Portable and Scalable FPGA-Based Acceleration of a Direct Linear System Solver

by

Wei Zhang

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of The Edward S. Rogers Sr. Department of
Electrical and Computer Engineering
University of Toronto

Abstract

Portable and Scalable FPGA-Based Acceleration of a Direct Linear System Solver

Wei Zhang

Master of Applied Science

Graduate Department of The Edward S. Rogers Sr. Department of Electrical and Computer Engineering

University of Toronto

2008

FPGAs are becoming an attractive platform for accelerating many computations including scientific applications. However, their adoption has been limited by the large development cost and short life span of FPGA designs. We believe that FPGA-based scientific computation would become far more practical if there were hardware libraries that were portable to any FPGA with performance that could scale with the resources of the FPGA. To illustrate this idea we have implemented one common supercomputing function: the LU factorization method for solving linear systems. This dissertation discusses issues in making the design both portable and scalable. The design is automatically generated to match the FPGA's capabilities and external memory through the use of parameters. We compared the performance of the design on the FPGA to a single processor core and found that it performs 2.2 times faster, and that the energy dissipated per computation is 5 times less.

Acknowledgements

I would like to thank Professor Jonathan Rose and Vaughn Betz for their guidance and motivation during the last two years. I have learnt a lot during this time, not just about FPGAs but about research in general.

I would like to thank my labmates in LP392 for the advice, encouragement and distractions.

I would like to thank my parents for their support throughout the years, no matter what I wanted to do. Finally, I would like to thank Sheena for taking the time to help me whenever I needed.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

As the logic and computational capacity of FPGAs has grown, they have become an attractive platform for accelerating many computations including scientific applications. The high level of parallelism and abundant flexibility available in the FPGA fabric offer the promise of significant speed up. A number of vendors offer platforms that enable a processor to offload computation to an FPGA-based accelerator including XtremeData [2], SRC [3], and Cray [4]. However, adoption of these FPGA accelerators by the scientific computing community has been limited because the creation of an FPGA design is difficult and time consuming, and outside the skill set of the typical scientific computing user. In addition, once a design has been created for one specific FPGA chip and board, the same design cannot be easily transferred to another. The design is locked onto that FPGA-based platform because it typically has a specific memory architecture that soon becomes outdated.

In contrast, software has a much lower development cost. Once a software application is completed, it can easily be recompiled for newer machines. This permits software programmers to develop and maintain rich libraries that solve important problems. Scientific computing users need not be highly skilled in creating optimized code because they can simply use the functions in these libraries. In hardware, IP cores do allow some design

reuse, but at a much lower level of abstraction than with high level software libraries.

One method that attempts to make FPGA programming more accessible is to employ high-level languages and synthesis tools that map software directly to an FPGA. Examples include Handel-C [5], Catapult C [6], and SystemC [7]. However, this approach is often not adequate to create an efficient hardware design from complex code as the programmer typically has to write the code in a stylized manner with the final architecture of the system in mind to obtain good performance. Thus software libraries cannot be easily and efficiency translated to FPGAs without significant manual manipulation.

In this work we present an alternative solution for making FPGA-based computation more accessible by creating a computational "library" that is portable to any FPGA platform with minimal effort. The key second feature of the library is that its performance should also scale with the capabilities and resources of the FPGA. Given an FPGA with more capacity and faster elements, the library performance should improve without extra effort from the designer. By creating a *portable* and *scalable* library, we can drastically reduce the development cost and increase the life span of the design, thus making it more attractive to scientific computing users.

Common software libraries for scientific computing include matrix manipulation packages such as BLAS [18], SAT solvers, and linear program solvers. If an equivalent library existed for FPGAs, it could enable broader adoption of FPGA acceleration.

1.1 Research Goals

This research lays out a framework to create such a hardware library, by illustrating the design issues and efforts needed to build *one* such compute engine. The compute engine in this library should have two key features:

1. Portability: the ability to maintain functionality while executing on different FPGA-based platforms. In order to achieve portability, the engine has to handle memory

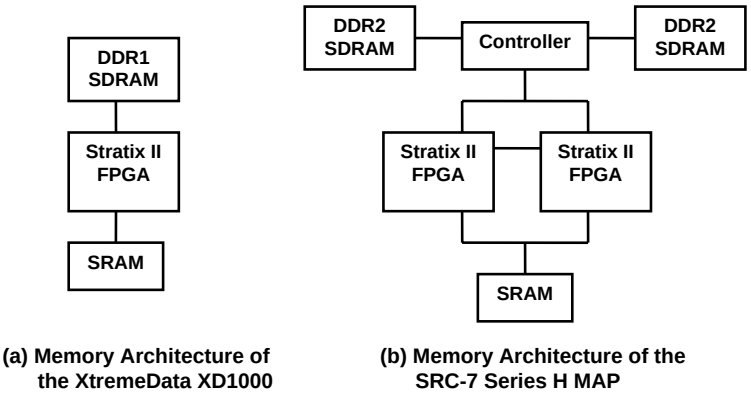

Figure 1.1: Two Different Commercial Memory Architectures of FPGA Accelerators

architectures of different platforms. For example, Figure 1.1(a) shows the XtremeData XD1000 FPGA accelerator, which has one Altera Stratix II FPGA. It has a 4MB On-Board ZBT SRAM memory and a 128 bit-wide DDR-333 SDRAM memory [2]. Figure 1.1(b) shows another FPGA accelerator, the SRC-7 Series H MAP, which has two Altera Stratix FPGAs, a 64 MB 8 bank On-Board SRAM and two DDR2 SDRAM memory banks [3]. Our goal is to build a compute engine generator that will be able to generate engines that can use these different memory architectures to perform the same functionality.

2. Scalability: the ability to improve performance by using all of the available resources on the FPGA while maintaining the same functionality. The compute engine should perform the same functionality faster on an FPGA with more resources than one with fewer resources.

To achieve these two features, a compute engine generator (which is software that produces synthesizable code) is created to automatically produce the engine that can interact with the targeted memory architecture while taking advantage of all the resources on the FPGA to improve performance. Parameters containing information about the FPGA-based platform are provided to the generator so that it can create a customized engine. The user only has to change these parameters when moving to another platform.

Our focus application is the solution of systems of linear equations, as this is a very

common problem and the computation time is high for large systems. We have created a generator that automatically creates a portable and scalable FPGA computer engine for the LU factorization method [23] to solve a linear system. The generator and engine are highly parameterized to permit any size of matrix (up to the external memory capacity) and to make use of any size of FPGA.

1.2 Organization

This thesis is organized as follows. Chapter 2 provides background on solving linear systems and summarizes previous work on using FPGAs to accelerate this computation. Chapter 3 outlines the architecture of our compute engine. Chapter 4 describes the broad space of parameters for which the generator can generate engines to achieve portability and scalability. Chapter 5 discusses the experimental results. Chapter 6 concludes the thesis and suggests future extensions to this work.

Chapter 2

Background

In this chapter we provide the background material related to our goal of a portable and scalable hardware linear equations solver. First, we outline the LU factorization method for solving linear equations, including the so-called blocking version that enables any arbitrary problem matrix size on an FPGA. Next, we review software linear equation solvers. Then we describe the FPGA architecture that we focus on in this work and contrast our goals to what is currently available in industry standard IP cores, which are also portable and scalable. Finally, we survey previous work on linear equation solvers on FPGAs.

2.1 Solutions of Systems of Linear Equations

A system of M linear equations with N unknown variables, as shown in Equation (2.1), is often represented in a matrix and vector form, as shown in Equation (2.2). The coefficients of the variables in each linear equation are represented in each row of an M x N matrix (A) and they are multiplied by the N-element vector of unknown variables (x). A solver must determine the values of x for which the product generates the M-dimensional constant (b). Given a system of linear equations, there can be one solution, an infinite number of solutions or no solution. For this research, we focus on systems of

linear equations where the matrix is square (M = N) and nonsingular; this implies [24] that the matrix is invertible and there is only one solution to the system.

$$\begin{aligned} a_{1,1}x_1 + a_{1,2}x_2 + \cdots + a_{1,N}x_N &= b_1 \\ a_{2,1}x_1 + a_{2,2}x_2 + \cdots + a_{2,N}x_N &= b_2 \\ &\vdots \\ a_{M,1}x_1 + a_{M,2}x_2 + \cdots + a_{M,N}x_N &= b_M \end{aligned} \tag{2.1}$$

$$Ax = b$$

$$A = \begin{bmatrix} a_{1,1} & a_{1,2} & \dots & a_{1,N} \\ a_{2,1} & a_{2,2} & \dots & a_{2,N} \\ \dots & \vdots & \ddots & \vdots \\ a_{M,1} & a_{M,2} & \dots & a_{M,N} \end{bmatrix} x = \begin{bmatrix} x_1 \\ x_2 \\ \dots \\ x_N \end{bmatrix} b = \begin{bmatrix} b_1 \\ b_2 \\ \dots \\ b_M \end{bmatrix} \tag{2.2}$$

There are two main classes of linear equation solvers: iterative and direct. Iterative solvers begin with an initial guess for the solution vector, x, and then refine it until the error is sufficiently small. To refine the guess vector, the coefficient matrix A is multiplied by the guess vector to produce a new vector. The difference between this new vector and the constant vector b is called the error vector. Based on the specific iterative solver, a new guess for the solution vector is obtained based on the previous guess vector and the error vector. Some examples of iterative solvers are the conjugate gradient [23] and the Jacobi iterative method [23]. For an iterative solver, the bottleneck in each refinement of the guess vector is the matrix-vector multiplication, which is $O(N^2)$ operations. In general, the number of times the iterative solver has to refine its guess vector is less than and independent of N for large systems. Thus the overall computation is typically stated as $O(N^2)$.

By contrast, direct solvers [24] manipulate the matrix and solution vector until the solution can be easily computed. Examples of direct solvers are Gaussian elimination

[24] and LU factorization [24]. The direct solution requires $O(N^3)$ operations, which is more than iterative solvers. However given a solution exists, it can always compute the solution. In contrast, the error for the iterative solver will not always converge to a sufficiently small value for all types of matrices. Some iterative solvers that do guarantee converge require N iterations and therefore, its computation complexity is $O(N^3)$, which is the same as direct solvers. Thus, both iterative and direct solvers are widely used. In this research, we focus on LU factorization, which is a direct solver. More detail on methods of solving linear equations can be found in [24] and [23].

2.1.1 LU Factorization Overview

The LU factorization method directly solves for x by breaking the coefficient matrix A into two matrices, $A = LU$. One of those matrices, called L, is a lower triangular matrix which has the diagonal elements equal to 1 and all elements above the diagonal equal to 0; the other matrix, called U, is an upper triangular matrix which has the elements below the diagonal equal to 0. Using Equation (2.3), if we set $y = Ux$, a forward substitution can be performed to compute y from $Ly = b$. Then a backward substitution can be performed to compute x from $Ux = y$. The most time consuming computation in this algorithm is the factorization of the coefficient matrix, which is the determination of the matrices L and U such that $A = LU$, as this requires $O(N^3)$ operations for a N x N matrix A. It is this factorization computation that this work seeks to accelerate on an FPGA. To save space, we do not need to store the L and U matrices separately; since we know the upper part of the L matrix (the diagonal is 1 and all elements above the diagonal is 0) and we know the lower part of the U matrix (all elements below the diagonal is 0), we do not have to explicitly store their values. To further save memory space, the solution of the L and U matrix can be stored in the same memory location where the A matrix was originally stored [24]. The following sections outline the steps required to perform the factorization.

$$LUx = b \tag{2.3}$$

2.1.2 Simple LU Factorization

A pseudo-code outline for a simple LU factorization algorithm is given in Algorithm 1 [24]. There are two kinds of operations that must be performed: the first is the division of all the column elements below the diagonal, $a_{k+1,k}$ to $a_{N,k}$, by the diagonal element, $a_{k,k}$. The second is the multiplication of all the column elements below the diagonal, $a_{k+1,k}$ to $a_{N,k}$, by the row element, $a_{k,j}$, and the subsequent subtraction of the result from all the column elements below the row element used in the multiplication, $a_{k+1,j}$ to $a_{N,j}$. The multiplication and subtraction are repeated for j from $k+1$ to N. All these operations are repeated for the next diagonal element until the last diagonal element is reached.

Algorithm 1 Pseudo-code for a simple LU factorization of N x N A matrix

```
for k = 1 to N − 1 do {for each diagonal element}
  for i = k + 1 to N do {for each element below it}
    a_{i,k} ← a_{i,k}/a_{k,k} {normalize}
  end for
  for j = k + 1 to N do {for each column right of current diagonal element}
    for i = k + 1 to N do {for each element below it}
      a_{i,j} ← a_{i,j} − a_{i,k} × a_{k,j}
    end for
  end for
end for
```

2.1.3 Block LU Factorization

For the simple LU factorization method described above, all of the elements in the matrix must be accessible during the computation. For problem sizes of interest, matrix size (N) is at least 10,000 x 10,000 single precision numbers (smaller than this the computation times on processor are reasonably fast), which requires roughly 0.4 GBytes of memory. This is far too large to store on a chip, either an FPGA or a processor's cache, and so

the matrix must be stored in off-chip memory. Thus all practical approaches must deal with the limitation to off-chip memory bandwidth. The common approach to deal with this is to perform the computation in a "blocked" manner - to bring on-chip subsections of the coefficient matrix A, each of size N_b x N_b and perform as many computations on that data as possible to minimize the number of times data has to be fetched from off-chip memory. We refer to a block as being "updated" to mean that computations are performed on the block to produce new values.

There are three common variants of the block LU factorization: "right-looking", "left-looking" and "Crout" [23]. In this work we employ the "right-looking" version, and so we focus on it here. In this method the current block being updated uses elements from the left-most and top-most block in its row and column. Figure 2.1(a) shows the blocks needed to update an example block in a matrix. In the figure, the current block is colored black, the left-most block is the grey block to the left of the black block and the top-most block is the grey block above the black block. Each block of the matrix has a size of N_b x N_b. With this blocking method there are four kinds of computations performed on the blocks based on the following cases:

Case 1: all three blocks (current, left-most, and top-most) are the same physical block.

Case 2: the current block is the same as left-most block.

Case 3: the current block is the same as the top-most block.

Case 4: all three blocks are different.

Figure 2.1(b) shows an example matrix in which the blocks are labeled with with the corresponding case type. The computation for these blocks are similar to the simple LU factorization algorithm described in Algorithm 1, except the loop indices are different and some elements are obtained from the left-most and top-most blocks. The pseudo-code for all the cases is shown in Algorithm 2, where $a_{i,j}$, $l_{i,j}$, and $u_{i,j}$ represent elements in the current block, left-most block and top-most block respectively. We will refer to the series

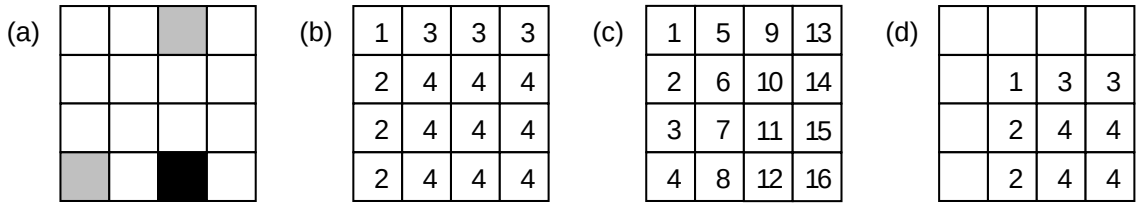

Figure 2.1: (a) blocks required, in grey, to update the black block; (b) type of computation for each matrix block in the first block pass; (c) order of blocks updated in the first pass; (d) computation performed in the second block pass.

of operations performed on the matrix block as the *block computation*. For case 1, the operations performed are the same as the simple LU factorization, except N is replaced by N_b. For case 2, the diagonal element in the top-most block, $u_{k,k}$ is divided from column elements below it in the current block, $a_{1,k}$ to $a_{N_b,k}$. Then the column elements in the current block, $a_{1,k}$ to $a_{N_b,k}$, is multiplied by the row element in the top-most block, $u_{k,j}$. The result is subsequently subtracted from the column elements in the current block below the row element used in the multiplication, $a_{1,j}$ to $a_{N_b,j}$. The multiplication and subtraction are repeated for j from $k+1$ to N. All these operations are repeated for the next diagonal element in the top-most block until the last diagonal element is reached.

Case 3 and 4 only perform the multiplication and subtraction and do not perform the division, as an earlier block operation will have already normalized the necessary elements. For case 3, the column elements in the left-most block below the diagonal, $l_{k+1,k}$ to $l_{N_b,k}$, is multiplied by the row element in the current block, $a_{k,j}$. The result is subtracted from the column elements in the current block that is below the row element used in the multiplication, $a_{k+1,j}$ to $a_{N_b,j}$. The multiplication and subtraction operations are repeated for j from $k+1$ to N. These operations are repeated for k from 1 to N_b. For case 4, the column elements in the left-most block, $l_{1,k}$ to $l_{N_b,k}$, is multiplied by the row element in the top block, $u_{k,j}$. The result is subsequently subtracted from the column elements in the current block that is below the row element used in the multiplication, $a_{1,j}$ to $a_{N_b,j}$. The multiplication and subtraction are repeated for j from 1 to N. These

Algorithm 2 Code for all 4 cases of block LU factorization

Case 1:
same as simple LU factorization with N_b instead of N {See Algorithm 1}

Case 2:
for $k = 1$ to N_b **do** {for each diagonal element in top-most block (u)}
  for $i = 1$ to N_b **do** {for each element below it in current block (a)}
    $a_{i,k} \leftarrow a_{i,k}/u_{k,k}$ {normalize}
  end for
  for $j = k + 1$ to N_b **do** {for each column right of current diagonal element}
    for $i = 1$ to N_b **do** {for each element below it in current block (a)}
      $a_{i,j} \leftarrow a_{i,j} - a_{i,k} \times u_{k,j}$
    end for
  end for
end for

Case 3:
for $k = 1$ to N_b **do** {for each column in left-most block (l)}
  for $j = 1$ to N_b **do** {for each column in current block (a)}
    for $i = k + 1$ to N_b **do** {for each element below it}
      $a_{i,j} \leftarrow a_{i,j} - l_{i,k} \times a_{k,j}$
    end for
  end for
end for

Case 4:
for $k = 1$ to N_b **do** {for each column in left-most block (l)}
  for $j = 1$ to N_b **do** {for each column in top-most block (u)}
    for $i = 1$ to N_b **do** {for each element below it in current block (a)}
      $a_{i,j} \leftarrow a_{i,j} - l_{i,k} \times u_{k,j}$
    end for
  end for
end for

computations are repeated for k from 1 to N_b. For a large matrix, case 4 is the most common and dominates the computation time.

Before a block can be updated (where new values are computed), all the other blocks that are required in the computation must be updated already. So one cannot randomly update the blocks and Figure 2.1(c) shows one possible order in which the blocks can be updated. After the first block pass in which every block is updated, the blocks in the first block column and block row have the final solution. The remaining blocks, which are all the case 4 blocks in the first block pass, are updated again. The current block updated now use elements from the second left-most and the second top-most blocks. It can be thought of as the remaining blocks forming a new matrix for which the above computations are repeated. Figure 2.1(d) shows which of the four cases in the above computation the remaining blocks fall under. After updating all the blocks and thus completing another block pass, the first column and the first row in the remaining blocks have the final solution. The new remaining blocks are updated again. This process repeats until no blocks are left, requiring N/N_b block passes in total.

2.2 Software Linear System Solvers

There exist many software libraries that can solve systems of linear equations. The BLAS (Basic Linear Algebra Subprograms) [18] are routines that perform various vector and matrix operations, which are the building blocks for solving linear equations. There are three levels of BLAS routines based on level of complexity. Level 1 BLAS perform scalar, vector and vector-vector operations, which are $O(N)$. Level 2 BLAS perform matrix-vector operations, which are $O(N^2)$, and Level 3 BLAS perform matrix-matrix operations, which are $O(N^3)$. The LAPACK (Linear Algebra PACKage) [17] is a library of routines that solves systems of linear equations, least-squares solutions of linear systems of equations, eigenvalue problems, and singular value problems. It uses the BLAS routines

to perform its computation. There is an LU factorization routine in LAPACK.

Intel Corporation has released its own version of LAPACK called Intel MKL (Math Kernel Library) [8]. The MKL library also performs fast Fourier transforms (FFT), vector trigonometry and vector statistic routines. It is highly optimized for Intel processors and uses the processor's Streaming SIMD Extensions (SSE), which is a single instruction, multiple data instructions set for Intel processors. Both Intel [9] and AMD [10] support SSE and its newer incarnations. The latest SSE (SSE4) on Intel processors have eight 128 bit registers on each core. Each register can be broken up to store four single precision (32 bit) or two double precision (64 bit) floating point elements. Thus it can perform four single precision or two double precision floating point operations simultaneously on these registers. This feature enables a large peak floating point performance potential for software. GFLOPS (FLoating point Operations Per Second in millions) is a common metric for this performance and it is calculated by taking the number of operations needed to perform a computation divided by the total execution time. To find the peak GFLOPS of a platform, one can estimate the maximum floating point operations it can perform per cycle and multiply it with the operating frequency. For example, the Xeon 5160 dual core 3.0 GHz processor has two cores and at maximum each core can perform four single precision floating point multiplication and addition/subtraction operations per cycle. Thus, this processor has a peak single precision performance of 48 GFLOPS (3GHz $\times$ 2 cores $\times$ 2 types of operations $\times$ 4 single precision operations per cycle).

2.3 FPGA Architecture Capabilities

An FPGA consists mainly of look up tables (LUTs) and registers with programmable routing. These LUTs are very flexible and can implement any logic function. As IC fabrication process has improved, the number of LUTs per chip has increased in FPGAs thus allowing more computational capacity. In addition, other hard components have

been added that have specific functions and offer speed and area improvements when these functions are used. These new additions include multipliers and on-chip memory blocks. For this research, we will use the Altera Stratix series of families of FPGA in our implementation. Figure 2.2 shows the FPGA architecture of the Altera Stratix III [1]. The Stratix III consists largely of adaptive logic modules (ALMs) with 10 ALMs forming a logic array block (LAB). The ALM is a fracturable LUT with two dedicated adders and registers. The DSP blocks support 9x9, 12x12, 18x18 or 36x36 bit multipliers. There are eight 18x18 bit hard multipliers in a DSP block. The Stratix III on-chip memory consists of M9K blocks, which has 9,216 bits of memory, M144K blocks, which can hold 147,456 bits of data, and MLAB blocks, which can store up to 640 bits of data. The MLAB is implemented using a LAB with all of its ALMs converted into small memory block. The LUT is effectively a memory block that can be configured to implement any function. The M9K and M144K blocks are dedicated on-chip memory blocks and they are true dual-port memory blocks, where any two operations, read or write, can be performed per cycle. The MLAB is only a simple dual-port, where one read and one write can be performed per cycle. All three types of memory blocks have maximum operating frequencies of 600 MHz. The largest Stratix III FPGA has 17 Mbits of on-chip memory, which can store a maximum size single precision matrix of 721x721. Thus, it is necessary for us to use off-chip memory to ensure matrices of larger dimensions can be solved. The FPGA has PLLs, DLLs and I/O banks that are used to interact with off-chip memory. The I/O banks provide input and output connections to on-chip memory while the PLLs and DLLs are used to align to the clock of the off-chip memory.

2.3.1 Vendor Supplied IP Cores

FPGA vendors, such as Altera and Xilinx, offer IP cores that implement specific functions that users employ in creating their design. The cores consist of functions for DSP algorithms, communication modules, interface modules, and soft processors. Similar to

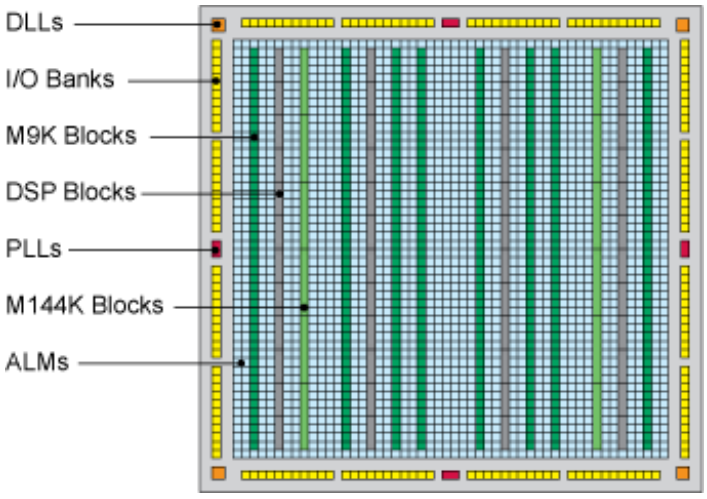

Figure 2.2: Altera Stratix III FPGA architecture [1]

our generator, the IP cores generator uses parameters and can create portable and scalable cores. The difference between our work and the IP cores is the size and complexity. Figure 2.3 shows examples of IP cores and how they compare to our engine in terms of size and complexity. Typically one IP core will occupy a small portion of the resources on an FPGA while our computation engine can use up all the resources on an FPGA. The cores are typically used as building blocks in the algorithm while our engine implements the whole algorithm. In terms of complexity, our engine is significantly more complex than today's typical IP cores. Many cores like the arithmetic functional units have little or no control logic and simply perform some manipulation of the input data. More complicated cores such as the processor or off-chip memory controller have some simple logic to control its datapath. Our engine generator creates the complete datapath and control logic to perform a complex algorithm including communication with off-chip memory. Our engine uses several Altera IP cores to generate components in the datapath such as floating point functional units and off-chip memory controller. One could classify our engine as a "super" IP core.

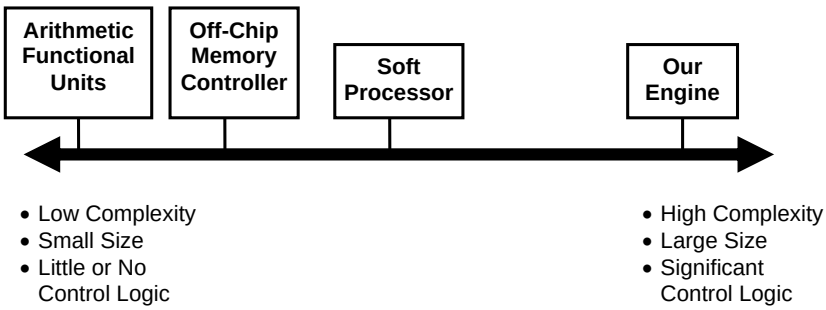

Figure 2.3: Comparison of the complexity and size between IP cores and our computation engine.

2.4 Prior Work

This section surveys some prior work in solving systems of linear equations on FPGA hardware and highlights the differences with the work in this thesis.

The work in [28] compared the performance of floating point matrix operations on FPGAs and CPUs. This work used data from five FPGAs and three CPUs from 1997 to 2003, which is specified in Table 2.1. However, the impact of newer CPUs with multi-cores was not evaluated. It compared FPGAs and CPUs based on one routine from each level of the BLAS library: dot-product, matrix-vector multiplication and matrix-matrix multiplication. The software measurement was obtained from BLAS routines and the FPGA measurement was estimated based on implementations using multiply-accumulate units. This work showed that a state-of-the-art FPGA has a higher peak floating point performance for various matrix operations than a CPU, and that an FPGA also has a higher performance increase trend than CPU, so over time, the performance advantage should increase. It also concluded that FPGAs can obtain a higher performance with less memory bandwidth and on-chip storage than CPU. Thus, FPGAs have an advantage in the future as they will not be limited by memory bandwidth as soon as CPUs.

We will review some prior research on implementing linear equation solvers in FPGAs. For iterative solvers, the bottleneck is a matrix-vector multiplication as explained in Section 2.1. The work in [29] and [21] implemented double precision sparse matrix-vector

Table 2.1: FPGAs and CPUs analyzed in [28].

Year	FPGA	CPU
1997	Xilinx XC4085XLA-09	Pentium-II 266 MHz
1999	Virtex 1000-5	
2000	Virtex-E 3200-7	Athlon 1.0 GHz
2001	Virtex-II	
2003	Virtex-II Pro 100-6	Pentium-4 3.06 GHz

multiplication for FPGA. A sparse matrix is a matrix that has many zero elements and a dense matrix is a matrix with few zero elements.

In [29], the matrix and vector is broken up into blocks so that the matrix and vector can be any size and are not limited by on-chip memory. A block matrix-vector multiplication method is performed until all the blocks have been multiplied. A matrix-vector multiplication can be broken up into one large dot-product operation per row of the matrix. Each such operation can be further broken into smaller dot products with additional logic needed to accumulate the results. The design in [29] has one dot-product operator with accumulation logic so that it can effectively perform a larger dot-product operation for the entire row of the matrix block. Each row of the matrix is computed in this manner until all the rows in the matrix have been used. Without the accumulation logic, the dot-product operator has to have an input size equal to the matrix block size. With the use of accumulation logic, the design can use a small input size dot-product operator, which uses less resources at the cost of a longer latency than a large dot-product. Also, the block size of the matrix-vector multiplication can be independent of the input size of the dot-product operator.

The dot-product operation is implemented by multipliers connected to an adder tree structure which accumulates all the results into one value. Figure 2.4 shows the block diagram of a size four dot-product operator, which performs on two input vectors of size four and produces one result. The design in [29] was targeted on a Xilinx Virtex-II Pro XC2VP70 and the performance was limited by the memory bandwidth. For a memory

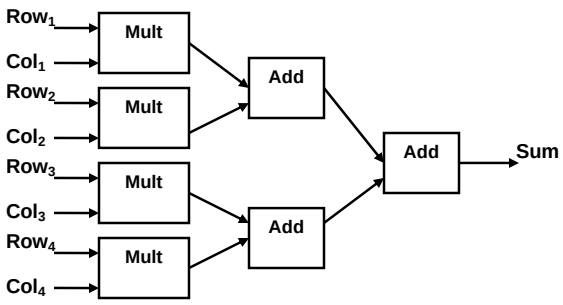

Figure 2.4: Block Diagram of a Dot-Product Operator

bandwidth of 8GB/s, this design achieves at least 0.35 GFLOPS in double precision depending on the matrix.

In [21], the matrix-vector multiplication was calculated using a bidirectional ring structure with many processing elements (PE), each of which contains a multiply accumulate operator, and a control element (CE). Figure 2.5 shows the block diagram of the structure. The CE uses switches (SW) in the ring structure to send out instructions to all the PEs. The CE has its own on-chip memory block to store the instructions that it sends out to the PEs. The PE operates on its own on-chip memory block based on the instructions. This ring structure mimics a SIMD processor.

The design does not use any off-chip memory. The matrix has to be divided among the PEs and the appropriate instructions have to be determined beforehand. Memory initialization files, which initializes an on-chip memory block to specific values, are used to load the matrix elements into PEs and instructions into the CE when the FPGA is programmed. Thus solving a different matrix requires the matrix to be partitioned and instruction scheduling to be performed for each problem matrix. New memory initialization files must produced and the FPGA will have to be reprogrammed, which is inconvenient for the user. For the matrices used in [21], the matrix partition and instruction scheduling requires a significant amount of time that is much larger than the actual execution time of the iterative solver. These overheads need to be significantly reduced before this design can be used. Because the matrix has to be stored all in on-chip

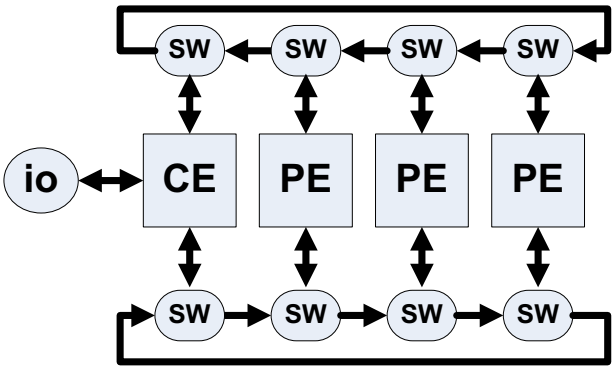

Figure 2.5: Bidirectional Ring Structure in [21]

memory, the size of the matrix that can be solved is limited. On a Virtex II 6000-4, this design achieved 1.5 GFLOPS. This design allows multiple FPGAs to be used and using 16 Virtex IIs, it achieved a total of 12 GFLOPS in double precision.

The work in [26] implemented a conjugate gradient (CG) and a Jacobi iterative solver for sparse matrices on FPGA in double precision. This work used a SRC-6 Workstation which has dual 2.8GHz Xeon processors with a 5120KB cache and 1GB of RAM and two Xilinx Virtex II 6000 FPGAs. For the CG solver, only the matrix-vector multiplication was implemented on the FPGA while the remainder of the algorithm was executed in software from the SPARSKIT library [27], which has routines for sparse matrix computation. For the Jacobi iterative solver, the whole algorithm was implemented on the FPGA. Both designs need to perform a matrix-vector multiplication, which is the bottleneck of the algorithm. To perform the multiplication, both designs use a dot-product operator and additional logic to accumulate the results similar to the work in [29]. The entire matrix is stored in on-chip memory on the FPGA and so there is a limit on the matrix size. The matrix size is limited to 4,096 for the CG solver and 2,048 for the Jacobi iterative solver. By having the whole matrix in on-chip memory, it reduces the memory bandwidth requirement of the design as the data only has to be loaded once when performing the entire algorithm and so performance is not memory bandwidth limited. This work

reported a speed up of 2.4 for the CG using the Virtex II over running the entire CG algorithm from SPARSKIT on the 2.8 GHz Xeon processor. The Jacobi iterative solver achieved a speed-up of 2.2 using the Virtex II over software from SPARSKIT running on the 2.8 GHz Xeon processor.

The work in [25] implemented a complete conjugate gradient (CG) for dense matrices on the FPGA in single precision. To perform the matrix-vector multiplication, the design implemented one dot-product operator, similar to the one in Figure 2.4, that operates on a whole row of the matrix. The input size of the dot-product operator is the size of the matrix, N. Each row of the matrix is sequentially operated in turn to complete the matrix-vector multiplication. Thus, a large amount of resources are dedicated to creating such a large dot-product operator and the input size of the dot-product operator limits the matrix size. All the data necessary to perform the CG algorithm is stored in on-chip memory. The design is fully pipelined and has a throughput of one iteration of the CG algorithm per cycle. However, if there is only one matrix to solve, most of time the components in the design are waiting for the next iteration. To maximize performance, multiple problem matrices with the same matrix size have to be solved at the same time to fully utilize the components in the design. On a Xilinx Virtex II 6000 with multiple matrices, the performance achieved is 5 GFLOPS in single precision and the matrix is limited to a size of 16. For a Virtex 5 LX330, the performance achieved is 35 GFLOPS in single precision and the matrix is limited to a size of 58.

In all these prior works for iterative solvers except for [29], a limit is imposed on the matrix size based on the on-chip memory capacity of the FPGA. For [29], there is no reuse of matrix data. For each iteration of the solver, the matrix has to be loaded from off-chip memory onto the FPGA to perform one matrix-vector multiplication. Thus, the memory bandwidth is $O(N^2)$ while the computation complexity is also $O(N^2)$; therefore, a large memory bandwidth is needed in order to obtain high performance. The performance is likely to be memory-bandwidth limited. For matrices with limited size, the entire matrix

can be stored on-chip and reused for each iteration of the solver. Since the matrix only has to be loaded once, the memory bandwidth requirement can be amortized across all the iterations of the algorithm. Thus, for the same amount of memory bandwidth, more computation can be performed. Thus the performance is limited by the computation and not by memory bandwidth. It seems that for iterative solvers, either the matrix size is limited or the performance will be bandwidth limited.

For direct solvers, the computation complexity order is higher, $O(N^3)$, than iterative solvers for the same matrix. Because the computation complexity is higher than the memory bandwidth required to load the matrix, performance will be limited by computation and not by memory bandwidth. Direct solvers are commonly used to solve dense matrices. Iterative solver either require less computation but do not guarantee convergence or require the same require the same order of computation as direct solvers to guarantee convergence. Therefore, direct solver is still very useful today in scientific applications. For these reasons, we decided to implement a direct solver.

The work in [30] implemented a direct solver using the same LU factorization method employed in our work. It used a circular linear array of processing elements (PEs) in double precision as shown in Figure 2.6. One of the PEs (PE_0) has a divider while the other PEs have a multiplier and adder. Matrix elements are passed from PE to PE starting with PE_0. PE_0 performs the normalization of the column elements while the other PEs perform the matrix multiple and subtraction operations. The implemented design stores all the matrix elements in on-chip memory and so it imposed a limit on the matrix size. A blocking version to remove the matrix size limit was proposed, but not implemented, in [20]. This design achieved 4 GFLOPS in double precision on a Virtex-II Pro XC2VP100. Comparing to software from an AMD optimized library called ACML [11] running on a 2.2 GHz Opteron with a L2 cache of 1MB, this work reported a speed up of about 1.2 in double precision using a Virtex-II Pro over software.

The work presented in this thesis can solve systems of linear equations of any size up

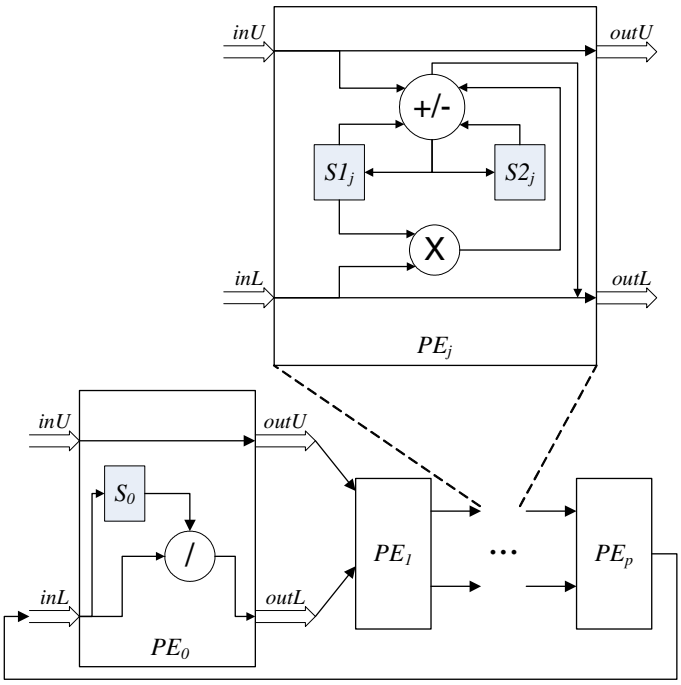

Figure 2.6: LU Factorization Design Block Diagram in [30]

to the capacity of the off-chip memory of the system, which is an important feature as it is the largest matrices which most need accelerated solutions. Many previous works do not mention external memory and some simply provide a bound on the required memory bandwidth. In contrast, this thesis explicitly considers external memory and outlines how portability and scalability can be achieved for different FPGAs with different external memories.

