Thesis

Hardware Implementation and Design Space Exploration for Wave 2D and Jacobi 2D Stencil Computations

Submitted by

Rajbharath Chandramohan

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

Spring 2017

Master's Committee:

Advisor: Sanjay Rajopadhye

Sudeep Pasricha
Oliver Pinaud

Copyright by Rajbharath Chandramohan 2017

All Rights Reserved

Abstract

Hardware Implementation and Design Space Exploration for Wave 2D and Jacobi 2D Stencil Computations

Hardware accelerators are highly optimized functional blocks designed to perform specific tasks at higher performance. We developed a hardware accelerator for Jacobi 2D and Wave 2D algorithms, two computations with a stencil pattern. They are used in many scientific applications in the field of acoustics, electro magnetics and Fluid dynamics. These problems have large problem sizes, memory limitations and bandwidth constraints that result in long run times. Hence, an approach which increases the performance of these problems that reduces bandwidth requirement is necessary. We developed analytical models for the performance, bandwidth and area models for the Wave 2D algorithm and Jacobi 2D algorithm and solved them for the optimal solution using posynomials and positivity property in MATLAB and using Excel Solver. In order to achieve an optimal design, we split the computation into two levels of tiling. The first level called passes is a rectangular prism that runs through the 3-D iteration space. Each pass is mapped to a grid of processing elements(PEs) in the hardware accelerator. The second level of tiling splits the vertical prism into smaller prisms executed by a single PE. These optimizations are implemented in the hardware accelerator designed using Verilog and simulated using ModelSIM. Results from ModelSIM provides an accurate model and an experimental verification of the design. We also achieved improved performance and lower bandwidth.

Table of Contents

Abstract ii

List of Tables v

List of Figures vi

Chapter 1. Introduction 1

1.1. Contributions 2

1.2. Related Work 2

1.3. Thesis Structure 4

Chapter 2. Background 5

2.1. Hardware Accelerators 5

2.2. FPGA Architecture 7

2.3. Stencil Computation 10

2.4. Wave 2D Algorithm 11

2.5. Jacobi 2D Algorithm 12

2.6. Loop Transformations 13

Chapter 3. Wave 2D accelerator design 15

3.1. Analysis of Data Dependencies of Wave 2D Stencil 15

3.2. Passes 18

3.3. Horizontal passes 20

3.4. Vertical passes 23

3.5. Tiling 26

3.6. Processor Allocation 28

3.7. Execution of one tile by a processing element 29

Chapter 4. Jacobi 2D accelerator design 32

4.1. Analysis of data dependencies 32

4.2. Processor Allocation 34

Chapter 5. Hardware Design 35

5.1. Compute unit 35

5.2. Control unit 36

5.3. Memory unit 37

5.4. Verilog Design 38

5.5. Design Verification 41

Chapter 6. Optimization Problem 43

6.1. Wave 2D optimization problem 43

6.2. Jacobi 2D optimization problem 50

Chapter 7. Conclusion and Future work 56

Bibliography 57

List of Tables

3.1 Notations 15

5.1 Timing Verification 42

6.1 Excel Solver Results for wave 2D 49

6.2 MATLAB Results for wave 2D 49

6.3 Excel Solver Results for jacobi 2D 54

6.4 MATLAB Results for jacobi 2D 54

List of Figures

2.1 Cluster of N BLEs *(Figure taken from [10])* 8

2.2 Basic Logic Element (BLE) *(Figure taken from [10])* 9

2.3 Grid of points in x-y plane 11

3.1 Wave 2D dependencies 16

3.2 Wave 2D inter-tile dependencies 17

3.3 Wave 2D dependencies after skewing $\vec{d_{1'}}$=(-3,-3) 17

3.4 Horizontal Pass projected on i-t plane 19

3.5 Horizontal Pass Dependencies 20

3.6 Vertical Passes visualized from i-t plane 22

3.7 Vertical pass 23

3.8 Vertical Passes Dependencies 24

3.9 Tiling along i and j plane 28

3.10 Processor Allocation 30

4.1 Jacobi 2D dependencies 32

4.2 Jacobi 2D dependencies after skewing $\vec{d_{1'}}$=(-1,-1) 33

5.1 Arithmetic pipelined unit 36

5.2 PE communication state1 38

5.3 PE communication state2 39

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 research has been done in building accelerators to solve stencil computations. We consider two different stencils namely, Jacobi 2D stencil and Wave 2D stencil. Jacobi is a popular algorithm for solving Laplace's differential equation on a square domain discretized. In Jacobi method, a body is represented by a 2D array of particles, each with an initial value of temperature and several iterations are computed over the data to compute average temperatures repeatedly until the desired accuracy is reached. Wave 2D algorithm solves wave 2D propagation equation. This equation is used to describe waves in space.

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 our iterative stencil computation, we exploit parallelism by making use of the independent nature of stencil computations. We design an FPGA-based hardware accelerator that efficiently implements the Jacobi 2D method and wave 2D method. We designed multiple, processing elements (PEs) on an FPGA that contain pipelined arithmetic units that produce one new value every cycle. Our hardware accelerator is intended to achieve high performance with low bandwidth and area requirements. To achieve this, we develop a three-step approach to implement Jacobi 2D and Wave 2D algorithm on FPGA. First, we analyze the data dependencies of Jacobi 2D and Wave 2D

and propose two levels of tiling. Second, we 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 and bandwidth constraint. Finally, we map the computation to the processing elements in the hardware accelerator and implement it on FPGA.

1.1. Contributions

The main contributions of this thesis are as follows:

(1) A well-defined approach for deriving FPGA-based hardware acceleration of Wave 2D and Jacobi 2D.

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

(3) Verilog implementation of a 2D grid of Processors that computes a Jacobi 2D computation, an arithmetic unit that delivers the result of one computation every clock cycle in steady state.

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

1.2. Related Work

FPGA technology was used for implementing FDTD method by Schneider et al. [1]. 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. Durbano et al. [2, 3] proposed a design for three-dimensional FDTD using floating point arithmetic. They 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. [4] 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. [5] 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. [6] 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. Wester et al. [7] 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 Wave 2D and Jacobi 2D 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

The rest of the thesis is organized as follows:

In chapter 2, we describe the need for hardware accelerators. We describe the structure of FPGAs and we discuss some program transformations. We then introduce the Wave 2D method and Jacobi 2D method.

In chapter 3, we analyze the data dependencies of Wave 2D and develop two levels of tiling leading to an optimal design. We perform design space exploration of wave 2D stencil and come up with an optimal solution. We develop analytical models characterizing the performance, Bandwidth and Area. We use MATLAB and excel solver to find the optimal solution.

In chapter 4, we analyze the data dependencies of Jacobi 2D stencil. We perform design space exploration of Jacobi 2D stencil. We discuss the hardware implementation of Jacobi 2D in Verilog. The Verilog design, the synthesized hardware and its characteristics are explained in detail.

Chapter 5 draws conclusions and gives suggestions for future work.

CHAPTER 2

Background

In this chapter, we present some of the background information that will help to follow the rest of the thesis. We discuss the advantages of hardware accelerators, FPGA architecture, wave 2D algorithm, Jacobi 2D algorithm and some program transformations.

2.1. Hardware Accelerators

The need of hardware accelerators can be associated with the drawbacks of 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.

(2) Energy and Heat Dissipation: Despite stopping the rise of clock frequencies, multicore 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.

(3) 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.

(4) 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. [8] 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. 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. The memory hierarchy, pipeline stages, interconnects between processing blocks can be customized for specific application. The accelerator may only be used to perform Stencil computation 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 mean more parallelism but also higher cost and more power consumption. Thus, there is a trade-off between hardware resources and performance. The prime considerations while designing the FPGA-based hardware accelerators are listed below.

(1) Designing the arithmetic unit in a pipelined fashion that can compute one value every clock cycle in steady state

(2) Designing efficient memory controller to control reads/writes from/to memory locations

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

(4) Increasing memory reuse

(5) Avoiding idling of resources

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

(7) Design parameter choices also include tile size selection and selecting the number of processing elements (PEs)

2.2. FPGA Architecture

Field Programmable Gate Array (FPGA) is an Integrated circuit designed to be configurable after manufacturing by the designer or by the customer. An FPGA contains an array of programmable logic blocks and a hierarchy of reconfigurable interconnects that allows the blocks to be wired together. The programmable logic and routing interconnect of FPGAs makes them flexible and general purpose. 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) [9]. 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 Languages (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. Configurable Logic Blocks. A configurable logic block is a fundamental building block of Field Programmable Gate Array. These blocks can be configured by the programmer. It is reconfigurable hardware where a hardware block can be used to implement more than one function. They can implement combinational and sequential logic. The block contains RAM for creating arbitrary combinatorial logic functions, also known as lookup tables (LUTs). It also contains flip-flops for clocked storage elements, along with multiplexers to route the logic within the block and to and from external resources. The multiplexers also allow polarity selection and reset and clear input selection. Any digital logic can be implemented by configuring one or more LUTs. FPGAs also contain some special purpose blocks like memory, multipliers, adders, DSP blocks etc. These are called as hard blocks and are integrated on chip to implement specific frequently used functions very efficiently.

