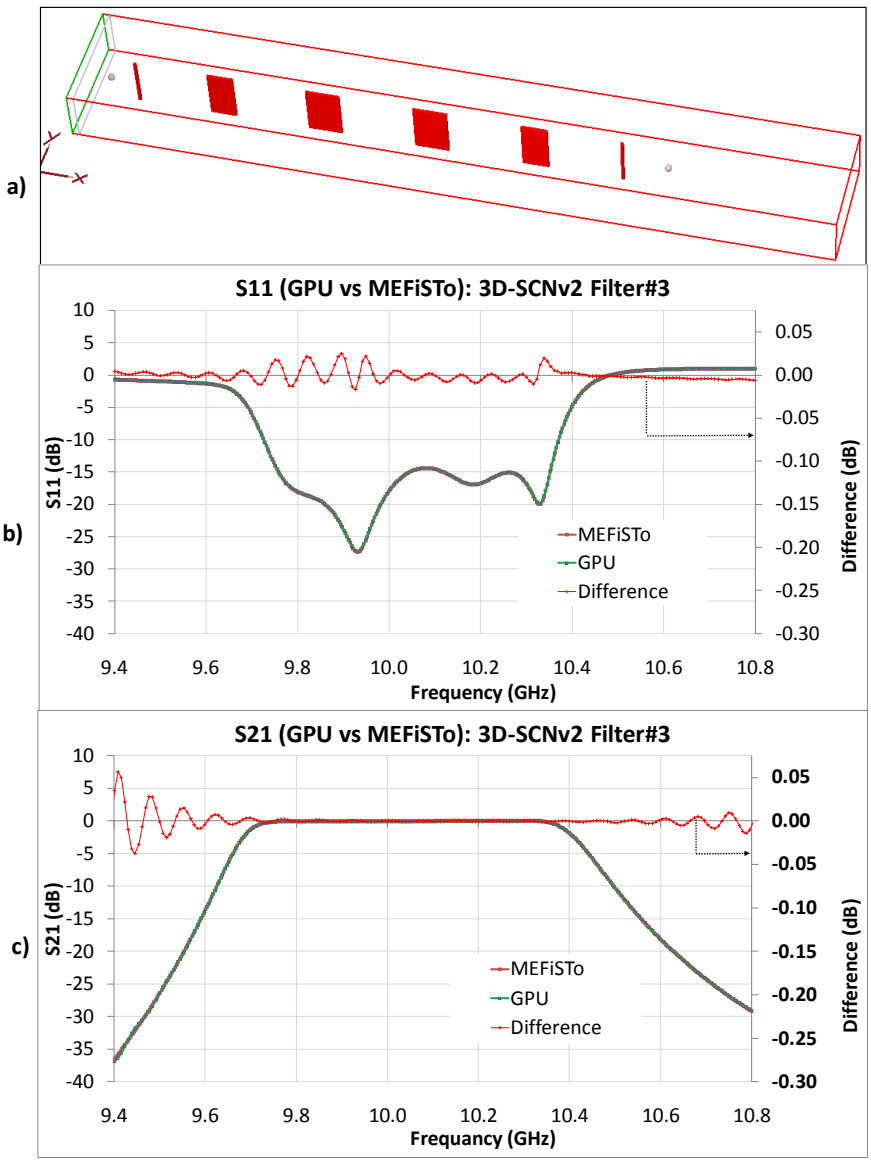


Figure 7.17: Comparison of the S-parameters of waveguide band-pass filter obtained with MEFiSTo and with the second version of the 3D-SCN GPU kernel: (a) An inductive post coupled band-pass filter. (b) Return Loss. (c) Insertion Loss.

7.11 Validation with a Resonant Cavity

The above comparisons indicate that the largest differences between the GPU results and that of MEFiSTo's occur at the resonant frequencies (where the S11curves have the dips) of the filters. In order to further confirm the correctness of the GPU-TLM implementation, the resonant frequency of a cavity is computed with the two programs, namely GPU-TLM and MEFiSTo. The dimension of the cavity is 64mm × 64mm × 64mm, Figure 7.18. The resonant frequencies due to a wide band impulse excitation can be computed with the following formulas [28]:

$$f_{mnl} = \frac{c}{2\pi\sqrt{\mu_r \varepsilon_r}} k_{mnl} \tag{7.1}$$

where

$$k_{mnl} = \sqrt{\left(\frac{m\pi}{a}\right)^2 + \left(\frac{n\pi}{b}\right)^2 + \left(\frac{l\pi}{d}\right)^2} \tag{7.2}$$

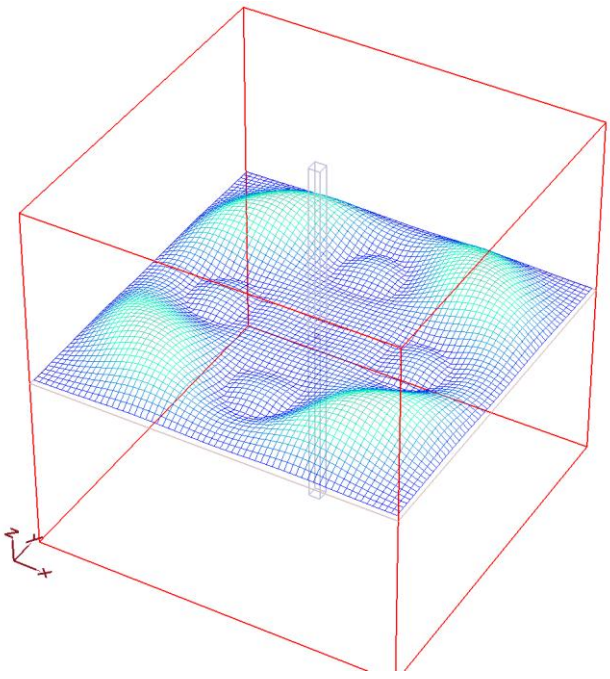


Figure 7.18: A resonating cavity (64mm × 64mm × 64mm) implemented in both the GPU 3D-SCN kernels as well as MEfiSTo-3D Pro to compare frequency responses to wide band excitation.

The spatial resolution of the simulated structure is 1mm in both MEFiSTo-3D Pro and the GPU-TLM programs. According to equations 2.13 the minimum guided wavelength should be less than 10 mm. In order for the guided wavelength to be smaller than this value, The first few resonant frequencies in that cavity have guided wavelengths much larger than 10 mm. Figure 7.19 shows the first four resonant frequencies of the cavity which are the 110, 310, 330, 510 modes. The theoretical values (blue crosses) are also plotted for comparison.

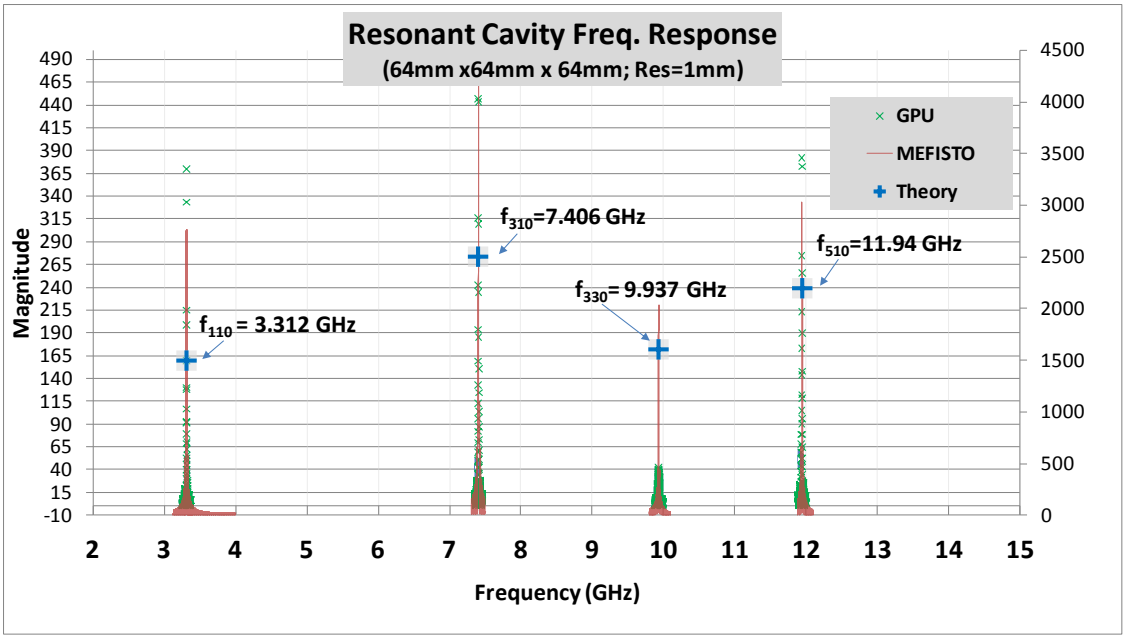


Figure 7.19: Comparison of the resonant frequencies of a rectangular cavity obtained with MEFiSTo and with the second version of the 3D-SCN GPU kernels. The dimension of the cavity is shown in Figure 7.17.

Figure 7.20shows four zoomed-in views of the frequencies shown in Figure 7.19. The differences from theoretical values, in percent error, are shown for both MEFiSTo and GPU models in Figure 7.21.

