2010

Scalable, Memory-Intensive Scientific Computing on Field Programmable Gate Arrays

Salma Mirza
University of Massachusetts Amherst

SCALABLE, MEMORY-INTENSIVE SCIENTIFIC COMPUTING ON FIELD PROGRAMMABLE GATE ARRAYS

A Thesis Presented

by

SALMA MIRZA

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL AND COMPUTER ENGINEERING
February 2010

Department of Electrical and Computer Engineering

SCALABLE, MEMORY-INTENSIVE SCIENTIFIC COMPUTING ON FIELD PROGRAMMABLE GATE ARRAYS

A Thesis Presented

by

SALMA MIRZA

Approved as to style and content by:

_____
Russell Tessier, Chair

_____
Blair Perot, Member

_____
Do-Hoon Kwon, Member

_____
C. Hollot, Department Head
Department of Electrical and Computer Engineering

FOR RYAN CHRISTOPHER JOHNSON

ACKNOWLEDGMENTS

I thank my advisor Professor Russell Tessier for teaching me Reconfigurable Computing, letting me work on a thesis under his supervision, motivating and guiding the research. I thank Professor Blair Perot for introducing me to Sparse Matrices, and for his help and enthusiasm in driving the work. I thank Professor Do-Hoon Kwon for serving on my thesis committee. I thank all my lab-mates at the Reconfigurable Computing Group, particularly Jia Zhao for helping me select the FPGA Boards for my thesis and Emmanuel Seguin for helping me debug timing issues. I thank Julie Staraitis, the technical manager during my coop at Advanced Micro Devices who unwittingly became my role model for a woman in engineering besides training me to be a circuit design engineer. I thank Steve Fundakowski of Student Affairs for letting me take days off work to work on my thesis. I would like to thank my parents for setting me an example of working hard throughout. I thank Ryan Johnson for teaching me how to write state machines in Verilog, and for his immeasurable help, encouragement and support. Lastly, I would also like to thank our cat Kitty for staying up late nights when I was working even if he did not have to.

ABSTRACT

SCALABLE, MEMORY-INTENSIVE SCIENTIFIC COMPUTING ON FIELD PROGRAMMABLE GATE ARRAYS

FEBRUARY 2010

SALMA MIRZA, B.S, MUMBAI UNIVERSITY

M.S.E.C.E., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Russell Tessier

Cache-based, general purpose CPUs perform at a small fraction of their maximum floating point performance when executing memory-intensive simulations, such as those required for many scientific computing problems. This is due to the memory bottleneck that is encountered with large arrays that must be stored in dynamic RAM. A system of FPGAs, with a large enough memory bandwidth, and clocked at only hundreds of MHz can outperform a CPU clocked at GHz in terms of floating point performance. An FPGA core designed for a target performance that does not unnecessarily exceed the memory imposed bottleneck can then be distributed, along with multiple memory interfaces, into a scalable architecture that overcomes the bandwidth limitation of a single interface. Interconnected cores can work together to solve a scientific computing problem and exploit a bandwidth that is the sum of the bandwidth available from all of their connected memory interfaces. The implementation demonstrates this concept of scalability with two memory interfaces through the use of available FPGA prototyping platforms. Even though the FPGAs operate at 133 MHz, which is twenty one times slower than an AMD Phenom X4 processor operating at 2.8 GHz, the system of two FPGAs performs eight times slower than the processor for the example problem of SMVM in heat transfer.

However, the system is demonstrated to be scalable with a run-time that decreases linearly with respect to the available memory bandwidth. The floating point performance of a single board implementation is 12 GFlops which doubles to 24 GFlops for a two board implementation, for a gather or scatter operation on matrices of varying sizes.

TABLE OF CONTENTS

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER 1

INTRODUCTION

Simulations allow scientists to quantitatively predict results of real-life phenomena for a range of input conditions and with a programmable degree of accuracy. In many cases, simulations are preferred to physical experiments because they are often cheaper, faster and less dangerous than these types of experiments. For a reasonably good mathematical model, the accuracy of the simulations is given by how closely a simulation set up can imitate a physical experimental set up. To increase accuracy, the problem must be made larger. This translates to an increase in the number of computations, which in turn is constrained by the available computing resources and their efficiency.

Low cost commodity computers are often used in clusters for scientific simulations. Commodity computers utilize a cache-based architecture which is ill-suited for scientific computations. Scientific computing performs poorly on cache-based CPUs because of the vast and constantly changing data associated with iterative simulations. The data is too big to fit on the CPU cache, and exhibits little temporal locality making cache hits rare. The speed of computation is limited by memory access times that, typically, are at least ten times slower than the time taken to perform an operation on the CPU.

This problem is particularly apparent in sparse matrix vector multiplication (SMVM). SMVMs are of the type: y += A.x; where x and y are dense vectors and A is

sparse matrix. SMVM often requires memory intensive operation. Often, the matrix A is extremely sparse and unstructured. For a dynamic problem, the matrix would change with time and would need to be rebuilt at every step. Construction and storage of a sparse matrix during computations is difficult, expensive and unnecessary. If constructed at all, the sparse matrix should be stored in an alternate representation – either a compressed row or compressed column representation, whichever may be more appropriate for the matrix at hand [1].

However, given the elegance with which most problems can be represented in matrix formats, mathematical journals present algorithms in terms of matrices. Engineers implementing these algorithms for their simulation purposes, inevitably write codes that create matrices and operate on them. Computer codes build these matrices that slow down simulations because of the memory access times involved. This ultimately restricts the accuracy at which these simulations can be performed in a constrained time period.

In Chapter 2, we explain the sparse matrix vector multiplication problem that arises in an iterative simulation of two dimensional heat transfer. We quantify results of the expected performance of the heat transfer simulation on a CPU. In Chapter 3, we review the previous work that has been done in this area, how our work relates to it, and highlight the differences. In Chapter 4, we explain our implementation algorithm for gather and the scatter operations and go through the implementation details of our algorithm on Field Programmable Gate Arrays (FPGAs). In Chapter 5, we calculate the

expected performance of SMVM on a system of FPGAs and then compare it with the measured performance. In Chapter 6, we conclude the thesis and discuss potential work.

CHAPTER 2

SPARSE MATRIX VECTOR MULTIPLICATION

2.1 Specific Example of SMVM

Throughout this document, we shall consider heat transfer as an example of SMVM. However, the implementation holds true for any SMVM that arises from any PDE solution. Consider the problem of two dimensional heat transfer on an unstructured mesh as shown in Figure 1.

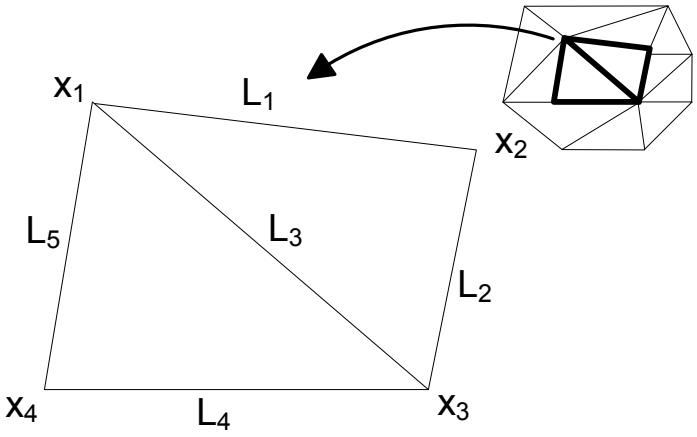

Figure 1: Discretization of PDE over an unstructured (triangular) mesh

To discretize the partial differential equations associated with this problem, the mesh is divided into smaller domains. In practice the mesh may consist of 100,000 sub-domains for a 2-D problem, and a million tetrahedras for a 3-D problem. Dividing the mesh into smaller sub domains results in a more accurate solution, but also involves more data and intensified computation. For the purpose of simplification, we consider two triangles that are a part of a larger mesh. The temperature unknowns are located at the four vertices and are calculated iteratively in two steps as discussed below.

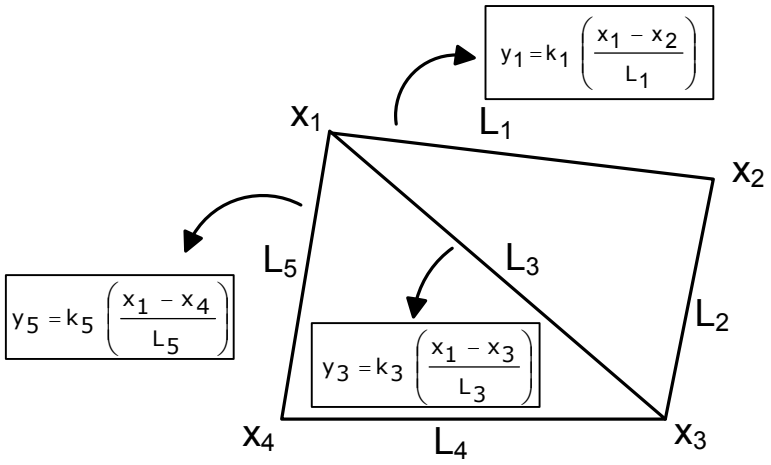

Figure 2: The Gradient Operation

Step 1: The Gradient or the Gather Operation

The first step to solve the heat transfer problem is to calculate the temperature gradient along the edges (Figure 2). Roughly, this is given by the difference in temperature at the vertices that connect an edge divided by the length of the edge. This is the gradient operation and in a matrix form, is given by $y = Gx$

$$\begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \end{bmatrix} = \begin{bmatrix} \frac{1}{L_1} & -\frac{1}{L_1} & 0 & 0 \\ 0 & \frac{1}{L_2} & -\frac{1}{L_2} & 0 \\ \frac{1}{L_3} & 0 & -\frac{1}{L_3} & 0 \\ 0 & 0 & \frac{1}{L_4} & -\frac{1}{L_4} \\ \frac{1}{L_5} & 0 & 0 & -\frac{1}{L_5} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix}$$

Matrix G is a sparse matrix. Every row of G contains exactly two non-zero items. For simplicity, we shall refer to non-zero values as items. In relation to the problem at

hand, the number of columns in the matrix depends on the number of nodes (or the number of unknowns), and the number of rows depends on the total number of edges. In general, the number of rows and columns is not equal. The location of items in a row indicates the nodes connected to that edge. The value of the item represents the length of that edge. The values in a row are repeated except for a minus sign. For example, in row 1, the items are located in column 1 and column 2, indicating that edge 1 is between nodes x_1 and x_2 and has length L_1.

Each row has two items even when the number of unknowns is 100000, so the matrix continues to remain extremely sparse as it increases in size. As a result, it is best if this matrix is not constructed to include all or most of the points. The construction and storage of such a sparse matrix is unnecessary and adds to the computational overhead. If a matrix is constructed at all, it is best stored in a compressed format. Matrix G can be stored in a compressed row format as follows:

$$\begin{bmatrix} L_{i1} & -L_{i1} & 0 & 0 \\ 0 & L_{i2} & -L_{i2} & 0 \\ L_{i3} & 0 & -L_{i3} & 0 \\ 0 & 0 & L_{i4} & -L_{i4} \\ L_{i5} & 0 & 0 & -L_{i5} \end{bmatrix}$$

L_i represents the inverse of length L. The above matrix becomes:

Multiplier	[	L_{i1}	$-L_{i1}$	L_{i2}	$-L_{i2}$	L_{i3}	$-L_{i3}$	L_{i4}	$-L_{i4}$	L_{i5}	$-L_{i5}$]
Destination	[	1	2	2	3	1	3	3	4	1	4]
Pointer	[	0	2	4	6	8	]				
Number	[	2	2	2	2	2	]				

The Multiplier array contains the non-zero values in the matrix obtained by traversing each row in the matrix from column 1 to column 4. The column to which the non-zero element belongs is stored in the Destination array. The pointer to the first multiplier value in each row is stored in the Pointer array. The Pointer array thus indicates the beginning of a new row. The number of non-zero elements in each row is stored in the "Number" Array. In case a row has all zero elements in it, the Number Array would indicate this.