2.5 Summary

In this chapter, we have outlined the LU factorization algorithm including a blocking version that will enable processing of arbitrary matrix sizes on an FPGA. We discussed software linear equation solvers and summarized prior research for linear equation solvers on an FPGA. Almost all FPGA designs had limitations on the matrix size.

Our engine, which will be discussed in detail in the rest of this thesis, is analogous to FPGA vendor supplied IP cores in that it is portable and scalable. However, our engine is significantly more complex than the existing IP cores, such that it can be considered a "super" or next-generation IP core.

Chapter 3

Hardware Implementation

In this chapter, we will discuss the hardware design of our compute engine. We will start with a high level overview and then will describe the components of the engine in detail. Finally, we will discuss the verification process for our engine.

3.1 High Level Design Overview

The goal of this research is to create a highly parameterized LU factorization hardware compute engine for floating-point matrices (primarily 32-bit single precision) that is scalable to different sizes of FPGA and portable to different memory systems. The matrices in need of solution acceleration are very large and require sufficient storage; a single precision 10,000 x 10,000 matrix requires approximately 0.4GB of memory. As explained in Section 2.3, typical modern FPGAs do not have enough on-chip memory to store these large matrices. Thus, a key feature of our approach is to employ large off-chip memories (we will assume a DDR2 SDRAM) to store the large input matrices.

We will employ the *block* LU factorization method described in Section 2.1.3, where blocks of the large matrix are brought into on-chip memory and processed separately to make efficient use of off-chip memory bandwidth. In the foregoing we assume that the input matrix is square of size N x N and has a single solution. The matrix is broken

into square blocks of size N_b x N_b. The result of the LU factorization will be stored in the same location as the input matrix on the off-chip memory.

Figure 3.1 shows a high level diagram of the compute engine, which performs two main tasks. The first is *data marshalling*, which is the loading and storing of matrix blocks onto the FPGA from the off-chip memory. The second function is the actual *computation* on each set of blocks brought into the FPGA.

The data marshalling is handled by the Data Transfer Unit (DTU) and the Memory Controller (MemC) modules as shown in Figure 3.1. The computation is performed by the LU Processing (LUP) module, which is controlled by the LU Controller (LUC) module. The Marshalling Controller (MC) is responsible for issuing commands to the DTU and LUC and to provide synchronization between the two tasks. The MC controls which blocks of memory to load and store and which series of operations are performed on the loaded blocks to complete the LU factorization. The matrix data flows from the off-chip memory through MemC and DTU to the LU Processing module, where computations are performed. The updated data are then written back into the off-chip memory through DTU and MemC. The generator source code that creates these modules are listed in Appendix A.

There are two clocks in this design; one clock controls the off-chip memory controller and part of the DTU; the second clock controls the computation and a portion of the DTU. The separation into two clocks is important for scalability as it is unlikely the maximum speed of the off-chip memory and that of the on-chip memory and computation units will be related.

The key inputs to the compute engine are: the size of the matrix (N), the starting memory address of the matrix in off-chip memory and a start signal. Before the computation starts, the coefficient matrix, A, has to be loaded into the off-chip memory by some other agent in the system. The L and U solution matrices are written back into the same location in off-chip memory where the A matrix was stored. An output done

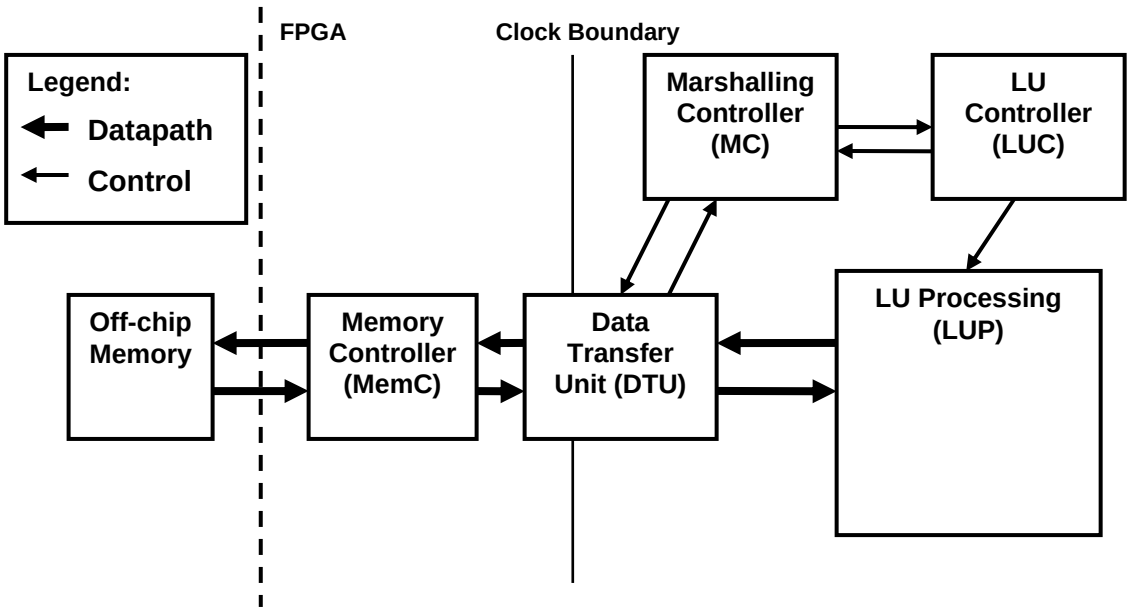

Figure 3.1: Block Diagram Block-Based Linear System Solver

signal is asserted when the LU factorization finishes.

The LU factorization compute engine is highly parameterized to enable the portability and scalability as described in Section 1.1. The modules of the compute engine are automatically created from a *compute engine generator* based on a set of input parameters. These parameters and the generator will be discussed in Chapter 4.

3.2 Setup of Computation

In creating our compute engine, we observed that the time required to transfer the matrix from off-chip memory onto the chip was on the same order as the computation itself. If the matrix size N is small and can fit all in on-chip memory, the matrix only has to be loaded once. So the data transfer takes $O(N^2)$ time (there are N^2 matrix elements) and the computation requires $O(N^3)$ time. In order to allow an arbitrary matrix size, we use the blocking algorithm described in Section 2.1.3. This approach requires the loading of the matrix in blocks and many of the blocks are loaded multiple times. There are N/N_b passes through the matrix, where N_b is the block matrix size and is an input

constant and a parameter of the engine. The data transfer for each pass through the matrix requires $O(N^2)$ read operations. Thus, the overall data transfer takes $O(N^3)$ time, although it is divided by N_b, and thus it is less but comparable to the computation. Therefore, performance is limited by the computation but data transfer is not something we can ignore and simply perform sequentially in between computations. To get the best overall performance, it is necessary to simultaneously fetch data and compute on it. This requires on-chip "double-buffering" of the memory to allow one on-chip memory block to perform transfers while the other is used in the computation.

The block LU factorization algorithm breaks the entire computation of a single matrix block into blocks. To update any given block, up to a total of three blocks are required as input, as described in Section 2.1.3: the current block being computed, the top-most block in the same column and the left-most block in the same row. We follow the order of updating blocks shown in Figure 2.1(c); thus, we update blocks in the same column first before we move to blocks in the next column. In most cases, the top-most block is the same for consecutive blocks being updated. Thus, we can reuse this block and do not need to load it again. The only case that the next update block has a different top-most block occurs when the updated block is the first block in a new column. However, in this case, the top-most block is also the current block. So by loading in the current block, we have also loaded the top-most block. Thus, we don't have to explicitly load the top-most block. In total, we only ever need to load a maximum of two blocks to perform any block computation, the current block and the left-most block. This reduces the off-chip memory bandwidth required to sustain the computation.

In summary, three blocks are needed to perform the computation while two blocks are simultaneously pre-loaded (to implement the double-buffering described above) for the next computation; thus a total of 5 matrix blocks of on-chip memory is required. The current block and the left-most block are double-buffered to overlap the data fetching with computation as described above. We will discuss, in Section 5.1.4, the effect of the

block size and blocking algorithm have on the performance. We now proceed to describe the computation and data marshalling functions in turn.

3.3 Computation

In this section we will describe the hardware needed to implement the block LU factorization computation in the LUP module shown in Figure 3.1. The Marshalling Controller instructs the LU Controller to perform one of four block computations described in Algorithm 2 in Section 2.1.3. The LU Controller controls the LU Processing to perform the block computation. The following subsections will describe the overall computation that has to be performed to complete the LU factorization algorithm and discuss how each of the four block computations is performed.

3.3.1 Overall Computation

The Marshalling Controller (MC) is responsible for directing the overall block LU factorization algorithm described in Section 2.1.3. First, it loads the necessary blocks to update the first block in Figure 2.1(c) by issuing load commands to the DTU. After loading all the blocks, it then instructs the LU Controller to perform the block computation corresponding to case 1 as shown in Figure 2.1(b). The LU Controller will direct the LU Processing module to perform the computation and returns a done signal when the LU Processing module completes the computation. While computation is being executed, the engine also fetches the necessary blocks for the next block computation. After the computation and block loading tasks are finished, the engine proceeds to compute on the newly loaded blocks. At the same time, the previously updated block is written back to off-chip memory. After that block is written, the next block to be processed is loaded. This process continues for all the blocks until the last block is processed.

It is important to note that the next blocks being loaded cannot be the current block

being computed, otherwise we are violating data dependencies in the algorithm. In the whole algorithm, this does not occur except for the last matrix block. Special handling is needed in this case. The last block to be updated in each pass through the matrix is the last diagonal block in the matrix. For the N/N_b pass, there is only one block that has to be updated. So the previous updated block is from the $N/N_b - 1$ pass through the matrix. The last block in that pass is also the last diagonal block in the matrix. This is a data dependency violation. To avoid this violation, we do not load while the second last block to be updated is being computed. After it has finished updating, the block is written to off-chip memory. Only then are the necessary blocks to update the last block loaded into on-chip memory. After loading is finished, computation is performed on the blocks. Finally, the updated block is written to off-chip memory and the algorithm is completed. Generally, the computation and matrix block transfer are performed simultaneously with some overhead at the start and end of the algorithm. When solving a large matrix, these overheads are small relative to the overall time required.

3.3.2 Block Computation

As described in Section 2.1.3, there are four different block computations that have to be performed. The LU Processing module contains the data path units that perform all four block computations. A diagram of the structure of the LU Processing module is shown in Figure 3.2. As described in the previous section, the computation requires three input blocks – labeled *top block*, *left block*, and *current block* in the figure. Recall that the engine must load two of the blocks for the subsequent computation as part of the double-buffering, and so the left and current blocks have "0" and "1" versions in the figure. The top block is only updated while computing a block in a new column.

Most of the area of the LU Processing module (and also the total design) is made up of the processing elements. The key engine input parameter is **k**, the number of such processing elements (PEs). A processing element consists of one multiplication and

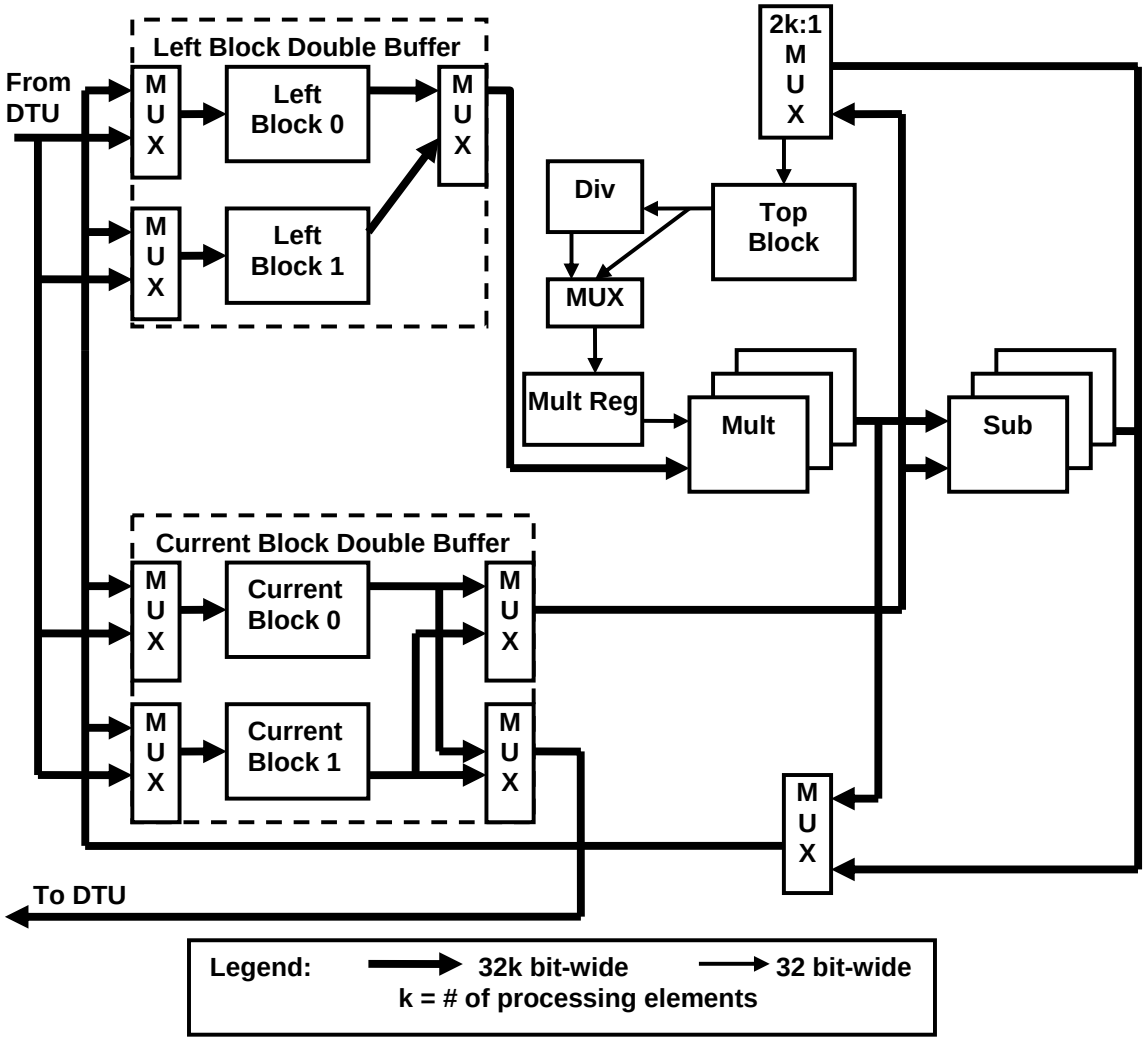

Figure 3.2: Diagram of the LU Processing module which performs the computation.

one subtraction floating-point unit. As specified in Section 2.1.3, most of the operations in block LU factorization consist of multiplication and subtraction, and therefore it is desirable to have as many of the processing elements in our engine as possible to maximize performance.

The LU Processing module uses the processing elements to perform the block computations in Algorithm 2. We will use the description of the algorithm in Section 2.1.3 to describe the data paths in Figure 3.2. For case 4, the multiplication units use data from the left block and top block. The subtraction units use data from the current block and the outputs from the multiplication units. The first column in the left block is multiplied

by the first row in the top block and the result is subtracted from elements in the current block. Then the second column and second row are multiplied and this process repeats until all columns and rows have been used. The output from the subtraction units are written to the current block.

For the other three cases, several extra steps are needed. For case 1 and 3, the top block needs to be updated with the current block. Instead of transferring the whole block before computation, only the first row of the block is transferred to the top block. The first row and the first column in the left block are used to update all the other rows in the current block using the PEs. As the updated values are written back to the current block, the next row of the current block is also stored into the top block and can be used later in the computation. At the end of the block computation, the top block will have the same data as the current block. The top block can be used for other block computations for blocks in the same column.

For case 1 and 2, division operations have to be performed, while case 3 and 4 do not perform any division operations. Rather than creating many parallel dividers that are infrequently used, we compute the reciprocal of the one divisor, which is the diagonal element, and use the multiplier units to compute the division (by multiplying by the reciprocal). This way we only need to create one division unit. By only having one division unit, we free up more area to be used for more processing elements. The reciprocal and the column elements from left block are multiplied and the results are stored into the current block and the left block. As a result, we only perform one division per column, which is a total of N_b divisions when updating blocks for case 1 and 2.

The multiplexers (labeled mux) shown in Figure 3.2 are needed to route the data among these memory block units, and are controlled by the LU Controller. Most multiplexers are 2 to 1 but there is one mux that is 2k-to-1 (labeled 2K:1 MUX). The top block, which has a data width of 32 bits for single precision, stores elements from the current block, which has a data width of 32×k bits, and the output from the k processing

elements, which also have a data width of 32×k bits. The 2k-to-1 mux is used to store elements into the top block.

The LU Controller generates the control signals for the LU Processing module. It controls the on-chip memory blocks to ensure that the correct data is read and written during computation. The LU Controller has a state machine that follows the algorithm of the block computation. To simplify the state machine, some of the control signals are generated and pipelined for later. For example, when performing the floating point operations, the control signals to read the inputs are generated at the same time as the control signals to store the output. The store control signals are passed through shift registers so that they are delayed for the correct timing. Many parameters, like number of processing elements and latency of floating point units, will alter the LU Controller to ensure correct functionality of the algorithm. The correct cycle timing is controlled by inserting registers on these signals. Details about how the LU Controller changes based on the parameters is discussed in Chapter 4.

3.3.3 Floating Point Units in Processing Elements

The floating point units used in the compute engine are generated using Altera's IP cores generator [12]. Altera has cores for single and double precision floating point addition (which can do subtraction), multiplication and division. These cores are portable to different Altera FPGAs; they offer some scalability by allowing users to choose the amount of pipelining (and therefore the clock speed and latency) of the floating point unit. As the latency increases, the operating frequency increases at the cost of more area used. Altera has reduced functionality floating point units that save area and improve operating frequency compared to fully functional floating point units. The reduced functionality units do not perform denormalization and do not provide the exception indicators underflow, overflow and not-a-number (NAN). We chose to use these reduced functionality cores as they are sufficient for our algorithm and can yield a higher performance. The

multiplication and subtraction units are fully pipelined and have a throughput of one per cycle. Since we only have to perform the division rarely, we allow the division unit to take multiple cycles (typically 35 - 50) to prevent it from being the critical path of the engine.

3.3.4 On-chip Memory

The on-chip memory blocks must supply enough data to keep the k processing elements busy. The left block and current block must supply a matrix element to each multiplication and subtraction operator respectively, on every cycle. The left block and current block store data in column major form (consecutive column elements are stored in consecutive memory locations) as the algorithm requires computations on elements in the same column. For single precision operations, the data width of the left block and current block has to be 32 times the number of processing elements. The k value is on the order of 100, so these on-chip memories can be very wide, on the order of 3200 data bits. This is only possible because of the high bandwidth of on-chip FPGA memories. For the top block, one matrix element is sent to all the multipliers or the one division operator, and so the top block has a data width of 32 bits in single precision. The top block stores data in row major order as the algorithm requires elements in the same row.

3.4 Data Marshalling

The data marshalling function (the transfer of blocks to and from off-chip memory) is performed by the Memory Controller (MemC), Data Transfer Unit (DTU) and Marshalling Controller (MC) as illustrated in Figure 3.1. The coefficient matrix A is stored in column major format in the off-chip memory to match how the matrix blocks need to be stored on the on-chip memory. The blocks that need to be loaded to and stored from off-chip are broken into contiguous sets of memory addresses. Each block transfer of contiguous

memory is a single "instruction" implemented by the MC. The MC issues these load and store instructions to the DTU. That instruction contains the off-chip memory address, the on-chip memory address, the size of transfer, a load signal and a store signal.

3.4.1 Memory Controller

The MemC is a DDR2 memory controller generated from the DDR2 SDRAM High Performance Memory Controller in Altera's IP functions [12]. This unit receives read or write commands up to the burst length of the DDR2 and converts it into appropriate DDR2 off-chip interface. More details on the DDR2 SDRAM High Performance Memory Controller are found in [13]. The DTU takes the memory "instruction" described above, a transfer of an arbitrary size of contiguous memory, and breaks it up into suitable sized read or write commands to the MemC. It handles communications between the MC and the MemC. This way the MC can use the same interface and issues the same commands independent of the specific configuration of the MemC (and therefore is part of the design "portability" infrastructure). The MC issues commands based on the computation modules and the problem size. The MemC is configured based on the specific DDR2 SDRAM that is used in the FPGA platform. Parameters are used to alter the memory controller to match different DDR2 SDRAM. The interface protocol remains the same with slight changes to the interface pins. The data width, address width and burst length of the interface to the DTU can change based on the parameters. The DTU is responsible for dealing with the variations in MemC and performing the commands given by the MC. The DTU is also altered based on the parameters for the MemC and MC to ensure that it can communicate properly with both modules. The effects of changing the parameters is discussed in Chapter 4.

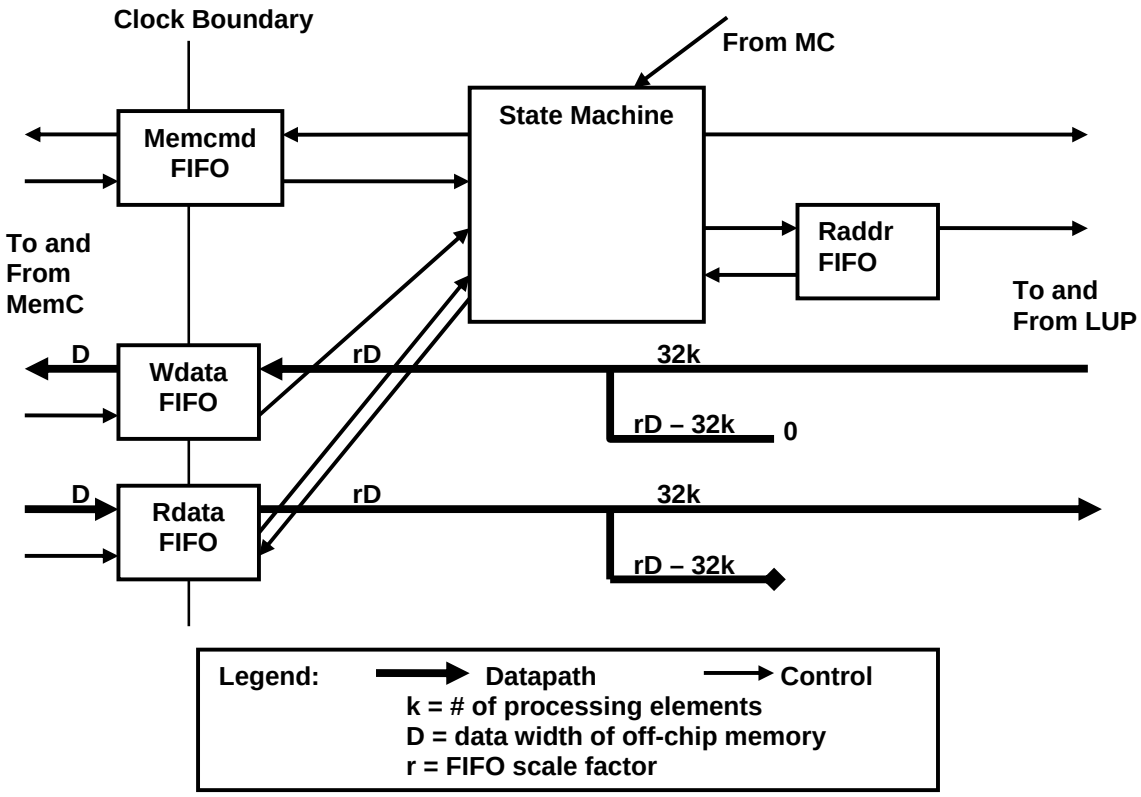

Figure 3.3: Diagram of the Data Transfer Unit which performs the data marshalling.

3.4.2 Data Transfer Unit

The Data Transfer Unit (DTU), shown in more detail in Figure 3.3, along with the MemC are the key modules that enable the portability of our compute engine generator to different FPGA-based boards regardless of the type of off-chip memory. We have designed the DTU to allow the operation speed of the off-chip memory to be independent of the speed and bandwidth (number of bits of width) of the on-chip memory. The speed and bus width of the off-chip memory are two key parameters to the compute engine generator. The decoupling is accomplished through the use of the dual clocked FIFOs illustrated in the Figure 3.3. The left hand side of the FIFOs operates at the off-chip memory clock speed, while the right hand side operates on the compute engine's clock speed.

The state machine in the DTU decodes the memory instructions from the MC and

For store:
1. DTU reads the store data from on-chip memory and writes it into Wdata FIFO
2. DTU writes the write command to the Memcmd FIFO
3. When MemC is ready, it reads the write command from the Memcmd FIFO
4. MemC reads the store data from Wdata FIFO

For load:
1. DTU writes the read command to the Memcmd FIFO and the on-chip memory address to the Raddr FIFO.
2. When the MemC is ready, it reads the command from the Memcmd FIFO
3. MemC writes the load data into Rdata FIFO.
4. When enough data is written into Rdata FIFO, the data is read from the Rdata FIFO and the on-chip address is read from Raddr FIFO.
5. The load data is written to the on-chip memory.

Figure 3.4: Sequence of steps to perform load or store to off-chip memory

breaks it into corresponding memory commands for the MemC, which are sent through dual-clock FIFOs. The dual-clock FIFOs are generated using Altera IP cores [12] and more details can be found in [13]. The sequence of steps taken by the DTU to perform load and store to the off-chip memory through the MemC is specified in Figure 3.4. In Figure 3.3, the Memcmd FIFO stores read, write or nop commands to the MemC. The MemC reads the commands from this FIFO when it is ready to do so. The Wdata FIFO stores the data for write commands and the Rdata FIFO stores the data returned from the off-chip read commands. The Raddr FIFO is a single clocked FIFO that stores the address of the data to be written to the on-chip memory for a load operation. A new command can be issued, before the previous commands are done or even issued, as long as there is room in the FIFOs. FIFOs automatically maintain the order of the commands. If any FIFO is full, the DTU will wait before issuing the next command. If the Memcmd FIFO is empty, nop commands are issued to the MemC.

The Altera FIFOs can have different input and output data width. The FIFO's input or output data width can be scaled by a ratio of r, which is limited to powers of 2. One side of the FIFO has to match the data width of the DDR2 memory controller and DDR2 SDRAM memory itself, which will be D bits; while the other side of the FIFO has to

match the data demand of the processing elements, which is 32×k bits. With a limited ratio value to change the input and output data width in the FIFO, it is unlikely the data widths of the FIFO will match the two sides perfectly. In the case that it can match exactly, which is $rD = 32 \times k$, all data loaded from off-chip memory can be written to on-chip memory and vice versa. However, when it does not match, There will be extra bits on the input side. One option is to have the extra bits contain useful data and we will add resources to use them in the next read or write. However, this solution requires shifting the next data to line up to the end of the extra bits, which is expensive to do on FPGA. We decided to waste the extra bits by padding them with zeroes. Since the on-chip memory resource is more scarce/valuable than off-chip memory, the off-chip memory is padded with zeroes. The exact amount of padding depends on the data width of the on-chip and off-chip memory. We scale up the FIFO so that the data width coming from or going to the off-chip memory is larger than on-chip memory.

Similarly, the size of the matrix will not always match the blocking size that is used in the engine. To simplify the data marshalling task, we pad the end of the column so that it is a multiple of the block size and each column starts some multiple of the block size from the previous column. These extra padded sections of the column are not loaded or stored. The cost of having internal padding is an increase in the total memory needed in the off-chip memory to store the input matrix, which we assume is sufficient to store any input matrix. The user is required to prepare the input matrix by adding the necessary padding.

3.4.3 Off-chip Memory Overhead

To illustrate the effect of padding, we will compute the off-chip memory overhead for an example compute engine. In this example, the compute engine has 120 processing elements (k = 120) with a block size of 120 ($N_b = 120$) and an off-chip memory controller data width of 128 bits (D = 128). Thus, the processing elements need 3,840 bits and the

closest match we can get with the FIFO ratio is 4,096 bits, with a scale ratio of 32 (r = 32). So instead of needing only 3,840 bits to store one packet of a transfer, 4,096 bits are required and so there is a 6.7% increase in off-chip memory storage from the on-chip and off-chip memory data width mismatch.

The waste due to block padding can be computed as follows, by example suppose that the matrix size N is 10,000, which is divided into 84 x 84 blocks. The last block in each block column has a block column of 40. Since we restrict all block columns to be multiplies of the block size N_b, the block in each block column has an actual storage block column of 120. So effectively, instead of storing a matrix of size 10,000 x 10,000, we actually store a matrix of size 10,080 x 10,000. This results in an additional 0.8% increase in off-chip memory and a total off-chip memory overhead of 7.5%. Of course the actual overhead will depend on the compute engine parameters and the problem matrix size.

3.5 Verification

To verify the functionality of the compute engine, we compared the result from a simulation of the synthesizable verilog to a software version of LU factorization. However, one cannot simply check if each value in the result matrix is the same for the two versions. When using floating point operations, there are rounding errors because of the inability to exactly represent decimal values due to the limited number of bits. When computing each mathematical operation, the result has to be rounded to fit the closest representation. If one takes a sequence of interchangeable operations and changes the exact sequence of these operations, different rounding decisions have to be made along the way. Given a long enough sequence of operations, these different rounding decisions can result in a different final floating point result. There is no way to ensure that the FPGA and software will have the exact same order of operations for all the computations and

so these rounding errors will occur. To verify the correctness of the engine, one cannot simply check if the results are the same as the software; a certain amount of error will have to be tolerated.

With many elements in the result LU factorization matrix, each one will have an error value. We need a method to group all these values in to one value for easy comparison. The error for each matrix element does not have the same weight. In general, the large values are more important in solving systems of linear equations. For this situation, the error norm is typically used as a metric. In fact, for some iterative solvers, the error norm is used to check if the error vector is sufficiently small, and so we shall use the same metric [23]. The exact mathematical definition of the error norm, e, for a matrix is shown in Equation (3.1) [22].

$$\begin{gathered} e = [\frac{1}{N^2}(\frac{1}{\Psi_{max}}\sum_{i=1}^{N}\sum_{j=i}^{N}|a(i,j)|^2)]^{\frac{1}{2}}, \\ where\ \Psi_{max} = max(|a(i,j)|)\ for\ 0 \leq i \leq N, 0 \leq j \leq N \end{gathered} \tag{3.1}$$

Now that we have a metric to measure the amount of error, we have to measure it. However, without knowing the exact application, it is hard to determine what amount of error is acceptable. The only difference between the FPGA and software should be the order of the operations. Thus if one uses another software version with a different order of operations, then the two error norms should be similar and we can use this difference as a standard for error. Instead of writing another software version, we used a different optimization level in the gcc compiler (level 2 and 3) to create two different programs from the same code. The different optimization in the gcc compiler results in different operations of the computation as the gcc compiler will try harder in the higher optimization level to reduce the runtime of the program by moving instructions and thus operations around.

Using randomly generated matrices of different dimensions with a range of -1000 to

Table 3.1: Error Norms of Different Test Matrices on FPGA and Software

Test Case	Matrix Size	FPGA Error Norm	Software Error Norm
1	70 x 70	0.000018	0.000011
2	75 x 75	0.000036	0.000027
3	75 x 75	0.000068	0.000085
4	75 x 75	0.000723	0.000978
5	100 x 100	0.000185	0.000351

1000, we computed the results for the two software programs and the FPGA. The FPGA results are obtained through simulation of synthesizable verilog using the ModelSim logic simulator [14]. Using the results from the software with optimization level 2 as the gold standard, we computed the error norms when compared to the other software version (optimization level 3) and the FPGA. Table 3.1 shows the error norms for five test matrices of various sizes. The FPGA results were obtained from simulation of a compute engine with 10 processing elements. The error norms for the FPGA and software are similar; in some cases, the FPGA was better, while in other case, the software was better. The exact value of the error norm changed with respect to the matrix size and the numerical stability of the matrix.

We used the same technique to verify that the software we use to compare with the FPGA was correct by comparing it to the MATLAB version [15], which is a popular commercial tool. The reason we did not directly compare it to the commercial tool is because the commercial algorithm includes pivoting in the LU factorization which is not implemented in the hardware design. Pivoting is the technique to improve numerical stability in direct solvers. The technique involves exchange of rows of the matrix so the diagonal element has the maximum absolute value before it is used to normalize the column. When comparing the software to the MATLAB version, we manually pivoted the matrix beforehand for the comparison. Using this process, our hardware implementation was verified to be correct with an acceptable amount of error.

3.6 Summary

In this chapter, we described the hardware design of our compute engine. We outlined the high level overview of our hardware implementation. The engine has to to perform data marshalling and computation of block LU factorization. We detailed the components involved in performing these two tasks. Finally, we discussed the verification process for our engine. In the next chapter, we will describe how portability and scalability is achieved through the use of parameters and a compute engine generator.

Chapter 4

Portability and Scalability

Recall that the key goal of this research is to illustrate the creation of computational designs on FPGAs that would be easy to use on different FPGA platforms, including those in the future with newer and larger FPGAs as well as platforms with different external memory architectures. This portability and scalability is achieved by having the compute engine adapt to the amount of available resources on the FPGA and the specific off-chip memory architecture. To be portable, the engine must be easy to move to a new FPGA and off-chip memory interface with minimal human effort, and to be scalable, the engine must automatically take advantage of the speed, capacity and memory bandwidth improvements in the new FPGA and memory system. We achieve portability and scalability by (1) defining parameters for the portions of the engine that should change as the FPGA or off-chip memory technology and architecture changes, and (2) creating a generator which can automatically produce an HDL design implementation that matches the desired parameters.

4.1 Compute Engine Generator Parameters

The parameters that are used in the generator can be divided into two categories. The first are the core parameters that must be supplied by the user and they are shown in

Table 4.1: Core Parameters Used in Our Generator

Name	Description	Units
k	Number of processing elements	#
Precision	Of numerical value	# of bits
NMax	Maximum matrix size	#
MatrixBlockSizeDivk	Size of internal matrix block divided by k	#
AdderLatency	Latency of adder unit	# of cycles
MultLatency	Latency of floating point multiplier unit	# of cycles
DivLatency	Latency of floating point divide unit	# of cycles
DDRWidth	Data width of the DDR2 memory interface	# of bits
DDRAddrWidth	Width of the DDR2 address bus	# of wires
DDRRowAddrWidth	Width of the DDR2 row address bus	# of wires
DDRBurstLen	Burst length of the DDR2 memory interface	#

Table 4.1. These parameters control the amount of resources used on the FPGA (which also ultimately control the achieved performance of the resulting engine) and specify the off-chip memory system. The parameters k, Precision, NMax, MatrixBlockSizeDivk, AdderLatency, MultLatency and DivLatency control the amount of resources used on the FPGA. The performance of the engine typically increases as more resources are used, but there are some dependencies between them and counteracting effects that influence performance as well, which we will discuss in Chapter 5. By changing these parameters, the user can create the most suitable engine that matches with the resources available on the FPGA. The parameters DDRWidth, DDRAddrWidth, DDRRowAddrWidth, and DDRBurstLen modify the off-chip memory interface of the engine. These parameters contribute to the portability of the engine by allowing the use of various off-chip memory systems.

The second category of parameters, which are referred to as advanced parameters and described in Table 4.2, are parameters less visible and obvious to the user and they are used to optimize the compute engine. The user does not have to input these parameters, as default values are calculated by the engine to produce a functional engine. For better performance, the user can modify these parameters to suit their FPGA platform. FIFOSize changes the depth of the FIFOs in the DTU (described in Section 3.4.2). This

Table 4.2: Advanced Parameters Used in Our Generator

Name	Description	Units
ExtraOnChipRamBlock-InputPortDelay	Extra registers added to input port of on chip current and left blocks	# of registers/bit
ExtraOnChipRamBlock-OutputPortDelay	Extra registers added to output port of on chip current and left blocks	# of registers/bit
ExtraOnChipTopBlock-InputPortDelay	Extra registers added to input port of on chip top block	# of registers/bit
ExtraOnChipTopBlock-OutputPortDelay	Extra registers added to output port of on chip top block	# of registers/bit
FIFOSize	Size of FIFOs used in Data Transfer Unit	# of elements

influences the data marshalling process, described in Section 3.4, which involves loading the blocks for the next computation and storing the result block from the previous computation. The four other parameters add registers to the inputs and outputs of on-chip memory blocks to increase operating frequency at the cost of increased area consumption (for those registers). More details about how these parameters can improve the performance of the compute engine and their tradeoffs will be discussed in Section 4.1.2.

From these two sets of parameters, the compute engine generator can determine other dependent variables for the engine. For example, the width of the registers needed to store the value of parameters k and the block size, N_b is a direct function of these parameters. The data width of the on-chip memory blocks and the FIFOs in the DTU depend on k and the off-chip memory data width, DDRWidth.

The core and advanced parameters will be described in greater detail in the following subsections. We will focus on how the engine has to be modified based on these parameters.

4.1.1 Core Parameters

Of the core parameters listed in Table 4.1, one that is key to the portability and scalability of the engine is k, the number of processing elements. The generator can be set to produce

an LU Processing module with any number of processing elements. As more FPGA logic capacity is available on a chip (or in moving to a larger chip) the number of processing elements can be increased, thus improving performance. A change in k also affects other modules in the engine. The data width for the on-chip memory blocks has to change to supply data to the processing elements. Similarly, the generator must produce an LU Controller that keeps the processing elements fully occupied. The generator will also have to change the data width of the FIFOs in the DTU, which supply data to the on-chip memory blocks during data marshalling.

The parameters AdderLatency, MultLatency and DivLatency allow the user to specify the latency of the adder, multiplier and divider respectively. Most FPGA vendors offer floating point unit implementations with a range of latencies to allow area/speed tradeoffs. By exposing this latency parameter for all the floating point units, we preserve this tradeoff in our compute engine. In the future, if a superior floating-point unit with a different latency is created, this parameter can be updated accordingly. To accommodate a change in latency, the generator must produce an LU Controller with the desired delay in the control signals to match the specific latency in the floating point units. This is handled by the generator in the following way: when using the floating point units, the control signals to store the outputs are generated at the start of the floating point operations. These control signals are passed through shift registers in the LU Controller so that they can be sent to the LU Processing module when the outputs are ready.

The MatrixBlockSizeDivk parameter determines the size of the matrix block, N_b. In Section 5.1.4, we will explain why N_b must be a multiple of k. When the matrix block size is changed, the generator creates an LU Controller with an altered state machine to perform block computation for that specific block size. Moreover, the generator creates a Marshalling Controller to handle changes in the number of blocks in the matrix and the loading and storing of different sized blocks.

Finally, the precision of the compute engine can be changed so that the engine can

operate on single precision or double precision floating point matrices. When changing the precision parameter, the generator will create an LU Processing module with modified data width in the data path involving the processing elements and will modify the data width of the on-chip memory blocks. The generator will also produce a DTU with a modified data width of the FIFOs. In addition, the user will have to create the desired precision floating point units from the Altera IP core set.

