Thesis

Optimal Design Space Exploration for FPGA-based Hardware Accelerators: A Case Study on 1-D FDTD

Submitted by

Mugdha Puranik

Department of Electrical and Computer Engineering

In partial fulfillment of the requirements

For the Degree of Master of Science

Colorado State University

Fort Collins, Colorado

Fall 2015

Master's Committee:

Advisor: Sanjay Rajopadhye

Sudeep Pasricha
Yashwant Malaiya

Abstract

Optimal Design Space Exploration for FPGA-based Hardware Accelerators: A Case Study on 1-D FDTD

Hardware accelerators are optimized functional blocks designed to offload specific tasks from the CPU, speed up them up and reduce their dynamic power consumption. It is important to develop a methodology to efficiently implement critical algorithms on the hardware accelerator and do systematic design space exploration to identify optimal designs. In this thesis, we design, as a case study, a hardware accelerator for the 1-D Finite Difference Time Domain (FDTD) algorithm, a compute intensive technique for modeling electromagnetic behavior. Memory limitations and bandwidth constraints result in long run times on large problems. Hence, an approach which increases the speed of the FDTD method and reduces bandwidth requirement is necessary. To achieve this, we design an FPGA based hardware accelerator.

We implement the accelerator based on time-space tiling. In our design, p processing elements (PEs) execute p parallelogram shaped tiles in parallel, each of which constitutes one tile pass. Our design uses a small amount of redundant computation to enable all PEs to start nearly concurrently, thereby fully exploiting the available parallelism. A further optimization allows us to reduce the main memory data transfers of this design by a factor of two. These optimizations are integrated in hardware, and implemented in Verilog in Alteras Quartus II, yielding a PE that delivers a throughput of one iteration (i.e., two results) per cycle. To explore the feasible design space systematically, we formulate an optimization problem with the objective of minimizing the total execution time for given resource constraints. We solve the optimization problem analytically, and therefore have a

provably optimal design in the feasible space. We also observe that for different problem sizes reveal that the optimal design may not always match the common sense intuition.

Table of Contents

List of Tables

List of Figures

CHAPTER 1

Introduction

Stencil computations constitute a large fraction of scientific computations in diverse areas such as electromagnetics, image processing and fluid dynamics. Stencil codes involve computations that are largely independent of each other and have a very regular pattern of execution. This makes them highly amenable for parallelization. A lot of effort has been invested in building accelerators to solve stencil computations at high speed. FPGAs have been increasingly used for building efficient hardware accelerators because of their reconfigurability, low cost, shorter development cycle, reasonable speed grades and low running power. To increase the performance of iterative stencil computation, we can exploit both temporal and spatial parallelism by designing multiple, deeply pipelined compute engines on FPGA. It is important to develop an effective design methodology for deriving optimal hardware for stencil computations. A clear specification of all the transformations needed to implement the stencil computations on FPGAs and design space exploration of potential designs to find the optimal design, would help in building an efficient hardware accelerator. Although most of the research in this field has led to FPGA accelerators with better performance than generic processors, an effective design methodology for deriving optimal hardware for stencil computations has not been well defined.

In this thesis, we design an FPGA-based hardware accelerator that efficiently implements the 1-D Finite Difference Time Domain (FDTD) method. FDTD is an important, compute intensive algorithmic technique for modeling electromagnetic behavior. Hence the accelerator should be designed in such a manner that the speed of execution of FDTD is improved and the off-chip bandwidth requirement is reduced. To achieve this, we develop a three

step approach to implement 1-D FDTD on FPGA. The first step is to analyze the data dependencies of FDTD and propose good tiling transformations. The second step involves the implementation of the actual hardware of the accelerator. And the third step is to perform a thorough design space exploration to understand the impact of various design parameters on the execution time and area and to determine the optimal design for the given problem size with given area constraint.

1.1. Contributions

The main contributions of this thesis are as follows:

(1) A well-defined approach for deriving FPGA-based hardware acceleration of 1-D FDTD

(2) Careful analysis of data dependencies, use of tiling transformations to fully exploit available parallelism and reduce off-chip memory accesses, integration of these optimizations in the hardware design of the accelerator

(3) Parametric Verilog implementation of our design, incorporating a processor that delivers the results of one iteration every clock cycle in the steady state.

(4) Systemic design space exploration by developing analytical models of area and performance and by formulating an optimization problem optimization problem with the objective of minimizing the total execution time for given resource constraints.

1.2. Related Work

1.2.1. Related work on FPGA Implementation of the FDTD method. FPGA technology was first used for implementing FDTD method by Schneider et al. [3] . In this paper the authors describe the custom FPGA-based hardware design of 1-D FDTD and

implements it on FPGA. They implemented a 10 computation cell pipelined bit-serial arithmetic design that runs at 37.5 MHz. Their experimental results show that their hardware design significantly accelerates the simulation of 1-D FDTD compared to the software implementation. Later Durbano et al. [4, 5] proposed a design for three-dimensional FDTD using floating point arithmetic. The describe in detail their hardware accelerator architecture consisting of computation engine, data storage and handling of special boundary nodes. However, due to the use of floating-point arithmetic, slow memory interface and lack of pipelining, the design runs only at 14 MHz and is 9 times slower than the software design running on 2 GHz PC. Chen et al. [6] implement a fixed-point, deeply pipelined custom hardware for two-dimensional FDTD. The design was described in VHDL and implemented on Xilinx Virtex II Pro FPGA chip. The throughput of their design is 13.8 Mcells/s (i.e, millions of grid points updated per second). Pless et al. [7] implemented a fixed-point, deeply pipelined custom hardware for two-dimensional FDTD on FPGA-based Maxeler dataflow computer. They achieved a throughput of 1486 Mcells/s. None of these works included tiling optimizations integrated in their custom hardware design.

Kameyama et al. [8] designed an FPGA-based hardware accelerator for two dimensional FDTD based on overlapped tiling in openCL. However, they do not describe the methodology and synthesized hardware in detail. Also, overlapped tiling involves a lot a redundant computations. This paper does not suggest optimal tile sizes or method for selection of optimal design parameters. We will show later that, based on its data dependencies, FDTD can be tiled in more optimal manner. Wester et al. [9] describe the methodology for transforming higher-order stencils into FPGA-based design in their recent paper on deriving FPGA-based hardware for stencil computations. Their approach applies space/time transformations to the higher-order stencils and they also mention the need to study the trade-offs between

execution time and FPGA resources to find out better design parameters. However they do not provide any mathematical formulation for this.

We present, in detail, a methodology for deriving an FPGA-based hardware accelerator for 1-D FDTD that uses tiling transformations. We also formulate an optimization problem to systematically explore the design space to study the execution time and area trade-offs and to find the optimal design amongst all possible designs.

1.3. Thesis Structure

In chapter 2 we first describe the need of hardware accelerators, then the structure of FPGAs and their suitability for parallelization. We then introduce the FDTD method and derive 1-D FDTD equations. Finally we explain the concept of tiling, as we would be integrating the tiling optimizations in our hardware.

In chapter 3 we analyze the data dependencies of 1D-FDTD, and develop tiling optimzations leading to "nearly" concurrent start for processing elements and reduced off-chip accesses. Later in this chapter we discuss the hardware implementation of 1-D FDTD in Verilog. The Verilog design, the synthesized hardware and its characteristics are explained in detail.

Chapter 4 describes the formulation and solution of the optimization problem to find optimal values of design parameters, with an objective of minimizing the execution time for given resource constraints. We show that this helps in systematic design space exploration which is important for efficient accelerator design. The results of the optimization problem for different problem sizes reveal that for some problem sizes, the optimal design may not always match the intuition. Such observations are useful in convergence testing which is also explained in this chapter.

Chapter 5 draws conclusions and gives suggestions for future work.

CHAPTER 2

Background

In this chapter we present the background information that will help to follow the rest of the thesis. We specifically discuss the advantages of hardware accelerators, FPGA architecture, the FDTD algorithm and space-time tiling.

2.1. Hardware Accelerators

The need of hardware accelerators can be associated with the drawbacks of the the multicore approaches. Ideally, processor performance should increase linearly with each additional core. But, there are many limitations of multicore processors that dampen their performance.

(1) Parallelism: Programs need to be first parallelized to maximize utilization of the computing resources provided by multicore processors. For many workloads additional work required to parallelize software is difficult.

(2) Operating system design: The design of OS for multicore processors is a challenging task, as the requirements for all the cores need to be satisfied.

(3) Energy and Heat Dissipation: In spite of curbing the rise clock frequencies, mulitcore architectures are beginning to hit energy limits. Some power is used up to keep track of shared resources like caches and system bus. Moreover, the higher the number of cores, the higher is the heat radiated from a processor. Large heat sinks are required to cool the processors.

(4) Slower clocks: The voltage scaling era allowed clock scaling and running chips at faster clock speeds. The clock speed of each of the cores on a multicore processor can be slower than those of single core processors they are replacing.

(5) Dark Silicon:For decades, Dennard Scaling model has allowed the chip designers to keep power density (power consumption per unit area of silicon) constant while moving from one technology node to another. However the dependence of leakage power consumption on the threshold voltage has constrained further threshold and supply voltage scaling. This has led to a sharp increase in the power densities that restricts powering-on all the transistors simultaneously, while keeping the chip temperature in safe operating range. Some of the cores cannot be powered-on at nominal voltage for a given thermal design power (TDP) constraint. Esmaeilzadeh et al. [10] refer to this as "Dark Silicon".

An approach that can help in overcoming the limitations of multicores, for certain class of tasks, is hardware acceleration. Hardware accelerators are circuits customized for specific tasks or classes of tasks. Accelerator architectures come in many forms, like Graphic Processing Units (GPUs), Digital Signal Processors (DSPs), ASIC-based accelerators and FPGA-based accelerators. In this thesis, we will be focusing on FPGA based accelerators, as a case study. FPGAs offer many advantages like reconfigurability, low cost, faster turn-around, reasonable speed grades and low running power.

An FPGA is an array of logic gates that can be hardware programmed to implement specific tasks. Special purpose functional units can be devised and used in parallel on an FPGA. FPGAs are good candidates for acceleration of certain applications. We can do custom hardware design for FPGA-based hardware accelerators to improve the performance, energy and power. The memory hierarchy, daptapath operators, pipeline stages, interconnects between processing blocks can be customized for specific application. The accelerator may only be used to perform certain task and turned off at other times. This is especially

useful in the future generations, due to the problem of Dark Silicon. Hardware accelerators are often limited by available resources. More resources means more parallelism but also higher cost and more power consumption. Thus there is a trade-off between hardware resources and speed.

The prime considerations while designing the FPGA-based hardware accelerators are listed below.

(1) Designing the compute engine to fully exploit the available parallelism

(2) Choosing pipeline depth and structure for the combinational path

(3) Designing efficient control logic

(4) Determining on-chip memory requirement and register file organization

(5) Reducing FPGA-to-global memory bandwidth requirements

(6) Increasing register reuse and avoiding idling of resources

(7) Comprehensive design space exploration, to make different architectural and design parameter choices effectively

2.2. FPGA Architecture

A Field Programmable Gate Arrays (FPGA) is an integrated circuit that can be electrically programmed to implement any digital circuit or system. An FPGA needs 20 to 35 times more area, has speed performance 3 to 4 times slower and consumes approximately 10 times more dynamic power as compared to standard cell Application Specific Integrated Circuit (ASIC) [11]. Also for large volume the cost is significantly higher since ASIC design and fab cost can be amortized over the larger market. However FPGAs offer many advantages like reconfigurability, faster turn-around, reasonable speed grades and low running power. FPGAs can be programmed using Hardware Description Languanges (HDL)

like Verilog and VHDL. By using the CAD tools, the design descriptions in HDL can be compiled, synthesized and placed and routed on the target FPGA platform.

