Mapping Sparse Matrix-Vector Multiplication on FPGAs

Junqing Sun[1], Gregory Peterson[1], Olaf Storaasli[2]
University of Tennessee, Knoxville[1]
Oak Ridge National Laboratory[2]
[jsun5, gdp]@utk.edu[1], Olaf@ornl.gov[2]

Abstract

Higher peak performance on Field Programmable Gate Arrays (FPGAs) than on microprocessors was shown for sparse matrix vector multiplication (SpMxV) accelerator designs. However due to the frequent memory movement in SpMxV, system performance is heavily affected by memory bandwidth and overheads in real applications. In this paper, we introduce an innovative SpMxV Solver, designed for FPGAs, SSF. Besides high computational throughput, system performance is optimized by minimizing and overlapping I/O operations, reducing initialization time and overhead, and increasing scalability. The potential of using mixed (64-bit, 32-bit) data formats to increase system performance is also explored. SSF accepts any matrix size and easily adapts to different data formats. SSF minimizes resource costs and uses concise control logic by taking advantage of the data flow via innovative floating point accumulation logic. To analyze the performance, a performance model is defined for SpMxV on FPGAs. Compared to microprocessors, SSF has speedups up to 20x and depends less on the sparsity structure.

Keywords

FPGA, Performance, Sparse Matrix

1. Introduction

Sparse matrix-vector multiplication (SpMxV), $y = Ax$, is one of the most important computational kernels in scientific computing, such as iterative linear equation solvers, least square and eigenvalue solvers [1]. To save storage and computational resources, usually only nonzero elements are stored and computed. Pointers are necessary to store the sparsity structure, but degrades memory operation efficiency since the vector 'x' is addressed by the pointers during computation. Furthermore, pointers require additional load operations and memory traffic. Despite numerous efforts to improve SpMxV performance on microprocessors [2]-[4], these algorithms rely heavily on the matrix sparsity structures and the computer architecture, resulting in degraded performance on irregular matrices.

FPGAs show great potential in Reconfigurable Computing because of their intrinsic parallelism, pipeline ability, and flexible architecture. With their rapid increase in gate capacity and frequency, FPGAs overcome microprocessors on both integer and floating point operations [5]. Many computational intensive algorithms achieve significant speedup on FPGAs when the I/O bandwidth requirement is low [6], [7], [21]-[26]. Application of reconfigurable computing now depends on effective system integration to effectively utilize these powerful accelerators to improve the overall performance. Although the results for one FPGA chip are promising, overall performance is often limited by the I/O bandwidth [8]. To efficiently integrate FPGA accelerators into a balanced computing system remains an open problem [9].

Several FPGA designs for SpMxV have been reported before. Zhuo and Prasanna designed an adder-tree-based SpMxV implementation for double precision floating point that accepts any size matrices in general CRS format. ElGindy and Shue proposed SpMxV on FPGA for fixed point data [18]. DeLorimier and DeHon arranged the PEs in a bidirectional ring to compute the equation $y = A^i x$, where A is a square matrix while i is an integer. The design they proposed reduces the I/O bandwidth requirement greatly by sharing the results between PEs. Because local memories are used to store the matrix and intermediate results, the matrix size is limited by the on-chip memory [11]. El-kurdi et al proposed stream-through architecture for finite element method matrices [12].

Because of its importance in scientific computing, we plan to develop FPGA libraries for Basic Linear Algebra Subprograms (BLAS). The designs will target the Cray XD-1 and XT-4 machines at the Oak Ridge National Lab (ORNL). To the best of our knowledge, the design in [10] reports the highest performance for single cores and minimum I/O requirements for

SpMxV on FPGAs. However, a reduction circuit is used in their design that needs to be changed according to different matrix parameters [10].

In this paper, we introduce an innovative SpMxV design for FPGAs. Because the hardware does not need to change for different matrices, the initialization time is minimized and the system integration is simplified. We reduce the I/O operations for our design by modifying the traditional Block Compressed Row Storage (BCRS) format. Due to the simpler control logic, our design has a better scalability than previous work.

Because floating point adders are usually deeply pipelined to achieve high frequency, accumulating floating point data is normally difficult in digital design. We propose an accumulation circuit for SpMxV. By taking the advantage of the data flow, we design an innovative summation circuit which has low resource requirements and simple control logic.

