Sparse Matrix-Vector Multiplication on FPGAs [*]

Ling Zhuo and Viktor K. Prasanna
Department of Electrical Engineering
University of Southern California
Los Angeles, California, 90089-2560 USA
{lzhuo, prasanna}@usc.edu

ABSTRACT

Floating-point Sparse Matrix-Vector Multiplication (SpMXV) is a key computational kernel in scientific and engineering applications. The poor data locality of sparse matrices significantly reduces the performance of SpMXV on general-purpose processors, which rely heavily on the cache hierarchy to achieve high performance. The abundant hardware resources on current FPGAs provide new opportunities to improve the performance of SpMXV. In this paper, we propose an FPGA-based design for SpMXV. Our design accepts sparse matrices in Compressed Row Storage format, and makes no assumptions about the sparsity structure of the input matrix. The design employs IEEE-754 format double-precision floating-point multipliers/adders, and performs multiple floating-point operations as well as I/O operations in parallel. The performance of our design for SpMXV is evaluated using various sparse matrices from the scientific computing community, with the Xilinx Virtex-II Pro XC2VP70 as the target device. The MFLOPS performance increases with the hardware resources on the device as well as the available memory bandwidth. For example, when the memory bandwidth is 8 GB/s, our design achieves over 350 MFLOPS for all the test matrices. It demonstrates significant speedup over general-purpose processors particularly for matrices with very irregular sparsity structure. Besides solving SpMXV problem, our design provides a parameterized and flexible tree-based design for floating-point applications on FPGAs.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms Implemented in Hardware*; F.2.1 [**Analysis of Algorithms and Problem Complexity**]: Numerical Algorithms and Problems—*Computations on matrices*

General Terms

Algorithms, Performance, Design

Keywords

FPGA, Floating-Point, Sparse Matrix, Reconfigurable Architecture, High Performance

1. INTRODUCTION

FPGAs have long been used for a number of integer and fixed-point applications, such as the signal processing applications. However, with the rapid advances in technology, current FPGAs contain much more configurable logic blocks (CLBs) than their predecessors. FPGAs are now feasible for a broader range of applications, including those requiring floating-point operations. Some researchers have suggested that FPGAs have become highly competitive with microprocessors in both peak performance and sustained performance [22]. Other researchers have employed FPGAs for several computationally intensive applications, such as molecular dynamics and dense matrix multiplication, and achieved very high performance [19, 27]. Meanwhile, vendors have begun to use FPGAs for high performance computing or even supercomputing. For example, both SRC Computers [20] and Cray [4] have developed FPGA-based high performance computer systems that employ general-purpose processors and FPGA-based application accelerators.

Floating-point Sparse Matrix-Vector Multiplication (SpMXV), $y = Ax$, is a key computational kernel that dominates the performance of many scientific and engineering applications. For example, algorithms for least squares problems and eigenvalue problems [18], as well as the image reconstruction in medical imaging [12] all need to solve sparse linear systems using iterative methods. Such methods involve large numbers of sparse matrix-vector multiplications. Unfortunately, the performance of sparse matrix algorithms tends to be much lower than their dense matrix counterparts for two primary reasons. First, the irregularity of memory accesses causes large numbers of cache misses, while the memory speed is much slower compared to the processing speed. Second, the high ratio of load and store operations to floating-point operations stresses the load/store units, while the floating-point units are often under-utilized [21]. Although some optimizations have been proposed to improve the performance of SpMXV on cache-based memory systems, they require information on the sparsity structure of the input matrix. For matrices with very irregular sparsity structure, these optimizations provide little improvement [10, 17].

FPGAs have become an attractive option for implementating SpMXV. Besides the high floating-point performance, the current FPGA fabrics also provide large amounts of on-chip memory as well as abundant I/O pins. These features enable FPGA designs to provide high on-chip and off-chip memory bandwidth to I/O-bound applications [22]. Thus, FPGA-based designs are able to avoid the long latency caused by the cache misses. As far as we know,

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

FPGA'05, February 20–22, 2005, Monterey, California, USA.

our work is the first to investigate implementation of *floating-point* SpMXV on FPGAs.

We propose a simple yet efficient design for SpMXV, which utilizes a tree of binary operators. Our design has the following advantages. First, it conforms to an existing sparse matrix storage format and makes no assumption on the sparsity structure of the input matrix. Our design accepts matrices in Compressed Row Storage (CRS) format [2], one of the most commonly used storage formats for sparse matrices. Our design can be applied to any sparse matrix, regardless of its sparsity structure or size. In particular, the performance of our design only depends on the number of nonzero elements in each row, and not on the distribution of these nonzeros among the columns. For matrices with very irregular sparsity structure, our design does not suffer as much performance degradation as do the general-purpose processors. For large input matrices, a block SpMXV algorithm is proposed.

Secondly, our design is able to achieve high performance for floating-point SpMXV. As the nonzero elements are accessed in the same order as they are stored in the memory, no complex cache management mechanism is needed, nor does our design suffer from the long access delay caused by the cache misses. By performing multiple I/O operations and floating-point multiplications/additions simultaneously, our design is able to achieve high I/O parallelism as well as computational parallelism. We employ IEEE-754 format double-precision floating-point multipliers/adders [11], which achieve high clock speed through pipelining. Since the deep pipelining in the floating-point adder may cause data hazards, a reduction circuit is developed to accumulate the intermediate results. Another advantage of our design is that it is parameterized and flexible. It is characterized by various parameters, such as the number of floating-point units, the block size, the size of the reduction circuit, etc. These parameters can be easily tuned according to the floating-point units employed, the available hardware resources and the available memory bandwidth. Besides solving the SpMXV problem, our design provides a general tree-based design for floating-point applications.

We have implemented our design on a Xilinx Virtex-II Pro XC2VP70 using Xilinx ISE 6.2i Suite [25]. We use matrices from a variety of application domains and with various sparsity structure. The MFLOPS performance of our design increases with the number of floating-point units used. Our design is also scalable in terms of the available memory bandwidth. With a memory bandwidth of 8 GB/s, the theoretical peak performance of any solution to the SpMXV problem is 1.6 GFLOPS (see Section 4). For this memory bandwidth, our design achieves 20% - 75% of this theoretical peak performance for the test matrices considered in this paper. Even after we deducted 30% of the performance as penalty for memory access, our design still achieves over 350 MFLOPS for all the test matrices. With the same memory bandwidth, the highly optimized program for SpMXV [10] on an Itanium 2 system achieves less than 100 MFLOPS for some of the test matrices.

The rest of the paper is organized as follows. Section 2 introduces the SpMXV and discusses related work. Section 3 presents our proposed architecture. Section 4 presents the experimental results. Section 5 concludes the paper.

2. BACKGROUND

2.1 SpMXV Problem

SpMXV is defined as follows. Consider the matrix-vector product $y = Ax$. A is an $n \times n$ sparse matrix with n_z nonzero elements. We use A_{ij} ($i = 0, ..., n-1; j = 0, ..., n-1$) to denote the elements of A. x and y are vectors of length n, and their elements are denoted by x_j,y_i ($j = 0, ..., n-1; i = 0, ..., n-1$) respectively. Initially A, x, y are in the external memory. Thus, to perform $y = Ax$, $n_z + n$ input operations are needed to bring A and x to the processor. Note that due to the irregular sparsity structure, either the row index i or the column index j or both of them need to be read together with A_{ij}, if $A_{ij} \neq 0$. For matrix element A_{ij}, the following computation is performed:

$$y_i = y_i + A_{ij} \times x_j, \textit{if } A_{ij} \neq 0 \tag{1}$$

As each nonzero element of A needs two floating-point operations, the total number of floating-point operations to be performed is $2n_z$. When the computation is completed, n output operations are needed to write y back to the external memory. Thus the total number of I/O operations is $n_z + n + n$.