Given the regularity the matrix displays in each row, it is best stored in the compressed row format. When stored in the compressed row format, the Pointer array and the Number Array are unnecessary and can be constructed on the fly.

To solve for y, without generating an explicit (sparse) matrix for the gradient operation, we construct the Edge to Node [E2N] data structure which holds the connectivity information for the mesh. The E2N structure contains the pointer information specifying which two nodes define each edge. In this sense, the E2N data structure is equivalent to compressed row sparse matrix because they hold the same connectivity information. The E2N data structure for the problem in consideration is given by:

E2N=			Edge1	Edge2	Edge3	Edge4	Edge5	
	Node1	[	x1	x2	x1	x3	x1	]
	Node2		x2	x3	x3	x4	x4	

Then in pseudo-code, the gradient operation can be implemented with a single line (and without generating an explicit matrix) as:

Equation 2.1

```
for (e = 0; e < num_edges; e++) {
y[e] = ( x[E2N[2,e]] – x[E2N[1,e]] ) * (1/L[e]); }
```

In the above operation, indirect memory reads of the type x=a[i] are performed in which the value of *x* is "gathered" from the array *a* at address *i*. Henceforth, we refer to the gradient operation as the **"gather operation"**

Step 2: The Discrete Divergence or the Scatter Operation

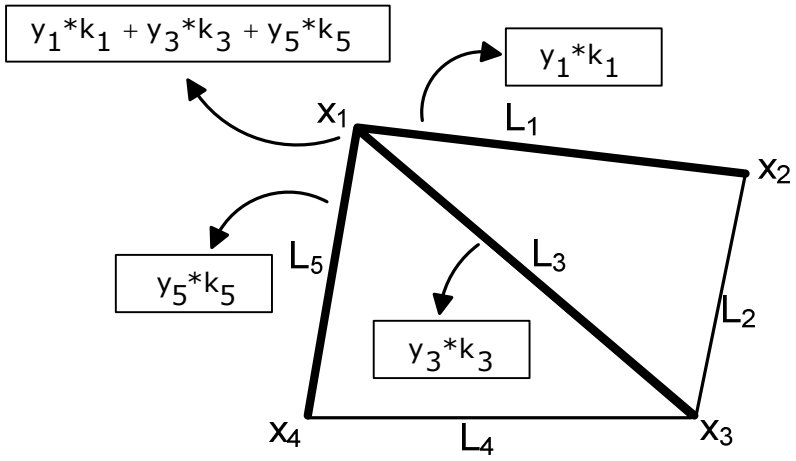

Figure 3: The Discrete Divergence Operation

The second step to solve this problem is to multiply the gradient on each edge ('y' as obtained in the first step) by the conductivity along the edge (k), to obtain the flux along each edge. The fluxes associated with each edge attached to a node are then summed up, to obtain the temperature unknown at the node. This is the Discrete Divergence Operation and in matrix form is given by *z=Dy*

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \\ z_4 \end{bmatrix} = \begin{bmatrix} k_1 & 0 & k_3 & 0 & k_5 \\ -k_1 & k_2 & 0 & 0 & 0 \\ 0 & -k_2 & -k_3 & k_4 & 0 \\ 0 & 0 & 0 & -k_4 & -k_5 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ y_4 \\ y_5 \end{bmatrix}$$

Matrix D is also a sparse matrix. The sparsity pattern of Matrix D is the transpose of the sparsity pattern of Matrix G. Every column of D contains exactly two items. In relation to the problem, the number of rows in the matrix depends on the number of nodes (or the number of unknowns), and the number of columns depends on the total number of edges. The location of items in a column indicates the nodes connected to each edge. The value of the item represents the conductivity of each edge. The values in a column are repeated except for a minus sign. For example, in column 1, the items are located in row 1 and row2, indicating that edge 1 is between nodes x_1 and x_2 and has conductivity k_1.

Each column has two items, as with the rows of G, even when the number of unknowns is 100000 so the matrix continues to remain extremely sparse as it increases in size. It is again best to not construct this matrix at all, and if it is constructed, it is best stored in the compressed column format, as opposed to the compressed row format. Matrix D can be stored in a compressed column as follows:

$$\begin{bmatrix} k_1 & 0 & k_3 & 0 & k_5 \\ -k_1 & k_2 & 0 & 0 & 0 \\ 0 & -k_2 & -k_3 & k_4 & 0 \\ 0 & 0 & 0 & -k_4 & -k_5 \end{bmatrix}$$

becomes

Multiplier	[	k_1	$-k_1$	k_2	$-k_2$	K_3	$-k_3$	k_4	$-k_4$	k_5	$-k_5$]
Destination	[	1	2	2	3	1	3	3	4	1	4]
Pointer	[	0	2	4	6	8	]				
Number	[	2	2	2	2	2	]				

The Multiplier array contains the non-zero values in the matrix obtained by traversing each column in the matrix from row 1 to row 5. The row to which the non-zero element belongs is stored in the Destination array. The pointer to the first multiplier value in each column is stored in the Pointer array. The Pointer array thus indicates the beginning of a new column. The number of non-zero elements in each column is stored in the "Number" Array. In case a column has all zero elements in it, the Number Array would indicate this.

Given the regularity the matrix displays in each column, it is best stored in the compressed column format. When stored in the compressed column format, the Pointer array and the Number Array are unnecessary and can be constructed on the fly.

This operation can be efficiently implemented using the same E2N connectivity structure (and no explicit matrix). In this case, the E2N data structure is equivalent to using a compressed column matrix format.

Then in pseudo-code, the discrete divergence operation can be implemented without generating an explicit matrix as:

Equation 2.2

```
z = 0;
for (e = 0, e < num_edges, e + +) {
z[E2N[1, e]] + = y[e] * k[e];
z[E2N[2, e]] − = y[e] * k[e];
}
```

In the above operation, indirect memory writes of the type a[i]=x are performed in which the value of *x* is "scattered" to the array *a* at address *i*. Henceforth we refer to the discrete divergence operation as the **"scatter operation"**

Both the Gradient and the Divergence matrix can be efficiently represented using the same E2N data structure. For the gradient operation, the E2N structure is equivalent to compressed row sparse matrix storage and in the divergence operation it is equivalent to compressed column sparse matrix storage. This mixed representation of the matrices (both row and column format, whichever is appropriate to the structure of the matrix) saves memory in addition to improving code efficiency.

Now that we have found the most efficient representations for the gradient and the divergence operators, in the next Section 2.2 of Chapter 2, we analyze the efficiency of a CPU in implementing the gather and the scatter operation. Finally, in Section 2.3 of Chapter 2, we discuss the implementation of the same operations on a system of FPGAs with memory banks.

2.2 Specific Example of SMVM

Advances in CMOS technology depend upon the minimum feature size that can be fabricated on an integrated circuit. The minimum feature size decreases at a rate of over forty percent every two years [2]. For the semiconductor industry, this coarsely translates as doubling of the number of transistors on a chip every other year. The minimum feature size available at a particular time is referred to as a technology node. The channel capacitances in the MOS transistor depend primarily on the product of the width (W) and the length (L) of the transistor. In order to keep the W/L ratio constant for a circuit design, a decrease in L is followed by a decrease in W of the same magnitude. This effectively reduces the channel capacitances by a squared value of the ratio of decrease in gate length. This decrease in channel capacitance coupled with a decrease in threshold voltage creates transistors with faster switching times, reducing the delays in the critical paths and allowing the synchronous circuits to be clocked faster.

The increased performance of CPUs is primarily attributed to faster clock rates that piggy back on the decrease in transistor gate length. For every jump in CPU performance, eighty percent of the contributions can be attributed to a faster clock rate and twenty percent of the improvements can be attributed to changes in architecture [3].

Cache based CPU's continue to remain the norm for commodity computers. In a cache based CPU, for a memory-fetch operation, data from the adjacent memory locations will be fetched into the cache. This is because spatially coherent data is also assumed to be temporally coherent. While this may be true for most general purpose

applications, this is not true for sparse matrix vector multiplication, where data is not temporally coherent and is too vast to be stored on a cache. In this case the data has to be fetched from the main memory through indirect memory accesses which are limited by the speed of the memory itself. The DRAM memory speed does not scale at the same rate that processor speeds do. They both increase exponentially, but the difference between two exponentials also increases exponentially [4]. This disparity between the memory and the processor speeds is especially apparent for memory intensive applications where the speed of the application is highly dependent on the speed of the memory.

Without a significant change in memory technology, cache based memory access times will rely heavily on memory performance for applications that frequently reference memory. As an example, consider the performance of an AMD Phenom X4 920 Processor which operates at a frequency of 2.8 GHz, has a memory bandwidth of 6.8 GBps for sequential access and a theoretical floating point performance of 6.8 GFlops, for the SMVM problem discussed earlier. The memory bandwidth degrades to 20% of the sequential memory access bandwidth for random access which is 1.36GBps. The performance of a memory bandwidth constrained problem can be calculated using the Processor Balance and Application Balance Metrics [5][10].

The Processor Balance for the Xeon is calculated as:

$$\textbf{Processor Balance} = \frac{\textbf{Bandwidth (GBytes / sec)}}{\textbf{Peak Performance (GFlops)}} = \frac{\textbf{1.36 GBytes/sec}}{\textbf{6.8 GFlops}} = \textbf{0.2 Bytes/Flop}$$

Consider the application balance for the gather operation:

```
y[e] += (x[E2N[2,e]] - x[E2N[1,e]]) * Li[e]
```

Consider the memory requirement for the gather operation. This requires the following data from memory:

```
2 - 4 byte sequential read for E2N[1,e], E2N[2,e]
1 - 8 byte sequential read for Li[e]
3 - 8 byte random reads for x[E2N[1,e]], x[E2N[2,e]] and y[e]
1 – 8 byte random write for y[e]
```

Each gather operation requires 48 bytes of memory, including both sequential and random accesses, read and write operations. Since the slowest memory operation is random reads, we consider it exclusively for performance calculations. The gather operation requires 8 x 3 = 24 bytes to be randomly read from the external memory. Three floating point operations are performed.

$$\text{Application Balance}_{\text{gather}} = \frac{\text{Number of memory references}}{\text{Number of flops}} = \frac{24\ \text{Bytes}}{3\ \text{Flops}} = 8\ \text{Bytes/Flop}$$

The FPGA performs with a maximum theoretical performance of:

$$\text{Performance}_{\text{gather}} = \frac{0.2\ \text{Bytes/Flop}}{8\ \text{Bytes/Flop}} \times 6.8\ \text{GFlops} = 170\ \text{MFlops}$$

The FPGA performs at ***2.5%*** of its maximum floating point performance for a gather.