Long size integer or floating point data are normally used for accuracy but also require more computational resources and I/O bandwidth, and result in longer latency. Potentially, mixed size data can be used to meet the required accuracy while achieving higher performance [13]. We explore this idea in SpMxV. Our preliminary results prove its significant savings in on-chip resources and I/O bandwidth while achieving higher frequency.

This paper also built a performance model for reconfigurable SpMxV accelerators. We discuss the impact on SpMxV performance from different factors, such as computational ability, I/O bandwidth, initialization time, synchronization time and other overheads. Our model shows that the performance of SpMxV on FPGAs depends on two primary factors: I/O bandwidth and computational ability, expressed as "chip capacity times frequency".

This rest of this paper is organized as follows. Firstly, we introduce the SpMxV algorithm and storage format used in our design. Section 3 introduces our basic design and application frameworks. The design for floating point is discussed in section 4. Implementation results are also given and compared to previous work. Section 5 discusses possible improvements of our design. In section 6, we describe a performance model, which is used to optimize the design parameters and estimate performance. Our performance is also compared to both previous designs and optimized algorithms on microprocessors.

2. SpMxV on FPGAs

The storage format plays an important role in SpMxV and affects the performance of optimization algorithms. We use the common format, Compressed Row Storage (CRS), for our FPGA design [14]. Our design requires the multiplicand vector x be stored in the FPGA local memory. For large problems, where x cannot be accommodated in a FPGA chip, we use block matrix multiplication algorithm. In contrast to traditional BCRS Storage, our matrix storage format is optimized for FPGA accelerators. As explained later, this format is compatible with algorithms using BCRS but reduces requirements on both I/O bandwidth and computational resources.

In general, the SpMxV computation $y = Ax$ is defined as:

$$y_i = \sum_{j=0}^{N} a_{i,j} x_j \ , \ (0 \le i < M) \qquad (1)$$

Where A is an $M \times N$ matrix, while y and x are $M \times 1$ and $N \times 1$ vectors, respectively. For efficiency, most sparse matrix algorithms and storage formats only operate on nonzero elements. For each nonzero element, there are two floating point operations (one add and one multiply). By convention, we assume A has n_{nz} nonzero elements. All the elements of A and have to be moved into the FPGAs, while computed results for y have to be moved out of the FPGAs. Because of the pointers used in storage formats, the indices for matrix A also need to be moved into FPGAs' local memories. Suppose there are n_p pointers needed. The total I/O requirement is at least:

$$n_{IO} = n_{nz} + n_p + N + M \ . \qquad (2)$$

Because of the loss of locality and limited memory size, matrix and vector data may have to be moved multiple times on traditional microprocessor-memory architectures. In the FPGA SpMxV design, I/O time is hidden by overlapping with computations to reduce the overall time. The time used to preload the data onto FPGAs is denoted as T_{Init}, which also includes hardware initialization and data formatting. We denote T_{syn} as the time for FPGAs to synchronize with hosts, and $T_{overhead}$ for other overheads. The overall time spent on FPGA accelerators is thus

$$T = \max(T_{comp}, T_{IO}) + T_{init} + T_{syn} + T_{overhead} \qquad (3)$$

In equation (3), computation time T_{comp} is the only part doing real matrix multiplication operations. Unfortunately, SpMxV FPGA cores also have tremendous I/O demands. To improve the overall performance, we need to overlap the I/O operations with computations as much as possible. At the same time, synchronization time and overheads needs to be minimized.

2.1 Sparse matrix storage format

The CRS format makes no assumptions about the sparsity structure of the matrix and has no unnecessary elements stored [14]. In the CRS format, 3 vectors are needed: the val vector stores subsequent nonzeros of the matrix in row order; the integer vector col stores the column indices of the elements in the val vector; while the integer vector len stores the number of non-zero elements of each row in the original matrix. As an example, consider the matrix A defined by

$$A = \begin{pmatrix} 2 & 0 & -3 & 0 & 0 \\ 0 & -1 & 0 & 0 & 6 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 \\ 5 & 8 & 0 & 6 & 0 \end{pmatrix}$$

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

Val:	2, -3, -1, 6, 1, 9, 5, 8, 6
Col:	0, 2, 1, 4, 0, 1, 0, 1, 3
Len:	2, 2, 1, 1,3

In our design, the maximum matrix size that can be fit into FPGA chips is restricted by the on-chip memory size. Big matrices need to be divided into sub-matrices. Our matrix division format is shown in Figure 1. The matrix is divided into stripes in row's direction. Each stripe is further divided to sub-matrices (shown in dashed lines). The sub-matrices are computed in the FPGAs, and those having only zeros are neither stored nor computed. We refer to this format as Row Blocked CRS (RBCRS).