Suppose the memory bandwidth is B words (along with their row or column indices) per second, and F floating-point operations are performed per second. Then, the computation time of any algorithm for SpMXV, T_{comp}, is at least $\frac{2n_z}{F}$. $T_{I/O}$, the I/O time, is at least $\frac{n_z+2n}{B}$. The total time required to perform SpMXV is thus

$$T \geq \max(T_{I/O}, T_{comp}) = \max(\frac{n_z + 2n}{B}, \frac{2n_z}{F}) \tag{2}$$

We see that a design is most efficient when F is approximately $2B$, so that $T_{I/O}$ is approximately equal to T_{comp}. To reduce T, we need to increase both B and F.

2.2 Related Work

A large amount of work has been done to improve the performance of sparse matrix-vector multiplication for general-purpose processors. Some researchers focus on performance tuning by code and data structure reorganization. Pinar and Heath [17] propose to pack contiguous nonzero elements into dense blocks to reduce the number of load operations. They also design a reordering algorithm to enlarge the dense blocks inside a matrix. Similarly, SPARSITY [10] proposes to improve the performance of SpMXV through loop transformations, register blocking and cache blocking. It provides a framework for selecting optimization parameters for a specific sparse matrix and machine. These optimizations all require information on the sparsity structure of the input matrix. For matrices with very irregular sparsity structure, these optimizations achieve little improvement.

Some algorithms aim on exploiting the parallelism of SpMXV on multiple machines. In [9], a parallel algorithm for SpMXV is proposed for machines with a hypercube architecture. In this architecture, the sparse matrix is partitioned into blocks which are distributed among the machines. The authors reduce the overall time by reducing the communication costs. In [13], Manzini finds the theoretical average communication costs in hypercubic networks for SpMXV. However, the hypercube architecture is impractical for FPGA-based designs because its routing complexity is prohibitive. An algorithm on a linear array of processing elements is presented in [8]. It is based on a special cover of the nonzero elements in the matrix. Such cover is called ”staircase”. The number of PEs (Processing Elements) required is equal to the size of the minimal staircase cover of the matrix. In our work, we prefer an architecture independent of the sparsity structure of the input matrix.

Abundant research has been conducted on implementing floating-point applications on FPGAs. Floating-point cores with various precisions as well as various numbers of pipeline stages have been designed [7]. In [27], floating-point dense matrix multiplication is implemented on an FPGA device and the GFLOPS performance is compared with that of general-purpose processors. Underwood et al. [22] examine the potential capacity of FPGAs in performing floating-point BLAS applications. Besides the basic linear algebra applications, more complex applications, such as floating-point LU decomposition, have also been implemented on FPGAs and high performance has been reported [6].

However, there has been little work on FPGA-based floating-point sparse matrix operations. In [5], fixed point SpMXV is implemented. They design a constant-coefficient multiplier that can be reloaded at run-time and use it for SpMXV. The schedule for the use of such multipliers is performed by a host computer. They have proposed schedules for dual-multiplier and triple multiplier systems only. LU decomposition for sparse block-diagonal-bordered matrices has been implemented in [24], using an FPGA-based multiprocessor machine. In the machine, each PE is an Altera Nios configurable processor IP [1], with attached floating-point adder, multiplier and divider. The data are stored in on-board memory, and the on-chip memory serves as the cache. The goal of [24] is to build a generic multiprocessor machine, while we focus on the application-specific design for SpMXV.

3. FPGA-BASED DESIGN FOR SpMXV

We first introduce the storage format of the input matrix. Then we present the basic architecture of the design. Next we complete the architecture by reducing the idle cycles, resolving data hazards and providing solutions for large matrices. Finally we discuss how to determine the various parameters in the design.

3.1 CRS Format

In our design, the sparse matrix is stored in CRS format, in which the nonzeros of the matrix rows are stored in contiguous memory locations. In CRS format, there are three vectors: val for floating-point nonzero elements; col for the column indices of the nonzeros; and ptr that stores the locations in the val vector that start a row [2].

As an example, consider the sparse matrix A as follows:

$$A = \begin{pmatrix} 10 & 0 & 0 & -2 \\ 3 & 0 & 9 & 0 \\ 0 & 7 & 0 & 0 \\ 0 & 0 & 8 & 4 \end{pmatrix}$$

The CRS format for this matrix is then specified by the vectors given below:

val	10	-2	3	9	7	8	4
col	0	3	0	2	1	2	3

ptr	0	2	4	5

By subtracting consecutive entries in the ptr vector, a new vector len can be generated. len vector stores the size of each row, that is, the number of nonzeros in each row. Note that the last entry in len vector is calculated using $n_z - ptr(n-1)$. For our example, the len vector is:

len	2	2	1	2

3.2 Basic Architecture

Our basic architecture consists of a binary tree with k leaves, as shown in Figure 1. In the figure, we use $k = 4$ for the purpose of illustration. Each leaf node contains a floating-point multiplier and local storage. Each of the $k-1$ internal nodes in the tree contains a floating-point adder. The multipliers and adders are numbered as shown in Figure 1, where the $(k-2)$th adder is the *root node* of the tree. The output of the tree is fed into a reduction circuit which is in charge of accumulating the intermediate results. This circuit will be explained in details in Section 3.3.2. There is also a control unit in our architecture, which controls the I/O operations as well as the operations of the multipliers and adders.

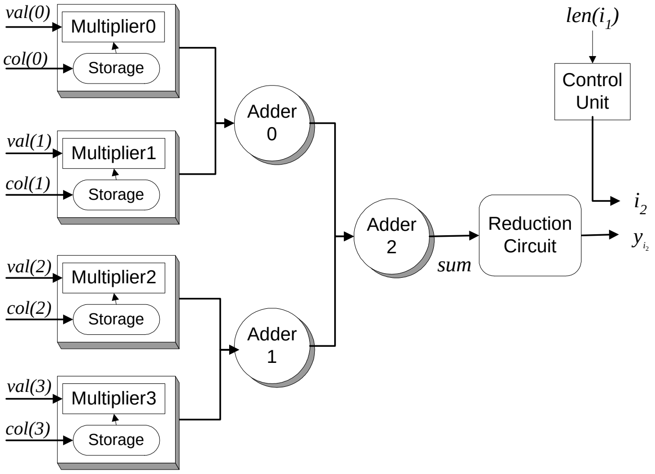


Figure 1: Basic architecture of the design when $k = 4$

```
{Control Unit}
if Output of the reduction circuit is valid then
  write it as $y_{i_2}$ to the external memory
  $i_2 = i_2 + 1$
end if
counter = counter - 1
if counter $\leq 0$ then
  $i_1 = i_1 + 1$
  read $len(i_1)$
  counter = $len(i_1)/k$
end if
{Multipliers}
for Multiplier $m = 0$ to $k-1$ (in parallel) do
  read $val(m)$ and $col(m)$
  read storage(m) using $col(m)$ as address
  compute $val(m) \times$ storage(m)[$col(m)$]
end for
{Adders}
for Adder $m = 0$ to $k-2$ (in parallel) do
  add two operands
  if $m = k-2$ then
    output is variable $sum$
  end if
end for
```

Figure 2: Cycle-specific operations for components in the basic architecture

In our design, matrix A and vector x are initially stored in the external memory. Before the computation starts, vector x is read in and stored in the local storage of all the leaf nodes. This time period is referred to as initialization time. After initialization, multiplier m ($m = 0, \ldots, k-1$) reads a nonzero element $val(m)$ during each clock cycle. The multiplier finds the corresponding element of the x vector in its local storage using the column index of the nonzero, $col(m)$, as the read address. It then multiplies these two floating-point numbers. The results of the multipliers are fed into the adders and are summed up. If the size of each row equals k, the root node yields the final result for each y element and the reduction circuit is not needed.

However, since the typical value of k is smaller than 13 (see Section 3.3.4), each row of A probably has more than k nonzero elements. For the basic architecture, we assume that $len(i)$ is a multiple of k ($i = 0, 1, \ldots, n-1$). Thus, row i is partitioned into $len(i)/k$ sub-rows. The sub-rows of row i are read into the architecture consecutively, and their sums are given to the reduction circuit by the root node. The reduction circuit accumulates the sums of the sub-rows, and yields the final result of y_i.

