

Design Tradeoffs for BLAS Operations on Reconfigurable Hardware *

Ling Zhuo and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles, California 90089-2562, USA
{lzhuo, prasanna}@usc.edu

Abstract

Numerical linear algebra operations are key primitives in scientific computing. Performance optimizations of such operations have been extensively investigated and some basic operations have been implemented as software libraries. With the rapid advances in technology, hardware acceleration of linear algebra applications using FPGAs (Field Programmable Gate Arrays) has become feasible. In this paper, we propose FPGA-based designs for several BLAS operations, including vector product, matrix-vector multiply, and matrix multiply. By identifying the design parameters for each BLAS operation, we analyze the design tradeoffs. In the implementations of the designs, the values of the design parameters are determined according to the hardware constraints, such as the available area, the size of on-chip memory, the external memory bandwidth and the number of I/O pins. The proposed designs are implemented on a Xilinx Virtex-II Pro FPGA.

1 Introduction

Numerical linear algebra, particularly the solution of linear systems of equations, linear least square problems, eigenvalue problems and singular value problems, is fundamental to most scientific applications. As it is often the most computationally intensive part of such applications, the performance improvement of numerical linear algebra has always been of interest to researchers. Certain basic operations, such as the vector and matrix operations, have been implemented as libraries. Such libraries provide highly optimized routines for linear algebra operations [11].

Despite the abundant research in software acceleration of basic linear algebra operations, to the best of our knowledge, we are not aware of any of their hardware acceleration. Due to the low density of earlier hardware devices

used in embedded systems, few floating-point operators can be implemented on those devices. However, with the advances in technology, these devices have become much more powerful. In particular, a state-of-the-art Field Programmable Gate Array (FPGA) device contains millions of gates so that many floating-point units can be configured on it. In addition, the current FPGA fabrics provide large amount of on-chip memory as well as abundant I/O pins. Some researchers have suggested that FPGAs can be highly competitive with microprocessors with respect to both peak performance and sustained performance for linear algebra applications [16, 20]. Meanwhile, vendors have begun to use FPGAs for high performance computing. For example, Cray [3] and SRC Computers [15] have developed high end computers that employ FPGAs as application accelerators.

On the other hand, FPGAs also pose new challenges in implementing (floating-point) BLAS operations. Limitations in the available resources, such as the number of slices, the size of on-chip memory and the number of I/O pins, impose multiple constraints on architectural design. These constraints, as well as the inherent characteristics of BLAS operations, result in various design tradeoffs. In this work, we focus on the design tradeoff analysis for BLAS operations. We choose the vector product from level 1 BLAS, matrix-vector multiply from level 2 BLAS, and matrix multiply from level 3 BLAS. To the best of our knowledge, the design tradeoffs analyzed in this paper have not been discussed for level 1 or level 2 BLAS operations on FPGAs. For level 3 BLAS operations, we extend our work in [20]. Although our work is not dependent on the data representation, we consider IEEE-754 double-precision numbers throughout the paper, as this representation is widely used in scientific computations [8].

For each BLAS operation, we analyze its inherent characteristics, such as the number of floating-point operations and I/O operations required, and identify various design parameters. By exploring the design space, we analyze the design tradeoffs among area, latency and size of on-chip memory needed by the design. Based on the analysis, we

*Supported by the United States National Science Foundation under award No. CCR-0311823 and in part by award No. ACI-0305763.

propose an FPGA-based design for each BLAS operation so as to achieve the optimal latency under the given hardware resources. The optimizations we employ are generic. Thus, they can be applied to various FPGA devices.

For both the vector product and the matrix-vector product, we propose a tree-based design. For the matrix multiply, we employ a linear array of Processing Elements (PEs). Block algorithms are employed to reduce the requirements on the memory bandwidth. The design for each BLAS operation is characterized through various parameters, such as the number of floating-point units and the block size. The parameters can be tuned according to various hardware resource constraints, which include the available area, the size of on-chip memory, and the number of I/O pins. We implemented our proposed designs using Xilinx ISE 6.2i [18], with Xilinx Virtex-II Pro XC2VP40 as our target device. For all of the designs, the area increases linearly with the number of floating-point units used. Moreover, the latency decreases almost proportionally with the number of floating-point units, if adequate memory bandwidth is provided. On the other hand, the size of on-chip memory needed and the memory bandwidth required by the designs are determined by both the number of floating-point units and the block size.

The rest of the paper is organized as follows. Section 2 introduces the background and the related work. Section 3 proposes our designs for the BLAS operations. Section 4 presents and analyzes the performance of our designs. Section 5 concludes the paper.

2 Background and Related Work

Scientific Computing on FPGAs

Field-Programmable Gate Arrays (FPGAs) are a form of programmable logic. They offer design flexibility like software, but with time performance closer to Application Specific Integrated Circuits (ASICs) [2]. An FPGA device consists of an array of logic blocks (*slices*) whose functionality is determined through programmable configuration bits. These logic blocks are connected using a set of routing resources that are also programmable. Thus, mapping an appropriate design to FPGAs consists of determining the functions to be computed by the logic blocks, and using the configurable routing resources to connect the blocks.

State-of-the-art FPGA devices also contain large amount of on-chip memory, including the memory that can be realized by the logic blocks and the embedded memory blocks called Block RAMs (BRAMs). Besides the on-chip memory, FPGA-based designs also have access to external memory through I/O pins. Throughout this paper, our analysis is based on this generic FPGA and (external) memory computing model. We assume that for each operation, the input data are initially stored in the external memory, and the size

of the data can be larger than the on-chip memory.