Consider the memory requirement for the scatter operation:

```
z[E2N[i, e]]+ = y[e] * k[e]
```

This requires the following data from memory:

```
1 - 4 byte sequential read for E2N[1,e] and E2N[2,e]
1 - 8 byte sequential read for Li[e]
2 - 8 byte random reads for z[E2N[i,e]] and y[e]
1 – 8 byte random write for y[e]
```

Each scatter operation requires 36 bytes of memory, including both sequential and random accesses, read and write operations. Out of these, 16 bytes have to be read randomly from the memory. Two floating point operations are performed.

$$\text{Application Balance}_{\text{scatter}} = \frac{\text{Number of memory references}}{\text{Number of flops}} = \frac{16\ \text{Bytes}}{2\ \text{Flops}} = 8\ \text{Bytes/Flop}$$

The FPGA performs with a maximum theoretical performance of:

$$\text{Performance}_{\text{gather}} = \frac{0.2\ \text{Bytes/Flop}}{8\ \text{Bytes/Flop}} \times 6.8\ \text{GFlops} = 170\ \text{MFlops}$$

The FPGA performs at ***2.5%*** of it's maximum floating point performance for a scatter.

From the above calculations it is clear that the Phenom performs poorly for a problem that is constrained by memory bandwidth. It exhibits a relatively small performance of 2.5% for a gather or a scatter operation.

Evidently, a cache based CPU performs poorly on the SMVM problem because of the memory bottleneck. This problem is primarily because of the cache-based architecture a commodity CPU is based on, that is ill-suited to handle scientific computations as discussed earlier. In the next section, we explore the implementation of the heat transfer problem on a system of Field Programmable Gate Arrays (FPGAs), in which each FPGA communicates with an on-board memory module.

2.3 SMVM on Field Programmable Gate Arrays

An FPGA can be considered as an "on-chip breadboard" that can implement digital logic functions. The logic functions are implemented in the form of four-input look up tables (LUTs). A LUT can implement any four input Boolean algebraic function. In Figure 4, a four input 'or' function is implemented using a LUT, but a more complication function can be implemented as easily, as long as it has not more than four inputs.

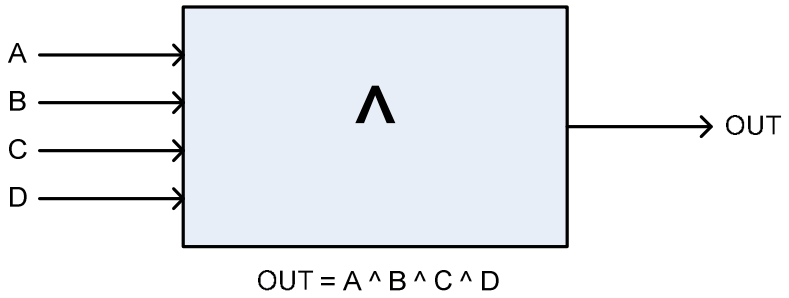

Figure 4: Look up Table implements Boolean algebraic functions

The LUTs are wired together by a programmable interconnect and this constitutes the fabric of the FPGA. The LUTs can be combined to implement functions with more inputs. Another, or perhaps the most significant feature of FPGAs is that many such LUTs (upto 254,400 for Stratix III) that implement Boolean algebra can function in parallel.

Configuring an FPGA for specific functionality requires that those functions be described in some manner. This can be done through the use of a hardware description language (HDL), typically Verilog or VHDL. Verilog, in particular, is a robust hardware description language that allows a user to create behavioral or data flow models of digital systems. Behavioral modeling is akin to computer programming, where a process is described sequentially. Data flow modeling is a better approach for digital circuits, which are more appropriately described using a parallel process. A specific approach to data flow modeling is known as Register Transfer Level (RTL) modeling and is synthesizable. The subset of Verilog language that is used in RTL descriptions can be automatically converted to circuits for a specific technology using CAD tools. The subset is known as synthesizable Verilog and the conversion process is known as Synthesis. For FPGAs,

synthesis of Verilog RTL descriptions generates circuits made of LUTs. The physical placement of LUTs on the FPGAs and the routing, or wiring, between them is decided by another CAD tool, the place and route tool. Finally, this information is transferred to the FPGA in a bitstream to configure it for the specific functionality.

Each FPGA consists of a series of LUTs, storage resources, and hardware multipliers. FPGAs contain logic elements to the order of tens of thousands, allowing for the construction of large parallel functions. Numerous implementations of hardware-multiplier based floating point multipliers and adders have been developed for FPGAs [6], [10]. Besides parallelism, another significant advantage that an FPGA based implementation offers over a CPU is regular memory access rates. This is because an FPGA has a memory-bank architecture as opposed to the cache-based architecture of a CPU. Each memory channel allows 200-400 MHz connections to on board DRAM. FPGAs provide significant parallelism in implementation because several floating point operations can be carried out simultaneously, the only constraint being the rate at which operands are loaded and saved to the memory and the size of the memory itself. This memory bandwidth and memory capacity can be multiplied, as can the number of operations by using multiple FPGAs in parallel - splitting the operands across memory banks, and splitting the operations between them to achieve optimal performance.

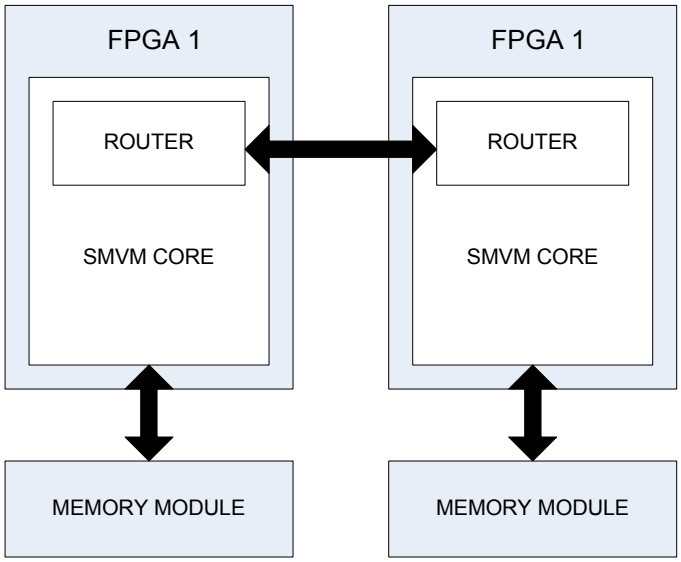

Figure 5: Implementation Architecture of the System

In our implementation, demonstrated in Figure 5, source and destination vectors and matrix values are partitioned across multiple memory banks attached to FPGAs. Data does not have to be accessed at fixed periods; data is fetched from off-chip memory as needed. The operations are data driven, and buffers are used rather than scheduling to keep the various sub-units of the computation busy. The configuration uses many FPGA-based load and store units to keep the series of compute elements implemented in the FPGAs busy. These load and store units are connected to DRAM memory banks to provide the necessary memory bandwidth. The compute elements perform floating point addition, subtraction, and multiplication and contain control circuitry to coordinate computation and data transport. An on-chip router is used as an interface between the memory and the compute elements. Any compute element can access any memory location via the router as long as the associated memory address is known. Efficient inter-FPGA connections can be accomplished via several mechanisms such as low-voltage

differential signaling (LVDS) and 12.8 GB/s multi-bit Hypertransport connections. The FPGAs can be connected to each other in a mesh or a bi-directional loop. The proposed architecture can implement both a gather and a scatter operation. Most importantly, this scheme is scalable: greater number of FPGAs or bigger memory banks can be used for problems of a larger size, without any changes in the implementation architecture.

The keynotes in our implementation are:

- Scalable to multiple FPGAs depending on problem size.
- Can implement scatter (compressed column) or gather (compressed row) operations without recompilation.
- Optimized for the common case (very sparse matrices of millions of rows – 2 items per row or column).
- Can operate on vectors and matrices that are too large to be locally stored in FPGA embedded memory.
- Effectively uses the FPGA pin count and existing I/O units to achieve peak memory bandwidth.

CHAPTER 3

PREVIOUS WORK

The primary criteria we use to evaluate previous work are:

1. Can the scheme implement both a gather and a scatter operation?
2. Is a specific matrix format used?
3. Is the scheme scalable to larger problem sizes?

Several previous research projects have implemented sparse matrix vector multiplication using FPGAs. de Lormier and DeHon [10] developed a multi-FPGA approach which uses matrices available in compressed row format. As discussed, some sparse matrices are best represented in compressed row format, while others are best represented in the compressed column format. Forceful representation of a matrix in a compressed row format might not be efficient. The limited size of the source and destination vectors (about 10000 values) allows them to be stored inside FPGA embedded memory. This approach is hence not scalable to larger problem sizes. This design was also optimized for repeated multiplication by the same matrix. Inter-FPGA communication is coordinated at compile time and hard-coded into FPGA hardware. Although efficient, this approach requires recompilation for every matrix. This is unsuitable for dynamic problems where matrices are continually changing or scientific codes where the device needs to operate on the order of 20 different matrices for each iteration/timestep of the solver.

Zhuo and Prasanna [7] also developed a sparse matrix-vector multiplication approach based on FPGAs which uses a matrix represented in compressed row format. For this implementation, the entire source value vector is again placed in each FPGA. This is a limiting factor on problem size and scalability.

A faster implementation is by Sun, Peterson and Storaasli [8]. They designed an FPGA approach which uses a non-conventional data format and takes advantage of a specialized accumulator. This approach is again limited to small matrices and uses a prescribed (but slightly nonstandard) matrix format as well as the assumption of an explicitly built matrix.

Our algorithm differs considerably from these prior designs by focusing on improving the memory bandwidth rather than improving the performance of the FPGA implementation. Previous works have surpassed the memory bottleneck by placing the data on FPGA memory blocks. This is convenient for problems that can fit on the memory blocks. However, for problems with larger data sizes, this architecture is not scalable. In our approach, the data is stored explicitly on on-board memory and accessed by the FPGA at DDR2 data rates. This data is stored in multiple memory banks, and the FPGA's capability of accessing multiple memory banks is used to overcome the memory wall. In this scheme, our algorithm is closer to that of El-kurdi, Gross, and Giannocopolos [9], which also focuses on very large vectors that cannot reside in FPGA embedded memory. The algorithm implemented by DuBois et al. [10] can work on very long vectors, but still assumes an explicit matrix is present in a prescribed format.

CHAPTER 4

FPGA IMPLEMENTATION OF SMVM

In Chapter 4, we shall discuss the choice of our FPGA platform in Section 4.1. In Section 4.2, we discuss the optimal way of saving SMVM data in the external memory. In Section 4.3, we broadly discuss the implementation algorithm for a gather and a scatter operation. In Section 4.4, we discuss the step by step implementation of the algorithm for a gather and a scatter operation by following the lifecycle of a packet. In Section 4.5, we explain the router module of the design. Finally, in Section 4.6 we discuss the memory access ordering.

4.1 FPGA Platform

The FPGA platform was carefully chosen to provide three capabilities:

1. Access to one or many off-chip DRAM Modules
2. Inter-board communication ability
3. On-chip hardware multipliers

A DE3 board (Figure 6) available from Terasic was chosen which contains a Stratix III EP3SL150 FPGA with 142,000 logic elements (LEs) and 384 18x18-bit Multiplier blocks. The DE3 has a single DDR2 SO-DIMM socket with a maximum capacity of up to 4GB. Each board has four HSTC Connectors, which can be used to stack up multiple such boards. The connectors provide a mechanism for inter-board communication while multiplying the memory bandwidth, which is essential for this implementation.