The control unit takes the size of a row as input, and calculates the number of sub-rows in the row. When the reduction circuit generates an valid output, the control unit writes it to the external memory together with the correct row index. Note that even if a row only consists of one sub-row, the sum of this sub-row still goes through the reduction circuit for the simplicity of implementation. Figure 2 shows the cycle-specific operations of the components (except the reduction circuit) in the basic architecture after the initialization completes. For the sake of understanding, we present these operations as a sequential program. In real implementations, the components work in parallel.

The basic architecture performs $n_z + 2n$ I/O operations, which is the minimum I/O operations for SpMXV. Since F is approximately $2B$, the architecture efficiently utilizes both the memory bandwidth and the computing power.

3.3 Complete Design

3.3.1 *Improved Architecture*

In the improved architecture, we consider the case where $len(i)$ is not necessarily a multiple of k ($i = 0, 1, \ldots, n-1$). Thus, the number of sub-rows in row i equals $\lceil len(i)/k \rceil$, and all sub-rows contain k nonzero elements except possibly one sub-row. If the size of one sub-row is less than k, it has to be expanded to k through zero padding, which is inserted by the control unit.

The performance of the architecture is affected adversely by the zero padding. Zero padding results in idle cycles of floating-point multipliers and adders (with zeros as operands), and its affect on performance is measured by the metric "overhead". The overhead of our design for matrix A, ov, is calculated as

$$\begin{aligned} ov &= \textit{total number of zeros needed for padding}/n_z \\ &= \sum_{i=0}^{n-1} (\lceil len(i)/k \rceil \times k - len(i))/n_z \\ &= (\sum_{i=0}^{n-1} \lceil len(i)/k \rceil \times k - n_z)/n_z \end{aligned} \tag{3}$$

For each zero padding, our design performs one floating-point multiplication and one floating-point addition. Thus, our design performs $2 \times ov \times n_z$ unwanted floating-point operations for the SpMXV with matrix A. To improve the performance of our design, we need to reduce ov.

In our design, ov is dependent on the number of multipliers k, as well as the sparsity structure of the input matrix. In general, the more multipliers we use, the higher the overhead. This is shown in Figure 3, which illustrates how the overhead for sparse matrix $lns3937$ changes with k (the application domain and properties of this matrix are outlined in Section 4). However, the curve drops steeply at $k = 6$. This is because more than 30% of the rows contain 6 nonzeros, making 6 the best value for k. It would be desirable if an algorithm is designed to analyze the sparsity structure of each specific matrix and find the optimum k. However, such an algorithm may not contribute to the overall performance. First, it may require detailed statistics for the given matrix, such as the distribution of nonzero elements. To get such statistics, extra I/O operations and processing time are needed. Second, the number of floating-point units that can be configured on the FPGA is more likely to be constrained by the available resources than by the input matrix. Such resources include the I/O pin count, the number of slices, and the memory bandwidth.

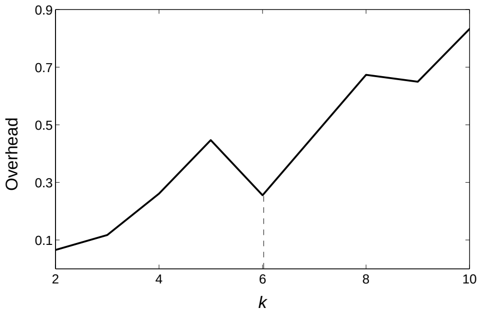


Figure 3: Overhead analysis for sparse matrix *lns3937*

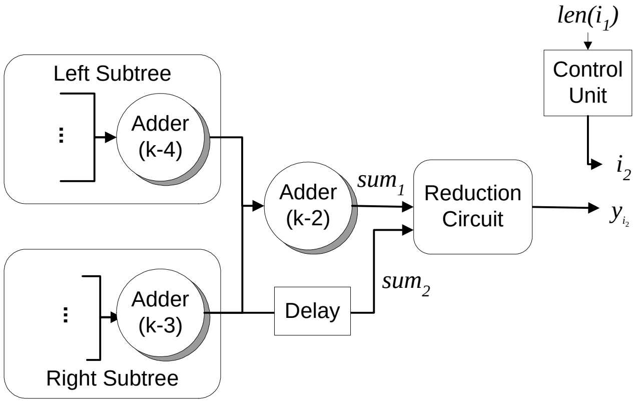


Figure 4: Improved architecture for the design

Thus, in the improved architecture, we reduce the overhead using a method called *merging*. This method is independent of the sparsity structure of matrix A. When merging is allowed, if the control unit detects that the last sub-row of the ith row is to be processed, it checks if this sub-row can be merged with a sub-row of the $(i+1)$th row. Merging is allowed if both rows have a sub-row which contains no more than $k/2$ nonzeros, that is, if $len(i) \mod k \leq k/2$ and $len(i+1) \mod k \leq k/2$. In the merging, the last sub-row of the ith row is read in the left subtree, and one sub-row of the $(i+1)$th row is read in the right subtree.

The improved architecture is shown in Figure 4, and the cycle-specific operations of the components are shown in Figure 5. The *root node of the left subtree* is the $(k-4)$th adder, and the *root node of the right subtree* is the $(k-3)$th adder. The sums of the two sub-rows are called sum_1 and sum_2 respectively, as shown in Figure 4. For simplicity of the logic, sum_1 goes through the root node, being added to 0. To guarantee that sum_1 and sum_2 are yielded at the same clock cycle, delay is inserted between the $(k-3)$th adder and sum_2. The delay is the same as the latency of the root node, the $(k-2)$th adder. Both sum_1 and sum_2 are given to the reduction circuit, which then yields the final result of y elements.

Through the merging, the overhead of the design for matrix A is reduced to

$$\begin{aligned} ov &= \textit{total number of zeros needed for padding}/n_z \\ &= \sum_{i=0}^{n-1} (\lceil len(i)/(k/2) \rceil \times k/2 - len(i))/n_z \\ &= (\sum_{i=0}^{n-1} \lceil 2 \times len(i)/k \rceil \times k/2 - n_z)/n_z \end{aligned} \tag{4}$$

We see that the performance improvement brought by the merging varies according to the input matrix. For the test matrices we used in the experiments, the merging is able to reduce 50% of the overhead. To further reduce overhead, we could merge 4 sub-rows with less than $k/4$ nonzeros, or even 8 sub-rows with $k/8$ nonzeros. However, such improvements will add additional complexity to the control, and hence are not considered.

According to Equation 3 and Equation 4, the overhead in our design only depends on $len(i)$ ($i = 0, 1, \ldots, n-1$) and k. Thus, when k is fixed, the performance of our design is only affected by the number of nonzero elements on each row, and not by the distribution of the nonzeros among the columns.

3.3.2 Reduction Circuit

In the basic architecture as well as the improved architecture, when a row has more than one sub-rows, the sums of the sub-rows have to be accumulated. Since floating-point adders are usually deeply pipelined to achieve high clock speed, a Read-After-Write(RAW) data hazard arises when a row has more than 2 sub-rows. Suppose a row contains 3 sub-rows which are denoted as r_1, r_2 and r_3. It is possible that the sum of r_3 is read by the adder before the sum of $r_1 + r_2$ has been generated. In this case, the final sum of this row will be invalid. This is a critical problem, and it not only exists for the accumulation of a set of floating-point values, but also for any reduction of values using pipelined binary operators. When $k = 4$, most of the test matrices we used in the experiments have more than 2 sub-rows in each row. Thus, in our design, we cannot simply use one floating-point adder for accumulation. Instead, we develop a reduction circuit to accumulate the sums of the sub-rows.