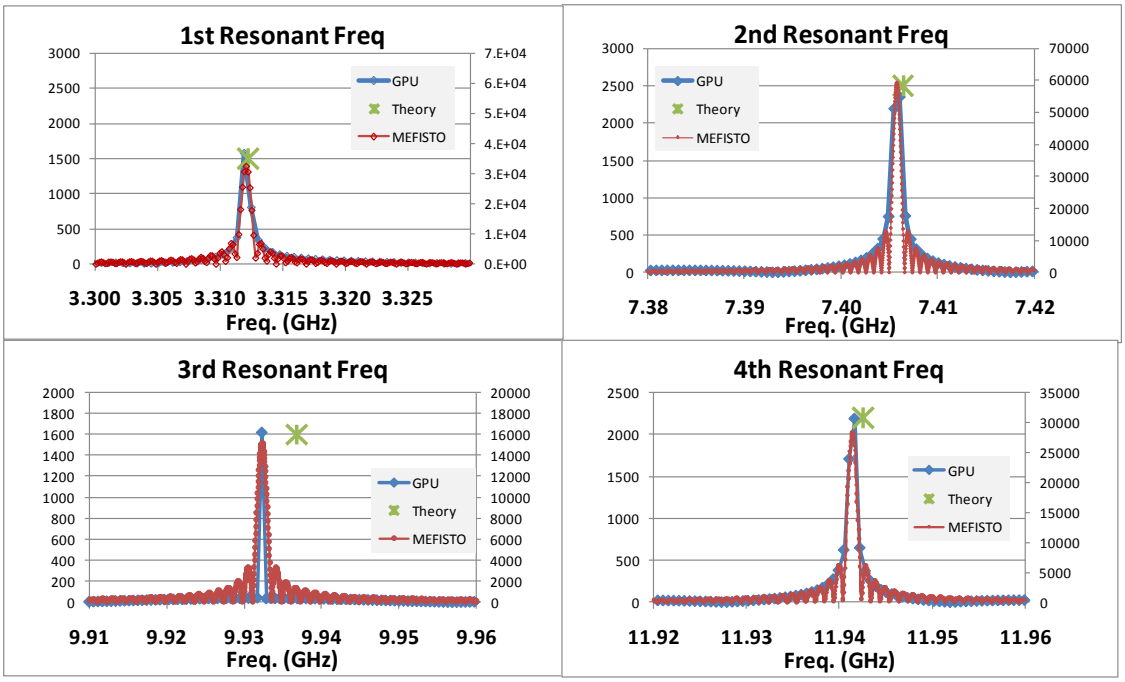


Figure 7.20: Close-up views of the 4 resonant frequencies in Figure 7.18

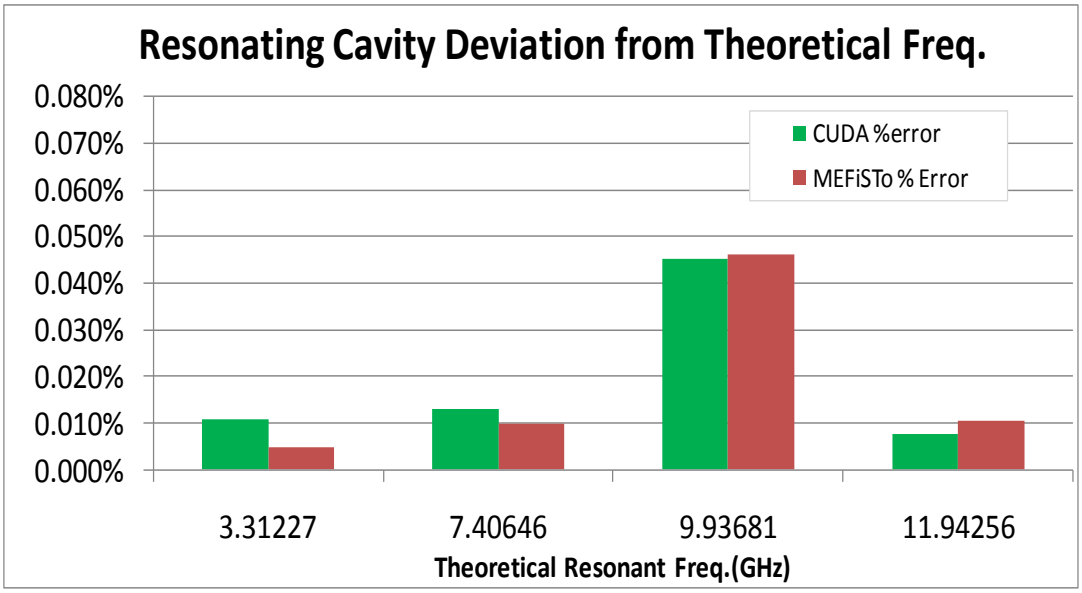


Figure 7.21: GPU and MEFiSTo deviation from theoretical resonant frequencies

The results depicted in Figure 7.21indicate that the errors of the four computed resonant frequencies are within 0.05% of the theoretical values. This ensures that the GPU-TLM program is indeed correctly implemented. Figure 7.21 also shows that the deviation from theoretically predicted frequencies for both the GPU-TLM and MEFiSTo are comparable to each other (<0.006%).

Having validated the computation of the second version of the 3D-SCN kernels, the next step is to measure their performance in speed. The next chapter reports the speed of execution of the 3D-SCN kernels implemented in both CUDA and OpenCL. These execution speeds are compared with the speed of MEFiSTo-3D Pro.

Chapter 8 3D-SCN Performance Results

The purpose of this chapter is to report the computational speed of the second version of the 3D-SCN kernels which are the core modules of the new GPU based 3D-SCN TLM applications. The performance of both the CUDA and OpenCL versions of the program are compared with MEFiSTo-3D Pro, as well as the first 3D-SCN kernel reported in Chapter 5. Three key performance indicators used extensively in this chapter to measure kernel speeds are: node-rate, node-data-rate, and speed-up.

N iterations of the main iteration loop (illustrated as pseudo-code in Figure 7.14) are timed and used to calculate speed. Instead of timing just the execution of the kernels (or just one iteration), timing the application iterating N steps such that more than one minute passes is considered a reasonable measure of overall performance. This is especially true when making comparisons with MEFiSTo-3D Pro. Furthermore, the node-rate and node-data-rate are taken at increasingly larger mesh sizes so that the overhead of using OpenCL and CUDA for small 3D-SCN mesh can be investigated.

8.1 Scattering Kernels' Performance

To measure the performance of the second version of the 3D-SCN scattering kernels (Scattering_Stage_1 and Scattering_Stage_2), the CUDA and OpenCL 3D-SCN applications were run without engaging boundaries, excitation, or sampling. A cubic

(N×N×N) empty mesh structure was used, where N was varied to sweep the structure's size to as large a mesh that was able to fit into the GPU's memory (1.5 GB).

The plot in Figure 8.1 contains the node-rate results of the first 3D-SCN program described in chapter 5, as well as the new 3D-SCN CUDA and OpenCL applications. The new 3D-SCN CUDA application achieves a peak performance of 570 MNodes/sec, the OpenCL version achieves 450 MNodes/sec, and the first 3D-SCN project peaks at 47 MNodes/sec. The overall performance of the CUDA version is better than that of OpenCL, where OpenCL requires considerable overhead compared to CUDA. The impact of this overhead is clearly demonstrated by this side-by-side comparison (a 21% difference in speed). The speed-up of the second version of 3D-SCN kernels compared to the first version is 12.1 times. Clearly, the revised kernel design yields more than an order of magnitude in incremental improvement over its predecessor.

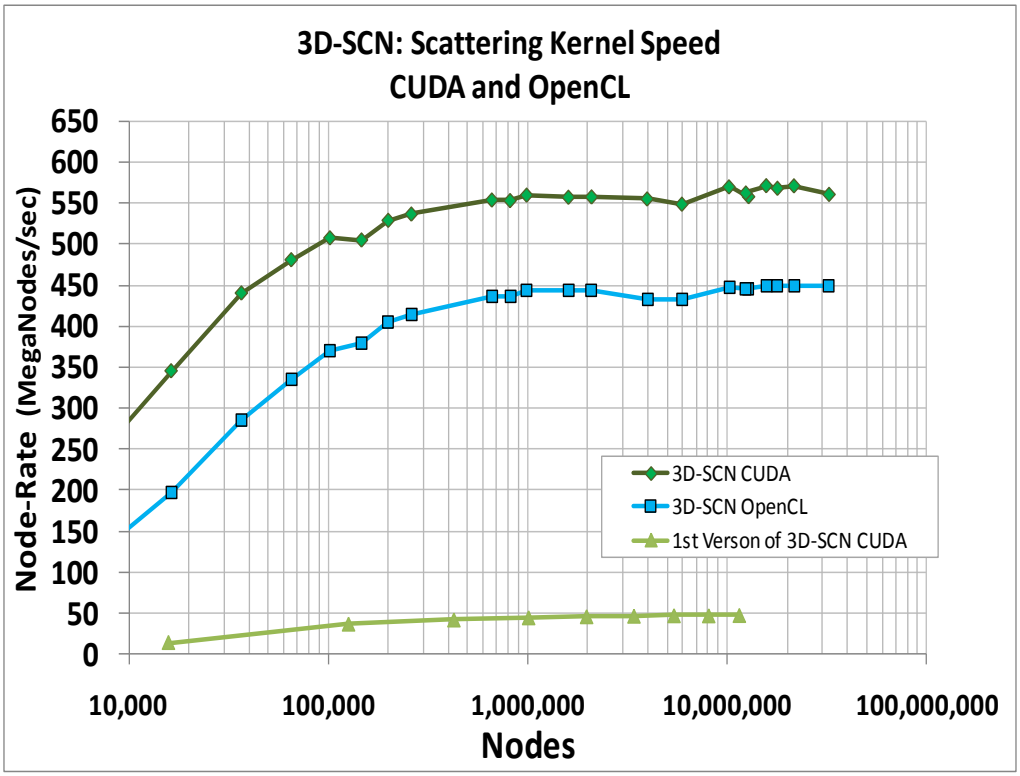


Figure 8.1: Comparison of speeds of the CUDA and OpenCL based 3D-SCN scattering kernels. The mesh contains no boundary.

The speed-up of the 3D-SCN OpenCL application over the first 3D-SCN CUDA program is 9.5 times faster. Since only a CUDA version of the first 3D-SCN kernel design was completed (prior to OpenCL being released) an OpenCL-to-OpenCL speed-up comparison between the two designs could not be done. Despite this, the comparison results clearly demonstrate the new design, whether it is implemented in CUDA or OpenCL, has an advantage over the non-optimized CUDA implementation.

8.2 Memory Model Comparisons

8.2.1 CUDA Memory Model Performance

The performance of the 3D-SCN CUDA application executing empty cubic structures is expressed as node-data-rates in the plots of Figure 8.2(green). The peak performance of the CUDA application is 27.4 GB/sec. This is 71% of the GPU's maximum effective memory transfer rate (38.4 GB/sec, determined in section 5.7).

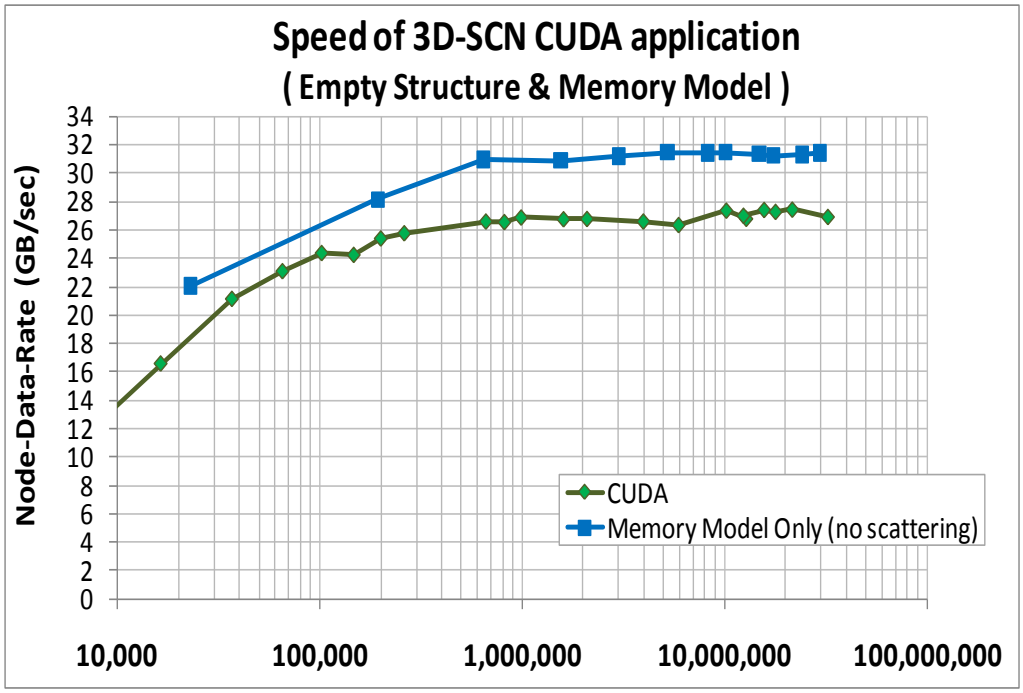


Figure 8.2: Speed of 3D-SCN CUDA application executing on empty structures, compared to 'memory model' transfer speeds.

To determine the maximum data transfer rate of the second version of the 3D-SCN kernels (described in chapters 6 and 7), both Scattering_Stage_1 and Scattering_Stage_2 were stripped of their SCN scattering calculations. The code that remained handled

memory transfer only. This rendered the operation of the kernel down to transferring data back and forth between global memory and the multiprocessors. The data transfer rate of these 'memory model' kernels is significant as it represents the upper bound speed of any kernels with SCN memory transfer operations. The plot in Figure 8.2 (blue) shows data rate of the memory model kernels with a peak speed of 32.7 GB/sec. At this speed the memory optimized kernel design achieves 85% of the maximum effective memory transfer rate (38.4 GB/s). This is a good indication that the memory optimization techniques applied to the scattering kernels fulfil their intended purpose.

With the scattering routines included in the kernels, the speed reduced to 84% of the memory model's upper bound. Achieving 84% represents a good indication that the scattering calculation does not overly hinder overall performance, and that further optimization would yield only marginal improvements.

8.2.2 OpenCL Memory Model Performance

The plots in Figure 8.3 shows the node-data-rates of the 3D-SCN OpenCL application processing the same empty mesh structures described previously. It also shows the data transfer rates when the kernels were stripped down to data transfer operations only (as was done in the previous section). Comparing the performance of the OpenCL and the CUDA applications, the memory model of both (Figure 8.2 and Figure 8.3) perform very much the same. The node-data-rate of the empty meshes in the OpenCL version drops 34% from the memory model's upper bound. Whereas the CUDA version drops only

16%. Since the memory model code of both CUDA and OpenCL versions are the same, the difference in performance is likely due to the inherent overhead in OpenCL.