The circuitry on each board includes DDR2 DRAM controllers, floating point multipliers, and adders, inter-FPGA routers, and control circuitry needed to dynamically coordinate data movement. This circuitry has been developed in register-transfer level (RTL) Verilog, simulated for correctness and finally has been tested on a single and two FPGA Boards. During computation, large arrays are stored on the DRAM memory banks, and intermediate values are not transferred to CPU memory, transfer of initial data or final results occurs only at the very beginning and at the end of the computation through the USB ByteBlaster Interface.

As mentioned, each DE3 Board has four 128-bit HSTC connectors, labeled HSTC-A through D. The DRAM module is connected to HSTC-B. It is possible to create a system of DE3 Boards by stacking the boards as show in Figure 7. However stacking the boards causes the HSTC-B connectors to short and the DDR interface on either board to function errantly due to timing violations. It was therefore decided to stack the DE3 Boards as demonstrated in Figure 8.

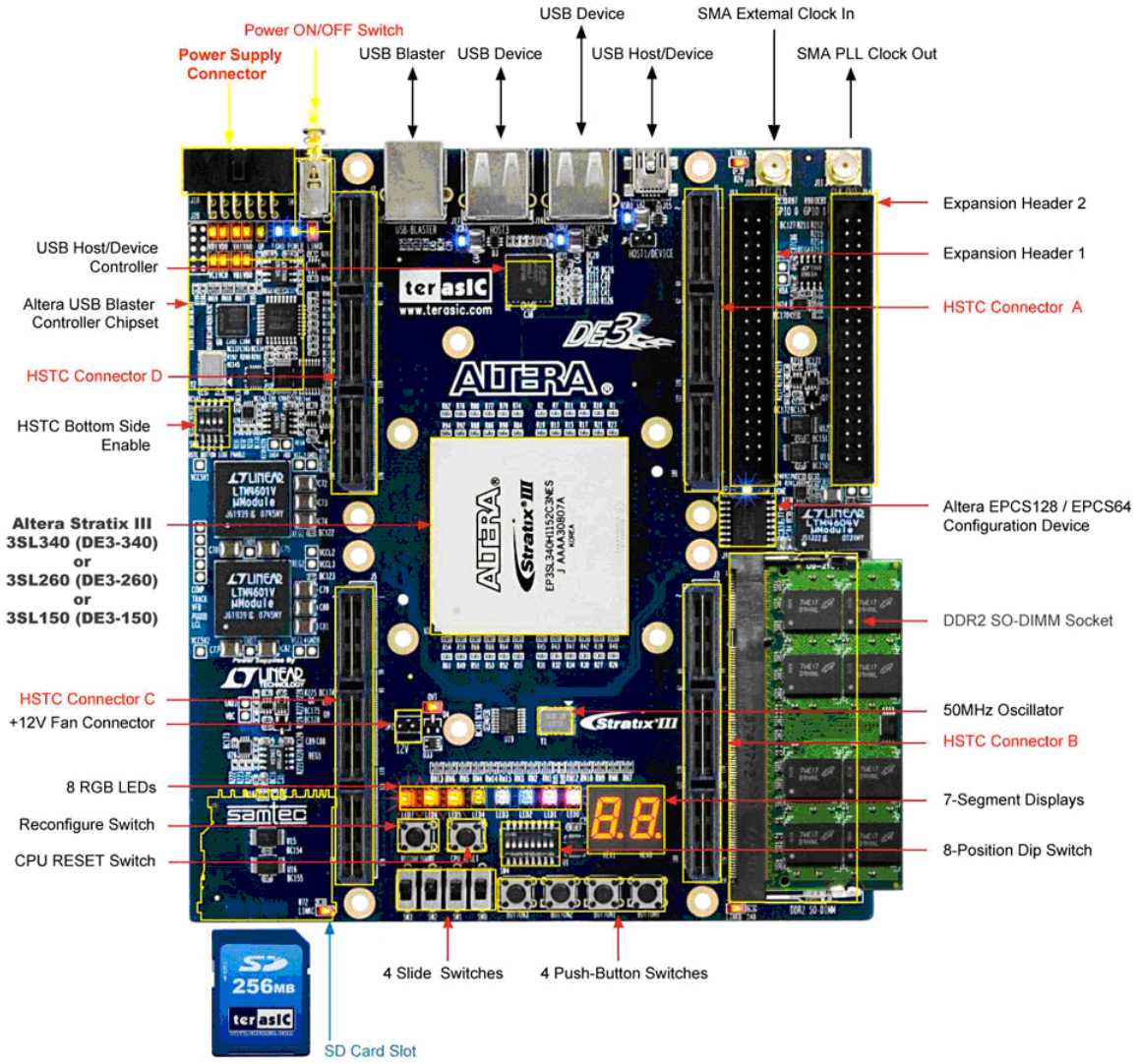

Figure 6. Implementation Platform [11]

Figure 7. Stacking Multiple DE3s (a)

Figure 8: Stacking Multiple DE3s (b)

4.2 Memory Management

The gather and the scatter operation require that the following arrays be stored on the external memory:

[1] The edge to node matrix : E2N[1,e], E2N[2,e]

[1] Temperature gradient along an edge : y[e]

[1] The inverse length of an edge : Li[e]

[1] Conductivity along an edge : k[e]

[2] The temperature at each node : x[E2N[i,e]]

[2] The sum of fluxes at each node : z[E2N[i,e]]

[1] The length of these arrays is the number of edges in a particular problem
[2] The length of these arrays is the number of nodes in a particular problem

The matrices are stored in banks of the external DRAM. The term 'banks' does not imply the presence of multiple memory interfaces. These banks cannot be accessed in parallel. Only one bank can be accessed at a time, through the single memory interface. However, we continue to use the bank structure to organize data, as if multiple interfaces were available to us. The motivation in doing this is that if multiple memory interfaces were indeed available, this data could be accessed in parallel in a way that would maximize the memory bandwidth of the design. Stratifying the data into banks is also helpful in estimating how the presence of multiple banks would improve performance of the design. For a multiple board implementation the matrix is evenly divided among the boards. In case the matrix cannot be evenly divided; one board hosts the larger part of the matrix.

Table 1: Memory Management in Architecture

	FPGA 0	FPGA 1
Bank 0	Gradient: L[0] to L[e_1] Divergence: k[0] to k[e_1]	Gradient: L[e_{1+1}] to L[e] Divergence: k[e_{1+1}] to k[e]
Bank 1	E2N[1,0] to E2N[1, e_1] E2N[2,0] to E2N[2, e_1]	E2N[1,e_{1+1}] to E2N[1,e] E2N[2,e_{1+1}] to E2N[1,e]x
Bank 2	y[0] to y[e_1]	E2N[1,e_{1+1}] to E2N[1,e] E2N[2,e_{1+1}] to E2N[1,e]
Bank 3	Gradient: x[0] to x[n_1] Divergence: z[0] to z[n_1]	Gradient: x[n_{1+1}] to x[n] Divergence: z[n_{1+1}] to z[n]

$0 < e_1 < e$; e = total number of edges
$0 < n_1 < n$; n = total number of nodes

4.3 Implementation Algorithm

In our implementation, both scatter and gather operations are supported with the same architecture without any reconfiguration. To indicate whether an operation is a gather or a scatter, a single slide switch on the board is flipped (SW[0]). The slide switch is in the ON position for a gather operation and in the OFF position for the scatter operation. The list of edges is distributed to memory modules attached to each FPGA. The distribution places an equal amount of matrix data on each memory bank. In order to distribute computations between FPGAs, data needed for operations is divided into packets which are sent between FPGAs. For bigger matrices, this architecture can be scaled by using more FPGAs or bigger memory modules. The implementation architectures for gather and scatter operations are discussed in subsection 4.3.1 and 4.3.2 of Chapter 4, respectively.

4.3.1 The Gather Operation

Consider a gather operation (the gradient operation) performed using the system configuration shown in Figure 5. The gather operation is given by:

```
for (e = 0; e < num_edges; e++) {
y[e] = ( (Li[e]*x[E2N[1,e]]) - (Li[e]*x[E2N[2,e]]) ); }
```

This can also be written as:

Equation 4.3.1

```
                    for (e = 0; e < num_edges; e++) {
Organized into Packet 1 ➡ y[e] += Li[e] * x[E2N[1, e]];
Organized into Packet 2 ➡ y[e]-= Li[e] * x[E2N[2, e]];
                    }
```

Broadly speaking, the gather operation consists of two steps:

A. The Source Operation

On each clock, every load unit reads the multiplier Li[e] and the source addresses E2N[i,e] for the source value x[E2N[i,e]]. The address for y[e] is internally generated because y[e] is laid out exactly like Li[e]. For each y[e], 'i' packets are generated. In this case, because the value of 'i' is 2, hence two packets are generated that contain:

the sign bit : + or -

the multiplier value : Li[e]

the source address : E2N[1,e] or E2N[2,e]

destination address : &y[e]

For each packet a source operation is initiated. The source operation propagates through the routing subsystem which moves it towards the correct source value memory bank. Depending on the source address, the packet may be transferred to a different FPGA. Upon its traversal through the routers (the routers operate using the source

address), the source data will be located on the memory module attached to the correct FPGA. The compute unit will proceed to load the source data and multiply it with Li[e]. The product x[E2N[1,e]]*Li[e] replaces Li[e] in the packet. The source operation is non-trivial and required, unlike the source operation in the scatter, as shall be seen in Section 4.2 of Chapter 4.

B. The Store Operation

A store operation for the result is then initiated. The store operation propagates through the router subsystem which moves it to the FPGA with the correct storage memory bank. In this case, the store operation directs it back to the FPGA where this operation was initiated. Upon its traversal through the routers (the routers operate using the destination address), the data will be located at the correct memory bank for storage, where the store unit will proceed to add it to / subtract it from the existing data in the desired location.

4.3.1 The Scatter Operation

Consider a scatter operation (the divergence operation) performed using the system configuration shown in Figure 5.

Equation 4.3.2

```
                          z = 0;
                          for (e = 0; e < num_edges; e++) {
Organized into Packet 1   z[E2N[1,e]]+= y[e]*k[e];
Organized into Packet 2   z[E2N[2,e]]+= y[e]*k[e];  }
```

Broadly speaking, the scatter operation consists of two steps:

A. The Source Operation

On each clock, every load unit reads the multiplier k[e] and the destination addresses E2N[i,e] for the value z[E2N[i,e]]. The source address for y[e] is internally generated because y[e] is laid out exactly like k[e]. For each E2N[i,e], a packet is generated that contains:

the sign bit : + or -

the multiplier value : k[e]

the source address : &y[e]

destination address : E2N[1,e] or E2N[2,e]

In this case two packets are generated. For each packet a source operation is initiated. In this case, the source data is located on the same FPGA. The compute unit will proceed to load the source data and multiply it with k[e]. The product y[e]*k[e] replaces k[e] in the packet. The source operation is trivial, in this case, and is performed simply to maintain consistency in the architecture of the gather and the scatter operation.

B. The Store Operation

For each (E2N[i,e]) destination, a store operation is initiated. For each destination address, the store operation propagates through a series of switches which slowly move it towards the correct destination value memory bank. Depending on the destination address, data may be transferred to a different FPGA. Upon its traversal through the switches (the switch operates using the source address), the data will be located at the

correct memory bank, where the store unit will proceed to add or subtract it to or from the existing data in the desired location.

In the following Section, the step by step implementation of a gather of a scatter operation will be discussed. For the sake of simplicity, the data present at the source address will be referred to as source data, and the data present at the destination address will be referred to as destination data.

4.4 Step by Step Implementation of a Gather or a Scatter Algorithm

The gather or scatter operation proceeds in ten steps as explained below. Figure 9 demonstrates the life cycle of a packet, from the time it is created to the time the data is finally written to the destination address. Figure 10 illustrates a system of FIFOs a packet passes through during its lifetime.