2.2.1. Basic FPGA Structure. FPGAs [12], as illustrated in Figure 2.1 , are composed of configurable logic blocks (CLBs) of different types such as general logic, memory, multiplier and input/output blocks surrounded by programmable routing, which can be configured to enable the interconnection between different blocks. The CLBs are arranged in a matrix form in FPGAs and connected by programmable routing fabric. The most widely used programming technologies in modern FPGAs are flash, static memory and anti-fuse. FPGAs can be programmed without detaching the chips from the target platform. It is also possible to program the reconfigurable logic on FPGA at run-time, when the circuit is running on the other part of the chip.

2.2.2. Configurable Logic Blocks. The configurable logic blocks (CLBs) are the fundamental building blocks of FPGA used for implementing the digital logic. A CLB either has a single basic logic element (BLE) or a cluster of locally interconnected BLEs, as shown in Figure 2.2. The structure of CLBs and BLEs may vary in different FPGA chips. Figure 2.3 shows the typical BLE which consists of a look-up table (LUT) and a flip-flop. It has only one output which may be taken directly from the LUT or from the flip-flop. A k-input look-up table is composed of 2^k bit memory units which can be programmed to implement a k-input boolean function or truth-table. Any digital logic can be implemented by configuring one or more LUTs. The output of one BLE is accessible to another BLE. By programming the routing fabric we can connect several BLEs together to fit bigger functions. Modern FPGAs, typically have 4 to 10 BLEs in one CLB. Along with the basic logic logic blocks FPGAs also contain some special purpose blocks like memory, multipliers, adders, DSP blocks etc. These

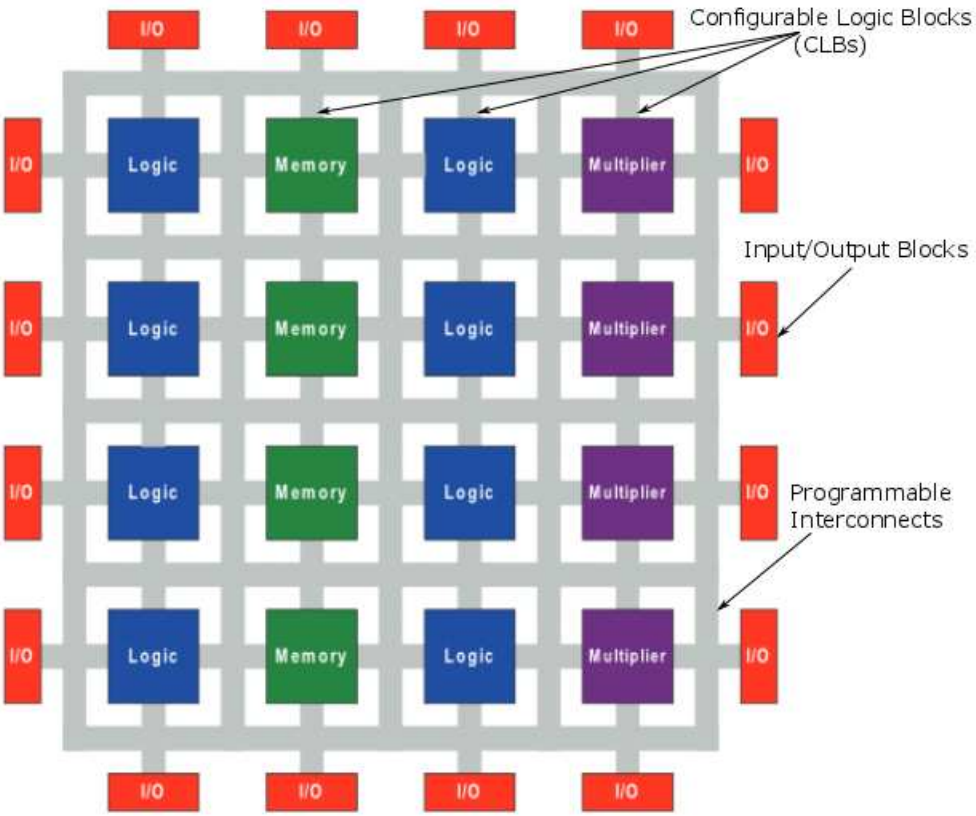

FIGURE 2.1. Basic FPGA Architecture *(Figure taken from [1])*

are called as hard blocks and are integrated on chip to implement specific frequently used functions very efficiently.

2.2.3. ADAPTIVE LOGIC MODULES. Over the last few years, the growing popularity of FPGAs has lead to several architectural innovations in the FPGA industry. In this thesis, we

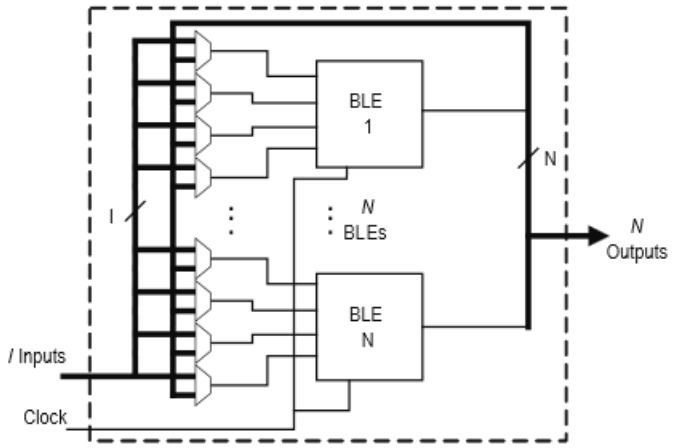

FIGURE 2.2. Cluster of N BLEs *(Figure taken from [1])*

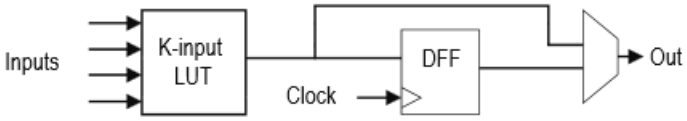

FIGURE 2.3. Basic Logic Element (BLE) *(Figure taken from [1])*

synthesized our design on Altera's Arria II GX EP2AGX65DF29I5. In the Arria II FPGA family the basic building block of logic is called Adaptive Logic Module (ALM). As shown in the Figure 2.4, each ALM contains a variety of LUT-based resources that can be divided into two combinational adaptive LUTs (ALUTs) and two registers. The ability to divide the LUT is what makes it adaptive. The entire ALM is an 8-input structure that can implement various combinations of logic functions including two 4-input logic functions, one 6-input logic function, one 5-input and one 3-input function and two 6-input functions which share the same logic function and two inputs. Apart from the ALUT based resources, each ALM contains two programmable registers, two full adders, a carry chain, a shared arithmetic chain and a register chain. Using these dedicated resources, an ALM can implement various arithmetic functions like adders, counters, comparators, accumulators etc. and shift registers efficiently. Each ALM has two sets of outputs that drive the routing resources. The ALM

outputs can be driven by either the LUT or adder or register. Another useful feature of the ALM is register packing in which the LUT or adder can drive one output while the register drives another output. This allows the use of the same ALM for register and combinational logic for unrelated functions. Thus, use of this feature improves the resource utilization of the device. The use of this feature can be enabled using the synthesizer setting in CAD tools.

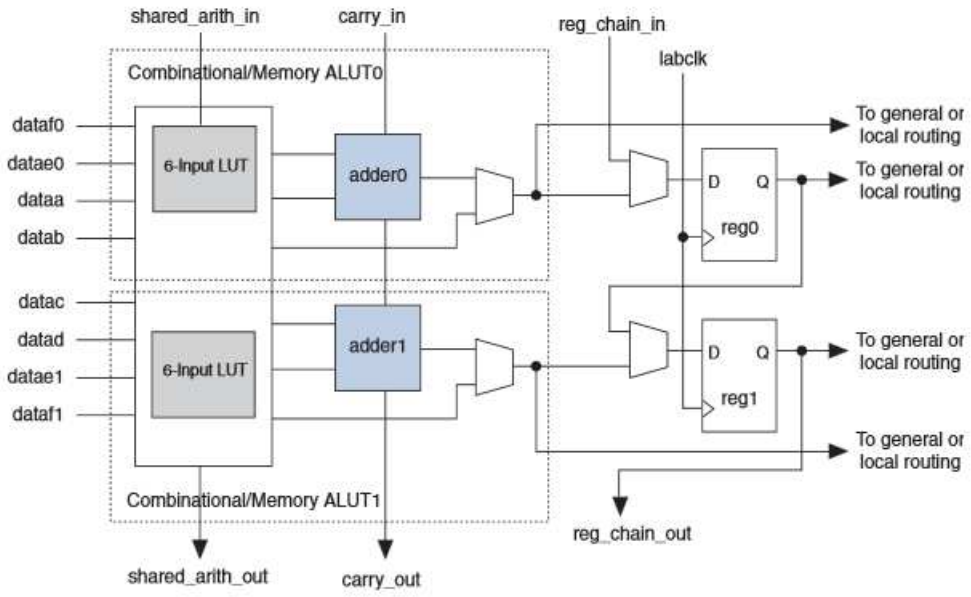

FIGURE 2.4. Block Diagram of Arria II Adaptive Logic Module (ALM) *(Figure adapted from [2])*

2.3. The FDTD Method

FDTD is extremely useful in solving electromagnetic problems. After Yee first introduced it in 1966 [13], it has been successfully applied to areas such as electromagnetic scattering, electromagnetic compatibility, antennas, microwave circuits, wave propagation, photonics and biomedical engineering because of its simple structure and accuracy. The FDTD method uses Maxwell's curl equations to solve for both electric and magnetic field in time and space. The basic idea of Yee's algorithm is to discretize both physical region and time interval of the

differential form of Maxwell's equations, on an interleaved Cartesian grid. For every point in this grid the electric and magnetic field are calculated for each time step. For the recent advances in FDTD modeling applications refer to the Tafflove and Hagness textbook [14] .

The FDTD method is compute intensive. The grid resolution in the computational domain depends on the smallest wavelength in the simulation and is typically sufficiently fine-grained. The smaller the resolution, the greater is the number of computational points in the domain. Thus some problems are often too large to be solved efficiently due to practical limits and memory and bandwidth constraints. This results in very long run times while solving problems of large sizes. Hence an approach which increases the speed of the FDTD method and reduces bandwidth requirement is necessary.