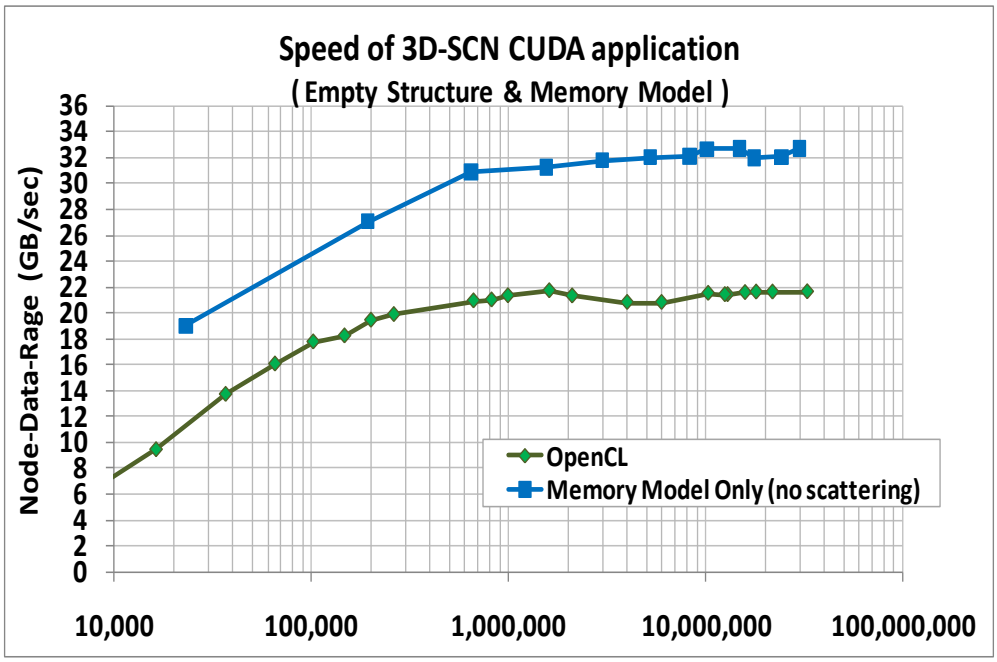


Figure 8.3: OpenCL: memory transfer speed of memory model and scattering procedure

8.3 Speed of Filter Simulations: OpenCL and CUDA

The WR-28 band-pass filter used previously (in Chapter 7) to validate the correctness of the GPU based 3D-SCN applications are employed again in this chapter to measure the speed of the CUDA and OpenCL 3D-SCN applications. To model the filter, the boundary kernels, excitation kernels, and sampling kernels must all be turned on. The resolution of the filter was varied thereby increasing the node density, and increasing the number of nodes required. The node-rates and node-data-rates were then measured at these various

structure densities. The maximum number of nodes that were able to fit into the available GPU memory (1.5 GB) was 30,000,000 nodes.

Figure 8.4 shows node-rates of the CUDA and OpenCL programs in modeling the filter. The lower the node count, the slower the speed. This was due to determine the influence of the host's overhead per iteration. Just as was found in the previous section, the overall performance of the CUDA version of the program was better than that of the OpenCL version. The CUDA version peaked at 530 MNodes/sec, while the OpenCL version peaked 421 MNodes/sec; a 20% difference.

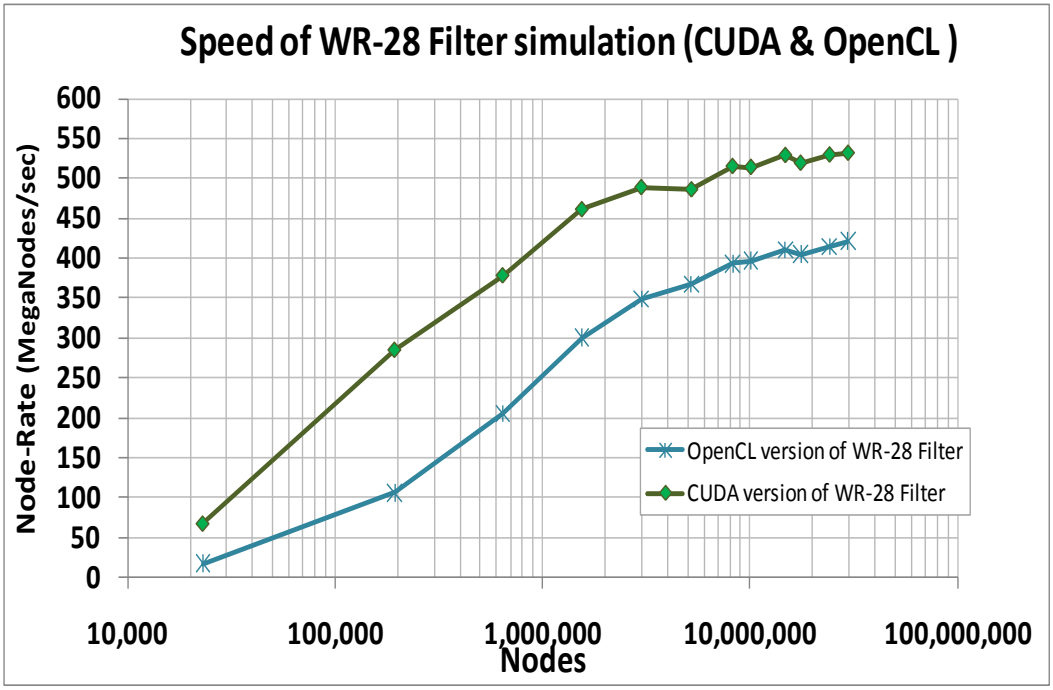


Figure 8.4: Speed of the 3D-SCN CUDA application and the 3D-SCN OpenCL application when modeling the WR-28 filter.

8.3.1 Impact of Boundary, Excitation, and Sampling Kernels: CUDA

The speed of the filter simulated in the CUDA application was also expressed as node-data-rate (Figure 8.5). The plot shows the impact on speed when excitation, sampling, and boundary kernels were incrementally omitted. This was done in order to gain insight into the influence each kernel had on overall speed. The trace of the memory model (blue) is also included to show the upper bound of these kernels. As can be seen, the impact of the sampling and excitation kernels is minimal. However, the inclusion of boundaries reduces the speed significantly.

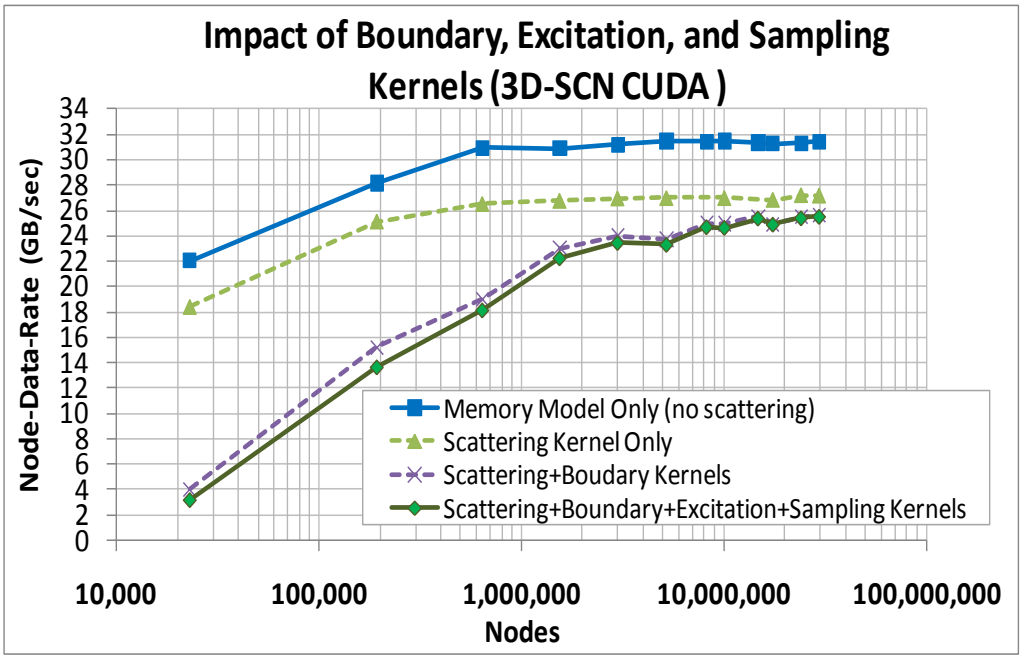


Figure 8.5: Impact of the boundary, excitation, and sampling kernels on the speed of the new CUDA based 3D-SCN application when modeling a WR-28 filter.

Since the performance of the boundary kernel influences the overall speed of the CUDA 3D-SCN application considerably more than the sampling and excitation kernels,

a boundary loading measurement plot was generated (Figure 8.6). Starting with a completely empty mesh structure with no boundaries (configured to a maximum size allowed on the GPU) the speed of the CUDA 3D-SCN application was measured and set as the maximum speed achievable for this plot (100% on the vertical axis). Boundary occupation (the horizontal axis) is defined as a percentage of nodes within the structure that are adjacent to boundaries. As the number of boundaries increases, the boundary occupation increases and the speed of execution decreases. When 9% of the structure is occupied by boundaries, the performance of the algorithm drops to 90% of its maximum. The performance of the algorithm drops fairly steeply where a structure with a 33% boundary occupancy results in a speed that is 51% of the maximum.

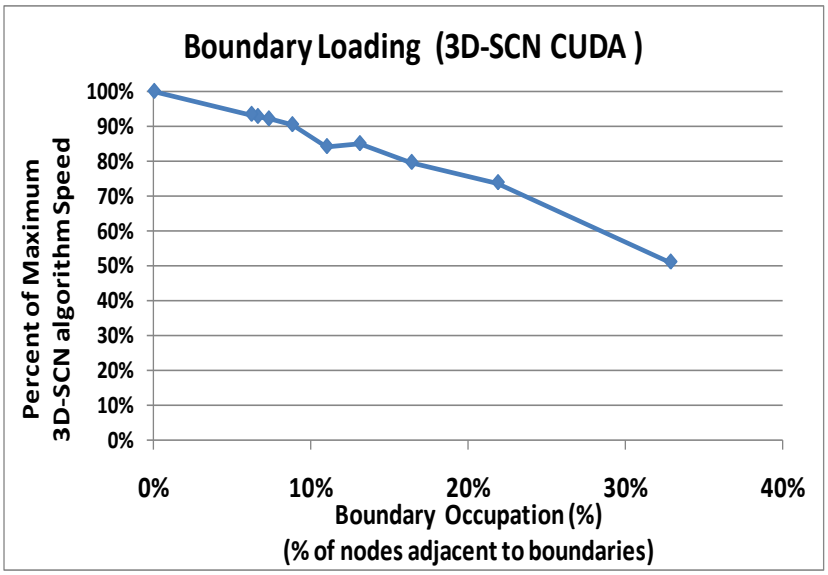


Figure 8.6: Impact on CUDA 3D-SCN algorithm speed by loading a mesh structure with boundaries. As the percentage of the structure occupied by boundaries (i.e. nodes adjacent to boundaries) increases, the 3D-SCN algorithm speed decreases from its boundary free execution speed.

8.3.2 Impact of Boundary, Excitation, and Sampling Kernels: OpenCL

Similar data was measured for the OpenCL 3D-SCN application (Figure 8.7). The speed of execution for the OpenCL version is, overall, lower than that of the CUDA version. It is interesting to note that the boundary kernel causes more of a performance penalty on the OpenCL based 3D-SCN application than its CUDA counterpart.