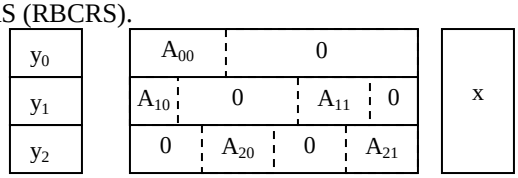

Figure 1: Row Blocked CRS (RBCRS)

During the computation, sub-matrices in the same stripe are assigned to the same FPGA accelerator. Note that the elements required from vector x will differ for each stripe based on the sparsity pattern. The vector x is kept in the FPGA off-chip memory, and part of it (x_j) is loaded before computing $A_{ij}x_j$. Note that the result $A_{ij}x_j$ is not sent out after being computed, but is stored in the FPGA and added with the result from the next sub-matrix vector multiplication in the same stripe. For example, the result of $A_{20} \times x_0$ will be stored in the FPGA to add with the result of $A_{21} \times x_1$. After all the matrices in a row are computed, the result y_2 is read out. This approach saves I/O bandwidth and computational resources.

3. Framework and basic design

This section introduces our basic design and the framework when used in software applications. The basic design discussed here is efficient for integers rather floating points. The reason is that the integer adders have one-clock-cycle latency, and therefore the accumulation circuit can be built with a simpler pipelined structure. The summation circuit can also be simply implemented by using an adder. This basic design could be adopted to support floating points, but at a lower clock frequency. The design for deeply pipelined floating point operators is more complicated because of the read after write hazard discussed in the next section.

3.1 Basic design and interfaces

Figure 2 shows the basic design of our SpMxV Solver designed for FPGAs (SSF) core and the framework for applications. The application program stores the matrix in RBCRS format. The matrix manager feeds sub-matrices to the SpMxV core in CRS format and reads back the result y_i. The application program may read back the y_i values from different FPGAs to determine the result y.

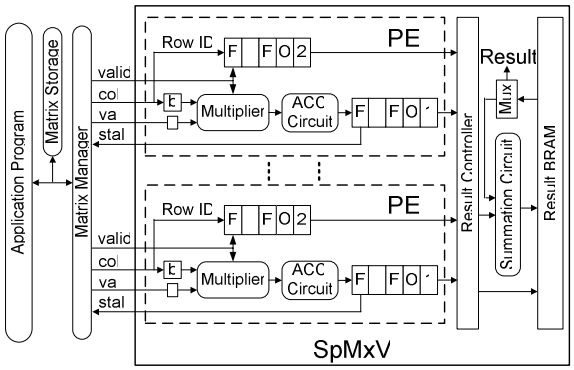

Figure 2: Data Path and Framework of SpMxV Design

In our design, each PE is a deep pipeline consisting of a multiplier, adder, and result adder. FIFO1 is used as a buffer for intermediate results. The values for "val" and "col" are synchronously imported into the PE. The multiplicand vector x_j is preloaded into the FPGA and addressed by "col". Because there is a one clock cycle latency to read data from Block RAM (BRAM), a buffer is inserted for "val" before the multipliers. The input signals are illustrated in Figure 3. Following the "col" values for each row, one clock

cycle is inserted for its Row ID in the sub-matrix. The Row IDs are stored into FIFO2 and used to address the Result BRAM. Zeros are inserted when there is a stall signal or when waiting for the I/O to feed the next rows. The signal "valid" is set when "val" and "col" data are being imported. It is also used to control the components: multiplier, adder, FIFO1, and FIFO2.

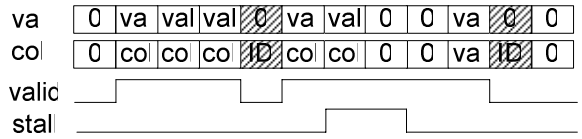

Figure 3: Signals for Processing Elements (PEs)

For a row with n_{nz} nonzero elements, the PE computes n_{nz} values and stores the results into FIFO1. The summation circuit adds the results from the PEs with the data in the result BRAM addressed by data read from FIFO2. Note that the data in the result BRAM are from previous sub-matrices.