4.1.2 Advanced Parameters

The advanced parameters allow the user to optimize the compute engine by improving operating frequency at the cost of increased area. These include ExtraOnChipRamBlockInputPortDelay, ExtraOnChipRamBlockOutputPortDelay, ExtraOnChipTopBlockInputPortDelay and ExtraOnChipTopBlockInputPortDelay, and they allow the user to add registers to the input and output port of the on-chip memory blocks. For compute engines with many processing elements, the floating point units will likely be placed across the entire FPGA and thus the on-chip memory blocks will have to connect to areas spanning the FPGA. In the development of the engines, we found that many of the critical paths in the compute engine involved the input and output port of the on-chip memory blocks. Thus, by adding pipeline registers in these paths, the critical path delays can be reduced or eliminated. However, simply adding one set of registers might not help reduce much of the delays since all these paths, which could be heading in different directions, will have to connect to these registers. Thus, it is important to duplicate these registers so that they can be spread out across the FPGA and each path can pick the closest registers to connect to minimize delay. We rely on the physical synthesis process in the CAD tools [19] to automatically duplicate these registers and determine which paths should connect to which duplicated registers.

Figure 4.1 shows a block diagram of the LU Processing module with locations where pipeline registers can be added marked with dark circles. Since we have to double buffer

the left and current blocks (as discussed in Section 3.2), the DTU has to write into both of these blocks. To simplify the logic, the left and current blocks should have the same input and output delay so that the DTU does not need to handle different delays in on-chip memory blocks when loading data into these blocks. The top block can have a different number of registers at its input and output port than the left and current blocks since the DTU does not load the top block. However, as more pipeline registers are added, more area is used, which may cause congestion and actually decrease maximum operating frequency. Table 4.3 shows the clock frequency of a 57 PE double precision compute engine with different advanced parameters on a Stratix III 3SL340F1760C3 FPGA. The other parameter values used to generate this engine are specified in Table C.1 in Appendix C.1. As more pipeline registers are added to the engine, the clock frequency improves as critical path delays are reduced. However, the clock frequency decreases for the last set of parameters in the table because of congestion on the FPGA. Thus, some experimentation is needed to determine a good number of registers to use.

The parameter FIFOSize allows the user to specify the size of the FIFOs in the DTU. Since we perform the computation and data marshalling concurrently, the performance will be limited by the computation as long as the time to perform the data marshalling is less than computation time. Thus, we want the off-chip memory to be busy enough so that the data marshalling time is less than computation time. During data marshalling, the DTU issues memory commands to the off-chip memory controller whenever it is ready. The memory controller can process a few memory commands at the same time. While it is processing other commands, it can still receive new commands. There will be a point when the memory controller cannot process any more memory commands concurrently and therefore, it will not accept any new commands. After it has finished processing a memory command, it can receive a new command from the DTU. Thus, most of the time the memory controller will be busy processing commands and occasionally it will be ready to receive new commands.

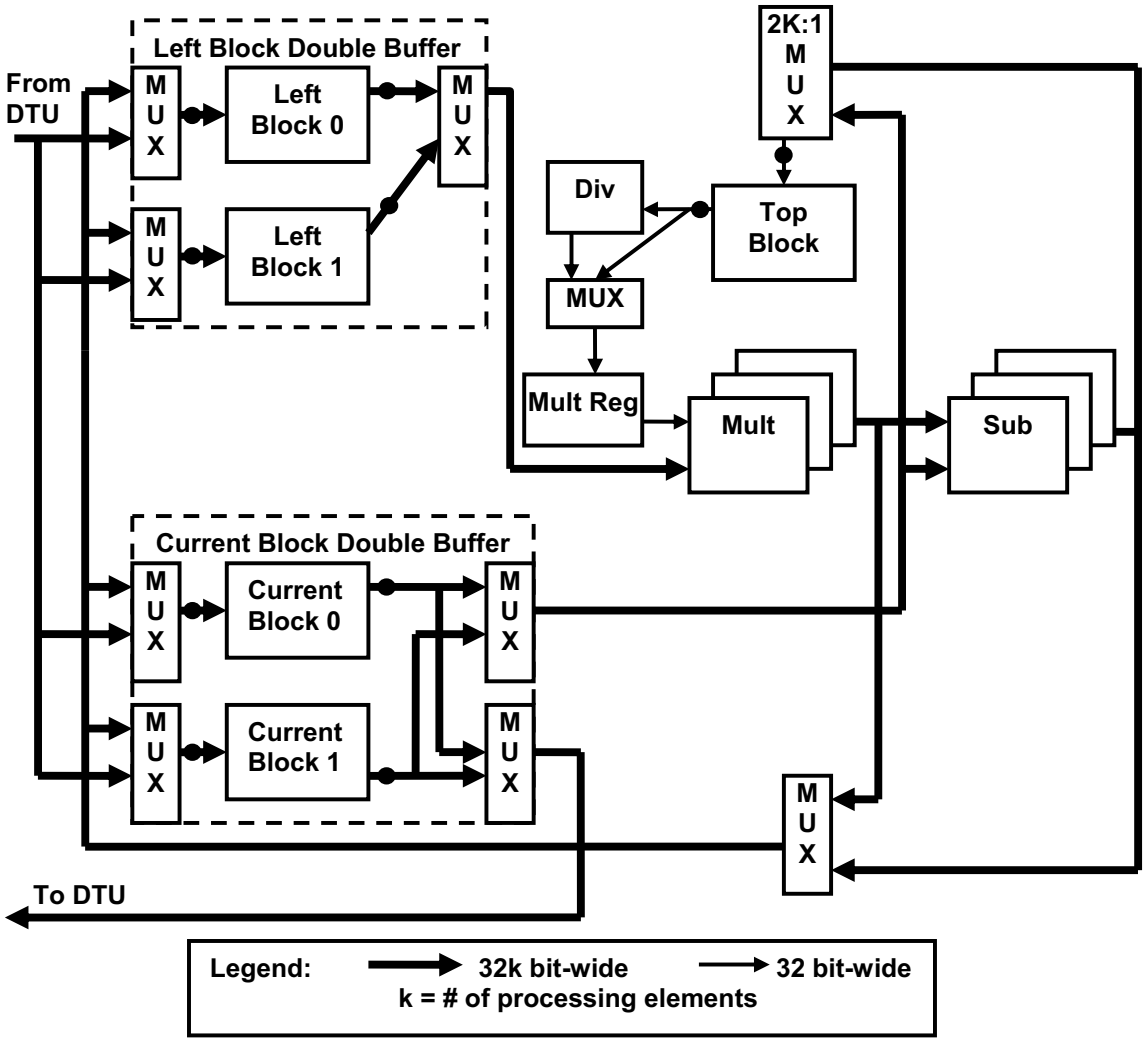

Figure 4.1: Identifies locations of advanced parameter pipeline registers that can be added to LU Processing module

To reduce data marshalling time, the DTU has to ensure that it can issue new memory commands to the memory controller when the memory controller becomes ready. As the memory controller operates on a different clock than the DTU, the DTU can require multiple cycles (with respect to the memory controller clock) to produce a memory command, especially if it is a store command where the data to be stored have to be obtained from on-chip memory. Since there are cycles when the memory controller is not ready for new commands, we can use those cycles to buffer up memory commands for when the memory controller becomes ready. We use dual-clock FIFOs to buffer the

Table 4.3: Clock Frequency for Compute Engine (57 PEs) with Different Advanced Parameters on Stratix III 3SL340 FPGA

ExtraOnChip-RamBlock-InputPortDelay	ExtraOnChip-RamBlock-OutputPortDelay	ExtraOnChip-TopBlock-InputPortDelay	ExtraOnChip-TopBlock-OutputPortDelay	Clock Frequency
2	1	4	2	125 MHz
2	1	6	4	155 MHz
3	1	6	4	170 MHz
3	2	7	2	115 MHz

memory commands; the memory controller can obtain the memory commands from the FIFOs whenever it is ready. For cases when the memory controller is operating faster than the DTU, the dual-clock FIFOs allow multiple memory commands to be issued to the memory controller for every cycle in the DTU. If the FIFO size is small, the FIFOs will be empty at some points during data marshalling and the off-chip memory will sit idle, thus increasing data marshalling time. As the FIFO size increases, the data marshalling time will decrease as there will be less time when the FIFOs are empty. The best FIFO size is one that results in a lower data marshalling time than the computation time. Any further increase in FIFO size will not improve overall performance as it will be computation-limited. In addition, by having a larger FIFO size than needed to achieve a lower data marshalling time than computation time, the extra area of the FPGA used to implement the FIFOs is wasted. That area could have been used to create more processing elements and improve performance. Because the engine can have different delays in accessing on-chip memory and the different clock speeds for the two clocks, the minimum FIFO size that maximizes performance can change. The FIFOSize parameter allows the user to control the size of the FIFOs in the engine to minimize the area used for data marshalling based on their specific FPGA and off-chip memory system.

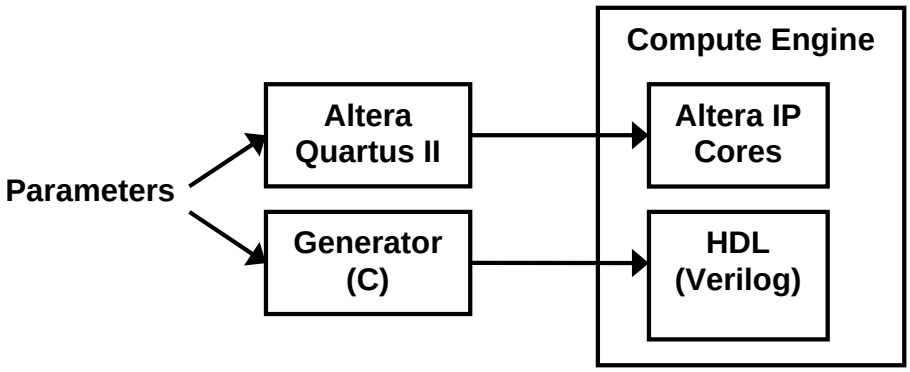

Figure 4.2: The generator flow to create parameterized compute engine

4.2 Compute Engine Generator

We found that Hardware Description Languages (HDL), such as Verilog, were not sufficiently powerful programming languages to fully adapt the compute engine in all the ways described above. Consequently, we implemented the compute engine as software (written in the C language) which generates HDL code in Verilog, based on parameters specific in Section 4.1. The generator source code is shown in Appendix A. Figure 4.2 shows the flow used to produce the compute engine. The compute engine will consist of automatically created HDL code from the generator and Altera IP cores. Currently there is no easy automated method to create these Altera IP cores, therefore they are manually created using the Altera Quartus II design tool based on the parameters. The Altera Quartus II design tool provides a graphic interface to input all the parameters to create the customized cores for the engine. The cores used in the engine consist of the floating point units used in the processing elements, on-chip memory blocks, dual-clock FIFOs used in the DTU and the off-chip memory controller.

The generator takes as input the user-specified parameters and then computes other necessary values. It then passes the necessary parameters and values to other functions in the software that will produce the HDL code. Each of these functions generates the HDL for a module in the engine. These functions mainly consist of C language fprintf

statements that output HDL statements into a file. The generator employs if-else and for-loop statements in C to alter specific HDL statements based on the parameters. This higher functionality in C permits clean adaptation of the engine to the specified parameters. The HDL has the ability to use parameter variables to represent constant values. We use these parameter variables as much as possible to minimize changes in generated HDL. This feature also improves the readability of the HDL code as it highlights the parameters used in specific modules and shows how they affect the HDL code.

Through the following examples, we will illustrate how the generator produces customized engines. First, consider the 2k-to-1 multiplexer used in the LU Processing module to store values into the top block shown in Figure 3.2. The size of this mux changes as a function of the k parameter, with the number of inputs to the mux based on the following equation: $2k \times \mathit{precision}$ (where precision is either 32 bits for single or 64 bits for double). The generator C code, shown in Figure 4.3(a), automatically creates the desired mux size based on the k and precision parameters. Two for-loops are used to create the 2k-to-1 mux. Each for-loop generates a k-to-1 mux and combined together, they form a 2k-to-1 mux. The generated HDL code for k = 4 and precision = 32 (single precision) is shown in Figure 4.3(b).

Another example of customization in the engine is the shift registers used to delay control signals in the LU Controller module. As described in Section 4.1.1, the control signals to store the output of the floating point units are passed through shift registers to create the necessary delay to match the latency of the floating point units. The generator can change the size of the shift registers so that the delay in the control signals corresponds to the latency of the floating point units, which are determined by the AdderLatency, MultLatency, DivLatency parameters. One for-loop is used in the generator C code to create the desired size of the shift registers for one control signal. Each iteration of the for-loop increases the size of the shift registers by one and the number of iterations is a function of the latency parameters.

(a)

```
fprintf(fp, "    case (topWriteSel)\n");
for (i = 0; i < k; i++)
{
   lowerIdx = (k-i-1)*precision;
   upperIdx = (k-i)*precision-1;
   fprintf(fp, "        %i:\n", i);
   fprintf(fp, "            topWriteDataLU = ramReadDataLU[%i:%i];\n", upperIdx, lowerIdx);
}
fprintf(fp, "        default:\n");
fprintf(fp, "            topWriteDataLU = ramReadDataLU[PRECISION-1:0];\n");
fprintf(fp, "    endcase\n");
fprintf(fp, "    else\n");
fprintf(fp, "    case (topWriteSel)\n");
for (i = 0; i < k; i++)
{
   fprintf(fp, "        %i:\n", i);
   fprintf(fp, "            topWriteDataLU = addResult[%i];\n", k-i-1);
}
fprintf(fp, "        default:\n");
fprintf(fp, "            topWriteDataLU = addResult[0];\n");
fprintf(fp, "    endcase\n");
```

(b)

```
if (topSourceSel == 0)
   case (topWriteSel)
      0:
         topWriteDataLU = ramReadDataLU[127:96];
      1:
          topWriteDataLU = ramReadDataLU[95:64];
      2:
          topWriteDataLU = ramReadDataLU[63:32];
      3:
          topWriteDataLU = ramReadDataLU[31:0];
      default:
          topWriteDataLU = ramReadDataLU[PRECISION-1:0];
   endcase
else
   case (topWriteSel)
      0:
          topWriteDataLU = addResult[3];
      1:
          topWriteDataLU = addResult[2];
      2:
          topWriteDataLU = addResult[1];
      3:
          topWriteDataLU = addResult[0];
      default:
          topWriteDataLU = addResult[0];
   endcase
```

Figure 4.3: (a) shows C code from Generator; (b) shows automatically created Verilog code

4.3 Scope of Portability and Scalability

In this section, we evaluate the portability and scalability available in our compute engine, discuss its limitations and offer solutions to these limitations.

To achieve portability and scalability for any FPGA, we used parameters and automated code generation. Our compute engine can scale for any number of processing elements to perform LU factorization. It can also perform the algorithm given various block sizes. These design features allow us to create the engine using different amounts of FPGA resources. The performance of the engine scales up as more resources are used. However, our compute engine uses Altera IP cores that can only be used on Altera FPGA. Therefore, with our generator, we can create an engine customized to fit any Altera FPGA or some portion of the Altera FPGA. To use another vendor's FPGA, we would have to use vendor-independent cores or that vendor's own cores. A simple wrapper can be created to use those cores with our compute engine. For example, Xilinx has its own arithmetic functional unit cores that can be used on their FPGAs. Their cores have similar functionality and structure as the Altera core, just with different names for the inputs and outputs. The wrapper would map the inputs and outputs of the Xilinx cores to match those in the engine.

We achieve portability and scalability of the off-chip memory interface by (1) parameterizing key aspects of the memory interface and (2) ensuring there is a clean divide between our design and the off-chip memory controller including different clock domains. Parameterizing the memory interface in this way allows the user to use DDR2 SDRAM of various data widths and speeds without any redesign.

Moving to a different (non-DDR2) memory technology is very low effort if the vendor-supplied off-chip memory controller has the same interface to the FPGA logic as that of the DDR2 off-chip memory controller we use. Altera also has memory controllers for DDR1 and DDR3 SDRAM and the interface used for these are similar to the interface for DDR2. Thus, it would require little effort use DDR1 and DDR3 SDRAM with our compute engine. If the memory controller has a different interface, some bridging hardware must be designed; this is akin to creating a new device driver in the software world. For example, the DDR2 memory controllers for Xilinx have a slightly different

interface protocol. During store commands to the off-chip memory, the write data is supplied to the memory controller at the same time as the command is issued. Instead, the Altera DDR2 memory controller will request the write data some cycles after the store command. Therefore, to use the Xilinx FPGA platform, a bridging hardware would be needed to handle this memory interface difference.

By using different clock domains for the off-chip memory interface and the main computation modules, we can run each part of our engine at its own maximum frequency. One does not have to slow the computation to match the clock speed of the off-chip memory or vice versa, and since the memory interface and on-FPGA clock speeds will probably increase at different rates, this flexibility is very important to the scalability of our engine. This flexibility allows the user to change either the FPGA or the off-chip memory independently of each other. When upgrading the FPGA, the off-chip memory does not have to be upgraded if data marshalling is not the limiting factor in the performance. To improve performance of data marshalling, the user can either obtain faster off-chip memory or increase the data width of off-chip memory by employing more banks of memory.

One final limitation involves setting up and initiating the engine. In our current design, we require a host processor to be able to fill the off-chip memory with the input matrix data and it must also initiate the computation on the FPGA. In some FPGA computation systems, the off-chip memory for the FPGA has a dedicated connection to the FPGA and the host processor has no access to it. In such a case the host would have to use the FPGA itself to fill the data in external memory. In general, an additional hardware module is needed to handle all possible board configurations for complete portability to any FPGA platform.

4.4 Summary

In this chapter, we have discussed how a large set of parameters are used to achieve portability and scalability in our compute engine. The core parameters are used to enable portability and scalability while the advanced parameters can be used to tune performance. The generator will use the parameters to adapt the engine to the target FPGA platform. Finally, we outlined the scope of the portability and scalability of our engine. The limitations to achieve complete portability and scalability are discussed and solutions to these limitations are suggested.

Chapter 5

Experimental Measurements

In this chapter, we describe the results of experiments that measure the performance of the compute engines for solving linear systems produced by our generator. The generator described in Section 4.2 can create many versions of a linear system solver engine for a specific FPGA. As discussed in Chapter 4, there are a number of tradeoffs that must be experimentally explored to find the best performance for any FPGA. In this chapter, we will start by describing the optimization strategy to determine the best performing engine for a specific FPGA. Then we will compare our best performing engine on a Stratix III FPGA to the best performing software version running on a processor in the same IC process technology. We will also analyze and compare the power consumption for the FPGA and the processor. Next, we will examine the impact of the problem matrix size on performance. Finally to illustrate the portability and scalability of our generator, we compare the performance on a Stratix III FPGA to a Stratix II FPGA. A complete set of data for the experiments described in this chapter can be found in Appendix B.

5.1 Optimizing Design for a Specific FPGA

The generator can create many engines for a specific FPGA platform by changing the parameters described in Chapter 4. Some of the parameters are determined based on

the target FPGA platform to allow portability. The remaining parameters scale the performance achieved and the amount of resources used on the FPGA. The number of processing elements (k), latency of floating point units and block size (N_b) are core parameters that affect the performance of the computation engine. The advanced parameters, which are described in Section 4.1.2, can be used to reduce the critical path delay and improve clock frequency. Using these parameters, the user can increase the resource usage on the FPGA to obtain better performance. This succeeds until the FPGA becomes too congested and performance begins to decrease. In this section, we will explore how these parameters impact performance. We will provide an approach that users can take to achieve the best performance on a specific FPGA. For this exploration, we targeted the largest FPGA available at the time, the Stratix III 3SL340F1760C3 FPGA, and optimize for a single precision compute engine.

5.1.1 Methodology

We will use GFLOPS, the number of billions of floating point operations per second as the metric to measure the performance of a linear equation solver. The maximum GFLOPS achievable for our compute engine can be computed by multiplying the maximum number of floating point operations performed per cycle with the number of cycles per second. In our engine, each processing element can perform two floating point operations (one multiply and one add) per cycle. So the maximum GFLOPS for an engine can be calculated by the following equation: $2 \times k \times clockfrequency$. However, there is overhead required to setup the computation and data dependencies in the LU factorization algorithm that prevents the engine from achieving this maximum performance. Thus, we measure the performance in useful GFLOPS, which counts only operations that are used to solve the linear system.

The useful GFLOPS can be calculated by dividing the total number of floating point operations needed to solve the LU factorization by the total runtime. Instead of using the

total runtime as determined from the actual hardware, we rely on measurements obtained from synthesizing the compute engine using the Altera Quartus II CAD tool and from simulating it using the Modelsim logic simulator. The total runtime is calculated by dividing the number of cycles required to perform the computation with the maximum achievable clock frequency of the design, as determined by post-placement and routing timing analysis performed by Altera's Quartus II version 7.2 on the synthesizable design. The number of cycles is obtained from a software model of the engine. The software model, written in C language calculates the number of cycles of the block LU factorization algorithm without actually computing any results. This software cycle count model was verified against measurements obtained by running the actual engine in the Modelsim logic simulator. This method is used because it is significantly faster to obtain the cycle count through the software model than through the simulator, especially for large matrices.

5.1.2 Number of Processing Elements

The key parameter that scales the performance and resource usage for the engine is the number of processing elements, k. There are two competing trends that impact performance as the number of processing elements is changed. On the one hand, as k increases, more computation can be performed per cycle and so the number of cycles required to compute the algorithm decreases. On the other hand, as more area is used, the clock frequency of the engine can decrease because of congestion on the FPGA. To investigate the impact of parameter k on performance, we created engines with the same core parameters except for k. Then we further optimized each engine by adjusting the advanced parameters. For each engine, we measured the critical path delay and modified the advanced parameters to add pipeline registers in these critical paths to reduce delays. We iterated this process until no further improvement in performance was achieved. As more resources on the FPGA are used by the processing elements, the

Table 5.1: Performance of Several Computing Engines on Stratix III 3SL340 FPGA

Number of Processing Elements	Maximum Clock Frequency	GFLOPS	Performance Ratio vs 30 PEs
30	240 MHz	14	1.00
60	220 MHz	25	1.8
90	215 MHz	38	2.7
120	185 MHz	43	3.1
128	180 MHz	45	3.2
136	100 MHz	26	1.9
144	95 MHz	26	1.9

amount of area we can use to improve critical paths is reduced. We note that because the highly pipelined nature of the design, increasing the latency in this way does not impact wall-clock performance.

Table 5.1 gives the performance in GFLOPS of several generated compute engines. The maximum number of single precision processing elements that can fit on the Stratix III 3SL340 FPGA is 144, and is limited by the number of hard 36x36 multipliers on the chip. The first column in the table specifies the number of processing elements (PEs) in the engine. The second column provides the maximum post-place and route clock frequency and the third column states the achievable useful GFLOPS. The fourth column in the table shows the performance relative to the compute engine with 30 processing elements (k = 30). The parameter values used to generate these compute engines are specified in Table C.2 and C.3 in Appendix C.2.1. Figure 5.1 is a plot of the second and third column versus the first column, the effect of varying the number of processing elements on clock frequency and performance. As can be observed, the performance increases with the number of processing elements up to approximately 128 PEs. Up to this point, the decrease in number of cycles to compute the algorithm due to more processing elements overshadows the slight decrease in clock frequency, resulting in improved performance. However, the performance decreases for higher numbers of PEs because of the large decrease in clock frequency.

The large decrease in clock frequency for when the number of PEs is higher than 128

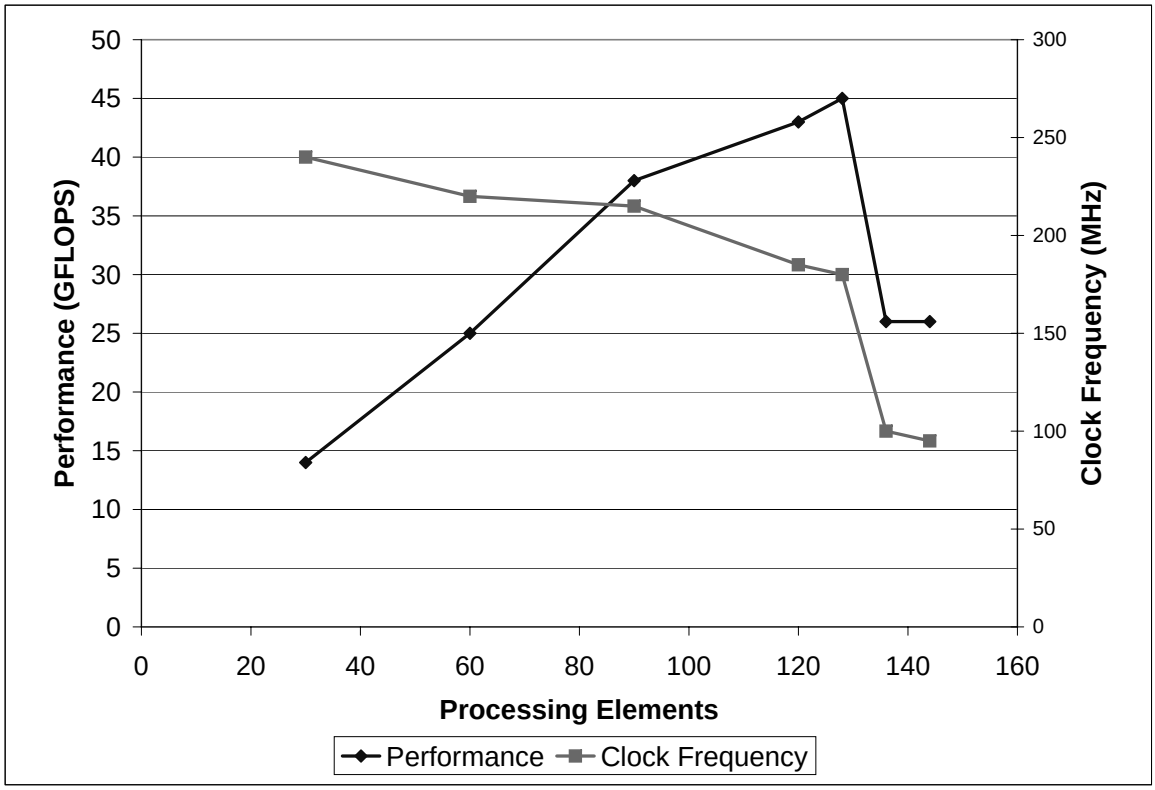

Figure 5.1: The effect of varying the number of processing elements on performance and clock frequency on Stratix III 3SL340

can be caused by several possible reasons: One of the main reasons is congestion on the FPGA, which makes it hard to place connected components closer together and tougher to route the paths in the shortest distance. Another reason is the paths in the engine become longer as k increases. The control signals and data in on-chip memory have to be sent to all PEs. With increasing PEs, the PEs will be more spread out on the FPGA and thus these paths will become longer. Also as k increases, the size of 2k-to-1 multiplexer in the LU Processing module shown in Figure 3.2 becomes larger. A larger multiplexer requires more levels of logic to implement and thus the paths through this multiplexer are longer. These longer paths often become the critical paths in the engine and cause the operating frequency of the whole engine to decrease.

It is possible to use the advanced parameters described in Section 4.1.2 to insert registers to these long paths to reduce their delay. However, the numbers of these pipeline

registers that can be added is limited by the remaining resources available on the FPGA. For a highly utilized FPGA, fewer additional registers are available. Also by using more area for these pipeline registers, we are further increasing congestion on the FPGA which could adversely affect performance. Using this method in order to increase the clock frequency, it may be necessary to reduce the number of processing elements in the engine to make more registers available. Thus, there are tradeoffs between using the area to decrease cycle count by increasing processing elements or to increase clock frequency by adding pipeline registers to reduce critical paths.

In order to maximize performance, it is best not to use all of the resources on the FPGA. Typically, we have found that engines that used over 90% of the resources on the FPGA would suffer these significant decreases in clock frequency.

5.1.3 Floating Point Unit Latency

The latency of floating point units can affect the performance of the engine: by changing the latency, the user can make area and speed tradeoffs. As latency increases, the operating frequency and area of the floating point units increase. As described in Section 3.3.3, the adder and multiplier floating point units can become critical paths in the engine. For engines with low k value, there would typically be sufficient area to add pipeline registers to reduce delay in other paths by adjusting the related pipeline parameters. This may then cause the adder and multiplier floating point units to become the critical paths and limit the performance of the engine. For these engines, the largest latency of the floating point units should be used in order to maximize performance. However, when most of the area on the FPGA is used, it is not possible to add many more pipeline registers to the other paths and so the critical paths in the engine are not in the floating point units. Since the floating point units are not on the critical path, we can use their latency to trade the area of the floating point units for other pipeline registers. Here the latency of the floating point units can be decreased, as long as they do not become the critical

paths, without decreasing overall performance. Because there are many floating point units, any small area decrease for each unit can reduce the overall area of the engine by a significant amount.

The Altera IP adder core can be set to have a latency ranging from 7 to 14 cycles, and the multiplier can have a latency of 5, 6, 10 and 11. The multiplier has a big gap in its latency range. The 5 and 6 latency multipliers are too slow to be useful, which only leaves the 10 and 11 latency multipliers. This limited latency range in the multiplier does not allow for many tradeoffs. The adder has a larger range of latency values and can be use to trade area for speed. Figure 5.2 shows the maximum clock frequencies achieved for a set of engines with 120 PEs and various adder latencies. For each engine, the advanced parameters (listed in Table 4.2) were optimized, similar to the process described in Section 5.1.2, to obtain the highest clock frequency. The parameter values used to generate these compute engines are specified in Table C.4 and C.5 in Appendix C.2.2. Given the same number of processing elements, the engine with the highest clock frequency has the best performance. As shown in the figure, the best performing engine has adder latency of 12. Comparing this engine to one with adder latency 14, this engine used less area for the floating point units which was instead used for pipeline registers to improve other critical path delays. For engines with latencies less than 12, the adder in those engines becomes the critical path in the engine and thus clock frequency is decreased.

5.1.4 Block Size

Recall that the engine computes on blocks of the whole matrix, based on the algorithm described in Section 2.1.3. All the blocks have the same size and are square. The generator has a parameter to change the size of these blocks, which will affect the performance of the engine in the following way: depending on the block size, N_b, the number of cycles to perform the algorithm changes, which directly impacts performance. The maximum

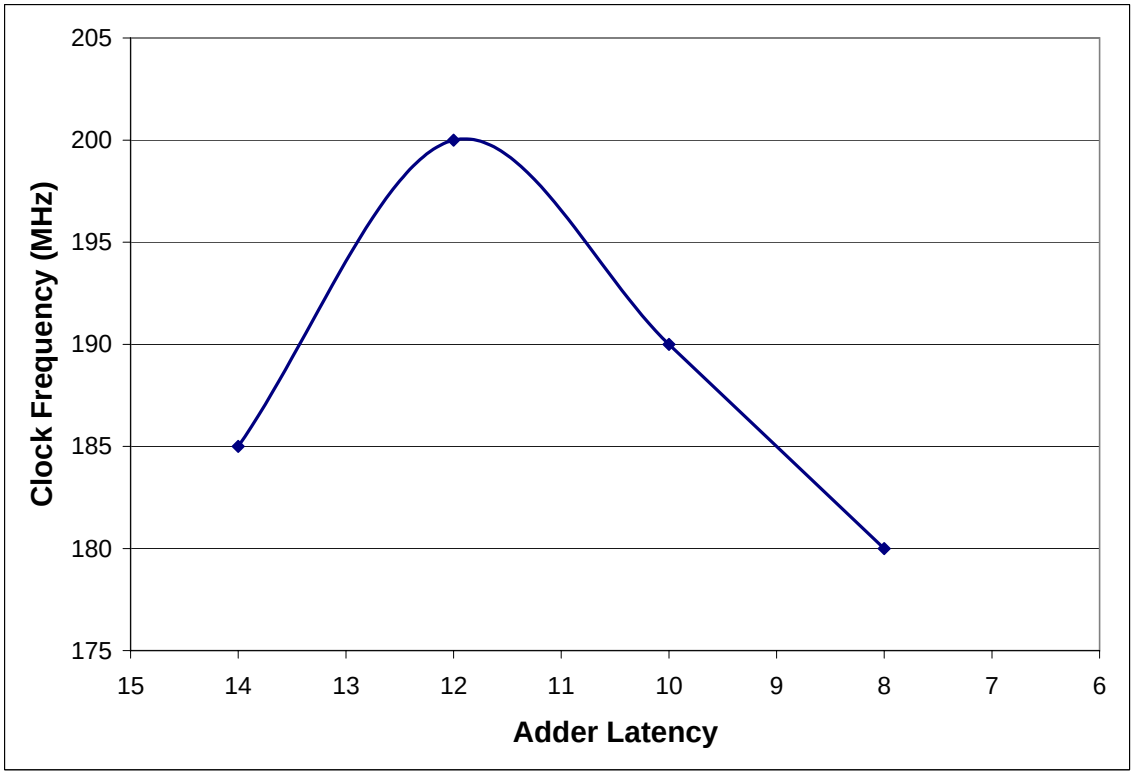

Figure 5.2: Clock Frequency for Compute Engines with Different Adder Latency on Stratix III 3SL340

block size is limited by the amount of on-chip memory on the FPGA. The minimum block size for an engine is $N_b = k$. With a blocking size less than k, some processing elements will never be used as there is no data for them to compute on.

Using the software model of the engine, we calculated the cycle count to complete LU factorization for several different engines. We compared compute engines with various block size for four different k values (40 PEs, 80 PEs, 120 PEs, and 160 PEs). The other parameter values used to generate these compute engines are specified in Table C.6 in Appendix C.2.3. Figure 5.3 shows the percentage change in cycle count as the blocking size, N_b is changed, with the "0%" base set at the cycle count for the minimum blocking size for each design, which is $N_b = k$. The cycle count is inversely proportional to the performance of the engine. As is evident in the figure, simply increasing the block size does not always improve performance. The performance for different block sizes depends

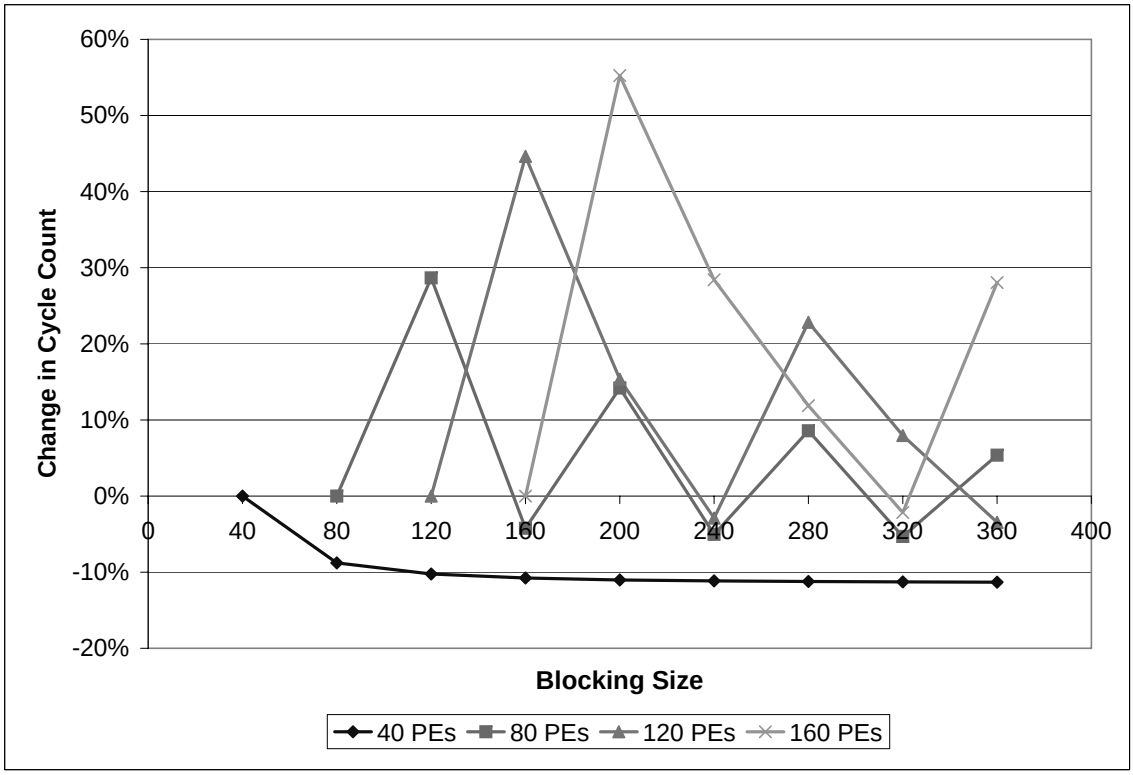

Figure 5.3: Change in cycle count for various blocking size normalized to the minimum block size.

on the particular engine, more specifically the parameter k. A block size that is aligned to k (a multiple of k) performs well and there can be significant increases in cycle count for unaligned block size.

The benefit of having a larger block size (N_b) is that more computation can be performed per matrix block and less data has to be loaded onto the FPGA. As explained in Section 2.1.3, there are N/N_b passes through the matrix and so if N_b increases, fewer passes are needed and thus less data has to be loaded onto the FPGA to perform the same number of computations. This reduces the memory bandwidth required for the computation. However, since our engine is not memory bandwidth limited, this reduction in memory bandwidth does not affect performance. The main benefit of having larger N_b comes from lower computation overhead. With more data available in the block, the PEs are kept busy for a longer period of time. Also, for every case 2 matrix block in

Table 5.2: Change in Cycle Count for Various Compute Engines

	Change in Cycle Count			
Block Size	40 PEs	80 PEs	120 PEs	160 PEs
1 × k	0.0%	0.0%	0.0%	0.0%
2 × k	-8.8%	-4.2%	-2.9%	-2.2%
3 × k	-10.2%	-5.0%	-3.4%	-2.6%
4 × k	-10.8%	-5.3%	-3.6%	-2.8%

Algorithm 2 (see Section 2.1.3, we have to recompute a reciprocal in order to do the normalization. By increasing N_b, there will be fewer blocks, including case 2 blocks, and so fewer division operations are required. However, if N_b is not aligned to k, then at the end of each column some of the PEs will not have any data to compute. For example, suppose $N_b = 1.5k$. Then for every two cycles while the pipeline in the PEs is full, one cycle will use all the PEs and the other cycle will only use 50% of the PEs. Thus, we have an overall efficiency of 75% when the pipeline in the PEs is full. Compared to when $N_b = k$, when the pipeline is full, we use all the PEs every cycle; an overall efficiency of 100%. When N_b is not a multiple of k, the idle cycles for some of the PEs outweighs the benefit gained by having a larger N_b. Thus, in our compute engine, we limit N_b to be multiples of k.