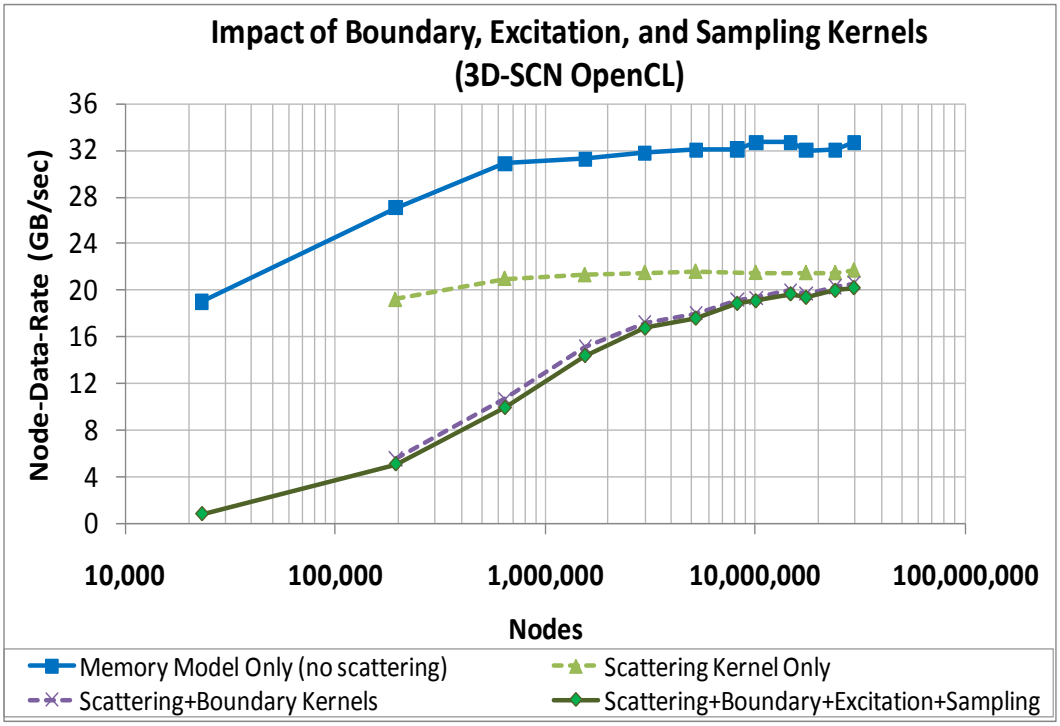


Figure 8.7: Impact of the boundary, excitation, and sampling kernels on the speed of an OpenCL 3D-SCN application that models a WR-28 waveguide band-pass filter.

The plot in Figure 8.8 shows how the performance of the OpenCL program drops as the number of boundaries increases. At 22% of the mesh structure occupied by boundaries, the performance of the OpenCL program drops to 53% of its maximum

speed. The rate of declining performance in the OpenCL version is much steeper than its CUDA counterpart.

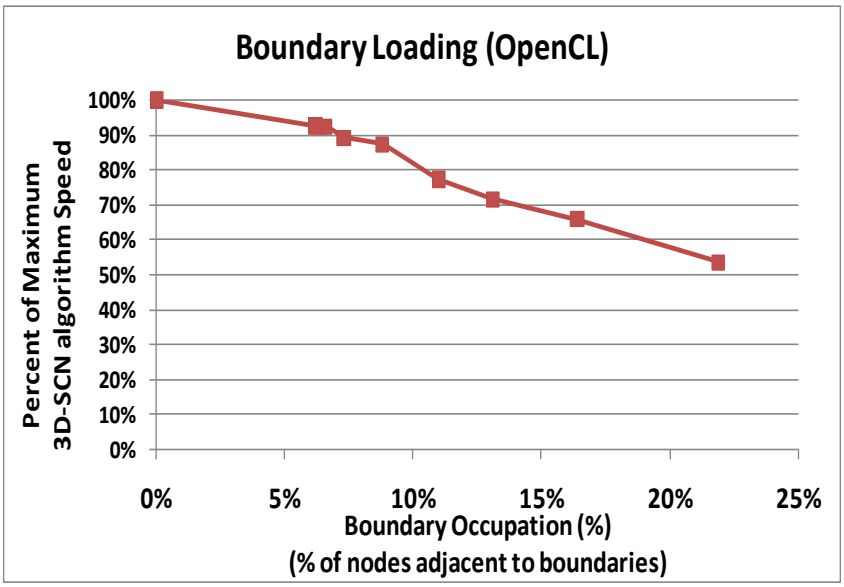


Figure 8.8:Impact on OpenCL 3D-SCN algorithm speed by loading a mesh structure with boundaries.

8.4 Comparing Against MEFiSTo

The WR-28 band-pass filter was modeled using MEFiSTo-3D Pro in order to compare the performance of the GPU based programs with that of a CPU based implementation. MEFiSTo modeled the filter using 1 to 4 CPU cores in the HP-xw9400 workstation, the host machine that controls the GPU hardware.

8.4.1 MEFiSTo Speed Measurements

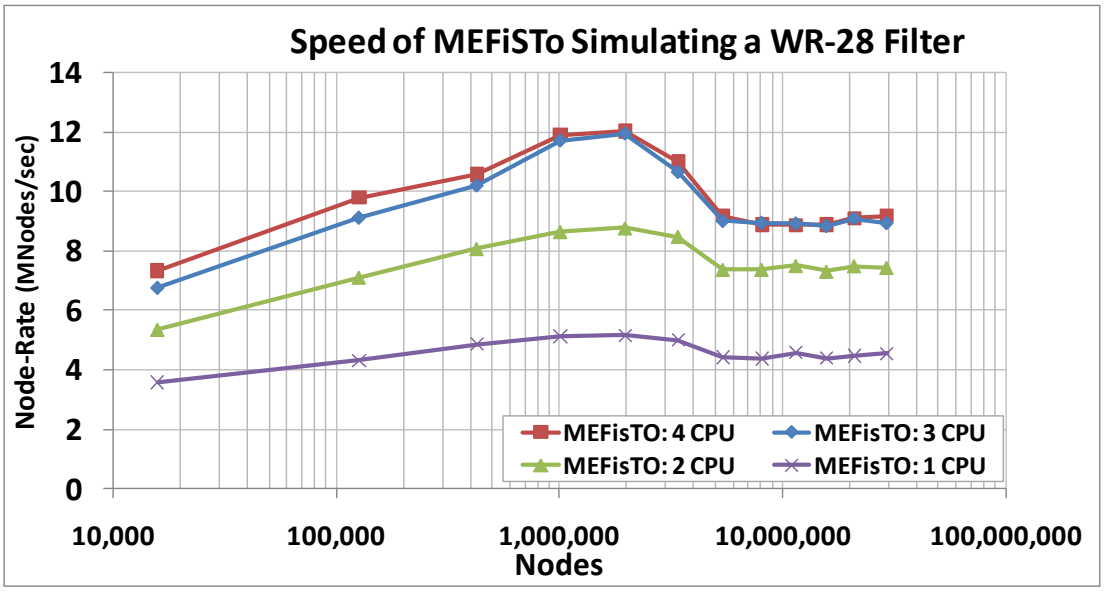


Figure 8.9: Performance of MEFiSTo when modeling a WR-28 band-pass filter with 1,2,3 and 4 CPU cores.

The plots in Figure 8.9 show the performance of MEFiSTo-3D Pro when engaged to model the filter with 1 to 4 CPU cores. A peak performance of 12 MNodes/sec is achieved when four cores are utilized. There are two characteristics of these plots worth noting. Firstly, the program performance is negatively impacted by the increase in mesh size when the number of nodes is between 2 to 6 million nodes. Beyond 6 million nodes the performance plateaus.

The other noted characteristic is that the difference in performance between 3 and 4 CPUs is minor. The HP workstation may be considered a Uniform Memory Access (UMA) multiprocessor [40], where memory is shared by all CPU cores. With this architecture a bottleneck condition could appear in the memory bus when all four cores

compete to access the RAM. Hence adding more threads to match the number of CPU cores on a CPU based program would increase performance very little if memory access becomes the bottleneck of the algorithm.

8.4.2 Aggregate Performance Evaluations

The node-rate results of the CUDA, OpenCL and MEFiSTo versions of the 3D-SCN programs, when modeling the WR-28 filter, are plotted in Figure 8.10. The vertical axis of this plot is in a log scale in order to clearly show all traces due to the significant difference between the node-rates of the MEFiSTo traces and any of the GPU traces.

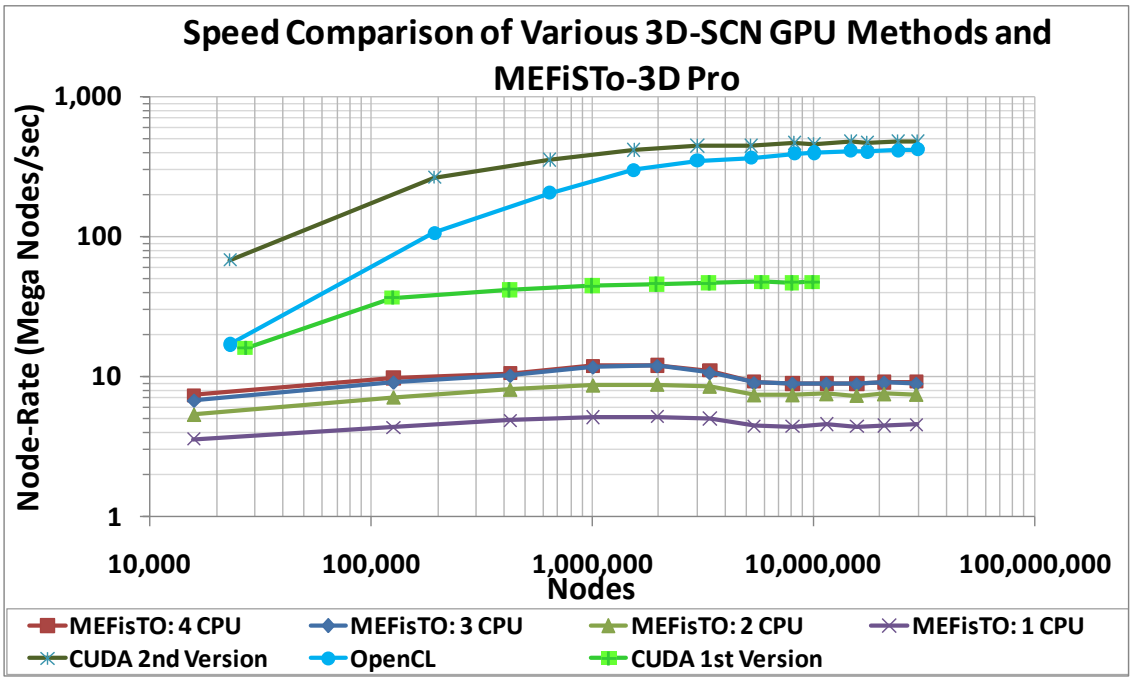


Figure 8.10: Comparison of the speeds of CUDA 3D-SCNand MEFiSTo. The programs were used to model a WR-28 band-pass filter with increasing mesh size.

Figure 8.11 show the speed-up comparisons between the performance of first version of the 3D-SCN algorithm (Chapter 5) and MEFiSTo (1-4 CPUs). The speed-up at higher node counts are 12 times compared to MEFiSTo running on 1 CPU core. Twelve times speed-up is a noteworthy benchmark, but since it is likely that all four cores would be used by MEFiSTo on this workstation and workstations of similar calibre, then a typical case of using 4 CPU cores is more appropriate. Compared to four CPU cores, the GPU version achieves a 5 times Speed-Up.