Step 1: *Sequential Read*

The DDR Interface is used in this step. The primary operation being performed is a sequential read. Two operations happen concurrently in this step:

1. The multiplier value (Li[e] for gather and k[e] for scatter) is fetched sequentially from the first sixteen addresses of Bank 0 of the DDR. Since each DDR location in Bank 0 stores four 64 bit multiplier values, this corresponds to 64 multiplier values. They are stored in a FIFO hence referred to as “seq0_fifo”.
2. The destination values for the gather operation (source values for the scatter operation), y[e], are laid out exactly as the multiplier values but in Bank 3. It is

possible to generate their addresses and store them in another FIFO as the multiplier value is being fetched from the DRAM. This FIFO will be referred to as “addr_fifo”. This operation does not require the memory interface hence can be performed in parallel with the memory operation.

Step 2: *Sequential Read*

The DDR Interface is used in this step. The operation being performed is a sequential read. The E2N values, which represent the source address for the gather operation (the destination address for the scatter operation) are fetched sequentially from the first sixteen addresses of Bank 1 of the DDR. Since each DDR location in Bank 1 stores eight 2-bit address values, this corresponds to 128 E2N address. These addresses are stored in a FIFO referred to as “seq1_fifo”. These addresses are fetched sequentially; however the addresses themselves are random. The addresses contain the complete DRAM address information, including the board address. Out of the 32 bits, the first eight bits are zero padded, the next bit specifies the board address in a double board implementation and the final 23 bits specify the DRAM address.

Step 3: *Packet Creation*

Packets are created in this step. This operation begins after the data first appears on the bus in Step 2. It is run parallel with Step 2, but is staggered by the read latency of the first read operation in Step 2. Packets are created from the data available in seq0_fifo, seq1_fifo and addr_fifo. A single bit is used to represent a “+” or “-“ sign, which will be

used to decide whether the value in the packet is added to, or subtracted from the final destination value. The packets are stored in a FIFO hence referred to as “packet_fifo”.

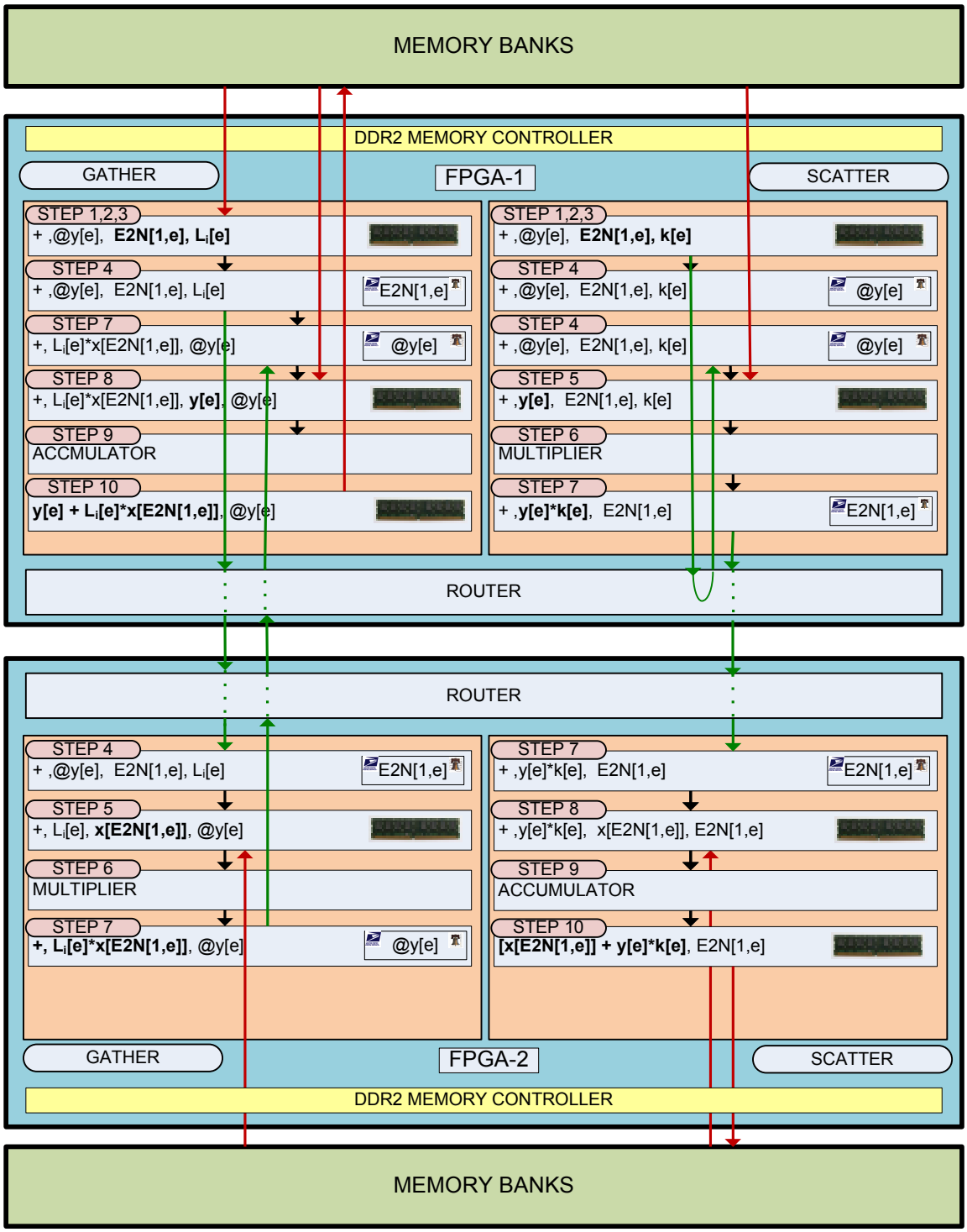

Figure 9: A Gather and Scatter Operation Demonstrated on Two FPGAs

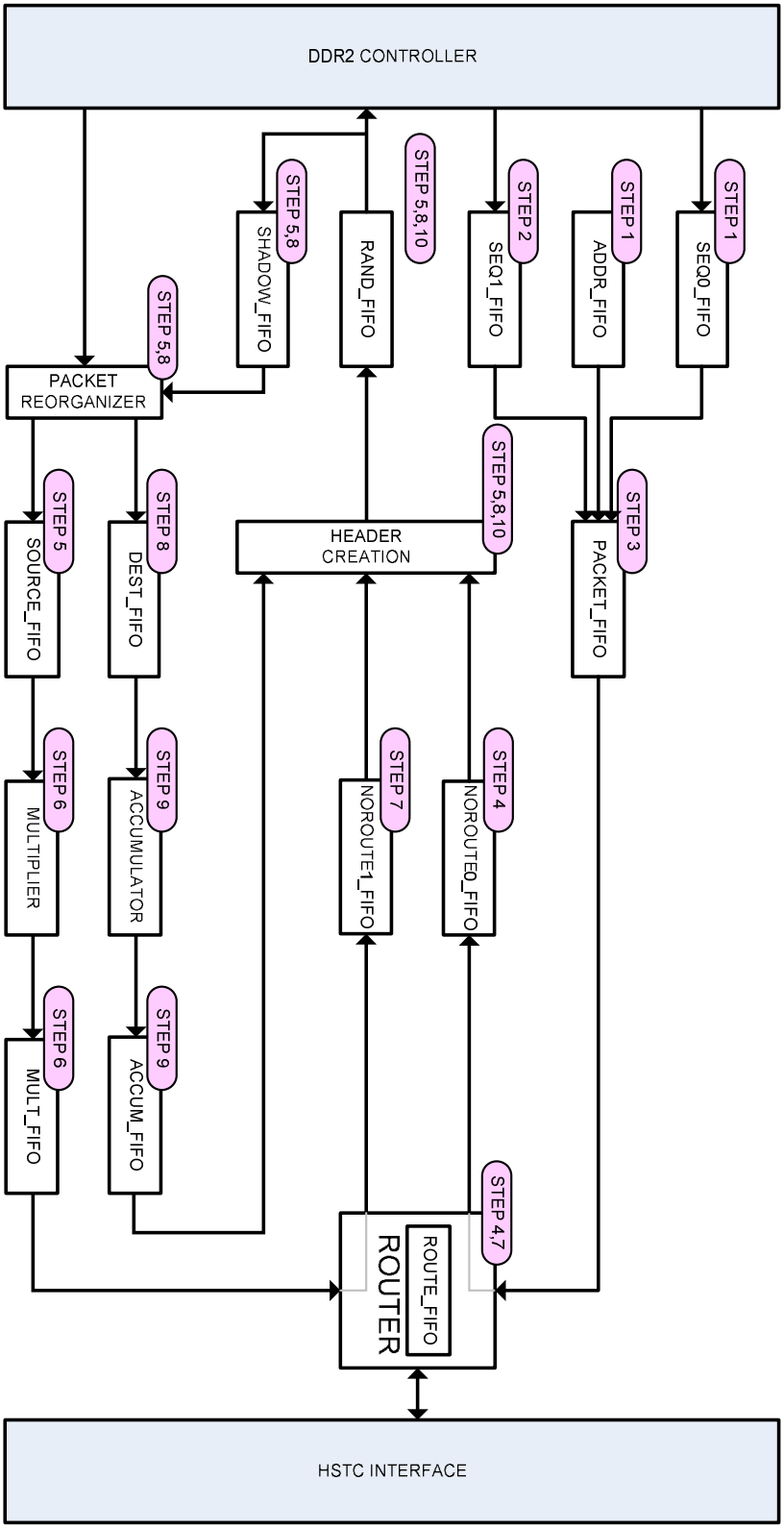

Figure 10: A system of FIFOs

The only difference between the gather and the scatter operations is in this step, where the packets are constructed differently. After this, the knowledge whether the operation is a gather or a scatter is unnecessary, as both operations proceed in exactly the same way.

The structure of the packets is as follows:

Gather

Sign	Destination Address	Source Address	Multiplier
(1 bit)	(32 bits)	(32 bits)	(64 bits)
+/-	@y[e]	E2N[1,e] or E2N[2,e]	L_i[e]

Scatter

Sign	Destination Address	Source Address	Multiplier
(1 bit)	(32 bits)	(32 bits)	(64 bits)
+/-	E2N[1,e] or E2N[2,e]	@y[e]	k[e]

Step 4: *Route to Source Address*

This step is one of the two route operations in the lifecycle of a packet. For a gather, this route operation is required because the source address could be located on either board. For a scatter operation, this operation is not required because the source address is located on the board where the packet is created, but is performed to maintain consistency between the gather and the scatter operations. This routing does not create any overhead on design performance, as shall be seen in the results in Chapter 5. Once a packet is present in packet_fifo, the route operation begins and continues until packet_fifo is empty. The router compares the location information present in the source address of the packet with the board identifier. If the address is a match, the packet does not need to be routed and is stored in a FIFO hence referred to as “noroute0_fifo”. If the packet needs to be routed it is stored in a FIFO called “route_fifo”. If the router detects

that route_fifo is not empty, the route operation is initiated. The route operation shall be discussed in a Section 4.5 of Chapter 4. Step 4 is run in parallel with Step 3, and is staggered by a clock cycle.