To maximally utilize the data input bandwidth, all the components in the PEs are synchronized to the pipelined data flow by using the signal "valid". Some intermediate signals are produced to indicate when the components have valid inputs and outputs. Most of these control signals can be generated by adding appropriate delays to the signal "valid". For example, the "input valid" signal for a multiplier is produced by adding one clock cycle delay to the signal "valid" because of the one clock cycle delay in the data flow. The "write enable" signal for FIFO1 is set for one clock cycle when the result for one row is accumulated. The "stall" signal is set if FIFO1 is close to full. When a stall is issued, zeroes are inserted as inputs while the "valid" signal does not change. If a row is being imported, the multipliers and accumulators operate on inserted zeros and will have no affect on the results. Note that the data already in the pipelines of multipliers and ACC circuits still need to be computed and stored into FIFO1. Therefore, FIFO1 needs to have certain free space when a stall signal is issued. The size of the free space should be bigger than half of the total pipeline stages of an adder and a multiplier.

The Result Controller checks the "empty" signal of all the FIFOs. If the FIFO is not empty, the result will be read out and added to the corresponding value in Result BRAM. The row ID is read out at the same time as the data address. The purpose of using the adder is to sum the results from all sub-matrices in the same stripe as explained in section 2.1. The result BRAM will be read and cleared when all the sub-matrices in the same stripe are computed.

4. Complete design

4.1 Pipelined accumulation circuit

Pipelined floating point operators can be used to improve the frequency of our design. However, the accumulator cannot be simply built as in the basic design because of read after write data hazards. The dataflow for a 5 stage pipelined floating point adder is shown in Figure 4. The hashed blocks are inserted zeros, which come when the valid signal is zero (invalid). The second row is the outputs when the output pin of the floating point adder is connected to one of the input pins. There are three problems in this circuit.

1. The output is not accumulated into a single value as in the integer design. For example, the first row has 6 numbers with a summation of 21. The circuit gives 5 outputs (2, 3, 4, 5, and 7).
2. The data is added to the output of previous rows. For example, 8, 9 and 10 are added to 3, 4 and 7.
3. To solve the first problem, we can use 5 registers to store the last 5 outputs of each row. However, these registers will have results from previous rows when the current row is short. For example, if 5 registers are used to store the outputs from the second row, the data should be captured once 13 (the correct output is 9) comes out. However, 5, 7 and 2 are also stored.

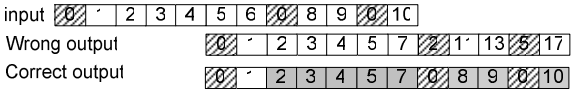

Figure 4: Data Flow for Pipelined ACC Circuit

To solve the data hazards mentioned above, we design an ACC circuit with a pipelined floating point adder. One of the inputs, (a), is connected to the output of multiplier and works as the input for ACC circuit. The last 5 outputs of the adder are stored in 5 registers as the output of the ACC circuit. The correct outputs from our design are also in Figure 4, where the blocks in grey are data stored in registers. For example, 8 and 9 are stored in two registers as the output of 8 and 9 in the first line. The other 3 registers have just zeros.

4.2 Adder tree

For pipelined adders with L clock cycle latency, L outputs will be stored into FIFO1 to add with the data in the result BRAM. One possible approach for this problem is to add these L outputs by an adder tree. Suppose L is equal to 4, we need to add 4 data from the FIFO and one value from the result BRAM. For these 5 inputs, an adder tree with 3 levels and 4 adders are needed as shown in Figure 5. If the number of inputs

is not a power of two, shifters with latency L can be used in an adder tree to take the place of adders to save resources.

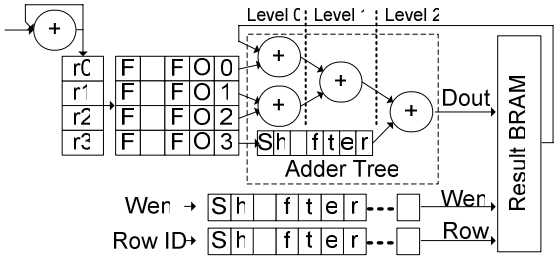

Figure 5: Adder Tree Used for Pipelined Adders

For our design with double precision data, 12 outputs from the FIFO and one data from the result BRAM need to be added. We use 13 floating point adders to build the adder tree, which costs 25% of the total slices of a Xilinx XC2VP70 FPGA. The data flow of the adder tree used in our design in shown in Figure 6. The rectangles represent data. The numbers in rectangles are the clock cycles when that data is available. The dashed line is a FIFO with a latency of 24 clock cycles. The final result comes out 48 clock cycles after the inputs are available, so it is very important to capture the output at the right clock cycle. We input the row ID and write enable signal to two shifters with depth of 48 at clock 0. They will come out with the result at clock 48 to be used as the address and write enable signal for the result BRAM.