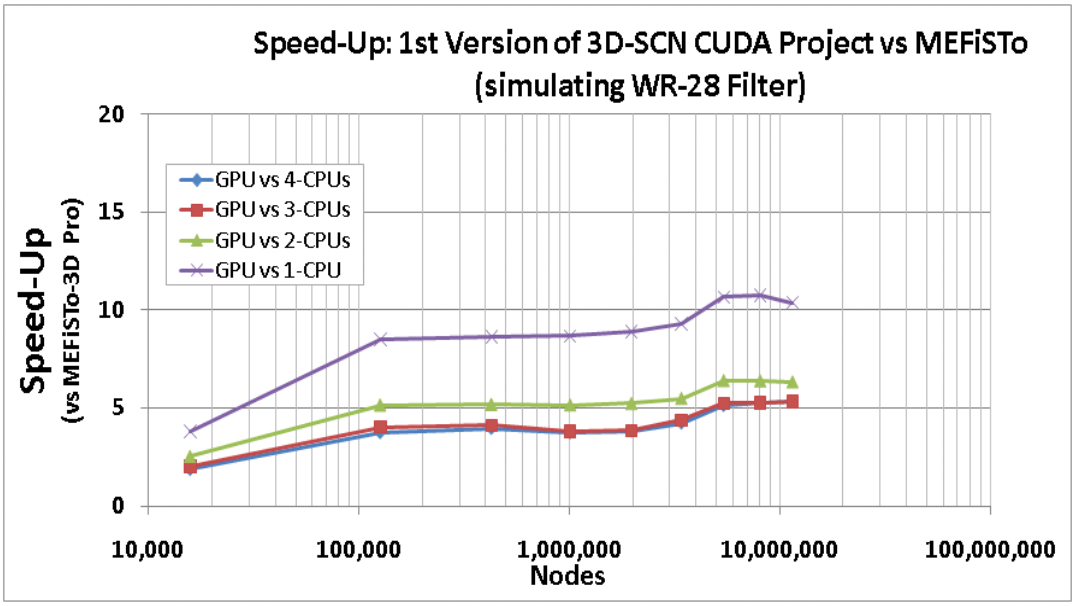


Figure 8.11: Speed-up of the first version of the 3D-SCN CUDA project vs MEFiSTo

Figure 8.12, shows speed-up comparisons of the first version of the 3D-SCN CUDA application versus MEFiSTo. At the worst case (low node counts) an 8.5 times speed-up was measured (compared to 4 CPUs). However, this dramatically changed as the volume of nodes in the filter increased. For the best case, (at high volume of nodes), the speed-up factor is 120 times. This occurs when MEFiSTo runs on a single CPU core. As was

mentioned previously, it would be likely that MEFiSTo-3D Pro would be configured to utilize as many cores as possible. The peak 3D-SCN CUDA application speed-up compared to 4 CPU cores is 60 times. This 60 times speed-up factor is considered a pronounced improvement over a conventional CPU centric SCN TLM solver.

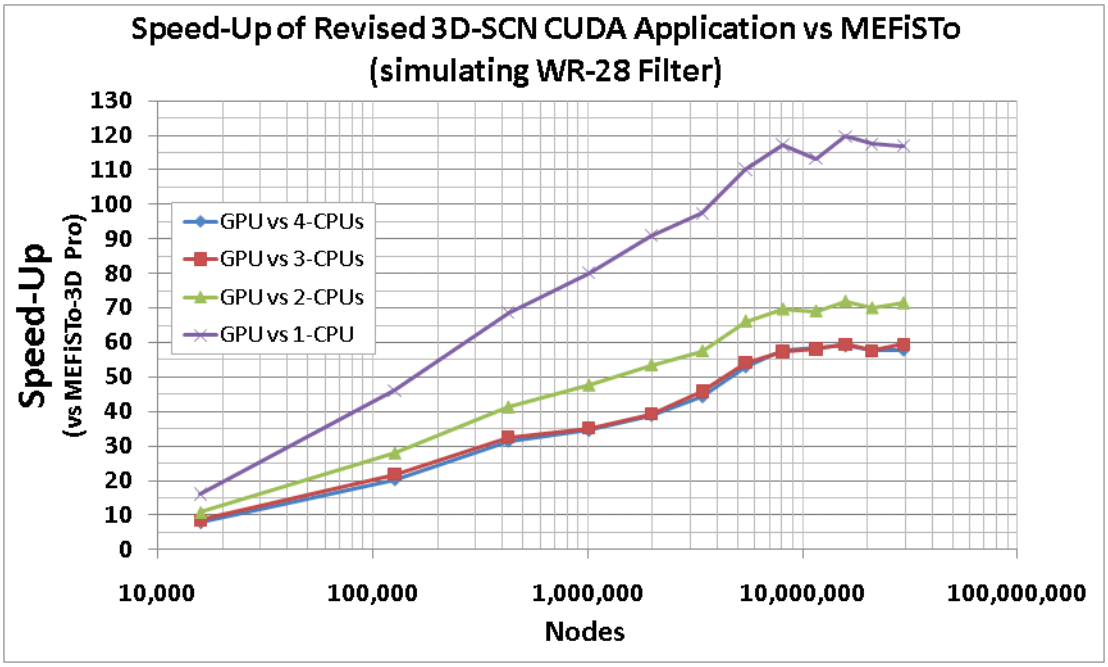


Figure 8.12: Speed-up of the 3D-SCN CUDA application (with revised scattering kernels) versus. MEFiSTo-3D Pro in simulating a WR-28 Filter

Figure 8.13 depicts a speed-up comparison between the first and second versions of the CUDA 3D-SCN scattering kernels. The plot shows that at the lower range of nodes, the CUDA 3D-SCN scattering kernels performed 4 times faster than the first version. However, at higher node counts the speed-up is 11, or a full order of magnitude difference. This demonstrates that the additional effort made to put into practice the advanced optimization techniques (memory coalescing and occupancy), yields a full

order of magnitude of improvement from an inexperienced/unoptimized approach to kernel design.

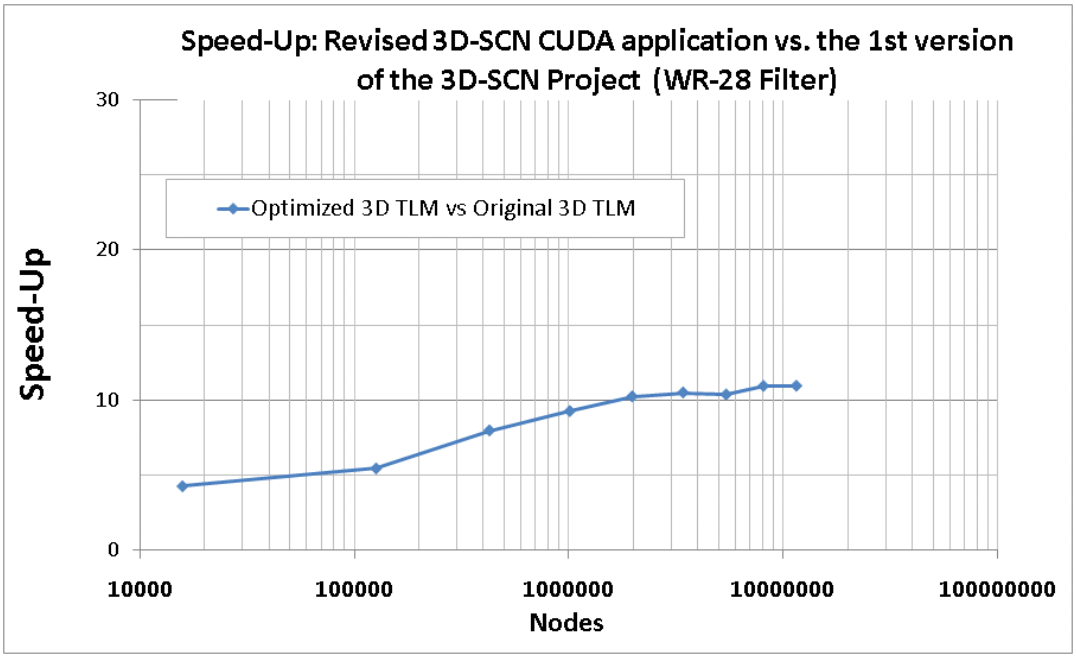


Figure 8.13: Comparison of the speed-up between the old and new versions of the CUDA 3D-SCN programs (both modeling a WR28 waveguide band-pass filter)

Chapter 9 Scalability and Performance

9.1 Introduction

Thus far, only one GPU was engaged to accelerate the 2D-TLM and 3D-SCN algorithms. A cluster of GPUs is required when mesh sizes extend beyond the confines what a GPU's memory can handle. This chapter predicts the scalability and performance of GPU based algorithms for running on a cluster of GPU..

A GPU contains only a finite amount of memory, and hence can only be used to model TLM meshes of finite size. The GPU used for this thesis contains 1.5GB of memory and can accommodate a maximum of 32 million nodes (or a 317×317×317 cubic node structure). Advanced GPUs that are currently available (M2050) contain up to 6GB of memory. Many electromagnetic applications require mesh sizes larger than what a single GPU allows. The mesh sizes depend on the dimensions and resolution of the structure to be simulated.

9.2 Scaling the Mesh Size and Resolution

When the resolution for a structure is doubled, the number of nodes increases by a factor of 8. As well, the time-step decreases by a factor of two, which requires a simulation run of twice the number of iterations to achieve the same elapsed simulation time. Electromagnetic applications which require higher resolutions also require higher time-steps to converge. To keep run-times down to reasonable durations (hours not days), a case can be made that GPU accelerated TLM packages are best suited for structures of

with high resolutions. Moreover, a cluster of GPUs can be used to overcome the memory limitation of a single GPU.

9.3 Implementation Suppositions

The TLM mesh to be modeled could be partitioned into as many GPUs that are in the cluster. An interface surface must be established between mesh-partitions where an exchange of voltage data must take place at every iteration. At the start of each iteration the workstation must read the voltage data from each interface surface of each mesh-partition. The workstation must then write the appropriate voltage data to their neighbouring mesh-partition (GPUs). Then the TLM kernels can be executed.

There are hardware limitations that must be taken into account when using a cluster of GPUs. Currently, motherboards can support up to four GPUs (on four PCI-Express slots). Recall that the bandwidth limit of the PCI-Express standard (v2) is 8GB/sec, which is an order of magnitude slower than the GPUs internal memory of 76.8GB/sec. A potential bottleneck can occur between the workstation and GPUs. Therefore, care must be taken to ensure that the interface between mesh partitions are as efficient as possible.

The software running on the PC must accommodate multithreading in order to issue command to each GPU simultaneously. The workstation must have as much memory as the total sum of memory on the GPUs in order to contain a copy of the mesh on the workstation. The workstation must also be able to synchronize the threads that are

launched for each GPU. This would effectively balance the load so that all GPUs are simultaneously executing kernels.

The 3D-SCN kernels would not require any modification. As long as the workstation can transfer interface values from adjacent mesh-partitions into the appropriate surface of meshes in the GPUs, the scattering kernels operate as they did before without consequence to continuity. Similarly, the boundary kernels and sampling kernels should require very few modifications. The excitation kernel, however, would require some modification since a half-sine Gaussian plane excitation may span across multiple mesh-partitions.

After scattering stage 1 or 2 completes, mesh-partitions requires a voltage exchange with neighbouring mesh-partitions. This cannot be done directly between GPUs, and must be facilitated by the workstation. The transfer of data between the GPU and workstation is best when memory is organized contiguously. Consider all the nodes that are adjacent to a partition-barrier. Only one link-line of each node needs to exchange voltage data with its counterpart in an adjacent mesh-partition. Using the second 3D-SCN design, the organization of link-line voltages in global memory are grouped by link-line voltages which help simplify the exchange process. In addition, each group of link-line voltages stored in memory are ordered first by the x-direction then by the y-direction. It is then convenient to enforce mesh partitions along the xy-plane.

9.4 Predicted Performance

A calculation of performance can be done to explore the impact of the PC-to-GPU data transfer limit. Figure 9.1 shows an estimate of the 3D-SCN kernel performance if a mesh is partitioned and distributed across a cluster of four GPUs. The estimate assumes that each GPU contains close to the maximum number of nodes it allows. For example, the 1.5 GB of memory in the Quadro FX 5600 allows a maximum of 32 million nodes per GPU. This would allow the GPUs to execute their TLM kernels near the maximum node rate. It is also assumed that each GPU mesh-partition requires an interface (exchange of voltage impulses) with adjacent mesh-partitions. It is recognized that the number of mesh structure interfaces (2D faces) that are required per partition mesh-depends on how these mesh-partitions are arranged. Figure 9.1 shows a side-by-side arrangement. Only one interface would be required for the mesh-partitions at the ends (single sided partitions), where two would be required by the middle partitions (double sided partitions).