Table 5.2 shows the percentage change in cycle count, normalized to N_b equal to k, for different engines with various N_b values that are multiples of k. The other parameter values used to generate these compute engines are specified in Table C.6 in Appendix C.2.3. As the block size increases, the cycle count decreases but the incremental difference gets smaller. The majority of cycle count improvement can be achieved with a block size of 2 × k or 3 × k. For compute engines with large k values, the minimum block size of N_b equal to k does not degrade performance significantly as the performance is within 5% of that of larger N_b. Thus, the amount of on-chip memory on an FPGA is not an important factor limiting performance. To maximize performance, the largest block size that is a multiple of k should still be used.

In summary, the parameters described in this section give the user flexibility to op-

timize the engine to get the most performance out of their FPGA. When optimizing an engine, the k parameter should be the first parameter to be varied since it has the greatest effect on performance. While varying k, one should use the largest latency value for the floating point units and the largest block size possible that is a multiple of k. Then using the engines with the best performing k values, the latency and advanced parameters are modified to get further improvement in performance. Some experimentation will be necessary to determine the parameter values that will yield the best performance for a particular FPGA. One may think that because more work and knowledge is required of the user, the portability of the engine is reduced. However, the parameter values that will achieve about 20% of the maximum performance for an engine can be determined fairly quickly. The latency parameters can be set to the highest value, with which one can typically get within 10% of the maximum performance as observed in Figure 5.2. One feature to work on for the future is for the generator to automatically estimate good values for the parameters based on the target FPGA platforms.

5.2 Performance

In this section, we compare the performance of the engine against a highly optimized software version running on a processor in the same technology process. This comparison is done for both single and double precision. We targeted the largest FPGA available at the time, which was the Stratix III 3SL340F1760C3 FPGA. We assume that the FPGA is attached to off-chip DDR2 SDRAM of size 256MB and 64bit wide. This is compared to two software versions: one from the Intel MKL library [8] and a more basic code written by the author. The processor used is an Intel Xeon 5160 dual core 3.0 GHz processor with 4MB of L2 cache and 8GB of RAM. The Intel MKL library is highly optimized multi-threaded code specifically created for the Intel processor. The more basic code is single-threaded and it is modeled after the pseudo-code in Section 2.1.2.

Table 5.3: Single Precision Performance on 65 nm FPGA and Processor Platforms

Platform	Clock Frequency	GFLOPS	Performance Ratio
FPGA: Stratix III 3SL340F1760C3	200 MHz	47	2.2
CPU: MKL on Xeon 5160 single core	3 GHz	21	1
CPU: MKL on Xeon 5160 dual core	3 GHz	42	2
CPU: basic code on Xeon 5160 single core	3 GHz	1.1	0.052

We include a performance comparison to this more basic code to show how impressive the vendor-optimized software is.

5.2.1 Single Precision Performance

The top-performing single precision compute engine on the Stratix III 3SL340 FPGA employs 120 processing elements and achieves a maximum operating frequency of 200MHz. The parameter values used to generate this compute engine are specified in Table C.7 in Appendix C.3.1. This engine does not have the maximum number of processing elements possible on the FPGA and has adder latency less than the maximum.

Table 5.3 gives the performance in GFLOPS of several platforms, with all chips fabricated in the same 65nm IC process. The first column lists the platforms, including the top-performing Stratix III 3SL340 FPGA implementation described above, a single core Intel Xeon processor, the dual core (both running the optimized MKL version) and the single core running the more basic code. The table also gives the operating frequency of the hardware (either of our design or the processor clock speed), the performance in GFLOPS, and the performance normalized to that of the single core optimized MKL code.

Our FPGA implementation achieves a performance of 47 GFLOPS, which is 2.2 times greater than the single core Xeon running the MKL code. The FPGA is essentially tied with the dual core processor, which is perhaps the most fair comparison as the second core resides on the same chip. This is a surprising result, as we expected to achieve far

more significant speed gains. The Intel optimized code for the Xeon processor makes use of the SSE2 instruction set, which employs 4-way data parallelism on 32 bit single precision quantities for multiplication and addition. We also believe that the optimized implementation uses a careful blocking scheme (similar to the one described in Section 2.1.3) to make the best use of the caches on the processor.

The quality of this optimization is illustrated by the performance of the basic code as shown in Table 5.3. The basic code is a factor of 19 times slower than the optimized MKL code running on the Xeon single core processor. We believe a key lesson of this research is that for FPGA-based computation accelerators, it is crucial to compare to the best performing software on large-scale problem instances, as we have done here. We understand that significant effort is given by Intel to produce this optimized library, perhaps related to the effort to create the FPGA-based engine.

5.3 Double Precision Performance

For the Stratix III 3SL340F1760C3 FPGA, we also determined the best performing double precision compute engine. The best performing engine has 57 processing elements and achieves a maximum operating frequency of 170MHz. The parameter values used to generate this compute engine are specified in Table C.8 in Appendix C.3.1. This engine utilizes the maximum number of processing elements available on the FPGA. This is because the number of processing elements is limited by the number of hard multipliers on the FPGA. The double precision multiplier uses 2.5 times more hard multipliers than the single precision multiplier, and it uses twice as many registers. Thus, the FPGA still has resources available to add pipeline registers and reduce critical path delay. Table 5.4 gives the performance in GFLOPS of several platforms in double precision, similar to Table 5.3.

For double precision, our FPGA implementation achieves a performance of 19 GFLOPS,

Table 5.4: Double Precision Performance on 65 nm FPGA and Processor Platforms

Platform	Clock Frequency	GFLOPS	Performance Ratio
FPGA: Stratix III 3SL340F1760C3	170 MHz	19	1.7
CPU: MKL on Xeon 5160 single core	3 GHz	11	1
CPU: MKL on Xeon 5160 dual core	3 GHz	21	1.9
CPU: basic code on Xeon 5160 single core	3 GHz	0.55	0.05

which is a 1.7 times greater than the single core Xeon running the optimized code. The FPGA engine is actually slower in performance than the dual core processor. The performance ratio for double precision FPGA implementation is lower than the performance ratio for single precision FPGA implementation, because of the higher cost for the FPGA to perform double precision floating point operations than the processor. The SSE2 instruction set in the processor enables 2-way data parallelism on 64-bit double precision quantities for multiplication and addition. So it can perform half the number of operations per cycle in double precision than single precision. Hence the GFLOPS for double precision is approximately half that of the single precision. For the FPGA, the double precision multiplication floating point unit uses 2.5 times the hard multipliers than the single precision unit. So less than half the number of processor elements can fit on the FPGA for double precision than single precision. Also the operating frequency is lower for double precision than single precision on the FPGA while the operating frequency remains the same for the processor in both cases.

5.4 Power Consumption

While it is true that in supercomputing, performance is the key metric, in recent years the power consumed for computation has become a significant issue, not just in the portable world but in the cost of electricity required to support supercomputers. Table 5.5 shows the power consumption for single precision LU factorization on each platform listed in Table 5.3 normalized to the single core processor running the MKL code. The second

Table 5.5: Single Precision Power Consumption and Energy Efficiency Comparison

Platform	Power (W)	Power Ratio	GFLOPS	GFLOPS per Watt	Efficiency Ratio
FPGA: Stratix III 3SL340F1760C3	18	0.45	47	2.61	5
CPU: MKL on Xeon 5160 single core	40	1	21	0.525	1
CPU: MKL on Xeon 5160 dual core	80	2	42	0.525	1
CPU: basic code on Xeon 5160 single core	40	1	1.1	0.0275	0.052

column lists the power consumption of each platform and the third column specifies the power ratio, which is the power of the platform divided by the power of the single core processor. The fourth column contains the energy efficiency in GFLOPS per Watt and the fifth column states the energy efficiency ratio normalized to the single core processor running MKL. The power consumption of the FPGA engine was measured using vectorless estimation in Altera's PowerPlay Power Analyzer. The power consumption of the Xeon dual core processor was determined from the specification on the Intel website [16]. The Xeon dual core processor requires 80W of power and we assume that a single core requires half the power. As shown in the table, the single precision FPGA implementation requires 2.2 times less power than the single core Xeon processor. Furthermore, the performance in GFLOPS per Watt, which is essentially the amount of energy used per computation, is 5 times better for the FPGA implementation than the processor. As the performance of the dual core is twice as fast as the single core but uses twice the power, the energy efficiency of the Xeon single and dual core processor is the same.

Table 5.6 shows the power consumption for double precision, similar to Table 5.5. The double precision FPGA implementation uses 2 times less power than the single core Xeon processor. In terms of GFLOPS per Watt, it is 3.5 times better than the single core processor. Similar to the performance ratio, the energy efficiency ratio for double precision FPGA implementation is lower than the energy efficiency ratio for single precision FPGA implementation. The energy efficiency of the single and dual core processor is about the same for double precision.

Table 5.6: Double Precision Power Consumption and Energy Efficiency Comparison

Platform	Power (W)	Power Ratio	GFLOPS	GFLOPS per Watt	Efficiency Ratio
FPGA: Stratix III 3SL340F1760C3	20	0.5	19	0.95	3.5
CPU: MKL on Xeon 5160 single core	40	1	11	0.275	1
CPU: MKL on Xeon 5160 dual core	80	2	21	0.263	0.96
CPU: basic code on Xeon 5160 single core	40	1	0.55	0.0138	0.052

5.5 Matrix Size

We observed that many previous works that implemented linear equation solver on FPGA typically only use on-chip FPGA memory to store the matrix, which severely limits the size of the problems addressed, and therefore the overall applicability. Our implementation employs off-chip large-scale memory and therefore is much more widely applicable. Figure 5.4 measures the performance (in GFLOPS) for the various platforms as a function of matrix dimensions, N. Here you can see that the performance for all platforms eventually levels out and reaches a maximum as matrix size increases. The performance comparison used the larger matrix size that achieves these leveled off performance values. Many previous works [21] [26] [25] [30] have a limited and small block size and so most of their comparison will be for a small matrix. When comparing for small matrices, the performance of the software and FPGA implementation have not reached their maximum performance. As shown in the figure, the FPGA implementation ramps up faster and is able to reach its maximum achievable performance faster than software. Thus, there is a larger speed up for FPGA over software when solving small matrices.

We will illustrate this larger speed up by comparing the performance for solving a smaller matrix size. Since a Stratix III FPGA can store a maximum single precision matrix of 721x721 in on-chip memory, we will compare performance of solving a 600x600 matrix on FPGA and processor. Table 5.7 shows the performance and energy efficiency in single precision for each platform listed in Table 5.3. The second and fourth column shows the performance in GFLOPS and energy efficiency in GFLOPS per Watt. The third and fifth column show the performance and energy efficiency ratio normalized to

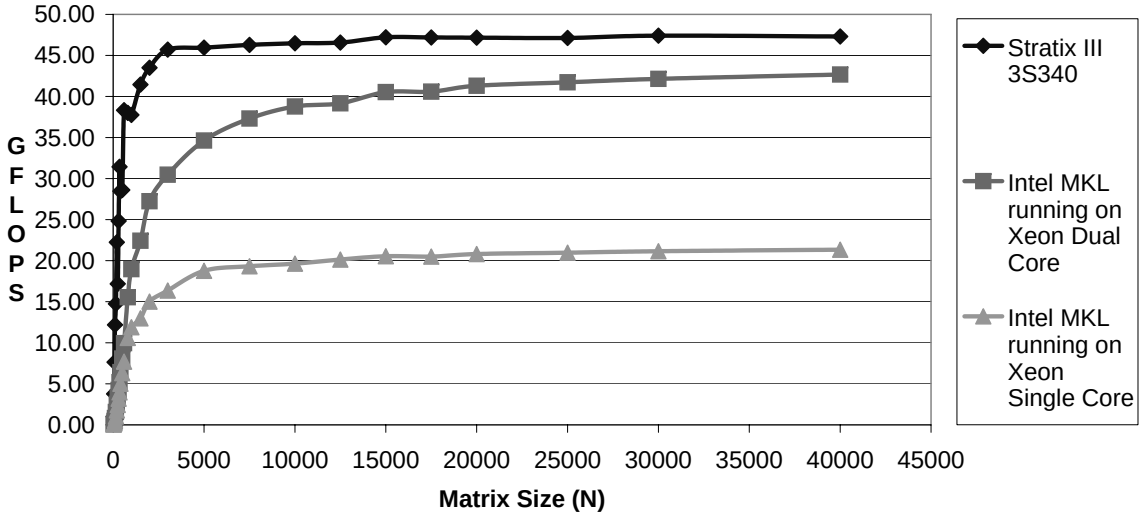

Figure 5.4: Performance as a Function of Matrix Dimension

Table 5.7: Performance on FPGA and Processor Platforms for Solving a 600x600 Single Precision Matrix

Platform	GFLOPS	Performance Ratio	GFLOPS per Watt	Efficiency Ratio
Stratix III 3SL340F1760C3	38	4.9	2.1	11
MKL on Xeon 5160 single core	7.7	1	0.19	1
MKL on Xeon 5160 dual core	10	1.3	0.125	0.65

the single core processor. Since the FPGA can reach its maximum performance faster than software, the FPGA has a higher performance and efficiency ratio than for solving larger matrices. The FPGA achieves a performance of 38 GFLOPS while the software running on single core and dual core processor has GFLOPS of 7.7 and 10 respectively. Comparing the FPGA to the single core, the FPGA has 5 times higher performance and is 11 times more energy efficient. The performance of the software on the dual core processor ramps up more slowly than on single core processor and therefore, the dual core processor actually has lower energy efficiency than single core processor. The FPGA has about 4 times higher performance and is 17 times more energy efficient than the dual core. In summary, when comparing for small matrices, the FPGA can achieve a much higher performance than software because it can reach its maximum performance faster.

Although there are a few applications that require acceleration of small matrices, the

problem of accelerating small matrices is not terribly useful as the overall computation time is already small. We are simply pointing out that not using leveled-off performance values is a common pitfall of FPGA acceleration research.

5.6 Portability to Different FPGAs

To demonstrate the portability and scalability of our generator and engine, we also targeted the Altera Stratix II 2S180F1508C3 FPGA and assumed it is attached to off-chip DDR2 SDRAM of size 256MB and 64bit wide. We determined the top-performing single and double precision engine in a similar fashion as described in Section 5.1. The best single precision compute engine we achieved has 64 processing elements and operates at 170MHz. The parameter values used to generate this compute engine are specified in Table C.9 in Appendix C.3.2. For double precision, our best performing engine has 29 processing elements with a block size of 58 and operates at 140MHz. The parameter values used to generate this compute engine are specified in Table C.10 in Appendix C.3.2. Table 5.8 compares the performance on Stratix II 2S180 FPGA to the performance on Stratix III 3SL340 FPGA as shown in Table 5.3 and 5.4 for both precision. The first and second column shows the target platform and precision respectively. The third and fourth column lists the number of processing elements and the clock frequency in the top-performing engine respectively. The fifth column shows the performance in GFLOPS and the sixth column shows the performance ratio normalized to the Stratix II 2S180 FPGA of the same precision. For single precision, the Stratix III 3SL340 FPGA has a 2.2 times performance improvement over Stratix II 2S180, while there was a 2.6 times performance improvement for double precision. Thus, the performance for both versions scales up by about the same amount. From a Stratix II FPGA to a Stratix III FPGA, the clock frequency increased by 18% for single precision and increased by 21% for double precision. The majority of the performance improvement comes from increasing the

Table 5.8: Performance on Stratix II 2S180 and Stratix III 3SL340

Platform	Precision	PEs	Clock Frequency	GFLOPS	Performance Ratio
Stratix II 2S180F1508C3	Single	64	170 MHz	21	1
Stratix III 3SL340F1760C3	Single	120	200 MHz	47	2.2
Stratix II 2S180F1508C3	Double	29	140 MHz	7.4	1
Stratix III 3SL340F1760C3	Double	57	170 MHz	19	2.6

number of processing elements, which increased by 87.5% for single precision and 93% for double precision. This scalability feature in our generator is key in achieving significant performance improvement as one moves from one FPGA to another.

5.7 Summary

In this chapter, we analyzed tradeoffs in the parameters of the generator that affect performance and provided a strategy to optimize the performance of the engine on an FPGA. We described the results of experiments that measure the performance and power consumption of the compute engines and compared it to highly optimized software running on a processor of the same IC process technology. We also examined the impact of the problem matrix size on performance. Finally, we illustrated the portability of our generator, measuring the performance and power consumption on another FPGA.

Chapter 6

Conclusion

In this thesis, we have created a portable and scalable computational engine generator for the LU factorization method for solving systems of linear equations. Using the generator, we obtained a performance of 47 GFLOPS on a Stratix III 3S340 FPGA and a performance of 21 GFLOPS on a Stratix II 2S180 FPGA in single precision. In double precision, we reached performances of 19 and 7.4 for Stratix III 3S340 FPGA and Stratix II 2S180 FPGA respectively. For both precisions, the performance scaled by more than twice when moving from Stratix II to III, largely due to the almost doubling of functional units in the engine. We have also shown that this engine has significant performance and performance per watt advantages over a single core processor, but the performance advantage was not nearly as large as expected when we compared to the vendor-optimized software library for the same computation. Our scalable single precision FPGA engine is 2.2 times faster than a single core processor (built in the same IC fabrication process) and 5 times more power efficient. For double precision, the FPGA engine is 1.7 times faster than a single core processor and 3.5 times more power efficient.

6.1 Contribution

This work has demonstrated one example of a portable and scalable computation engine for FPGAs, many of which would be needed to make it easy to employ FPGAs in supercomputing applications. Through the use of parameters, we can adapt the engine for various FPGAs and their off-chip memories. The generator can create either single or double precision LU factorization engines that can solve any arbitrary matrix size. This engine establishes a framework to create other portable and scalable engines. The data marshalling modules can be reused in other engines; only the computational modules have to be changed. With the scalability of the compute engine, we can use any amount of FPGA resources. This scalability allows for various different sized engines that can be used for benchmarking purposes.

6.2 Future Work

Even though our compute engine is portable and scalable to different FPGA platforms, there are a few restrictions that prevent complete portability. Currently our engine uses Altera IP cores and so we are limited to using Altera FPGAs. To use other vendor supplied cores, simple wrappers must be implemented. A second limiting assumption is that the host processor has to be able to interact directly with the FPGA and its off-chip memory. This setup is not always possible as the host processor might not be able to interact with both the FPGA and its off-chip memory. An additional hardware module would be required to deal various different FPGA system structure. The wrappers and hardware module are needed to allow complete portability.

Another future improvement is for the generator to automatically optimize the performance of the engine on any FPGA. The generator will have built in knowledge about the FPGA so that it can determine the best set of parameters to achieve maximum performance. Experimental data can be used to create a function that the generator can

use to determine the best values of the parameters. This allows greater portability as the user does not have to manually optimize the engine; the user simply has to specify the FPGA platform.

Appendix A

Compute Engine Generator Source Code

The compute engine generator has six functions including a main function and creates five modules of the compute engine. These five modules are the top module, the LU Processing module, the LU Controller module, the Data Transfer Unit module and the Marshalling Controller module which are all described in Chapter 3. The main function calls the other five functions which individually create a Verilog file that contains a module of the compute engine. This appendix will show the C source code for the six functions of the generator. Appendix A.1 shows the source code for the main function. Appendix A.2 lists the source code for the top module. Appendix A.3 shows the source code for the Marshalling Controller. Appendix A.4 lists the source code for the LU Processing module. Appendix A.5 shows the source code for the LU Controller. Appendix A.6 lists the source code for the Data Transfer Unit. More details about the generator can be found in Section 4.2.

A.1 Main Function

The main function computes internal variables from the parameters and calls the other five functions. Currently, the parameters are hard coded in the main function, requiring the user to modify the function directly to change the parameters. In future versions, the main function will read the parameters from an input file that the user can modify.

```
#include <stdio.h>
#include <math.h>
#define intlog2(x) (int)ceil(log(x)/log(2))

int main(int argc, char *argv[])
{
    int ramsize, ramwidth, ramsizewidth;
    int Nmax, Nwidth, mdivk, m, mwidth, n;
    int ddrwidth, ddrsizewidth, memconwidth, ddrrowwidth, full_rate;
    int ddrburstlen, burstlen, ratio_factor;
    int numPE, precision, PEwidth;
    int topsize, topsizewidth;
    int ramlat, toplat;
    int multlat, addlat, divlat, totaldivlat;
    int intdivlat;
    int topinputdelay, topoutputdelay;
    int meminputdelay, memoutputdelay;
    int datawidth, ratio;
    int fifosize;

    // core parameters
    numPE = 4;          // number of processing elements
    precision = 32;     // bits of precision
    ramsize = 64;       // address size of on chip ram block
    mdivk = 4;          // internal block size in multiple of # of PEs
    ddrwidth = 32;      // ddr datawidth
    ddrsizewidth = 24;  // ddr total address width
    ddrrowwidth = 13;   // ddr row address width
    ddrburstlen = 4;    // ddr burst length
    Nmax = 2000000;     // max matrix dimension
    addlat = 14;        // output latency of fp_add
    multlat = 11;       // output latency of fp_mult
    divlat = 33;        // output latency of fp_div

    // advanced parameters
    fifosize = 16;      // size of fifo buffer
    topinputdelay = 4;  // # of registers in top block input ports
    topoutputdelay = 2; // # of registers in top block output ports
    meminputdelay = 3;  // # of registers in current and left blocks input ports
    memoutputdelay = 1; // # of registers in current and left blocks output ports

    totaldivlat = divlat;
    ramlat = 2 + meminputdelay + memoutputdelay;     // total latency for reading ram
        block
    toplat = 2 + topinputdelay + topoutputdelay;     // total latency for reading
        temp_row block
    intdivlat = 2;
    full_rate = 1;
    m = numPE*mdivk;
    n = m;
    ramwidth = numPE*precision;
    topsize = numPE*mdivk*mdivk*numPE;  // address size of temp_row block
    ratio_factor = (full_rate == 0) ? 4 : 2;
    burstlen = ddrburstlen/ratio_factor;
```

```
    memconwidth = ddrwidth*ratio_factor;
    ramwidth = numPE*precision;
    ramsizewidth = intlog2(ramsize);
    topsizewidth = intlog2(topsize);
    Nwidth = intlog2(Nmax+1);
    mwidth = intlog2(m+2);
    ratio = pow(2, ceil(log(ramwidth*1.0/memconwidth)/log(2)));
    datawidth = memconwidth*ratio;

    // call functions to produce module in compute engine
    gentop(Nwidth, mwidth, ddrwidth, ddrsizewidth, burstlen, ddrrowwidth, memconwidth,
        ramwidth, ramsizewidth, 1);

    genLUP(numPE, precision, mwidth, ramwidth, ramsizewidth, topsizewidth, meminputdelay
        , memoutputdelay, topinputdelay, topoutputdelay);

    genLUC(numPE, precision, mwidth, mdivk, intdivlat, multlat, addlat, ramlat, ramlat,
        toplat, totaldivlat, ramwidth, ramsizewidth, topsizewidth);

    genMC(m, Nwidth, mdivk, ratio, ddrsizewidth, ramsizewidth);

    genDTU(burstlen, datawidth, memconwidth, ddrsizewidth, fifosize, ramwidth,
        ramsizewidth, ratio, ramlat, mwidth);

    return 0;
}
```

A.2 Top Module Function

The source code provided below creates the top module of the compute engine. This module implements the high level diagram of the compute engine shown in Figure 3.1.

```
#include <stdio.h>
#include <math.h>
#define intlog2(x) (int)ceil(log(x)/log(2))

void gentop(int Nwidth, int mwidth, int ddrwidth, int ddrsizewidth, int burstlen, int
    ddrrowwidth,
    int memconwidth, int ramwidth, int ramsizewidth, int mem_dqsn)
{
    FILE *fp;

    fp = fopen("top.v", "w");

    fprintf(fp, "//auto-generated top.v\n");
    fprintf(fp, "//top level module of LU factorization\n");
    fprintf(fp, "//by Wei Zhang\n\n");
    fprintf(fp, "module top (clk, ref_clk, global_reset_n, start, N, offset, done, ready
        ,\n");
    fprintf(fp, "    mem_addr, mem_ba, mem_cas_n, mem_cke, mem_clk, mem_clk_n, mem_cs_n,\
        n");
    if (mem_dqsn == 1)
    {
        fprintf(fp, "    mem_dm, mem_dq, mem_dqs, mem_dqsn, mem_odt, mem_ras_n, mem_we_n)
            ;\n");
    }
    else
    {
        fprintf(fp, "    mem_dm, mem_dq, mem_dqs, mem_odt, mem_ras_n, mem_we_n);\n");
    }
```

```
fprintf(fp, "\n");
fprintf(fp, "parameter NWIDTH = %i, BLOCKWIDTH = %i;\n", Nwidth, mwidth);
fprintf(fp, "parameter DDRWIDTH = %i, DDRNUMBYTES = %i, DDRSIZEWIDTH = %i;\n",
    ddrwidth, ddrwidth/8, ddrsizewidth);
fprintf(fp, "parameter BURSTLEN = %i;\n", burstlen);
fprintf(fp, "parameter MEMCONWIDTH = %i, MEMCONNUMBYTES = %i;\n", memconwidth,
    memconwidth/8);
fprintf(fp, "parameter RAMWIDTH = %i, RAMNUMBYTES = %i, RAMSIZEWIDTH = %i;\n",
    ramwidth, ramwidth/8, ramsizewidth);
fprintf(fp, "\n");
fprintf(fp, "input start;\n");
fprintf(fp, "input[NWIDTH-1:0] N;\n");
fprintf(fp, "input[DDRSIZEWIDTH-1:0] offset;\n");
fprintf(fp, "output done;\n");
fprintf(fp, "output [%i:0] mem_addr;\n", ddrrowwidth-1);
fprintf(fp, "output [1:0] mem_ba;\n");
fprintf(fp, "output mem_cas_n;\n");
fprintf(fp, "output mem_cke;\n");
fprintf(fp, "inout mem_clk;\n");
fprintf(fp, "inout mem_clk_n;\n");
fprintf(fp, "output mem_cs_n;\n");
fprintf(fp, "output [DDRNUMBYTES-1:0] mem_dm;\n");
fprintf(fp, "inout [DDRWIDTH-1:0] mem_dq;\n");
fprintf(fp, "inout [DDRNUMBYTES-1:0] mem_dqs;\n");
if (mem_dqsn == 1)
{
    fprintf(fp, "inout [DDRNUMBYTES-1:0] mem_dqsn;\n");
}
fprintf(fp, "output mem_odt;\n");
fprintf(fp, "output mem_ras_n;\n");
fprintf(fp, "output mem_we_n;\n");
fprintf(fp, "input clk, ref_clk;\n");
fprintf(fp, "input global_reset_n;\n");
fprintf(fp, "output ready;\n");
fprintf(fp, "\n");
fprintf(fp, "wire phy_clk;\n");
fprintf(fp, "wire[BLOCKWIDTH-1:0] m, n, loop;\n");
fprintf(fp, "wire[1:0] mode;\n");
fprintf(fp, "wire comp_start, comp_done;\n");
fprintf(fp, "wire dtu_write_req, dtu_read_req, dtu_ack, dtu_done;\n");
fprintf(fp, "wire [DDRSIZEWIDTH-1:0] dtu_mem_addr;\n");
fprintf(fp, "wire [RAMSIZEWIDTH-1:0] dtu_ram_addr;\n");
fprintf(fp, "wire [BLOCKWIDTH-1:0] dtu_size;\n");
fprintf(fp, "wire left_sel;\n");
fprintf(fp, "\n");
fprintf(fp, "wire[RAMWIDTH-1:0] curWriteDataMem, curReadDataMem;\n");
fprintf(fp, "wire[RAMSIZEWIDTH-1:0] curWriteAddrMem, curReadAddrMem;\n");
fprintf(fp, "wire[RAMNUMBYTES-1:0] curWriteByteEnMem;\n");
fprintf(fp, "wire curWriteEnMem;\n");
fprintf(fp, "wire[RAMWIDTH-1:0] leftWriteDataMem;\n");
fprintf(fp, "wire[RAMSIZEWIDTH-1:0] leftWriteAddrMem;\n");
fprintf(fp, "wire[RAMNUMBYTES-1:0] leftWriteByteEnMem;\n");
fprintf(fp, "wire leftWriteEnMem;\n");
fprintf(fp, "wire curMemSel, leftMemSel;\n");
fprintf(fp, "\n");
fprintf(fp, "wire burst_begin;\n");
fprintf(fp, "wire [MEMCONNUMBYTES-1:0] mem_local_be;\n");
fprintf(fp, "wire mem_local_read_req;\n");
fprintf(fp, "wire [BURSTLEN-1:0] mem_local_size;\n");
fprintf(fp, "wire [MEMCONWIDTH-1:0] mem_local_wdata;\n");
fprintf(fp, "wire mem_local_write_req;\n");
fprintf(fp, "wire [MEMCONWIDTH-1:0] mem_local_rdata;\n");
fprintf(fp, "wire mem_local_rdata_valid;\n");
fprintf(fp, "wire mem_local_ready;\n");
fprintf(fp, "wire mem_local_wdata_req;\n");
fprintf(fp, "wire reset_n;\n");
fprintf(fp, "wire [DDRSIZEWIDTH-1:0] mem_local_addr;\n");
fprintf(fp, "\n");
```

```
fprintf(fp, "wire[RAMWIDTH-1:0] ram_write_data, ram_read_data;\n");
fprintf(fp, "wire[RAMSIZEWIDTH-1:0] ram_write_addr, ram_read_addr;\n");
fprintf(fp, "wire[RAMNUMBYTES-1:0] ram_write_byte_en;\n");
fprintf(fp, "wire ram_write_en;\n");
fprintf(fp, "\n");
fprintf(fp, "MarshallingController MC (clk, start, done, N, offset,\n");
fprintf(fp, "    comp_start, m, n, loop, mode, comp_done, curMemSel, leftMemSel,\n");
fprintf(fp, "    dtu_write_req, dtu_read_req, dtu_mem_addr, dtu_ram_addr, dtu_size,
    dtu_ack, dtu_done, left_sel);\n");
fprintf(fp, "\n");
fprintf(fp, "// block that computes the LU factorization, with answer stored back
    into ram block\n");
fprintf(fp, "LUProcessing LUP (clk, comp_start, m, n, loop, mode, comp_done,\n");
fprintf(fp, "            curReadAddrMem, curReadDataMem, curWriteByteEnMem,
    curWriteDataMem, curWriteAddrMem, curWriteEnMem, curMemSel,\n");
fprintf(fp, "            leftWriteByteEnMem, leftWriteDataMem, leftWriteAddrMem,
    leftWriteEnMem, leftMemSel);\n");
fprintf(fp, "\n");
fprintf(fp, "DataTransferUnit DTU (clk, phy_clk, dtu_write_req, dtu_read_req,
    dtu_mem_addr, dtu_ram_addr, dtu_size, dtu_ack, dtu_done,\n");
fprintf(fp, "       ram_read_addr, ram_read_data, ram_write_byte_en, ram_write_data,
     ram_write_addr, ram_write_en,\n");
fprintf(fp, "       mem_local_rdata, mem_local_rdata_valid, mem_local_ready,
    mem_local_wdata_req, reset_n,\n");
fprintf(fp, "       burst_begin, mem_local_addr, mem_local_be, mem_local_read_req,
    mem_local_size,\n");
fprintf(fp, "       mem_local_wdata, mem_local_write_req);\n");
fprintf(fp, "\n");
fprintf(fp, "ddr2 memController (\n");
fprintf(fp, "    .global_reset_n (global_reset_n),\n");
fprintf(fp, "    .local_address (mem_local_addr),\n");
fprintf(fp, "    .local_be (mem_local_be),\n");
fprintf(fp, "    .local_init_done (ready),\n");
fprintf(fp, "    .local_rdata (mem_local_rdata),\n");
fprintf(fp, "    .local_rdata_valid (mem_local_rdata_valid),\n");
fprintf(fp, "    .local_read_req (mem_local_read_req),\n");
fprintf(fp, "    .local_ready (mem_local_ready),\n");
fprintf(fp, "    .local_refresh_ack (),\n");
fprintf(fp, "    .local_size (mem_local_size),\n");
fprintf(fp, "    .local_wdata (mem_local_wdata),\n");
fprintf(fp, "    .local_wdata_req (mem_local_wdata_req),\n");
fprintf(fp, "    .local_write_req (mem_local_write_req),\n");
fprintf(fp, "    .mem_addr (mem_addr),\n");
fprintf(fp, "    .mem_ba (mem_ba),\n");
fprintf(fp, "    .mem_cas_n (mem_cas_n),\n");
fprintf(fp, "    .mem_cke (mem_cke),\n");
fprintf(fp, "    .mem_clk (mem_clk),\n");
fprintf(fp, "    .mem_clk_n (mem_clk_n),\n");
fprintf(fp, "    .mem_cs_n (mem_cs_n),\n");
fprintf(fp, "    .mem_dm (mem_dm),\n");
fprintf(fp, "    .mem_dq (mem_dq),\n");
fprintf(fp, "    .mem_dqs (mem_dqs),\n");
fprintf(fp, "    .mem_odt (mem_odt),\n");
fprintf(fp, "    .mem_ras_n (mem_ras_n),\n");
fprintf(fp, "    .mem_we_n (mem_we_n),\n");
if (mem_dqsn == 1)
{
    fprintf(fp, "    .mem_dqsn (mem_dqsn),\n");
    fprintf(fp, "    .oct_ctl_rs_value (0),\n");
    fprintf(fp, "    .oct_ctl_rt_value (0),\n");
}
fprintf(fp, "    .phy_clk (phy_clk),\n");
fprintf(fp, "    .pll_ref_clk (ref_clk),\n");
fprintf(fp, "    .reset_phy_clk_n (reset_n),\n");
fprintf(fp, "    .soft_reset_n (1)\n");
fprintf(fp, "    );\n");
fprintf(fp, "\n");
fprintf(fp, "assign curReadAddrMem = ram_read_addr;\n");
```

```
    fprintf(fp, "assign curWriteByteEnMem = ram_write_byte_en;\n");
    fprintf(fp, "assign curWriteDataMem = ram_write_data;\n");
    fprintf(fp, "assign curWriteAddrMem = ram_write_addr;\n");
    fprintf(fp, "assign curWriteEnMem = ram_write_en && (left_sel == 0);\n");
    fprintf(fp, "assign leftWriteByteEnMem = ram_write_byte_en;\n");
    fprintf(fp, "assign leftWriteDataMem = ram_write_data;\n");
    fprintf(fp, "assign leftWriteAddrMem = ram_write_addr;\n");
    fprintf(fp, "assign leftWriteEnMem = ram_write_en && (left_sel == 1);\n");
    fprintf(fp, "assign ram_read_data = curReadDataMem;\n");

    fprintf(fp, "endmodule\n");
    fclose(fp);
}
```