2.3.1. Reduction of Maxwell's Equations to 1D. FDTD method starts with Maxwell's Curl Equations in free space for a homogeneous medium are:

$$\frac{\partial H}{\partial t} = -\frac{1}{\mu_0}\Delta \times E \tag{2.3.1}$$

$$\frac{\partial E}{\partial t} = -\frac{1}{\epsilon_0}\Delta \times H \tag{2.3.2}$$

In equations (2.3.1) and (2.3.2) H and E represent the magnetic and electric field, μ_0 is the electric permeability of free space and ϵ_0 is the electrical permittivity of free space. The stencil equations for H and E field values are given below. In equations (2.3.3) and (2.3.4) i represents the spatial dimension and t represents the time in the computation. All values of i and t are integers as they are associated the spatial grid and computation step. c_a, c_b, d_a and d_b are the reflection coefficients which depend on the material or transmission media.

$$H_{\mathrm{i}}{}^{t} = d_a H_i^{t-1} + d_b(E_i^{t-1} - E_{i+1}^{t-1}) \tag{2.3.3}$$

$$E_{\mathrm{i}}{}^{t} = c_a E_i^{t-1} + c_b(H_i^{t} - H_{i-1}^{t}) \tag{2.3.4}$$

2.4. Tiling

Tiling [15, 16] is a technique in which the iteration space is partitioned to group/block the computations. One of the goals of performing tiling transformations is to reduce the number of off-chip memory accesses by improving data locality. This is particularly important for stencil computations. The data dependencies in the iteration space imply the precedence constraints among computations. Based on these dependencies the tile shape and the schedule for execution should be chosen in such way that there are no deadlocks i.e., there are no dependence cycles between tiles. The tile shapes are determined by the bounding hyperplanes. The concept of slicing the iteration space using hyperplanes was first introduced by Irigoin et al. in [16] . There are different tiling techniques like *overlapped tiling* [17], *split tiling* [17], *diamond tiling* [18], *hexagonal tiling* [19] and *time skewing* [20, 21] followed by rectangular tiling. Moreover, tiling can be applied hierarchically. Selection of a tiling technique amongst these is application specific and depends on which approach leads to better computation to communication ratio for the given application. Once tiling is applied, we can determine the execution wavefronts, where all the tiles in a wavefront are executed in parallel. Thus tiling exposes available parallelism of the given application. We will see later, the different tiling techniques that can be applied to 1-D FDTD and the tiling optimizations we perform to implement the FDTD on FPGA efficiently.

CHAPTER 3

Hardware Implementation of 1D FDTD

In this chapter, we first analyze the data dependencies and the computations of 1-D FDTD. Such an analysis is important as it assists in deciding the architectural features and control flow of the architecture. This analysis influences the tiling optimzations we discuss later in this chapter. Then we discuss the hardware accelerator design, which is based on the dependence analysis and tiling optimizations. We also, describe in detail the Verilog implementation of our design.

3.1. Analysis of Data Dependencies of 1D FDTD

As discussed in section 2.3, the magnetic and electric field equations for FDTD are derived from the Maxwell's curl equations, and are given as,

$$H_{\mathrm{i}}{}^{t} = d_a H_i^{t-1} + d_b(E_i^{t-1} - E_{i+1}^{t-1}) \tag{3.1.1}$$

$$E_{\mathrm{i}}{}^{t} = c_a E_i^{t-1} + c_b(H_i^{t} - H_{i-1}^{t}) \tag{3.1.2}$$

By analyzing equations (3.1.1) and (3.1.2) we can draw a Data Dependence Graph (DDG) as shown in Figure 3.1, which represents the dependence of the grid points with edges and nodes.In Figure 3.1 the squares represent the magnetic field and the circles represent the electric field. The uniform cyclic nature of the algorithm is evident from the magnetic and the electric field equations and DDG. The calculations for updating the magnetic field and electric field at each grid point are not independent of each other. Both of them need each

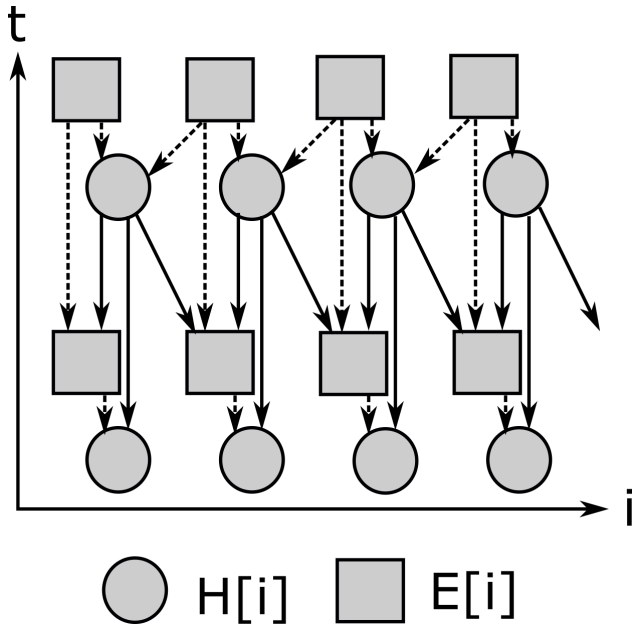

FIGURE 3.1. DDG for Magnetic and Electric field of FDTD 1D

others former result as inputs. the Electric field equation also needs the current time step result of the magnetic fields H_i^t and H_{i-1}^t.

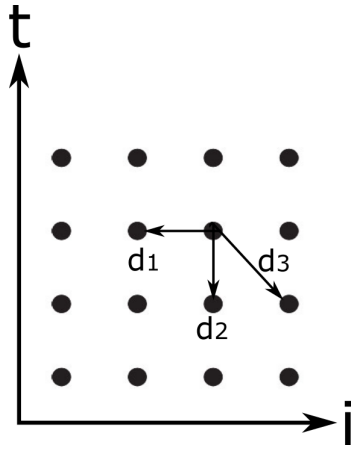

FIGURE 3.2. DDG for FDTD 1D with dependences $\vec{d_1}$=(0,1), $\vec{d_2}$=(1,0) and $\vec{d_3}$=(1,1)

For better understanding of the data dependencies, Figure 3.2 shows a simplified DDG where the E and H at each grid point are represented as a single grid point and the edges represent the three dependencies $\vec{d_1}$, $\vec{d_2}$ and $\vec{d_3}$. Since E and H are represented as single grid

point the dependence of E on the result of H at same grid point and same time step H_i^t is not shown.

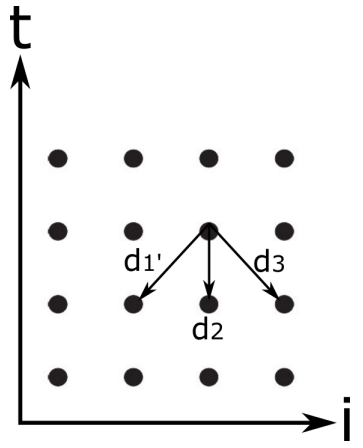

FIGURE 3.3. DDG for FDTD 1D with transformed dependence $\vec{d_{1'}}$=(-1,-1)

Of the the three dependencies shown in Figure 3.2, dependence *d1* indicates the dependence of $E_i{}^t$ on H_{i-1}^t. However H_{i-1}^t can be independently computed as:

$$H_{i-1}^t = d_a H_{i-1}^{t-1} + d_b(E_{i-1}^{t-1} - E_i^{t-1}) \tag{3.1.3}$$

In the above equation (3.1.3), the two operands different from equations (3.1.1) and (3.1.2) are H_{i-1}^{t-1} and E_{i-1}^{t-1}. This indicates that dependence d_1 can be transformed into dependence $d_{1'}$ as shown in Figure 3.3 and H_{i-1}^t can be computed independently. This transformation is important for fully exploiting the available parallelism and for efficient partitioning of the iteration space which is explained later in section 3.2.

3.2. TILING OPTIMIZATIONS

The naive way of implementing this stencil would be to traverse the iteration space sequentially and execute the grid points one after the other. However such an implementation

is not very efficient due to lack of data locality and parallelism. It leads to large volume of data transfers from the main memory and low computation-to-communication ratio. The FDTD algorithm offers abundant parallelism. A key transformation which we use here is the iteration space ***tiling***. Tiling transformations enable parallelization and data-locality optimization. We propose two levels of tiling for implementing FDTD algorithm on the hardware accelerator. The outer level corresponds to multiple passes while the inner level corresponds to tiling optimizations within a tile pass. Multi-pass tiling has been recently proposed for CPUs and GPUs while implementing stencil computations, because it reduces off-chip memory accesses and thus allows considerable energy savings [22, 23]. We implement this strategy for our design of hardware accelerator. The hyperplane which defines the pass boundaries should be such that all the data dependences should always lie on one side or along the hyperplane. Figure 3.4 and Figure 3.5 show the two valid hyperplanes for 1-D FDTD $\vec{h_1}$=(1,1) and $\vec{h_2}$=(0,1).

Passes can be mapped on the hardware accelerator either be along the hyperplane $\vec{h_1}$ or along hyperplane $\vec{h_2}$ in the horizontal direction. In order to achieve load balance between the PEs, it is essential to have the inter-PE boundary parallel to the pass boundary hyperplane. In Figure 3.4 and Figure 3.5, the red lines represent the PE boundaries. A pass of d-dimensional iteration space is nothing but a tiling strategy that uses only d-1 hyperplanes. So along one direction, the iteration space is not tiled, and this leads to tubes that can be arbitrarily long. In our tiling scheme we choose parallelogram tiling for inner level too. We justify this choice later on in this section. Let us first address the problem of choosing a pass boundary. Tile pass along the hyperplane $\vec{h_1}$ means that each PE executes a parallelogram with width x and height T in one pass. While tile pass along hyperplane $\vec{h_2}$ means that each PE executes a horizontal strip with height x and width N in one pass. Within a pass, each

PE executes parallelogram shaped tiles sequentially. Selecting the direction of traversal is primarily governed by maximization of data re-use. It is beneficial to have larger pass width while executing the tiles along $\vec{h_1}$ and larger pass height while executing the tiles along $\vec{h_2}$, to maximize data re-use. An important point to note here is that the other dimension, ie., tile height in Figure 3.4 and tile width in Figure 3.5, is made small to shorten the synchronization interval between the PEs. Looking at the inter-tile data dependences we can say that, both these tiling schemes inhibit concurrent execution start of all the PEs. However, we propose a key transformation to eliminate this limitation for the tiling scheme with tile passes along $\vec{h_1}$.