In the past, FPGAs were mainly used for integer and fixed-point applications. However, with the advances in technology, FPGAs are now feasible to be used for a much broader range of applications, including those requiring floating-point operations. Floating-point units with various precisions have been designed [6]. FPGA-based architectures also have been proposed for computationally intensive applications [14].

Recently, high end computers have been built with FPGAs serving as application accelerators. Examples include the SRC MAPstation [15] and the Cray XD1 [3]. These computers combine general-purpose computing systems with reconfigurable hardware. The general-purpose microprocessors and the FPGAs share memory with each other. In these computers, FPGA-based designs have access to large external memory and high bandwidth interconnect.

BLAS

The set of Basic Linear Algebra Subprograms, which is commonly referred to as BLAS [11], has been used in a wide range of software including LINPACK [4]. BLAS provide building block routines for performing basic vector and matrix operations. Optimizations for the BLAS library on general-purpose processors include loop unrolling, register blocking and cache blocking [17]. Since many of the optimizations are platform specific, ATLAS was proposed which automatically generates and optimizes numerical software for processors with deep memory hierarchies and pipelined functional units [17].

Although BLAS and ATLAS have been widely accepted and used by the scientific computing community, building libraries of hardware implementations for linear algebra operations has not been well studied. There have been some researches on FPGA-based implementations of linear algebra applications. However, some of them consider fixed-point arithmetic only [1, 10]; while others only discuss one floating-point BLAS operation [20, 5]. The only prior work that considers FPGA-based implementations of operations from all BLAS levels is conducted by Underwood et al [16]. In that work, the authors examine the potential capacity of FPGAs in performing floating-point BLAS operations, and compare the computing capacity of FPGAs with general-purpose processors. Although some architectures are proposed for BLAS operations in [16], the performance results are mainly based on estimation. Moreover, in [16], only the number of I/O pins is considered as a constraint. However, as we show in Section 4, the available area and on-chip memory of the device can also constrain the performance of the designs. In contrast to [16], we provide design tradeoffs analysis based on multiple hardware resource constraints. In addition, we discuss techniques to reduce the requirement on memory bandwidth.

3 FPGA-based Designs for BLAS

In our work, we adopt a parameterized approach for basic linear algebra operations on FPGAs. For each BLAS operation, we identify the design parameters and analyze the resulting design tradeoffs. Note that the analysis in this section does not consider the implementation issues, such as the routing complexity and the target device. We employ the following mathematical notation throughout the paper:

- u, v, x, y : vectors of length n . Each vector can be either a column vector or a row vector, according to the context.
- A, B, C : $n \times n$ matrices, with elements a_{ij}, b_{ij}, c_{ij} ($i, j = 0, \dots, n-1$)
- α : pipeline delay of the floating-point adder
- w, w_j : number of bits in a floating-point word and a column index, respectively
- k, l : number of floating-point multiplications and floating-point additions that can be performed in each clock cycle, respectively
- bw : number of data words (double precision floating-point) that are input/output in each clock cycle
- m : total size of on-chip memory needed by the design
- T : number of clock cycles needed to complete the BLAS operation
- T_{comp} : number of clock cycles needed for floating-point computations
- $T_{I/O}$: number of clock cycles needed for input/output

3.1 Level 1 BLAS

Level 1 BLAS are the specification and implementation of subprograms for scalar and vector operations. In this paper, we consider product of a row vector and a column vector, which can be formulated as:

$$u \times v = \sum_{i=0}^{n-1} u_i v_i$$

A. BLAS Operation Analysis: For the vector product, the minimum number of I/O operations is $2n + 1$, and n floating-point multiplications as well as n floating-point additions need to be performed. We can easily identify k and l as the design parameters, because they affect T_{comp} . Since there is no data reuse in this operation, no on-chip memory is needed by the design. To reduce the latency, we can overlap T_{comp} and $T_{I/O}$. Furthermore, we can overlap the floating-point multiplications and additions. Therefore, the lower bound on the latency of the operation (in cycles) is derived as follows:

$$T \geq \max(T_{comp}, T_{I/O}) \geq \max(\max(\frac{n}{k}, \frac{n}{l}), \frac{2n+1}{bw})$$

B. Architecture: To achieve the lower bound, $k = l$ and $k \approx \frac{bw}{2}$. Thus we propose an architecture which consists of identical number of floating-point adders and multipliers. The floating-point adders and multipliers are pipelined so that additions, multiplications and I/O operations can be overlapped. Moreover, pipelining for the adder and multiplier results in high clock speed.

During each clock cycle, each multiplier reads one element from each of the two vectors, and multiplies these two floating-point numbers. An adder tree is employed to sum up the outputs of the multipliers. When $k < n$, we need an additional adder to sum up the outputs of the adder tree. Thus, we use k adders, including the $k - 1$ adders in the adder tree and the additional adder. If we ignore the clock cycles used to fill the multiplier and the adder pipelines, the effective latency of the architecture is $T = \frac{n}{k}$.

The adder tree in the architecture yields one output in each clock cycle. Thus, the task of the additional adder is to reduce sets of sequentially delivered floating-point values. However, the pipelining in the floating-point adder can cause read-after-write data hazards during the reduction. This problem has been investigated in [19] and [13]. Therefore, we replace the additional adder outside the adder tree using a reduction circuit proposed in [13]. This reduction circuit can reduce a series of inputs of arbitrary length. It employs two floating-point adders and $\Theta(\lg(s))$ buffer size for s sequential inputs. Let $T_{red}(s)$ denote the time to complete the reduction. As shown in [13], $T_{red}(s) = \Theta(s)$. The characteristics of the reduction circuit are presented in Section 4.