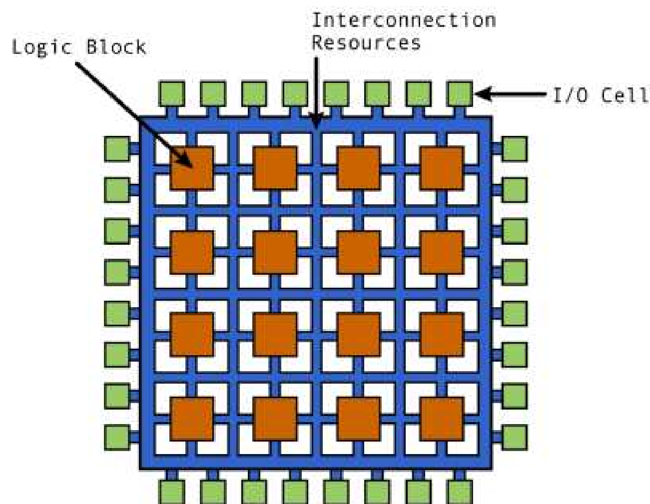

Figure 2.1. Cluster of N BLEs *(Figure taken from [10])*

2.2.2. Programming Technologies. Every FPGA relies on an underlying programming technology. There are different kinds of programming technologies and they can have a significant impact on programmable logic architecture. The approaches that have been used in the past are EPROM, EEPROM, flash, static memory and anti-fuses. Most of the

FIGURE 2.2. Basic Logic Element (BLE) *(Figure taken from [10])*

modern FPGAs use Flash, static memory and Anti-fuse. SRAM are reprogrammable and volatile but they have a high area cost. Flash memories are not volatile but they are reprogrammable. Antifuse are not reprogrammable and not volatile but they have low area. Most devices from Xilinx and Altera have static memory cells distributed throughout the FPGA to provide reconfigurability. SRAM remains the most dominant programmable technologies for most FPGA because of its re-programmability and use of standard CMOS technology. Since they use standard CMOS technology, they can make use of higher speeds and lower dynamic power with new process technologies. The volatility of SRAM cell necessitates the use of external devices to store configuration data when the device is powered off. External flash or EEPROM devices add to the cost of an SRAM-based FPGA. Flash memories are non-volatile. This eliminates the need for external resources required to store and load configuration data when SRAM based programming technology is used. Flash based approach is more area efficient. One disadvantage of Flash based devices is that it cannot be reprogrammed infinite number of times. Charge buildup eventually prevents a flash based device from being properly erased and programmed. A trend that has recently emerged is to use Flash storage in combination with SRAM technology with Flash memory used to provide

non-volatile storage while SRAM cells are still used to control the programmable elements in the design. Anti-fuse technology is an alternative to SRAM and flash based technologies. Unlike SRAM or floating gate programming technologies, this is not reprogrammable. The primary advantage of using Anti-fuse technology is its low area. It is non-volatile. This technology uses a non-standard CMOS process and are typically well behind in the manufacturing process. The inability to reprogram makes it unsuitable for applications where configuration changes are required. We use cyclone V GX family FPGA where SRAM is coupled with flash memory.

2.3. Stencil Computation

Stencil codes are a class of iterative kernels which update array elements according to some fixed pattern, called stencil. They are most commonly found in the codes of computer simulations, e.g. for computational fluid dynamics in the context of scientific and engineering applications. Other notable examples include solving partial differential equations, the Jacobi kernel, the GaussSeidel method, image processing and cellular automata. The regular structure of the arrays sets stencil codes apart from other modeling methods such as the Finite element method. Most finite difference codes which operate on regular grids can be formulated as stencil codes. Stencil codes perform a sequence of sweeps through a given array. Generally, this is a 2 or 3-dimensional regular grid. In each time step, the stencil code updates all array elements. Using neighboring array elements in a fixed pattern, each cell's new value is computed. In most cases boundary values are left unchanged, but in some cases, they need to be adjusted during the computation as well. Since the stencil is the same for each element, the pattern of data accesses is repeated. A stencil can also be defined as a geometric arrangement of a nodal group that relate to the point of interest by using a

numerical approximation routine. Stencils are the basis for many algorithms to numerically solve partial differential equations (PDE).

2.4. Wave 2D Algorithm

2D wave propagation is governed by the following equation.

$$\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = v^{-2}\frac{\partial^2 u}{\partial t^2}$$

where, u(x,y,t) is the wave field and v(x,y) is the velocity of the medium. In order to

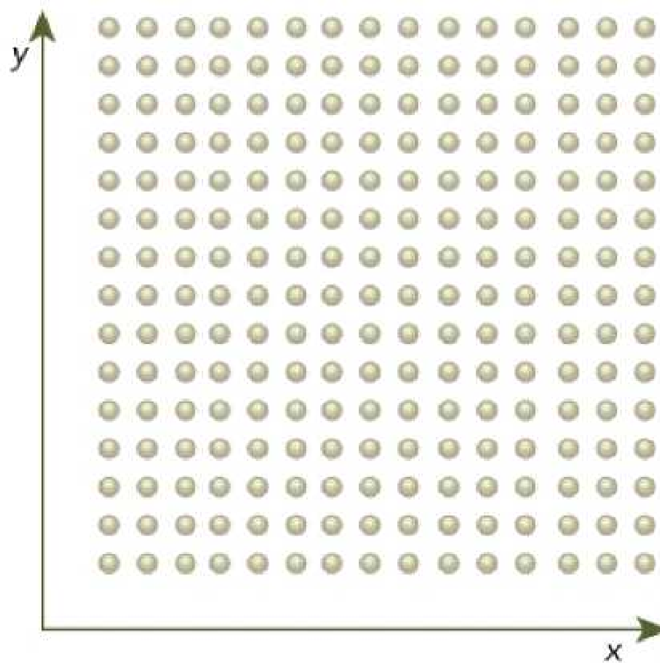

Figure 2.3. Grid of points in x-y plane

solve a continuous time equation using a computer, we have to discretize it. A discrete model can approximate a continuous one to any desired degree of accuracy. Developing such approximations is important in the field of applied mathematics and it provides a practical approach to these problems. It is possible to create a discrete wave model starting with the continuous wave equation. Wave equation describes how the behavior of a wave changes

based on conditions in its immediate neighborhood. One of the ways of discretization is to define a grid or array of points as shown in Figure 2.3. Instead of defining the function everywhere, we consider only selected points. The more closely these points are spaced the more accurate an approximation to the continuous case and the more time consuming the computation. In our case, we consider two dimensions in space and one in time. Only the space dimensions are shown in the grid. Indices are used to locate points in the grid. f(i,j,t) is the location i,j in space at time t. It has neighboring points in space and time. Continuous differential equations are defined by taking the limit of finitely spaced locations as the distance between points goes to zero. The first order difference is computed by subtracting neighboring values along the relevant dimension (i,j or t). The wave equation uses the second order difference or rate of acceleration. To get the second order difference, we compute a difference of differences. In generating the difference equation from the differential equation, we must consider the time and distance scale of the points on the grid. The second order difference is computed by subtracting one first order difference from the other. Based on the level of accuracy, after discretization we end up with a n-point stencil.

2.5. Jacobi 2D Algorithm

Jacobi is a popular algorithm for solving Laplace's differential equation on a square domain discretized. This arises in heat flow, electrostatics, gravity, and other situations. In 2-dimensions the heat equation is,

$$u_t = c^2(u_{xx} + u_{yy}) \tag{2.5.1}$$

Jacobi method is an iterative approach to solving Laplace equation. Let us consider a body represented by a 2D array of particles, each with an initial value of temperature. This

body is in contact with a fixed value of temperature on the four boundaries, and Laplace equation is solved for all internal points to determine their temperature as a function of the four neighboring particles. These temperature values are expressed as a 2D array. Several iterations are performed over the array to recompute new temperatures repeatedly. At each step, a new value of temperature is obtained. Note that all new values can be computed independently of one another. This is used to parallelize the algorithm. The computed values gradually converge to a finer solution until the desired accuracy is reached.

2.6. Loop Transformations

Loop transformations can be used for memory optimization. They can improve memory performance or enable other optimizations that have been impossible before due to bad data dependencies.

Loop blocking or tiling is a common loop transformation which consists of breaking the entire loop into chunks. This is mainly done on the iteration space and can be seen as a task partitioning. Loop Tiling or Loop blocking is a loop Optimization technique used by programmers or compilers to make the execution of certain types of loops more efficient. It is done to make sure that data that is currently accessed stays in the cache until its reused. It is used to avoid cache misses. In our case, Tiling is used so that we can parallelize the code enabling different processing elements to compute tiles in parallel. Loop tiling reorganizes a loop to iterate over blocks of data sized to fit in the cache. Loop blocking can be considered to improve cache performance, derive a coarse-grained parallelism from a fine-grained model or to handle memory constraints.

Loop splitting breaks a loop into multiple loops which have the same bodies but iterate over different contiguous portions of the index range. Loop Fusion merges adjacent loops

with identical control into one loop. This transformation is valid if the fusion does not introduce any lexically backward data dependence. Its purpose is to reduce loop overheads, improve immediate data reuse and reduce data transfers. If the granularity of a loop, or the work performed by a loop, is small, the performance gain from distribution may be insignificant. This is because the overhead of parallel loop start-up is too high compared to the loop workload.

The loop skewing transformation changes the shape of the iteration space without changing the dependencies. It looks like a geometrical transformation, which can help to expose a canonical parallelism. Skewing is a program transformation that is done if the program that is currently being computed cannot be computed in parallel due to the nature of the dependence. Skewing the program results in changing the dependencies of the program that would then allow us to implement it in parallel thereby improving the performance.

Scheduling is the method by which tasks are assigned to processors in a computer system. Each task is given a time at which it is executed based on the priority. In our case, Scheduling helps us assign times at which each point in the iteration space is computed. A schedule should be chosen based on the performance it delivers. It is beneficial to divide the work evenly among the different processing elements in the hardware accelerator when the work performed by different iterations of a loop is the same. Provided there are no interruptions and each processing element progresses at the same rate, all the processing elements will complete at the same time.

CHAPTER 3

Wave 2D accelerator design

In this chapter, we first analyze the data dependencies of Wave-2D computation. Analyzing the data dependencies of a computation plays a crucial role in deciding the architectural features and control flow. We look at different design choices and analyze how they impact the design.

Table 3.1. Notations

parameter	description
M	Total number of points in i-th dimension
N	Total number of points in the j-dimension
T	Total number of points in the t-dimension
p_{i}	Number of processing elements (PE) along i-direction
p_{j}	Number of processing elements (PE) along j-direction
x	Tile size along i-direction
y	Tile size along j-direction
z	Tile size along k-direction
d	Depth of the arithmetic pipeline
L	Pass length along i-direction
W	Pass width along j-direction
H	Pass height along k-direction

3.1. Analysis of Data Dependencies of Wave 2D Stencil

The table above shows some notations we use in this chapter. The wave 2D equation is,

$$B(t,i,j) = cB[t-1,i-3,j] + B[t-1,i-2,j] + B[t-1,i-1,j] + B[t-1,i+3,j]$$
$$+ B[t-1,i+2,j] + B[t-1,i+1,j] + B[t-1,i,j-3] + B[t-1,i,j-2] + B[t-1,i,j-1]$$
$$+ B[t-1,i,j] + B[t-1,i,j+1] + B[t-1,i,j+2] + B[t-1,i,j+3] \quad (3.1.1)$$

As we can see from wave-2D equation, we have an array that updates based on its neighbors. We have 13 dependencies. We use these data dependencies to draw the data

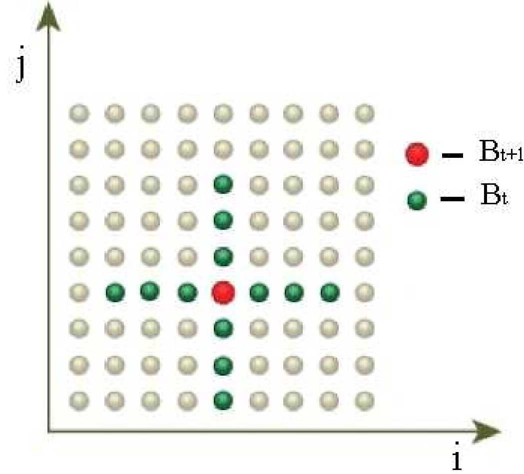

FIGURE 3.1. Wave 2D dependencies

dependence graph. The Data Dependence Graph (DDG) is shown in Figure 3.1, which represents the dependence of points in the iteration space. In Figure 3.1 the circles represents a point in the iteration space. The red circle depends on all the green circles. The red circle is located at time plane t+1 and the green circles are located at time plane t. The red circle depends on all the green circles including the one under it. There is a green circle underneath the red circle. The nature of this dependence is uniform as depicted in the figure Figure 3.1 and as we can see from the equation.