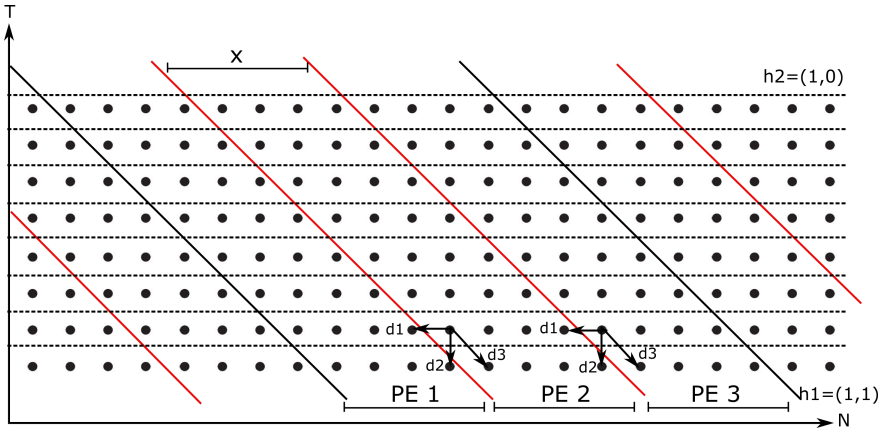

FIGURE 3.4. Parallelogram tile passes along $\vec{h_1}$

The key idea we propose while implementing the tiling scheme with tile passes along $\vec{h_1}$ is, to retain the dependence d_1 across the pass boundaries but to use, instead, the transformed dependence $d_{1'}$ across PE boundaries within a pass. This can be seen in Figure 3.6 which shows one of the steady state passes. The advantages of this scheme is twofold:

(1) **Near-concurrent start**: For updating the Electric field E_i^t we need the values of H_{i-1}^t. For the points on the left boundary of the tile, this value lies across the PE boundary. By making use of the dependence $d_{1'}$ we can remove this dependence

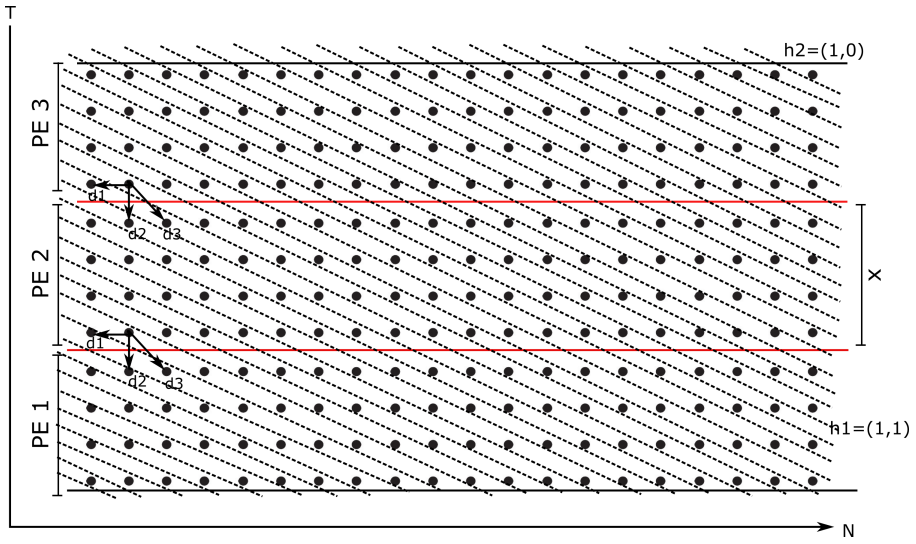

FIGURE 3.5. Horizontal tile passes along $\vec{h_2}$

between the PEs. Each PE redundantly calculates one column of H needed for the electric field values on the left boundary. This allows concurrent start of all the PEs. However, the inputs E_i^{t-1}, E_{i+1}^{t-1} and H_{i-1}^{t-1} needed for calculating H_{i-1}^t are taken from the previous PE. This introduces a small latency between the PEs. This latency is no more than the time to compute a single iteration, which is d cycles, d is the pipeline depth of computation data-path and is independent of the tile width x. Hence we say that the PEs have a *"near-concurrent start"*.

(2) **Reduced I/O**: Across the pass boundary we retain the dependence d_1. The values of H_{i-1}^t needed for the left boundary of the first PE have been calculated in earlier passes and are taken from main memory. This allows us to fetch only one column of data (E and H) from the main memory. The other alternative of retaining dependence $d_{1'}$ across passes would have required two columns of data to be transferred (E and H). To avoid this, the first PE takes the previously calculated values from the main memory instead of computing H_{i-1}^t redundantly. However to maintain uniformity in timing characteristics and control logic of PEs, the first PE

does a dummy calculation to account for the time spent by other PEs in doing the redundant calculation.

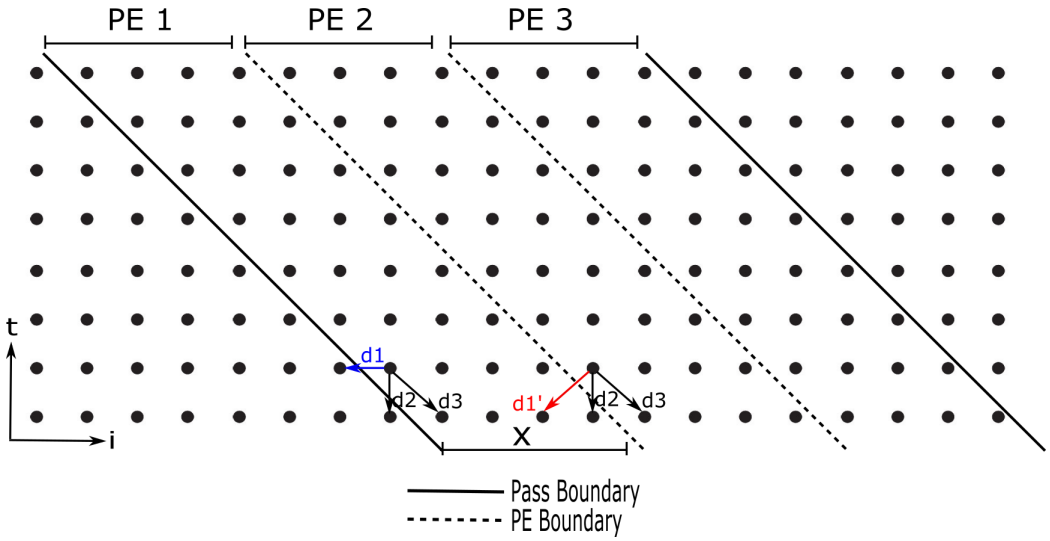

FIGURE 3.6. Steady state pass with $\vec{d_1}$=(0,1) across pass boundaries and $\vec{d_{1'}}$=(-1,-1) across PE boundaries within a pass

We are not sure whether such transformations are possible for the tiling scheme with tile passes along $\vec{h_2}$. Hence we chose to perform the tile passes along $\vec{h_1} = (1, 1)$ direction. Based on the DDG shown in Figure 3.3, many alternative tiling strategies are possible for the 1-D FDTD namely overlapped tiling, split tiling, triangular tiling, hexagonal tiling and diamond tiling. Let us look at the tiling scheme with diamond tiling as the inner level of tiling. For diamond tiling the valid hyperplane for outer level tile passes is $\vec{h_1}$. The overall tiling scheme is shown in Figure 3.7. Within the diamond tile, the iteration points are executed row-wise. The advantage of this scheme is that the PEs can have a fully concurrent start. However the number of iteration points executed by a PE, within a diamond, varies at each time step, which leads to a complex control logic inside each PE. Moreover the data-dependences are such each PE needs to store $2c$ values of E and H on on-chip buffers because they are needed by the next PE for execution of next diamond. This increases the on-chip storage

requirements. It is crucial for the FPGA-based hardware accelerators to have a simple control logic and minimal storage requirements for efficient use of the available hardware resources. We believe that similar arguments can be made for all tile shaped other than parallelogram. Hence in this this thesis we implement the tiling scheme shown in Figure 3.6 with parallelogram tile passes and parallelogram inner tiles with tile height equal to one timestep.

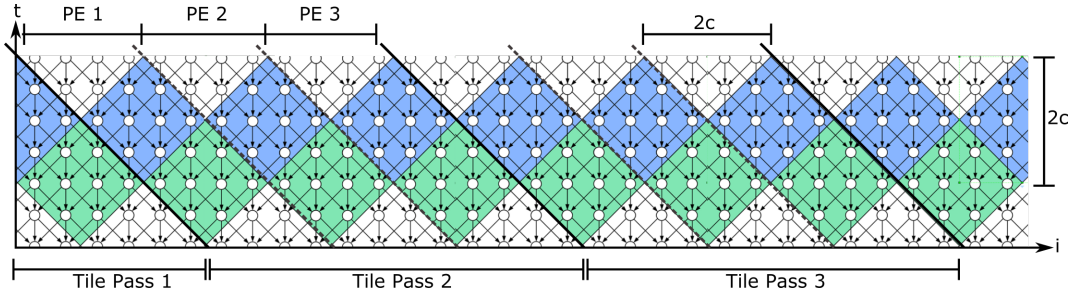

FIGURE 3.7. Diamond Tiling with Parallelogram-shaped Tile Passes

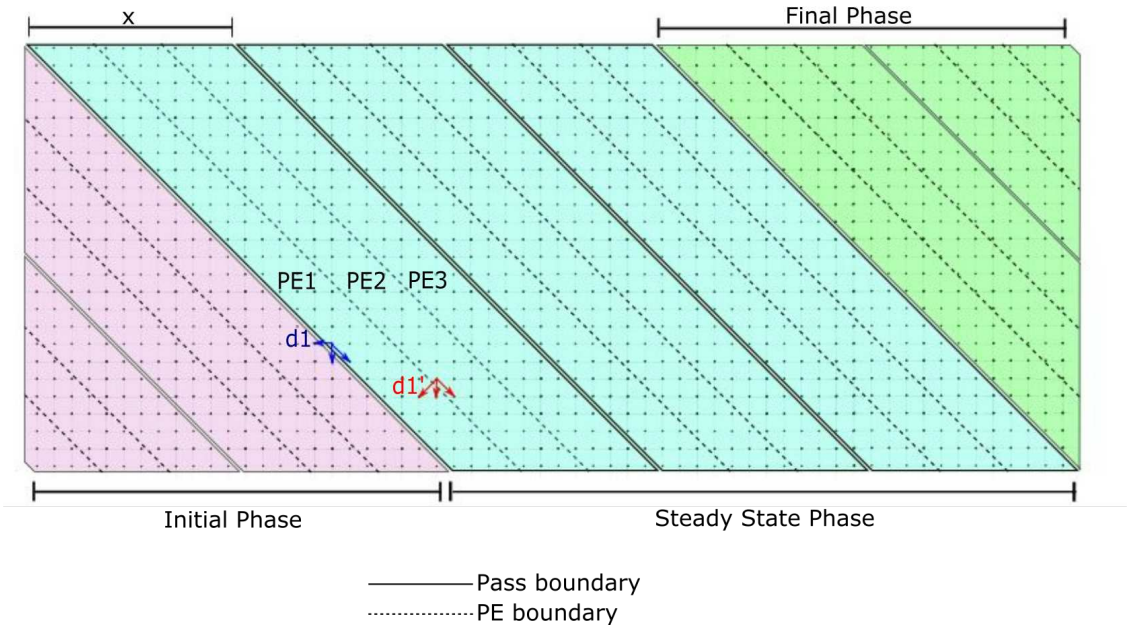

FIGURE 3.8. Tiled iteration space with Initial phase, Steady state and Final phase passes

This tiling scheme divides the iteration space into three phases as showm in Figure 3.8.

(1) Initial Phase: All the purple tiles constitute the initial phase. The passes in this phase are unbalanced causing some processing elements to be idle for certain period of time. All PEs have near concurrent start but the finish time of PE i is a linear function of i.

(2) Steady State: The balanced blue passes represent the steady state passes. In this phase the PEs can have near-concurrent start and finish with no load imbalance.

(3) Final Phase: Similar to initial phase passes, all the green passes cause idling of PEs for certain period of time. In this phase the PEs have linear start times and concurrent finish times.

3.3. Hardware Design

This section describes the design of the hardware accelerator based on the tiling optimizations mentioned in section 3.2. In the context parallel implementation of FDTD based on the data dependence analysis, tile shape selection and the computations, our hardware design is made up of three main components:

(1) Compute engine and control unit within each PE

(2) Memory Hierarchy

(3) Communication between PEs.