The architecture for the vector product is shown in Figure 1. The effective latency of the design is

$$T = \frac{n}{k} + T_{red}(\frac{n}{k}) = \Theta(\frac{n}{k}) \quad (1)$$

To achieve this latency, the design performs $2k + 1$ I/O operations during each clock cycle, that is, $bw = 2k + 1$. The number of I/O pins used is $(2k + 1)w$.

3.2 Level 2 BLAS

In level 2 BLAS, we select the matrix vector multiply, which is formulated as:

$$y = Ax, y_i = \sum_{j=0}^{n-1} a_{ij} x_j \quad (i = 0, 1, \dots, n-1)$$

A. BLAS Operation Analysis: In the matrix-vector multiply, the total number of floating-point operations is $2n^2$.

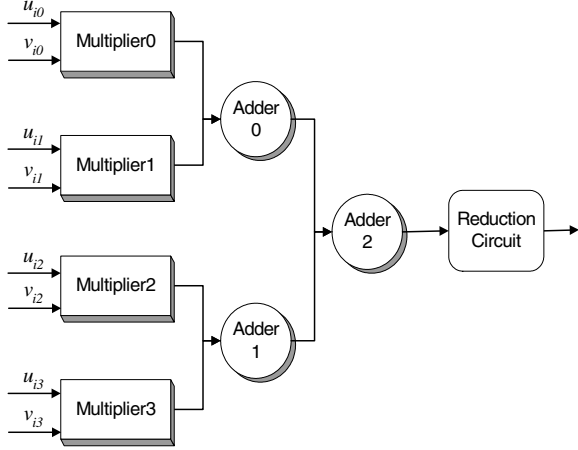


Figure 1. Architecture for vector product ($k = 4$)

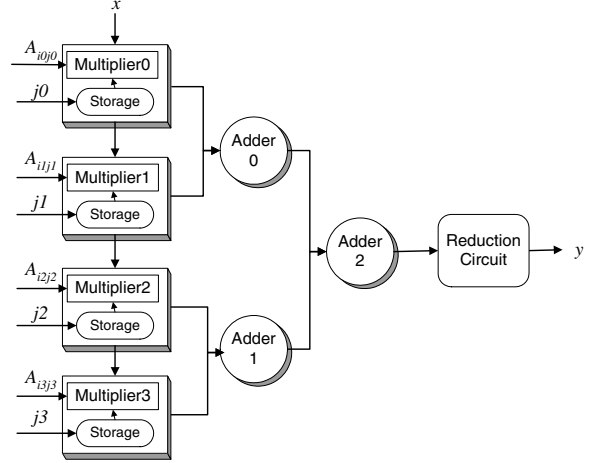


Figure 2. Architecture for matrix-vector product ($k = 4$)

As in the case of the vector product, we identify k and l as the design parameters. However, in this operation, each element in x can be reused n times. If we do not store any element of x in the architecture, the total number of I/O operations is $2n^2$.

To partially reduce the I/O costs, we use block matrix-vector multiply. Thus, a new parameter, b , needs to be introduced. Without loss of generality, we assume n is a multiple of b . In the blocked version, A is partitioned into $\frac{n}{b}$ blocks of columns, with each block of size $n \times b$. Similarly, x is partitioned into $\frac{n}{b}$ blocks of size $b \times 1$. We denote the blocks using A^g and x^g , respectively ($g = 0, 1, \dots, \frac{n}{b} - 1$). For each block matrix-vector multiply $A^g \times x^g$, A^g and x^g need to be read from and the results need to be written to the external memory. Therefore, the total number of I/O operations with block size b equals $\frac{n}{b} \times (nb + b + n) = n^2 + \frac{n^2}{b} + n$.

We set $k = l$ to reduce T_{comp} . The lower bound on the latency of the operation can be derived as follows:

$$T \geq \max(T_{comp}, T_{I/O}) \geq \max\left(\frac{n^2}{k}, \frac{n^2 + \frac{n^2}{b} + n}{bw}\right)$$

The above equation shows a design tradeoff between the block size and the I/O time, even though asymptotically the I/O time is $\frac{n^2}{bw}$. The smaller is the block size, the more I/O operations are required; and vice versa. Another tradeoff exists between the storage size and the data access time. If we only store one copy of x^g in the architecture, $m = b$. However, since each BRAM on FPGAs only has two I/O ports, reading k distinct values from the BRAM takes multiple clock cycles. On the other hand, if x^g is duplicated at each multiplier, each multiplier can access the needed data in each clock cycle. However, the size of on-chip memory

needed by the design (m) increases to kb .

B. Architecture: Based on the design tradeoffs, we propose an architecture for the matrix-vector multiply, as shown in Figure 2. This architecture is very similar to the architecture for the vector product, except that each multiplier is attached to a local storage. The local storage stores a copy of x^g . Initially, the storage is loaded with x^0 . During the computation, the multiplier reads one element from A^g in each clock cycle, then uses the column index to find the corresponding x^g element from its local storage, and finally multiplies these two numbers. To reduce the latency, the initialization of block x^g is overlapped with the computations of $A^{g-1} \times x^{g-1}$ ($g = 1, \dots, \frac{n}{b} - 1$).