We have a three dimensional iteration space with two space dimensions i,j and one time dimension t. We update each point in the iteration space using the equation. The basic 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 Wave 2D algorithm offers abundant parallelism. Based on the nature of dependencies of wave 2D computation, we conclude that these dependencies are uniform and all points in the plane t+1 depends on points in t-plane. Therefore, the points in the plane t+1 can be computed independent of

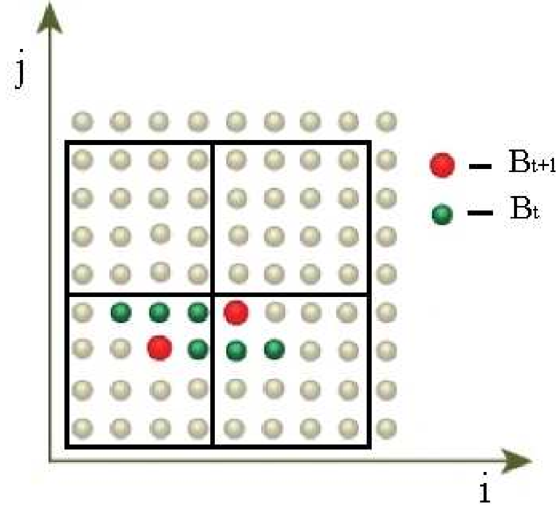

FIGURE 3.2. Wave 2D inter-tile dependencies

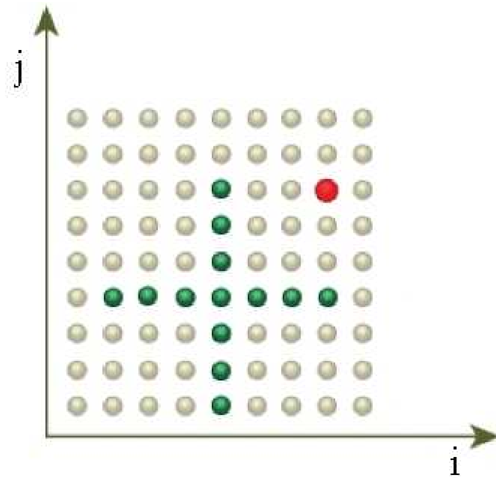

FIGURE 3.3. Wave 2D dependencies after skewing $\vec{d_{1'}}$=(-3,-3)

each other. We make use of this independent nature to exploit parallelism. We can execute these points in the same plane in parallel enabling the use of multiple processing elements (PEs) that can compute different points in the iteration space at the same time thereby improving the performance.

The iteration space is of size M,N,T. Most problems have large values of M,N and T. Therefore, it is not possible to fit all the array values in the hardware accelerator. So, we use loop Blocking/Loop Tiling to split the iteration space into smaller chunks that can fit

on the hardware accelerator. Tiling transformations enable parallelization and data-locality optimization. We propose two levels of tiling for implementing Wave 2D algorithm on the hardware accelerator. The outermost level is called passes and the innermost level is called tiles. 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. We do the inner level of tiling to provide parallelism and to have less idle time. To tile safely, we must make sure that there are no cyclic dependencies among tiles. Assuming that we use rectangular tiles, we can clearly see from Figure 3.2 that the tile on the left depends on the tile on the right and the tile on the right depends on the tile on the left. We have shown the dependence to the right along i axis of the left tile and the dependence to the left along i axis of the right tile for clarity. We have inter-tile dependencies. It is not safe to tile with our dependencies. To overcome this, we use a loop transformation called skewing.

The loop skewing transformation changes the shape of the iteration space without changing the dependencies. It looks like a geometrical transformation, which can help to expose a canonical parallelism. Skewing the program results in changing the dependencies of the program. Skewing our iteration space by a factor of 3 would result in a different dependence as shown in Figure 3.3. Because of skewing, we have dependencies to the left tile and the tile below. Now, we can tile the iteration space without causing any cyclic dependencies. There are different ways to break the iteration space using passes and tiles. We will look at some of them in the next section.

3.2. Passes

As mentioned in the previous section, the iteration space is split into smaller chunks called passes. These passes can be in the form of rectangular prisms. These rectangular

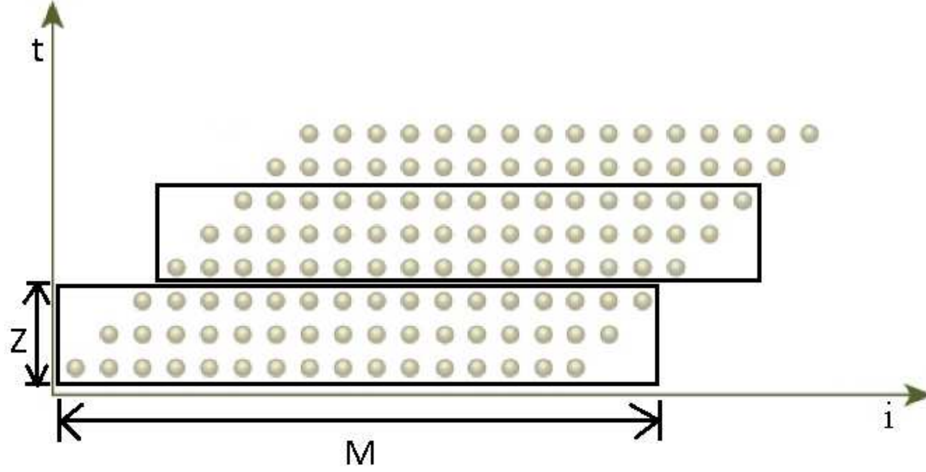

FIGURE 3.4. Horizontal Pass projected on i-t plane

prisms are executed on the hardware accelerator. There are different ways of splitting the iteration space into rectangular prisms. These prisms can be along the i-direction, j-direction or the t-direction. The problem size is M * N * T. The pass dimensions of a pass along i direction is given by M * W * H, along j direction is given by N * L * H, along k direction is given by T * L * W, where L, W and H are given by,

$$L = x * p_x \tag{3.2.1}$$

$$W = y * p_y \tag{3.2.2}$$

$$H = z * p_z \tag{3.2.3}$$

Based on the direction of passes described above, We consider two different kinds of passes.

1. Horizontal passes and

2. Vertical passes.

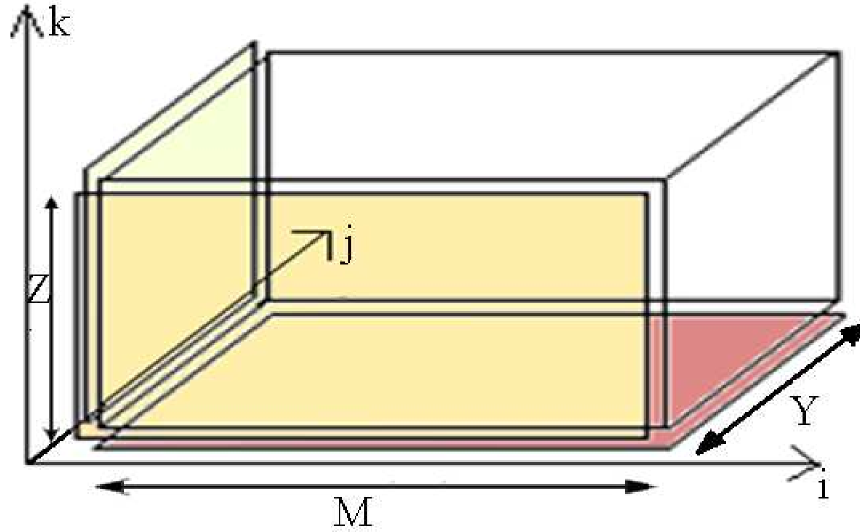

Figure 3.5. Horizontal Pass Dependencies

3.3. Horizontal passes

Horizontal passes are rectangular prisms along i direction or j direction. Consider a pass in the i direction, the pass would have a length M, width W and height H as shown in Figure 3.4. Based on the nature of our dependencies, we see that each pass needs data from neighboring passes on its left boundary(i direction), from the pass below(t direction) and from the pass in front(j direction) as we can see from Figure 3.5. There are some advantages and disadvantages to using horizontal passes. Horizontal passes have an uneven boundary because of skewing as we see in Figure 3.4. The uneven boundaries increase the complexity of the memory controller. The pass is mapped to a grid of processing elements in the j-t plane. Due to the nature of data dependencies, some processing elements (PEs) must wait for its neighboring processing elements (PEs) to execute its first tile before starting its computation. So, there is no concurrent start for processing elements (PEs). We develop a cost function for horizontal passes along i direction. This cost function is based on the I/O between passes. We compute that in two steps.