We implemented our hardware design in Altera's Quartus II using Verilog. In a tile pass described in section 3.2 each PE computes a parallelogram shaped tile of height equal to one time-step from left to right and then moves on to the next tile of the tile pass. We first did the design entry for one PE and then instantiated the PE module in a top level Verilog module multiple times. After doing the preliminary functional simulation, we synthesized and placed

and routed our design on Altera's Arria II GX device. We then performed post-synthesis functional verification and timing checks.

The number of points executed by each PE in one pass i.e., the tile width in the space dimension, and the total number of PEs determine the number of passes needed for execution of complete iteration space. In chapter 4 we discuss the selection of the optimal tile width and optimal number of PEs based on an analytical cost model and resource utilization profiling.

3.3.1. Verilog Design. We will now discuss in detail the Verilog implementation of the FDTD hardware accelerator. In chapter 2 we introduced the FDTD algorithm. For a quick review, the algorithm starts by loading input data for electric and magnetic field and values of reflection coefficients from main memory into the storage elements in PEs. Each PE then starts the calculation which involves updating E and H fields at each grid point in a tile for which the PE is responsible at every time step till the E and H field values reach the convergence condition, after which the iterations are terminated.

The notations used in this section are mentioned in Table 3.3.1.

Table 3.1. Notations

parameter	description
N	Total number of grid points in i-th dimension
T	Total number of time steps needed for convergence
p	Number of processing elements (PE)
x	Grid point executed per PE per pass
s	Total number of passes
f	FPGA Frequency
d	Depth of pipeline
η	Main memory bandwidth

As described in section 3.2 each PE is allocated parallelogram tiles . The tiles executed by all the PEs in parallel constitute one pass.

The two key optimizations mentioned in section 3.2 viz. use of d_1, dependence across PE boundaries and use of the d_1 dependence across pass boundaries, in the interest of reducing off-chip memory transfers and allowing near-concurrent start, results in:

(1) Loading of T values of H from main memory for the first PE

(2) Redundant calculation of the magnetic field of T points outside the left boundary of a tile

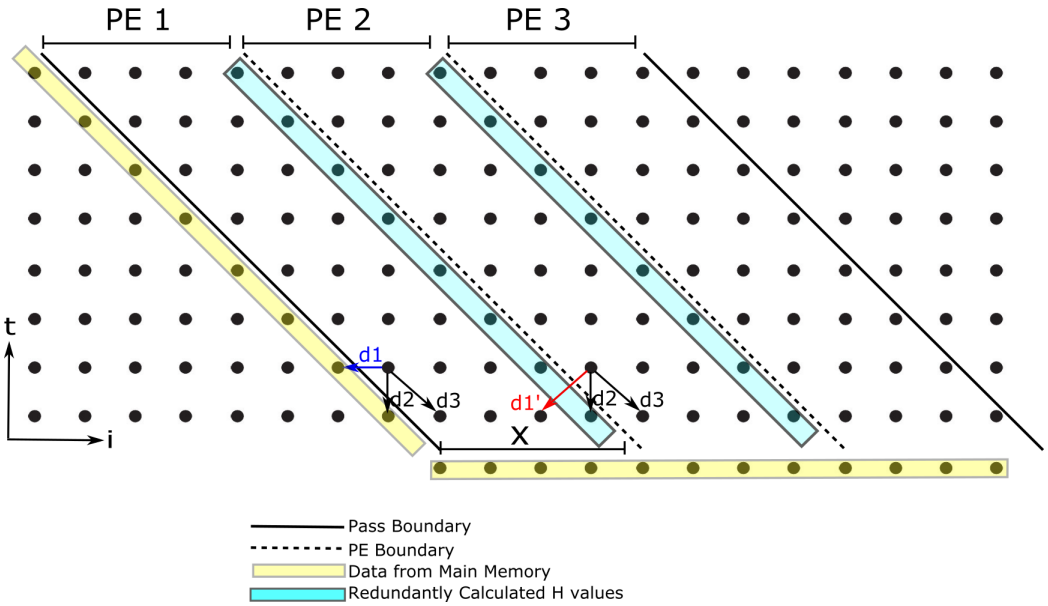

FIGURE 3.9. Memory transfers and redundant computations in a steady state pass

In Figure 3.9 the points within the yellow rectangles represent the data to be loaded from main memory and the ones within blue rectangles represent the redundant computations of PEs. Execution within a tile is row-wise. Each PE starts execution by updating the H field values in a row where the first set of inputs fed in the pipelined datapath correspond to the redundantly computed H. For computing E field values, the H values from the current time step are needed. Hence as soon as the first H field value of a row is available the PE starts updating the E field values of that row.

The datapath of each PE consists of an arithmetic logic unit (ALU) for simultaneous execution of H and E field computations. We use only integer arithmetic in our Verilog implementation. Verilog code snippet for the computation of H and E field equations (3.1.1) and (3.1.2) is shown below.

```
// H Field Equation
    sub1 = a[0] - a[1];
    hp0 = da*a[2];
    hp1 = db*r_sub1;
    x[2] = r_hp0 + r_hp1;

    // E Field Equation
    sub2= b[0] - b[1];
    ep0 = ca*b[2];
    ep1 = cb*r_sub2;
    y[2] = r_ep0 + r_ep1;
```

In the arithmetic units, inputs for the H and E fields for all the iteration points of a row are fed one after the other to ensure that the pipeline is full all the time. This essentially means that for an efficient hardware implementation the value of x, which represent the independent computations which can be fed in the pipeline, should be equal to or greater than the pipeline depth of the data-path. Apart from the data-path, the other important unit inside a PE is the control unit. By doing a cycle by cycle analysis we designed a control logic for each PE to feed in the correct input values at the right cycles into the pipeline registers $a[0]$, $a[1]$, $a[2]$, $b[0]$, $b[1]$, $b[2]$ in the above code listing. The Verilog design for this control path is shown below. While executing a row of points, the pipeline registers are fed inputs from either the E and H registers within a PE or from the main memory for the first PE and from the previous PE for remaining PEs. The output values of the H and E field of the current row replace the previous values in the registers E and H. This reduces the storage needed inside each PE.

```
//Control logic for H field values
```

```
case (index)
        0:begin
                a[0] <= (count1<num_h)?16'b0000000001011010:ein1;
                a[1] <= (count1<num_h)?16'b0000000001010101:ein2;
                a[2] <= (count1<num_h)?16'b0000000000000110:hin1;
        end
        1:begin
                a[0] <= (count1<num_h)?16'b0000000001010101:ein2;
                a[1] <= E[index-1];
                a[2] <= H[index-1];
        end
        2,3:begin
                a[0] <= E[index-2];
                a[1] <= E[index-1];
                a[2] <= H[index-1];
        end
        default: begin
                a[0] <= E[index-2];
                a[1] <= E[index-1];
                H[index-4] <= x[3];
                a[2] <= H[index-1];
                E[index + (z-8)]=(count1<num_h)?E[index]:y[3];
        end
        num_e: begin
                a[0] <= E[index-2];
                a[1] <= E[index-1];
                H[index-4] <= x[3];
                a[2] <= H[index-1];
        end
        endcase

//Control logic for E field values
case (index_e)
        0: begin
                b[0] <= x[3];  //h1
                b[2] <= (count1<num_e)?16'b0000000001010101:ein2;
                b[1] <= H[index_e];//h0
        end
        1,2,3: begin
                b[0] <= x[3];//h2
                b[1] <= H[index_e];//h1
                b[2] <=E[index_e-1];
        end
        default: begin
                b[0] <= x[3];
                b[1] <= H[index_e];
                b[2] <= E[index_e-1];
                H[index_e+1] <= x[3];
                E[index_e-4] <= y[3];
        end
```

```
num_e: begin
end
endcase
```

It can be seen from the case statement for H fields, that the inputs for the first H field value in a row, which corresponds to the redundant computation done by a PE, are ein1, ein2 and hin1 which are the values communicated by the previous PE. These dependences $d1'$ and *d2* are shown in Figure 3.9. This inter-PE dependence is the reason why the PEs have a near-concurrent start and not fully concurrent start. We call the time interval between the start of execution of successive PEs as the ***inter-PE latency***. The inter-PE latency for our design is equal to the pipeline depth of the datapath. This is the time required to get the first output value for the E field in a row. Delaying successive PEs by this interval ensures that once a PE starts executing, there are no pipeline stalls due to data dependency on the results of previous PE. Figure 3.10 is the block level representation of the synthesized hardware of each PE.

In a top level Verilog module for the hardware accelerator, multiple instances of PEs were instantiated using a *generate* block loop. The generate loops are useful for generating multiple instances of modules, user defined primitives, gate primitives etc. A new data type variable *genvar* is used inside such a generate loop. It differs from the other variables in that it can be assigned values and can be changed at compilation time. The code listing for the generate block, in which w PEs are instantiated is shown below. The PE instantiation is done in such a way that, w PEs get synthesized along with the input and output interconnections between them. Figure 3.11 shows three PEs synthesized using Quartus II.

```
genvar n;
generate for (n=2; n<=w; n=n+1) begin : PEs
localparam y = 8'b00001000*(n-1) + 1;
PE1 u (.clk(clk), .rst(rst), .ein1(path1[n-1]), .ein2(path2[n-1]),
.hin1(path3[n-1]), .eo1(path1[n]), .eo2(path2[n]), .ho1(path3[n]),
```

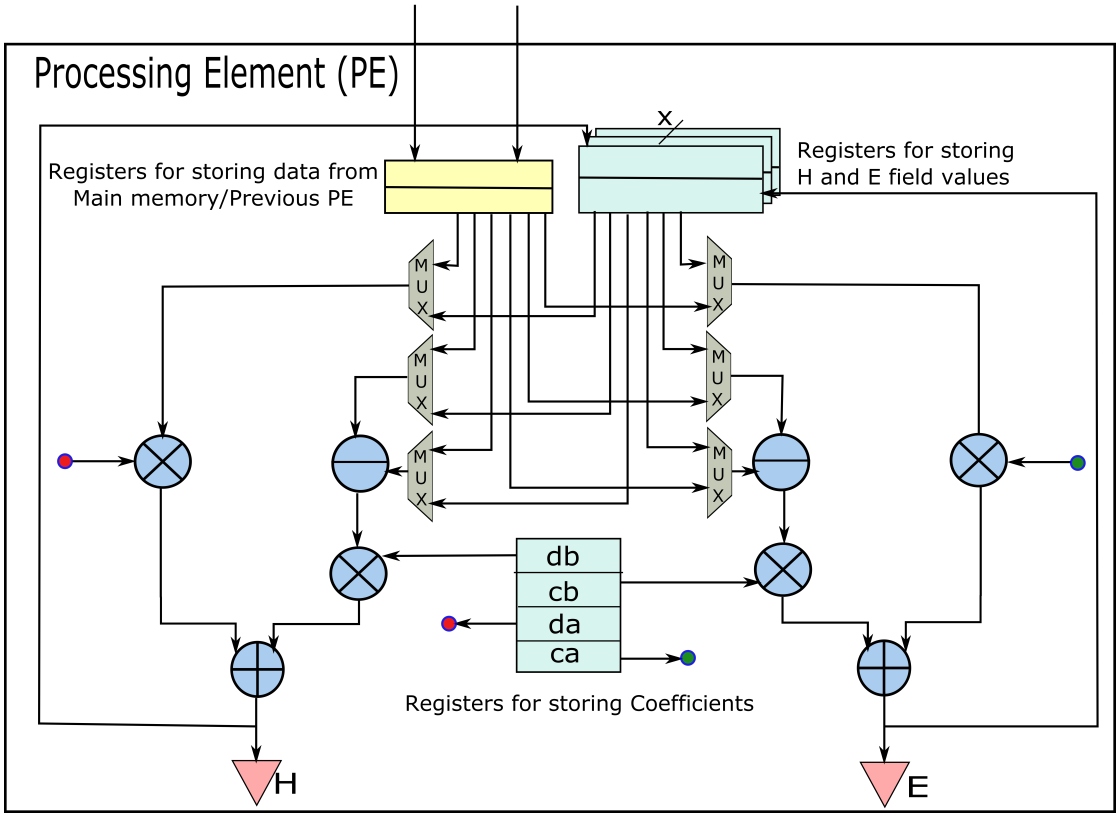

FIGURE 3.10. Block level representation of hardware synthesized for one Processing Element

```
.ho2(path4[n]), .counter(counter), .start(y), .h(h), .e(e));
end
endgenerate
```