A.3 Marshalling Controller Module Function

The source code provided below creates the Marshalling Controller module, which is described in Section 3.1. This module directs the overall computation and data marshalling in the compute engine to perform the block LU factorization.

```
#include <stdio.h>
#include <math.h>
#define intlog2(x) (int)ceil(log(x)/log(2))

void genMC(int m, int Nwidth, int mdivk, int ratio, int ddrsizewidth, int ramsizewidth)
{
    FILE *fp;
    int n, ndivk, blockwidth;

    n = m;
    ndivk = mdivk;
    blockwidth = intlog2(m+2);

    fp = fopen("MarshallingController.v", "w");

    fprintf(fp, "module MarshallingController (clk, start, done, input_N, offset,\n");
    fprintf(fp, "    comp_start, block_m, block_n, loop, mode, comp_done, cur_mem_sel,
        left_mem_sel,\n");
    fprintf(fp, "    dtu_write_req, dtu_read_req, dtu_mem_addr, dtu_ram_addr, dtu_size,
        dtu_ack, dtu_done, left_sel);\n");
    fprintf(fp, "\n");
    fprintf(fp, "parameter BLOCKM = %i, BLOCKN = %i;\n", m, n);
    fprintf(fp, "parameter BLOCKMDIVK = %i;\n", mdivk);
    fprintf(fp, "parameter MEMBLOCKM = %i, MEMBLOCKN = %i;\n", mdivk*ratio, ndivk*ratio)
        ;
    fprintf(fp, "parameter NWIDTH = %i, BLOCKWIDTH = %i;\n", Nwidth, blockwidth);
    fprintf(fp, "parameter DDRSIZEWIDTH = %i, RAMSIZEWIDTH = %i;\n", ddrsizewidth,
        ramsizewidth);
    fprintf(fp, "\n");
    fprintf(fp, "input clk;\n");
    fprintf(fp, "input start;\n");
    fprintf(fp, "output done;\n");
    fprintf(fp, "input [NWIDTH-1:0] input_N;\n");
    fprintf(fp, "input [DDRSIZEWIDTH-1:0] offset;\n");
    fprintf(fp, "\n");
    fprintf(fp, "// for computation section\n");
    fprintf(fp, "output comp_start;\n");
    fprintf(fp, "output [BLOCKWIDTH-1:0] block_m, block_n, loop;\n");
```

```
fprintf(fp, "output [1:0] mode;\n");
fprintf(fp, "input comp_done;\n");
fprintf(fp, "output cur_mem_sel, left_mem_sel;\n");
fprintf(fp, "\n");
fprintf(fp, "// for data marshaller section\n");
fprintf(fp, "output dtu_write_req, dtu_read_req;\n");
fprintf(fp, "output [DDRSIZEWIDTH-1:0] dtu_mem_addr;\n");
fprintf(fp, "output [RAMSIZEWIDTH-1:0] dtu_ram_addr;\n");
fprintf(fp, "output [BLOCKWIDTH-1:0] dtu_size;\n");
fprintf(fp, "input dtu_ack, dtu_done;\n");
fprintf(fp, "output left_sel;\n");
fprintf(fp, "\n");
fprintf(fp, "parameter START = 0, SETUP = 1, FIRST = 2, MODE0_SETUP = 3, MODE0_WAIT
    = 4, MODE0 = 5, MODE1_SETUP = 6, MODE1_WAIT = 7, MODE1 = 8, \n");
fprintf(fp, "   MODE2_SETUP = 9, MODE2_WAIT = 10, MODE2 = 11, MODE3_SETUP = 12,
   MODE3_WAIT = 13, MODE3 = 14, STALL = 15, STALL_WAIT = 16, WAIT = 17,\n");
fprintf(fp, "   FINAL_WRITE = 18, FINAL_WAIT = 19, IDLE = 20, LAST_SETUP = 21,
   LAST_SETUP_WAIT = 22, LAST = 23, LAST_WAIT = 24;\n");
fprintf(fp, "parameter MEM_IDLE = 0, MEM_WRITE = 1, MEM_WRITE_WAIT = 2,
   MEM_CHECK_DONE = 3, MEM_READ = 4, MEM_READ_WAIT = 5, MEM_DONE = 6, MEM_WAIT_DONE
    = 7;\n");
fprintf(fp, "\n");
fprintf(fp, "reg [4:0] cur_state, next_state;\n");
fprintf(fp, "reg [NWIDTH-1:0] comp_N, N, mcount, ncount, Ndivk, mem_N;\n");
fprintf(fp, "reg [1:0] mode;\n");
fprintf(fp, "reg [BLOCKWIDTH-1:0] block_m, block_n, loop, read_n;\n");
fprintf(fp, "reg [BLOCKWIDTH-1:0] write_n, write_n_buf;\n");
fprintf(fp, "reg left_mem_sel, cur_mem_sel, no_left_switch;\n");
fprintf(fp, "\n");
fprintf(fp, "reg [3:0] cur_mem_state, next_mem_state;\n");
fprintf(fp, "reg [RAMSIZEWIDTH:0] ram_addr;\n");
fprintf(fp, "reg [DDRSIZEWIDTH-1:0] mem_addr;\n");
fprintf(fp, "reg [DDRSIZEWIDTH-1:0] mem_base, mem_top, mem_write, mem_left, mem_cur
   ;\n");
fprintf(fp, "reg [DDRSIZEWIDTH-1:0] mem_write_buf;\n");
fprintf(fp, "reg [BLOCKWIDTH-1:0] mem_count;\n");
fprintf(fp, "reg [1:0] mem_read;\n");
fprintf(fp, "reg [BLOCKWIDTH-1:0] mem_write_size, mem_write_size_buf, mem_read_size
   ;\n");
fprintf(fp, "wire mem_done;\n");
fprintf(fp, "\n");
fprintf(fp, "assign done = (cur_state == IDLE);\n");
fprintf(fp, "assign dtu_ram_addr = ram_addr;\n");
fprintf(fp, "assign dtu_mem_addr = mem_addr;\n");
fprintf(fp, "assign dtu_size = (cur_mem_state == MEM_WRITE) ? mem_write_size :
   mem_read_size;\n");
fprintf(fp, "assign comp_start = (cur_state == MODE0)||(cur_state == MODE1)||(
   cur_state == MODE2)||(cur_state == MODE3)||(cur_state == FIRST)||(cur_state ==
   LAST);\n");
fprintf(fp, "assign dtu_write_req = (cur_mem_state == MEM_WRITE);\n");
fprintf(fp, "assign dtu_read_req = (cur_mem_state == MEM_READ);\n");
fprintf(fp, "assign mem_done = (cur_mem_state == MEM_DONE)&&(dtu_done == 1);\n");
fprintf(fp, "assign left_sel = mem_read == 1 && (cur_mem_state == MEM_READ ||
   cur_mem_state == MEM_READ_WAIT || cur_mem_state == MEM_WAIT_DONE);\n");
fprintf(fp, "\n");

fprintf(fp, "// FSM to produce memory instructions to DTU\n");
fprintf(fp, "always @ (*)\n");
fprintf(fp, "begin\n");
fprintf(fp, "   case (cur_mem_state)\n");
fprintf(fp, "   MEM_IDLE:\n");
fprintf(fp, "   begin\n");
fprintf(fp, "       if (cur_state == START)\n");
fprintf(fp, "           next_mem_state <= MEM_CHECK_DONE;\n");
fprintf(fp, "       else\n");
fprintf(fp, "           next_mem_state <= MEM_IDLE;\n");
fprintf(fp, "   end\n");
fprintf(fp, "   MEM_DONE:\n");
```

```
fprintf(fp, "    begin\n");
fprintf(fp, "        if (cur_state == MODE0 || cur_state == MODE1 || cur_state ==
    MODE2 || \n");
fprintf(fp, "            cur_state == MODE3 || cur_state == FINAL_WRITE || cur_state
    == LAST_SETUP)\n");
fprintf(fp, "            next_mem_state <= MEM_WRITE;\n");
fprintf(fp, "        else if (cur_state == FIRST)\n");
fprintf(fp, "            next_mem_state <= MEM_CHECK_DONE;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_mem_state <= MEM_DONE;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MEM_WRITE:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        next_mem_state <= MEM_WRITE_WAIT;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MEM_WRITE_WAIT:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (dtu_ack == 1)\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            if (mem_count == write_n)\n");
fprintf(fp, "                next_mem_state <= MEM_WAIT_DONE;\n");
fprintf(fp, "            else \n");
fprintf(fp, "                next_mem_state <= MEM_WRITE;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_mem_state <= MEM_WRITE_WAIT;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MEM_WAIT_DONE:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (dtu_done == 1)\n");
fprintf(fp, "            next_mem_state <= MEM_CHECK_DONE;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_mem_state <= MEM_WAIT_DONE;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MEM_CHECK_DONE:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mem_read == 0)\n");
fprintf(fp, "            next_mem_state <= MEM_DONE;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_mem_state <= MEM_READ;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MEM_READ:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        next_mem_state <= MEM_READ_WAIT;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MEM_READ_WAIT:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (dtu_ack == 1)\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            if (mem_count == read_n)\n");
fprintf(fp, "                next_mem_state <= MEM_WAIT_DONE;\n");
fprintf(fp, "            else\n");
fprintf(fp, "                next_mem_state <= MEM_READ;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_mem_state <= MEM_READ_WAIT;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    default:\n");
fprintf(fp, "        next_mem_state <= MEM_IDLE;\n");
fprintf(fp, "    endcase\n");
fprintf(fp, "end\n");
fprintf(fp, "\n");

fprintf(fp, "always @ (posedge clk)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    if (cur_mem_state == MEM_DONE || cur_mem_state == MEM_IDLE)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        ram_addr <= 0;\n");
```

```
fprintf(fp, "        mem_addr <= mem_write;\n");
fprintf(fp, "        if (next_state == LAST_WAIT || next_state == FINAL_WAIT ||
    next_state == STALL)\n");
fprintf(fp, "            mem_read <= 0;\n");
fprintf(fp, "        else if (next_state == MODE0_SETUP || next_state == SETUP ||
    cur_state == MODE0 || next_state == LAST_SETUP_WAIT)\n");
fprintf(fp, "            mem_read <= 1;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            mem_read <= 2;\n");
fprintf(fp, "        mem_count <= 0;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else if (cur_mem_state == MEM_CHECK_DONE)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mem_read == 2)\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            mem_addr <= mem_left;\n");
fprintf(fp, "            read_n <= loop;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        else\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            mem_addr <= mem_cur;\n");
fprintf(fp, "            read_n <= block_n;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        mem_read <= mem_read - 1;\n");
fprintf(fp, "        mem_count <= 0;\n");
fprintf(fp, "        ram_addr <= 0;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else if (cur_mem_state == MEM_WRITE || cur_mem_state == MEM_READ)\n"
    );
fprintf(fp, "    begin\n");
fprintf(fp, "        ram_addr <= ram_addr + BLOCKMDIVK;\n");
fprintf(fp, "        mem_addr <= mem_addr + Ndivk;\n");
fprintf(fp, "        mem_count <= mem_count + 1;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    \n");
fprintf(fp, "end\n");
fprintf(fp, "\n");

fprintf(fp, "// FSM to determine the block LU factorization algorithm\n");
fprintf(fp, "always @ (*)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    case (cur_state)\n");
fprintf(fp, "    START:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        next_state <= SETUP;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    SETUP:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        next_state <= WAIT;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    WAIT:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mem_done == 1)\n");
fprintf(fp, "            next_state <= FIRST;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= WAIT;\n");
fprintf(fp, "\n");
fprintf(fp, "    end\n");
fprintf(fp, "    FIRST:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mcount < comp_N)\n");
fprintf(fp, "            next_state <= MODE1_SETUP;\n");
fprintf(fp, "        else if (ncount < comp_N)\n");
fprintf(fp, "            next_state <= MODE2_SETUP;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= LAST_WAIT;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MODE0_SETUP:\n");
```

```
fprintf(fp, "    begin\n");
fprintf(fp, "        next_state <= MODE0_WAIT;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MODE0_WAIT:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mem_done == 1 && comp_done == 1)\n");
fprintf(fp, "            next_state <= MODE0;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= MODE0_WAIT;\n");
fprintf(fp, "\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MODE0:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mcount < comp_N)\n");
fprintf(fp, "            next_state <= MODE1_SETUP;\n");
fprintf(fp, "        else if (ncount < comp_N)\n");
fprintf(fp, "            next_state <= MODE2_SETUP;\n");
fprintf(fp, "        else\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            next_state <= LAST_WAIT;\n");
fprintf(fp, "        end\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MODE1_SETUP:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        next_state <= MODE1_WAIT;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MODE1_WAIT:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mem_done == 1 && comp_done == 1)\n");
fprintf(fp, "            next_state <= MODE1;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= MODE1_WAIT;\n");
fprintf(fp, "\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MODE1:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mcount < comp_N)\n");
fprintf(fp, "            next_state <= MODE1_SETUP;\n");
fprintf(fp, "        else if (ncount < comp_N)\n");
fprintf(fp, "            next_state <= MODE2_SETUP;\n");
fprintf(fp, "        else if (comp_N <= BLOCKN + BLOCKN)\n");
fprintf(fp, "            next_state <= STALL;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= MODE0_SETUP;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MODE2_SETUP:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        next_state <= MODE2_WAIT;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MODE2_WAIT:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mem_done == 1 && comp_done == 1)\n");
fprintf(fp, "            next_state <= MODE2;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= MODE2_WAIT;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MODE2:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mcount < comp_N)\n");
fprintf(fp, "            next_state <= MODE3_SETUP;\n");
fprintf(fp, "        else if (ncount < comp_N)\n");
fprintf(fp, "            next_state <= MODE2_SETUP;\n");
fprintf(fp, "        else if (comp_N <= BLOCKN + BLOCKN)\n");
fprintf(fp, "            next_state <= STALL;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= MODE0_SETUP;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MODE3_SETUP:\n");
```

```
fprintf(fp, "    begin\n");
fprintf(fp, "        next_state <= MODE3_WAIT;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MODE3_WAIT:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mem_done == 1 && comp_done == 1)\n");
fprintf(fp, "            next_state <= MODE3;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= MODE3_WAIT;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MODE3:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mcount < comp_N)\n");
fprintf(fp, "            next_state <= MODE3_SETUP;\n");
fprintf(fp, "        else if (ncount < comp_N)\n");
fprintf(fp, "            next_state <= MODE2_SETUP;\n");
fprintf(fp, "        else if (comp_N <= BLOCKN + BLOCKN)\n");
fprintf(fp, "            next_state <= STALL;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= MODE0_SETUP;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    STALL:\n");
fprintf(fp, "        next_state <= STALL_WAIT;\n");
fprintf(fp, "    STALL_WAIT:\n");
fprintf(fp, "        if (mem_done == 1 && comp_done == 1)\n");
fprintf(fp, "            next_state <= LAST_SETUP;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= STALL_WAIT;\n");
fprintf(fp, "    LAST_SETUP:\n");
fprintf(fp, "        next_state <= LAST_SETUP_WAIT;\n");
fprintf(fp, "    LAST_SETUP_WAIT:\n");
fprintf(fp, "        if (mem_done == 1 && comp_done == 1)\n");
fprintf(fp, "            next_state <= LAST;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= LAST_SETUP_WAIT;\n");
fprintf(fp, "    LAST:\n");
fprintf(fp, "        next_state <= LAST_WAIT;\n");
fprintf(fp, "    LAST_WAIT:\n");
fprintf(fp, "        if (mem_done == 1 && comp_done == 1)\n");
fprintf(fp, "            next_state <= FINAL_WRITE;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= LAST_WAIT;\n");
fprintf(fp, "    FINAL_WRITE:\n");
fprintf(fp, "        next_state <= FINAL_WAIT;\n");
fprintf(fp, "    FINAL_WAIT:\n");
fprintf(fp, "        if (mem_done == 1)\n");
fprintf(fp, "            next_state <= IDLE;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= FINAL_WAIT;\n");
fprintf(fp, "    IDLE:\n");
fprintf(fp, "        if (start)\n");
fprintf(fp, "            next_state <= SETUP;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            next_state <= IDLE;\n");
fprintf(fp, "    default:\n");
fprintf(fp, "        next_state <= START;\n");
fprintf(fp, "    endcase\n");
fprintf(fp, "end\n");
fprintf(fp, "\n");

fprintf(fp, "always @ (posedge clk)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    if (start)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        cur_state <= START;\n");
fprintf(fp, "        cur_mem_state <= MEM_IDLE;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else\n");
```

```
fprintf(fp, "    begin\n");
fprintf(fp, "        cur_state <= next_state;\n");
fprintf(fp, "        cur_mem_state <= next_mem_state;\n");
fprintf(fp, "    end\n");
fprintf(fp, "end\n");
fprintf(fp, "\n");

fprintf(fp, "always @ (*)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    case (cur_state)\n");
fprintf(fp, "    MODE1:\n");
fprintf(fp, "        mode = 1;\n");
fprintf(fp, "    MODE2:\n");
fprintf(fp, "        mode = 2;\n");
fprintf(fp, "    MODE3:\n");
fprintf(fp, "        mode = 3;\n");
fprintf(fp, "    default:\n");
fprintf(fp, "        mode = 0;\n");
fprintf(fp, "    endcase\n");
fprintf(fp, "end\n");
fprintf(fp, "\n");
fprintf(fp, "always @ (posedge clk)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    if (start)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        comp_N <= input_N;\n");
fprintf(fp, "        N <= input_N;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else if (next_state == MODE0)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        comp_N <= comp_N - BLOCKN;\n");
fprintf(fp, "    end\n");
fprintf(fp, "\n");
fprintf(fp, "    Ndivk <= ((N+BLOCKM-1)/BLOCKM)*MEMBLOCKM;\n");
fprintf(fp, "    mem_N <= Ndivk*BLOCKN;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (start)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        mem_base <= offset;\n");
fprintf(fp, "        mem_top <= offset;\n");
fprintf(fp, "        mem_left <= offset;\n");
fprintf(fp, "        mem_cur <= offset;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else if (cur_state == MODE0_SETUP)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        mem_base <= mem_base + mem_N+MEMBLOCKN;\n");
fprintf(fp, "        mem_top <= mem_base + mem_N+MEMBLOCKN;\n");
fprintf(fp, "        mem_cur <= mem_base + mem_N+MEMBLOCKN;\n");
fprintf(fp, "        mem_left <= mem_base + mem_N+MEMBLOCKN;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else if (cur_state == MODE1_SETUP)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        mem_cur <= mem_cur + MEMBLOCKM;\n");
fprintf(fp, "    end \n");
fprintf(fp, "    else if (cur_state == MODE3_SETUP)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        mem_cur <= mem_cur + MEMBLOCKM;\n");
fprintf(fp, "        mem_left <= mem_left + MEMBLOCKM;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else if (cur_state == MODE2_SETUP)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        mem_cur <= mem_top + mem_N;\n");
fprintf(fp, "        mem_top <= mem_top + mem_N;\n");
fprintf(fp, "        mem_left <= mem_base;\n");
fprintf(fp, "    end\n");
fprintf(fp, "\n");
fprintf(fp, "    if (cur_state == SETUP)\n");
fprintf(fp, "    begin\n");
```

```
fprintf(fp, "        mem_write <= 0;\n");
fprintf(fp, "        mem_write_buf <= 0;\n");
fprintf(fp, "        mem_write_size <= BLOCKMDIVK;\n");
fprintf(fp, "        mem_write_size_buf <= BLOCKMDIVK;\n");
fprintf(fp, "        write_n <= block_n;\n");
fprintf(fp, "        write_n_buf <= block_n;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else if (cur_mem_state == MEM_CHECK_DONE && mem_read == 0)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        mem_write <= mem_write_buf;\n");
fprintf(fp, "        mem_write_buf <= mem_cur;\n");
fprintf(fp, "        mem_write_size <= mem_write_size_buf;\n");
fprintf(fp, "        mem_write_size_buf <= mem_read_size;\n");
fprintf(fp, "        write_n <= write_n_buf;\n");
fprintf(fp, "        write_n_buf <= block_n;\n");
fprintf(fp, "    end\n");
fprintf(fp, "\n");
fprintf(fp, "    mem_read_size <= BLOCKMDIVK;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (start)\n");
fprintf(fp, "        loop <= BLOCKN;\n");
fprintf(fp, "    else if (next_state == LAST)\n");
fprintf(fp, "        loop <= comp_N[8:0] - BLOCKN;\n");
fprintf(fp, "    \n");
fprintf(fp, "    if (next_state == MODE0_SETUP || next_state == MODE2_SETUP || start)
    \n");
fprintf(fp, "        mcount <= BLOCKM;\n");
fprintf(fp, "    else if (next_state == MODE1_SETUP || next_state == MODE3_SETUP)\n")
    ;
fprintf(fp, "        mcount <= mcount+BLOCKM;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (next_state == MODE0_SETUP || start)\n");
fprintf(fp, "        ncount <= BLOCKN;\n");
fprintf(fp, "    else if (next_state == MODE2_SETUP)\n");
fprintf(fp, "        ncount <= ncount+BLOCKN;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (mcount < comp_N)\n");
fprintf(fp, "        block_m <= BLOCKM;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        block_m <= comp_N - mcount + BLOCKM;\n");
fprintf(fp, "    \n");
fprintf(fp, "    if (ncount < comp_N)\n");
fprintf(fp, "        block_n <= BLOCKN;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        block_n <= comp_N - ncount + BLOCKN;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (start)\n");
fprintf(fp, "        cur_mem_sel <= 0;\n");
fprintf(fp, "    else if ((cur_state == MODE0)||(cur_state == MODE1)||(cur_state ==
    MODE2)||(cur_state == MODE3)||\n");
fprintf(fp, "         (cur_state == FIRST)||(cur_state == FINAL_WRITE)||(cur_state ==
     LAST_SETUP)||(cur_state == LAST))\n");
fprintf(fp, "        cur_mem_sel <= !cur_mem_sel;\n");
fprintf(fp, "    \n");
fprintf(fp, "    if (start)\n");
fprintf(fp, "        no_left_switch <= 0;\n");
fprintf(fp, "    else if ((cur_state == MODE0)||(cur_state == FIRST))\n");
fprintf(fp, "        no_left_switch <= 1;\n");
fprintf(fp, "    else if ((cur_state == MODE1)||(cur_state == MODE2)||(cur_state ==
    MODE3)||\n");
fprintf(fp, "         (cur_state == FINAL_WRITE)||(cur_state == LAST_SETUP))\n");
fprintf(fp, "        no_left_switch <= 0;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (start)\n");
fprintf(fp, "        left_mem_sel <= 0;\n");
fprintf(fp, "    else if (((cur_state == MODE0)||(cur_state == MODE1)||(cur_state ==
    MODE2)||(cur_state == MODE3)||\n");
```

```
    fprintf(fp, "        (cur_state == FIRST)||(cur_state == FINAL_WRITE)||(cur_state ==
        LAST_SETUP))&&(no_left_switch == 0))\n");
    fprintf(fp, "        left_mem_sel <= !left_mem_sel;\n");
    fprintf(fp, "end\n");
    fprintf(fp, "\n");

    fprintf(fp, "endmodule\n");
    fclose(fp);
}
```

A.4 LU Processing Module Function

The source code provided below creates the LU Processing module, which is described in Section 3.3. This module contains the processing elements that perform the desired computation.

```
#include <stdio.h>
#include <math.h>
#define intlog2(x) (int)ceil(log(x)/log(2))

void genLUP(int numPE, int precision, int mwidth, int ramwidth, int ramsizewidth, int
    topsizewidth, int meminputdelay, int memoutputdelay, int topinputdelay, int
    topoutputdelay)
{
    FILE *fp;
    int PEwidth;
    int i, upperIdx, lowerIdx;

    PEwidth = intlog2(numPE);

    fp = fopen("LUProcessing.v", "w");

    fprintf(fp, "//auto-generated LUProcessing.v\n");
    fprintf(fp, "//datapath for computating LU factorization\n");
    fprintf(fp, "//by Wei Zhang\n\n");
    fprintf(fp, "module LUProcessing (clk, start, m, n, loop, mode, done, \n");
    fprintf(fp, "            curReadAddrMem, curReadDataMem, curWriteByteEnMem,
        curWriteDataMem, curWriteAddrMem, curWriteEnMem, curMemSel,\n");
    fprintf(fp, "            leftWriteByteEnMem, leftWriteDataMem, leftWriteAddrMem,
        leftWriteEnMem, leftMemSel);\n");
    fprintf(fp, "\n\n");

    fprintf(fp, "parameter PRECISION = %i, NUMPE = %i, PEWIDTH = %i, BLOCKWIDTH = %i;\n"
        , precision, numPE, PEwidth, mwidth);
    fprintf(fp, "parameter RAMWIDTH = %i, RAMNUMBYTES = %i, RAMSIZEWIDTH = %i,
        TOPSIZEWIDTH = %i;\n", ramwidth, ramwidth/8, ramsizewidth, topsizewidth);
    fprintf(fp, "parameter TOPINPUTDELAY = %i, TOPOUTPUTDELAY = %i;\n", topinputdelay,
        topoutputdelay);
    fprintf(fp, "parameter MEMINPUTDELAY = %i, MEMOUTPUTDELAY = %i;\n", meminputdelay,
        memoutputdelay);
    fprintf(fp, "\n");

    fprintf(fp, "input clk, start;\n");
    fprintf(fp, "input[BLOCKWIDTH-1:0] m, n, loop;\n");
    fprintf(fp, "input[1:0] mode;\n");
    fprintf(fp, "output done;\n");
    fprintf(fp, "wire[RAMWIDTH-1:0] curWriteData0, curWriteData1;\n");
```

```
fprintf(fp, "wire[RAMSIZEWIDTH-1:0] curWriteAddr0, curReadAddr0, curWriteAddr1,
    curReadAddr1;\n");
fprintf(fp, "wire[RAMWIDTH-1:0] curReadData0, curReadData1;\n");
fprintf(fp, "wire[RAMNUMBYTES-1:0] curWriteByteEn0, curWriteByteEn1;\n");
fprintf(fp, "wire curWriteEn0, curWriteEn1;\n\n");

fprintf(fp, "input[RAMWIDTH-1:0] curWriteDataMem;\n");
fprintf(fp, "output[RAMWIDTH-1:0] curReadDataMem;\n");
fprintf(fp, "input[RAMSIZEWIDTH-1:0] curWriteAddrMem, curReadAddrMem;\n");
fprintf(fp, "input[RAMNUMBYTES-1:0] curWriteByteEnMem;\n");
fprintf(fp, "input curWriteEnMem;\n");
fprintf(fp, "input[RAMWIDTH-1:0] leftWriteDataMem;\n");
fprintf(fp, "input[RAMSIZEWIDTH-1:0] leftWriteAddrMem;\n");
fprintf(fp, "input[RAMNUMBYTES-1:0] leftWriteByteEnMem;\n");
fprintf(fp, "input leftWriteEnMem;\n");
fprintf(fp, "input leftMemSel, curMemSel;\n\n");

fprintf(fp, "wire[RAMWIDTH-1:0] curReadDataLU, curReadDataMem;\n");
fprintf(fp, "wire[RAMWIDTH-1:0] curWriteDataLU, curWriteDataMem;\n");
fprintf(fp, "wire[RAMSIZEWIDTH-1:0] curWriteAddrLU, curWriteAddrMem, curReadAddrLU,
    curReadAddrMem;\n");
fprintf(fp, "wire[RAMNUMBYTES-1:0] curWriteByteEnLU, curWriteByteEnMem;\n");
fprintf(fp, "wire curWriteEnLU, curWriteEnMem;\n\n");

if (memoutputdelay > 0) {
    fprintf(fp, "reg[RAMWIDTH-1:0] curReadData0Reg[MEMOUTPUTDELAY-1:0],
        curReadData1Reg[MEMOUTPUTDELAY-1:0];\n");
    fprintf(fp, "reg[RAMWIDTH-1:0] leftReadData0Reg[MEMOUTPUTDELAY-1:0],
        leftReadData1Reg[MEMOUTPUTDELAY-1:0];\n");
}

if (meminputdelay > 0)
{
    fprintf(fp, "reg[RAMWIDTH-1:0] curWriteData0Reg[MEMINPUTDELAY-1:0],
        curWriteData1Reg[MEMINPUTDELAY-1:0];\n");
    fprintf(fp, "reg[RAMSIZEWIDTH-1:0] curWriteAddr0Reg[MEMINPUTDELAY-1:0],
        curReadAddr0Reg[MEMINPUTDELAY-1:0];\n");
    fprintf(fp, "reg[RAMSIZEWIDTH-1:0] curWriteAddr1Reg[MEMINPUTDELAY-1:0],
        curReadAddr1Reg[MEMINPUTDELAY-1:0];\n");
    fprintf(fp, "reg[RAMNUMBYTES-1:0] curWriteByteEn0Reg[MEMINPUTDELAY-1:0],
        curWriteByteEn1Reg[MEMINPUTDELAY-1:0];\n");
    fprintf(fp, "reg curWriteEn0Reg[MEMINPUTDELAY-1:0], curWriteEn1Reg[MEMINPUTDELAY
        -1:0];\n");
    fprintf(fp, "reg[RAMWIDTH-1:0] leftWriteData0Reg[MEMINPUTDELAY-1:0],
        leftWriteData1Reg[MEMINPUTDELAY-1:0];\n");
    fprintf(fp, "reg[RAMSIZEWIDTH-1:0] leftWriteAddr0Reg[MEMINPUTDELAY-1:0],
        leftReadAddr0Reg[MEMINPUTDELAY-1:0];\n");
    fprintf(fp, "reg[RAMSIZEWIDTH-1:0] leftWriteAddr1Reg[MEMINPUTDELAY-1:0],
        leftReadAddr1Reg[MEMINPUTDELAY-1:0];\n");
    fprintf(fp, "reg[RAMNUMBYTES-1:0] leftWriteByteEn0Reg[MEMINPUTDELAY-1:0],
        leftWriteByteEn1Reg[MEMINPUTDELAY-1:0];\n");
    fprintf(fp, "reg leftWriteEn0Reg[MEMINPUTDELAY-1:0], leftWriteEn1Reg[
        MEMINPUTDELAY-1:0];\n");
}
fprintf(fp, "\n");

fprintf(fp, "reg[PRECISION-1:0] multOperand;\n");
fprintf(fp, "reg[PRECISION-1:0] diag;\n");
fprintf(fp, "wire[PRECISION-1:0] recResult;\n");
fprintf(fp, "wire[PRECISION-1:0] multA[NUMPE-1:0], multResult[NUMPE-1:0];\n");
fprintf(fp, "wire[PRECISION-1:0] addA[NUMPE-1:0], addResult[NUMPE-1:0];\n\n");

fprintf(fp, "wire[RAMWIDTH-1:0] leftReadData0, leftReadData1, leftWriteData0,
    leftWriteData1;\n");
fprintf(fp, "wire[RAMSIZEWIDTH-1:0] leftWriteAddr0, leftWriteAddr1, leftReadAddr0,
    leftReadAddr1;\n");
fprintf(fp, "wire[RAMNUMBYTES-1:0] leftWriteByteEn0, leftWriteByteEn1;\n");
fprintf(fp, "wire leftWriteEn0, leftWriteEn1;\n");
```

```
fprintf(fp, "wire[RAMWIDTH-1:0] leftReadDataLU, leftWriteDataLU, leftWriteDataMem;\n
    ");
fprintf(fp, "wire[RAMSIZEWIDTH-1:0] leftWriteAddrLU, leftWriteAddrMem,
    leftReadAddrLU;\n");
fprintf(fp, "wire[RAMNUMBYTES-1:0] leftWriteByteEnLU, leftWriteByteEnMem;\n");
fprintf(fp, "wire leftWriteEnLU, leftWriteEnMem;\n\n");

fprintf(fp, "wire[PRECISION-1:0] topWriteData;\n");
fprintf(fp, "reg[PRECISION-1:0] topWriteDataLU;\n");
fprintf(fp, "wire[PRECISION-1:0] topReadData, topReadDataLU;\n");
fprintf(fp, "wire[TOPSIZEWIDTH-1:0] topWriteAddr, topWriteAddrLU, topReadAddr,
    topReadAddrLU;\n");
fprintf(fp, "wire topWriteEn, topWriteEnLU;\n\n");

if (topoutputdelay != 0) {
    fprintf(fp, "reg[PRECISION-1:0] topReadDataReg[TOPOUTPUTDELAY-1:0];\n");
}
if (topinputdelay != 0) {
    fprintf(fp, "reg[PRECISION-1:0] topWriteDataReg[TOPINPUTDELAY-1:0];\n");
    fprintf(fp, "reg[TOPSIZEWIDTH-1:0] topWriteAddrReg[TOPINPUTDELAY-1:0],
        topReadAddrReg[TOPINPUTDELAY-1:0];\n");
    fprintf(fp, "reg topWriteEnReg[TOPINPUTDELAY-1:0];\n\n");
}

fprintf(fp, "wire[RAMWIDTH-1:0] rcWriteData;\n");
fprintf(fp, "wire leftWriteSel, curWriteSel, topSourceSel;\n");
fprintf(fp, "wire diagEn;\n");
fprintf(fp, "wire[PEWIDTH-1:0] diagSel, topWriteSel;\n\n");

fprintf(fp, "wire MOSel;\n");
fprintf(fp, "wire MOEn;\n\n");

fprintf(fp, "// control block\n");
fprintf(fp, "LUController conBlock (clk, start, m, n, loop, mode, done, \n");
fprintf(fp, "                       curReadAddrLU, curWriteAddrLU, curWriteByteEnLU,
    curWriteEnLU, curWriteSel,\n");
fprintf(fp, "                       leftReadAddrLU, leftWriteAddrLU, leftWriteByteEnLU,
     leftWriteEnLU,  leftWriteSel,\n");
fprintf(fp, "                       topReadAddrLU, topWriteAddrLU, topWriteEnLU,
    topWriteSel, topSourceSel, diagSel, diagEn, MOSel, MOEn);\n\n");
fprintf(fp, "// fp_div unit\n");

if (precision == 64) {
    fprintf(fp, "div rec (clk, 64'h3FF0000000000000, diag, recResult);\n\n");
}
else {
    fprintf(fp, "div rec (clk, 32'h3F800000, diag, recResult);\n\n");
}

fprintf(fp, "// on-chip memory blocks that store the matrix to be LU factorized\n");
    fprintf(fp, "// store current blocks data\n");
fprintf(fp, "ram currentBlock0 (curWriteByteEn0, clk, curWriteData0, curReadAddr0,
    curWriteAddr0, curWriteEn0, curReadData0);\n");
fprintf(fp, "ram currentBlock1 (curWriteByteEn1, clk, curWriteData1, curReadAddr1,
    curWriteAddr1, curWriteEn1, curReadData1);\n");
fprintf(fp, "// store left blocks data\n");
fprintf(fp, "ram leftBlock0(leftWriteByteEn0, clk, leftWriteData0, leftReadAddr0,
    leftWriteAddr0, leftWriteEn0, leftReadData0);\n\n");
fprintf(fp, "ram leftBlock1(leftWriteByteEn1, clk, leftWriteData1, leftReadAddr1,
    leftWriteAddr1, leftWriteEn1, leftReadData1);\n\n");
fprintf(fp, "// store top block data\n");
fprintf(fp, "temp_row topBlock(clk, topWriteData, topReadAddr, topWriteAddr,
    topWriteEn, topReadDataLU);\n\n");

fprintf(fp, "// processing elements that does the main computation of LU
    factorization\n");
for (i = 0; i < numPE; i++) {
```

```
        fprintf(fp, "mult_add PE%i (clk, multA[%i], multOperand, addA[%i], multResult[%i
            ], addResult[%i]);\n", i, i, i, i, i);
    }
    fprintf(fp, "\n");

    fprintf(fp, "// connect to ports of the left blocks\n");
    fprintf(fp, "assign leftWriteDataLU = (leftWriteSel == 0) ? curReadDataLU :
        rcWriteData;\n");
    if (meminputdelay == 0)
    {
      fprintf(fp, "assign leftWriteData0 = (leftMemSel == 0) ? leftWriteDataMem :
          leftWriteDataLU;\n");
      fprintf(fp, "assign leftWriteAddr0 = (leftMemSel == 0) ? leftWriteAddrMem :
          leftWriteAddrLU;\n");
      fprintf(fp, "assign leftReadAddr0 = leftReadAddrLU;\n");
      fprintf(fp, "assign leftWriteByteEn0 = (leftMemSel == 0) ? leftWriteByteEnMem :
          leftWriteByteEnLU;\n");
      fprintf(fp, "assign leftWriteEn0 = (leftMemSel == 0) ? leftWriteEnMem :
          leftWriteEnLU;\n");
      fprintf(fp, "assign leftWriteData1 = (leftMemSel == 0) ? leftWriteDataLU :
          leftWriteDataMem;\n");
      fprintf(fp, "assign leftWriteAddr1 = (leftMemSel == 0) ? leftWriteAddrLU :
          leftWriteAddrMem;\n");
      fprintf(fp, "assign leftReadAddr1 = leftReadAddrLU;\n");
      fprintf(fp, "assign leftWriteByteEn1 = (leftMemSel == 0) ? leftWriteByteEnLU :
          leftWriteByteEnMem;\n");
      fprintf(fp, "assign leftWriteEn1 = (leftMemSel == 0) ? leftWriteEnLU :
          leftWriteEnMem;\n");
    }
    else
    {
        fprintf(fp, "always @ (posedge clk)\n");
        fprintf(fp, "begin\n");
        fprintf(fp, "    if(leftMemSel == 0)\n");
        fprintf(fp, "    begin\n");
        fprintf(fp, "        leftWriteData0Reg[0] <= leftWriteDataMem;\n");
        fprintf(fp, "        leftWriteAddr0Reg[0] <= leftWriteAddrMem;\n");
        fprintf(fp, "        leftWriteByteEn0Reg[0] <= leftWriteByteEnMem;\n");
        fprintf(fp, "        leftWriteEn0Reg[0] <= leftWriteEnMem;\n");
        fprintf(fp, "        leftWriteData1Reg[0] <= leftWriteDataLU;\n");
        fprintf(fp, "        leftWriteAddr1Reg[0] <= leftWriteAddrLU;\n");
        fprintf(fp, "        leftWriteByteEn1Reg[0] <= leftWriteByteEnLU;\n");
        fprintf(fp, "        leftWriteEn1Reg[0] <= leftWriteEnLU;\n");
        fprintf(fp, "    end\n");
        fprintf(fp, "    else\n");
        fprintf(fp, "    begin\n");
        fprintf(fp, "        leftWriteData0Reg[0] <= leftWriteDataLU;\n");
        fprintf(fp, "        leftWriteAddr0Reg[0] <= leftWriteAddrLU;\n");
        fprintf(fp, "        leftWriteByteEn0Reg[0] <= leftWriteByteEnLU;\n");
        fprintf(fp, "        leftWriteEn0Reg[0] <= leftWriteEnLU;\n");
        fprintf(fp, "        leftWriteData1Reg[0] <= leftWriteDataMem;\n");
        fprintf(fp, "        leftWriteAddr1Reg[0] <= leftWriteAddrMem;\n");
        fprintf(fp, "        leftWriteByteEn1Reg[0] <= leftWriteByteEnMem;\n");
        fprintf(fp, "        leftWriteEn1Reg[0] <= leftWriteEnMem;\n");
        fprintf(fp, "    end\n");
        fprintf(fp, "    leftReadAddr0Reg[0] <= leftReadAddrLU;\n");
        fprintf(fp, "    leftReadAddr1Reg[0] <= leftReadAddrLU;\n");
        for (i = 0; i < meminputdelay-1; i++)
        {
            fprintf(fp, "    leftWriteData0Reg[%i] <= leftWriteData0Reg[%i];\n", i+1, i);
            fprintf(fp, "    leftWriteAddr0Reg[%i] <= leftWriteAddr0Reg[%i];\n", i+1, i);
            fprintf(fp, "    leftReadAddr0Reg[%i] <= leftReadAddr0Reg[%i];\n", i+1, i);
            fprintf(fp, "    leftWriteByteEn0Reg[%i] <= leftWriteByteEn0Reg[%i];\n", i+1,
                i);
            fprintf(fp, "    leftWriteEn0Reg[%i] <= leftWriteEn0Reg[%i];\n", i+1, i);
            fprintf(fp, "    leftWriteData1Reg[%i] <= leftWriteData1Reg[%i];\n", i+1, i);
            fprintf(fp, "    leftWriteAddr1Reg[%i] <= leftWriteAddr1Reg[%i];\n", i+1, i);
            fprintf(fp, "    leftReadAddr1Reg[%i] <= leftReadAddr1Reg[%i];\n", i+1, i);
```

```
            fprintf(fp, "    leftWriteByteEn1Reg[%i] <= leftWriteByteEn1Reg[%i];\n", i+1,
                i);
            fprintf(fp, "    leftWriteEn1Reg[%i] <= leftWriteEn1Reg[%i];\n", i+1, i);
        }
        fprintf(fp, "end\n");
        fprintf(fp, "assign leftWriteData0 = leftWriteData0Reg[%i];\n", i);
        fprintf(fp, "assign leftWriteAddr0 = leftWriteAddr0Reg[%i];\n", i);
        fprintf(fp, "assign leftReadAddr0 = leftReadAddr0Reg[%i];\n", i);
        fprintf(fp, "assign leftWriteByteEn0 = leftWriteByteEn0Reg[%i];\n", i);
        fprintf(fp, "assign leftWriteEn0 = leftWriteEn0Reg[%i];\n", i);
        fprintf(fp, "assign leftWriteData1 = leftWriteData1Reg[%i];\n", i);
        fprintf(fp, "assign leftWriteAddr1 = leftWriteAddr1Reg[%i];\n", i);
        fprintf(fp, "assign leftReadAddr1 = leftReadAddr1Reg[%i];\n", i);
        fprintf(fp, "assign leftWriteByteEn1 = leftWriteByteEn1Reg[%i];\n", i);
        fprintf(fp, "assign leftWriteEn1 = leftWriteEn1Reg[%i];\n", i);
    }
    fprintf(fp, "\n");

    if (memoutputdelay == 0) {
        fprintf(fp, "assign leftReadDataLU = (leftMemSel == 0) ? leftReadData1 :
            leftReadData0;\n");
    }
    else {
        fprintf(fp, "always @ (posedge clk)\n");
        fprintf(fp, "begin\n");
        fprintf(fp, "        leftReadData0Reg[0] <= leftReadData0;\n");
        fprintf(fp, "        leftReadData1Reg[0] <= leftReadData1;\n");
        for (i = 0; i < memoutputdelay-1; i++)
        {
            fprintf(fp, "    leftReadData0Reg[%i] <= leftReadData0Reg[%i];\n", i+1, i);
            fprintf(fp, "    leftReadData1Reg[%i] <= leftReadData1Reg[%i];\n", i+1, i);
        }
        fprintf(fp, "end\n");
        fprintf(fp, "assign leftReadDataLU = (leftMemSel == 0) ? leftReadData1Reg[%i] :
            leftReadData0Reg[%i];\n", i, i);
    }

    fprintf(fp, "// data feed to fp div unit\n");
    fprintf(fp, "always @ (posedge clk)\n");
    fprintf(fp, "begin\n");
    fprintf(fp, "    if (diagEn == 1)\n");
    fprintf(fp, "    begin\n");
    fprintf(fp, "            diag <= topReadData;\n");
    fprintf(fp, "    end\n");
    fprintf(fp, "end\n");

    fprintf(fp, "// one of the inputs to the PE\n");
    fprintf(fp, "always @ (posedge clk)\n");
    fprintf(fp, "begin\n");
    fprintf(fp, "    if (start == 1)\n");
    fprintf(fp, "        multOperand <= 0;\n");
    fprintf(fp, "    else if (MOEn == 1)\n");
    fprintf(fp, "    begin\n");
    fprintf(fp, "        if (MOSel == 0)\n");
    fprintf(fp, "            multOperand <= recResult;\n");
    fprintf(fp, "        else\n");
    fprintf(fp, "            multOperand <= topReadData;\n");
    fprintf(fp, "    end\n");
    fprintf(fp, "end\n\n");

    fprintf(fp, "// connections to top block memory ports\n");
    fprintf(fp, "always @ (*)\n");
    fprintf(fp, "begin\n");
    fprintf(fp, "    if (topSourceSel == 0)\n");
    fprintf(fp, "        case (topWriteSel)\n");
    for (i = 0; i < numPE; i++)
    {
        lowerIdx = (numPE-i-1)*precision;
```

```
        upperIdx = (numPE-i)*precision-1;
        fprintf(fp, "        %i:\n", i);
        fprintf(fp, "            topWriteDataLU = curReadDataLU[%i:%i];\n", upperIdx,
            lowerIdx);
    }
    fprintf(fp, "        default:\n");
    fprintf(fp, "            topWriteDataLU = curReadDataLU[PRECISION-1:0];\n");
    fprintf(fp, "        endcase\n");
    fprintf(fp, "    else\n");
    fprintf(fp, "        case (topWriteSel)\n");
    for (i = 0; i < numPE; i++)
    {
        fprintf(fp, "        %i:\n", i);
        fprintf(fp, "            topWriteDataLU = addResult[%i];\n", numPE-i-1);
    }
    fprintf(fp, "        default:\n");
    fprintf(fp, "            topWriteDataLU = addResult[0];\n");
    fprintf(fp, "        endcase\n");
    fprintf(fp, "end\n\n");

    if (topinputdelay == 0) {
        fprintf(fp, "assign topWriteData = topWriteDataLU;\n");
        fprintf(fp, "assign topReadAddr = topReadAddrLU;\n");
        fprintf(fp, "assign topWriteAddr = topWriteAddrLU;\n");
        fprintf(fp, "assign topWriteEn = topWriteEnLU;\n");
    }
    else {
        fprintf(fp, "always @ (posedge clk)\n");
        fprintf(fp, "begin\n");
        fprintf(fp, "    topWriteDataReg[0] <= topWriteDataLU;\n");
        fprintf(fp, "    topReadAddrReg[0] <= topReadAddrLU;\n");
        fprintf(fp, "    topWriteAddrReg[0] <= topWriteAddrLU;\n");
        fprintf(fp, "    topWriteEnReg[0] <= topWriteEnLU;\n");
        for (i = 0; i < topinputdelay-1; i++) {
            fprintf(fp, "    topWriteDataReg[%i] <= topWriteDataReg[%i];\n", i+1, i);
            fprintf(fp, "    topReadAddrReg[%i] <= topReadAddrReg[%i];\n", i+1, i);
            fprintf(fp, "    topWriteAddrReg[%i] <= topWriteAddrReg[%i];\n", i+1, i);
            fprintf(fp, "    topWriteEnReg[%i] <= topWriteEnReg[%i];\n", i+1, i);
        }
        fprintf(fp, "end\n");
        fprintf(fp, "assign topWriteData = topWriteDataReg[%i];\n", i);
        fprintf(fp, "assign topReadAddr = topReadAddrReg[%i];\n", i);
        fprintf(fp, "assign topWriteAddr = topWriteAddrReg[%i];\n", i);
        fprintf(fp, "assign topWriteEn = topWriteEnReg[%i];\n", i);
    }
    if (topoutputdelay == 0) {
        fprintf(fp, "assign topReadData = topReadDataLU;\n");
    }
    else {
        fprintf(fp, "always @ (posedge clk)\n");
        fprintf(fp, "begin\n");
        fprintf(fp, "    topReadDataReg[0] <= topReadDataLU;\n");
        for (i = 0; i < topoutputdelay-1; i++) {
            fprintf(fp, "    topReadDataReg[%i] <= topReadDataReg[%i];\n", i+1, i);
        }
        fprintf(fp, "end\n");
        fprintf(fp, "assign topReadData = topReadDataReg[%i];\n", i);
    }
    fprintf(fp, "\n");

    fprintf(fp, "// connections to processing element\n");
    for (i = 0; i < numPE; i++)
    {
        lowerIdx = i*precision;
        upperIdx = (i+1)*precision-1;
        fprintf(fp, "assign multA[%i] = leftReadDataLU[%i:%i];\n", i, upperIdx, lowerIdx
            );
    }
```

```
fprintf(fp, "\n");

for (i = 0; i < numPE; i++)
{
    lowerIdx = i*precision;
    upperIdx = (i+1)*precision-1;
    fprintf(fp, "assign addA[%i] = curReadDataLU[%i:%i];\n", i, upperIdx, lowerIdx);
}
fprintf(fp, "\n");

fprintf(fp, "// connections to ports of the current blocks\n");
for (i = 0; i < numPE; i++)
{
    lowerIdx = i*precision;
    upperIdx = (i+1)*precision-1;
    fprintf(fp, "assign rcWriteData[%i:%i] = (curWriteSel == 0) ? multResult[%i] :
        addResult[%i];\n", upperIdx, lowerIdx, i, i);
}
fprintf(fp, "assign curWriteDataLU = rcWriteData;\n");
fprintf(fp, "\n");

if (meminputdelay == 0)
{
  fprintf(fp, "assign curWriteData0 = (curMemSel == 0) ? curWriteDataMem :
      curWriteDataLU;\n");
  fprintf(fp, "assign curWriteAddr0 = (curMemSel == 0) ? curWriteAddrMem :
      curWriteAddrLU;\n");
  fprintf(fp, "assign curReadAddr0 = (curMemSel == 0) ? curReadAddrMem :
      curReadAddrLU;\n");
  fprintf(fp, "assign curWriteByteEn0 = (curMemSel == 0) ? curWriteByteEnMem :
      curWriteByteEnLU;\n");
  fprintf(fp, "assign curWriteEn0 = (curMemSel == 0) ? curWriteEnMem : curWriteEnLU
      ;\n");
  fprintf(fp, "assign curWriteData1 = (curMemSel == 0) ? curWriteDataLU :
      curWriteDataMem;\n");
  fprintf(fp, "assign curWriteAddr1 = (curMemSel == 0) ? curWriteAddrLU :
      curWriteAddrMem;\n");
  fprintf(fp, "assign curReadAddr1 = (curMemSel == 0) ? curReadAddrLU :
      curReadAddrMem;\n");
  fprintf(fp, "assign curWriteByteEn1 = (curMemSel == 0) ? curWriteByteEnLU :
      curWriteByteEnMem;\n");
  fprintf(fp, "assign curWriteEn1 = (curMemSel == 0) ? curWriteEnLU : curWriteEnMem
      ;\n");
}
else
{
    fprintf(fp, "always @ (posedge clk)\n");
    fprintf(fp, "begin\n");
    fprintf(fp, "    if(curMemSel == 0)\n");
    fprintf(fp, "    begin\n");
    fprintf(fp, "        curWriteData0Reg[0] <= curWriteDataMem;\n");
    fprintf(fp, "        curWriteAddr0Reg[0] <= curWriteAddrMem;\n");
    fprintf(fp, "        curReadAddr0Reg[0] <= curReadAddrMem;\n");
    fprintf(fp, "        curWriteByteEn0Reg[0] <= curWriteByteEnMem;\n");
    fprintf(fp, "        curWriteEn0Reg[0] <= curWriteEnMem;\n");
    fprintf(fp, "        curWriteData1Reg[0] <= curWriteDataLU;\n");
    fprintf(fp, "        curWriteAddr1Reg[0] <= curWriteAddrLU;\n");
    fprintf(fp, "        curReadAddr1Reg[0] <= curReadAddrLU;\n");
    fprintf(fp, "        curWriteByteEn1Reg[0] <= curWriteByteEnLU;\n");
    fprintf(fp, "        curWriteEn1Reg[0] <= curWriteEnLU;\n");
    fprintf(fp, "    end\n");
    fprintf(fp, "    else\n");
    fprintf(fp, "    begin\n");
    fprintf(fp, "        curWriteData0Reg[0] <= curWriteDataLU;\n");
    fprintf(fp, "        curWriteAddr0Reg[0] <= curWriteAddrLU;\n");
    fprintf(fp, "        curReadAddr0Reg[0] <= curReadAddrLU;\n");
    fprintf(fp, "        curWriteByteEn0Reg[0] <= curWriteByteEnLU;\n");
    fprintf(fp, "        curWriteEn0Reg[0] <= curWriteEnLU;\n");
```

```
        fprintf(fp, "        curWriteData1Reg[0] <= curWriteDataMem;\n");
        fprintf(fp, "        curWriteAddr1Reg[0] <= curWriteAddrMem;\n");
        fprintf(fp, "        curReadAddr1Reg[0] <= curReadAddrMem;\n");
        fprintf(fp, "        curWriteByteEn1Reg[0] <= curWriteByteEnMem;\n");
        fprintf(fp, "        curWriteEn1Reg[0] <= curWriteEnMem;\n");
        fprintf(fp, "    end\n");
        for (i = 0; i < meminputdelay-1; i++)
        {
            fprintf(fp, "    curWriteData0Reg[%i] <= curWriteData0Reg[%i];\n", i+1, i);
            fprintf(fp, "    curWriteAddr0Reg[%i] <= curWriteAddr0Reg[%i];\n", i+1, i);
            fprintf(fp, "    curReadAddr0Reg[%i] <= curReadAddr0Reg[%i];\n", i+1, i);
            fprintf(fp, "    curWriteByteEn0Reg[%i] <= curWriteByteEn0Reg[%i];\n", i+1, i
                );
            fprintf(fp, "    curWriteEn0Reg[%i] <= curWriteEn0Reg[%i];\n", i+1, i);
            fprintf(fp, "    curWriteData1Reg[%i] <= curWriteData1Reg[%i];\n", i+1, i);
            fprintf(fp, "    curWriteAddr1Reg[%i] <= curWriteAddr1Reg[%i];\n", i+1, i);
            fprintf(fp, "    curReadAddr1Reg[%i] <= curReadAddr1Reg[%i];\n", i+1, i);
            fprintf(fp, "    curWriteByteEn1Reg[%i] <= curWriteByteEn1Reg[%i];\n", i+1, i
                );
            fprintf(fp, "    curWriteEn1Reg[%i] <= curWriteEn1Reg[%i];\n", i+1, i);
        }
        fprintf(fp, "end\n");
        fprintf(fp, "assign curWriteData0 = curWriteData0Reg[%i];\n", i);
        fprintf(fp, "assign curWriteAddr0 = curWriteAddr0Reg[%i];\n", i);
        fprintf(fp, "assign curReadAddr0 = curReadAddr0Reg[%i];\n", i);
        fprintf(fp, "assign curWriteByteEn0 = curWriteByteEn0Reg[%i];\n", i);
        fprintf(fp, "assign curWriteEn0 = curWriteEn0Reg[%i];\n", i);
        fprintf(fp, "assign curWriteData1 = curWriteData1Reg[%i];\n", i);
        fprintf(fp, "assign curWriteAddr1 = curWriteAddr1Reg[%i];\n", i);
        fprintf(fp, "assign curReadAddr1 = curReadAddr1Reg[%i];\n", i);
        fprintf(fp, "assign curWriteByteEn1 = curWriteByteEn1Reg[%i];\n", i);
        fprintf(fp, "assign curWriteEn1 = curWriteEn1Reg[%i];\n", i);
    }
    fprintf(fp, "\n");

    if (memoutputdelay == 0) {
        fprintf(fp, "assign curReadDataMem = (curMemSel == 0) ? curReadData0 :
            curReadData1;\n");
        fprintf(fp, "assign curReadDataLU = (curMemSel == 0) ? curReadData1 :
            curReadData0;\n");
    }
    else {
        fprintf(fp, "always @ (posedge clk)\n");
        fprintf(fp, "begin\n");
        fprintf(fp, "        curReadData0Reg[0] <= curReadData0;\n");
        fprintf(fp, "        curReadData1Reg[0] <= curReadData1;\n");
        for (i = 0; i < memoutputdelay-1; i++)
        {
            fprintf(fp, "    curReadData0Reg[%i] <= curReadData0Reg[%i];\n", i+1, i);
            fprintf(fp, "    curReadData1Reg[%i] <= curReadData1Reg[%i];\n", i+1, i);
        }
        fprintf(fp, "end\n");
        fprintf(fp, "assign curReadDataMem = (curMemSel == 0) ? curReadData0Reg[%i] :
            curReadData1Reg[%i];\n", i, i);
        fprintf(fp, "assign curReadDataLU = (curMemSel == 0) ? curReadData1Reg[%i] :
            curReadData0Reg[%i];\n", i, i);
    }

    fprintf(fp, "endmodule\n");
    fclose(fp);
}
```