1. I/O between one pass and another
2. Total I/O from all the passes

In order to compute I/O, we require the amount of data needed by a pass. The amount of data needed by a pass in i direction is,

$$D_t = M * W \tag{3.3.1}$$

$$D_i = 6 * W * H \tag{3.3.2}$$

$$D_j = 6 * M * H \tag{3.3.3}$$

$$D_{pass} = M * W + 6 * M * H + 6 * W * H \tag{3.3.4}$$

The size of the problem is,

$$P_{size} = M * N * T \tag{3.3.5}$$

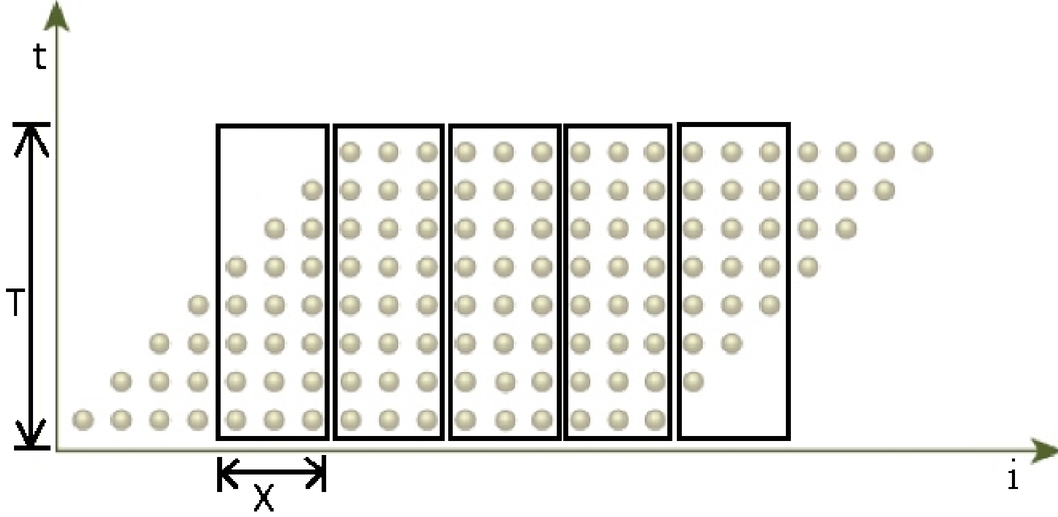

FIGURE 3.6. Vertical Passes visualized from i-t plane

The total number of horizontal passes is given by,

$$Pass_{tot} = \frac{T}{H} * \frac{N}{W} \tag{3.3.6}$$

Total I/O for all the passes is,

$$I/O_{pass} = \frac{T}{H} * \frac{N}{W} * (M * W + 6 * M * H + 6 * W * H) \tag{3.3.7}$$

The amount of data needed by a pass is a function of M which is quite large. To start computing a pass, we need $M * W + 6 * M + 6 * W$ values which is quite large. With some optimizations, the data needed to start the computation can be reduced. But, we still have the problem of uneven boundaries and a complex memory controller. It would lead to a suboptimal design. There is a different version of horizontal pass where the prism is along the j direction. All the values computed earlier are the same except that, they will be corresponding to j direction. So, we do not dicuss horizontal pass along j direction.

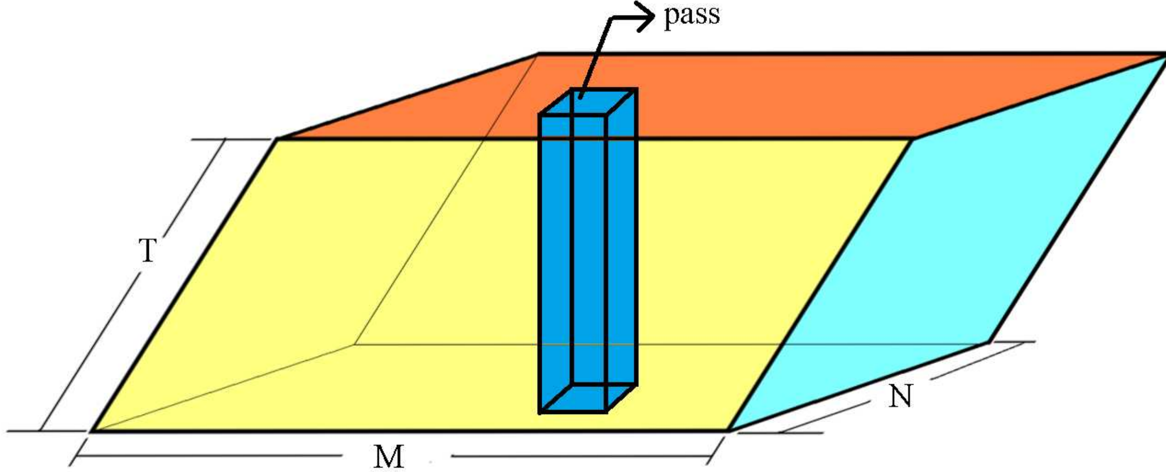

Figure 3.7. Vertical pass

3.4. Vertical passes

Vertical passes are rectangular prisms along t direction. Consider a pass in the t direction, the pass would have a length W, width H and height T as shown in Figure 3.6. A vertical pass in the iteration space is shown in Figure 3.7. Based on the nature of our dependencies, we see that each pass needs data from neighboring passes on its left boundary(i-direction), from below(t-direction) and from front(j-direction) as we can see from Figure 3.8. There are some advantages and disadvantages to using vertical passes. There are some uneven passes at the boundary because of skewing. There is a near concurrent start for processing elements (PEs) as we will explain in the next sections. The pass is mapped to a grid of processing elements in the i-j plane.

Vertical passes have an easier control because they are rectangular prisms with the same boundary conditions. The uneven passes at the boundary is out of scope of our thesis and can be resolved using other methods. Our objective is to implement a single pass on the hardware accelerator in the most efficient way. In most cases, T is larger than W or H because W and H are cut off from a larger M and N. The hardware accelerator computes a

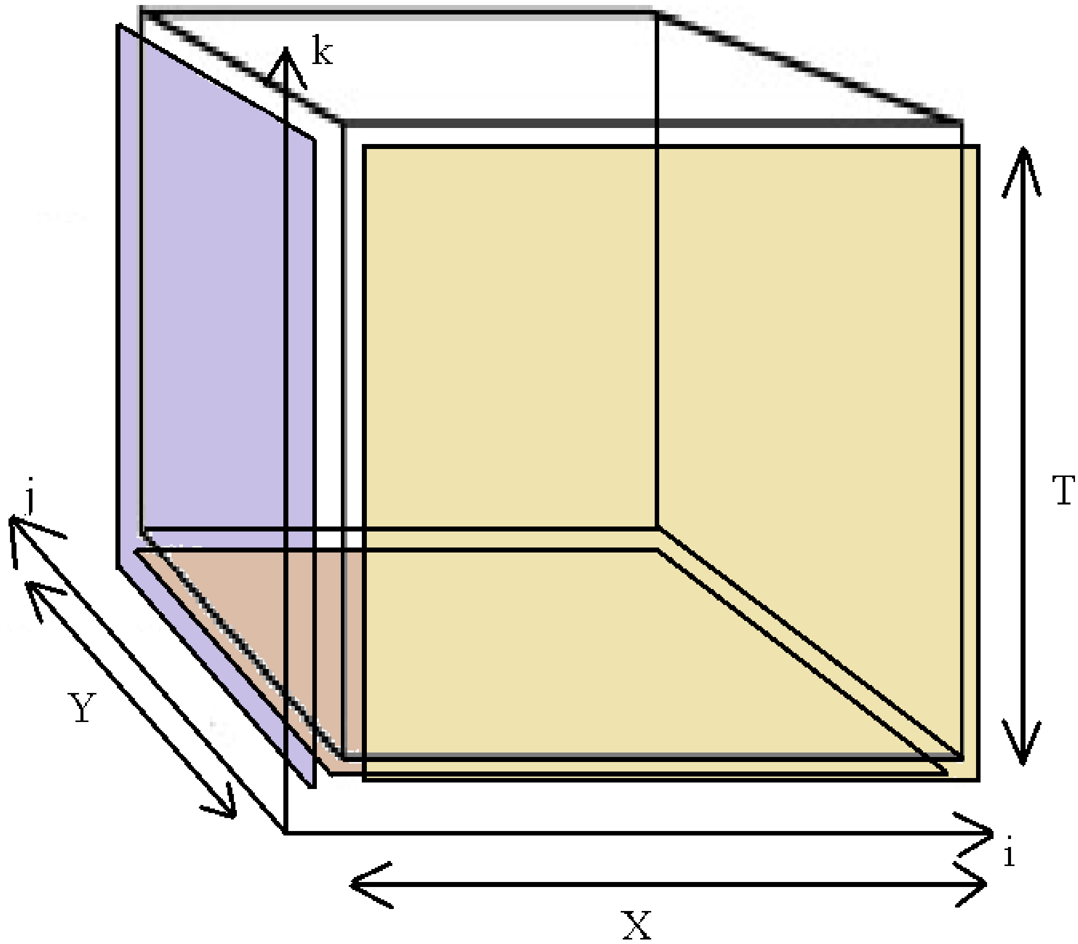

FIGURE 3.8. Vertical Passes Dependencies

tile pass as a vertical prism with length W, width H and height T in one pass. Within a pass, each processing element (PE) executes rectangular tiles sequentially. A pass is mapped to the hardware accelerator and it is executed by all the PEs.

We develop a cost function for horizontal passes along t direction. This cost function is based on the I/O between passes. We compute that in two steps.

1. I/O between one pass and another

2. Total I/O from all the passes

To compute I/O, we require the amount of data needed by a pass. The amount of data needed by a pass in t direction is,

$$\text{Data}_t = L * W \tag{3.4.1}$$

$$\text{Data}_i = 6 * W * T \tag{3.4.2}$$

$$\text{Data}_j = 6 * L * T \tag{3.4.3}$$

$$\text{Data}_{tot} = x * W + 6 * W * T + 6 * L * T \tag{3.4.4}$$

$$\text{Problem size} = M * N * T \tag{3.4.5}$$

$$\text{No. of Vertical Passes} = \frac{M}{L} + \frac{N}{W} \tag{3.4.6}$$

Total I/O for all the passes is,

$$IO_{pass} = \frac{M}{L} + \frac{N}{W} * (x * W + 6 * W * T + 6 * L * T) \quad (3.4.7)$$

When we use vertical passes, the total amount of data needed by a pass is a function of T which is quite large. But, we do not need all its input data to start the computation. We only need $x * W + 6 * L + 6 * W$ values to start the computation which is considerably small. We chose to use vertical passes in our design since less data is required to start the computation and we have a simpler memory controller. Also, we have a concurrent start for processing elements in i direction and a slight delay based on pipeline depth for processing elements(PEs) in j direction. Each pass is executed by the hardware accelerator in a tiled fashion. The tile dimensions are x,y and z. We will discuss different types of tiling in the next section.