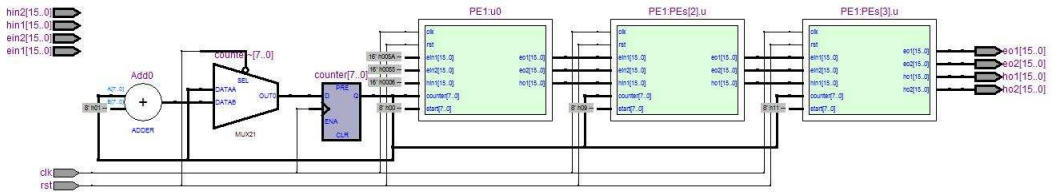

FIGURE 3.11. Interconnected Processing Elements synthesized in Quartus II

3.3.2. DESIGN VERIFICATION. We now describe the post-synthesis simulation and verification of the Verilog design. We first performed functional verification to verify the logical operation and fundamental correctness of the design. We then performed timing simulation

to check the timing of the signals in synthesized circuit and analyze maximum frequency of operation of the circuit.

Functional verification was performed by writing a Verilog testbench and simulating the design in Altera's ModelSim software. Using a Perl script the testbench was simulated for different values of x and p, and the outputs of the simulations were stored in a file. The outputs were checked against the outputs generated using a C program generated from the equational specification of the FDTD algorithm in the Alpha language. Both Verilog and and the C code were given the same set of inputs and the corresponding outputs where matched. Identical outputs indicate the functional correctness of the Verilog code.

We did timing verification of the synthesized design using the TimeQuest Analyzer in Quartus II. We performed cycle by cycle verification of the circuit's behavior. This helped in architecture verification and performance analysis. TimeQuest also performs Static Timing Analysis of the circuit, which helps in determining the maximum frequency of operation of the circuit. Table 3.3.2 lists the maximum frequency of operation along with resource utilization, for circuits with different values of x and p on Altera's Arria II GX. From the table we can see that x and p play a major role in determining the resource utilization and performance of the hardware accelerator.

In the Chapter 4 we formulate an optimization problem to find out optimal values of x and p in order to achieve high performance and efficient resource utilization.

TABLE 3.2. Summary of Resource Utilization and Maximum Frequency

	$x = 8, p = 32$	$x = 8, p = 52$	$x = 16, p = 27$	$x = 32, p = 15$
Logic Utilization	83%	98%	99%	94%
Combinational ALUTs	26,000 (51%)	33,423 (66%)	34,381 (68%)	30,298 (60%)
Dedicated Logic Registers	15,880 (31%)	25,192 (50%)	20,312 (40%)	16,440 (32%)
DSP Block Elements	128 (41%)	208 (67%)	112 (36%)	52 (17%)
Maximum Frequency	92 MHz	80 MHz	75 MHz	89 MHz

CHAPTER 4

Selection of Optimal Design Parameters

Advances in FPGA technology lead to increased logic resources available, resulting in a huge design space for exploration. The performance of different designs with the same logic utilization is not the same. Therefore, it is not trivial to find the optimal design solution, especially when there are limited computation resources. In addition, optimization techniques like tiling transformations further expand the design space to be explored. Consequently, it is more difficult to find the optimal solution in the design space. Hence, a systematic method is required to explore the design space of FPGA-based accelerators.

In chapter 3 we introduced the two parameters x and p which represent the width of a tile in the space dimension (i) and the number of processing elements, respectively. These design paramters x and p govern the run-time of the algorithm on the hardware accelerator. We now address the problem of choosing these two optimally. The goal is to determine the provably "good" values of x and p, in the sense that they minimize the execution time. Two crucial considerations while choosing the values of x and p are the problem sizes and resource utilization. The tile size will determine the amount of FPGA resources needed to synthesize one PE. Hence the optimal values for x and p are not independent of each other. In this chapter we first develop performance and area models, and then formulate and solve analytically, an optimization problem with the objective of minimizing the total execution time for given problem size with the given resource constraints. The total execution time is a function of x and p. Hence, the analytical solution to this problem allows us to choose the tile size and number of processing elements in an optimal manner.

After solving the optimization problem, we will also investigate how the optimal values of x and p, are affected by the problem size parameters N and T. N is the total number of grid points in the space dimension and T is the total number of timesteps. To a first, intuitive approximation, we can say that, since we have p PEs and each one delivers the result of one iteration point point per clock cycle in the steady state, and assuming all PEs are always active, the total execution time is equal to $\frac{NT}{p}$. Hence maximizing the number of PEs leads to higher performance. The results of optimization problem for FDTD algorithm with conservative problem sizes agree with this insight. However, we will see that, under certain conditions this is not optimal.

4.1. Performance Model

In this section we develop performance model for our hardware accelerator in terms of total execution time. As described in section 3.2 , the entire iteration space can be divided in three phases- initial phases, steady state phase and final phase. The total execution time is the sum of the time required for execution of tile passes in each of these phases. So we develop the performance model by initially modeling the execution time for the three phases separately. Table 4.1 shows the various parameters used in this section.

4.1.1. Execution Time For Steady State Passes. As explained in section 3.2 , in one steady state pass, p processing elements execute p tiles of size x in the space dimension i and T in time dimension. The total time taken for execution of one pass depends on the following four components:

(1) Time taken for computation ($t_{\text{computation}}$)

(2) Time taken for communication ($t_{\text{communication}}$)

(3) Time for which each PE is idle due to inter-PE dependence ($t_{\text{inter\_PE}}$)

TABLE 4.1. Model Parameters

paramter	description
N	Total number of grid points in the i dimension
T	Total number of timesteps
p	Number of processing elements (PE)
x	Grid points executed per PE per pass
s	Total number of passes
f	Frequency (MHz)
d	Pipeline depth of the datapath
η	Main memory bandwidth(bytes/seconds)
$t_{\text{computation}}$	Time taken for performing computations (seconds)
$t_{\text{communication}}$	Time taken for off-chip memory transfers (seconds)
$t_{\text{inter\_PE}}$	Time for which each PE is idle due to inter-PE dependence (seconds)
$t_{\text{s\_pass}}$	Time taken for execution of one steady state pass (seconds)
$t_{\text{steady\_state}}$	Total execution time of all steady state passes (seconds)
t_{corner}	Total execution time of all corner passes (seconds)
t_{total}	Total execution time of the algorithm on the hardware accelerator (seconds)

In our design one iteration point is computed per cycle. Hence the time taken for computation of one pass can be given as:

$$t_{\text{computation}} = \frac{\text{Total number of grid points in one pass}}{pf}$$

In chapter 3 we described how each processing element performs some redundant computations to allow the "near-concurrent" start for PEs. Each PE computes T values of magnetic field H_{i-1}^t redundantly because all the electric fields E_i^t near the left boundary of a PE need this value from across the PE boundary. Thus we calculate $(x+1)$ magnetic field values and x electric field values for one row in a tile. To satisfy the dependence between electric and magnetic field for the next row, no input is fed in the electric field datapath for one cycle and the last value of electric field is held for two cycles. So the redundant computation of one H value per row plus one cycle stall per row in the electric field datapath essentially means that each PE calculates $(x+1)$ iteration points per row. Hence $t_{\text{computation}}$

can be written as:

$$t_{\text{computation}} = \frac{(x+1)Tp}{pf} = \frac{(x+1)T}{f} \tag{4.1.1}$$

For each tile pass, $(x+T)$ values need to written to the off-chip memory and the same amount of data needs to be read from the off-chip memory. The time taken for reading and writing $(x+T)$ single precision values of E and H to the main memory is given by:

$$t_{\text{communication}} = \frac{2 \times 2 \times 4 \times (\mathrm{x} + \mathrm{T})}{\eta} \tag{4.1.2}$$

The next component to be considered for calculating the execution time of a pass is the inter-PE latency since it determines the idle time of PEs. The redundant computation of the magnetic field value across the PE boundary needs input E and H values calculated by the previous PE. Because of this inter-PE dependence the PEs have a "near-concurrent" start. Inter-PE latency is the time interval between the start of execution of the successive PEs. As mentioned in section 3.3 for our design the inter-PE latency is equal to the pipeline depth of electric and magnetic datapath. This ensures that once a PE starts executing, there are no pipeline stalls due to inter-PE dependency.

$$t_{\text{inter\_PE}} = \frac{d}{f} \tag{4.1.3}$$

Except for the first PE, all the other PEs start their execution d cycles after its previous PE. Thus the total number of idle cycles of PEs is a linear function of its id (identity number). The $j-th$ PE will starts its execution after dj cycles, where j ranges from 0 to $p-1$. The

idle cycles of the last PE account for the idle cycles of all other PEs. The last PE will start d cycles after the second last PE. Hence the total idle cycles due to inter-PE latency is given as:

$$t_{\text{inter\_PE}}(p-1) = \frac{d(p-1)}{f} \tag{4.1.4}$$

The total execution time for a steady state pass is the sum of the three components described above. However the computation and communication of PEs can be overlapped and hence the greater of the two needs to be considered while finding out the execution time.

$$t_{\text{s\_pass}} = \max(t_{\text{computation}}, t_{\text{communication}}) + t_{\text{inter\_PE}}(p-1) \tag{4.1.5}$$

It is important to note that not all of the data $(x+T)$ is needed from off-chip at the beginning of a pass. The data can be prefetched in smaller chunks and stored in the on-chip buffers. For our design, one value needs to be fetched and written into the off-chip memory every x cycles. The bandwidth required is highest when x is minimum. The minimum value of x in our design is equal to the pipeline datapath d. Thus the required bandwidth can be given as 8 bytes in d cycles. Looking at the typical pipeline depths and frequencies of the most of the modern FPGAs, we can say that this bandwidth requirement is not hard to satisfy. Hence $t_{\text{computation}}$ dominates $t_{\text{communication}}$ and the total time for execution of one steady state pass is given as:

$$t_{\text{s\_pass}} = t_{\text{computation}} + t_{\text{inter\_PE}}(p-1) = \frac{(x+1)T}{f} + \frac{d(p-1)}{f} \tag{4.1.6}$$

The total number of steady state passes is $\frac{N-T}{xp}$ (assuming $N \geq T$. Hence the total time taken for executing all the passes in steady state phase is:

$$t_{\text{steady\_state}} = \left(\frac{(x+1)T}{f} + \frac{d(p-1)}{f}\right)\frac{N-T}{xp} \tag{4.1.7}$$

4.1.2. Execution Time For Corner Passes. The corner passes are the ones in the initial and final phases of the computation. These passes are not of equal length and furthermore, the number of iteration points executed by a PE in any such pass is also different from one PE to another. This leads to load imbalance and causes idling of PEs. Similar to the steady state passes, the communication required for the corner passes can be overlapped with the computation by prefetching smaller chunks of data and storing them in on-chip buffer. Hence the two main components of execution time for corner passes are $t_{\text{computation}}$ and $t_{\text{inter\_PE}}$.