A.5 LU Controller Module Function

The source code provided below creates the LU Controller module, which is described in Section 3.3. This module produces the control signals that direct the LU Processing module to perform the computation.

```
#include <stdio.h>
#include <math.h>
#define intlog2(x) (int)ceil(log(x)/log(2))

void genLUC(int numPE, int precision, int mwidth, int mdivk, int intdivlat, int multlat,
     int addlat,
    int curlat, int leftlat, int toplat, int divlat, int ramwidth, int ramsizewidth, int
         topsizewidth)
{
    FILE *fp;
    int PEwidth, mdivkwidth;
    int maxcurleftlat, maxtopleftlat, maxcurtoplat, maxmemlat;
    int divlatwidth;
    int madelay, mcdelay, mdelay, rdelay, cdelay;
    int i;

    maxtopleftlat = (toplat > leftlat) ? toplat : leftlat;
    maxcurleftlat = (curlat > leftlat) ? curlat : leftlat;
    maxcurtoplat = (curlat > toplat) ? curlat : toplat;
    maxmemlat = (maxcurtoplat > leftlat) ? maxcurtoplat : leftlat;
    PEwidth = intlog2(numPE);
    divlat += toplat;
    divlatwidth = intlog2(divlat+1);
    mdivkwidth = intlog2(mdivk+1);
    madelay = multlat+addlat+maxtopleftlat+1;
    mdelay = multlat+maxtopleftlat-curlat;
    mcdelay = multlat+leftlat;
    rdelay = toplat-leftlat;
    cdelay = leftlat-toplat;

    fp = fopen("LUController.v", "w");

    fprintf(fp, "//auto-generated LUController.v\n");
    fprintf(fp, "//control block that creates all the control signals\n");
    fprintf(fp, "//by Wei Zhang\n\n");
    fprintf(fp, "module LUController (clk, start_in, m_in, n_in, loop_in, mode_in, done
        ,\n");
    fprintf(fp, "                     curReadAddr, curWriteAddr, curWriteByteEn,
        curWriteEn, curWriteSel, \n");
    fprintf(fp, "                     leftReadAddr, leftWriteAddr, leftWriteByteEn,
        leftWriteEn,  leftWriteSel,\n");
    fprintf(fp, "                     topReadAddr, topWriteAddr, topWriteEn, topWriteSel,
        topSourceSel, diagSel, diagEn, MOSel, MOEn);\n");
    fprintf(fp, "\n");
    fprintf(fp, "parameter NUMPE = %i, PEWIDTH = %i;\n", numPE, PEwidth);
    fprintf(fp, "parameter BLOCKWIDTH = %i, BLOCKDIVK = %i, BLOCKDIVKWIDTH = %i;\n",
        mwidth, mdivk, mdivkwidth);
    fprintf(fp, "parameter INTDIVLAT = %i;\n", intdivlat);
    fprintf(fp, "parameter CURLAT = %i, LEFTLAT = %i, TOPLAT = %i, DIVLAT = %i;\n",
        curlat, leftlat, toplat, divlat);
    fprintf(fp, "parameter DIVLATWIDTH = %i, MADELAY = %i, MCDELAY = %i, MDELAY = %i;\n"
        , divlatwidth, madelay, mcdelay, mdelay);
    fprintf(fp, "parameter RDELAY = %i, CDELAY = %i;\n", rdelay, cdelay);
    fprintf(fp, "parameter MAXCURLEFTLAT = %i, MAXTOPLEFTLAT = %i, MAXCURTOPLAT = %i;\n"
        , maxcurleftlat, maxtopleftlat, maxcurtoplat);
```

```
fprintf(fp, "parameter RAMNUMBYTES = %i, RAMSIZEWIDTH = %i, TOPSIZEWIDTH = %i;\n",
    ramwidth/8, ramsizewidth, topsizewidth);
fprintf(fp, "\n");
fprintf(fp, "input clk, start_in;\n");
fprintf(fp, "input[BLOCKWIDTH-1:0] m_in, n_in, loop_in;\n");
fprintf(fp, "input[1:0] mode_in;\n");
fprintf(fp, "output done;\n");
fprintf(fp, "\n");
fprintf(fp, "output[RAMNUMBYTES-1:0] curWriteByteEn;\n");
fprintf(fp, "output[RAMSIZEWIDTH-1:0] curWriteAddr, curReadAddr;\n");
fprintf(fp, "output curWriteEn;\n");
fprintf(fp, "\n");
fprintf(fp, "output[RAMNUMBYTES-1:0] leftWriteByteEn;\n");
fprintf(fp, "output[RAMSIZEWIDTH-1:0] leftWriteAddr, leftReadAddr;\n");
fprintf(fp, "output leftWriteEn;\n");
fprintf(fp, "\n");
fprintf(fp, "output[TOPSIZEWIDTH-1:0] topWriteAddr, topReadAddr;\n");
fprintf(fp, "output topWriteEn;\n");
fprintf(fp, "\n");
fprintf(fp, "output leftWriteSel, curWriteSel, topSourceSel, diagEn;\n");
fprintf(fp, "output[PEWIDTH-1:0] diagSel, topWriteSel;\n");
fprintf(fp, "\n");
fprintf(fp, "output MOSel;\n");
fprintf(fp, "output MOEn;\n");
fprintf(fp, "\n");
fprintf(fp, "parameter SETUP = 0, START = 1, FETCH_COL = 2, WAIT_COL = 3, FIND_REC =
     4, MULT_COL = 5, UPDATE_J = 6,\n");
fprintf(fp, "          STORE_MO = 7, MULT_SUB = 8, INCRE_I = 9, WAIT = 10, DONE = 11,
      STORE_DIAG = 12, STORE_DIAG2 = 13, START_FETCH_ROW = 14;\n");
fprintf(fp, "parameter ROW_WAIT = 0, FETCH_ROW = 1, DONE_FETCH_ROW = 2,
    LOAD_ROW_INC_J = 3;\n");
fprintf(fp, "\n");
fprintf(fp, "reg start, startDelay[INTDIVLAT:0];\n");
fprintf(fp, "reg[BLOCKWIDTH-1:0] m, n, stop, stop2, loop;\n");
fprintf(fp, "reg[1:0] mode;\n");
fprintf(fp, "reg[3:0] nextState, currentState;\n");
fprintf(fp, "reg[1:0] nextRowState, currentRowState;\n");
fprintf(fp, "reg startFetchRow, doneFetchRow, loadRow, writeRow;\n");
fprintf(fp, "reg updateCounter;\n");
fprintf(fp, "\n");
fprintf(fp, "reg[BLOCKWIDTH-1:0] i1, j;\n");
fprintf(fp, "reg[TOPSIZEWIDTH-1:0] nextTopIdx, nextTopIdx2, curTopIdx,
    nextTopIdxCounter;\n");
fprintf(fp, "reg[BLOCKDIVKWIDTH-1:0] topIdx, topIdxCounter, mdivk;\n");
fprintf(fp, "reg[RAMSIZEWIDTH-1:0] diagIdx, leftIdx, msIdx;\n");
fprintf(fp, "reg[PEWIDTH-1:0] imodk, i1modk;\n");
fprintf(fp, "reg[RAMSIZEWIDTH-1:0] diagIdxCounter, leftIdxCounter, msIdxCounter,
    readRowCounter, topWriteCounter;\n");
fprintf(fp, "reg[RAMNUMBYTES-1:0] byteEn, i1modkByteEn;\n");
fprintf(fp, "\n");
fprintf(fp, "reg done;\n");
fprintf(fp, "\n");
fprintf(fp, "reg[RAMNUMBYTES-1:0] curWriteByteEn;\n");
fprintf(fp, "reg[RAMSIZEWIDTH-1:0] curWriteAddr, curReadAddr;\n");
fprintf(fp, "reg curWriteEn;\n");
fprintf(fp, "\n");
fprintf(fp, "reg[RAMNUMBYTES-1:0] leftWriteByteEn;\n");
fprintf(fp, "reg[RAMSIZEWIDTH-1:0] leftWriteAddr, leftReadAddr;\n");
fprintf(fp, "reg leftWriteEn;\n");
fprintf(fp, "\n");
fprintf(fp, "reg[TOPSIZEWIDTH-1:0] topWriteAddr, topReadAddr;\n");
fprintf(fp, "reg topWriteEn;\n");
fprintf(fp, "\n");
fprintf(fp, "reg leftWriteSel, curWriteSel, topSourceSel, diagEn;\n");
fprintf(fp, "reg[PEWIDTH-1:0] diagSel, topWriteSel;\n");
fprintf(fp, "\n");
fprintf(fp, "reg MOSel;\n");
fprintf(fp, "reg MOEn;\n");
```

```
fprintf(fp, "\n");
fprintf(fp, "reg[RAMSIZEWIDTH-1:0] counter;\n");
fprintf(fp, "reg[DIVLATWIDTH-1:0] divCounter;\n");
fprintf(fp, "\n");
fprintf(fp, "reg[RAMNUMBYTES-1:0] writeByteEnDelay[MADELAY-1:0];\n");
fprintf(fp, "reg[RAMSIZEWIDTH-1:0] curWriteAddrDelay[MADELAY-1:0], curReadAddrDelay[
    MDELAY-1:0];\n");
fprintf(fp, "reg leftWriteEnDelay[MADELAY-1:0], curWriteEnDelay[MADELAY-1:0];\n");
fprintf(fp, "reg leftWriteSelDelay[CURLAT-1:0], curWriteSelDelay[MCDELAY-1:0];\n");
if (toplat > leftlat) {
    fprintf(fp, "reg[RAMSIZEWIDTH-1:0] leftReadAddrDelay[RDELAY-1:0];\n");
}
else if (toplat < leftlat) {
    fprintf(fp, "reg[TOPSIZEWIDTH-1:0] topReadAddrDelay[CDELAY-1:0];\n");
}
fprintf(fp, "reg[TOPSIZEWIDTH-1:0] topWriteAddrDelay[MADELAY-1:0];\n");
fprintf(fp, "reg topWriteEnDelay[MADELAY-1:0], topSourceSelDelay[CURLAT-1:0];\n");
fprintf(fp, "reg[PEWIDTH-1:0] topWriteSelDelay[MADELAY-1:0];\n");
fprintf(fp, "reg[PEWIDTH-1:0] diagSelDelay[LEFTLAT-1:0];\n");
fprintf(fp, "reg diagEnDelay[TOPLAT-1:0];\n");
fprintf(fp, "reg MOSelDelay[LEFTLAT-1:0], MOEnDelay[TOPLAT-1:0];\n");
fprintf(fp, "reg [RAMSIZEWIDTH-1:0] waitCycles;\n");
fprintf(fp, "\n");
fprintf(fp, "// register store m, n and mdivk value\n");
fprintf(fp, "always @ (posedge clk)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    if (start_in == 1)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        n <= n_in;\n");
fprintf(fp, "        m <= m_in;\n");
fprintf(fp, "        loop <= loop_in;\n");
fprintf(fp, "        mode <= mode_in;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    if (mode[0] == 0 && m == loop)\n");
fprintf(fp, "        stop <= loop;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        stop <= loop+1;\n");
fprintf(fp, "    stop2 <= loop;\n");
fprintf(fp, "    startDelay[0] <= start_in;\n");
for(i = 0; i < intdivlat; i++)
{
    fprintf(fp, "    startDelay[%i] <= startDelay[%i];\n", i+1, i);
}
fprintf(fp, "    start <= startDelay[%i];\n", intdivlat);
fprintf(fp, "    mdivk <= (m+NUMPE-1)/NUMPE;\n");
fprintf(fp, "end\n\n");
fprintf(fp, "// registers that store values that are used in FSM, dependent on i and
    /or j\n");
fprintf(fp, "always @ (posedge clk)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    if (start == 1)\n");
fprintf(fp, "        topIdx <= 0; //offset1divk;\n");
fprintf(fp, "    else if (currentState == INCRE_I && i1modk == NUMPE-1 && mode[0] ==
    0)\n");
fprintf(fp, "        topIdx <= topIdx + 1;\n");
fprintf(fp, "        \n");
fprintf(fp, "    if (start == 1)\n");
fprintf(fp, "        diagIdx <= 0; //offsetdivk;\n");
fprintf(fp, "    else if (currentState == STORE_DIAG && mode == 1)\n");
fprintf(fp, "        diagIdx <= BLOCKDIVK; //uppermdivk;\n");
fprintf(fp, "    else if (currentState == INCRE_I)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if ((imodk == NUMPE-1 && mode == 0) || (i1modk == NUMPE-1 &&
    mode == 1))\n");
fprintf(fp, "            diagIdx <= diagIdx + BLOCKDIVK + 1;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            diagIdx <= diagIdx + BLOCKDIVK;\n");
fprintf(fp, "    end\n");
```

```
fprintf(fp, "    \n");
fprintf(fp, "    if (start == 1)\n");
fprintf(fp, "        leftIdx <= 0;\n");
fprintf(fp, "    else if (currentState == INCRE_I)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (i1modk == NUMPE-1 && mode[0] == 0)\n");
fprintf(fp, "            leftIdx <= leftIdx + BLOCKDIVK + 1;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            leftIdx <= leftIdx + BLOCKDIVK;\n");
fprintf(fp, "    end\n");
fprintf(fp, "\n");
fprintf(fp, "    if (start == 1)\n");
fprintf(fp, "        msIdx <= 0;\n");
fprintf(fp, "    else if (currentState == UPDATE_J)\n");
fprintf(fp, "        if (mode[1] == 0)\n");
fprintf(fp, "            msIdx <= leftIdx + BLOCKDIVK;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            msIdx <= topIdx;\n");
fprintf(fp, "    else if (nextRowState == LOAD_ROW_INC_J)\n");
fprintf(fp, "        msIdx <= msIdx + BLOCKDIVK;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (start == 1)\n");
fprintf(fp, "        imodk <= 0; //offsetmodk;\n");
fprintf(fp, "    else if (currentState == INCRE_I)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (imodk == NUMPE-1)\n");
fprintf(fp, "            imodk <= 0;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            imodk <= imodk + 1;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    \n");
fprintf(fp, "    if (start == 1)\n");
fprintf(fp, "        i1modk <= 1; //offset1modk;\n");
fprintf(fp, "    else if (currentState == INCRE_I)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (i1modk == NUMPE-1)\n");
fprintf(fp, "            i1modk <= 0;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            i1modk <= i1modk + 1;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    \n");
fprintf(fp, "    if (start == 1)\n");
fprintf(fp, "            nextTopIdx <= 0;\n");
fprintf(fp, "    else if (currentState == INCRE_I)\n");
fprintf(fp, "        if (mode[1] == 0)\n");
fprintf(fp, "            nextTopIdx <= nextTopIdx + n + 1;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            nextTopIdx <= nextTopIdx + n;\n");
fprintf(fp, " nextTopIdx2 <= nextTopIdx + n + 1;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (start == 1)\n");
fprintf(fp, "        curTopIdx <= 1;\n");
fprintf(fp, "    else if (currentState == UPDATE_J)\n");
fprintf(fp, "    if (mode[1] == 0)\n");
fprintf(fp, "          curTopIdx <= nextTopIdx+1;\n");
fprintf(fp, "    else\n");
fprintf(fp, "          curTopIdx <= nextTopIdx;\n");
fprintf(fp, "    else if (nextRowState == LOAD_ROW_INC_J)\n");
fprintf(fp, "        curTopIdx <= curTopIdx + 1;\n");
fprintf(fp, "    \n");
fprintf(fp, "    if (start == 1)\n");
fprintf(fp, "       i1 <= 1;\n");
fprintf(fp, "    else if (currentState == INCRE_I)\n");
fprintf(fp, "       i1 <= i1 + 1;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (start == 1)\n");
fprintf(fp, "        j <= 0;\n");
fprintf(fp, "    else if (currentState == UPDATE_J)\n");
```

```
fprintf(fp, "        if (mode[1] == 0)\n");
fprintf(fp, "            j <= i1;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            j <= 0;\n");
fprintf(fp, "    else if (currentRowState == LOAD_ROW_INC_J)\n");
fprintf(fp, "        j <= j + 1;\n");
fprintf(fp, "\n");
fprintf(fp, "// compute cycles of delay in FSM\n");
fprintf(fp, "    if (currentState == STORE_MO)\n");
fprintf(fp, "        waitCycles <= MADELAY-1;\n");
fprintf(fp, "    else if (currentState == INCRE_I)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (i1 == stop-1)\n");
fprintf(fp, "            if (mode[1] == 1)\n");
fprintf(fp, "                waitCycles <= MADELAY-1 + MAXCURTOPLAT - 3;\n");
fprintf(fp, "            else\n");
fprintf(fp, "                waitCycles <= waitCycles + MAXCURLEFTLAT - 2;\n");
fprintf(fp, "        else if (mode == 1 && waitCycles < MADELAY-1 - (MCDELAY-1) - 4)\
    n");
fprintf(fp, "            waitCycles <= MADELAY-1 - (MCDELAY-1) - 4;\n");
fprintf(fp, "        else if (mode == 2 && i1modk == NUMPE-1)\n");
fprintf(fp, "            waitCycles <= MADELAY-1 + MAXCURTOPLAT - 3;\n");
if (toplat > leftlat) {
    fprintf(fp, "        else if (mode == 0)\n");
    fprintf(fp, "            waitCycles <= waitCycles + TOPLAT ;\n");
}
fprintf(fp, "    end\n");
fprintf(fp, "    else if (waitCycles > 0)\n");
fprintf(fp, "        waitCycles <= waitCycles - 1;\n");
fprintf(fp, "\n");
fprintf(fp, "end\n");
fprintf(fp, "\n");
fprintf(fp, "// determining next state of main FSM\n");
fprintf(fp, "always @ (*)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    case (currentState)\n");
fprintf(fp, "    SETUP:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (start == 1)\n");
fprintf(fp, "            nextState = START;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            nextState = SETUP;\n");
fprintf(fp, "        updateCounter = 1;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    START:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (mode == 0)\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            if (m == 1 && n == 1)\n");
fprintf(fp, "                nextState = DONE;\n");
fprintf(fp, "            else\n");
fprintf(fp, "                nextState = FETCH_COL;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        else if (mode == 1)\n");
fprintf(fp, "            nextState = STORE_DIAG;\n");
fprintf(fp, "        else if (mode == 2)\n");
fprintf(fp, "            nextState = START_FETCH_ROW;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            nextState = UPDATE_J;\n");
fprintf(fp, "        updateCounter = 1;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    START_FETCH_ROW:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (counter == CURLAT+TOPLAT-1)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "          if (mode == 0)\n");
fprintf(fp, "              nextState = STORE_DIAG;\n");
fprintf(fp, "          else\n");
```

```
fprintf(fp, "              nextState = UPDATE_J;\n");
fprintf(fp, "     end\n");
fprintf(fp, "       else\n");
fprintf(fp, "           nextState = START_FETCH_ROW;\n");
fprintf(fp, "       updateCounter = 0;\n");
fprintf(fp, "   end\n");
fprintf(fp, "   FETCH_COL:\n");
fprintf(fp, "       if (counter >= mdivk-1)\n");
fprintf(fp, "       begin\n");
fprintf(fp, "           if (mode == 0 && counter < CURLAT)\n");
fprintf(fp, "           begin\n");
fprintf(fp, "               nextState = WAIT_COL;\n");
fprintf(fp, "               updateCounter = 0;\n");
fprintf(fp, "           end\n");
fprintf(fp, "           else\n");
fprintf(fp, "           begin\n");
fprintf(fp, "               if (mode == 0)\n");
fprintf(fp, "                   nextState = START_FETCH_ROW;\n");
fprintf(fp, "               else\n");
fprintf(fp, "                   nextState = FIND_REC;\n");
fprintf(fp, "               updateCounter = 1;\n");
fprintf(fp, "           end\n");
fprintf(fp, "       end\n");
fprintf(fp, "       else\n");
fprintf(fp, "       begin\n");
fprintf(fp, "           nextState = FETCH_COL;\n");
fprintf(fp, "           updateCounter = 0;\n");
fprintf(fp, "       end\n");
fprintf(fp, "   WAIT_COL:\n");
fprintf(fp, "       if (counter >= CURLAT)\n");
fprintf(fp, "       begin\n");
fprintf(fp, "           if (mode == 0)\n");
fprintf(fp, "               nextState = START_FETCH_ROW;\n");
fprintf(fp, "           else\n");
fprintf(fp, "               nextState = FIND_REC;\n");
fprintf(fp, "           updateCounter = 1;\n");
fprintf(fp, "       end\n");
fprintf(fp, "       else\n");
fprintf(fp, "       begin\n");
fprintf(fp, "           nextState = WAIT_COL;\n");
fprintf(fp, "           updateCounter = 0;\n");
fprintf(fp, "       end\n");
fprintf(fp, "   STORE_DIAG:\n");
fprintf(fp, "   begin\n");
fprintf(fp, "       if (mode == 0)\n");
fprintf(fp, "           nextState =  FIND_REC;\n");
fprintf(fp, "       else\n");
fprintf(fp, "           nextState =  FETCH_COL;\n");
fprintf(fp, "       updateCounter = 1;\n");
fprintf(fp, "   end\n");
fprintf(fp, "   FIND_REC:\n");
fprintf(fp, "       if (divCounter == DIVLAT)\n");
fprintf(fp, "       begin\n");
fprintf(fp, "           if (mode == 0)\n");
fprintf(fp, "               nextState = MULT_COL;\n");
fprintf(fp, "           else\n");
fprintf(fp, "               nextState = STORE_DIAG2;\n");
fprintf(fp, "           updateCounter = 1;\n");
fprintf(fp, "       end\n");
fprintf(fp, "       else\n");
fprintf(fp, "       begin\n");
fprintf(fp, "           nextState = FIND_REC;\n");
fprintf(fp, "           updateCounter = 0;\n");
fprintf(fp, "       end\n");
fprintf(fp, "   STORE_DIAG2:\n");
fprintf(fp, "   begin\n");
fprintf(fp, "       nextState =  MULT_COL;\n");
fprintf(fp, "       updateCounter = 1;\n");
```

```
fprintf(fp, "    end\n");
fprintf(fp, "    MULT_COL:\n");
fprintf(fp, "        if (topIdxCounter == mdivk-1)\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            nextState = UPDATE_J;\n");
fprintf(fp, "            updateCounter = 0;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        else\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            nextState = MULT_COL;\n");
fprintf(fp, "            updateCounter = 0;\n");
fprintf(fp, "        end\n");
fprintf(fp, "    UPDATE_J:\n");
fprintf(fp, "        if ((mode[1] == 1 || counter >= MCDELAY-1) && doneFetchRow == 1)
    \n");
fprintf(fp, "        begin\n");
fprintf(fp, "            nextState = STORE_MO;\n");
fprintf(fp, "            updateCounter = 1;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        else\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            nextState = UPDATE_J;\n");
fprintf(fp, "            updateCounter = 0;\n");
fprintf(fp, "        end\n");
fprintf(fp, "    STORE_MO:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        if (j == stop2)\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            if (counter == mdivk-1+MAXCURLEFTLAT-2)\n");
fprintf(fp, "                nextState = DONE;\n");
fprintf(fp, "            else\n");
fprintf(fp, "                nextState = STORE_MO;\n");
fprintf(fp, "            updateCounter = 0;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        else\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            nextState =  MULT_SUB;\n");
fprintf(fp, "            updateCounter = 1;\n");
fprintf(fp, "        end\n");
fprintf(fp, "    end\n");
fprintf(fp, "    MULT_SUB:\n");
fprintf(fp, "        if (topIdxCounter == mdivk-1)\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            if (j == n-1)\n");
fprintf(fp, "                nextState = INCRE_I;\n");
fprintf(fp, "            else\n");
fprintf(fp, "                nextState = MULT_SUB;\n");
fprintf(fp, "            updateCounter = 1;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        else\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            nextState = MULT_SUB;\n");
fprintf(fp, "            updateCounter = 0;\n");
fprintf(fp, "        end\n");
fprintf(fp, "    INCRE_I:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        nextState = WAIT;\n");
fprintf(fp, "        updateCounter = 1;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    WAIT:\n");
fprintf(fp, "        if (waitCycles == 0)\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            if (i1 == stop)\n");
fprintf(fp, "                nextState = DONE;\n");
fprintf(fp, "            else if (mode == 0)\n");
fprintf(fp, "                nextState = STORE_DIAG;\n");
fprintf(fp, "            else if (mode == 1)\n");
fprintf(fp, "                nextState = FIND_REC;\n");
```

```
fprintf(fp, "            else\n");
fprintf(fp, "                nextState = UPDATE_J;\n");
fprintf(fp, "            updateCounter = 1;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        else\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            nextState = WAIT;\n");
fprintf(fp, "            updateCounter = 0;\n");
fprintf(fp, "        end\n");
fprintf(fp, "    DONE:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        nextState = DONE;\n");
fprintf(fp, "        updateCounter = 0;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    default:\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        nextState = SETUP;\n");
fprintf(fp, "        updateCounter = 1;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    endcase\n");
fprintf(fp, "end\n");
fprintf(fp, "\n");
fprintf(fp, "always @ (*)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    if (currentRowState == DONE_FETCH_ROW)\n");
fprintf(fp, "        doneFetchRow = 1;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        doneFetchRow = 0;\n");
fprintf(fp, "        if((nextState == START_FETCH_ROW && currentState !=
    START_FETCH_ROW && i1 == 1))\n");
fprintf(fp, "        startFetchRow = 1;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        startFetchRow = 0;\n");
fprintf(fp, "    if (currentState == MULT_SUB && topIdxCounter+2 == mdivk)\n");
fprintf(fp, "        loadRow = 1;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        loadRow = 0;\n");
fprintf(fp, "    writeRow = (msIdxCounter == readRowCounter)&&(currentState==MULT_SUB
    )&&(j!=n)&&(mode[0] == 0);\n");
fprintf(fp, "end\n");
fprintf(fp, "\n");
fprintf(fp, "// second FSM that controls the control signals to temp_top block\n");
fprintf(fp, "always @ (*)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    case (currentRowState)\n");
fprintf(fp, "    FETCH_ROW:\n");
fprintf(fp, "        if (nextTopIdxCounter == n-1)\n");
fprintf(fp, "            nextRowState = DONE_FETCH_ROW;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            nextRowState = FETCH_ROW;\n");
fprintf(fp, "    DONE_FETCH_ROW:\n");
fprintf(fp, "        if (startFetchRow == 1)\n");
fprintf(fp, "            nextRowState = FETCH_ROW;\n");
fprintf(fp, "        else if (loadRow == 1 || (topIdx+1 == mdivk && nextState ==
    MULT_SUB))\n");
fprintf(fp, "            nextRowState = LOAD_ROW_INC_J;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            nextRowState = DONE_FETCH_ROW;\n");
fprintf(fp, "    LOAD_ROW_INC_J:\n");
fprintf(fp, "        if (topIdx+1 == mdivk && nextState == MULT_SUB)\n");
fprintf(fp, "            nextRowState = LOAD_ROW_INC_J;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            nextRowState = DONE_FETCH_ROW;\n");
fprintf(fp, "    default:\n");
fprintf(fp, "        nextRowState = DONE_FETCH_ROW;\n");
fprintf(fp, "    endcase\n");
fprintf(fp, "end\n");
fprintf(fp, "\n");
```

```
fprintf(fp, "// address counters\n");
fprintf(fp, "always @ (posedge clk)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    if (updateCounter == 1 || currentRowState == LOAD_ROW_INC_J)\n");
fprintf(fp, "        topIdxCounter <= topIdx;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        topIdxCounter <= topIdxCounter + 1;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (updateCounter == 1)\n");
fprintf(fp, "        diagIdxCounter <= diagIdx;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        diagIdxCounter <= diagIdxCounter + 1;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (updateCounter == 1 || currentRowState == LOAD_ROW_INC_J)\n");
fprintf(fp, "        msIdxCounter <= msIdx;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        msIdxCounter <= msIdxCounter + 1;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (updateCounter == 1 || currentRowState == LOAD_ROW_INC_J)\n");
fprintf(fp, "        leftIdxCounter <= leftIdx;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        leftIdxCounter <= leftIdxCounter + 1;\n");
fprintf(fp, "    \n");
fprintf(fp, "    if (currentState == FETCH_COL || currentState == STORE_MO)\n");
fprintf(fp, "        topWriteCounter <= i1;\n");
fprintf(fp, "    else if (writeRow == 1 || currentRowState == FETCH_ROW)\n");
fprintf(fp, "        topWriteCounter <= topWriteCounter + 1;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (currentState == START)\n");
fprintf(fp, "        nextTopIdxCounter <= nextTopIdx;\n");
fprintf(fp, "    else if (currentState == STORE_MO)\n");
fprintf(fp, "        if (mode[1] == 0)\n");
fprintf(fp, "            nextTopIdxCounter <= nextTopIdx + n + 1;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            nextTopIdxCounter <= nextTopIdx + n;\n");
fprintf(fp, "    else if (writeRow == 1 || currentRowState == FETCH_ROW)\n");
fprintf(fp, "        nextTopIdxCounter <= nextTopIdxCounter + 1;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (currentState == START)\n");
fprintf(fp, "            readRowCounter <= 0; //offsetdivk;\n");
fprintf(fp, "    else if (currentState == STORE_MO)\n");
fprintf(fp, "        if (mode[1] == 0)\n");
fprintf(fp, "            readRowCounter <= leftIdx + BLOCKDIVK;\n");
fprintf(fp, "        else\n");
fprintf(fp, "            readRowCounter <= topIdx;\n");
fprintf(fp, "    else if (writeRow == 1 || currentRowState == FETCH_ROW)\n");
fprintf(fp, "       readRowCounter <= readRowCounter + BLOCKDIVK;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (updateCounter == 1)\n");
fprintf(fp, "        counter <= 0;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        counter <= counter + 1;\n");
fprintf(fp, "\n");
fprintf(fp, "    if (currentState == STORE_DIAG || currentState == STORE_DIAG2)\n");
fprintf(fp, "        divCounter <= 0;\n");
fprintf(fp, "    else if (divCounter < DIVLAT)\n");
fprintf(fp, "        divCounter <= divCounter + 1;\n");
fprintf(fp, "\n");
if (precision == 64)
{
    fprintf(fp, "    i1modkByteEn <= ~(%i'h0) >> (i1modk*8);\n", ramwidth/8);
}
else
{
    fprintf(fp, "    i1modkByteEn <= ~(%i'h0) >> (i1modk*4);\n", ramwidth/8);
}
fprintf(fp, "end\n");
fprintf(fp, "\n");
```

```
fprintf(fp, "// compute Byte Enable\n");
fprintf(fp, "always @ (posedge clk)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    if ((nextState == MULT_COL && currentState != MULT_COL) || (
    currentState == STORE_MO) || currentRowState == LOAD_ROW_INC_J)\n");
fprintf(fp, "        byteEn <= i1modkByteEn;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        byteEn <= ~0;\n");
fprintf(fp, "end\n");
fprintf(fp, "\n");
fprintf(fp, "// update FSM state register\n");
fprintf(fp, "always @ (posedge clk)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    if (start_in == 1)\n");
fprintf(fp, "        currentState <= SETUP;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        currentState <= nextState;\n");
fprintf(fp, "    if (start == 1)\n");
fprintf(fp, "        currentRowState <= DONE_FETCH_ROW;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        currentRowState <= nextRowState;\n");
fprintf(fp, "end\n");
fprintf(fp, "\n");
fprintf(fp, "// delay register for control signals\n");
fprintf(fp, "// control signals are delayed to match latency of operations and/or
    memory access\n");
fprintf(fp, "always @ (posedge clk)\n");
fprintf(fp, "begin\n");

for (i = 0; i < mdelay-1; i++) {
    fprintf(fp, "    curReadAddrDelay[%i] <= curReadAddrDelay[%i];\n", i, i+1);
}
fprintf(fp, "    curReadAddrDelay[%i] <= msIdxCounter;\n", i);
fprintf(fp, "    \n");

for (i = 0; i < curlat-1; i++) {
    fprintf(fp, "    curWriteAddrDelay[%i] <= curWriteAddrDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentState == FETCH_COL)\n");
fprintf(fp, "        curWriteAddrDelay[%i] <= diagIdxCounter;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        curWriteAddrDelay[%i] <= curWriteAddrDelay[%i];\n", i, i+1);
for (i = curlat; i < mcdelay-1; i++) {
    fprintf(fp, "    curWriteAddrDelay[%i] <= curWriteAddrDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentState == MULT_COL)\n");
fprintf(fp, "        curWriteAddrDelay[%i] <= leftIdxCounter;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        curWriteAddrDelay[%i] <= curWriteAddrDelay[%i];\n", i, i+1);
for (i = mcdelay; i < madelay-1; i++) {
    fprintf(fp, "    curWriteAddrDelay[%i] <= curWriteAddrDelay[%i];\n", i, i+1);
}
fprintf(fp, "    curWriteAddrDelay[%i] <= msIdxCounter;\n", i);
fprintf(fp, "    \n");

for (i = 0; i < curlat-1; i++) {
    fprintf(fp, "    writeByteEnDelay[%i] <= writeByteEnDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (mode[0] == 1)\n");
fprintf(fp, "        writeByteEnDelay[%i] <= ~0;\n", i);
fprintf(fp, "    else if (currentState == FETCH_COL)\n");
fprintf(fp, "        writeByteEnDelay[%i] <= byteEn;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        writeByteEnDelay[%i] <= writeByteEnDelay[%i];\n", i, i+1);
for (i = curlat; i < mcdelay-1; i++) {
    fprintf(fp, "    writeByteEnDelay[%i] <= writeByteEnDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentState == MULT_COL)\n");
```

```
fprintf(fp, "        writeByteEnDelay[%i] <= byteEn;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        writeByteEnDelay[%i] <= writeByteEnDelay[%i];\n", i, i+1);
for (i = mcdelay; i < madelay-1; i++) {
    fprintf(fp, "    writeByteEnDelay[%i] <= writeByteEnDelay[%i];\n", i, i+1);
}
fprintf(fp, "    writeByteEnDelay[%i] <= byteEn;\n", i);
fprintf(fp, "    \n");

for (i = 0; i < mcdelay-1; i++) {
    fprintf(fp, "    curWriteSelDelay[%i] <= curWriteSelDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentState == MULT_COL)\n");
fprintf(fp, "        curWriteSelDelay[%i] <= 0;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        curWriteSelDelay[%i] <= 1;\n", i);
fprintf(fp, "\n");

for (i = 0; i < mcdelay-1; i++) {
    fprintf(fp, "    curWriteEnDelay[%i] <= curWriteEnDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentState == MULT_COL)\n");
fprintf(fp, "        curWriteEnDelay[%i] <= 1;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        curWriteEnDelay[%i] <= curWriteEnDelay[%i];\n", i, i+1);
for (i = mcdelay; i < madelay-1; i++) {
    fprintf(fp, "    curWriteEnDelay[%i] <= curWriteEnDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentState == MULT_SUB)\n");
fprintf(fp, "        curWriteEnDelay[%i] <= 1;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        curWriteEnDelay[%i] <= 0;\n", i);
fprintf(fp, "\n");

for (i = 0; i < curlat-1; i++) {
    fprintf(fp, "    leftWriteSelDelay[%i] <= leftWriteSelDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentState == FETCH_COL)\n");
fprintf(fp, "        leftWriteSelDelay[%i] <= 0;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        leftWriteSelDelay[%i] <= 1;\n", i);
fprintf(fp, "\n");

for (i = 0; i < curlat-1; i++) {
    fprintf(fp, "    leftWriteEnDelay[%i] <= leftWriteEnDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentState == FETCH_COL)\n");
fprintf(fp, "        leftWriteEnDelay[%i] <= 1;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        leftWriteEnDelay[%i] <= leftWriteEnDelay[%i];\n", i, i+1);
for (i = curlat; i < mcdelay-1; i++) {
    fprintf(fp, "    leftWriteEnDelay[%i] <= leftWriteEnDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentState == MULT_COL)\n");
fprintf(fp, "        leftWriteEnDelay[%i] <= 1;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        leftWriteEnDelay[%i] <= leftWriteEnDelay[%i];\n", i, i+1);
for (i = mcdelay; i < madelay-1; i++) {
    fprintf(fp, "    leftWriteEnDelay[%i] <= leftWriteEnDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentState == MULT_SUB && (mode == 0 || (mode == 1 && j == i1)
    ))\n");
fprintf(fp, "        leftWriteEnDelay[%i] <= 1;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        leftWriteEnDelay[%i] <= 0;\n", i);
fprintf(fp, "\n");

for (i = 0; i < curlat-1; i++) {
```

```
    fprintf(fp, "    topWriteAddrDelay[%i] <= topWriteAddrDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentRowState == FETCH_ROW)\n");
fprintf(fp, "        topWriteAddrDelay[%i] <= nextTopIdxCounter;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        topWriteAddrDelay[%i] <=  topWriteAddrDelay[%i];\n", i, i+1);
for (i = curlat; i < madelay-1; i++) {
    fprintf(fp, "    topWriteAddrDelay[%i] <= topWriteAddrDelay[%i];\n", i, i+1);
}
fprintf(fp, "        topWriteAddrDelay[%i] <= nextTopIdxCounter;\n", i);
fprintf(fp, "\n");

for (i = 0; i < curlat-1; i++) {
    fprintf(fp, "    topWriteEnDelay[%i] <= topWriteEnDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentRowState == FETCH_ROW)\n");
fprintf(fp, "        topWriteEnDelay[%i] <= 1;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        topWriteEnDelay[%i] <=  topWriteEnDelay[%i];\n", i, i+1);
for (i = curlat; i < madelay-1; i++) {
    fprintf(fp, "    topWriteEnDelay[%i] <= topWriteEnDelay[%i];\n", i, i+1);
}
fprintf(fp, "    topWriteEnDelay[%i] <= writeRow;\n", i);
fprintf(fp, "\n");

for (i = 0; i < curlat-1; i++) {
    fprintf(fp, "    topWriteSelDelay[%i] <= topWriteSelDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (currentRowState == FETCH_ROW || currentState == UPDATE_J && i1
    == 1)\n");
fprintf(fp, "        topWriteSelDelay[%i] <= imodk;\n", i);
fprintf(fp, "    else\n");
fprintf(fp, "        topWriteSelDelay[%i] <=  topWriteSelDelay[%i];\n", i, i+1);
for (i = curlat; i < madelay-1; i++) {
    fprintf(fp, "    topWriteSelDelay[%i] <= topWriteSelDelay[%i];\n", i, i+1);
}
fprintf(fp, "    topWriteSelDelay[%i] <= i1modk;\n", i);
fprintf(fp, "\n");

for (i = 0; i < curlat-1; i++) {
    fprintf(fp, "    topSourceSelDelay[%i] <= topSourceSelDelay[%i];\n", i, i+1);
}
fprintf(fp, "    if (start == 1)\n");
fprintf(fp, "        topSourceSelDelay[%i] <= 0;\n", i);
fprintf(fp, "    else if (currentState == STORE_MO)\n");
fprintf(fp, "        topSourceSelDelay[%i] <= 1;\n", i);
fprintf(fp, "\n");

if (leftlat > toplat) {
    for (i = 0; i < cdelay-1; i++) {
        fprintf(fp, "    topReadAddrDelay[%i] <= topReadAddrDelay[%i];\n", i, i+1);
    }
    fprintf(fp, "    topReadAddrDelay[%i] <= curTopIdx;\n", i);
    fprintf(fp, "\n");
}

if (toplat > leftlat) {
    for (i = 0; i < rdelay-1; i++) {
        fprintf(fp, "    leftReadAddrDelay[%i] <= leftReadAddrDelay[%i];\n", i, i+1);
    }
    fprintf(fp, "    leftReadAddrDelay[%i] <= leftIdxCounter;\n", i);
    fprintf(fp, "\n");
}
fprintf(fp, "\n");

for (i = 0; i < toplat-1; i++) {
    fprintf(fp, "    diagEnDelay[%i] <= diagEnDelay[%i];\n", i, i+1);
}
```

```
fprintf(fp, "    diagEnDelay[%i] <= (currentState == STORE_DIAG || currentState ==
    STORE_DIAG2);\n", i);
fprintf(fp, "\n");

for (i = 0; i < toplat-1; i++) {
    fprintf(fp, "    MOEnDelay[%i] <= MOEnDelay[%i];\n", i, i+1);
}
    fprintf(fp, "    if (currentState == STORE_MO || currentRowState ==
        LOAD_ROW_INC_J)\n");
    fprintf(fp, "        MOEnDelay[%i] <= 1;\n", i);
    fprintf(fp, "    else\n");
    fprintf(fp, "        MOEnDelay[%i] <= 0;\n", i);

fprintf(fp, "end\n");
fprintf(fp, "\n");

fprintf(fp, "// output contorl signals\n");
fprintf(fp, "always @ (*)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    if (currentState == FETCH_COL)\n");
fprintf(fp, "        curReadAddr <= diagIdxCounter;\n");
fprintf(fp, "    else if (currentRowState == FETCH_ROW)\n");
fprintf(fp, "       curReadAddr <= readRowCounter;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        curReadAddr <= curReadAddrDelay[0];\n");
fprintf(fp, "    curWriteAddr <= curWriteAddrDelay[0];\n");
fprintf(fp, "    curWriteByteEn <= writeByteEnDelay[0];\n");
fprintf(fp, "    curWriteSel <= curWriteSelDelay[0];\n");
fprintf(fp, "    curWriteEn <= curWriteEnDelay[0];\n");
fprintf(fp, "\n");
if (toplat > leftlat) {
    fprintf(fp, "    if (currentState == MULT_COL)\n");
    fprintf(fp, "        leftReadAddr <= leftIdxCounter;\n");
    fprintf(fp, "    else\n");
    fprintf(fp, "        leftReadAddr <= leftReadAddrDelay[0];\n");
}
else {
    fprintf(fp, "    leftReadAddr <= leftIdxCounter;\n");
}
fprintf(fp, "    leftWriteAddr <= curWriteAddrDelay[0];\n");
fprintf(fp, "    leftWriteByteEn <= writeByteEnDelay[0];\n");
fprintf(fp, "    leftWriteSel <= leftWriteSelDelay[0];\n");
fprintf(fp, "    leftWriteEn <= leftWriteEnDelay[0];\n");
fprintf(fp, "\n");

fprintf(fp, "    if (currentState == STORE_DIAG)\n");
fprintf(fp, "        topReadAddr <= nextTopIdx;\n");
fprintf(fp, "else if (currentState == STORE_DIAG2)\n");
fprintf(fp, "    topReadAddr <= nextTopIdx2;\n");
fprintf(fp, "    else\n");
if (leftlat > toplat) {
    fprintf(fp, "    topReadAddr <= topReadAddrDelay[0];\n");
}
else {
    fprintf(fp, "    topReadAddr <= curTopIdx;\n");
}
fprintf(fp, "    topWriteAddr <= topWriteAddrDelay[0];\n");
fprintf(fp, "    topWriteEn <= topWriteEnDelay[0];\n");
fprintf(fp, "    topWriteSel <= topWriteSelDelay[0];\n");
fprintf(fp, "    topSourceSel <= topSourceSelDelay[0];\n");
fprintf(fp, "\n");
fprintf(fp, "    MOSel <= ~(currentState == FIND_REC);\n");
fprintf(fp, "if (currentState == FIND_REC)\n");
fprintf(fp, "        MOEn <= 1;\n");
fprintf(fp, "    else\n");
fprintf(fp, "        MOEn <= MOEnDelay[0];\n");
fprintf(fp, "\n");
```

```
    fprintf(fp, "    diagSel <= diagSelDelay[0];\n");
    fprintf(fp, "    diagEn <= diagEnDelay[0];\n");
    fprintf(fp, "\n");
    fprintf(fp, "    if (currentState == DONE)\n");
    fprintf(fp, "        done <= 1;\n");
    fprintf(fp, "    else\n");
    fprintf(fp, "        done <= 0;\n");
    fprintf(fp, "end\n");
    fprintf(fp, "\n");
    fprintf(fp, "endmodule\n");
    fclose(fp);
}
```