When $b > k$, one row in A^g is further partitioned into $\frac{b}{k}$ sub-rows which are fed into the architecture in order. A reduction circuit is used in the architecture to accumulate the sums of the sub-rows.

To generate the final y , we also need to accumulate the results generated by $A^0 \times x^0, A^1 \times x^1, \dots, A^{\frac{n}{b}-1} \times x^{\frac{n}{b}-1}$. For each y element, such intermediate results are generated every n clock cycles. Since the typical value of α is less than 20 and is usually much smaller than n , an additional floating-point adder suffices for such accumulation.

The effective latency of our design is:

$$\begin{aligned} T &= (b + k) + \left(\frac{n \times b}{k} + k\right) \times \left(\frac{n}{b} - 1\right) + \frac{n \times b}{k} + T_{red}\left(\frac{b}{k}\right) \\ &= \frac{n^2}{k} + \frac{nk}{b} + b + T_{red}\left(\frac{b}{k}\right) \\ &= \Theta\left(\frac{n^2}{k}\right) \end{aligned} \tag{2}$$

To achieve the latency, the design needs to perform $bw \approx k + \frac{k}{b}$ floating-point I/O operations during each clock cycle. The number of I/O pins used is $k(w + w_j) + 2w$ because:

each multiplier reads one element of A and its column index in each clock cycle; the first multiplier reads one element of x per clock cycle for initialization. Note that we can reduce the number of I/O pins used by using a counter to keep track of the column index. The total size of on-chip memory needed by the design is $m = kb$.

3.3 Level 3 BLAS

The last BLAS operation we consider in this paper is dense matrix multiply, from level 3 BLAS. This BLAS operation is formulated as:

$$C = AB, c_{ij} = \sum_{q=0}^{n-1} a_{iq}b_{qj} \quad (i, j = 0, 1, \dots, n-1)$$

A. BLAS Operation Analysis: Each element of A and B can be used n times, and the total number of floating-point operations is $2n^3$. Since there is lots of data reuse in the matrix multiply, m , the size of on-chip memory needed by the design becomes an important design parameter. We use *I/O complexity* to refer to the total number of I/O operations performed by an algorithm. It has been proven [7] that the I/O complexity of any implementations of the usual matrix multiply algorithm is $\Omega(\frac{n^3}{\sqrt{m}})$, when $\Theta(1) \leq m \leq \Theta(n^2)$.

Again we set $k = l$ to optimize T_{comp} . The lower bound on the latency of matrix multiply is:

$$T \geq \max(T_{comp}, T_{I/O}) \geq \max(\frac{n^3}{k}, \frac{n^3/\sqrt{m}}{bw}) \quad (m \leq n^2) \quad (3)$$

Thus, tradeoffs exist among the number of PEs, the total size of on-chip memory, and the required memory bandwidth. Based on Equation 3, following cases arise for various values of k and m :

- **Case 1:** $k \geq n, m = n^2$
- **Case 2:** $k \geq n, m < n^2$
- **Case 3:** $k < n, m = n^2$
- **Case 4:** $k < n, m < n^2$

In our prior work, we have proposed an architecture for Case 1: $\frac{n}{s}$ ($1 \leq s \leq n$) PEs are connected in a linear array [20]. Each PE contains one floating-point multiplier, one floating-point adder and local storage of s words. During each clock cycle, n PEs are performing steps of the matrix multiply, while the other PEs wait for data. The required memory bandwidth is 3 words per clock cycle. The total size of on-chip memory needed is $m = \frac{n^2}{s} \times s = n^2$. Since $1 \leq s \leq n$, this design considers the case $k \geq n$. However, for FPGA-based implementations of floating-point matrix

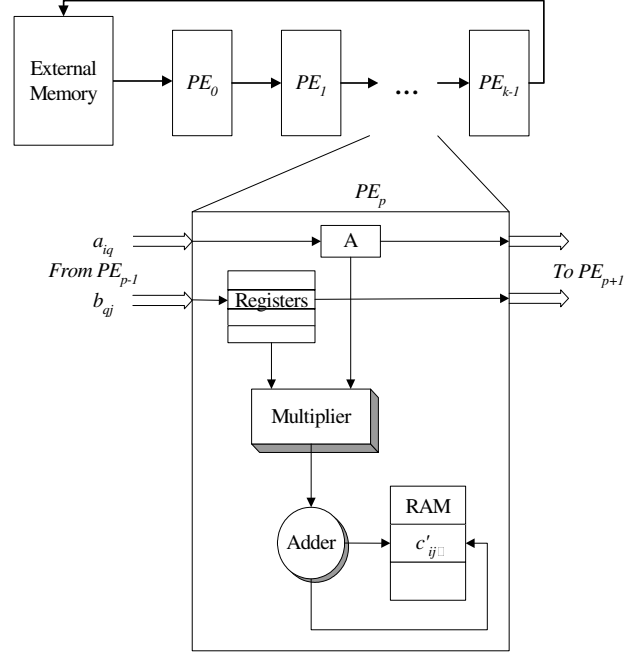


Figure 3. Architecture for matrix multiply

multiply, the condition $k \geq n$ is usually hard to satisfy. Due to the complexity of the floating-point units, only tens of floating-point units can be configured on an FPGA device, while n can be large. Moreover, n^2 can be much larger than the total size of on-chip memory on FPGAs. Hence the requirement $m = n^2$ will be difficult to meet. Thus, we do not consider Case 2 and Case 3 in this paper.