The computation time of the corner passes will depend on the size of the tile passes. Each PE will still be executing $(x+1)$ grid points in a row. However the number of rows are not the same for all the passes. The maximum number of rows in a pass, is some multiple of xp depending on the pass number. For one corner phase, there are $\frac{T}{xp}$ passes. The maximum number of rows in each pass can be given by multiplying *(xp)* with the corresponding pass number as given in equation (4.1.8). An important point to note here is that, though only one PE executes the parallelgram with maximum number of rows in a pass, in equation

(4.1.8) we consider that all the PEs execute the same number of rows so as to account for the idle cycles of the PEs. The the time taken for computation of all the passes in the initial and final phase is thus given as:

$$t_{\text{computation}} = \frac{2}{f}\sum_{q=1}^{\frac{T}{xp}}(x+1)(xpq) \tag{4.1.8}$$

Solving the above summation we get:

$$t_{\text{computation}} = \frac{1}{f}\left((x+1)T\right)\left(\frac{T}{xp}+1\right) \tag{4.1.9}$$

The total number of corner passes in initial and final phase are $2 \times \frac{T}{x \times p}$. Similar to steady state passes, the corner passes too have the inter-PE latency, given by *d(id-1)*. The total idle time of PEs in one pass is given by *d(p-1)*. The total idle time for corner passes can be written as:

$$2t_{\text{inter\_PE}}(p-1)\frac{T}{xp} = \frac{2}{f}d(p-1)\frac{T}{xp} \tag{4.1.10}$$

The total time taken for execution of all the passes in the initial and final phase is equal to the sum of the computation time and time for which the PEs are idle.

$$t_{\text{corner}} = \frac{1}{f}\left((x+1)T\right)\left(\frac{T}{xp}+1\right) + \frac{2}{f}d(p-1)\frac{T}{xp} \tag{4.1.11}$$

$$= \left((x+1)T + 2d(p-1)\right)\frac{T}{xpf} + (x+1)T \tag{4.1.12}$$

The execution time of the entire algorithm on the hardware accelerator is the addition of time taken to execute all the passes in the initial, final and steady state phases. Hence from Equation (4.1.7) and Equation (4.1.12) the total execution time can be written as:

$$t_{\text{total}}(x,p) = \frac{1}{f}\left(\frac{NT}{p} + \frac{NT - Nd - Td}{xp} + \frac{Nd + Td}{x} + (x+1)T + \frac{Td}{x} - \frac{Td}{xp}\right) \tag{4.1.13}$$

The execution time is a function of our design parameters x and p and we seek to minimize $t_{\text{total}}(x,p)$ subject to resource and other hardware constraints. First we notice that in equation (4.1.13) p occurs only in the denominator of terms where the numerator is guaranteed to be positive. Hence $t_{\text{total}}(x,p)$ monotonically decreases with p. Hence to minimize $t_{\text{total}}(x,p)$ with a given value of x, p should be maximized. However p cannot be increased arbitrarily since the FPGA resources are constrained. Therefore we develop a cost model for resource utilization.

4.2. Area Model

The motivation behind developing an analytical area model is to relate our design parameters to FPGA resource utilization. The utilized chip area of the FPGA is calculated in terms of LUTs (Look-up Table) used. Our objective is to improve the performance of our hardware accelerator by minimizing $t_{\text{total}}(x,p)$. In order to accomplish this goal, we need to

find out the optimal values of design parameters like x and p. The area model will guide us in this task, by determining the constraints on x and p with respect to the target FPGA platform.

The chip utilization in terms of LUTs can be further subdivided as the number of LUTs utilized to implement combinational logic and the number of LUTs used as registers. In our design, the former is mainly the function of number of processing elements synthesized and nearly independent of the tile size whereas the utilization of LUTs for synthesizing storage elements is a function of both the number of processing elements and the tile size. On these grounds, we can relate x and p to the logic utilization, as given in equation (4.2.1) , where $A(x, p)$ is the resource utilization in terms of LUTs, α and β are constants interrelating combinational LUTs with p and storage LUTs with *(xp)* respectively.

$$A(x, p) = \alpha p + \beta(xp) \tag{4.2.1}$$

The Verilog code was synthesized was different values of x and p on Altera's Arria II GX EP2AGX65DF29I5, and then the resource utilization for each case was obtained from the post-synthesis report from the tool. Table 4.2 gives ALUTs (Adaptive Look-up Table) utilized by some of the sets of x and p which utilize the resources of Arria II GX EP2AGX65DF29I5 completely. For finding out the values of α and β, we did curve fitting with nonlinear regression method on the empirical data using DataFit [24] to find the "best-fit" values of α and β. For the selected FPGA device we got α and β to be 222 and 94 respectively. Hence equation (4.2.1) can be written as:

$$A(x,p) = 222p + 94(xp) \tag{4.2.2}$$

TABLE 4.2. Resource utilization for different pairs of x and p

[x,p]	[8,52]	[16,27]	[20,22]	[24,19]	[32,15]	[64,9]
ALUTs (out of 50600)	50094	50030	50034	49753	49049	49152
DSP Block Multipliers (out of 312)	208	108	88	76	60	36

In the next section we will see how to formulate and solve the optimization problem to find optimal x and p using the equation for total execution time and the area model.

4.3. Optimization Problem Formulation and Solution

Several values of design parameters would satisfy the minimal constraint of the resource utilization. But, significant performance gains can be achieved by searching for an optimal design among all the designs that satisfy constraints that determine feasible design space. For our design, the resource constraint decides the upper bound for design parameters like x and p. Additionally, in order to maximize utilization and throughput we need to ensure that the pipeline for arithmetic unit is always full. Consequently the lower bound on x is equal to the pipeline depth of the datapath. These constraints define the feasible region of x and p. Figure 4.1 shows the bounded solution space. As seen in the Figure 4.1 $x_{\min}$ = d, $p_{\min} = 1$ whereas $x_{\max}$ and $p_{\max}$ are determined by the hyperbola $\alpha p + \beta(xp) = L_{\max}$.

We saw in the previous section that, for any given value of x $t_{\text{total}}(x,p)$ monotonically decreases with p and $t_{\text{total}}(x,p)$ will be minimized when the hyperbolic constraint is saturated. Therefore, we can say that, in Figure 4.1 for any value of x, the optimal design point is obtained by traversing vertically upwards towards the corresponding maximum p that lies

on the hyperbola formed by $L_{\max} = \alpha p + \beta(xp)$. We can replace p as $\frac{L_{\max}}{\alpha+\beta x}$ in equation (4.1.13) and t_{total} now becomes function of only x and is given as:

$$t_{\text{total}}(x) = \frac{1}{f}\left(\left(\frac{(NTx + NT - Nd - 2Td)(\alpha + \beta x)}{L_{\max} x}\right) + \frac{Nd + 2Td}{x} + (x+1)T\right) \quad (4.3.1)$$

For the sake of simplicity, in equation (4.3.1) let us substitute $(Nd + 2Td)$ by N'. So (4.3.1) becomes,

$$t_{\text{total}}(x) = \frac{1}{f}\left(\left(\frac{(NTx + NT - N')(\alpha + \beta x)}{L_{\max} x}\right) + \frac{N'}{x} + (x+1)T\right) \quad (4.3.2)$$

We can now formulate an optimization problem with the objective of minimizing the total execution time given by equation (4.3.2), to determine the optimal values of design parameters.

$$\begin{aligned} &\text{minimize:} \quad t_{\text{total}}(x) = \frac{1}{f}\left(\left(\frac{(NTx + NT - N')(\alpha + \beta x)}{L_{\max} x}\right) + \frac{N'}{x} + (x+1)T\right) \\ &\text{subject to:} \quad d \leq x \leq \frac{L_{\max} - \alpha}{\beta} \end{aligned}$$

By simplifying the objective function we get,

$$t_{\text{total}}(x) = \frac{1}{f L_{\max}}\left((NT\beta + TL_{\max})x + \frac{\alpha(NT - N') + L_{\max}(N')}{x} + (NT\alpha + \beta(NT - N') + TL_{\max})\right) \quad (4.3.3)$$

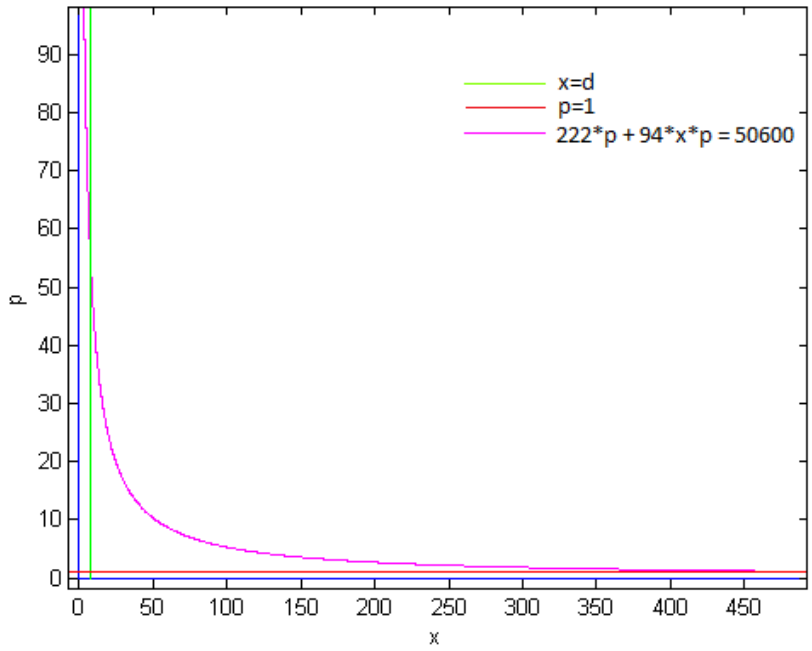

FIGURE 4.1. Feasible solution space for optimal values of x and p

We can see form (4.3.3) that t_{total} has a form of $F(x) = A/x + Bx + C$ which is a the sum of the hyperbolic term, a linear term and a constant. This is a strongly convex function, and has a unique minimum (for positive x) at $x^* = \sqrt{A/B}$. Thus for solution of our problem that optimal value of x lies at:

$$x^* = \sqrt{\frac{\alpha(NT - N') + L_{\max}(N')}{NT\beta + TL_{\max}}} = \sqrt{\frac{\alpha(NT - Nd - 2Td) + L_{\max}(Nd - 2Td)}{\beta NT + TL_{\max}}} \tag{4.3.4}$$

Using equation (4.3.4), the optimal value for p can be given as:

$$p^* = \frac{L}{\alpha + \beta\sqrt{\frac{\alpha(NT - Nd - 2Td) + L_{\max}(Nd - 2Td)}{\beta NT + TL}}} \tag{4.3.5}$$

The optimal values of x and p, which minimize the total execution time and satisfy all the three constraints, is given by,

$$\begin{array}{ll} [d, p_{\max}] & \text{if } x^* < d \\ [x_{\max}, p_{\min}] & \text{if } x^* > x_{\max} \\ [x^*, p^*] & \text{otherwise} \end{array}$$

4.4. What will be the optimal design?

We will now study how the optimal design depends on the problem size parameters N and T. We assume that N and T grow asymptotically large (otherwise it does not make sense to implement the computation on an accelerator). We also assume that $L_{\max}$, although potentially large, is not an increasing resource. It remains fixed once the target FPGA platform is chosen. Thus the area of the accelerator is bounded relative to the size of parameters we seek to implement on the accelerator.