A.6 Data Transfer Unit Module Function

The source code provided below creates the Data Transfer Unit module, which is described in Section 3.4. This module handles communication between the off-chip memory controller and the Marshalling Controller module in order to perform data marshalling.

```
#include <stdio.h>
#include <math.h>
#define intlog2(x) (int)ceil(log(x)/log(2))

void genDTU(int burstlen, int datawidth, int memconwidth, int ddrsizewidth, int fifosize
    ,
    int ramwidth, int ramsizewidth, int ratio, int ramlat, int transfersize)
{
    FILE *fp;
    int i, j, k;
    int burstwidth, fifowidth, countwidth;

    burstwidth = intlog2(burstlen+1);
    countwidth = intlog2(ratio+burstlen);
    fifowidth = intlog2(fifosize);

    fp = fopen("DataTransferUnit.v", "w");

    fprintf(fp, "\n");
    fprintf(fp, "module DataTransferUnit (clk, phy_clk, dtu_write_req, dtu_read_req,
        dtu_mem_addr, dtu_ram_addr, dtu_size, dtu_ack, dtu_done,\n");
    fprintf(fp, "        ram_read_addr, ram_read_data, ram_write_byte_en, ram_write_data,
         ram_write_addr, ram_write_en,\n");
    fprintf(fp, "        mem_rdata, mem_rdata_valid, mem_ready, mem_wdata_req, reset_n,\n
        ");
    fprintf(fp, "        burst_begin, mem_local_addr, mem_be, mem_read_req, mem_size,
        mem_wdata, mem_write_req);\n");
    fprintf(fp, "\n");
    fprintf(fp, "parameter BURSTLEN = %i, BURSTWIDTH = %i;\n", burstlen, burstwidth);
    fprintf(fp, "parameter DATAWIDTH = %i, DATANUMBYTES = %i;\n", datawidth, datawidth
        /8);
    fprintf(fp, "parameter MEMCONWIDTH = %i, MEMCONNUMBYTES = %i, DDRSIZEWIDTH = %i;\n",
         memconwidth, memconwidth/8, ddrsizewidth);
    fprintf(fp, "parameter FIFOSIZE = %i, FIFOWIDTH = %i;\n", fifosize, fifowidth);
    fprintf(fp, "parameter RAMWIDTH = %i, RAMNUMBYTES = %i, RAMSIZEWIDTH = %i;\n",
        ramwidth, ramwidth/8, ramsizewidth);
    fprintf(fp, "parameter RATIO = %i, RAMLAT = %i;\n", ratio, ramlat);
    fprintf(fp, "\n");
    fprintf(fp, "output burst_begin;\n");
```

```
fprintf(fp, "output [DDRSIZEWIDTH-1:0] mem_local_addr;\n");
fprintf(fp, "output [MEMCONNUMBYTES-1: 0] mem_be;\n");
fprintf(fp, "output mem_read_req;\n");
fprintf(fp, "output [BURSTWIDTH-1:0] mem_size;\n");
fprintf(fp, "output [MEMCONWIDTH-1:0] mem_wdata;\n");
fprintf(fp, "output mem_write_req;\n");
fprintf(fp, "input clk, phy_clk;\n");
fprintf(fp, "input [MEMCONWIDTH-1:0] mem_rdata;\n");
fprintf(fp, "input mem_rdata_valid;\n");
fprintf(fp, "input mem_ready;\n");
fprintf(fp, "input mem_wdata_req;\n");
fprintf(fp, "input reset_n;\n");
fprintf(fp, "\n");
fprintf(fp, "input dtu_write_req;\n");
fprintf(fp, "input dtu_read_req;\n");
fprintf(fp, "input [DDRSIZEWIDTH-1:0] dtu_mem_addr;\n");
fprintf(fp, "input [RAMSIZEWIDTH-1:0] dtu_ram_addr;\n");
fprintf(fp, "input [%i:0] dtu_size;\n", transfersize-1);
fprintf(fp, "output dtu_ack;\n");
fprintf(fp, "output dtu_done;\n");
fprintf(fp, "\n");
fprintf(fp, "output[RAMWIDTH-1:0] ram_write_data;\n");
fprintf(fp, "input[RAMWIDTH-1:0] ram_read_data;\n");
fprintf(fp, "output[RAMSIZEWIDTH-1:0] ram_write_addr, ram_read_addr;\n");
fprintf(fp, "output[RAMNUMBYTES-1:0] ram_write_byte_en;\n");
fprintf(fp, "output ram_write_en;\n");
fprintf(fp, "\n");
fprintf(fp, "parameter IDLE = 0, WRITE = 1, READ = 2;\n");
fprintf(fp, "reg [DDRSIZEWIDTH-1:0] mem_addr[RAMLAT:0];\n");
fprintf(fp, "reg [2:0] state;\n");
fprintf(fp, "wire [DATAWIDTH-1:0] rdata, ram_write_dataw, ram_read_dataw;\n");
fprintf(fp, "\n");
fprintf(fp, "wire [RAMSIZEWIDTH-1:0] rfifo_addr;\n");
fprintf(fp, "reg fifo_write_reg[RAMLAT-1:0];\n");
fprintf(fp, "reg write_req_reg[RAMLAT-1:0];\n");
fprintf(fp, "reg read_req_reg[RAMLAT-1:0];\n");
fprintf(fp, "reg fifo_read_reg[0:0];\n");
fprintf(fp, "reg rdata_valid;\n");
fprintf(fp, "reg test_complete_reg[1:0];\n");
fprintf(fp, "reg [BURSTWIDTH-1:0] size_count[RAMLAT-1:0];\n");
fprintf(fp, "reg [RAMSIZEWIDTH-1:0] size;\n");
fprintf(fp, "reg [RAMSIZEWIDTH-1:0] ram_addr[RAMLAT-1:0];\n");
fprintf(fp, "reg [%i:0] data_count;\n", countwidth-1);
fprintf(fp, "reg ram_write_en_reg;\n");
fprintf(fp, "\n");
fprintf(fp, "wire read_req;\n");
fprintf(fp, "wire write_req;\n");
fprintf(fp, "wire [FIFOWIDTH-1:0] wfifo_count;\n");
fprintf(fp, "wire rfull, wempty, rempty, rdcmd_empty, wrcmd_full, wrcmd_empty;\n");
fprintf(fp, "wire [DATAWIDTH-1:0] mem_data;\n");
fprintf(fp, "wire not_stall;\n");
fprintf(fp, "wire fifo_write, fifo_read;\n");
fprintf(fp, "wire rdata_req;\n");
fprintf(fp, "wire [BURSTLEN+DDRSIZEWIDTH+1:0] wrmem_cmd, rdmem_cmd;\n");
fprintf(fp, "wire mem_cmd_ready, mem_cmd_issue;\n");
fprintf(fp, "\n");

fprintf(fp, "// FIFOs to interact with off-chip memory\n");
fprintf(fp, "memcmd_fifo cmd_store(\n");
fprintf(fp, "    .rdclk(phy_clk),\n");
fprintf(fp, "    .wrclk(clk),\n");
fprintf(fp, "    .data(wrmem_cmd),\n");
fprintf(fp, "    .rdreq(mem_cmd_ready),\n");
fprintf(fp, "    .rdempty(rdcmd_empty),\n");
fprintf(fp, "    .wrreq(mem_cmd_issue),\n");
fprintf(fp, "    .wrfull(wrcmd_full),\n");
fprintf(fp, "    .wrempty(wrcmd_empty),\n");
fprintf(fp, "    .q(rdmem_cmd));\n");
```

```
fprintf(fp, "\n");
fprintf(fp, "wfifo wdata_store(\n");
fprintf(fp, "    .rdclk(phy_clk),\n");
fprintf(fp, "    .wrclk(clk),\n");
fprintf(fp, "    .data(mem_data),\n");
fprintf(fp, "    .rdreq(mem_wdata_req),\n");
fprintf(fp, "    .wrreq(fifo_write),\n");
fprintf(fp, "    .wrempty(wempty),\n");
fprintf(fp, "    .q(mem_wdata),\n");
fprintf(fp, "    .wrusedw(wfifo_count));\n");
fprintf(fp, "\n");
fprintf(fp, "addr_fifo raddress_store (\n");
fprintf(fp, "    .clock(clk),\n");
fprintf(fp, "    .data(ram_addr[RAMLAT-2]),\n");
fprintf(fp, "    .rdreq(rdata_req),\n");
fprintf(fp, "    .wrreq(fifo_read),\n");
fprintf(fp, "    .empty(rempty),\n");
fprintf(fp, "    .full(rfull),\n");
fprintf(fp, "    .q(rfifo_addr));\n");
fprintf(fp, "\n");
fprintf(fp, "rfifo rdata_store(\n");
fprintf(fp, "    .data(mem_rdata),\n");
fprintf(fp, "    .rdclk(clk),\n");
fprintf(fp, "    .rdreq(rdata_req),\n");
fprintf(fp, "    .wrclk(phy_clk),\n");
fprintf(fp, "    .wrreq(mem_rdata_valid),\n");
fprintf(fp, "    .q(rdata),\n");
fprintf(fp, "    .rdempty(rdata_empty));\n");
fprintf(fp, "\n");

fprintf(fp, "assign mem_cmd_ready = (mem_ready == 1) && (rdcmd_empty == 0);\n");
fprintf(fp, "assign mem_cmd_issue = (wrcmd_full == 0) && (write_req == 1 || read_req
    == 1 || wrcmd_empty == 1);\n");
fprintf(fp, "assign wrmem_cmd[BURSTLEN+DDRSIZEWIDTH+1:DDRSIZEWIDTH+2] = size_count
    [0];\n");
fprintf(fp, "assign wrmem_cmd[DDRSIZEWIDTH+1:2] = mem_addr[0];\n");
fprintf(fp, "assign wrmem_cmd[1] = read_req;\n");
fprintf(fp, "assign wrmem_cmd[0] = write_req;\n");
fprintf(fp, "assign mem_write_req = rdmem_cmd[0] && rdcmd_empty == 0;\n");
fprintf(fp, "assign mem_read_req = rdmem_cmd[1] && rdcmd_empty == 0;\n");
fprintf(fp, "assign mem_local_addr = rdmem_cmd[DDRSIZEWIDTH+1:2];\n");
fprintf(fp, "assign burst_begin = 0;\n");
fprintf(fp, "assign mem_size = rdmem_cmd[BURSTLEN+DDRSIZEWIDTH+1:DDRSIZEWIDTH+2];\n"
    );
fprintf(fp, "assign mem_be = ~0;\n");
fprintf(fp, "assign fifo_write = fifo_write_reg[0];\n");
fprintf(fp, "assign write_req = (not_stall) ? write_req_reg[0] : 0;\n");
fprintf(fp, "assign read_req = (not_stall) ? read_req_reg[0] : 0;\n");
fprintf(fp, "assign fifo_read = (not_stall) ? fifo_read_reg[0] : 0;\n");
fprintf(fp, "assign not_stall = (wfifo_count < FIFOSIZE-%i) && (rfull == 0) && (
    wrcmd_full == 0);\n", ramlat);
fprintf(fp, "assign dtu_ack = (state == IDLE);\n");
fprintf(fp, "assign dtu_done = (state == IDLE) && wempty && rempty;\n");
fprintf(fp, "\n");
// match how memory is stored (like switching endianness)
j = memconwidth;
k = datawidth;
for (i = 0; i < ratio; i++)
{
    fprintf(fp, "assign ram_write_dataw[%i:%i] = rdata[%i:%i];\n", j-1, j-
        memconwidth, k-1, k-memconwidth);
    fprintf(fp, "assign mem_data[%i:%i] = ram_read_dataw[%i:%i];\n", j-1, j-
        memconwidth, k-1, k-memconwidth);
    j += memconwidth;
    k -= memconwidth;
}
fprintf(fp, "assign ram_write_data = ram_write_dataw[%i:%i];\n", datawidth-1,
    datawidth-ramwidth);
```

```
fprintf(fp, "assign ram_read_dataw[%i:%i] = ram_read_data;\n", datawidth-1,
    datawidth-ramwidth);
if (datawidth != ramwidth) {
    fprintf(fp, "assign ram_read_dataw[%i:0] = 0;\n", datawidth-ramwidth-1);
}
fprintf(fp, "assign ram_write_addr = rfifo_addr;\n");
fprintf(fp, "assign ram_read_addr = ram_addr[RAMLAT-1];\n");
fprintf(fp, "assign ram_write_byte_en = ~0;\n");
fprintf(fp, "assign ram_write_en = ram_write_en_reg;\n");
fprintf(fp, "assign rdata_req = !rdata_empty;\n");
fprintf(fp, "\n");

fprintf(fp, "// FSM to produce off-chip memory commands\n");
fprintf(fp, "always @ (posedge clk)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    if (reset_n == 0)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        state <= IDLE;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        case (state)\n");
fprintf(fp, "        IDLE:\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            if (dtu_write_req)\n");
fprintf(fp, "                state <= WRITE;\n");
fprintf(fp, "            else if (dtu_read_req)\n");
fprintf(fp, "                state <= READ;\n");
fprintf(fp, "            else\n");
fprintf(fp, "                state <= IDLE;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        WRITE:\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            if (not_stall && size == 0 && data_count < BURSTLEN)\n");
fprintf(fp, "                state <= IDLE;\n");
fprintf(fp, "            else\n");
fprintf(fp, "                state <= WRITE;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        READ:\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            if (not_stall && size == 0 && data_count < BURSTLEN)\n");
fprintf(fp, "                state <= IDLE;\n");
fprintf(fp, "            else\n");
fprintf(fp, "                state <= READ;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        default:\n");
fprintf(fp, "        begin\n");
fprintf(fp, "            state <= IDLE;\n");
fprintf(fp, "        end\n");
fprintf(fp, "        endcase\n");
fprintf(fp, "    end\n");
fprintf(fp, "end\n");
fprintf(fp, "\n");

fprintf(fp, "always @ (posedge clk)\n");
fprintf(fp, "begin\n");
fprintf(fp, "\n");
fprintf(fp, "    if (reset_n == 0)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        size <= 0;\n");
fprintf(fp, "        data_count <= 0;\n");
fprintf(fp, "        size_count[RAMLAT-1] <= 1;\n");
fprintf(fp, "        mem_addr[RAMLAT] <= 0;\n");
fprintf(fp, "        ram_addr[RAMLAT-1] <= 0;\n");
fprintf(fp, "        fifo_write_reg[RAMLAT-1] <= 0;\n");
fprintf(fp, "        write_req_reg[RAMLAT-1] <= 0;\n");
fprintf(fp, "        fifo_read_reg[0] <= 0;\n");
fprintf(fp, "        read_req_reg[RAMLAT-1] <= 0;\n");
```

```
fprintf(fp, "    end\n");
fprintf(fp, "    else if (state == IDLE)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        size <= dtu_size;\n");
fprintf(fp, "        size_count[RAMLAT-1] <= BURSTLEN;\n");
fprintf(fp, "        mem_addr[RAMLAT] <= dtu_mem_addr;\n");
fprintf(fp, "        ram_addr[RAMLAT-1] <= dtu_ram_addr;\n");
fprintf(fp, "        fifo_write_reg[RAMLAT-1] <= 0;\n");
fprintf(fp, "        write_req_reg[RAMLAT-1] <= 0;\n");
fprintf(fp, "        fifo_read_reg[0] <= 0;\n");
fprintf(fp, "        read_req_reg[RAMLAT-1] <= 0;\n");
fprintf(fp, "        data_count <= 0;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else if (data_count >= BURSTLEN && not_stall)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        data_count <= data_count - BURSTLEN;\n");
fprintf(fp, "        mem_addr[RAMLAT] <= mem_addr[RAMLAT] + BURSTLEN;\n");
fprintf(fp, "        fifo_write_reg[RAMLAT-1] <= 0;\n");
fprintf(fp, "        write_req_reg[RAMLAT-1] <= state == WRITE;\n");
fprintf(fp, "        fifo_read_reg[0] <= 0;\n");
fprintf(fp, "        read_req_reg[RAMLAT-1] <= state == READ;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else if (size == 0 && data_count == 0 && not_stall)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        fifo_write_reg[RAMLAT-1] <= 0;\n");
fprintf(fp, "        write_req_reg[RAMLAT-1] <= 0;\n");
fprintf(fp, "        fifo_read_reg[0] <= 0;\n");
fprintf(fp, "        read_req_reg[RAMLAT-1] <= 0;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else if (size == 0 && not_stall)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        size_count[RAMLAT-1] <= data_count;\n");
fprintf(fp, "        fifo_write_reg[RAMLAT-1] <= 0;\n");
fprintf(fp, "        write_req_reg[RAMLAT-1] <= state == WRITE;\n");
fprintf(fp, "        fifo_read_reg[0] <= 0;\n");
fprintf(fp, "        read_req_reg[RAMLAT-1] <= state == READ;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else if (not_stall)\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        size <= size - 1;\n");
fprintf(fp, "        data_count <= data_count + RATIO - BURSTLEN;\n");
fprintf(fp, "        mem_addr[RAMLAT] <= mem_addr[RAMLAT] + BURSTLEN;\n");
fprintf(fp, "        ram_addr[RAMLAT-1] <= ram_addr[RAMLAT-1]+1;\n");
fprintf(fp, "        fifo_write_reg[RAMLAT-1] <= state == WRITE;\n");
fprintf(fp, "        write_req_reg[RAMLAT-1] <= state == WRITE;\n");
fprintf(fp, "        fifo_read_reg[0] <= state == READ;\n");
fprintf(fp, "        read_req_reg[RAMLAT-1] <= state == READ;\n");
fprintf(fp, "    end\n");
fprintf(fp, "    else\n");
fprintf(fp, "    begin\n");
fprintf(fp, "        fifo_write_reg[RAMLAT-1] <= 0;\n");
fprintf(fp, "    end\n");
fprintf(fp, "end\n");
fprintf(fp, "\n");
fprintf(fp, "\n");

fprintf(fp, "always @ (posedge clk)\n");
fprintf(fp, "begin\n");
fprintf(fp, "    if (reset_n == 0)\n");
fprintf(fp, "    begin\n");
for (i = 0; i < ramlat-1; i++)
{
    fprintf(fp, "        fifo_write_reg[%i] <= 0;\n", i);
}
fprintf(fp, "    end\n");
fprintf(fp, "    else\n");
fprintf(fp, "    begin\n");
for (i = 0; i < ramlat-1; i++)
```

```
    {
        fprintf(fp, "        fifo_write_reg[%i] <= fifo_write_reg[%i];\n", i, i+1);
    }
    fprintf(fp, "    end\n");
    fprintf(fp, "\n");
    fprintf(fp, "    if (reset_n == 0)\n");
    fprintf(fp, "    begin\n");
    for (i = 0; i < ramlat-1; i++)
    {
        fprintf(fp, "        mem_addr[%i] <= 0;\n", i);
        fprintf(fp, "        ram_addr[%i] <= 0;\n", i);
        fprintf(fp, "        size_count[%i] <= 1;\n", i);
        fprintf(fp, "        write_req_reg[%i] <= 0;\n", i);
        fprintf(fp, "        read_req_reg[%i] <= 0;\n", i);
    }
    fprintf(fp, "        mem_addr[%i] <= 0;\n", ramlat-1);
    fprintf(fp, "    end\n");
    fprintf(fp, "    else if (not_stall)\n");
    fprintf(fp, "    begin\n");
    for (i = 0; i < ramlat-1; i++)
    {
        fprintf(fp, "        size_count[%i] <= size_count[%i];\n", i, i+1);
        fprintf(fp, "        mem_addr[%i] <= mem_addr[%i];\n", i, i+1);
        fprintf(fp, "        ram_addr[%i] <= ram_addr[%i];\n", i, i+1);
        fprintf(fp, "        write_req_reg[%i] <= write_req_reg[%i];\n", i, i+1);
        fprintf(fp, "        read_req_reg[%i] <= read_req_reg[%i];\n", i, i+1);
    }
    fprintf(fp, "        mem_addr[%i] <= mem_addr[%i];\n", ramlat-1, ramlat);
    fprintf(fp, "    end\n");
    fprintf(fp, "    \n");
    fprintf(fp, "    ram_write_en_reg  <= rdata_req;\n");
    fprintf(fp, "end\n");
    fprintf(fp, "\n");
    fprintf(fp, "endmodule\n");
    fclose(fp);
}
```