B. Architecture: In this paper, we extend our previous work [20] to Case 4: k can be smaller than n , and m can be smaller than n^2 . Another contribution of this design is that we take the memory bandwidth into account in our analysis. For a given k and m , we are able to minimize the required memory bandwidth.

The architecture is shown in Figure 3, and the algorithm is shown in Figure 4. In the architecture, there are k PEs connected in a linear array. Each PE consists of one floating-point multiplier, one floating-point adder, local storage of size $\frac{m}{k}$, and $\frac{2\sqrt{m}}{k}$ registers. The PEs are labeled from left to right, as $PE_0, PE_1, \dots, PE_{k-1}$. PE_0 reads A and B from the external memory. PE_{k-1} writes the final elements of C to the memory.

In the design, we are actually performing block matrix multiply where the block size is $\sqrt{m} \times \sqrt{m}$. The blocks are denoted as A^{gz} and B^{zh} , where $g, z, h = 0, 1, \dots, \frac{n}{\sqrt{m}} - 1$. Without loss of generality, we assume n is a multiple of \sqrt{m} , and \sqrt{m} is a multiple of k .

In our design, for each block matrix multiply $A^{gz} \times B^{zh}$, A^{gz} is read in column-major order, while B^{zh} is read in

row-major order. PE_p , $p = 0, \dots, k - 1$, is in charge of computing the columns $p, (k + p), \dots, ((\frac{\sqrt{m}}{k} - 1)k + p)$ of C^{gh} . Before the computation starts, the first row of B^{zh} is read into the architecture. As these \sqrt{m} numbers traverse the linear array, PE_p , $p = 0, \dots, k - 1$, stores the p th, $(k + p)$ th, $\dots, ((\frac{\sqrt{m}}{k} - 1)k + p)$ th numbers into its registers. Afterwards, every $\frac{\sqrt{m}}{k}$ clock cycles, one element of A^{gz} and one element of B^{zh} are read into the architecture. As a_{iq} passes through a PE, it is multiplied with every element of B stored in the PE whose row index is q . The intermediate results for C^{gh} are denoted as c'_{ij} , and are stored in the local storage of the PEs.

During the computation, the value of c'_{ij} is updated every $\frac{m}{k}$ cycles. If $\frac{m}{k} > \alpha$, there is no data hazard. Otherwise we need to either decrease k to increase $\frac{m}{k}$, or replace the floating-point adder in each PE with a reduction circuit.

As in the matrix-vector multiply, the intermediate results of C need to be accumulated. The effective latency for each block matrix multiply, T' , equals $(\frac{\sqrt{m}}{k})^3 = \frac{m\sqrt{m}}{k}$. Thus, every T' clock cycles, an intermediate result of C is generated. Since T' is usually much larger than α , we can simply use a floating-point adder for the accumulation.

```

{block matrix multiply}
for PE p = 0 to k - 1 (in parallel) do
  if p = j mod k then
    copy  $b_{qj}$  to a register
  end if
  if q = 1 (the first row of B block) then
    shift  $b_{qj}$  right to  $PE_{p+1}$ 
  else
    shift  $b_{qj}, a_{iq}$  right to  $PE_{p+1}$ 
    for every  $b_{qj'}$  in the registers do
       $c'_{ij'} \leftarrow c'_{ij'} + a_{iq} \times b_{qj'}$ 
    end for
  end if
end for

```

Figure 4. Algorithm for matrix multiply

In our design, $\frac{n^3}{\sqrt{m}}$ block matrix multiplies are performed. Thus, the total latency of our design is:

$$\begin{aligned}
T &= \sqrt{m} + \left(\frac{n}{\sqrt{m}}\right)^3 \times \frac{m\sqrt{m}}{k} \\
&= \sqrt{m} + \frac{n^3}{k} \\
&= \Theta\left(\frac{n^3}{k}\right)
\end{aligned} \tag{4}$$

To achieve the latency, 2 words are read from and 1 word is written to the external memory every $\frac{\sqrt{m}}{k}$ cycles. Thus, $bw = \Theta(\frac{k}{\sqrt{m}})$. According to [7] and Equation 4, the memory bandwidth required by our design is minimum for a given m . The number of I/O pins needed is $3w$.

4 Performance Analysis and Discussion

4.1 Experimental Setup

To understand the tradeoffs, we used Xilinx ISE 6.2i [18] and Mentor Graphics ModelSim 5.7 [12] development tools in our experiments. Our target device is Xilinx Virtex-II Pro XC2VP40 [18], which contains 19392 slices, about 3 Mbit on-chip memory and 804 I/O pins. This is a typical state-of-art FPGA device at the time of writing.

The building blocks used in our designs include floating-point adder, floating-point multiplier, and reduction circuit. In all of the designs, the implementation of the building blocks has no effect on the architecture or the control logic of the designs. These blocks can be independently designed and then plugged into any of our designs, with no or few modifications to the design. Table 1 gives the characteristics of these blocks, whose implementation details can be found in [6, 13]. Note that our floating-point units comply with the IEEE-754 double-precision format [8].

In our experiments, we first determine the range for each design parameter based on the hardware resource constraints. Then we vary the values of the parameters, and measure the area and clock speed of the designs after place and route. The latency and the required memory bandwidth are calculated based on the clock speed and the analysis in Section 3. Suppose the clock speed is f MHz, the latency is calculated as $\frac{T}{f} \times 10^{-6}$ second; the required memory bandwidth is $bw \times 64 \times f$ Mb/s. Next, we discuss the tradeoffs in determining the values of the parameters for each operation. In all the experiments, the problem size n is set as 2048, so that the input data for matrix-vector multiply and matrix multiply cannot be stored in the on-chip memory of the FPGA device.