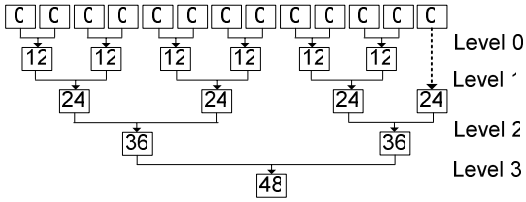

Figure 6: Data Flow for Adder Tree

4.3 Reduced summation circuit

Because of the large adder tree, we propose a reduced summation circuit as shown in Figure 7. The idea is to reduce the number of adders by importing just two data each clock cycle. The data coming out first is stored in a buffer and computed with the next. By inserting a certain number of buffers between the adders and taking advantage of the data flow, we designed a summation circuit for this function without control logic. For our double precision design, 4 adders and 7 buffers are used in total. 16 registers are used to store the data from the FIFO, the result Block RAM, and 3 zeros to fill the pipeline for correctness. This will be explained later in the data flow.

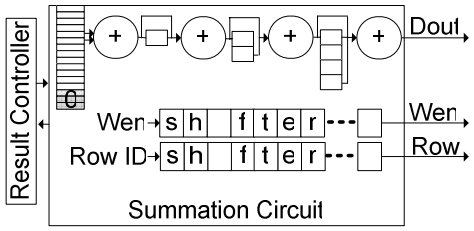

Figure 7: Reduced Summation Circuit

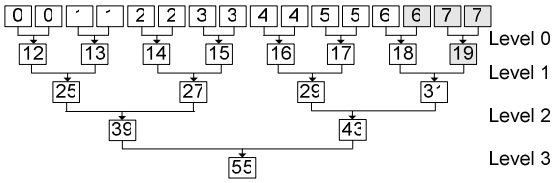

Figure 8: Data Flow for Summation Circuit

The data flow here is more complicated than in the adder tree, as shown in Figure 8. The data in a row are added by the same adder in serial, while buffers are used to delay the intermediate data for the appropriate time. For example, the datum on clock 12 should be added to that on clock 13, so a buffer needs to be added before adder 1. We can see that the data on clock cycle 18 does not have a counterpart for the addition operation. We pad with zeros to obtain the correct sum. The shaded rectangles are inserted zeros. Figure 8 shows that the final result can be captured 55 clock cycles after the data is available in the buffer. In our design, a "Write Enable" signal for the result BRAM is stored to a shifter with length of 55 at clock 0. When the "Write Enable" comes out of the shifter, the final result will also be ready.

The Result Controllers of these two circuits are very similar. Because of their long latency, the Result Controller uses stream-through architecture. We insert the row ID and write enable signals to be written into shifters at clock cycle 0. If the three outputs of these two circuits are connected to corresponding pins of the result BRAM, the results should be captured automatically. Table 1 compares the summation circuit and adder tree.

Table 1: Comparing Adder Tree and Summation Circuit

Design	Adders No.	latency
Adder Tree	12	48
Summation Circuit	4	55

4.4 Implementation results and comparison

We implemented SSF design by using Xilinx ISE and EDK 8.1 [17]. ModelSim and Chipscope are used for verification and debugging. For mathematic opera-

tions, we use Xilinx IP cores which follow the IEEE 754 standard and that can also be customized [17]. Considering the limited size of the FPGAs, we use a summation circuit for the floating point design. The BRAM size for x_i, y_i are 1000. The adders and multipliers are provided by Xilinx [17]. To compare our results with previously reported designs, we target the Xilinx XC2VP70-7. The characteristics are summarized in Table 2.

Table 2: Characteristics of SpMxV on XC2VP70-7

Design	64 bit Int	Single FP	Double FP
Achievable Frequency	175MHz	200MHz	165MHz
Slices	8282(25%)	10528 (31%)	24129 (72%)
BRAMs	36 (10%)	50 (15%)	92 (28%)
MULT18X18	128 (39%)	32 (9%)	128 (39%)

The slice usage and the frequency of our design are dominated by the mathematic operators, while the effect from control logic is almost negligible. If high speed floating point operators are used, the speed of our design can be improved accordingly. Our design can easily adapt to different data formats by simply replacing IP cores. The only change for the control logic is the latency of operators and interface width, which are defined as a variable in the VHDL. Our design is deeply pipelined. Ignoring I/O bandwidth limitations and communication overheads, two floating point or integer operations (one addition and one multiplication) can be done per clock cycle by each PE.