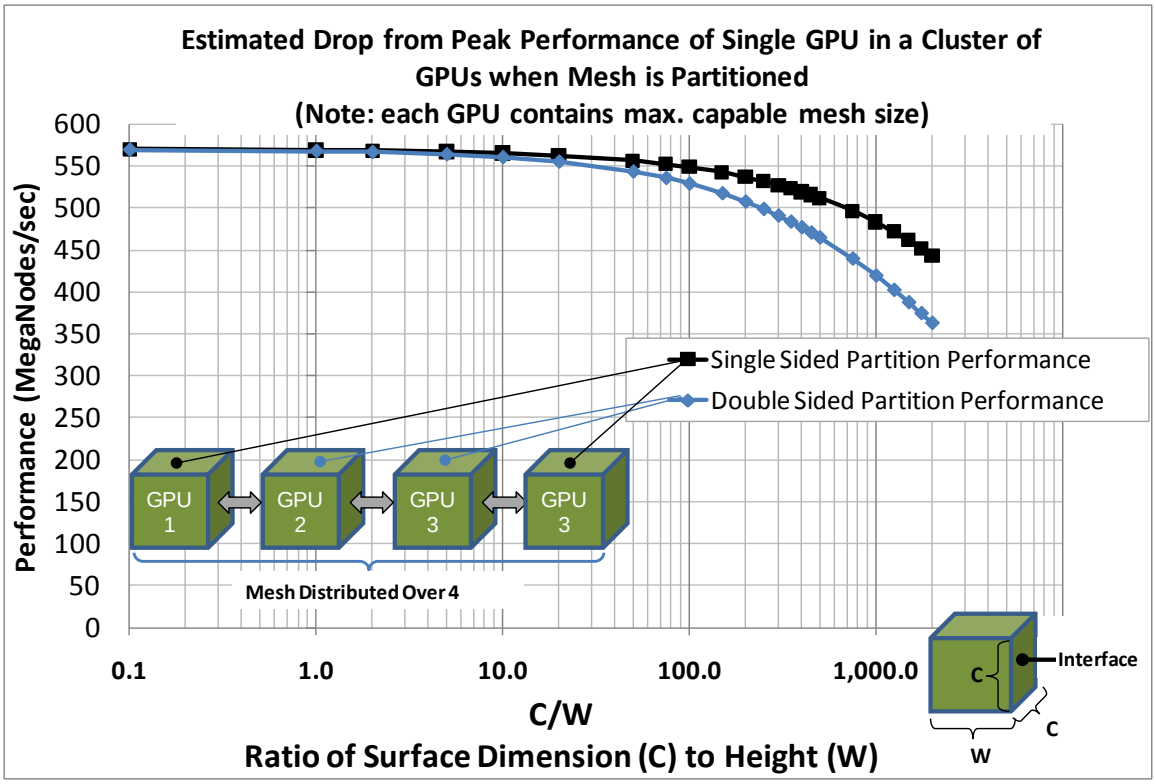


Figure 9.1: Estimate of performance of a single GPU engaged in a cluster of GPUs. Each GPU is filled with the maximum mesh size it contains, but the dimension of the mesh is varied such that the interfacing area between GPU meshes vary.

Figure 9.1 shows that as the surface dimension (C) increases (and surface area increases) the impact of PC-to-GPU data transfer reduces the overall speed. At unity (C/W=1) the partitions are cubic, and less than a 0.4% drop in performance is noted. When the ratio (C/W) is at 150, the width of the mesh-partition (W) is 11 nodes, and the height and depth dimension (C) is 1687 nodes. This very flat mesh partition would cause the system to only suffer a 5% drop in performance. Therefore, for all practical purposes, the transfer rate between the workstation and GPU should not negatively impact performance of the overall TLM execution.

Figure 9.2 shows the predicted performance of a cluster of 4 GPUs using the second 3D-SCN design outlined in chapter 7. The prediction utilizes calculations that are made above, but ensures that conservative estimates are used. This means that only the double sided mesh partition speeds are used in the calculation, as they were the slowest.

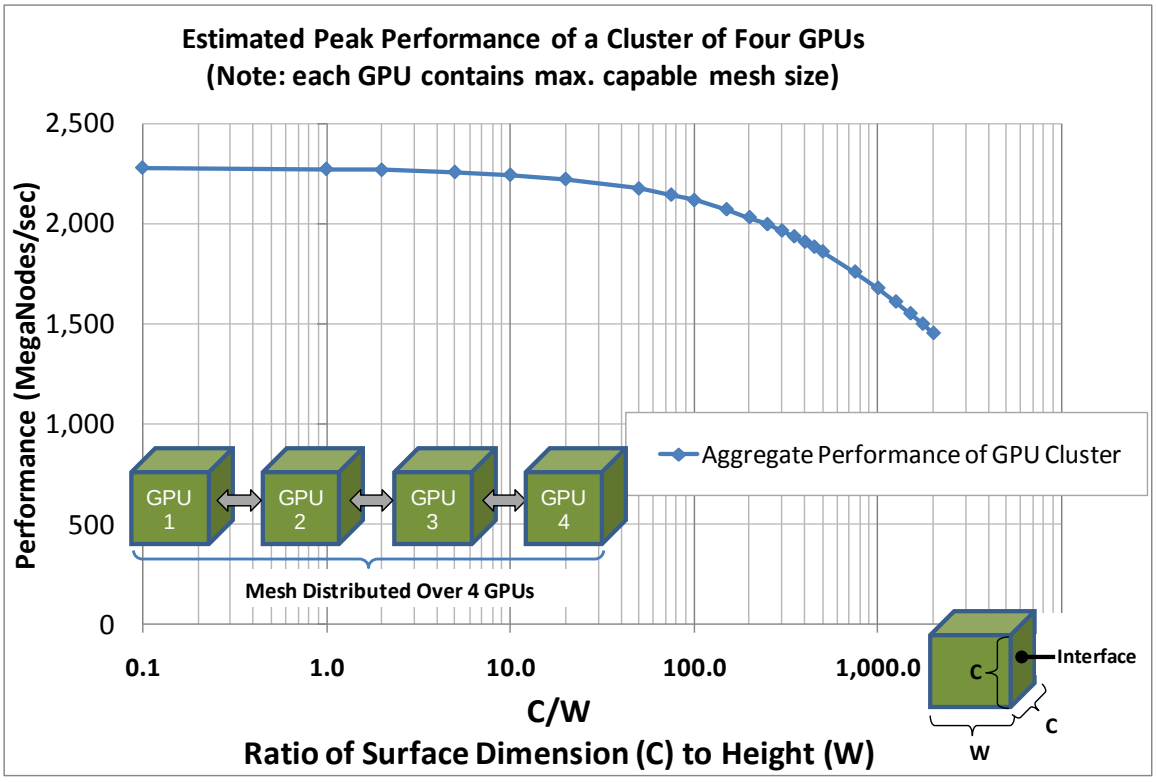


Figure 9.2: The predicted performance of a cluster of 4 GPUs executing the second design of the 3D-SCN kernels.

The aforementioned calculations shed some insight into the predicted performance of a cluster of GPU used for 3D-SCN execution. The results are optimistic, where little impact is predicted by the PC-to-GPU transfer limit. However, an actual implementation of a GPU cluster must be carried out to compare the predicted node-rates with measured ones. Other factors, such as excessive overhead for kernel calls, or unbalanced timing for

threads on the workstation may impact speed to a greater extent than what is explored here.

The imperative to scale to GPU clusters is evident if larger or more refined mesh structures are sought. In order to extend mesh sizes beyond four GPUs, a more elaborate scheme must be employed, where MPI (Message Passing Interface) systems is one good option. It is left as future work to explore the scalability of GPUs in clusters.

Chapter 10 Conclusions and Future Work

10.1 Overview of Completed Work

10.1.1 2D-TLM

Initially, the 2D-TLM algorithm was adapted to the GPU with only a fundamental understanding of the GPU architecture and its programming model. Even when utilizing only elementary GPU concepts, the performance of the CUDA 2D-TLM program achieved faster speeds than CPU based TLM programs, namely MEFiSTo-2D Classic and a TLM code with multithread capability implemented with OpenMP. The peak speedups are summarized in Table 10-1.

	Speed-Up 2D-TLM CUDA Kernel vs.
MEFiSTo-2D Classic	25.1
2D-TLM Serial Code	7.4
2D-TLM OpenMP Code	4.6

Table 10-1: Speed-up of the CUDA 2D-TLM kernels compared with:(1) MEFiSTo-2D Classic, (2) a serial 2D-TLM algorithm running on the CPU, and (3) a 2D-TLM OpenMP version.

10.1.2 3D-SCN

Using similar methods that were used for the GPU based 2D-TLM adaptation, a 3D-SCN kernel was created, validated, and measured against MEFiSTo-3D Pro (chapter 5). Subsequent to this, GPU optimization techniques were explored in chapter 6, and were used to revise the 3D-SCN kernel design (chapter 7). In addition, a unique approach was formulated to augment the 3D-SCN scattering execution stage to reduce the number of

memory transactions per iteration by designing two scattering kernels (Scattering_Stage_1 and Scattering_Stage_2). Table 9-2 summarizes the speed-up results when these kernels were used to compute the response of a WR28 waveguide filter.

	Speed-Up 3D-SCN CUDA (ver 2)	Speed-Up 3D-SCN OpenCL
MEFiSTo-3D Pro (1-CPU core)	119.7	94.9
MEFiSTo-3D Pro (4-CPU cores)	59.4	47.1
3D-SCN CUDA (ver 1)	11.0	9.0
3D-SCN OpenCL	1.3	-

Table 10-2: Comparison of performance among various versions of 3D-SCN programs: the CUDA and OpenCL versions, MEFiSTo-3D Pro (1 and 4 CPU cores), and the first version of CUDA 3D-SCN.

10.1.3 Memory Model Performance

The memory transfer rate of the memory model design, described in chapter 6 achieves 85% of the maximum effective memory bandwidth of the GPU hardware. When this memory model was adapted to implement the 3D-SCN algorithm, (chapter 7 and 8) its node-data-rate achieves 71 % of the effective maximum memory bandwidth of the GPU. The speed of the first version of the 3D-SCN kernels, in comparison, achieves only 6% of the maximum effective memory bandwidth. We can conclude that any further improvements in speed of the latest scattering routine would be marginal, at best, for the current version of the CUDA hardware. However, further research work based on OpenCL to improve the performance of other kernels (boundary, excitation, and sampling) is warranted as OpenCL has become the industrial standard for GPU computing.

10.2 Performance conclusions

Transmission Line Matrix modeling of electromagnetic problems can be very taxing on computing resources. When new forms of computing technologies emerge, such as the GPU, it is worth investigating their potential application to this computationally intensive field. The principle objective of this work was to investigate this new hardware paradigm in order to accelerate 2D and 3D Transmission Line Matrix algorithms.

In all attempts, the adaptation of TLM algorithms resulted in speed-ups that were superior to both commercially available TLM solvers (MEFiSTo) and in-house TLM code (2D TLM OpenMP). Speed-up results varied depending on the approach of each kernel design. In the first design of the 3D-SCN kernels described in chapter 5, modest speed-ups were attained even without using some advanced GPU optimization techniques. Chapter 6 addressed the GPU performance optimization techniques and proposed a design to balance the requirements of the 3D-SCN operations and the constraints of GPUs. The new kernel design achieved a full order of magnitude increase in computation speed for the 3D-SCN engine.

10.3 GPU programming effort vs. payback

Speed-ups are quantifiable, but the development effort to adapt existing algorithms (such as TLM) to unconventional hardware such as the GPU can be difficult to measure. Even an experienced CUDA or OpenCL developer may need to spend a great deal of effort and time in the conversion process as each algorithm presents unique implementation challenges. This is especially true when optimized GPU methods are sought after. When a naïve approach is applied to adapting algorithms to GPUs, one can still obtain speed-ups on the order of 2 to 10 times. This may be more than enough acceleration for many applications. To achieve a further order of magnitude in performance, additional effort must be made to ensure memory access is efficient, and multiprocessor resources are conserved. This may also mean reformulating the manner in which the target algorithm functions, as was the case for the two separate 3D-SCN scattering kernels.