3.5. Tiling

We split one pass into multiple tiles that are mapped to the hardware accelerator. We map the tiles to a grid of processing elements(PEs). A set of tiles are executed sequentially by a processing element(PE). We develop analytical models characterizing the data reuse that results from executing the tiles along different directions.

The data reuse from executing tiles along i axis is,

$$\mathrm{R}_i = 6 * W * H \quad (3.5.1)$$

The data reuse from executing tiles along j axis is,

$$\mathrm{R}_j = 6 * L * H \tag{3.5.2}$$

The data reuse from executing tiles along t axis is,

$$\mathrm{R}_t = L * W \tag{3.5.3}$$

We have tiles in all 3 directions. Each tile is executed by one processing element(PE). Each processing element(PE) has memory and an arithmetic unit that performs the computation. We will discuss the structure of a processing element(PE) in the next chapter. We have a grid of processing elements (PEs) computing different independent tiles at the same time. As we discussed earlier, we have dependencies along i axis, j axis and t axis . Based on the nature of dependencies, each tile depends on the tile to its left(i direction), the tile in front of it(j direction) and the tile below it(t direction). Each processing element(PE) can communicate with its neighbors. The communication network and the nature of the communication is covered in the next chapter. We have shown the tiling along i-j plane in Figure 3.9. The data needed for computing the first tile in a pass is,

$$data = L * W + 6 * W * H + 6 * L * H \tag{3.5.4}$$

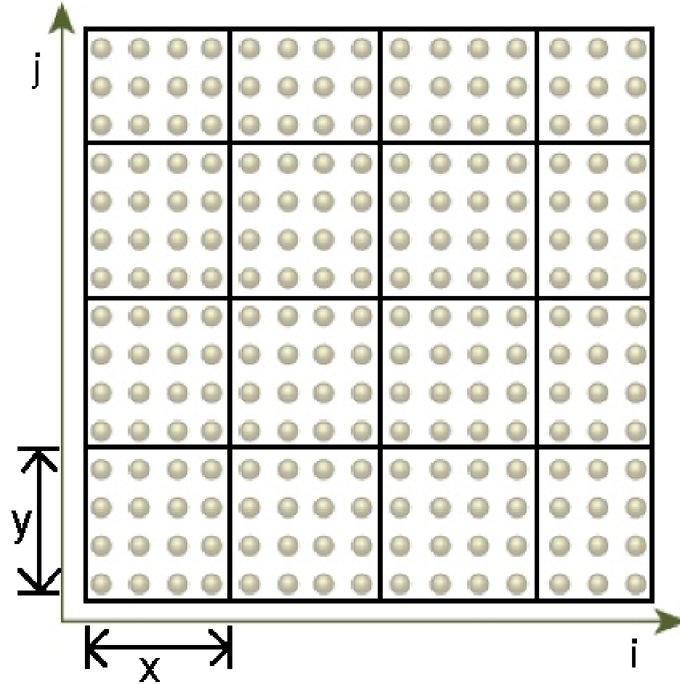

Figure 3.9. Tiling along i and j plane

3.6. Processor Allocation

There are different ways of allocating tiles to processing elements (PEs). Some of them are Rowwise Allocation, Columnwise Allocation, Planewise Allocation and their combinations. In case of Rowwise allocation, we have PE's computing one row of tiles in each plane and the corresponding rows above it. Every other PE's are responsible for computing other rows in the same timeplane in the j direction. This requires some communication between neighboring PEs since some of the results computed by the current PE is needed by the neighboring PEs to compute their result. The advantages of this scheme is that we have concurrent startup because the data computed by a PE is needed by its neighbor only after it completes the current set of rows. However, the problem is that it needs a lot of local memory. The total amount of memory required is determined by the pass length W. This is a waste of resource on on silicon area. In case of Columnwise allocation, we assign multiple

columns to each PE in a similar fashion to rowwise allocation. But we still have the same problem as earlier. The total amount of memory required is determined by the pass width H. The other allocation is planewise allocation where a PE is responsible for one plane of tiles and the neighboring PE is responsible for the plane of tiles above it and they are assigned in a Round Robin fashion. The amount of local memory still depends on the size of a pass. Also, it needs a lot of data from the previous PE before it can start. Since, there is no concurrent start, there is no parallelism. However, each plane can be broken into smaller tiles. And these tiles are assigned to the processing elements(PEs). In this scheme, the data for each PE is lowered but we still have the problem of concurrent start and each PE has to wait for the results from the neighboring PEs. If we reduce the size of each pass, it reduces the amount of memory needed for each PE. But, we still have the problem with concurrent start. We could use wavefront schedule to expliot some parallelism except for the boundary tiles. However, that still requires a substantial inter PE bandwidth.

In order to overcome the problems from the previous scheme, we chose to allocate a single plane of tiles among all the PEs. So, now the inter PE bandwidth is greatly reduced and we also have concurrent start. We have a concurrent start because the data needed by a PE is available from its neighboring PE by the time it computes its first tile. So, basically the 3D iteration space is mapped to a 2D grid of processors in the j-k plane. Each Processing element is responsible for multiple tiles in the k-direction. The processor allocation is shown in Figure 3.10

3.7. Execution of one tile by a processing element

In this section, we will explain the execution of one tile by a PE. As we know from previous sections, each tile has dimensions x, y and z. The points in a tile are executed in a sequential

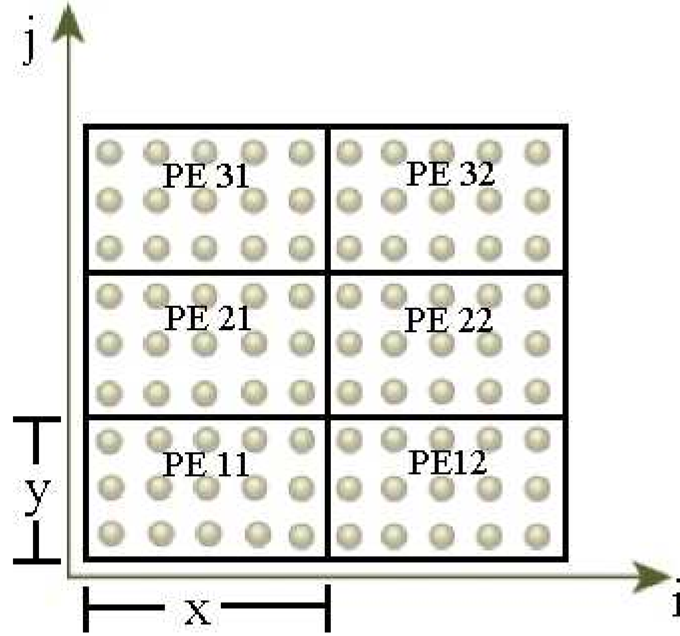

FIGURE 3.10. Processor Allocation

fashion by one processing element. The order of execution of points in a tile is important to determine the performance, memory requirement and bandwidth. The tile can be executed in different ways. We can execute a tile along i direction first, then along j direction and finally along t direction or execute along j-direction first and then along other directions or even execute random points in tile in a sequential fashion. If we execute poins along t-direction first, the data reuse is really low since only the currently computed value can be reused. Even if we increase the data reuse by using additional memory, the performance would be greatly hindered since we have a pipelined arithmetic unit. The pipelined arithmetic unit would take a few cycles equal to pipeline depth to produce the current output. It leads to a sub-optimal design. If we execute points in random fashion, we have no data reuse and it would be a suboptimal design. If we execute points along i direction or j direction, the amount of data reuse is the same. Therefore, we choose to execute points in the i-direction in a processing element in a sequential fashion. As we mentioned earlier, we compute points

in a tile in a pipelined fashion. The total number of independent computations in a tile is given by,

$$Independent computations = x * y \tag{3.7.1}$$

In order to use pipelining effectively, we need to make sure that the total number of independent computations in a tile is greter than the pipeline depth. Therefore,

$$Independent computations = x * y \geq d \tag{3.7.2}$$

CHAPTER 4

Jacobi 2D accelerator design

In this chapter, we discuss the design of Jacobi 2D accelerator.

4.1. Analysis of data dependencies

Jacobi 2D stencil is very similar to wave 2D stencil. As discussed earlier, the Jacobi 2D equation is given by,

$$B[t+1,i,j] = c1(B[t,i-1,j]+B[t,i+1,j]+B[t,i,j-1]+B[t,i,j+1])+c2*B[t,i,j] \quad (4.1.1)$$

By analyzing equation (4.1.1), we can draw a Data Dependence Graph (DDG) that represents the dependence of points in the Iteration space as shown in Figure 4.1. The light circles in the figure represents a point in the iteration space. The red point depends on all the green points and the point below it and there is a green point underneath the red point. We have uniform dependencies and we update each point in the iteration space using equation (4.1.1). As we can see from the equation, this update requires points from the

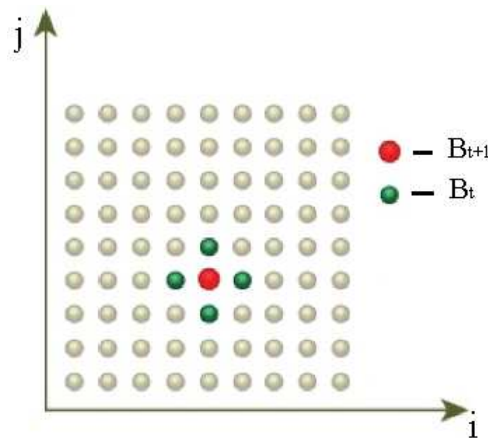

Figure 4.1. Jacobi 2D dependencies

previous timestamp since we have a point in time t+1 as a function of 5 other points in t-plane. We use two levels of tiling. The first level of tiling is called passes and it is defined as a vertical rectangular prism passing through the iteration space. Then, we split them into smaller prisms called Tiles and they are mapped to processing elements(PEs). We apply tiling transformations to optimize our design. We use passes since the data that can fit on the hardware accelerator is limited. We use tiling so that we can allocate processing elements (PEs) to compute different parts of the computation in a parallel fashion. In order to use rectangular tiles, we have to make sure that there are no cyclic dependencies between tiles. As we can see from Figure 4.1, we have dependencies to the left, right, above and below along i and j axis. If we use rectangular tiles, we would have a cyclic dependence. So, in order to use rectangular tiling, we first skew the iteration space. After skewing the iteration space, we show a section in the middle of the iteration space away from the boundary in Figure 4.2.