Appendix B

Experimental Data

B.1 Software Performance

B.1.1 Intel MKL

		Intel MKL								
		SGETRF (Single Precision)								
		4 Cores			2 Cores			1 Cores		
N	Number of Operations	Runtime (s)	GFLOPS	GFLOPS per Watt	Runtime (s)	GFLOPS	GFLOPS per Watt	Runtime (s)	GFLOPS	GFLOPS per Watt
25	1.14E+04	2.43E-05	0.47	0.003	1.63E-04	0.07	0.001	2.85E-04	0.04	0.001
50	8.71E+04	7.38E-05	1.18	0.007	3.97E-04	0.22	0.003	6.64E-04	0.13	0.003
75	2.90E+05	1.29E-04	2.24	0.014	6.05E-04	0.48	0.006	1.01E-03	0.29	0.007
100	6.82E+05	2.00E-04	3.41	0.021	8.27E-04	0.82	0.010	1.29E-03	0.53	0.013
150	2.28E+06	4.26E-04	5.36	0.033	1.45E-03	1.57	0.020	2.21E-03	1.03	0.026
200	5.39E+06	6.77E-04	7.97	0.050	2.11E-03	2.55	0.032	3.06E-03	1.76	0.044
250	1.05E+07	1.09E-03	9.62	0.060	3.15E-03	3.33	0.042	4.32E-03	2.43	0.061
300	1.81E+07	1.50E-03	12.07	0.075	4.27E-03	4.24	0.053	5.69E-03	3.19	0.080
350	2.88E+07	2.10E-03	13.70	0.086	5.54E-03	5.19	0.065	7.27E-03	3.96	0.099
400	4.29E+07	2.65E-03	16.18	0.101	6.69E-03	6.42	0.080	8.57E-03	5.01	0.125
500	8.37E+07	4.39E-03	19.06	0.119	1.04E-02	8.07	0.101	1.33E-02	6.30	0.157
600	1.45E+08	6.39E-03	22.62	0.141	1.45E-02	9.95	0.124	1.88E-02	7.71	0.193
800	3.42E+08	1.03E-02	33.11	0.207	2.20E-02	15.55	0.194	3.23E-02	10.59	0.265
1000	6.68E+08	1.65E-02	40.46	0.253	3.52E-02	18.96	0.237	5.62E-02	11.90	0.297
1500	2.25E+09	5.01E-02	44.99	0.281	1.00E-01	22.43	0.280	1.74E-01	12.97	0.324
2000	5.34E+09	1.01E-01	52.99	0.331	1.96E-01	27.24	0.340	3.56E-01	15.01	0.375
3000	1.80E+10	3.04E-01	59.27	0.370	5.91E-01	30.48	0.381	1.10E+00	16.36	0.409
4000	4.27E+10	6.68E-01	63.93	0.400	1.29E+00	33.04	0.413	2.36E+00	18.11	0.453
5000	8.34E+10	1.26E+00	66.28	0.414	2.41E+00	34.64	0.433	4.44E+00	18.77	0.469
7500	2.81E+11	3.99E+00	70.46	0.440	7.54E+00	37.32	0.466	1.46E+01	19.32	0.483
10000	6.67E+11	9.00E+00	74.05	0.463	1.72E+01	38.78	0.485	3.40E+01	19.64	0.491
12500	1.30E+12	1.74E+01	74.78	0.467	3.33E+01	39.17	0.490	6.47E+01	20.14	0.504
15000	2.25E+12	2.92E+01	77.07	0.482	5.55E+01	40.53	0.507	1.10E+02	20.55	0.514
17500	3.57E+12	4.61E+01	77.46	0.484	8.80E+01	40.59	0.507	1.74E+02	20.50	0.512
20000	5.33E+12	6.74E+01	79.10	0.494	1.29E+02	41.30	0.516	2.56E+02	20.80	0.520
25000	1.04E+13	1.30E+02	80.05	0.500	2.50E+02	41.72	0.522	4.97E+02	20.98	0.524
30000	1.80E+13	2.22E+02	81.16	0.507	4.27E+02	42.14	0.527	8.51E+02	21.15	0.529
40000	4.27E+13	5.19E+02	82.28	0.514	1.00E+03	42.67	0.533	2.00E+03	21.34	0.533

Table B.1: Single Precision Performance of MKL on Xeon 5160 for Various Matrix Size

		Intel MKL DGETRF (Double Precision)								
		4 Cores			2 Cores			1 Cores		
N	Number of Operations	Runtime (s)	GFLOPS	GFLOPS per Watt	Runtime (s)	GFLOPS	GFLOPS per Watt	Runtime (s)	GFLOPS	GFLOPS per Watt
25	1.14E+04	2.77E-05	0.41	0.003	1.72E-04	0.07	0.001	2.83E-04	0.04	0.001
50	8.71E+04	8.50E-05	1.02	0.006	3.96E-04	0.22	0.003	6.58E-04	0.13	0.003
75	2.90E+05	1.79E-04	1.62	0.010	6.49E-04	0.45	0.006	9.98E-04	0.29	0.007
100	6.82E+05	2.46E-04	2.77	0.017	9.07E-03	0.08	0.001	1.26E-03	0.54	0.014
150	2.28E+06	5.91E-04	3.87	0.024	1.74E-03	1.32	0.016	2.24E-03	1.02	0.025
200	5.39E+06	9.67E-04	5.58	0.035	2.67E-03	2.02	0.025	3.21E-03	1.68	0.042
250	1.05E+07	1.56E-03	6.75	0.042	3.86E-03	2.72	0.034	4.64E-03	2.26	0.057
300	1.81E+07	2.23E-03	8.12	0.051	5.33E-03	3.40	0.043	6.18E-03	2.93	0.073
350	2.88E+07	3.22E-03	8.93	0.056	7.09E-03	4.06	0.051	8.31E-03	3.46	0.087
400	4.29E+07	4.23E-03	10.14	0.063	9.07E-03	4.73	0.059	1.08E-02	3.99	0.100
500	8.37E+07	6.92E-03	12.10	0.076	1.39E-02	6.01	0.075	1.83E-02	4.57	0.114
600	1.45E+08	1.02E-02	14.16	0.089	2.02E-02	7.16	0.089	2.84E-02	5.09	0.127
800	3.42E+08	1.76E-02	19.40	0.121	3.32E-02	10.31	0.129	5.31E-02	6.44	0.161
1000	6.68E+08	2.78E-02	24.07	0.150	5.51E-02	12.13	0.152	9.68E-02	6.90	0.173
1500	2.25E+09	8.01E-02	28.13	0.176	1.54E-01	14.59	0.182	2.88E-01	7.82	0.195
2000	5.34E+09	1.79E-01	29.90	0.187	3.41E-01	15.66	0.196	6.40E-01	8.34	0.208
3000	1.80E+10	5.56E-01	32.39	0.202	1.05E+00	17.16	0.214	2.02E+00	8.92	0.223
4000	4.27E+10	1.26E+00	33.82	0.211	2.37E+00	18.01	0.225	4.52E+00	9.45	0.236
5000	8.34E+10	2.36E+00	35.32	0.221	4.54E+00	18.37	0.230	8.65E+00	9.64	0.241
7500	2.81E+11	7.53E+00	37.38	0.234	1.43E+01	19.61	0.245	2.80E+01	10.05	0.251
10000	6.67E+11	1.74E+01	38.30	0.239	3.32E+01	20.11	0.251	6.52E+01	10.22	0.255
12500	1.30E+12	3.33E+01	39.09	0.244	6.30E+01	20.66	0.258	1.26E+02	10.36	0.259
15000	2.25E+12	5.67E+01	39.71	0.248	1.07E+02	20.95	0.262	2.14E+02	10.51	0.263
17500	3.57E+12	8.94E+01	39.99	0.250	1.69E+02	21.15	0.264	3.38E+02	10.56	0.264
20000	5.33E+12	1.32E+02	40.37	0.252	2.50E+02	21.33	0.267	5.00E+02	10.66	0.267
25000	1.04E+13	NA	NA	NA	4.85E+02	21.50	0.269	9.72E+02	10.71	0.268
30000	1.80E+13	NA	NA	NA	8.35E+02	21.56	0.269	1.67E+03	10.77	0.269

Table B.2: Double Precision Performance of MKL on Xeon 5160 for Various Matrix Size

B.1.2 Basic Code

		Simple C code		
		Single Precision		
N	Number of Operations	Runtime (s)	GFLOPS	GFLOPS per Watt
25	1.14E+04	6.74E-06	1.68	0.0421
50	8.71E+04	4.70E-05	1.85	0.0463
75	2.90E+05	1.57E-04	1.84	0.0460
100	6.82E+05	3.83E-04	1.78	0.0445
150	2.28E+06	1.29E-03	1.77	0.0443
200	5.39E+06	3.04E-03	1.78	0.0444
250	1.05E+07	5.86E-03	1.79	0.0448
300	1.81E+07	1.00E-02	1.81	0.0452
350	2.88E+07	1.58E-02	1.82	0.0454
400	4.29E+07	2.36E-02	1.82	0.0455
500	8.37E+07	4.62E-02	1.81	0.0453
600	1.45E+08	7.97E-02	1.81	0.0453
800	3.42E+08	1.89E-01	1.81	0.0453
1000	6.68E+08	3.83E-01	1.74	0.0436
1500	2.25E+09	1.68E+00	1.34	0.0335
2000	5.34E+09	4.55E+00	1.17	0.0293
3000	1.80E+10	1.64E+01	1.10	0.0274
4000	4.27E+10	3.95E+01	1.08	0.0271
5000	8.34E+10	7.68E+01	1.09	0.0271
7500	2.81E+11	2.58E+02	1.09	0.0273
10000	6.67E+11	6.10E+02	1.09	0.0273
12500	1.30E+12	1.19E+03	1.10	0.0275
15000	2.25E+12	2.05E+03	1.10	0.0275
17500	3.57E+12	3.24E+03	1.10	0.0276
20000	5.33E+12	4.84E+03	1.10	0.0275
25000	1.04E+13	9.47E+03	1.10	0.0275
30000	1.80E+13	1.63E+04	1.10	0.0275
40000	4.27E+13	3.87E+04	1.10	0.0276

Table B.3: Single Precision Performance of Basic Code on Xeon 5160 for Various Matrix Size

		Simple C code Double Precision		
N	Number of Operations	Runtime (s)	GFLOPS	GFLOPS per Watt
17500	3.57E+12	6.51E+03	0.549	0.0137
20000	5.33E+12	9.71E+03	0.550	0.0137
25000	1.04E+13	1.91E+04	0.545	0.0136
30000	1.80E+13	3.27E+04	0.550	0.0137

Table B.4: Double Precision Performance of Basic Code on Xeon 5160 for Various Matrix Size

B.2 FPGA Performance

B.2.1 Stratix III 3SL340F1760C3

		Single Precision 120 PEs, Block Size 240x240				Double Precision 57 PEs, Block Size 114x114			
N	Number of Operations	Cycle Count	Runtime (s)	GFLOPS	GFLOPS per Watt	Cycle Count	Runtime (s)	GFLOPS	GFLOPS per Watt
25	1.14E+04	2.23E+03	1.11E-05	1.02	0.06	3.80E+03	2.24E-05	0.51	0.03
50	8.71E+04	4.62E+03	2.31E-05	3.77	0.21	7.87E+03	4.63E-05	1.88	0.09
75	2.90E+05	7.59E+03	3.80E-05	7.63	0.42	2.02E+04	1.19E-04	2.44	0.12
100	6.82E+05	1.12E+04	5.60E-05	12.18	0.68	2.87E+04	1.69E-04	4.04	0.20
150	2.28E+06	3.10E+04	1.55E-04	14.75	0.82	6.36E+04	3.74E-04	6.10	0.31
200	5.39E+06	4.85E+04	2.43E-04	22.24	1.24	1.18E+05	6.92E-04	7.79	0.39
250	1.05E+07	1.22E+05	6.12E-04	17.17	0.95	2.10E+05	1.23E-03	8.51	0.43
300	1.81E+07	1.46E+05	7.30E-04	24.84	1.38	3.02E+05	1.78E-03	10.20	0.51
350	2.88E+07	1.83E+05	9.15E-04	31.43	1.75	5.02E+05	2.95E-03	9.75	0.49
400	4.29E+07	3.01E+05	1.51E-03	28.46	1.58	6.40E+05	3.76E-03	11.40	0.57
500	8.37E+07	5.85E+05	2.93E-03	28.61	1.59	1.07E+06	6.32E-03	13.24	0.66
600	1.45E+08	7.54E+05	3.77E-03	38.33	2.13	1.83E+06	1.07E-02	13.45	0.67
800	3.42E+08	1.80E+06	9.01E-03	38.00	2.11	4.27E+06	2.51E-02	13.63	0.68
1000	6.68E+08	3.54E+06	1.77E-02	37.76	2.10	7.02E+06	4.13E-02	16.18	0.81
1500	2.25E+09	1.09E+07	5.44E-02	41.45	2.30	2.37E+07	1.40E-01	16.15	0.81
2000	5.34E+09	2.45E+07	1.23E-01	43.50	2.42	5.29E+07	3.11E-01	17.16	0.86
3000	1.80E+10	7.88E+07	3.94E-01	45.71	2.54	1.73E+08	1.02E+00	17.70	0.89
4000	4.27E+10	1.90E+08	9.51E-01	44.90	2.49	4.08E+08	2.40E+00	17.78	0.89
5000	8.34E+10	3.63E+08	1.81E+00	45.96	2.55	7.67E+08	4.51E+00	18.47	0.92
7500	2.81E+11	1.22E+09	6.08E+00	46.28	2.57	2.57E+09	1.51E+01	18.62	0.93
10000	6.67E+11	2.87E+09	1.43E+01	46.48	2.58	6.06E+09	3.57E+01	18.70	0.93
12500	1.30E+12	5.59E+09	2.80E+01	46.57	2.59	1.18E+10	6.95E+01	18.74	0.94
15000	2.25E+12	9.53E+09	4.77E+01	47.21	2.62	2.04E+10	1.20E+02	18.77	0.94
17500	3.57E+12	1.51E+10	7.57E+01	47.19	2.62	3.23E+10	1.90E+02	18.78	0.94
20000	5.33E+12	2.26E+10	1.13E+02	47.16	2.62	4.81E+10	2.83E+02	18.84	0.94
25000	1.04E+13	4.42E+10	2.21E+02	47.13	2.62	9.40E+10	5.53E+02	18.85	0.94
30000	1.80E+13	7.60E+10	3.80E+02	47.40	2.63	1.62E+11	9.55E+02	18.86	0.94
40000	4.27E+13	1.80E+11	9.02E+02	47.30	2.63	3.83E+11	2.25E+03	18.95	0.95

Table B.5: Performance on Stratix III 3SL340 for Various Matrix Size

B.2.2 Stratix II 2S180F1508C3

		Single Precision				Double Precision			
		60 PEs, Block Size 120x120				29 PEs, Block Size 58x58			
N	Number of Operations	Cycle Count	Runtime (s)	GFLOPS	GFLOPS per Watt	Cycle Count	Runtime (s)	GFLOPS	GFLOPS per Watt
25	1.14E+04	2.95E+03	1.60E-05	0.71	0.04	2.53E+03	1.81E-05	0.63	0.03
50	8.71E+04	6.17E+03	3.33E-05	2.61	0.15	7.56E+03	5.40E-05	1.61	0.08
75	2.90E+05	1.51E+04	8.14E-05	3.56	0.20	1.73E+04	1.24E-04	2.34	0.12
100	6.82E+05	2.18E+04	1.18E-04	5.77	0.32	3.15E+04	2.25E-04	3.03	0.15
150	2.28E+06	5.24E+04	2.83E-04	8.06	0.45	8.70E+04	6.21E-04	3.68	0.18
200	5.39E+06	1.00E+05	5.43E-04	9.93	0.55	1.45E+05	1.04E-03	5.19	0.26
250	1.05E+07	1.84E+05	9.94E-04	10.57	0.59	2.80E+05	2.00E-03	5.25	0.26
300	1.81E+07	2.20E+05	1.19E-03	15.23	0.85	4.81E+05	3.44E-03	5.27	0.26
350	2.88E+07	3.43E+05	1.86E-03	15.50	0.86	7.63E+05	5.45E-03	5.28	0.26
400	4.29E+07	5.13E+05	2.77E-03	15.48	0.86	9.74E+05	6.96E-03	6.16	0.31
500	8.37E+07	1.00E+06	5.41E-03	15.46	0.86	1.96E+06	1.40E-02	5.99	0.30
600	1.45E+08	1.46E+06	7.88E-03	18.33	1.02	3.12E+06	2.23E-02	6.49	0.32
800	3.42E+08	3.62E+06	1.95E-02	17.51	0.97	7.19E+06	5.13E-02	6.67	0.33
1000	6.68E+08	6.55E+06	3.54E-02	18.86	1.05	1.38E+07	9.88E-02	6.76	0.34
1500	2.25E+09	2.07E+07	1.12E-01	20.16	1.12	4.47E+07	3.19E-01	7.06	0.35
2000	5.34E+09	5.01E+07	2.71E-01	19.73	1.10	1.04E+08	7.41E-01	7.21	0.36
3000	1.80E+10	1.60E+08	8.67E-01	20.77	1.15	3.50E+08	2.50E+00	7.20	0.36
4000	4.27E+10	3.81E+08	2.06E+00	20.72	1.15	8.17E+08	5.84E+00	7.32	0.37
5000	8.34E+10	7.45E+08	4.03E+00	20.70	1.15	1.60E+09	1.14E+01	7.29	0.36
7500	2.81E+11	2.46E+09	1.33E+01	21.13	1.17	5.36E+09	3.83E+01	7.35	0.37
10000	6.67E+11	5.84E+09	3.16E+01	21.11	1.17	1.26E+10	9.03E+01	7.38	0.37
12500	1.30E+12	1.14E+10	6.17E+01	21.10	1.17	2.48E+10	1.77E+02	7.36	0.37
15000	2.25E+12	1.96E+10	1.06E+02	21.25	1.18	4.27E+10	3.05E+02	7.38	0.37
17500	3.57E+12	3.11E+10	1.68E+02	21.22	1.18	6.77E+10	4.83E+02	7.39	0.37
20000	5.33E+12	4.65E+10	2.52E+02	21.20	1.18	1.01E+11	7.20E+02	7.40	0.37
25000	1.04E+13	9.06E+10	4.90E+02	21.27	1.18	1.97E+11	1.41E+03	7.40	0.37
30000	1.80E+13	1.56E+11	8.45E+02	21.31	1.18	3.40E+11	2.43E+03	7.41	0.37
40000	4.27E+13	3.70E+11	2.00E+03	21.31	1.18	8.06E+11	5.75E+03	7.42	0.37

Table B.6: Performance on Stratix II 2S180 for Various Matrix Size

B.3 Effect of Block Size on Performance

B.3.1 Square Block Size

	Cycle Count			
Block Size	**40 PEs**	**80 PEs**	**120 Pes**	**160 Pes**
40	9.44E+09	NA	NA	NA
80	8.61E+09	4.45E+09	NA	NA
120	8.47E+09	5.72E+09	2.95E+09	NA
160	8.42E+09	4.26E+09	4.27E+09	2.20E+09
200	8.40E+09	5.08E+09	3.41E+09	3.42E+09
240	8.39E+09	4.22E+09	2.87E+09	2.83E+09
280	8.38E+09	4.83E+09	3.63E+09	2.46E+09
320	8.38E+09	4.21E+09	3.19E+09	2.15E+09
360	8.37E+09	4.69E+09	2.85E+09	2.82E+09

Table B.7: Cycle Count for Various Block Size

	Simple LU Factorization	Block LU Factorization							
		Block Size 144x144		Block Size 256x256		Block Size 288x288		Block Size 432x432	
N	**Cycle Count**	**Cycle Count**	**Blocking Overhead**	**Cycle Count**	**Blocking Overhead**	**Cycle Count**	**Blocking Overhead**	**Cycle Count**	**Blocking Overhead**
25	1.96E+03	1.96E+03	0.00%	1.96E+03	0.00%	1.96E+03	0.00%	1.96E+03	0.00%
50	4.22E+03	4.22E+03	0.00%	4.22E+03	0.00%	4.22E+03	0.00%	4.22E+03	0.00%
75	7.12E+03	7.12E+03	0.00%	7.12E+03	0.00%	7.12E+03	0.00%	7.12E+03	0.00%
100	1.06E+04	1.06E+04	0.00%	1.06E+04	0.00%	1.06E+04	0.00%	1.06E+04	0.00%
150	3.05E+04	4.28E+04	40.13%	3.05E+04	0.00%	3.05E+04	0.00%	3.05E+04	0.00%
200	4.93E+04	5.54E+04	12.32%	4.93E+04	0.00%	4.93E+04	0.00%	4.93E+04	0.00%
250	7.04E+04	7.65E+04	8.70%	7.04E+04	0.00%	7.04E+04	0.00%	7.04E+04	0.00%
300	1.39E+05	1.69E+05	22.06%	1.48E+05	6.87%	1.58E+05	13.69%	1.39E+05	0.00%
350	1.79E+05	1.98E+05	10.74%	1.87E+05	4.38%	1.91E+05	6.45%	1.79E+05	0.00%
400	2.22E+05	2.41E+05	8.71%	2.28E+05	2.84%	2.34E+05	5.22%	2.22E+05	0.00%
500	4.38E+05	4.79E+05	9.31%	4.38E+05	0.10%	4.49E+05	2.65%	4.55E+05	3.89%
600	7.64E+05	8.43E+05	10.43%	7.77E+05	1.70%	8.04E+05	5.27%	7.81E+05	2.23%
800	1.52E+06	1.63E+06	7.49%	1.85E+06	22.14%	1.55E+06	2.44%	1.54E+06	1.13%
1000	2.63E+06	2.80E+06	6.38%	3.09E+06	17.37%	2.71E+06	2.99%	2.68E+06	2.08%
1500	9.08E+06	9.61E+06	5.80%	9.99E+06	9.94%	9.30E+06	2.41%	9.20E+06	1.28%
2000	1.98E+07	2.08E+07	4.96%	2.32E+07	17.02%	2.01E+07	1.64%	2.00E+07	1.04%
3000	6.56E+07	6.84E+07	4.36%	7.66E+07	16.78%	6.66E+07	1.55%	6.61E+07	0.73%
4000	1.54E+08	1.60E+08	4.04%	1.80E+08	16.64%	1.56E+08	1.23%	1.55E+08	0.76%
5000	2.99E+08	3.11E+08	3.84%	3.36E+08	12.37%	3.03E+08	1.22%	3.01E+08	0.62%
7500	1.02E+09	1.06E+09	3.93%	1.14E+09	12.04%	1.03E+09	1.12%	1.02E+09	0.53%
10000	2.37E+09	2.45E+09	3.42%	2.70E+09	14.24%	2.39E+09	0.93%	2.38E+09	0.49%
12500	4.58E+09	4.73E+09	3.31%	5.21E+09	13.82%	4.62E+09	0.92%	4.59E+09	0.42%
15000	7.96E+09	8.23E+09	3.32%	9.03E+09	13.40%	8.03E+09	0.89%	8.00E+09	0.41%
17500	1.26E+10	1.30E+10	3.22%	1.42E+10	13.26%	1.27E+10	0.84%	1.26E+10	0.40%
20000	1.86E+10	1.92E+10	3.17%	2.13E+10	14.23%	1.88E+10	0.84%	1.87E+10	0.40%
25000	3.64E+10	3.76E+10	3.14%	4.14E+10	13.73%	3.67E+10	0.80%	3.66E+10	0.37%
30000	6.30E+10	6.50E+10	3.11%	7.15E+10	13.41%	6.35E+10	0.80%	6.33E+10	0.37%
40000	1.49E+11	1.53E+11	3.06%	1.69E+11	13.62%	1.50E+11	0.77%	1.49E+11	0.35%

Table B.8: Cycle Count for Various Matrix Size on 144 PEs Compute Engines with Different Block Size

B.3.2 Rectangular Block Size

	Simple LU Factorization	Block LU Factorization															
		Block Size 144x512		Block Size 288x256		Block Size 432x170		Block Size 576x128		Block Size 1152x64		Block Size 2304x32	
N	Cycle Count	Cycle Count	Blocking Overhead	Cycle Count	Blocking Overhead	Cycle Count	Blocking Overhead	Cycle Count	Blocking Overhead	Cycle Count	Blocking Overhead	Cycle Count	Blocking Overhead
25	1.96E+03	1.96E+03	0.00%	1.96E+03	0.00%	1.96E+03	0.00%	1.96E+03	0.00%	1.96E+03	0.00%	1.96E+03	0.00%
50	4.22E+03	4.22E+03	0.00%	4.22E+03	0.00%	4.22E+03	0.00%	4.22E+03	0.00%	4.22E+03	0.00%	5.05E+03	19.63%
75	7.12E+03	7.12E+03	0.00%	7.12E+03	0.00%	7.12E+03	0.00%	7.12E+03	0.00%	8.72E+03	22.51%	9.32E+03	30.98%
100	1.06E+04	1.06E+04	0.00%	1.06E+04	0.00%	1.06E+04	0.00%	1.06E+04	0.00%	1.13E+04	6.43%	1.48E+04	39.16%
150	3.05E+04	3.51E+04	14.82%	3.05E+04	0.00%	3.05E+04	0.00%	3.12E+04	2.26%	2.89E+04	-5.24%	2.76E+04	-9.75%
200	4.93E+04	5.36E+04	8.65%	4.93E+04	0.00%	5.04E+04	2.25%	4.91E+04	-0.39%	4.76E+04	-3.39%	5.05E+04	2.48%
250	7.04E+04	7.46E+04	6.06%	7.04E+04	0.00%	7.15E+04	1.62%	6.95E+04	-1.27%	7.14E+04	1.49%	7.61E+04	8.07%
300	1.39E+05	1.51E+05	9.29%	1.48E+05	6.97%	1.31E+05	-5.19%	1.37E+05	-0.80%	1.22E+05	-12.16%	1.19E+05	-14.32%
350	1.79E+05	1.92E+05	7.08%	1.87E+05	4.49%	1.85E+05	3.24%	1.76E+05	-1.92%	1.63E+05	-8.82%	1.67E+05	-6.96%
400	2.22E+05	2.35E+05	5.69%	2.29E+05	2.94%	2.28E+05	2.48%	2.20E+05	-0.97%	2.24E+05	0.85%	2.33E+05	5.01%
500	4.38E+05	4.63E+05	5.69%	4.39E+05	0.17%	4.55E+05	3.88%	4.25E+05	-2.87%	4.32E+05	-1.38%	4.27E+05	-2.50%
600	7.64E+05	8.08E+05	5.82%	7.77E+05	1.78%	7.00E+05	-8.34%	7.47E+05	-2.16%	6.79E+05	-11.10%	6.77E+05	-11.30%
800	1.52E+06	1.59E+06	4.51%	1.51E+06	-0.77%	1.59E+06	4.79%	1.49E+06	-1.97%	1.48E+06	-2.22%	1.47E+06	-2.89%
1000	2.63E+06	2.73E+06	3.80%	2.87E+06	8.96%	2.79E+06	6.20%	2.92E+06	10.90%	2.76E+06	5.08%	2.76E+06	5.10%
1500	9.08E+06	9.37E+06	3.12%	9.13E+06	0.56%	9.18E+06	1.04%	9.01E+06	-0.81%	8.82E+06	-2.95%	8.76E+06	-3.51%
2000	1.98E+07	2.03E+07	2.59%	2.08E+07	5.02%	2.06E+07	3.93%	2.07E+07	4.31%	2.02E+07	2.08%	2.01E+07	1.46%
3000	6.56E+07	6.70E+07	2.15%	6.79E+07	3.60%	6.76E+07	3.09%	6.77E+07	3.20%	6.67E+07	1.70%	6.65E+07	1.46%
4000	1.54E+08	1.57E+08	1.78%	1.57E+08	2.11%	1.58E+08	2.30%	1.57E+08	2.00%	1.56E+08	0.96%	1.55E+08	0.78%
5000	2.99E+08	3.04E+08	1.60%	3.05E+08	1.92%	3.04E+08	1.77%	3.05E+08	1.83%	3.02E+08	0.97%	3.02E+08	0.83%
7500	1.02E+09	1.03E+09	1.35%	1.01E+09	-0.16%	1.01E+09	-0.29%	1.01E+09	-0.28%	1.01E+09	-0.89%	1.01E+09	-0.98%
10000	2.37E+09	2.40E+09	1.22%	2.39E+09	0.74%	2.39E+09	0.75%	2.38E+09	0.63%	2.37E+09	0.23%	2.37E+09	0.20%
12500	4.58E+09	4.63E+09	1.16%	4.64E+09	1.49%	4.64E+09	1.37%	4.64E+09	1.44%	4.63E+09	1.11%	4.63E+09	1.11%
15000	7.96E+09	8.05E+09	1.10%	8.00E+09	0.44%	7.99E+09	0.36%	7.99E+09	0.40%	7.97E+09	0.13%	7.97E+09	0.14%
17500	1.26E+10	1.27E+10	1.06%	1.27E+10	0.88%	1.27E+10	0.87%	1.27E+10	0.85%	1.26E+10	0.64%	1.26E+10	0.67%
20000	1.86E+10	1.88E+10	1.03%	1.89E+10	1.25%	1.89E+10	1.24%	1.89E+10	1.22%	1.88E+10	1.03%	1.88E+10	1.07%
25000	3.64E+10	3.68E+10	0.99%	3.68E+10	0.96%	3.68E+10	0.94%	3.68E+10	0.94%	3.67E+10	0.81%	3.68E+10	0.87%
30000	6.30E+10	6.36E+10	0.96%	6.35E+10	0.70%	6.35E+10	0.71%	6.35E+10	0.69%	6.34E+10	0.58%	6.34E+10	0.65%
40000	1.49E+11	1.50E+11	0.93%	1.50E+11	0.96%	1.50E+11	0.98%	1.50E+11	0.96%	1.50E+11	0.90%	1.50E+11	0.98%

Table B.9: Cycle Count for Various Block Size That Can in a 512x4608 On-Chip Memory

Appendix C

Parameter Values Used in Experiments

C.1 Effect on Clock Frequency by Varying Pipeline Registers

Table C.1: Parameter Values for Compute Engine Shown in Table 4.3

Name	Value
k	57
Precision	64
NMax	20,000
BlockSizeDivk	114
AdderLatency	12
MultLatency	11
DivLatency	33
DDRWidth	64
DDRAddrWidth	24
DDRRowAddrWidth	13
DDRBurstLen	4
FIFOSize	16

C.2 Effect on Performance

C.2.1 Numbers of Processing Elements

Table C.2: Common Parameter Values for Compute Engines Shown in Table 5.1

Name	Value
Precision	32
NMax	20,000
AdderLatency	14
MultLatency	11
DivLatency	33
DDRWidth	64
DDRAddrWidth	24
DDRRowAddrWidth	13
DDRBurstLen	4
FIFOSize	16

Table C.3: Parameter Values for Each Compute Engine Shown in Table 5.1

k	MatrixBlock-SizeDivk	ExtraOnChip-RamBlock-InputPortDelay	ExtraOnChip-RamBlock-OutputPortDelay	ExtraOnChip-TopBlock-InputPortDelay	ExtraOnChip-TopBlock-OutputPortDelay
30	60	3	2	5	3
60	120	3	2	5	3
90	180	3	2	5	3
120	240	1	1	4	3
128	256	1	1	4	3
136	136	1	1	3	1
144	144	1	1	5	3

C.2.2 Floating Point Unit Latency

Table C.4: Common Parameter Values for Compute Engines Shown in Figure 5.2

Name	Value
k	120
Precision	32
NMax	20,000
MatrixBlockSizeDivk	240
MultLatency	11
DivLatency	33
DDRWidth	64
DDRAddrWidth	24
DDRRowAddrWidth	13
DDRBurstLen	4
FIFOSize	16

Table C.5: Parameter Values for Each Compute Engine Shown in Figure 5.2

AdderLatency	ExtraOnChip-RamBlock-InputPortDelay	ExtraOnChip-RamBlock-OutputPortDelay	ExtraOnChip-TopBlock-InputPortDelay	ExtraOnChip-TopBlock-OutputPortDelay
14	1	1	4	3
12	2	1	4	4
10	3	1	5	3
8	3	2	5	3

C.2.3 Block Size

Table C.6: Common Parameter Values for Compute Engines Shown in Figure 5.3 and Table 5.2

Name	Value
Precision	32
NMax	20,000
AdderLatency	14
MultLatency	11
DivLatency	33
DDRWidth	64
DDRAddrWidth	24
DDRRowAddrWidth	13
DDRBurstLen	4
FIFOSize	16

C.3 Performance and Power Consumption

C.3.1 Stratix III 3SL340F1760C3

Table C.7: Parameter Values for Single Precision Compute Engine Shown in Table 5.3

Name	Value
k	120
Precision	32
NMax	20,000
BlockSizeDivk	240
AdderLatency	12
MultLatency	11
DivLatency	33
DDRWidth	64
DDRAddrWidth	24
DDRRowAddrWidth	13
DDRBurstLen	4
FIFOSize	16
ExtraOnChipRamBlockInputPortDelay	2
ExtraOnChipRamBlockOutputPortDelay	1
ExtraOnChipTopBlockInputPortDelay	4
ExtraOnChipTopBlockOutputPortDelay	4

Table C.8: Parameter Values for Double Precision Compute Engine Shown in Table 5.4

Name	Value
k	57
Precision	64
NMax	20,000
BlockSizeDivk	114
AdderLatency	12
MultLatency	11
DivLatency	33
DDRWidth	64
DDRAddrWidth	24
DDRRowAddrWidth	13
DDRBurstLen	4
FIFOSize	16
ExtraOnChipRamBlockInputPortDelay	3
ExtraOnChipRamBlockOutputPortDelay	1
ExtraOnChipTopBlockInputPortDelay	6
ExtraOnChipTopBlockOutputPortDelay	4

C.3.2 Stratix II 2S180F1508C3

Table C.9: Parameter Values for Single Precision Compute Engine on Stratix II 2S180 Shown in Table 5.8

Name	Value
k	60
Precision	32
NMax	20,000
BlockSizeDivk	120
AdderLatency	12
MultLatency	11
DivLatency	33
DDRWidth	64
DDRAddrWidth	24
DDRRowAddrWidth	13
DDRBurstLen	4
FIFOSize	16
ExtraOnChipRamBlockInputPortDelay	3
ExtraOnChipRamBlockOutputPortDelay	1
ExtraOnChipTopBlockInputPortDelay	3
ExtraOnChipTopBlockOutputPortDelay	1

Table C.10: Parameter Values for Double Precision Compute Engine on Stratix II 2S180 Shown in Table 5.8

Name	Value
k	29
Precision	64
NMax	20,000
BlockSizeDivk	58
AdderLatency	12
MultLatency	11
DivLatency	33
DDRWidth	64
DDRAddrWidth	24
DDRRowAddrWidth	13
DDRBurstLen	4
FIFOSize	16
ExtraOnChipRamBlockInputPortDelay	2
ExtraOnChipRamBlockOutputPortDelay	1
ExtraOnChipTopBlockInputPortDelay	3
ExtraOnChipTopBlockOutputPortDelay	1

Bibliography

[1] Altera Corporation. Stratix III FPGA Architecture. http://www.altera.com/products/devices/stratix-fpgas/stratix-iii/overview/architecture/st3-why-use.html.

[2] XtremeData, Inc. http://www.xtremedatainc.com.

[3] SRC Computers, Inc. http://www.srccomp.com.

[4] Cray Inc. http://www.cray.com.

[5] Agility Design Solutions Inc. Handel-C. http://www.agilityds.com/products/c_based_products/dk_design_suite/handel-c.aspx.

[6] Mentor Graphics Corporation. Catapult Synthesis. http://www.mentor.com/products/esl/high_level_synthesis/catapult_synthesis/index.cfm.

[7] Open SystemC Initiative. http://www.systemc.org.

[8] Intel. Intel Math Kernel Library. http://www.intel.com/cd/software/products/asmo-na/eng/307757.htm.

[9] Intel. Intel Streaming SIMD Extensions 4 (SSE4) Instruction Set Innovation. http://www.intel.com/techology/architecture-silicon/sse4-instructions/index.htm.

[10] AMD. SSE5. http://developer.amd.com/cpu/SSE5/Pages/default.aspx.

[11] AMD. AMD Core Math Library. http://developer.amd.com/cpu/libraries/acml/Pages/default.aspx.

[12] Altera Corporation. http://www.altera.com/products/ip/ipm-index.html.

[13] Altera Corporation. DDR and DDR2 SDRAM High-Performance Controller MegaCore Functions. http://www.altera.com/products/ip/iup/memory/m-alt-hpddr2.html.

[14] Mentor Graphics. ModelSim PE http://www.model.com/products/products_pe.asp.

[15] The MathWorks. MATLAB http://www.mathworks.com/products/matlab.

[16] Intel. Intel Xeon Processor 5160. http://processorfinder.intel.com/Details.aspx?sSpec=SLABS.

[17] E. Anderson, Z. Bai, C. Bischof, S. Blackford, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, and D. Sorensen. *LAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA, third edition, 1999.

[18] L. S. Blackford, J. Demmel, J. Dongarra, I. Duff, S. Hammarling, G. Henry, M. Heroux, L. Kaufman, A. Lumsdaine, A. Petitet, R. Pozo, K. Remington, and R. C. Whaley. An updated set of basic linear algebra subprograms (BLAS). *ACM Trans. Math. Soft.*, 28(2):135–151, 2002.

[19] Altera Corporation. 11. Netlist optimizations and physical synthesis, Quartus II 8.0 handbook volume 2. Technical report, Altera Corporation, 2008. http://www.altera.com/literature/hb/qts/qts_qii52007.pdf.

[20] Vikash Daga, Gokul Govindu, Sridhar Gangadharpalli, V. Sridhar, and Viktor K. Prasanna. Efficient floating-point based block LU decomposition on FPGAs. In *Pro-*

ceedings of the International Conference on Engineering of Reconfigureable Systems and Algorithms, 2004.

[21] Micheal deLorimier and Andre DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays*, pages 75–85, 2005.

[22] H J G Diersch. Error Norm. `http://www1.wasy.de/deutsch/produkte/feflow/hilfe/general/theory/whitepapers/error_norms/enornorm.html`.

[23] Jack J. Dongarra, Iain S. Duff, Danny C. Sorensen, and Henk A. van der Vorst. *Numerical Linear Algebra for High-Performance Computers*. SIAM, Philadelphia, PA, 1998.

[24] William W. Hager. *Applied Numerical Linear Algebra*. Prentice Hall, Englewood Cliffs, NJ, 1988.

[25] A. R. Lopes and G. A. Constantinides. A high throughput FPGA-based floating point conjugate gradient implementation. *Lecture Notes in Computer Science*, (4943):75–86, 2008.

[26] Gerald R. Morris and Viktor K. Prasanna. Sparse matrix computations on reconfigurable hardware. *Computer*, 40(3):58–64, March 2007.

[27] Yousef Saad. SPARSKIT. `http://www-users.cs.umn.edu/~saad/software/SPARSKIT/sparskit.html`.

[28] Keith D. Underwood and K. Scott Hemmert. Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance. In *Proceedings of the 12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 219–228, 2004.

[29] Ling Zhuo and Viktor K. Prasanna. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field Programmable Gate Arrays*, pages 63–74, 2005.

[30] Ling Zhuo and Viktor K. Prasanna. High-performance and parameterized matrix factorization on FPGAs. In *Proceedings of the 2006 International Conference on Field Programmable Logic and Applications*, pages 1–6, 2007.