10.4 Future Work

10.4.1 Optimizing the 2D-TLM Kernels

The advanced optimization techniques explored in chapter 6 could be applied to revise the 2D-TLM kernel design. The memory model design that was used for the revised 3D-SCN design, described in chapter 7, could also be applied to a 2D-TLM kernel design. The only difference in the memory model design would be that only one layer of nodes in the *z*-plane is required. The design could start with the 3D-SCN kernels, then altering

each of the kernels' code to include 2D-TLM computations. Since the 2D-TLM node requires only 1/3 of the memory occupied by the 3D-SCN node, it is not unreasonable to expect a further order of magnitude improvement to the speed of the 2D-TLM kernels if this revision is made. However, this may not be necessary as it is possible to configure a 3D-SCN mesh with only one layer of nodes (*z*-dimension = 1) to simulate 2D-TLM functionality. It is recommended that the 3D-SCN kernels be augmented for 2D computations, and asses its performance before attempting to redesign the 2D-TLM kernel.

10.4.2 GPU Clusters

As is mentioned in chapter 9 the effort, so far, has been focused on the implementation on a single GPU. Further improvements to speed-up rates can be achieved if clusters of GPUs are used. Two advantages that were explored in chapter 9 are:

- The total size of a mesh can span beyond the confines of a single GPU since each GPU's global memory is fixed.
- The speed-up can increase further by employing additional GPU multiprocessors to a partitioned mesh.

Further work on adapting TLM to GPUs should focus on coordinating clusters of GPUs to solve larger problems faster. As OpenCL and GPU hardware become mature, direct GPU to GPU data transfer may be available in the not too distant future.

10.4.3 Boundary Kernels

As was revealed in the speed tests discussed in chapter 8, the boundary kernel could be improved as it significantly reduces the speed of the current 3D-SCN implementations. In addition, the current boundary kernels can only handle simple boundary planes that must be parallel with the *xy*-, *xz*-, and *yz*- planes. The boundary embedding technique described in chapter 5 is much more flexible, yet it requires the boundary computations to be included in the same kernel as the scattering kernel. If applied to the second version of the 3D-SCN scattering kernels, this boundary technique could impact resource usage in the kernel, thus affecting thread-block occupancy. The impact on speed is hard to deduce without direct tests, so it is recommended that this technique be attempted and tested.

10.4.4 Expanding adaptation of TLM techniques

So far, the types of electromagnetic structures that can be modeled with 3D-SCN and 2D-TLM are air filled. The absorbing boundaries that were implemented can only absorb relatively narrow bandwidths. The boundaries that define structures were restricted to simple planes. Computational electromagnetic engineering problems include a multitude of irregular geometries not just for filter simulations, but also for mode converters, antenna design, microstrip design, and many others. Further work should focus implementing the generalized symmetrical condensed node (GSCN) algorithms which can handle inhomogeneous and anisotropic materials.

Bibliography

[1] S.E. Krakiwsky, L.E. Turner and M.M. Okoniewski, "Graphics Processor Unit Acceleration of Finite-Difference Time-Domain Algorithm", *Proceedings of IEEE International Symposium on Circuits and Systems*,vol.5, pp. V-265 – V268, May 23-26, 2004.

[2] M.J. Inman, M.J and A.Z. Elsherbeni, "Programming video cards for computational electromagnetics applications", IEEE Antennas and Propagation Magazine, vol. 47, no. 6, pp. 71 – 78, December 2005.

[3] H. Takizawa, N. Yamada, S. Sakai, and H. Kobayashi, "Radiative Heat Transfer Simulation Using Programmable Graphics Hardware", *5th IEEE/ACIS International Conference on Computer and Information Science*, pp. 29 – 37, July 10-12, 2006.

[4] Z. Luo; H. Liu; X. Wu, "Artificial Neural Network Computation on Graphic Process Unit", *Proceedings of IEEE International Joint Conference on Neural Networks*, vol. 1, pp. 622 – 626, Jul. 31 to Aug. 4, 2005.

[5] S. Harding, W. Banzhaf, "Fast Genetic Programming and Artificial Developmental Systems on GPUs", *21st International Symposium on High Performance Computing Systems and Applications*, pp. 2, May 2007.

[6] F. Zhe, Q. Feng, A. Kaufman and S. Yoakum-Stover, "GPU Cluster for High Performance Computing", *Proceedings of the ACM/IEEE Conference on Supercomputing*, pp. 47, 2004.

[7] folding.stanford.edu/FAQ-ATI.html [June 2010].

[8] Blaauw, David, and Shidhartha Das. "CPU, Heal Thyself." *IEEE Spectrum* Aug. 2009. Internet:http://spectrum.ieee.org/semiconductors/processors/cpu-heal-thyself/0, June 2010.

[9] Matsuoka1, Satoshi. "GPU Accelerated Computing—from Hype to Mainstream, the Rebirth of Vector Computing." *Journal of Physics* 180th ser. (2009).

[10] Dokken, Tor, Trond R. Hagen, and Jon M. Hjelmervik. "The GPU as a High Performance Computational Resource."*Proceedings of the 21st Spring Conference on Computer Graphics*. Spring Conference on Computer Graphics, Budmerice, Slovakia.21-26.

[11] C. Christopoulos, *The Transmission-Line Modeling Method: TLM*, IEEE Press, New York, 1995.

[12] P.B. Johns, and R.L. Beurle, "Numerical Solution of 2-Dimensional Scattering Problems using a Transmission-Line Matrix", Proc. IEEE, Vol.118, No.9, pp.1203-1208, Sept. 1971.

[13] P.B. Johns, "A Symmetrical Condensed Node for the TLM Method", *IEEE Transactions on Microwave Theory Tech*., Vol. MTT-35, No. 4, pp 370-377, Apr. 1987.

[14] Dimopoulos, Nikitas J., and Kin F. Li. *High Performance Computing Systems and Applications*. Boston: Kluwer Academic, 2002, pp. 123-128.

[15] *http://www.cst.com/Content/Events/Details.aspx?eventId=1483*, [June 2010].

[16] *http://www.faustcorp.com/products/mefisto3dpro/index.html,* [May 2010].

[17] Quadro FX5600 Product Info. 2008. http://www.nvidia.com/object/quadro_fx_5600_4600.html, 2008 [June 2010].

[18] NVIDIA CUDA: Compute Device Architecture Programming Guide, v3.0, NVIDIA Corporation, Santa Clara, CA, USA. 2010.

[19] OpenCL Programming Guide for the CUDA Architecture, v2.3, NVIDIA Corporation, Santa Clara, CA, USA. 2010.

[20] "Khronos OpenCL API Registry." *The Khronos Group: Open Standards, Royalty Free, Dynamic Media Technologies*. Internet: http://www.khronos.org/registry/cl/ [May 2010].

[21] S. Akhtarzad and P.B. Johns, "Solution of 6-component electromagnetic fields in three space dimensions and time by the T.L.M. method," *Electron.Lett.*, Vol. 10, No. 25/26, pp.535-537, Dec. 1974.

[22] C. Huygens, "Traite de la Lumiere", Leiden, 1690, Pierre vanderAa, reprinted in "Oevres Completes de Christiaan Huygens", Societe Hollandaise de Sciences, Vo. 19, Amsterdam 1967, Swets and Zeitlinger.

[23] Taflove, Allen, and Susan C. Hagness. *Computational Electrodynamics: the Finite-difference Time-domain Method.* Boston: Artech House, 2005.

[24] W.J.R. Hoefer, and P.P.M So, *The MEFiSTo-2D Theory*. Victoria: s.n., 1998, pp. 3-180.

[25] "HP Xw9400 Workstation."Internet: http://www.bluefish444.com/downloads/Hardware/xw9400.pdf, [May 2010].

[26] Chandra, Rohit. *Parallel Programming in OpenMP*. San Francisco, CA: Morgan Kaufmann, 2001.

[27] IEEE Standard 1597.1 Standard for Validation of Computational Electromagnetics Computer Modeling and Simulations, 2009.

[28] Pozar, David M. *Microwave Engineering*. Hoboken, NJ: John Wiley, 2005.

[29] "Hardware - SIMD Executive Summary." *Apple Developer*. Internet: http://developer.apple.com/hardwaredrivers/ve/summary.html, [May 2010].

[30] F.V. Rossi, P.P.M. So, N. Fichtner and Peter Russer, “Massively Parallel Two-Dimensional TLM Algorithm on Graphics Processing Units,” IEEE International Microwave Symposium, June 2008.

[31] "ClusterInABox Quad (Q30) Product Info."Internet: http://www.acceleware.com/default/index.cfm/our-products/clusterinabox-quad,2008, [April 2009.],

[32] "PCI-SIG - PCI Express." *PCI-SIG - Home*. Internet: http://www.pcisig.com/specifications/pciexpress, [May 2010].

[33] NVIDIA CUDA: Best Practices Guide, v3.0, NVIDIA Corporation, Santa Clara, CA, USA. 2010.

[34] Kirk, David, and Wen-mei Hwu. *Programming Massively Parallel Processors: a Hands-on Approach*. Burlington, MA: Morgan Kaufmann, 2010.

[35] Ryoo, Shane, Christopher I. Rodrigues, and Sara S. Baghsorkhi. "Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA."*Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. Principles and Practice of Parallel Programming, Salt Lake City, UT, USA. 2008. 73-82.

[36] F. Rossi and P.P.M. So, Parallelized Three- Dimensional TLM Algorithms on a Graphics Processing Unit, in the 25th International Review of Progress in Applied Computational Electromagnetics Symposium (ACES 2009), Monterey, CA, pp. 110–114, March 8–12, March 2009.

[37] Rossi, F.; So, P.P.M.; , "Hardware accelerated symmetric condensed node TLM procedure for NVIDIA graphics processing units," *Antennas and Propagation Society International Symposium, 2009. APSURSI '09.IEEE* , vol., no., pp.1-4, 1-5 June 2009.

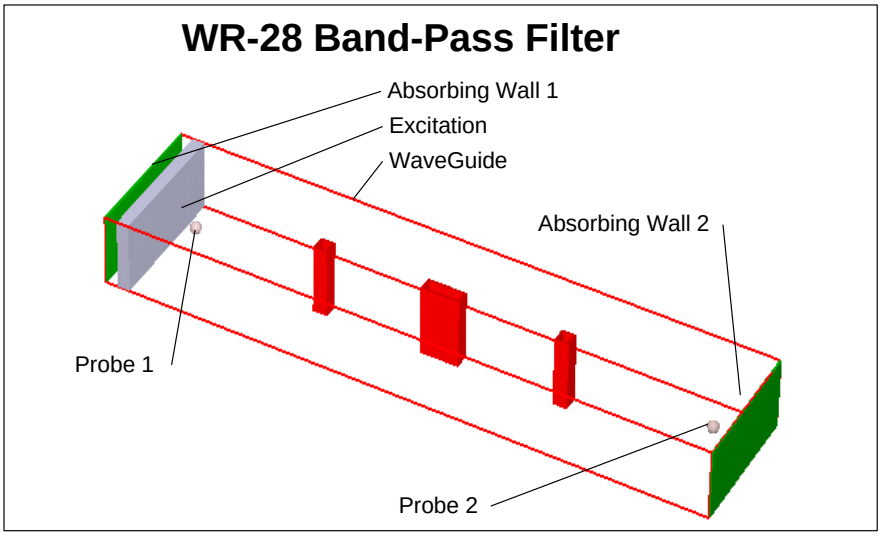
[38] Rossi, F.; So, P.; , "Parallelized computational electromagnetics TLM algorithms on NVIDIA graphics processing units," *Communications, Computers and Signal Processing, 2009. PacRim 2009. IEEE Pacific Rim Conference on* , vol., no., pp.814-819, 23-26 Aug. 2009