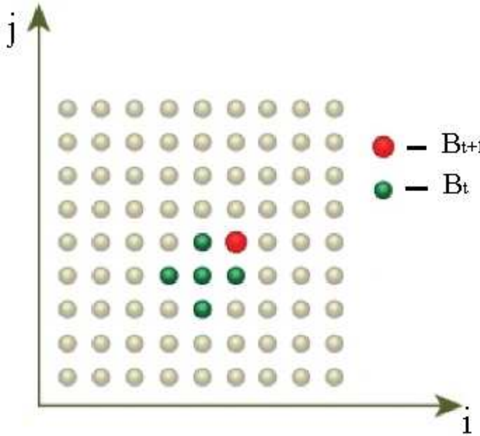

FIGURE 4.2. Jacobi 2D dependencies after skewing $\vec{d_{1'}}$=(-1,-1)

We can see the dependence of $B[t, i, j]$ on values from previous timestamp from Figure 4.2. Each point in the iteration space now depends on points to the left along i-direction and

below along j-direction. As a result, we can tile the iteration space without causing any cyclic dependencies.

4.2. Processor Allocation

There are different ways of allocating tiles to PEs. We perform two levels of Tiling. The first level of tiling is called a pass. A pass is a vertical rectangular prism in 3 dimensions. The pass has dimensions of x,y and T. The second level of tiling is done by blocking along all 3 directions. After tiling, we assign processing elements (PEs) to the set of tiles they compute. We assign each processor to execute a set of consecutive tiles in t-direction. We have a concurrent start for the processing elements (PEs) in i direction because the data needed by a processing element (PE) is available from the processing element to its left by the time it computes its first plane. We have a latency for the PEs in j direction because the data needed by a PE is available from its neighboring PE at a delay equal to the pipeline depth. Assume we have a pipelined arithmetic unit with a pipeline depth of 6. It takes 6 cycles to produce the first output. After the 6th cycle, one new value is produced every cycle. So, the PEs in j direction gets delayed by a factor of pipeline depth(d) since the data needed to start a processing element is only available d cycles after the start of the first processing element. So, basically the 3D iteration space is mapped to a 2D grid of processors in the j-k plane. Each PE is responsible for multiple tiles in t-direction.

CHAPTER 5

Hardware Design

In this chapter, we discuss the design of hardware accelerator. Our hardware design comprises three main components.

(1) Compute unit and control unit within each PE

(2) Memory unit

(3) Communication between PEs.

We implemented jacobi 2D stencil in Verilog. Jacobi 2D stencil is similar to wave 2D except the total number of dependencies is 5. We used Jacobi 2D because it wave 2D stencil is more complicated to implement and the results from Jacobi 2D can be translated to wave 2D stencil. We implemented our hardware design in Altera's Quartus II using Verilog. In a tile pass, 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. Now, we will discuss each unit in detail.

5.1. Compute unit

We have an arithmetic unit that can produce one value every clock cycle. It is a pipelined arithmetic unit. In order to make use of the arithmetic unit to its maximum efficiency, we use independent computations to keep the pipeline full at all times. The arithmetic unit has some registers to store the intermediate results. So, in our case, we have a 5 input 1 output arithmetic unit as shown in Figure 5.1.

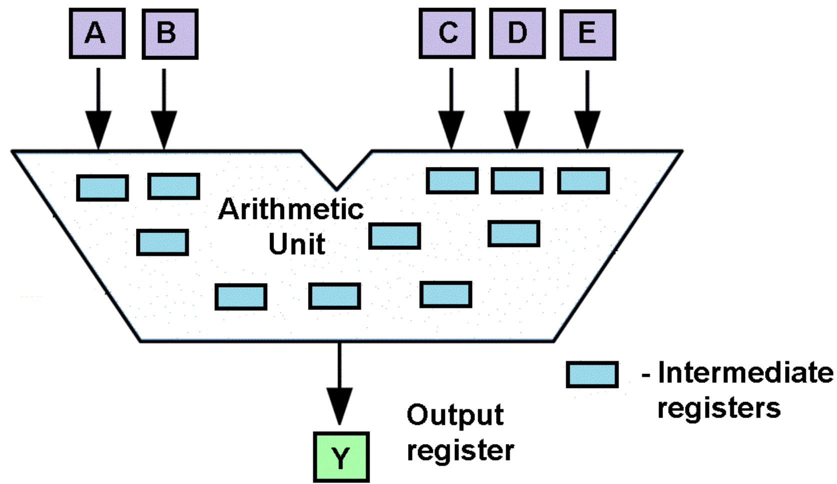

FIGURE 5.1. Arithmetic pipelined unit

5.2. Control unit

In order to maximize the efficiency of the arithmetic unit, we calculate a rectangular block of data. The rectangular block of data has independent computations because the computation of these data require values from previous time. Each Processing element (PE) has memory to store the rectangular block of data. The control unit in a processing element(PE) is responsible for calculating the addresses that we read from and the addresses that we write into. These addresses are used to access the memory inside each Processing element (PE). The row addresses are calculated based on a row counter that counts up to the number of elements in a row and then it resets to zero. The column address is calculated by incrementing by 1 every time the row counter resets. The control unit is also responsible for reading boundary values. The first two rows of values depend on values at the boundary. Also, the first two elements in every row depend on values from the boundary. Muxes are used to select the values from buffers at these points. In steady state, we need five points to compute one value. We notice that two of the five values can be reused if we compute

values along i direction or j direction. In order to make use of this reuse, we use a 3 word shift register. As a result, we read 3 new values every cycle. These values get shifted and therefore are reused. There are additional buffers outside a processing element that store boundary values. To make sure that we read the correct values from the boundary and to have an easier control, we use two buffers and swap them at the end of computing each plane. So, when we read values from one buffer, values are written on to the other buffer. We only have one buffer for boundary PEs.

5.3. Memory unit

We require memory inside and outside a PE. We have memories to store a rectangular block of data inside each PE. The buffers are memories outside Processing elements(PEs) that are used to store boundary values needed by them. There is a memory controller outside PEs that control the flow of data in the PE grid. This memory controller works using a toggle mechanism among all the Processing elements(PEs). Each Processing element has two input buffers and two output buffers. The Processing element reads values from the two input buffers and writes values into the two output buffers. The input buffer that a processing element reads from is swapped with the output buffer of its neighbor at the end of a rectangular block/tile. At the end of a rectangular Block/Tile, every processing element finished writing its output to its corresponding output buffer. This output buffer is swapped with the input buffer of the neighboring Processing element. Now, each PE has the right set of inputs to read from. The new output buffers of each Processing element are now ready to receive new values. This method is used to make sure that none of the values are overwritten before they are last used. The memory inside a PE is a collection of 1D memories. As we discussed earlier, the control unit generates row and column addresses. The column address

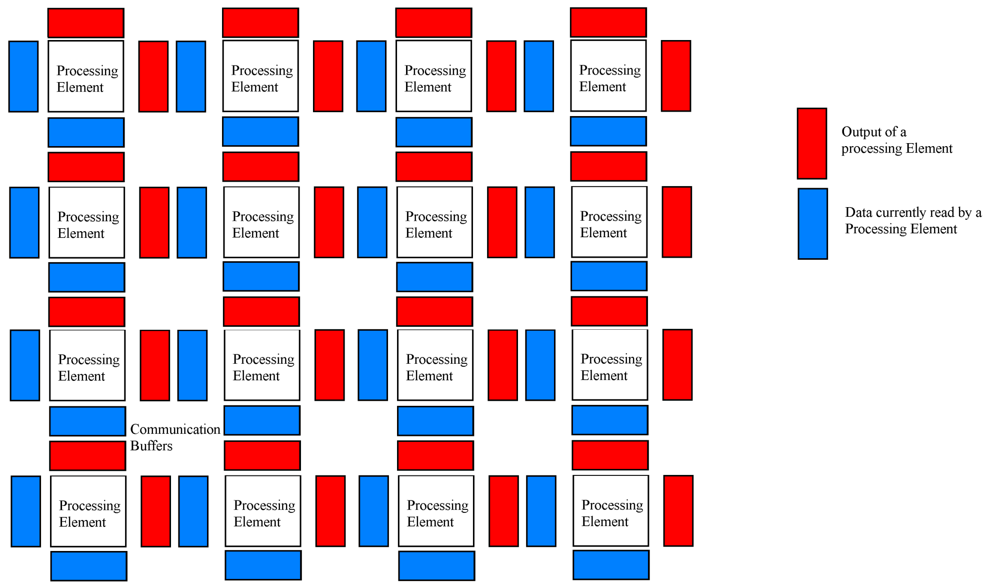

FIGURE 5.2. PE communication state1

is used to select the memory and row address is used to select the corresponding memory location.

5.4. Verilog Design

We will now discuss in detail the Verilog implementation of the Jacobi-2D hardware accelerator. Each PE loads its memories with values from the boundary. Then, it loads the values into the MAC unit which performs a multiply add. There is a pipeline depth of 6 in our design. We have a pipeline depth of 6 because we have two multiplications that takes 2 cycles and 4 additions that take 4 cycles. There are two multipliers each one of which takes 2 cycles. But since this is done in parallel, we have a pipeline depth of 6. Each PE is allocated a tile of size x*y.