In this paper, we propose a simple design for reduction circuit. It accumulates all the intermediate results for a row and outputs the final result. Suppose a row has m sub-rows, thus m inputs are sequentially delivered to the reduction circuit. To sum up these m inputs, a straightforward method is to construct a binary adder tree with $m-1$ adders and $\lceil \lg(m) \rceil$ tree levels. Since the inputs arrive at the reduction circuit sequentially, we observe that the adders at the same tree level is used in different clock cycles. Therefore, we replace the level 0 adders with a buffer feeding a single adder, replace the level 1 adders with another adder with buffer, and so forth. Thus instead of using $m-1$ floating-point adders, we use $\lceil \lg(m) \rceil$ floating-point adders and $\lceil \lg(m) \rceil$ buffers, as shown in Figure 6. At level l, the adder and buffer is denoted as *adder*$_l$ and *buffer*$_l$ ($l = 0, \ldots, \lceil \lg(m) \rceil - 1$).

In each clock cycle, each adder checks if it has enough operands to begin execution. Normally, the adder proceeds if the buffer contains two inputs. However, when m is not a power of 2, adders at certain levels need to proceed even though their corresponding buffers only contain a single input. Take $m = 5$ as an example. There are 5 inputs at level 0, and the 5th input cannot be paired up with any other input. Thus if the 5th input is detected in *buffer*$_0$, it is added with 0 by *adder*$_0$ even if *buffer*$_0$ only contains one input. To identify such cases, we label each input with a number, which indicates how many sub-rows of the same row remain to be accumulated. At level l, if the label of an input is less than $2^l + 1$, it must be the last input on that level. Thus when such an input is in *buffer*$_l$ and no other input is waiting to pair up with it, *adder*$_l$ adds the input with 0.

```
{Control Unit}
if Output of the reduction circuit is valid then
  write it as $y_{i_2}$ to the external memory
  $i_2 = i_2 + 1$
end if
if merge = 1 then
  counter = counter1, count = count1, merge = 0
else
  counter = counter - 1
end if
if counter $\leq 0$ then
  $i_1 = i_1 + 1$
  read $len(i_1)$
  count = $len(i_1)$, counter = $\lceil len(i_1)/k \rceil$
end if
for $m = 0$ to $k$ - 1 do
  zeropadding(m) = 1
end for
if counter $> 1$ then
  for $m = 0$ to $k$-1 do
    zeropadding(m) = 0
  end for
  count = count - $k$
else if counter = 1 then
  for $m = 0$ to count-1 do
    zeropadding(m) = 0
  end for
  if (count $\leq k/2$) and ($len(i+1) \mod k \leq k/2$) then
    merge = 1, counter1 = $\lceil len(i+1)/k \rceil$ - 1
    count1 = $len(i+1) - (len(i+1) \mod k)$
    for $m = k/2$ to $k/2 + (len(i+1) \mod k - 1)$ do
      zeropadding(m) = 0
    end for
  end if
end if
{Multipliers}
for Multiplier $m = 0$ to $k-1$ (in parallel) do
  if zeropadding(m) = 0 then
    read $val(m)$ and $col(m)$
    read storage(m) using $col(m)$ as address
    compute $val(m) \times$ storage(m)[$col(m)$]
  else
    compute $0 \times 0$
  end if
end for
{Adders}
for Adder $m = 0$ to $k-2$ (in parallel) do
  add two operands
  if $m = k-3$ then
    output is variable $sum2$
  end if
  if $m = k-2$ then
    output is variable $sum1$
  end if
end for
```

Figure 5: Cycle-specific operations for components in the improved architecture

Figure 7 illustrates how we accumulate a row with 5 sub-rows using 3 adders. The small boxes represent the inputs and their labels are shown below them. The inputs labeled as 0 do not really exist; they are shown in the figure to form a binary tree. If an input is added with an input labeled as 0, it is the last input on its level and cannot not be paired up with any other input. We see on both level 0 and 1, the last input is labeled as 1 and is added with 0. At level 2, however, although the last input is still labeled as 1, it is paired up with another input in *buffer*$_2$. This time, these two inputs are summed up, and the final result is yielded.

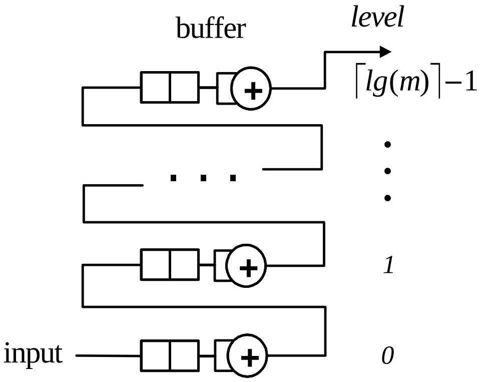


Figure 6: Reduction circuit with $\lceil \lg(m) \rceil$ adders

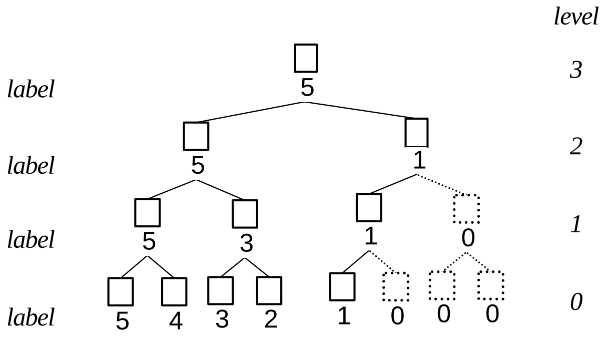


Figure 7: Illustrations of the additions of 5 sub-rows using 3 adders

We compute the latency of the reduction circuit by examining the last input on level 0 as it goes through the reduction circuit. The last input enters *buffer*$_0$ at cycle m, and then is read by *adder*$_0$ in the next clock cycle; *buffer*$_1$, then *adder*$_1$; $\ldots$; *buffer*$_{\lceil \lg(m) \rceil - 1}$, then *adder*$_{\lceil \lg(m) \rceil - 1}$. Therefore, the latency of the reduction circuit is $m + \lceil \lg(m) \rceil \times \alpha$, where α is the pipeline delay of the floating-point adder. Since *adder*$_l$ reads from *buffer*$_l$ whenever it contains two inputs, each buffer never contains more than 2 inputs. Thus the size of each buffer is 2 and the total buffer size is $O(\lceil \lg(m) \rceil)$.

The reduction circuit can be easily incorporated into our SpMXV architecture. Modification is only required when merging is allowed for SpMXV. As discussed in Section 3.3.1, if two rows are merged, the sums of two sub-rows are given to the reduction circuit in one clock cycle. In this case, we modify *adder*$_0$ so that it can accept two inputs that belong to different rows. However, if both inputs are the last inputs of their own rows, the size of *buffer*$_0$ has to be increased. Therefore, in order to avoid overflowing *buffer*$_0$, two consecutive rows that both contain only one sub-row are not allowed to be merged.

With the reduction circuit, our design can be abstracted as shown in Figure 8. In this figure, the leaf nodes are not multipliers, but subtrees. It shows a large binary tree can be realized using a small one and the reduction circuit. For example, if the row has 32 nonzeros, it requires a binary tree of 6 levels, with 32 multipliers as the leaf nodes. That is, it requires four level-3 binary trees as shown in the upper part of Figure 8. However, in our design, we can accumulate these 32 nonzeros using only one level-3 binary tree and the reduction circuit, where only 8 multipliers are used. Compared to the level-6 binary tree, such a design occupies less resources, but has longer latency.

Although the reduction circuit introduced here is simple, it uses $\lceil \lg(m) \rceil$ floating-point adders and is not an optimal design. We have thus proposed several scalable reduction circuits. In [26], we present a design which employs *one* floating-point adder to reduce input sets of sequentially delivered floating-point values. However, this design only handles input sets whose sizes are power of 2 [26]. In [15], we further propose designs that use *two* floating-point adders and can accumulate input sets of arbitrary sizes.