Previously reported work describes an implementation that achieves 2340 MIPS at 28.57 MHz frequency by using 3 multipliers [18]. However, that design is for fixed point data. The closest related work is [10], which develops an adder-tree-based design for double precision floating point numbers. A reduction circuit is used in their design to sum up the floating points. Because the frequency is mostly dependent on the floating point operations for both designs, the achievable speed is similar in these two designs if the same mathematical IP cores are used. When 8 multipliers are utilized, both designs achieve a peak performance of 16 floating point operation per clock cycle. Their design uses high performance floating point cores with clock latencies of 19 for the adder and 12 for the multiplier. The number of adders depends on the size of the reduction circuit, which changes with different matrices. For the test matrices in [10], the size of reduction circuit is 7. Our design accepts any input matrices with no hardware changes required. There is no a priori analysis on the matrix or extra hardware initialization time needed for our design. For the tree based design [10], zeros need to be padded when the number of nonzero in a row is not a multiple of the number of multipliers. To reduce the overhead caused by zero padding, [10] uses a technique called merging. As the PE number increases, the tree based design will face a choice between high overhead and complicated control logic [10]. Our design does not have the zero padding overhead and can scale very easily.

5. Potential performance

5.1 Mixed data format

One important advantage of FPGAs over microprocessors is that they allow customized data formats. As Table 2 shows, different data formats result in different frequency and resource consumption. The bus traffic could also be reduced by using smaller size data.

We try to increase the performance of SpMxV by applying shorter data formats as much as we can. The potential impact on performance improvement is explained here by a simple example. Suppose 32 bit integers provide enough resolution for the matrices and vectors given. However, it is possible that the output data are bigger, and 64 bit integers are needed. Instead of using 64 bit data for both input and output data, we can use two different data formats: 32 bits for input and 64 bits for the output. Table 3 shows that a design with mixed data formats has higher frequency, lower latency, and less I/O bandwidth and resources.

Table 3: 64 bit and 32/64 bit Mixed Integer Comparison

Design	32/64 bit Mixed	64 bit
Achievable Frequency	183Mhz	175Mhz
Slices	3475 (10%)	8282 (25%)
BRAMs	20 (6%)	36 (10%)
MULT18X18	32 (9%)	128 (39%)
Multiplier Latency	4 cycles	6 cycles
Required I/O Bandwidth	8.8GB/s	14GB/s

5.2 Reducing Summation Circuit Latency

In our design, the summation circuit is shared by all the PEs to add the data from the FIFOs and the result BRAM. When the design scales up, care must be taken that it will not become the bottleneck of the whole pipeline. That is, the time the result adder uses to transport data should be overlapped by communication or computation time. We analyze this potential bottleneck problem by considering the reduced summation circuit because it takes the most time (8 clock cycles). We compare the time the I/O and the summation circuit

needs to transport data when each PE computes just one row. For a design with 8 PEs, the time needed by the result adder to transport data is $8 \times 8 = 64$ clock cycles.

The communication time is decided by the I/O bandwidth and matrix sparsity. On the Cray XD-1, the peak speed for the bus between FPGA chip and QDR II RAM is 1.6GB/s in each direction [15]. Suppose the matrix sparsity is 1% and sub-matrix size is 1000 by 1000. Then on average, there are 10 double precision floating point data (8 Bytes) for val and 10 integer index data (2 Bytes) for each col, that is 100 Bytes per column. Even assuming the I/O bandwidth can be fully utilized with no other communication overheads, the communication time for the double precision floating point design is at least $100 \times 8 \times 165Mhz / 1.6G = 83$ clock cycles for 8 engines. The overhead for 8 PEs results in an extra $10 \times 8 \times 165Mhz / 1.6G = 8$ clock cycles for a total of 91 clock cycles.