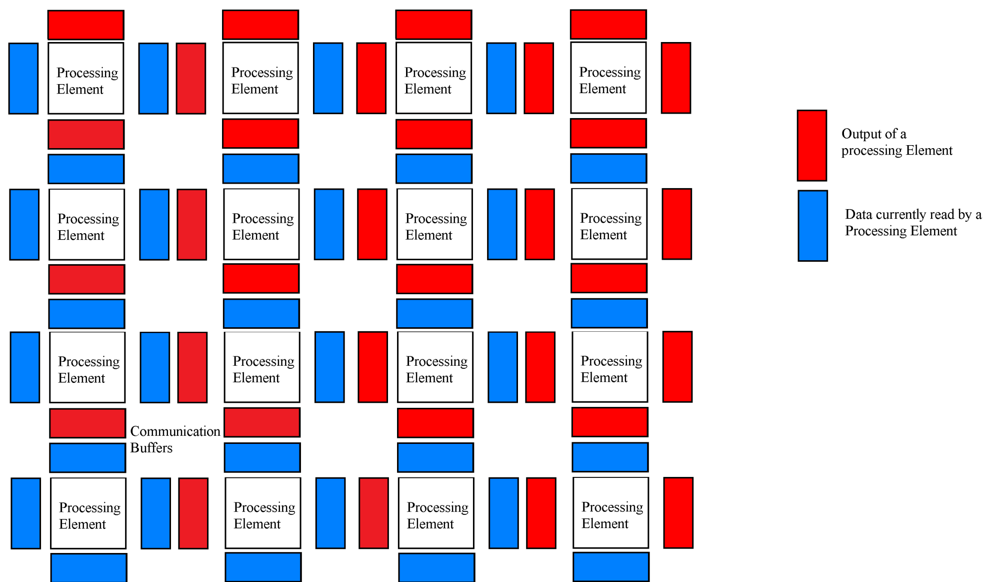

FIGURE 5.3. PE communication state2

```
if (j1>1 && j1<y && i1>1 && i1<x)
begin
a1[2]<=q0[h];
b1[2]<=q0[h+1];
c1[2]<=q0[j2];
a1[1]<=a1[2];
b1[1]<=b1[2];
c1[1]<=c1[2];
a1[0]<=a1[1];
b1[0]<=b1[1];
c1[0]<=c1[1];
end
```

As we can see from the code snippet, We have 3 shift registers that read 3 new values every cycle. Some of these values are shifted as we move through the iteration space. When we reach the end, we refresh the values corresponding to the next row. We read 9 new values at the beginning of the iteration. Five out of 9 registers are fed into the MAC. The result of the computation is available after 6 cycles. So, we place a restriction on minimum tile size

to be 6. We start the computation at the first point in the tile (0,0) and then we move along i direction. When we reach the end we move to the next row. When we reach the end, we move to the next plane. We have two extra rows of memories. We use these memories to avoid overwriting of any value before its last read. This happens because, the value that is currently being computed is 6 cycles later but the last read of the value is two rows later.

```
genvar n;
generate for (n=1; n<=h+2; n=n+1) begin : ms
m1 v0(.data(data),.waddr(wadd[n-1]),.raddr(radd[n-1]),.clk(clk),.count(count)
   ,.q0(q0[n-1]),.we(wen[n-1]));
end
endgenerate
```

Outputs from a PE are sent to the neighboring PE as they are produced. Therefore the boundary values from a PE will be available 6 cycles later to the next PE. As we discussed earlier, each PE has to be able to store (w+2) * (h+2) values in a PE. In order to accomplish that, we use a generate for statement that synthesizes h+2 memories each one of size w+2. I built a memory controller that controls which values are being read from the memory and which values are written onto it. Our processor allocation involves assigning a pass to a grid of PEs. Therefore, I used a similar generate for sequence to generate a grid of PEs in which neighboring PEs can communicate to each other. As the memory controller determines which memories to read from, it also determines which memory location to write into. In order to ensure safe writes and to make sure that values are not accessed while writing, we add 2 extra memories which is used to store data. This ensures safe reads and writes because the lifetime of a value is two rows.

We designed the memory controller in a way that it would read and write from memories that are separated by 2. So, if I am currently reading from row 2, I would be writing into row 0. Also, it works perfectly while reading data from adjacent memories. Since they would

have the right values written onto them. There is a communication module that performs inter PE communication. Now, PEs write values to the boundary of the PE to the right at different times compared to the PE above it. So, each PE communicates at certain times specified by the memory controller based on what value is being read and at what times. Now, because of the pipeline delay of 6, the PE above has to start 6 cycles late for all its inputs to be available. Now, the PE to the right can start at the same time because its inputs become available as they are produced from the previous PE. We also place a restriction on the width of a tile to be 4. This restriction is made due to design constraints and if the tile height is less than 4 we still have to add 2 extra memories, perform communication and so on which makes the overhead substantial making it a worse design. These overheads include 2 extra rows of memories and communications to neighboring PEs. The delay of 6 for the PEs above occurs only once at start. So, in steady state, all PEs are computing one value each cycle.

5.5. Design Verification

We did a functional verification and timing verification on our design. The functional verification was done by comparing the results of the Verilog program with the results from a C program. We found the results to be accurate. We did functional verification on different design parameter values and problem sizes and verified each case. To do timing verification, we calculated the total number of cycles to compute one pass of computation analytically. We also calculated the total number of cycles to compute one pass of computation by the Verilog design. We found out that the number of cycles predicted by the analytical model is exactly the same as the total number of cycles required by the hardware accelerator. We did not consider the cycles to fill FPGA memories with initial values.

Table 5.1. Timing Verification

px	py	x	y	Analytical output(cycles)	Verilog output(cycles)
16	16	4	4	16096	16096
12	12	8	8	64072	64072
12	14	6	8	48084	48084
12	14	6	6	36084	36084
12	14	6	4	24084	24084
14	12	6	6	36072	36072
14	12	8	6	48072	48072
14	12	8	8	64072	64072
16	12	8	6	48072	48072
16	12	8	8	64072	64072
16	14	4	4	16084	16084
16	16	6	6	36096	36096
16	16	10	10	100096	100096
12	12	14	14	196072	196072
12	14	12	14	168084	168084
12	14	12	12	144084	144084
12	14	12	10	120084	120084
14	12	12	12	144072	144072
14	12	14	12	168072	168072
14	12	14	14	196072	196072
16	12	14	12	168072	168072
16	12	14	14	196072	196072
16	14	10	10	100084	100084
16	16	12	12	144096	144096
16	18	4	4	16108	16108
16	20	8	8	64120	64120
16	22	6	8	48132	48132
16	24	6	6	36144	36144
18	16	6	4	24096	24096
18	18	6	6	36108	36108
18	20	8	6	48120	48120
18	22	8	8	64132	64132
18	24	8	6	48144	48144

CHAPTER 6

Optimization Problem

In this chapter, we formulate an optimization problem that minimizes the execution time of the program subject to bandwidth and area for wave 2D and Jacobi 2D algorithm.

6.1. Wave 2D optimization problem

We formulate the optimization problem corresponding to wave 2D algorithm. In this section, we formulate analytical models for execution time, bandwidth and area.

6.1.1. Bandwidth equation.

We formulate the analytical models for Bandwidth equation. There are two different BW equations.

(1) Inter PE bandwidth
(2) offchip-PE bandwidth

Number of cycles required to compute one tile is $x * y * z$. The data needed is,

$$Data_off = 2 * y * z + 2 * x * z + x * y \tag{6.1.1}$$

We store $x * y$ initial values of the array in each processing element(PE). The amount of off-chip data needed for the computation is given by,

$$Data_off = 2 * y * z + 2 * x * z \tag{6.1.2}$$

6.1.2. Inter PE Bandwidth.

The Inter PE bandwidth is given by,

$$BW_{PE} = \frac{2 * y * z + 2 * x * z}{x * y * z} \tag{6.1.3}$$

Simplifying the above equation, we get

$$BW_{PE} = \frac{2}{x} + \frac{2}{y} \tag{6.1.4}$$

6.1.3. Offchip-PE Bandwidth. Off chip-PE bandwidth represents the amount of data that has to be transferred from off chip to the hardware accelerator. The off-chip to PE bandwidth computed in this section does not involve the filling up of initial x*y values in each tile. The bandwidth computed here involves the transfer of boundary values to the boundary processing elements(PEs) from off chip memory in steady state. The processing elements(PEs) in the hardware accelerator operate in parallel. Therefore boundary values corresponding to one plane of tiles has to be transferred in xyz cycles. As discussed earlier, x*y*z is the total number of cycles required by one PE to compute its tile. The total amount of data needed from off-chip memory depends on the total number of boundary PEs. Total amount of data needed from the left along i direction is given by,

$$Data_i = 6 * y * p_y * z \tag{6.1.5}$$

Total amount of data needed from the front along j direction is given by,

$$Data_j = 6 * x * p_x * z \tag{6.1.6}$$

The amount of data needed from below is not taken into account because it is being reused by the same PE. Also, the data is being read and written from/to the communication

buffers. So, the amount of data is multiplied by a factor of 2. Total amount of data needed is given by,

$$Data = 12 * y * p_y * z + 12 * x * p_x * z \tag{6.1.7}$$

Total number of cycles is given by,

$$Cycles = x * y * z \tag{6.1.8}$$

$$BW_{off} = \frac{12 * y * p_y * z + 12 * x * p_x * z}{x * y * z} \tag{6.1.9}$$

$$BW_{off} = \frac{12 * y * p_y + 12 * x * p_x}{x * y} \tag{6.1.10}$$

$$BW_{off} = \frac{12 * p_y}{x} + \frac{12 * p_x}{y} \tag{6.1.11}$$

Note that both inter PE bandwidth and Off-chip to PE bandwidth are independent of z (tile size in t direction).

6.1.4. Execution Time. Execution time represents the execution time of the whole program. In order to compute the execution time of the program, we first compute the

execution time of a pass and multiply the value by the total number of passes. Execution Time of a pass assuming concurrent start by all processing elements (PEs) and a throughput of one value every cycle is given by,

$$T_{pass} = x * y * T \tag{6.1.12}$$

But, when we simulated a grid of PEs using verilog, we found that there is a delay equal to the pipeline depth for every PE in the y direction. So, each PE in y direction starts d cycles later than the previous row of PEs. So, a more accurate version of execution time of one pass is given by,

$$T_{pass} = x * y * T + d * p_y \tag{6.1.13}$$

The total number of passes is given by,

$$N_{pass} = \frac{M}{x * p_x} * \frac{N}{y * p_y} \tag{6.1.14}$$

The total execution time is given by,

$$T_{tot} = (x * y * T + d * p_y)(\frac{M}{x * p_x} * \frac{N}{y * p_y}) \tag{6.1.15}$$

If we have a pipeline depth of 6, we get

$$T_{tot} = (\frac{M * N * T}{p_x * p_y}) + (\frac{6 * M * N}{x * p_x * y}) \tag{6.1.16}$$

6.1.5. Area Model. In this section, we develop area models for our architecture. The total area can be divided into two different components.

(1) Compute units area
(2) Memory area

Area of compute units is given by a multiple of the total number of PEs.

$$C_{Area} = \alpha * p_x * p_y \tag{6.1.17}$$

Area of memory units is given by,

$$M_{Area} = \beta * p_x * p_y * (x * y + 6 * x * z + 6 * y * z) \tag{6.1.18}$$

Total area is,

$$T_{Area} = \alpha * p_x * p_y + \beta * p_x * p_y * (x * y + 6 * x * z + 6 * y * z) \tag{6.1.19}$$

Based on the BW model, Execution time and Area Model, we formulate the optimization problem as follows:

$$\text{Min } T_{total} = (\frac{M * N * T}{p_x * p_y}) + (\frac{6 * M * N}{x * p_x * y}) \text{ subject to,} \tag{6.1.20}$$

$$\frac{12 * p_y}{x} + \frac{12 * p_x}{y} \leq B_{max} \tag{6.1.21}$$

$$\alpha * p_x * p_y + \beta * p_x * p_y * (x * y + 6 * x * z + 6 * y * z) \leq A_{max} \tag{6.1.22}$$

6.1.6. Solving the Optimization Problem. We solved the optimization problem using Excel solver and Yalmip toolbox in MATLAB. Excel uses GRC Nonlinear method to solve the optimization problem. In order to solve the problem using matlab, we used two properties called posynomials and positivity.