Let us now see when is the inequality $x^* < d$ satisfied, in which case the smallest feasible value of $x = d$, is the optimal. Equation (4.3.4) can be written as given below. Here we have substituted $x^* = d$

$$\alpha(NT - Nd - 2Td) + L_{\max}(Nd - 2Td) = d^2\beta NT + TL_{\max} \tag{4.4.1}$$

From equation (4.4.1) we can say that, for large values of N and T, the two quadratic terms αNT and $(x^*)^2\beta NT$ are the dominant terms. Hence as long as $\alpha < (d)^2\beta$, the optimal value of x is going to be $x_{\min}$ i.e., d the depth of pipeline. The optimal value of p will be the corresponding p given by equation (4.3.5), which in this case is $p_{\max}$. Thus the optimization

problem solution in this case, matches with the common sense intuition of maximizing PEs for faster execution.

Most of the FDTD applications use large values of N and T to maintain high accuracy and resolution. Hence as per the above explanation, for most of the problems, the optimal design is going to be the design with minimum x and maximum p. However, when both N and T do not grow large simultaneously, this is not true. When N is considerably large compared to T, apart from the quadratic terms, the other terms in equation (4.4.1) also play a dominant role. For smaller T, it might be the case that, $d^2\beta NT + TL_{\max}$ is much smaller compared to $\alpha(NT - Nd - 2Td) + L_{\max}(Nd - 2Td)$. Therefore $x_{\min}$, will not be the optimal. The optimal x will be greater than $x_{\min}$, given by (4.3.4), and p will not be $p_{\max}$, but will be given by (4.3.5). Intuitively we can say that, when $N \gg T$ the inter-PE latency contributes significantly in the total execution time. As discussed before the inter-PE latency for a pass is given by $d(p-1)$. Thus maximizing PEs is not optimal in this case.

Formulating and solving the optimization problem not only helped us in systematic design space exploration to determine optimal parameter values but also lead to the important conclusions regarding parameter tuning. Tuning the design parameters as per the given problem, enhances the efficiency of the hardware accelerator. Observation made for larger N and smaller T are very useful in convergence testing. FDTD simulations are complicated to setup and analyze and to ensure convergence. Convergence testing is an important part of any simulation. A common way to perform the convergence test is to first quantify the level of convergence by determining the acceptable level of error, then perform simulations for fixed number of timesteps, check for convergence and decide whether to simulate further. The number of timesteps (T) here can be really less , to run the simulations quickly. This is especially useful during the development phase when saving simulation time is more critical

than precision. The resolution of grid in time direction can be made large, reducing the total number of timesteps. When T is considerably small, the optimal values and x and p can be chosen as per the equations (4.3.4) and (4.3.5), to minimize the total execution time.

One of the assumptions made while formulating the optimization problem is that the maximum frequency of operation of different designs is the same. We simulated different designs in Quartus II and performed static timing analysis using TimeQuest to obtain the maximum frequency of operation. For different problem sizes, Table 4.4 lists the values of maximum frequency and execution time for sets of x and p. Though the maximum frequency of operation is different for different designs, the designs with the optimal values of x and p given by our analytical equations, have the minimum execution time.

TABLE 4.3. Optimal design parameters for different problem sizes

N	T	x	p	Cycles	Frequency	Execution time(μ sec)	Optimal x and p given by the analytical equations
5000	1000	8	52	125176	80 MHz	1564.7	x=$x_{\min}$(8), p=$p_{\max}$(52)
		16	27	216648	75 MHz	2888.64	
		32	15	378150	89 MHz	4248.87	
5000	100	8	52	16930	80 MHz	211.62	x=$x_{\min}$(8), p=$p_{\max}$(52)
		16	27	23832	75 MHz	317.76	
		32	15	38865	89 MHz	436.68	
5000	40	8	52	9715	80 MHz	121.43	x=10, p=43
		16	27	10944	75 MHz	146.36	
		32	15	16246	89 MHz	182.53	
		10	43	9410	79 MHz	119.16	
5000	20	8	52	7307	80 MHz	91.33	x=15, p=31
		16	27	6692	75 MHz	89.23	
		32	15	8706	89 MHz	97.82	
		15	31	6350	77 MHz	82.46	

CHAPTER 5

Conclusion and Future Work

The goal of this thesis was to present a systematic design flow to generate the hardware accelerator for stencil computations like FDTD. We used a three step approach in which we first analyzed the data dependencies and logic structure of FDTD 1-D and proposed tiling transformations to fully exploit parallelism and reduce off-chip memory accesses. Next we developed a parametric design, consisting of a linear array of processing elements (PEs), each one with a combinational datapath, a set of storage registers and a control unit. We described it in Verilog and synthesized it on Altera's Arria II GX EP2AGX65DF29I5. We verified the functionality and timing of our design by performing cycle accurate simulation. We also formulated and resolved analytically, a discrete non-linear optimization problem that allows us to choose the optimal design within a significant design space. Hence, the design space "exploration" for this problem consists of a "one-shot" solution.

The case study presented in this thesis can guide the future developments in the area of hardware accelerators. It would be interesting to apply the approach to higher dimensional FDTD stencils as well as other stencil computations. The higher dimensional stencils would have more parallelism to exploit, several tiling optimization options, more complicated combinational datapath, added bandwidth requirements, and larger design space. The approach should thus be altered to take these factors into consideration. For higher dimensional FDTD, the current Verilog design can be upgraded by using BRAMs instead of LUTs for storing larger volumes data, so as to make maximum use of available FPGA resources and free some LUTs for implementing deeply pipelined, highly parallel combinational logic. Also, for 2D stencils, tiling can be done along both the dimensions yielding a prism shaped tile.

Instead on a linear array of processing elements, more options like a grid of interconnected PEs can be explored. The task of finding the optimal design would become more complicated because of increased number of design parameters and expanded design space. The optimization problem we formulated must be appropriately modified.

Bibliography

[1] I. Kuon, R. Tessier, and J. Rose, "FPGA architecture: Survey and challenges," *Foundations and Trends in Electronic Design Automation*, vol. 2, no. 2, pp. 135–253, 2008.

[2] Altera Corporation, "Arria II Device Handbook Volume I: Device Interfaces and Integration." `https://www.altera.com/en_US/pdfs/literature/hb/arria-ii-gx/arria-ii-gx_handbook.pdf`, 2014.

[3] R. N. Schneider, L. E. Turner, and M. M. Okoniewski, "Application of FPGA technology to accelerate the finite-difference time-domain (FDTD) method," in *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*, FPGA '02, (New York, NY, USA), pp. 97–105, ACM, 2002.

[4] J. P. Durbano, F. E. Ortiz, J. R. Humphrey, D. W. Prather, and M. S. Mirotznik, "Implementation of three-dimensional FPGA-based FDTD solvers: An architectural overview," in *Field-Programmable Custom Computing Machines, 2003. FCCM 2003. 11th Annual IEEE Symposium on*, pp. 269–270, IEEE, 2003.

[5] J. P. Durbano, F. E. Ortiz, J. R. Humphrey, M. S. Mirotznik, and D. W. Prather, "Hardware implementation of a three-dimensional finite-difference time-domain algorithm," *Antennas and Wireless Propagation Letters, IEEE*, vol. 2, no. 1, pp. 54–57, 2003.

[6] W. Chen, P. Kosmas, M. Leeser, and C. Rappaport, "An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm," in *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, (New York, NY, USA), pp. 213–222, ACM, 2004.

[7] H. Giefers, C. Plessl, and J. Förstner, "Accelerating finite difference time domain simulations with reconfigurable dataflow computers," *SIGARCH Comput. Archit. News*, vol. 41, pp. 65–70, June 2014.

[8] Y. Takei, H. M. Waidyasooriya, M. Hariyama, and M. Kameyama, "FPGA-oriented design of an FDTD accelerator based on overlapped tiling," in *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, p. 72, The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp), 2015.

[9] R. Wester and J. Kuper, "Deriving stencil hardware accelerators from a single higher-order function," *Communicating Process Architectures, CPA*, 2014.

[10] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," *SIGARCH Comput. Archit. News*, vol. 39, pp. 365–376, June 2011.

[11] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 26, pp. 203–215, Feb 2007.

[12] R. H. Freeman, "Configurable electrical circuit having configurable logic elements and configurable interconnects," Sept. 26 1989. US Patent 4,870,302.

[13] K. Yee, "Numerical solution of initial boundary value problems involving maxwell's equations in isotropic media," *Antennas and Propagation, IEEE Transactions on*, vol. 14, pp. 302–307, May 1966.

[14] A. Taflove and S. C. Hagness, *Computational Electrodynamics*. Artech house, 2005.

[15] M. Wolfe, "Iteration space tiling for memory hierarchies," in *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, (Philadelphia, PA, USA), pp. 357–361, Society for Industrial and Applied Mathematics, 1989.

[16] F. Irigoin and R. Triolet, "Supernode partitioning," in *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, (New York, NY, USA), pp. 319–329, ACM, 1988.

[17] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *ACM Sigplan Notices*, vol. 42, pp. 235–244, ACM, 2007.

[18] V. Bandishti, I. Pananilath, and U. Bondhugula, "Tiling stencil computations to maximize parallelism," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, p. 40, IEEE Computer Society Press, 2012.

[19] T. Grosser, A. Cohen, J. Holewinski, P. Sadayappan, and S. Verdoolaege, "Hybrid hexagonal/classical tiling for GPUs," in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, p. 66, ACM, 2014.

[20] J. McCalpin and D. Wonnacott, "Time skewing: A value-based approach to optimizing for memory locality," tech. rep., Technical Report DCS-TR-379, Department of Computer Science, Rugers University, 1999.

[21] D. Wonnacott, "Achieving scalable locality with time skewing," *International Journal of Parallel Programming*, vol. 30, no. 3, pp. 181–221, 2002.

[22] Y. Zou and S. Rajopadhye, "Automatic energy efficient parallelization of uniform dependence computations," in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ICS '15, (New York, NY, USA), pp. 373–382, ACM, 2015.

[23] W. Ranasinghe, "Reducing off-chip memory accesses of wavefront parallel programs in graphics processing units." https://dspace.library.colostate.edu/bitstream/handle/10217/88551/Ranasinghe_colostate_0053N_12785.pdf?sequence=1&isAllowed=y.

[24] O. Engineerin, "DataFit." http://www.oakdaleengr.com/.

[25] S. Krishnamoorthy, M. Baskaran, U. Bondhugula, J. Ramanujam, A. Rountev, and P. Sadayappan, "Effective automatic parallelization of stencil computations," in *ACM Sigplan Notices*, vol. 42, pp. 235–244, ACM, 2007.

[26] R. Andonov and S. Rajopadhye, "Optimal orthogonal tiling of 2-d iterations," *Journal of Parallel and Distributed computing*, vol. 45, no. 2, pp. 159–165, 1997.