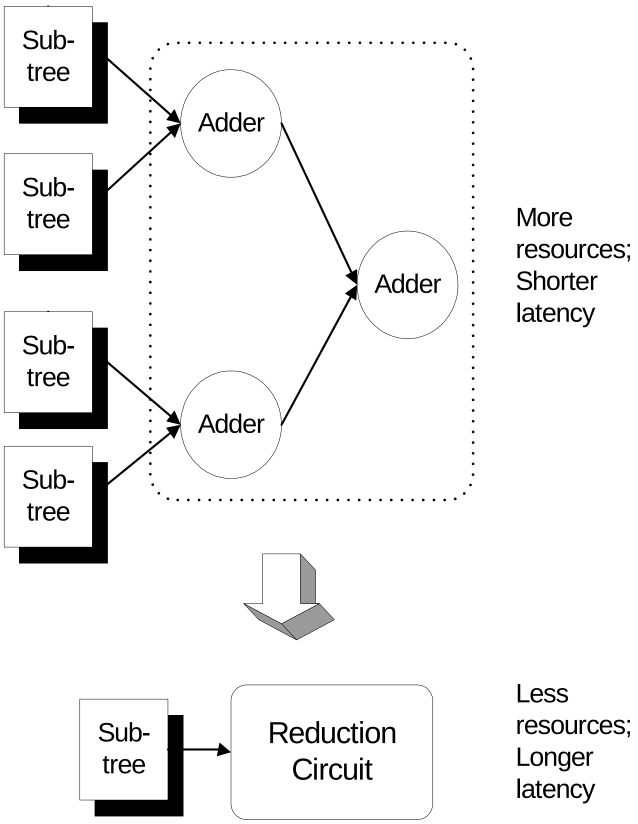


Figure 8: Comparison between designs with and without the reduction circuit

3.3.3 Block SpMXV

In our design, we need to store vector x in the local storage of every multiplier. When the size of the input matrix and vector x is large, we employ a block SpMXV algorithm. Suppose the block size is b. Then the $n \times n$ matrix A is partitioned into $\lceil n/b \rceil$ blocks, of size $n \times b$. That is, each block contains b columns of A. At the same time, the $n \times 1$ vector x is partitioned into $\lceil n/b \rceil$ blocks, of size $b \times 1$. The blocks of A and x are denoted as A^u and x^u ($u = 0, \ldots, \lceil n/b \rceil - 1$) respectively. The partition is shown in Figure 10. The blocks of A are stored in CRS format. As most elements in A are zeros, it is possible that some consecutive rows in A^u only contain zeros. Such *zero sub-blocks* are illustrated in Figure 10, surrounded by dash lines. In CRS format, these zero sub-blocks are not stored in the external memory.

In the block SpMXV algorithm, each A^u is multiplied with the corresponding x^u ($u = 0, \ldots, \lceil n/b \rceil - 1$). For each block multiply, the local storage needs to be initialized using x^u before the computation starts. To reduce the time of the block SpMXV, we overlap the initialization time of x^u with the computation time of $A^{u-1} \times x^{u-1}$ ($u = 1, \ldots, \lceil n/b \rceil - 1$). In the best case, only the initialization time of x^0 needs to be counted. However, due to the zero sub-blocks, it is possible that the computation of $A^{u-1} \times x^{u-1}$ takes less time than the initialization of x^u. Thus, the computation of $A^u \times x^u$ cannot start until the local storage is initialized using x^u. In the worse case, we need to count the initialization time of all x^u ($u = 0, \ldots, \lceil n/b \rceil - 1$), which equals the initialization time of the entire vector x.

Block size b is an important parameter in the block SpMXV algorithm. It results in tradeoff between the computation time and the initialization time. As b increases, the initialization time for x^u ($u = 0, \ldots, \lceil n/b \rceil - 1$) increases. However, a smaller b may result in fewer nonzeros in A^u. Thus the computation time may increase because of increased overhead. On the other hand, the number of zero sub-blocks increases as b decreases. With a small b, it is possible that the computation has to be suspended to wait for the initialization to complete. To show how the computation time, the initialization time, and the total time of the block SpMXV algorithm vary with b, we choose matrix *memplus* from our test matrices as an example. The size of *memplus* is 17758, and it has 126150 nonzeros. Its sparsity strcture is irregular so that different values of b result in different numbers of nonzero elements in each block. As shown in Figure 9, when b increases from 2000 to 18000, the initialization time increases, and the computation time decreases. The total time for SpMXV does not vary much when b is larger than 4000. The optimal performance is achieved when no block SpMXV is performed, that is, when the block size is larger than the size of the matrix.

When the block SpMXV algorithm is implemented on an FPGA device, block size b is usually determined by the available on-chip memory. Suppose the size of on-chip memory is M, and the number of multipliers is k. When the problem size n exceeds M/k, block size b is set as $M/2k$. The denominator is $2k$ to enable the overlapping of the initialization time and the computation time.

3.3.4 Parameterized Design

The improved architecture, reduction circuit and the block algorithm constitute a complete FPGA-based design for SpMXV. The design is characterized by various parameters, including the number of pipeline stages in the floating-point units, α; the number of floating-point multipliers, k; the size of the reduction circuit, u; and the block size, b. The selection of b has been discussed in Section 3.3.3. We now discuss how to determine the other parameters.

The size of the reduction circuit refers to the number of floating-point adders and buffers used in the circuit. It is the only parameter dependent on the input matrix, and is determined as

$$u = \lceil \lg(\max\{\frac{len(i)}{k}\}) \rceil, (i = 0, \ldots, n-1). \quad (5)$$

Both α and k depend on the floating-point units employed. Meanwhile, k is also determined by several other factors, including the number of slices, the number of I/O pins, and the available memory bandwidths. Thus, determination of k would require a complicated model. However, as shown later in Section 4, k is usually constrained by the I/O pin count or the available memory bandwidth. Thus we do not consider the number of slices in the model. In this way, we simplify the calculation of k as:

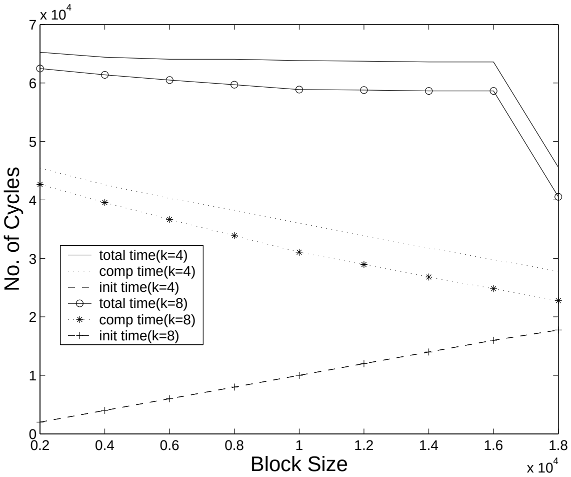


Figure 9: Total time/Computation time/Initialization time vs. block size for matrix *memplus*

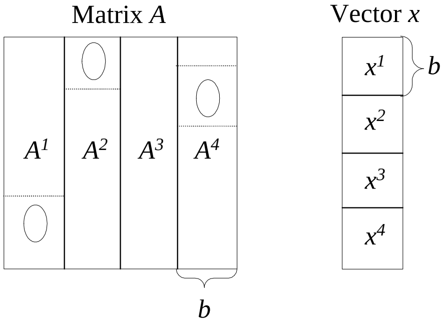


Figure 10: Partition of matrix A and vector x for block SpMXV

$$k \approx \frac{\textit{memory bandwidth}}{\textit{clock speed} \times \textit{I/O pins needed by the design}} \quad (6)$$

The I/O pin count of the largest FPGA device, Xilinx Virtex-II Pro XC2VP125, is 1200. Thus, the maximum value of k is 13 (we assume the row/column index is no more than 16-bit wide and ignore other hardware constraints). As our proposed architecture is simple and flexible, increasing or decreasing k requires little effort. Thus, our design is adaptive to various situations with different available memory bandwidth, different numbers of I/O pins as well as different clock speed requirements.

Besides SpMXV, our design provides a flexible design for many other applications. For example, the dot product of two vectors is widely used to determine whether or not the vectors are othogonal. This kernel can be implemented using our design with only minor modifications to the control unit. Our design can also implement summation of large number of floating-point numbers by replacing the multipliers with adders. Moreover, our design is useful for image and video processing applications discussed in [3].