A function f is called a posynomial function of x if it has the form,

$$f(x1, x2, ..., xn) = c_k * {x_1}^{a_1 k} + c_k * {x_2}^{a_2 k} + ... \tag{6.1.23}$$

c_k must be greater than or equal to 0. a_k muse be real. So, in other words, all the coefficients must be positive and the powers can be any real number including negative or fractions. If we look closely at our optimization problem, we realize that all three equations follow this criteria. Positivity is a property that ensures that all the terms and coefficients in an equation are positive. We can solve such optimization problems using Yalmip toolbox in MATLAB.

We have an optimization problem involving 4 variables x,y,p_x and p_y. We considered a reasonable A_{max} and B_{max} and solved the problem using excel and MATLAB. Excel Solver solves the problem using GRG Non-Linear method. We found results from MATLAB to be consistently optimal. We have shown the results in a table below.

TABLE 6.1. Excel Solver Results for wave 2D

BW	px	py	x	y	px*py	Execution Time(cycles)
20	10	5	8	10	50	5.00E+08
22	8	7	9	8	56	4.47E+08
24	10	6	7	9	60	4.17E+08
26	8	8	7	8	64	3.91E+08
28	11	6	6	9	66	3.79E+08
30	9	8	8	6	72	3.48E+08
32	9	8	8	6	72	3.48E+08
34	10	8	6	7	80	3.13E+08
36	9	9	6	6	81	3.09E+08
38	11	8	6	6	88	2.84E+08
40	11	8	6	6	88	2.84E+08

TABLE 6.2. MATLAB Results for wave 2D

BW	px	py	x	y	px*py	Execution Time(cycles)
20	10	5	9	9	50	5.00E+08
22	8	7	8	9	56	4.47E+08
24	10	6	8	8	60	4.16E+08
26	8	8	8	7	64	3.91E+08
28	11	6	8	7	66	3.79E+08
30	12	6	6	8	72	3.47E+08
32	12	6	7	7	72	3.47E+08
34	10	8	7	6	80	3.13E+08
36	9	9	6	6	81	3.09E+08
38	11	8	6	6	88	2.84E+08
40	11	8	6	6	88	2.84E+08

As we can see from the table above, the results from MATLAB are slightly better. The values of BW varies from 20 to 40 in both tables. We computed these bandwidth values using normal FPGA BW and frequency values. The area of 500000 is taken from cyclone V FPGA with appropriate coefficient values.

6.2. Jacobi 2D optimization problem

We formulate the optimization problem corresponding to jacobi 2D algorithm. In this section, we formulate analytical models for execution time, bandwidth and area.

6.2.1. Bandwidth Model.

Now, we formulate the analytical models for Bandwidth equation. There are two different BW equations.

(1) Inter PE bandwidth

(2) offchip-PE bandwidth

Number of cycles required to compute one tile is x * y * z. Amount of data needed for the computation is

$$Data = 2 * y * z + 2 * x * z \tag{6.2.1}$$

6.2.2. Inter PE Bandwidth.

The Inter PE bandwidth is given by,

$$BW = \frac{2 * y * z + 2 * x * z}{x * y * z} \tag{6.2.2}$$

Simplifying the above equation, we get

$$BW = \frac{2}{x} + \frac{2}{y} \tag{6.2.3}$$

6.2.3. Execution Time. Execution time represents the execution time of the whole program. In order to compute the execution time of the program, we first compute the execution time of a pass and multiply the value by the total number of passes. Execution Time of a pass assuming concurrent start by all processing elements (PEs) and a throughput of one value every cycle is given by,

$$T_{pass} = x * y * T \tag{6.2.4}$$

But, when we simulated a grid of PEs using verilog, we found that there is a delay equal to the pipeline depth for every PE in the y direction. So, each PE in y direction starts d cycles later than the previous row of PEs. So, a more accurate version of execution time of one pass is given by,

$$T_{pass} = x * y * T + d * p_y \tag{6.2.5}$$

The total number of passes is given by,

$$N_{pass} = \frac{M}{x * p_x} * \frac{N}{y * p_y} \tag{6.2.6}$$

The total execution time is given by,

$$T_{tot} = (x * y * T + d * p_y)(\frac{M}{x * p_x} * \frac{N}{y * p_y}) \tag{6.2.7}$$

If we have a pipeline depth of 6, we get

$$T_{tot} = (\frac{M * N * T}{p_x * p_y}) + (\frac{6 * M * N}{x * p_x * y}) \tag{6.2.8}$$

6.2.4. Area Model. Now, we develop area models or our architecture. The total area of the chip can be divided into two different components.

(1) Compute units area
(2) Memory area

Area of compute units is given by a multiple of the total number of PEs.

$$C_{Area} = \alpha * p_x * p_y \tag{6.2.9}$$

Area of memory units is given by,

$$M_{Area} = \beta * p_x * p_y * (x * y + 2 * x * z + 2 * y * z) \tag{6.2.10}$$

Total area is,

$$T_{Area} = \alpha * p_x * p_y + \beta * p_x * p_y * (x * y + 2 * x * z + 2 * y * z) \tag{6.2.11}$$

Based on the BW model, Execution time and Area Model, we formulate the optimization problem as follows:

$$\text{Min } T_{total} = (\frac{M*N*T}{p_x*p_y}) + (\frac{6*M*N}{x*p_x*y}) \text{ subject to,} \tag{6.2.12}$$

$$\frac{2*p_y}{x} + \frac{2*p_x}{y} \leq B_{max} \tag{6.2.13}$$

$$\alpha*p_x*p_y + \beta*p_x*p_y*(x*y+2*x*z+2*y*z) \leq A_{max} \tag{6.2.14}$$

6.2.5. Solving the Optimization Problem. We solved the optimization problem using Excel solver and Yalmip toolbox in MATLAB. We have an optimization problem involving 4 variables x,y,p_x and p_y. We considered a reasonable A_{max} and B_{max} and solved the problem using excel and MATLAB. Excel Solver solves the problem using GRG Non-Linear method. We found results from MATLAB to be consistently optimal. We have shown the results in a table below.

The values of BW varies from 10 to 30 in both tables. We computed these bandwidth values using normal FPGA BW and frequency values. The area of 500000 is taken from cyclone V FPGA with appropriate coefficient values. As we can see in this case, Area is more constraining than Bandwidth. It is because we require less bandwidth due to less data required from off chip.

```
sdpvar xi xj xk pj pk
A = 200.*pj.*pk+300.*pk.*xk.*pj.*xj+100.*(pj.*xj+pk.*xk);
B = 2.*(pj./xk)+12.*(pk./xj);
C1= -(xi.*xj)./6;
C2= (xi.*xj)./6;
```

Table 6.3. Excel Solver Results for jacobi 2D

BW	px	py	x	y	px*py	Execution Time(cycles)
10	11	16	6	5	176	1.42E+08
12	13	16	6	4	208	1.21E+08
14	10	22	6	3	220	1.14E+08
16	10	25	6	3	250	1.01E+08
18	8	34	7	2	272	9.33E+07
20	11	27	6	2	297	3.48E+08
22	13	24	6	2	312	8.11E+07
24	13	24	6	2	312	8.11E+07
26	13	24	6	2	312	8.11E+07
28	13	24	6	2	312	8.11E+07
30	13	24	6	2	312	8.11E+07

Table 6.4. MATLAB Results for jacobi 2D

BW	px	py	x	y	px*py	Execution Time(cycles)
10	11	16	6	5	176	1.42E+08
12	13	16	6	4	208	1.21E+08
14	10	22	6	3	220	1.14E+08
16	10	25	6	3	250	1.01E+08
18	8	34	7	2	272	9.33E+07
20	10	30	6	2	297	3.48E+08
22	13	24	6	2	312	8.11E+07
24	13	24	6	2	312	8.11E+07
26	13	24	6	2	312	8.11E+07
28	13	24	6	2	312	8.11E+07
30	13	24	6	2	312	8.11E+07

```
C3= (xi.*xj);

% Constraints
F = [xi >= 1, xj >= 1, xk >= 1, pj >= 1, pk >= 1, C3 >= 6, B <= 30, A <=
    50000];

% Objective
D = min((5000000.*(5000+3.*xk.*pk+3.*pj.*xi))./(pj.*pk));

% Solve!
optimize([F, integer(xi), integer(xj),integer(xk), integer(pj),integer(pk)],D)
```

```
value(pj)
value(pk)
value(xi)
value(xj)
value(xk)
value(D)
```

The above code is a snippet that can optimize a posynomial and produce the output values of tile sizes.

CHAPTER 7

Conclusion and Future work

The goal of this thesis is to present the design flow for hardware acceleration of 2D stencil computations. In this thesis, we designed an FPGA-based hardware accelerator that efficiently implements the Jacobi 2D method and wave 2D method. We built the accelerator in a way that we achieve high performance with low Bandwidth and area requirements. The Optimal design was achieved using several steps. The first step is to analyze the nature of dependencies and using a multi-level Tiling approach. The second step is to create a timestamp for all the points in the iteration space. The third step is to create hardware that meets all the design requirements. The fourth step is to verify the functionality of the hardware generated by Verilog. This is done by comparing the results produced by Verilog code with a C program. The fifth step involves solving the optimization problem formulated to minimize the execution time subject to bandwidth and Area constraints to compute the optimal values of our design parameters x,y,p_x and p_y.

The design space exploration from this thesis can be applied to any other stencil. This thesis can guide future developments in the area of hardware accelerators. Our approach could be altered slightly to come up with good implementations of other stencils. The task of finding a analytical solution is tricky because of the number of parameters that needs to be solved. In applications with special nature like having square tiles, we could come up with an analytical solution to the optimization problem. The hardware implementation is quite complicated due to multiple levels of tiling and multiple design parameters. Our current hardware implementation could be automated for all stencils which would make the design scalable for any 2D-stencil.

Bibliography

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.