Step 5: *Random Read from Source Address*

This is the first of the two random read operations in the lifecycle of a packet. After the DDR interface has completed the sequential operations in Step 1 and 2, the random reads from the source address can begin. At this point, if noroute0_fifo is not empty, this signifies that a random read access to the memory is needed, to either the source address or the destination address (as shall be seen in Step 7). All random memory accesses are directed through a single FIFO that interfaces with the DDR2 Controller. This FIFO will be referred to as "rand_fifo". The packet from noroute0_fifo joins the random access request queue in rand_fifo with a packet header, which indicates whether the random access request is a read operation from the source address or the destination address, or a write operation to the destination address. This information can be represented by an additional two bits. For the gather operation, this step is a read operation from the source address.

Header	Interpretation
00	Read from source address
01	Read from destination address
11	Write to destination address

Gather

Header	Sign	Destination Address	Source Address	Multiplier
(2 bits)	(1 bit)	(32 bits)	(32 bits)	(64 bits)
00	+/-	@y[e]	E2N[1,e] or E2N[2,e]	L_i[e]

Scatter

Header	Sign	Destination Address	Source Address	Multiplier
(2 bits)	(1 bit)	(32 bits)	(32 bits)	(64 bits)
00	+/-	E2N[1,e] or E2N[2,e]	@y[e]	k[e]

A read request is sent to the source address via the DDR Controller. If rand_fifo continues to be non-empty and the DDR Controller is accepting read requests, another read request is sent out. This continues until rand_fifo is empty. Now that some read requests have been sent, the corresponding data will appear on the bus in the same order as the requests that were sent. There is a need to maintain a record of the read requests that were sent, that indicate whether a read from the source or the destination address was requested, thus indicting the FIFO that the packet with read data will be saved in – the multiply or the accumulate FIFO. In addition to the extended header, other information such as the multiplier value, the destination address and the sign bit are required to reconstruct the packet. This record is maintained in another FIFO called “shadow_fifo”. Therefore as each read request is sent to the DDR, the packet information from rand_fifo is captured into shadow_fifo except for the source address which is discarded.

When the rdata_valid signal is asserted by the DDR controller, indicating presence of valid data on the data bus, the data is buffered in a register. This data is then stored into the packet, which is retrieved from shadow_fifo and is written into “source_fifo”. The header is now discarded.

Gather

Sign	Destination Address	Source Value	Multiplier
(1 bit)	(32 bits)	(64 bits)	(64 bits)
+/-	@y[e]	x[E2N[1,e]] or x[E2N[2,e]]	L_i[e]

Scatter

Sign	Destination Address	Source Value	Multiplier
(1 bit)	(32 bits)	(32 bits)	(64 bits)
+/-	E2N[1,e] or E2N[2,e]	y[e]	k[e]

Step 6: *Multiply*

This is the double precision floating point (DPFP) multiply operation. If source_fifo is found to be non-empty, the DPFP multiply operation is initiated. This operation multiplies the multiplier value (that was fetched sequentially from the DDR in Step 1) with the source value obtained by a random access in the Step 6. The multiply operation takes seven clock cycles, thus seven multipliers are used in rotation, such that a multiply operation can be initiated in each clock cycle, if the multiplicands are available. Seven clock cycles after the multiply was initiated, the product replaces the multiplier value in the packet that is written to "mult_fifo". The source data is discarded.

Gather

Sign	Destination Address	Product
(1 bit)	(32 bits)	(64 bits)
+/-	@y[e]	L[e] * x[E2N[1,e]] or L[e]*x[E2N[2,e]]

Scatter

Sign	Destination Address	Product
(1 bit)	(32 bits)	(64 bits)
+/-	E2N[1,e] or E2N[2,e]	y[e]*k[e]

Step 7: *Route to Destination Address*

This is the second route operation in the lifecycle of a packet. Once a packet is present in mult_fifo, the route operation begins and continues until mult_fifo is empty. If the packet does not need to be routed it is stored in a FIFO called "noroute1_fifo". If the packet needs to be routed it is stored in route_fifo. The route operation is discussed in detail in Section 4.5.

Step 8: *Random Read from Destination Address*

This is the second of the two random read operations in the lifecycle of a packet. The packet from noroute1_fifo once again joins the random access request queue in rand_fifo with a packet header. In this case, the header indicates it is a read operation from the destination address.

Gather

Header	Sign	Destination Address	Product
(2 bits)	(1 bit)	(32 bits)	(64 bits)
01	+/-	@y[e]	L[e] * x[E2N[1,e]] or L[e]*x[E2N[2,e]]

Scatter

Header	Sign	Destination Address	Product
(2 bits)	(1 bit)	(32 bits)	(64 bits)
01	+/-	E2N[1,e] or E2N[2,e]	y[e]*k[e]

Read requests are sent to the destination address via the DDR Controller constantly if the DDR Controller is accepting read requests, until rand_fifo is empty. Once again, shadow_fifo is used to store the header and the remaining packet information until the data corresponding to the packet appears on the data bus of the DDR controller. The

modified packet is then stored in a FIFO called “dest_fifo”. The destination address is retained, because it will be required for the packet to be written back.

Gather

Sign	Destination Address	Destination Data	Product
(1 bit)	(32 bits)	(64 bits)	(64 bits)
+/-	@y[e]	y[e]	L_i[e]*x[E2N[1,e]] or L[e]*x[E2N[2,e]]

Scatter

Sign	Destination Address	Destination Data	Product
(1 bit)	(32 bits)	(64 bits)	(64 bits)
+/-	E2N[1,e] or E2N[2,e]	x[E2N[1,e]] or x]E2N[2,e]]	y[e]*k[e]

Step 9: *Accumulate*

This is a 64-bit DPFP accumulate operation. If dest_fifo is non-empty, the DPFP accumulate operation is initiated. Based on the sign bit information that each packet holds, the accumulate operation adds or subtracts the product obtained in Step 6 with the destination value obtained by the random access Step 7. The accumulate operation takes seven clock cycles, thus seven accumulators are used in rotation, such that an accumulate operation can be initiated in each clock cycle.

Seven clock cycles after the accumulate was initiated, the accumulator output replaces the multiplier value in the packet that is written to a FIFO called “accum_fifo”. The sign bit, destination data and the product are discarded.

Gather

Destination Address	Accumulated Product
(32 bits)	(64 bits)
@y[e]	y[e] + L[e]*x[E2N[1,e]] or y[e] - L[e]*x[E2N[2,e]]

Scatter

Destination Address	Accumulated Product
(32 bits)	(64 bits)
E2N[1,e] or E2N[2,e]	x[E2N[1,e]] + y[e]*k[e], x[E2N[2,e]] - y[e]*k[e]

Step 10: *Write to Destination Address*

If accum_fifo is found to be non-empty, its contents are written into rand_fifo with the packet header indicating that this is a random write operation. The data is finally written to the destination address.

Gather

Header (2 bits)	Destination Address (32 bits)	Accumulated Product (64 bits)
11	@y[e]	y[e] + L[e]*x[E2N[1,e]] or y[e] - L[e]*x[E2N[2,e]]

Scatter

Header (2 bits)	Destination Address (32 bits)	Accumulated Product (64 bits)
11	E2N[1,e] or E2N[2,e]	x[E2N[1,e]] + y[e]*k[e], x[E2N[2,e]] - y[e]*k[e]

The operation goes back to Step 1, where the next set of data is fetched, and so on, until all the data has been processed.

4.5 The Router Module

Each FPGA has in the system has a board identifier hard-coded into the RTL. The FPGA's are connected to each other over a 128-bit bi-directional High Speed Terasic Connector (HSTC) channel. Since there is a single 128-bit channel present to route the packets from any FPGA to any other FPGA in the system, a bus arbitration mechanism is required. For this purpose a single board is designated as the bus controller, and all the other boards are designated as slaves. A single slide switch - SW[2], indicates whether a board is a Master or a Slave. If the slide switch is in the ON position, the board is the Master. There can be only one Master board in the system. The Master board receives and processes all routing requests according to a rotating priority mechanism.

On each board, the router sorts and routes packets over the HSTC interface that connects all the boards in the system. The router compares the board address present in the source (or the destination) address of each packet with the board identifier of the board the packet is present on. If the packet is present on the correct board already, it is not routed, but is written to rand_fifo for random access to the memory module attached to the board. If the comparison fails, the packet needs to be routed to a different board, and the route operation is initiated.

A 129 bit packet (for a route to the source address) needs to be routed over a 128 bit bus, in addition to various control signals. To do this, the packet is compressed, so that only 24 useful bits out of 32 bits for the source or the destination address are retained. This compresses a 129 bit packet into a 113 bit packet. The packet size for a route to the destination address is 97 bits, and this is not a problem to route across a 128 bit channel. However, for consistency, this packet is compressed into an 89 bit packet as well and buffered to be 113 bits.

Each packet is assigned a single bit header that indicates whether it is being routed to the source address or the destination address. Once the packet reaches the destination board, this information is useful in knowing whether the data to be fetched is from the source or the destination address of the packet.

The interface between the Master and the Slave consists of the following signals:

1. A 114 bit bi-directional data bus
2. A “Busy” signal driven by the Master
3. Dedicated bus requests signals between the Master and each Slave (1 in this case)
4. Dedicated bus grant signals between the Master and each Slave (1 in this case)

If the board that requires a route is a Slave, the following set of events takes place:

- The Slave requests the Master to release the bus using the dedicated bus request line between them.
- If the bus is not busy, the Master does the following:
    i. Stops driving the data bus
    ii. Drives high the dedicated bus grant signal
    iii. Drives high the busy signal
    iv. When the Slave receives the bus grant signal, it starts driving the data bus to transfer packets.
- If the bus is busy, the Master waits for the current transfer to complete then assigns the bus to the Slave. If there are other requests present then the Master uses a rotating priority scheme to assign the bus.

The Slave now drives the data bus, sending packet information across along with the destination board ID. The Slave will maintain the bus request signal high for as long as it requires transferring data. This data is sent to all the boards in the system, but only the board with the correct board ID processes this information. Once the Slave has

transferred all the packets, it releases the data bus and drives the bus request signal low. The Master takes control of the bus, drives the Busy signal low and processes the next request, if any.

If the board requiring a route is the Master, the following set of events takes place:

- If the busy signal is driven low, and no other bus requests are available, the Master drives high the busy signal and broadcasts the destination board id and the packet information.
- If the busy signal is driven high, it indicates that a transfer is currently in progress. The Master will send the packets across, once that transfer is complete.

4.6 Memory Request Ordering

All packets that require random memory access are directed to rand_fifo. This is because multiple packets may require random memory access at the same time through a single memory interface and a robust priority scheme needs to be implemented. The memory operation required could be to:

1. read from the source address(for packets present in noroute0_fifo)
2. to read from the destination address(for packets present in noroute1_fifo)
3. to write to the destination address(for packets present in accum_fifo).

If the three aforementioned FIFOs are non-empty, then to choose the packet that will first be written into rand_fifo, a fixed priority scheme is used.

Random read $_{\text{source address}}$ > Random read $_{\text{destination address}}$ > Write

This is to improve the DDR controller efficiency by preventing the controller from swapping between writes and reads to different rows, forcing a row to be opened and closed on every transaction.

CHAPTER 5

RESULTS

In Section 5.1 of Chapter 5, we analyze the floating point performance of the system based on the available resources. In Section 5.2, we compare the expected performance of the system with its measured performance, and compare the performance versus an AMD Phenom X4 9200 CPU. Based on the resource utilization summarized in Section 5.3, we discuss parallelization techniques in Section 5.4. In Section 5.5, we briefly compare the performance of our implementation with implementations in [6] and [10]

5.1 Expected Performance

First, we calculate the maximum floating point performance of a single DE3 board. The DE3 Board has a Stratix EP3SL150 with 142,000 logic elements (LEs) and 384 18x18-bit multiplier blocks as indicated in Table 2 below.