4. EXPERIMENTAL RESULTS

4.1 Experiment Setup

We used Xilinx ISE 6.2i and Mentor Graphics ModelSim 5.7 development tools in our experiments [14, 25]. Our target device is Xilinx Virtex-II Pro XC2VP70 [25], which contains 33088 slices and more than 5 Mbit on-chip memory.

We evaluated our design using sparse matrices from a variety of engineering and science applications, including circuit physics, astrophysics, fluid mechanics, etc. These matrices are available from University of Florida Sparse Matrix Collection [23]. All of the matrices are square, most of which are neither symmetric nor diagonal. Table 1 gives the size and application domain of each matrix used in our experiments. It also shows the number of nonzeros in the matrix as well as the size of the longest row. The matrices are roughly ordered by the regularity of sparsity structure, with the more regular ones at the top.

In our design, the implementations of floating-point adder and multiplier have no effect on the architecture or the operations of the control unit. These floating-point units are independent modules and can be plugged into our design easily, with no or few modifications to the interface between them and the control unit. In our experiments, we used our own 64-bit floating-point adders and multipliers. Table 2 gives the characteristics of these floating-point units, whose implementation details can be found in [7].

Table 2: 64-bit Floating-Point Units

	Adder	Multiplier
Number of Pipeline Stages	19	12
Number of Slices	933	910
Achievable Clock Speed(MHz)	200	205

4.2 Performance

We have experimented with two implementations of the complete design for SpMXV. The difference between them is the first implementation does not perform the merging. These two implementations are referred to **Architecture 1** and **Architecture 2** in the rest of the paper.

In our experiments, we choose $k = 4, 6, 8$. k floating-point multipliers and $k - 1$ floating-point adders are needed. We determined the values of other design parameters according to the resource constraints of the target device as well as the test matrices we selected. For all the test matrices, the longest row has less than $2^7 \times k$ nonzeros ($k = 4, 6, 8$); the size of the reduction circuit u equals 7. The block size b for large matrices, such as matrices 1, 2, and 4, is determined by k and the size of on-chip memory on the device. Without loss of generality, b is always set as a power of 2 in our experiments.

The number of slices occupied and the achievable clock speed of the implementations are shown in Table 3. The characteristics of the reduction circuits are also shown in the table. Since the reduction circuit contains 7 floating-point adders, it occupies about 28% of the total slices on the device, It only runs at 165 MHz, although the adder can be clocked at 200 MHz. We see that for both Architecture 1 and Architecture 2, the area of the design is mostly determined by the size of the floating-point units and the reduction circuits. The area for the control unit and the routing occupies less than 5% of the total area. Since Architecture 2 allows merging, the reduction circuit it uses is about 1% larger than the reduction circuit in Architecture 1. Thus, Architecture 2 occupies slightly more slices than Architecture 1. As k increases, more adders and multipliers are used and the area of our design increases linearly.

We also observe that the achievable clock speed of the implementations is bounded by the speed of the reduction circuit. Therefore, the speed of both Architecture 1 and Architecture 2 remains the same as k varies from 4 to 8. The last row of Table 3 shows the required memory bandwidth. The memory bandwidth is calculated according to Equation 6, based on the number of I/O pins used and the achievable clock speed. The more multipliers are employed, the more I/O pins are used. Since the clock speed of the design varies little with k, the required memory bandwidth thus increases linearly as k increases. When $k = 8$, Architecture 1 only occupies about 73% of the total slices on the device, while the required memory bandwidth is about 13.7 GB/s. Thus, in the implementations, the maximum value of k is more likely to be determined by the available memory bandwidth or the number of I/O pins than by the number of slices.

From Table 3, we know that both the area and the speed of our design is mostly determined by the floating-point units and the reduction circuit used. Since the floating-point units and the reduction circuit are independent modules in our design, they can be easily replaced when improved ones are available. With smaller and faster floating-point units and reduction circuits, the area and the speed of our design will be improved accordingly.

We use MFLOPS to measure the sustained performance of our design, as it is the most widely used performance metric for SpMXV. The MFLOPS performance is calculated as

$$\mathit{MFLOPS} = \frac{\textit{total number of floating-point operations}}{\textit{total time}} \tag{7}$$

The total time of our design is the period of time from reading x to initialize the local storage to completing the computations for all the elements of y. The other metric used is the overhead, which is calculated as in Equation 3 and Equation 4 for Architecture 1 and Architecture 2, respectively. The performance of our design is shown in Figures 11 and 12. For the sake of readability, only the results for $k = 4$ and $k = 8$ are shown in the figures. Our observations from the figures are also true for $k = 6$.

We see that the MFLOPS performance of our design is greatly affected by the overhead. The smaller the overhead, the higher the performance. As an extreme case, matrix 1 consists entirely of uniformly aligned 8×8 blocks. Hence both Architecture 1 and 2 have 0 overhead and achieve very high performance for matrix 1. By merging small sub-rows, Architecture 2 is able to reduce 50% of the overhead of Architecture 1, and thus increases the MFLOPS performance of most of the matrices. Matrix 4 is a special case. Over 85% of its rows contain less than 4 nonzeros, hence many rows only contain 1 sub-row. In Section 3.3.2, we stated that no merging is allowed for two consecutive rows that both contain 1 sub-row. Thus, for matrix 4, the overhead in Architecture 1 and Architecture 2 are almost the same.

Despite the fact that the overhead increases with k, for most of the test matrices, the MFLOPS performance improves as k increases due to the increased parallelism. We also notice that for some matrices, such as matrices 9 and 10, the performance improvement is bounded by the initialization time. However, in general, the MFLOPS performance of our design increases when more hardware resources are available.

We next study how the sustained MFLOPS performance of our design varies with the available memory bandwidth and the clock speed. We use matrix 3 as an example, which is neither too regular nor too irregular. The results are shown in Figure 13. For the sake of illustration, the memory bandwidth is varied from 4 GB/s to 20 GB/s, while the clock speed varies from 100 MHz to 200 MHz. In this scenario, k ranges from the minimum value 1 to the maximum

Table 1: Sparse matrices used in our experiments

	Name	Application Area	Size n	Nonzeros n_z	(n_z/n^2)%	$\max\{len(i)\}$
1	*raefsky3*	Fluid/structure	21200	1488768	0.33	80
2	*bcsstk35*	Automobile frame	30237	1450163	0.16	222
3	*rdist1*	Chemical processes	4134	94408	0.55	81
4	*memplus*	Circuit simulation	17758	126150	0.04	353
5	*gemat11*	Power flow	4929	33185	0.14	27
6	*lns3937*	Fluid flow	3937	25407	0.16	11
7	*sherman5*	Oil reservoir	3312	20793	0.19	21
8	*mcfe*	Astrophysics	765	24382	4.17	81
9	*jpwh991*	Circuit physics	991	6027	0.61	16
10	*bp1600*	Simplex method	822	4841	0.72	304
11	*str600*	Simplex method	363	3279	2.49	33

Table 3: Characteristics of SpMXV Design

	Architecture 1				Architecture 2			
	Reduction Circuit	$k=4$	$k=6$	$k=8$	Reduction Circuit	$k=4$	$k=6$	$k=8$
Block Size b	-	8192	5120	4096	-	8192	5120	4096
No. of Slices	9211	16613	20132	24113	10209	16931	20123	23856
% of Total Slices on Device	28%	50%	61%	73%	31%	51%	61%	73%
Achievable Clock Speed(MHz)	165	160	160	160	165	160	160	160
Memory Bandwidth(GB/s)	-	7.6	9.3	13.7	-	8.0	10.9	14.1

value 13. From Equation 6, we know that when the memory bandwidth is fixed, k decreases as the clock speed increases. However, as shown in Figure 13, in some cases, the MFLOPS performance still increases with the clock speed. This is because the performance of our design suffers less from overhead with a small k. On the other hand, when the clock speed is fixed, the MFLOPS performance increases with the memory bandwidth because the increased k provides more parallelism.