4.2 Performance Analysis

Level 1 BLAS For the vector product, the only parameter is k , the number of floating-point multiplications that can be performed in each clock cycle. Since each multiplication needs two 64-bit numbers, the range of k is determined by the number of available I/O pins. For the target device, $k \leq 5$. In the design, the control logic occupies less than 5% of the total area. Thus, as shown in Figure 5, when k increases from 1 to 5, the area of the design increases linearly. The clock speed of the design is limited by that of the reduction circuit, and remains 170 MHz (Table 1). Hence the latency of the design decreases proportionally as k increases. The memory bandwidth required by the design also increases linearly with k (figure not shown due to space limitation). Thus, k causes tradeoff among area, latency and required memory bandwidth. Larger k leads to smaller latency, but requires more area and higher memory bandwidth.

Table 1. Characteristics of 64-bit Floating-Point Units and Reduction Circuit

	Adder	Multiplier	Reduction Circuit
Number of Pipeline Stages	19	12	-
Number of Slices	933	910	3313
Achievable Clock Speed(MHz)	200	205	170

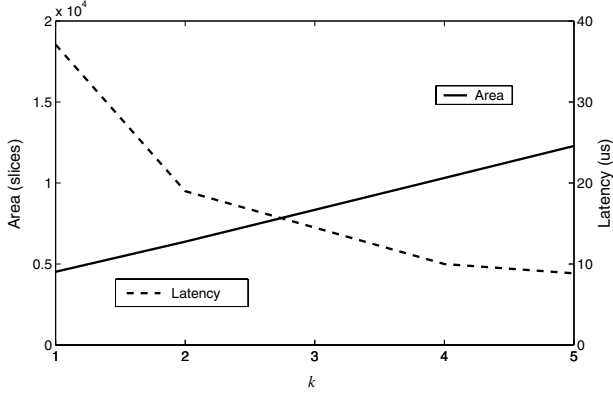


Figure 5. Area and latency for vector product ($n = 2048$)

Level 2 BLAS For the matrix-vector multiply, we need to tune both k and b . The range of k is determined by the total size of available area and the area of the floating-point adders, the floating-point multipliers and the reduction circuit. For the target device, $k \leq 8$. The range of b is determined by the size of on-chip memory on the FPGA device. For simplicity, we set the storage constraint as 1 Mbit, and vary b from 256 to 2048 words. We use BRAMs on the FPGAs as the local storage in the design. Note that BRAMs occupy few additional slices. The area of the design increases linearly with k , similar as in Figure 5. According to Equation 2, T depends on both k and b . However, Figure 6(a) shows that the latency is primarily determined by k . When k is fixed, the latency varies little as b increases. This is because when $n \gg k$ and $b \gg k$, $\frac{nk}{b} + b + T_{red}(\frac{b}{k})$ is negligible compared to $\frac{n^2}{k}$.

Each point in Figure 6(a) has a corresponding point in Figure 6(b) which indicates the memory bandwidth required to achieve the optimal performance. Figure 6(b) shows that the required memory bandwidth increases linearly with k . Thus, as in the case of the vector product, k causes tradeoffs among area, latency and required memory bandwidth.

Note that with current FPGA fabrics, k is usually in the order of tens, while n can be much larger. Therefore, although block matrix-vector multiply can reduce the number of memory accesses, a large block size does not decrease

the latency or the required memory bandwidth. On the other hand, the total size of on-chip memory needed in the design increases with k as well as b (figure not shown due to space limitation). Thus, when determining the value of b , a small block size is preferred.

Level 3 BLAS In the design for the matrix multiply discussed in Section 3, k is actually equal to the number of PEs. The maximum value of k is determined by the total size of available area and the area of the PEs. Each PE contains one floating-point adder, one floating-point multiplier, some BRAMs and a control unit. In contrast to the control logic of the entire design, this unit is in charge of the routing and the control circuitry inside one PE. Our experiments show that the control unit occupies less than 30% of the area of the PE. For the target device, $k \leq 8$. As in the other two design, the area of the design increases linearly with k , as shown in Figure 7(a).

Different from the other two designs, the size of on-chip memory needed by the design, m , is a design parameter for matrix multiply. $\sqrt{m} \times \sqrt{m}$ is the block size for block matrix multiply. In our experiments, we vary m from 16^2 words to 128^2 words.

Since the reduction circuit is not used in this design, the clock speed of the design is not limited by it. Instead, the clock speed of the design decreases as k increases due to the increasing routing complexity. However, Figure 7(b) shows that the latency still decreases proportionally with k . This is because the decrease in the clock speed is not as fast as the decrease in T . In particular, when k increases from 1 to 8, the degradation in the clock speed is less than 10%. Figure 7(b) also shows that the latency is not affected by \sqrt{m} . According to Equation 4, this is because when $n \gg k$ and $n \gg \sqrt{m}$, \sqrt{m} is negligible compared to $\frac{n^3}{k}$. However, the required memory bandwidth is dependent on both k and m , as shown in Figure 7(c). The larger the k , the higher the required memory bandwidth. On the other hand, larger \sqrt{m} reduces the requirement on the memory bandwidth.