Table 2: EP3SL150 Resources

Device	ALUTs	Equivalent Les	18x18 Multipliers	PLLs
EP3SL150	113600	142000	384	8

Using Altera's intellectual property tool, it is possible to implement a double precision floating point (DPFP) multiplier using 9 on-chip 18x18 hardware multipliers and 900 LUTs. It is possible to implement a DPFP accumulator using 1721 ALUTs.

Table 3: Floating Point Core Utilization

	Synthesis	Multipliers	ALUTs	Latency	Performance
DPFP Multiplier	**Logic + Muliplier**	**9**	**900**	**13**	**300 MHz**
DPFP Adder	**Logic**	**0**	**1721**	**17**	**300MHz**

Summarizing the ideal floating point capability of the DE3:

Number of hardware multipliers available	: 384 18x18 bit hardware multipliers
Max DPFP Multipliers that can be implemented	: 384/9 = 42
42 Multipliers Utilize	: 42 * 900 = 37800 ALUTs
Maintaining a 1:1 Multiplier:Adder Ratio	: 42 DP Adders can be implemented
42 Adders Utilize	: 42 * 1721 = 72282 ALUTs
Total Resources Used	
ALUTs	: 110082 out of 113600 ALUTs
Multipliers	: 378 out of 384 multipliers
Theoretical Maximum Peak Floating Point Performance	: 300*84 = 25.2 GFlops

When many DPFP cores are used in parallel, data sheet performance may be unachievable. It is difficult to route 64 bit data paths while populating the FPGA with DPFP cores without a considerable decrease in system performance. This may result in

unused logic and decrease in clock speed for the DPFP functions. The clock speed will degrade by 1/3rd and 15 percent of the logic will be unused [10]

Thus 39 (42*0.15) adders and 39 multipliers can fit on the FPGA and will operate at a conservative frequency of 133 MHz. This leads to a maximum floating point performance of 10.374 GFlops and not 25.2 GFlops as calculated. Extrapolating the results to a two FPGA DE3 System:

Table 4: Peak Performance on a Single and Double Board Implementation

	Single Board DE3 Implementation		Two Board DE3 System	
	Number	LUTS	Number	LUTS
DP Multipliers	39	35100	78	70200
DP Adders	39	67119	78	134238
DDR2 Interfaces	1	1946	2	3892
Switches	1	5000	2	10000
Total ALUTs	109165		218330	
[1] Memory Bandwidth	0.396 GBps		0.792 GBps	
[2] Expected Performance	10.374 GFlops		20.748 GFlops	

[1] Memory Bandwidth for a single Stratix EP3SL150 based DE3 Board:
= DDR SDRAM Bus Width * 2 * frequency of operation * efficiency
= 8 Bytes * 2 clock edges * 133 MHz * 0.2
= 0.396 GBps

[2] Expected Performance for a single Stratix EP3SL150 based DE3 Board:
= (39 DPFP Multiplications + 39 DPFP Additions) x 133 MHz
= 10.374 GFlops

In the next few steps, we will calculate the application balance, the processor balance and finally the performance of a single DE3 Board for a gather and a scatter operation.

The processor balance for a single DE3 Board is calculated as:

$$\text{Processor Balance} = \frac{\text{Bandwidth (GBytes / sec)}}{\text{Peak Performance (GFlops)}} = \frac{0.396 \text{ GBytes/sec}}{10.374 \text{ GFlops}} = 0.038 \text{ Bytes/Flop}$$

Consider the memory requirement for a single packet of the **gather operation**:

```
y[e] + = x[E2N[1,e]* Li[e]
```

The gather operation requires the following data from memory:

```
1 - 4 byte sequential read for E2N[i,e]
1 - 8 byte sequential read for Li[e]
2 - 8 byte random reads for x[E2N[i,e]] and y[e]
1 – 8 byte random write for y[e]
```

Now consider the memory requirement for a single packet for the **scatter operation**:

```
z[E2N[i,e]]+ = y[e] * k[e]
```

The scatter operation requires the following data from memory:

```
1 - 4 byte sequential read for E2N[i,e]
1 - 8 byte sequential read for k[e]
2 - 8 byte random reads for z[E2N[i,e]] and y[e]
1 – 8 byte random write for z[E2N[i,e]]
```

The random accesses are the bottleneck in this operation, and we shall exclude other memory operations in the successive calculations. The random memory access requirements for a packet in the gather or the scatter operation are the same. 16 bytes of data needs to be randomly read from the memory for each packet. This takes place in two separate read transactions. Since 32 bytes of data is fetched or sent by the controller on the local side at a time, it corresponds to a 64 byte random read operation instead of a 16 byte operation. Dividing the gather and the scatter operations into packets equalizes them with regards to number of floating point operations needed. Both the gather and the scatter require two floating point operations per packet, making their performance alike.

$$\text{Application Balance}_{\text{gather}} = \frac{\text{Number of memory references}}{\text{Number of flops}} = \frac{64\ \text{Bytes}}{2\ \text{Flops}} = 32\ \text{Bytes/Flop}$$

The FPGA performs with a maximum theoretical performance of:

$$\text{Performance} = \frac{0.038\ \text{Bytes/Flop}}{32\ \text{Bytes/Flop}} \times 10.374\ \text{GFlops} = 12.32\ \text{MFlops}$$

The FPGA performs at ***0.12%*** of it's maximum floating point performance for a gather or a scatter.

Another way to understand this is as follows:

The memory bandwidth offered by a DE3 Board is 0.374 GBps. At a clock speed of a 133 MHz, 2.98 bytes can be transferred to the DE3 from the memory per clock cycle. Each y calculation for the gather, or z calculation for the scatter requires 64 bytes, so we can effectively implement 2.98/64 = 0.0466 y operations per clock cycle. Each y operation has 2 DPFP operations. At a clock speed of 133 MHz clock, and a 0.396 GBps bandwidth, we can implement 2*0.0466 = 0.093 DPFP operations.

The DE3 Board can implement 78 DPFP operations, so at a clock speed of 133 MHz, we effectively utilize 0.12% of the available DPFP resources. To utilize 100% of the available DPFP resources, a practical memory bandwidth of 330 GBps is needed per DE3 board. This is not possible as only 0.12% of this bandwidth is available. Alternatively, the FPGA can be operated at a lower frequency to increase the percentage of on-chip DPFP resources that can be used. At a frequency of 120 KHz, 100% of the DPFP resources can be used.

The FPGA can be operated at a lower frequency to utilize more of the on-chip DPFP resources, or at a higher frequency and use fewer resources. The floating point performance of the FPGA remains constant. To calculate the tradeoff between operating the FPGA at a higher frequency with fewer DPFP resources, and a lower frequency with higher DPFP resources, we consider the power consumption metric. To estimate the power that would be used by our design we used PowerPlay – an early power estimator by Altera. The power consumption remains constant for any combination of frequency and consequent number of DPFP resources we would use and is of the order of hundreds of mWatts.

Consequently, we have the freedom of choosing any operating frequency, without affecting the DPFP performance or the power performance of our system. We choose to maximize the frequency of operation of the FPGA (and decreasing total number of DPFP resources we use). The reasoning for this is that when many DPFP cores are used in parallel, routing 64 bit data paths is complicated. At a frequency of 133 MHz, the gather and scatter operations can at the most utilize 0.12% of the DPFP resources i.e. 0.09 compute elements. Since the DPFP resource needed may be either a multiplier or an accumulator, we need to implement 1 DPFP multiplier and 1 DPFP accumulator. The multiplier and the accumulator have a latency of 7 clock cycles. For this reason, 7 multipliers and 7 accumulators are implemented.

For a single DE3 board:

Expected Performance for a single DE3 board for a Gather or Scatter:
= 0.09 x 133 MHz
= 11.97 MFlops

This concurs with our previous calculations. On a system of 2 DE3 boards:

Expected Performance for a DE3 System for a Gather or Scatter:
= 2 x 0.09 x 133 MHz
= 23.94 MFlops

Finally, the expected performance for a gather or a scatter operation on the system of two FPGAs is:

Table 5: Final Estimates

Performance	Single DE3 Board	DE3 System	CPU
Gather / Scatter	11.97 MFlops	23.94 MFlops	

5.2 Measured Performance

The experiments were run on a single board, and then on a dual board set-up. The type of operation (a gather or a scatter) was set by using an on-board slide switch. For a multiple board implementation the Master board was also determined by the position of a slide switch. The matrices were loaded into the on board DRAM using Perl scripts for efficiency. Another slide switch (SW[2]) was flipped to begin the operations. An on-chip counter counted the time taken from start to completion of an SMVM. Signal Tap Logic Analyzer was used to view the counter at the end of the operation.

The results were found to be as indicated in Table 6. The results for the FPGA were found to be very close to the estimated values in the previous section. The results for the CPU could only be recorded for the largest SMVM. The CPU performed 8 times better than the two FPGA system, even when it is clocked 21 times faster than the FPGAs.

Table 6: Measured Performance

Nodes	Edges	DPFP Calculations	Calculation Time (msec)			DPFP Performance (MFlops)		
			1 Board	2 Board	CPU	1 Board	2 Board	CPU
11k	34k	136k	11.5	5.8		11.82	23.44	
128k	383k	1532k	125	65		12.26	23.56	
237k	710k	2840k	228	118	~15	12.45	24.06	~190

As demonstrated in Figure 11, the system is scalable. The floating point performance of a two board implementation is twice that of a single board implementation. This is primarily because the memory bandwidth doubles for a two board implementation, due to the presence of twice as many memory interfaces and this directly improves the floating point performance of the system. This also demonstrates that there is no routing overhead involved in using multiple boards due to the memory performance limitation.

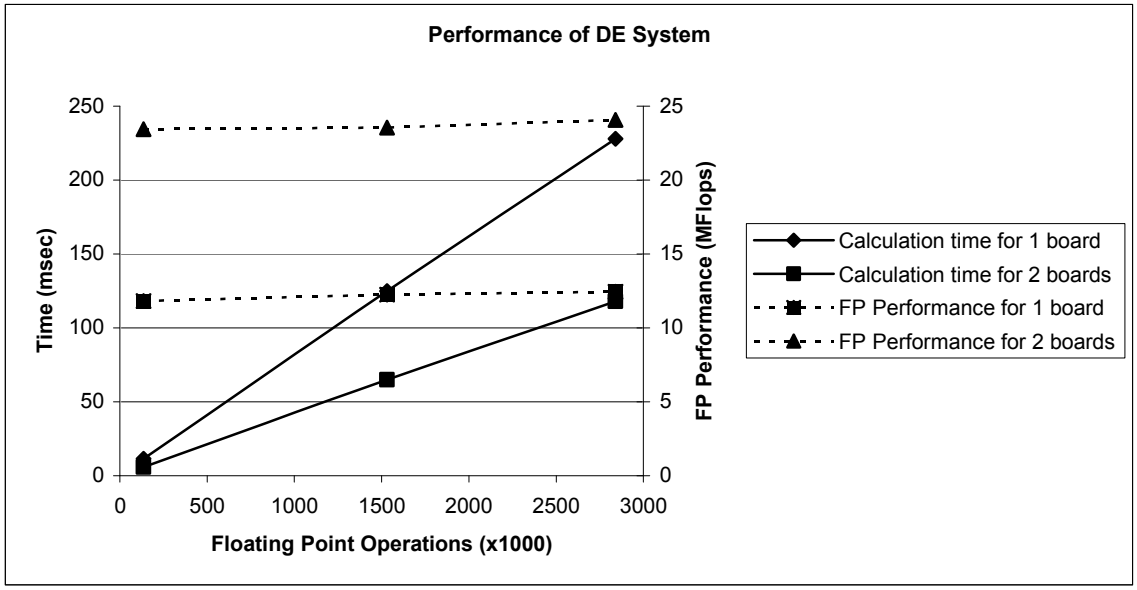

Figure 11: Performance of the DE3 System

The FPGA performs a random read each time it requires some data from the DRAM, buffering up to four reads at a time, but still suffering from longer access latencies that plague random reads on dynamic memory. It is possible to improve the performance of the system of FPGAs by adding more memory interfaces per board, and increasing the number of boards in the system. This will be discussed in detail in Section 5.4 of Chapter 5.