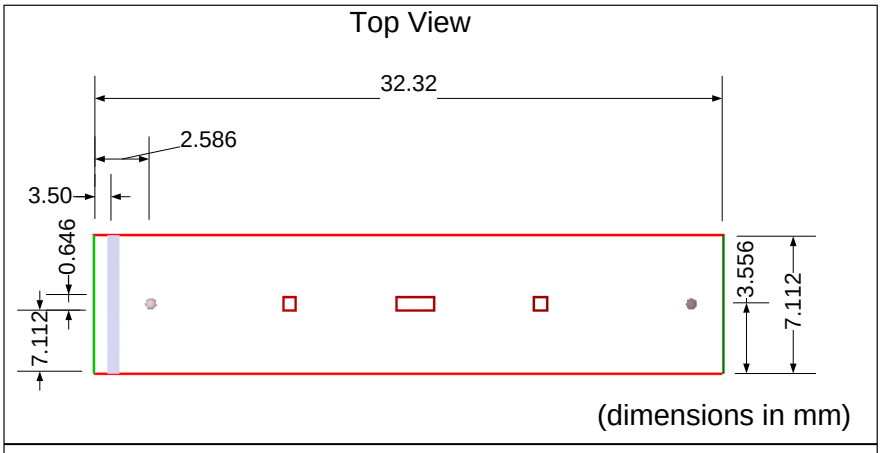
[39] Rossi, F.; So, P.P.M.; , "Accelerated symmetrical condensed node TLM algorithms for NVIDIA CUDA enabled graphics processing units," *Electromagnetics in Advanced Applications, 2009. ICEAA '09. International Conference on* , vol., no., pp.170-173, 14-18 Sept. 2009

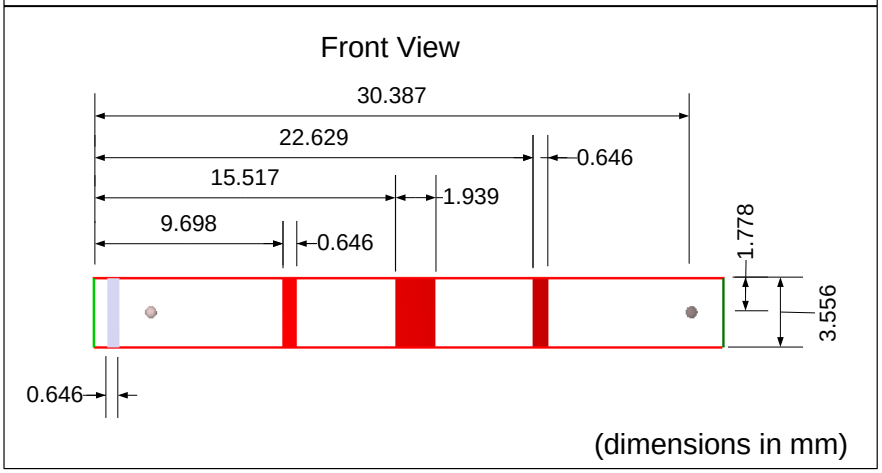
[40] Quinn, Michael. *Parallel programming in C with MPI and OpenMP*. McGraw-Hill Science/Engineering/Math, 2004.

[41] " CST: Computer Simulation Technology", Internet: http://www.cst.com/ [June 2010].

Appendix A: Filter Specifications

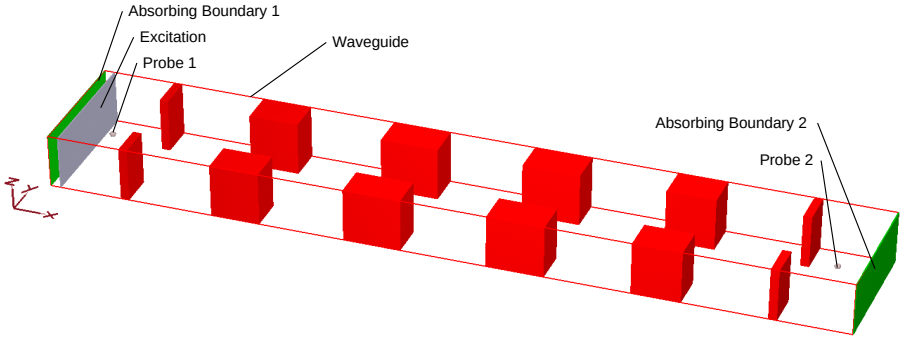


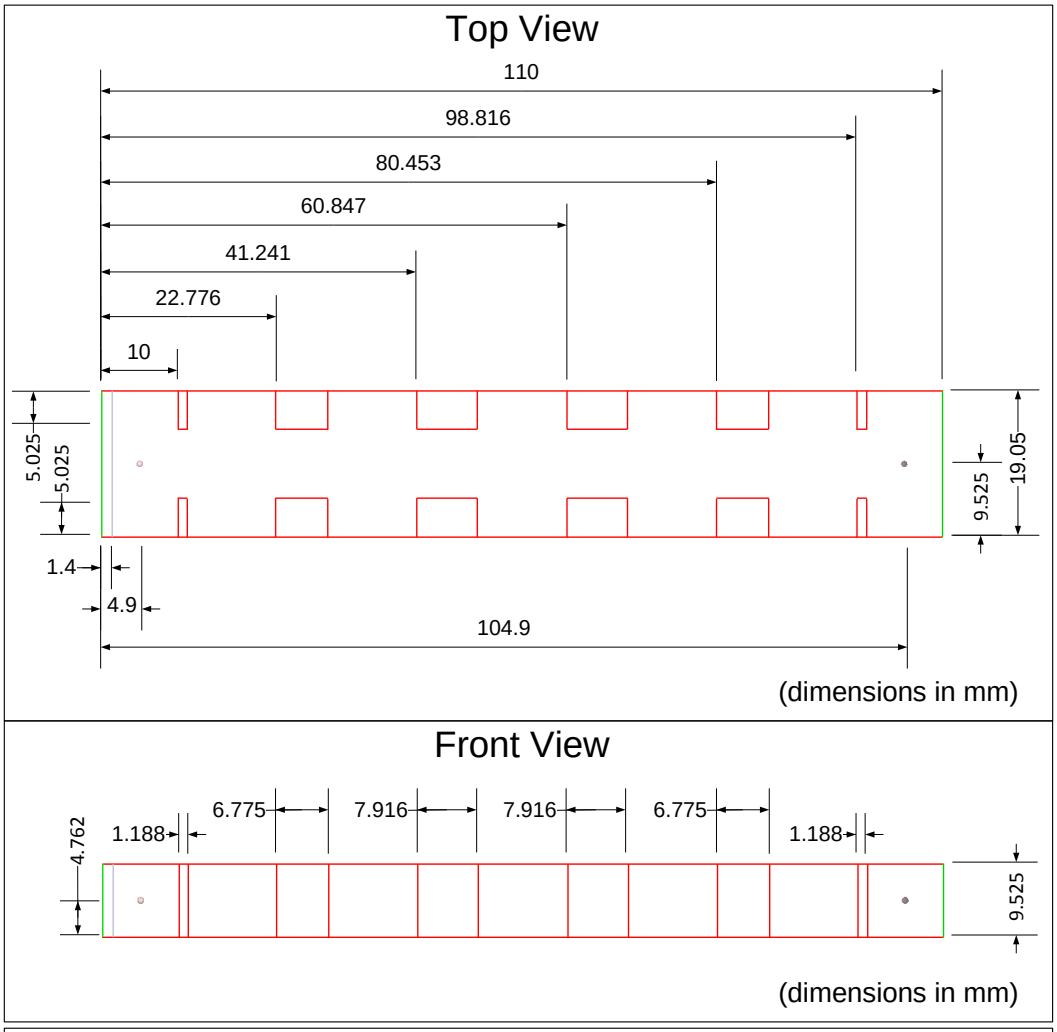




Excitation:
Centre Freq. = 31 GHz , 3dB Bandwidth = 5 dB

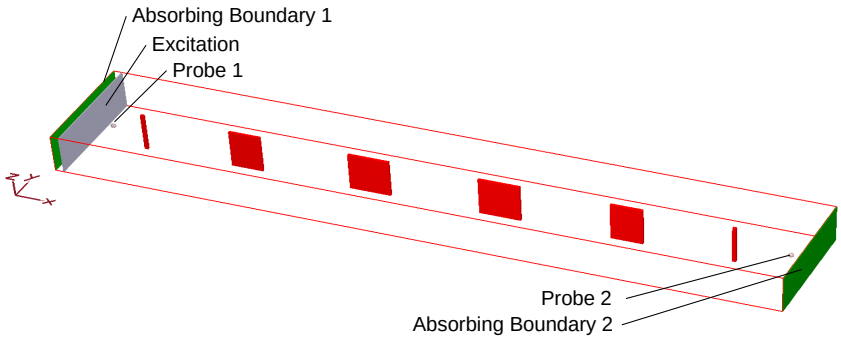
WR-75 Band-Pass Filter

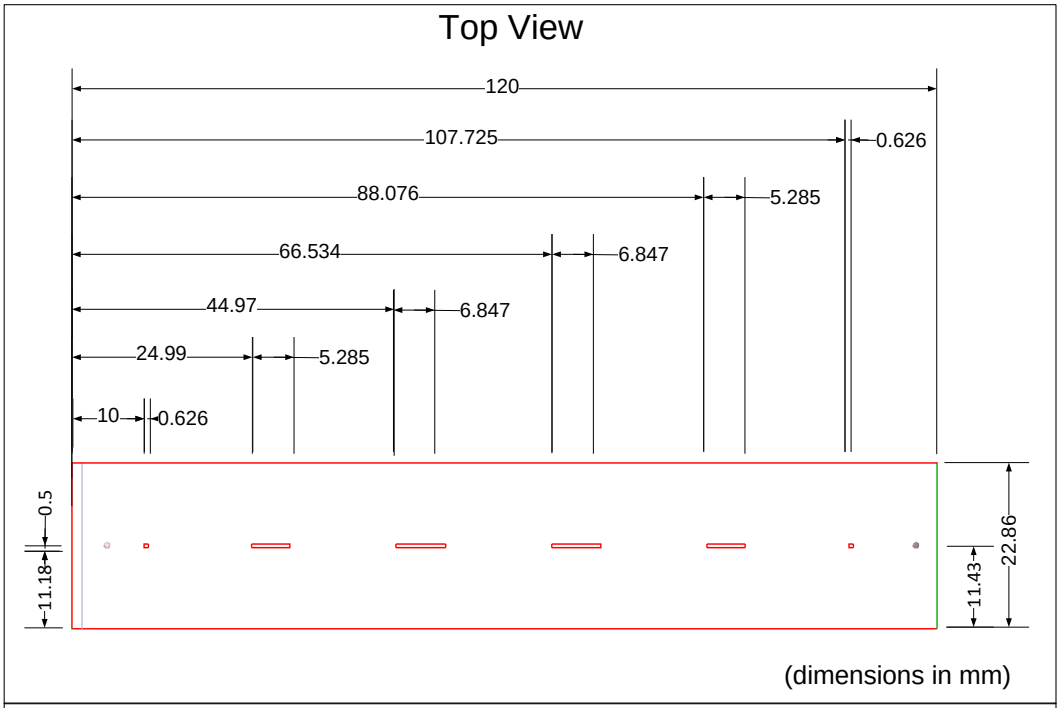


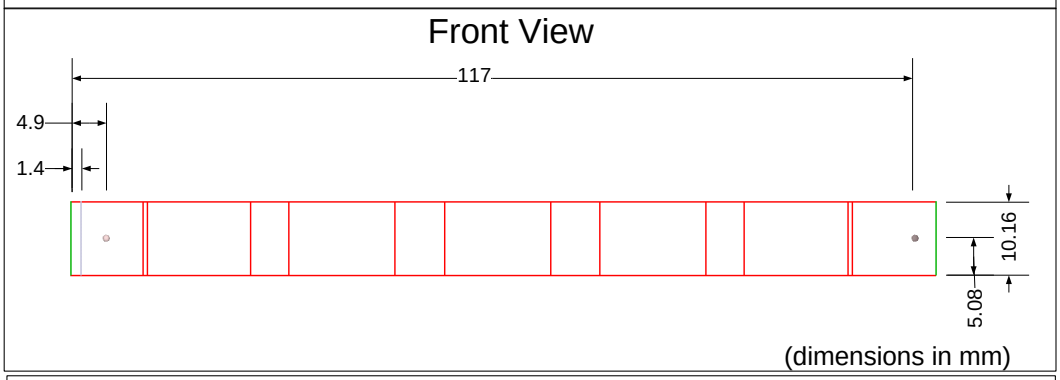


Excitation:
Centre Freq. = 12.5 GHz , 3dB Bandwidth = 1 dB

WR-90 Band-Pass Filter







Excitation: Centre Freq. = 9.95 GHz , 3dB Bandwidth = 1.5 dB

Index