As in the case of the other two operations, k causes trade-off among area, latency and required memory bandwidth for matrix multiply. Another tradeoff exists between the required on-chip memory and the required memory bandwidth.

In our experiments, using 18428 slices, 1 Mbit of on-chip memory, 192 I/O pins and a memory bandwidth of 2.1 Gb/s, our design for 64-bit matrix multiply can

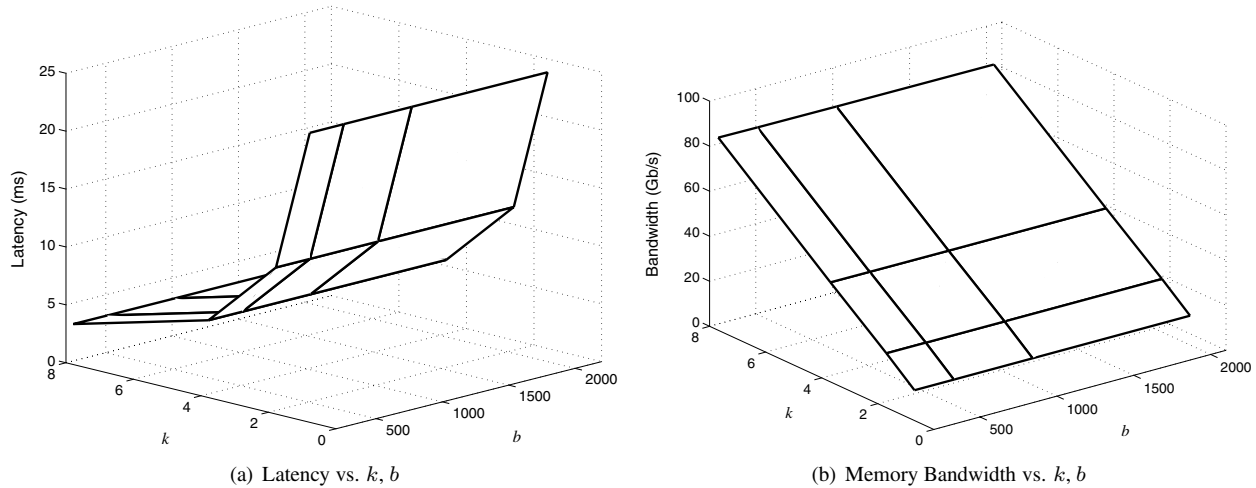


Figure 6. Performance analysis for matrix-vector multiply ($n = 2048$)

achieve 2.8 GFLOPS. This performance compares favorably with general-purpose processors. For example, a Xeon processor-based platform at 3.2 GHz with 1 MB L3 cache achieves 5.5 GFLOPS performing 64-bit matrix multiplication, while a 3-GHz Pentium 4 processor with 512 KB L2 cache achieves 5.0 GFLOPS [9]. These numbers are obtained by executing Intel Math Kernel Library [9]. This library not only employs common software optimizations such as loop unrolling and cache blocking, but also exploits the features of the Intel processors through specific optimizations. Note that our design is not optimized with respect to area or clock speed. In addition, no device specific optimizations were performed in our design.

We next compare the GFLOPS performance of our design for matrix multiply with the peak performance of the target device. In an ideal situation, the control logic occupies no slices; each floating-point unit (adder/multiplier) performs one floating-point operation during every clock cycle. A Xilinx Virtex-II Pro XC2VP40 can be configured to include at most 10 floating-point adders and 10 floating-point multipliers; the maximum clock speed of the design is 200 MHz. These estimations are based on the floating-point units in Table 1. Therefore, the peak performance of the device is $2 \times 10 \times 200$ MFLOPS = 4 GFLOPS. Our design achieves 70% of the peak performance, using only 33% of the on-chip memory.

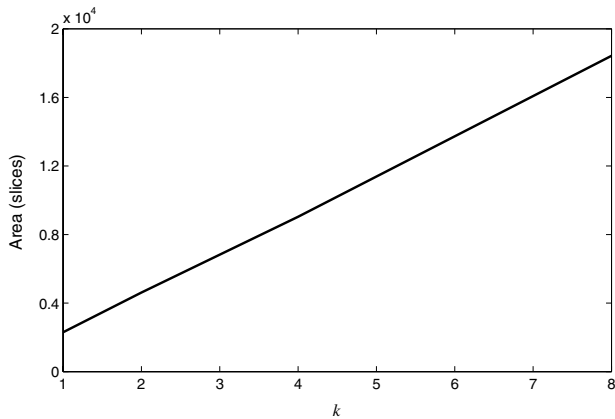
5 Conclusion

We have proposed optimized FPGA-based implementations for the vector product, matrix-vector multiply and matrix multiply. Various design parameters were identified for each BLAS operation, and the resulting design trade-offs were analyzed. The parameters include the number of