5.3 Resource Utilization

70 of the 384 hardware multipliers are used (18%) to perform DPFP multiplication. 0.5MB of on chip memory is used of the available 0.68 MB (76%). Out of the 0.5 MB block memory being utilized in the design, 0.25MB is being used to load

the matrices into the DRAM through the USB Byte Blaster, and the remaining is used by various FIFOs being used in the design.

The DDR Utilization was measured with another counter, that counted for each clock cycle that the DDR interface was being used, either sending a requesting, or waiting for data to be read or written. The DDR interface was found to be busy 90% of the total run time. Further analysis revealed that the DDR interface spent 88% of the time sending or awaiting random reads, 10% of the time doing random writes and only 2% of the time doing sequential reads. This conforms to our choice of having taken only the random reads into consideration for calculating the floating point performance of the FPGAs.

5.4 Potential Parallization

The parallelization in the steps mentioned in Chapter 2, Section 2.4 can be summarized as show in Figure 12. As mentioned in the previous section, the memory interface utilization can be split as show below:

Table 7: DDR Utilization

Operation	Percentage of total DDR Busy time
Sequential Read	2%
Random Read (Source Address)	44%
Random Read (Destination Address)	44%
Write (Destination Address)	10%

The sequential read operations in Step 1 and Step 2, are performed by the DDR Controller for only 2% of total time that the DDR is busy. Similarly, the write operation in Step 10 occupies only 10% of the DDR time, while 98% of the time is spent doing the random reads from the source and the destination addresses.

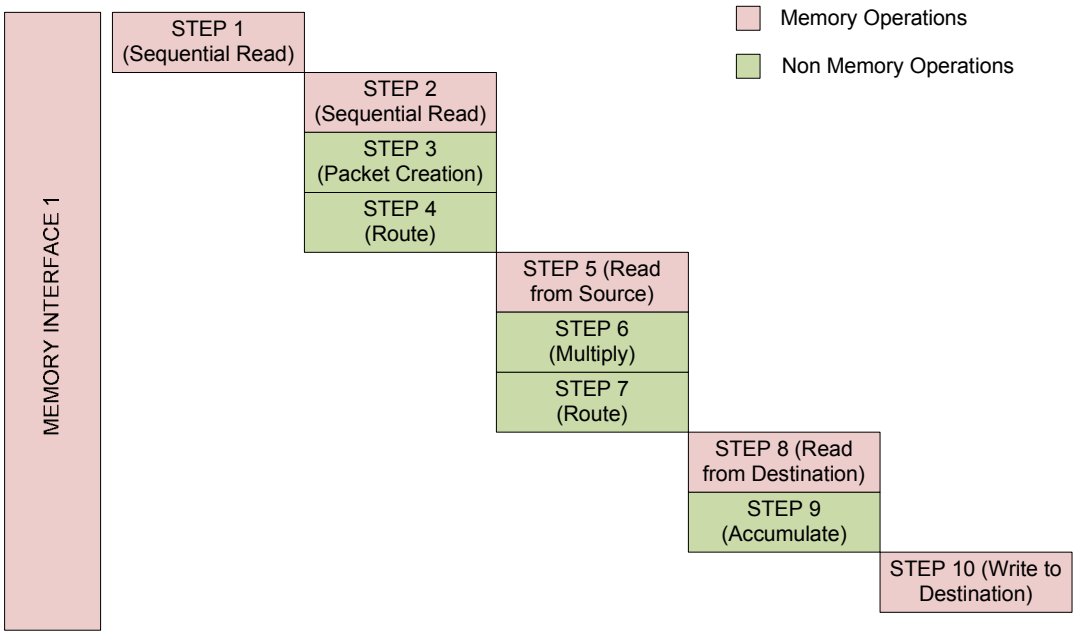

Figure 12: Parallelization using a single memory interface

The operations that require memory access are constrained by the presence of a single memory interface and have to be performed sequentially. If three memory interfaces were present and it would be possible to parallelize the steps as summarized in Figure 13. In this figure, for simplicity in representation, the random read operations are performed using Interface 2 and Interface 3 only. Realistically, it would be possible to perform random reads from all three interfaces, interleaving random reads and writes in Interface 1. Since the random read memory access latency is the largest, dividing the random reads between three interfaces, would decrease the computation time to 33% of the time taken with a single interface.

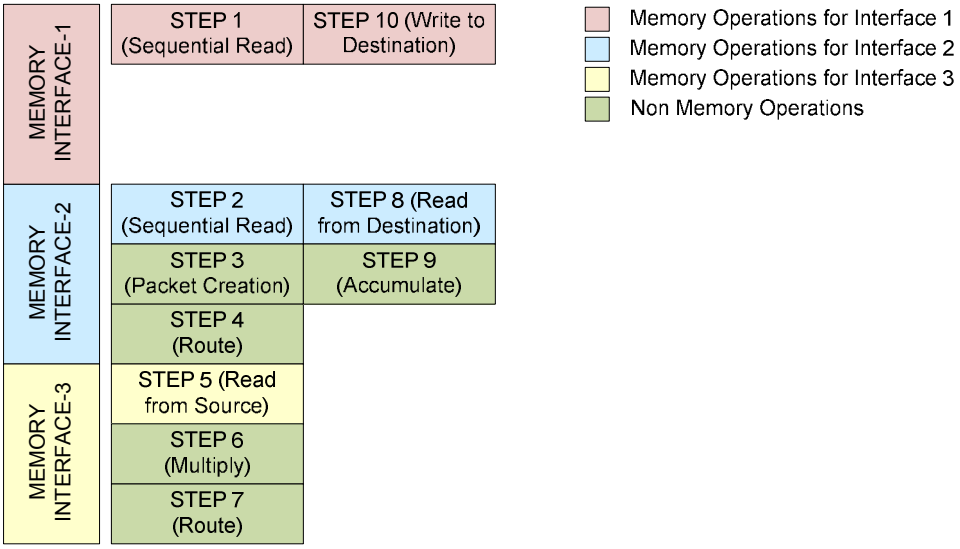

Figure 13: Parallelization using three memory interfaces

5.5 Comparison to Previous Work

In the SMVM implemented by DuBois et al. [10], sparse matrices are represented in the ELLPACK-ITPACK format. This format is a row-packet format and offers an efficient representation for matrices that can benefit from the compressed row format. The rank of the compressed matrix depends upon the number of non-zero elements that the most populated row contains. For the gradient matrix discussed in Chapter 2, this does not pose a potential problem because each row in the Gradient matrix contains only two elements. The divergence matrix, however is best represented in the compressed column format, given its sparsity pattern. The ELLPACK-ITPACK format is not an efficient format for the divergence matrix. The E2N data structure can store the sparse matrix in the compressed row or the compressed column format with equal ease. In the SMVM implementation by DeHon et al[6], the matrix is stored in the Compressed Sparse

Row structure. This too is efficient for storing the gradient matrix, but not for storing the divergence matrix.

In both [6] and [10] matrices are stored on the on-chip SRAM, and the computational capability of the FPGA is used to its maximum to achieve the highest floating point performance possible. Storing the matrices on the FPGA itself limits the matrix sizes to those that can fit on the low capacity on-chip memory, than on high-capacity external memory. For example, in [6], 16 Virtex II boards are used to fit a matrix with 460k non-zero elements in it. This data could fit on a single FPGA in our implementation. However, on-chip SRAM blocks can be accessed with a two clock cycle latency, and data is available at a much faster rate to the on-chip computational resources than from off-chip memory access. Accordingly, in [6] the maximum floating point performance achieved is 1500 MFlops and in [10] the maximum floating point performance achieved is about 400 MFlops.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

A hardware engine to perform SMVM has been implemented using a system of FPGAs boards, each board having a single off-chip memory interface. Although the SMVM example in consideration is for heat transfer, the design remains the same for any SMVM that arises from any PDE solution such as those for fluid flow, quantum mechanics, electromagnetism, gravitation, heat transfer and solid mechanics,. The two FPGA implementation performs eight times slower than the CPU. However, the performance of the FPGA system is easily scalable by adding more memory interfaces or more boards to the system. It is more intuitive to parallelize operations on an FPGA than it is on a cluster of CPUs, because parallelization is inherent to FPGAs. All the FPGAs in the system have the same configuration, and the data is stratified across their memories in a straight forward manner.

Future work involves building a system with multiple boards, each having multiple memory interfaces on it, to perform SMVM. The FPGA is primarily a prototyping device and hence has a limited clock speed. Once the memory bottleneck is somewhat overcome, it would be possible to design an ASIC for SMVMs and clock it at a higher frequency, that does not unnecessarily exceed the memory bandwidth.

BIBLIOGRAPHY

[1] Y. Saad, “Iterative Methods for Sparse Linear Systems,” SIAM, 2003, page 90

[2] G. Moore, “Cramming More Components Onto Integrated Circuits,” Electronics Magazine, Vol. 38, no. 8, April 19, 1965, pages 114-117

[3] K. Underwood, “FPGAs vs. CPUs: Trends in Peak Floating-Point Performance,” In Proceedings of the International Symposium on Field-Programmable Gate Arrays, pages 171-180, February 2004

[4] W. Wulf, S. McKee, “Hitting the Memory Wall: Implications of the Obvious,” Computer Architecture News, pages 20-24, 1995

[5] G. Wellein, G. Hager, T. Zeiser, “Basic principles of modern processors: Memory Hierarchy Optimization of Data Access,” April, 2005

[6] M. deLorimier and A. DeHon, “Floating-point Sparse Matrix-vector Multiply for FPGAs,” In FPGA ’05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays, pages 75–85, New York, NY, USA, 2005. ACM Press

[7] Zhou, Ling and Prasanna, Viktor, “Sparse Matrix-Vector Multiplication on FPGAs.” Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate array, pages 63-74. 2005

[8] J. Sun, G. Peterson and O. Storaasli, “Sparse Matrix-Vector Multiplication Design on FPGAs,” Field-Programmable Custom Computing Machines Conference, April, 2007

[9] Y. El-Kurdi, D. Fernández, E. Souleimanov, D. Giannacopoulos, W. J. Gross, “FPGA architecture and implementation of sparse matrix-vector multiplication for the finite element method,” Computer Physics Communications 178(8): 558-570 (2008)

[10] D. Dubois, A. Dubois, C. Connor, S. Poole, "Sparse Matrix-Vector Multiplication on a Reconfigurable Supercomputer," IEEE Symposium On Field-Programmable Custom Computing Machines

[11] Altera Whitepaper, “Designing and Using FPGAs for Double-Precision Floating-Point Math”

[12] http://tinyurl.com/alterade3board