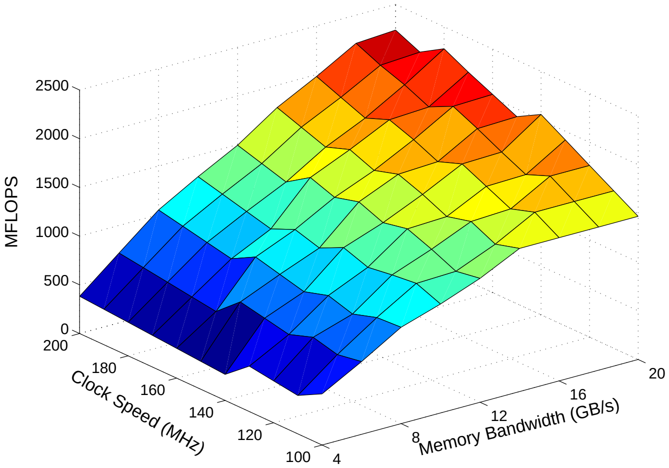


Figure 13: MFLOPS Performance of Achitecture 1 for matrix *rdist1*, as the memory bandwidth and the clock speed vary

4.3 Performance Comparison

To study the efficiency of our design, we compare its performance with the theoretical peak performance of any algorithm for SpMXV. To compute this peak performance, we assume that the computing power is unlimited, and no overhead or data hazard exist. It is denoted as MFLOPS$_P$, and is calculated as

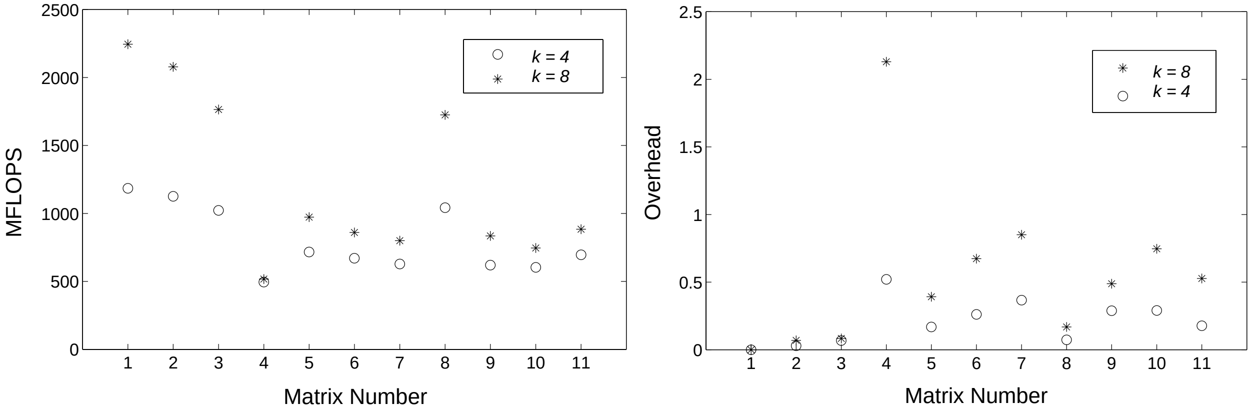
$$\begin{aligned} MFLOPS_P &= \textit{total number of floating-point operations}/T \\ &= 2n_z/T \end{aligned}$$

T is the lower bound on the execution time of any algorithm for SpMXV. According to Equation 2, when the computing power is unlimited, T is lower bounded by $T_{I/O}$. B is the memory bandwidth as defined in Equation 2. If we ignore the I/O time for reading and writing vector x and y, the peak performance is thus

$$MFLOPS_P = \frac{2n_z}{n_z/B} = 2B \qquad (8)$$

Therefore, when the memory bandwidth is 8 GB/s, MFLOPS$_P$ equals 1.6 GFLOPS; when the memory bandwidth is 14.4 GB/s, the peak performance is 2.88 GFLOPS (we assume the row/column index is 16-bit wide). Figure 14 shows the percentage of the peak performance that Architecture 1 achieves. We see that our design achieves up to 75% of the peak performance when the memory bandwidth is 8 GB/s and 14.4 GB/s.

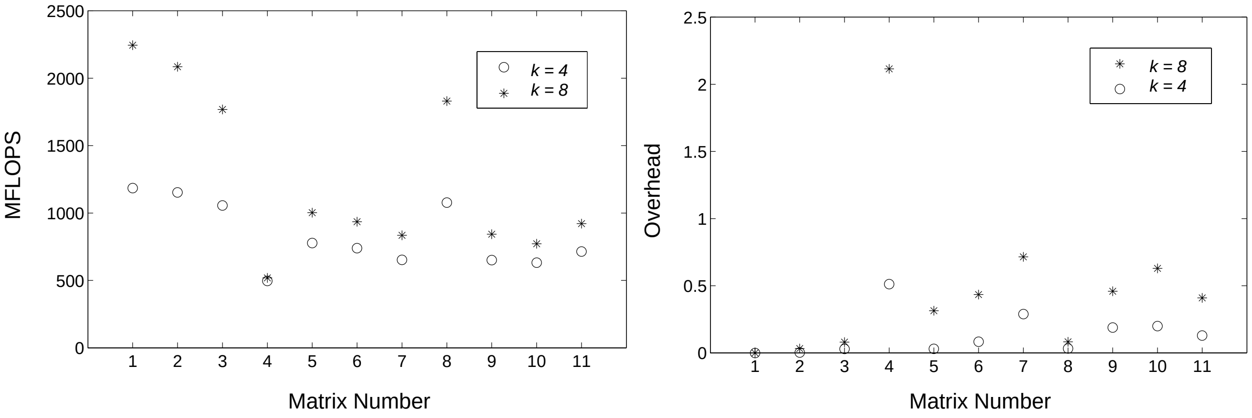
We next compare our design with existing FPGA designs. This is difficult because there are few FPGA-based designs for sparse matrix computation. The most relevant work is [5]. This design achieves 2340 MIPS at a clock speed of 28.57 MHz using 3 multipliers. However, our design is for floating-point SpMXV, while [5] focuses on fixed-point SpMXV. Also, our design does not depend on a host machine to schedule the multiplications and additions. Another related work is the FPGA-based multiprocessor machine proposed in [24]. In this design, 5 Nios soft processors [1] are implemented on an FPGA device. Each Nios processor is aided by an attached floating-point unit that consists of one floating-point adder, one multiplier and one divider. The operations of the processors are controlled by a centralized system controller, which is also



(a) MFLOPS

(b) Overhead

Figure 11: Performance of Architecture 1



(a) MFLOPS

(b) Overhead

Figure 12: Performance of Architecture 2

a Nios processor. The scheduling algorithm used by the controller has significant affect on the performance of the design. With 5 Nios processors, this design can at most achieve a speedup of 4.37 over a uni-processor machine. In our design, however, we do not employ centralized control for scheduling. Moreover, we have performed optimizations specific to SpMXV, while the design in [24] aims to provide an FPGA-based general purpose multiprocessor machine.

We also compare our design with a general-purpose processor system. The system consists of a dual 900 MHz Itanium 2 processor, an L1 cache of 16 KB, an L2 cache of 256 KB, and a peak memory bandwidth of 8.5 GB/s. Note that the dense vector x for each test matrix fits well in the L2 cache of the Itanium system. The system runs the Red Hat Linux Advanced Server 2.1 Operating System. For SpMXV on the Itanium system, we use SPARSITY [10]. As far as we know, this system has reported the highest performance for floating-point SpMXV on general-purpose processors. The programs were compiled using the Intel C Compiler 7.0 for Linux, with -fast (highest optimization, static linking, interprocedural optimization) and -Ob2 (allows the compiler to inline any function that it sees fit) optimizations. The compiled code used only one of the two processors in the system.