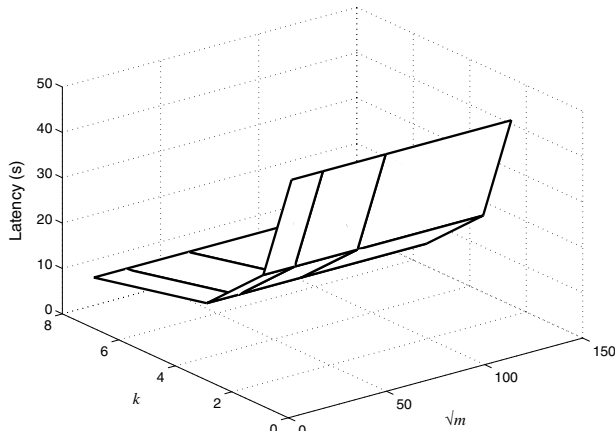
floating-point units, the size of on-chip memory needed by the design and the block size. We implemented our designs on a Xilinx Virtex-II Pro XC2VP40 FPGA device. The performance results show that using more floating-point units in a design leads to smaller latency, but requires more area and higher memory bandwidth. The block size causes trade-offs between the size of on-chip memory needed and the required memory bandwidth for matrix-vector multiply and matrix multiply. However, the block size has little effect on the latency of these two operations. In all of our proposed designs, the performance increases proportionally with the available hardware resources. We are now optimizing the floating-point units to further improve the GFLOPS performance of the designs. In the future, we plan to implement our designs for BLAS library on FPGA-augmented computers, such as Cray XD1 [3] and the MAPstation of SRC [15]. We also plan to extend our FPGA-based linear algebra library, by proposing designs for more linear algebra applications.

References

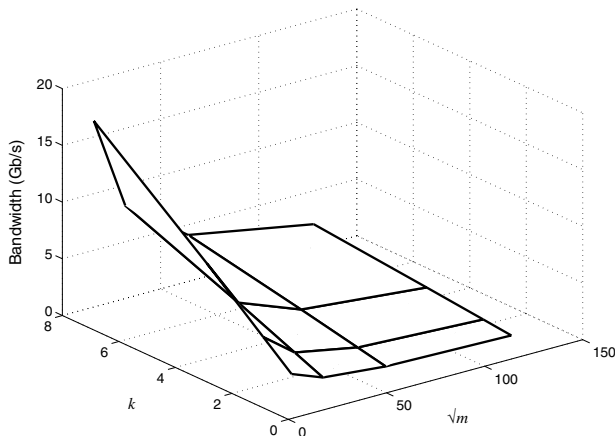
- [1] D. Benyamin, W. Luk, and J. Villasenor. Optimizing FPGA-based Vector Product Designs. In *Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines*, Napa Valley, CA, April 1999.
- [2] K. Compton and S. Hauck. Reconfigurable Computing: A Survey of Systems and Software. *ACM Computing Surveys*, 34(2):171–210, June 2002.
- [3] Cray Inc. <http://www.cray.com/>.
- [4] J. Dongarra, J. Bunch, C. Moler, and G. Stewart. *LINPACK Users' Guide*. Society for Industrial and Applied Mathematics, 1979.
- [5] Y. Dou, S. Vassiliadis, G. Kuzmanov, and G. Gaydadjiev. 64-bit Floating-Point FPGA Matrix Multiplication. In



(a) Area vs. k



(b) Latency vs. k, \sqrt{m}



(c) Memory Bandwidth vs. k, \sqrt{m}

Figure 7. Performance analysis for matrix multiply ($n = 2048$)

- Proc. of the 13th International Symposium on Field Programmable Gate Arrays*, California, USA, February 2005.
- [6] G. Govindu, L. Zhuo, S. Choi, and V. K. Prasanna. Analysis of High-Performance Floating-Point Arithmetic on FPGAs. In *Proc. of the 11th Reconfigurable Architectures Workshop*, New Mexico, USA, April 2004.
 - [7] J. Hong and H. Kung. I/O Complexity: The Red Blue Pebble Game. In *Proc. of ACM Symposium on Theory of Computing*, Wisconsin, USA, May 1981.
 - [8] Institute of Electrical and Electronics Engineers. *IEEE 754 Standard for Binary Floating-Point Arithmetic*. IEEE, 1984.
 - [9] Intel. <http://www.intel.com>.
 - [10] J. W. Jang, S. Choi, and V. K. Prasanna. Area and Time Efficient Implementation of Matrix Multiplication on FPGAs. In *Proc. of The First IEEE International Conference on Field Programmable Technology*, California, USA, December 2002.
 - [11] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic Linear Algebra Subprograms for FORTRAN usage. *ACM Transaction on Mathematical Software*, 5(3):308–323, 1979.
 - [12] Mentor Graphics Corp. <http://www.mentor.com/>.
 - [13] G. R. Morris, L. Zhuo, and V. K. Prasanna. High-Performance FPGA-Based General Reduction Methods. In *Proc. of 10th IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2005.
 - [14] R. Scrofano and V. K. Prasanna. Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware. In *Proc. of International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2004.
 - [15] SRC Computers, Inc. <http://www.srccomp.com/>.
 - [16] K. D. Underwood and K. S. Hemmert. Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance. In *Proc. of 2004 IEEE Symposium on Field-Programmable Custom Computing Machines*, California, USA, April 2004.
 - [17] R. C. Whaley, A. Petitet, and J. J. Dongarra. Automated empirical optimization of software and the ATLAS project. *Parallel Computing*, 27(1–2):3–35, 2001.
 - [18] Xilinx Incorporated. <http://www.xilinx.com>.
 - [19] L. Zhuo, G. R. Morris, and V. K. Prasanna. Designing Scalable FPGA-Based Reduction Circuits Using Pipelined Floating-Point Cores. In *Proc. of the 12th Reconfigurable Architectures Workshop*, Colorado, USA, April 2005.
 - [20] L. Zhuo and V. K. Prasanna. Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs. In *Proc. of the 18th International Parallel & Distributed Processing Symposium*, New Mexico, USA, April 2004.