We compare the Itanium system with the implementation of Architecture 1 with $k = 4$, which requires a memory bandwidth of 8 GB/s. This memory bandwidth can be provided by existing FPGA modules, e.g. BenDATA-WS of Nallatech [16]. BenDATA-WS consists of six independent ZBT SRAM banks, and each memory bank contains a 64-bit data bus. We made a conservative performance estimation by deducting 30% off the performance of our design for the control overhead of the SRAM devices [22]. The MFLOPS speedup of our design over the Itanium 2 system is shown in Figure 15. The speedup increases as the matrix number increases. Our design performs best over the general-purpose processors for matrices with most irregular sparsity structure, such as matrix 10 and 11. The reason is the performance of our design depends on the number of nonzeros in each row. Even if the nonzeros are distributed randomly, the performance of our design is not af-

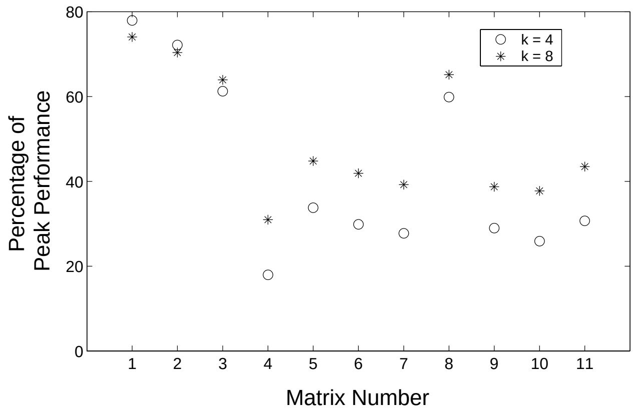


Figure 14: The percentage of peak performance that Architecture 1 achieves. When the available memory bandwidth is 8 GB/s, $k = 4$; when the available memory bandwidth is 14.4 GB/s, $k = 8$.

fected. On the other hand, the random distribution of the nonzeros causes large numbers of cache misses in general-purpose systems.

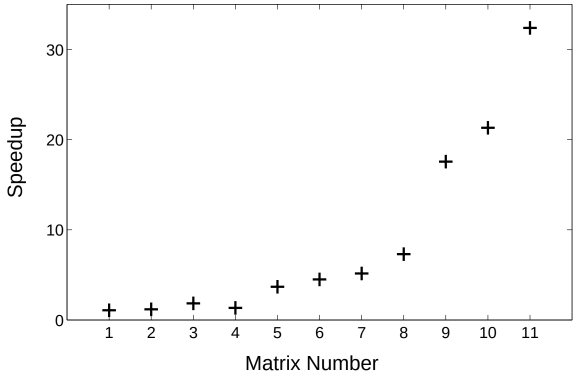


Figure 15: Speedup of Architecture 1 ($k = 4$) over the Itanium 2 system

5. CONCLUSION

We have proposed an FPGA-based design for floating-point SpMXV. Our design utilizes the commonly used CRS format for sparse matrices. The design requires no information on the sparsity structure of the input matrix. Various design parameters are identified. The values of these parameters are determined according to the input matrix, the floating-point units used, the hardware device and the available memory bandwidth. Thus, our design can be adapted to a variety of scenarios with different resource constraints. When implemented on a Xilinx Virtex-II Pro XC2VP70 and with a memory bandwidth of 8 GB/s, our design achieved over 350 MFLOPS for all the test matrices. Our design achieved high speedup over an Itanium 2 processor for matrices with very irregular sparsity structure. Through our work, we have shown that FPGAs provide a good alternative for implementations of I/O-bound applications. We also developed a flexible tree-based design which can be applied to many other applications. In the future, we plan to implement the design for SpMXV on an FPGA-based computer, such as the MAPStation of SRC [20]. We also plan to further generalize the tree-based design and develop an FPGA-based library for linear algebra applications.

Acknowledgement

The authors thank Cong Zhang, Gerald Morris and Ronald Scrofano for valuable discussions and useful feedback on earlier versions of this manuscript.

6. REFERENCES

[1] Altera Corporation. `http://www.altera.com`.

[2] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H. van der Vorst. *Templates for the Solution of Algebraic Eigenvalue Problems: A Practical Guide*. Society for Industrial and Applied Mathematics, 2000.

[3] G. Benkrid, D. Crookes, J. Smith, and K. Benkrid. High Level Programming for FPGA based Image and Video Processing Using Hardware Skeletons. In *Proceedings of IEEE Symposium on Field Programmable Custom Computing Machines (FCCM'01)*, California, USA, April 2001.

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

[5] H. A. ElGindy and Y. L. Shue. On Sparse Matrix-Vector Multiplication with FPGA-based System. In *Proceedings of The 10th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'02)*, California, USA, April 2002.

[6] G. Govindu, S. Choi, and V. K. Prasanna. Efficient Floating-Point Based Block LU Decomposition on FPGAs. In *Proceedings of the 11th Reconfigurable Architectures Workshop*, New Mexico, USA, April 2004.

[7] G. Govindu, L. Zhuo, S. Choi, and V. K. Prasanna. Analysis of High-Performance Floating-Point Arithmetic on FPGAs. In *Proceedings of the 11th Reconfigurable Architectures Workshop*, New Mexico, USA, April 2004.

[8] L. S. Heath, S. V. Pemmaraju, and C. J. Ribbens. Sparse Matrix-Vector Multiplication on A Small Linear Array. Technical report, Tech 93-11, Department of Computer Science, University of Iowa, 1993.

[9] B. Hendrickson, R. Leland, and S. Plimpton. An Efficient Parallel Algorithm for Matrix-Vector Multiplication. *International Journal of High Speed Computing (IJHSC)*, 7(1), 1995.

[10] E. J. Im, K. A. Yelick, and R. Vuduc. SPARSITY: An Optimization Framework for Sparse Matrix Kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, February 2004.

[11] Institute of Electrical and Electronics Engineers. *IEEE 754 Standard for Binary Floating-Point Arithmetic*. 1984.

[12] Y. M. Kadah. A New Solution to the Gridding Problem. In *Proceedings of SPIE Medical Imaging*, California, USA, February 2002.

[13] G. Manzini. Lower Bounds for Sparse Matrix Vector Multiplication on Hypercubic Networks. *Discrete Mathematics and Theoretical Computer Science*, 2(1):35–47, 1998.

[14] Mentor Graphics Corp. `http://www.mentor.com/`.

[15] G. R. Morris, L. Zhuo, and V. K. Prasanna. A High-Performance FPGA-Based General Reduction Method. manuscript in preparation, December 2004.

[16] Nallatech. `http://www.nallatech.com`.

[17] A. Pinar and M. T. Heath. Improving Performance of Sparse

Matrix-Vector Multiplication. In *Proceedings of Supercomputing 1999*, Oregon, USA, November 1999.

[18] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes in C : The Art of Scientific Computing*. Cambridge University Press, 1992.

[19] R. Scrofano and V. K. Prasanna. Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware. In *International Conference on Engineering of Reconfigurable Systems and Algorithms*, June 2004.

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

[21] S. Toledo. Improving Memory-System Performance of Sparse Matrix-Vector Multiplication. *IBM Journal of Research and Development*, 41(6):711–725, 1997.

[22] K. D. Underwood and K. S. Hemmert. Closing the Gap: CPU and FPGA Trends in Sustainable Floating-Point BLAS Performance. In *Proceedings of 2004 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM'04)*, California, USA, April 2004.

[23] University of Florida Sparse Matrix Collection. `http://www.cise.ufl.edu/research/sparse/matrices/`.

[24] X. Wang and S. G. Ziavras. Performance Optimization of an FPGA-Based Configurable Multiprocessor for Matrix Operations. In *Proceedings of IEEE International Conference on Field-Programmable Technology*, Tokyo, Japanese, December 2003.

[25] Xilinx Incorporated. `http://www.xilinx.com`.

[26] L. Zhuo, G. R. Morris, and V. K. Prasanna. Designing Scalable FPGA-Based Reduction Circuits Using Pipelined Floating-Point Cores. submitted to the 12th Reconfigurable Architectures Workshop, April 2005.

[27] L. Zhuo and V. K. Prasanna. Scalable and Modular Algorithms for Floating-Point Matrix Multiplication on FPGAs. In *Proceedings of The 18th International Parallel & Distributed Processing Symposium*, New Mexico, USA, April 2004.