If more PEs are implemented, the time spent by both the result adder and I/O operations increases linearly. Therefore, the I/O is still the bottleneck instead of the adder tree under the conditions above. If the sub-matrix size increases, the time spent on I/O will increase accordingly. Therefore, the summation circuit has less possibility to become the bottleneck. In cases that faster I/Os are used, the time for I/O will be smaller and may not overlap the time for the summation circuit. Multiple summation circuits can work in parallel to increase the throughput until it is overlapped by the communication or computation time.

6. Performance

6.1 Performance model for SpMxV on FPGAs

In our design, the time for moving "val" and "col" into FPGAs is overlapped with the computation time. When a sub-matrix is being computed, the multiplicand vector x_i for the next matrices can be loaded. The I/O time for x_i ($i \geq 1$) can be overlapped, so it is not included here. The time for initialization and synchronization should also be counted, so the total time spent by SpMxV core is

$$T = \max(T_{comp}, T_{IO}) + T_{init} + T_{syn} + T_{overhead} \quad (4)$$

In equation 4, the real computational work only contributes T_{comp} to the total time. To increase overall performance, we need to overlap the communication time and reduce the initialization and synchronization time besides reducing T_{comp}.

T_{comp} is determined by the frequency and number of computational engines. We assume F floating point operations are executed per second. The communication time is limited by the host memory bandwidth and by the I/O bus speed. Suppose the bandwidth for each I/O bus is B_{IO} and that the matrix *A* and vector *y* are transported by separate I/O buses. To compute a nonzero element, both its value and pointer have to be moved into FPGAs. The time spent on the FPGA accelerator is thus

$$T = \max\left(\frac{2n_{nz}^*}{F}, \frac{n_{nz}^* \times (val\ width + col\ width)}{B_{IO}}\right) + T_{init} + T_{syn} + T_{overhead} \quad (5)$$

Where n_{nz}^* is the number of total nonzero elements for all sub-matrices assigned to an FPGA accelerator.

To minimize equation 4 and 5, we have discussed several approaches to accelerate the computation: increasing the frequency and number of PEs to improve F; optimizing the matrix mapping to reduce I/O operations; making the design general to all different matrices so no hardware initialization or preparation on inputs is required; designing a simple interface which only needs a start signal and matrix/vector address; and not requiring any participation of the host during the computation.

We still need to discuss the block RAM size. The effect from the block size of x_i is a double edged sword. The overheads in our design mainly come from the one clock cycle control signal between rows. Therefore increasing the block size of x_i reduces the ratio of overheads by having more nonzeros each row. However, it also results in a longer initialization time for loading x_0. The result BRAM size determines the number of rows of sub-matrices, which affects the number of nonzero elements of sub-matrices. Under certain sparsity, the I/O time to move sub vectors x_j can be overlapped with a big enough result BRAM size.

For very large matrices, many sub-matrices will be assigned to a particular FPGA. T_{init} in our design comes from loading x_o and can be ignored in that case. The synchronization time with hosts is also just a function call, so we can also neglect T_{syn} for simplicity. For the double precision design, the data width is 8 Bytes and index width is 2 Bytes. So equation 5 becomes:

$$T < \max\left(\frac{2n_{nz}^*}{F}, \frac{10n_{nz}^*}{B}\right) \quad (6)$$

If unlimited resources are assumed, F is also infinite. Then the achievable MFLOPS performance is limited by B.

$$MFLOPS = \frac{2n_z}{T} < \frac{2n_z}{10n_z / B} = B/5 \quad (7)$$

The floating point operations F take advantage of both the frequency and capacity of FPGAs and result in 4 times improvement every two years [5]. However to build a balanced system, the number of PEs is limited not just by chip capacity but also I/O bandwidth. For double precision floating point discussed before, the maximum number limited by I/O bandwidth is:

$$\text{PEs} \le \frac{B/5}{2\,frequency} = \frac{B}{10\,frequency} \quad (8)$$

To the best of our knowledge, the work in [10] reports the highest previous performance for SpMxV on FPGAs. Given the same size design as shown in Table 2, we have similar peak performance and I/O requirements. However, our design does not need to change the hardware for different matrices, so the initialization and synchronization time is shorter. We also do not suffer from either high overheads or very complicated control logic due to zero padding when the system scales. For large matrices, the results from the design [10] are just for sub-matrices and needs to be summed up for the final result. Our design allows saving the immediate result in FPGA and computes the final result without this additional I/O operation requirement.

6.2 Comparison with previous work

Table 4: Test Matrices [20]

ID	Matrix	Area	Size (N)	Nonzeros (N_{nz})	Sparsity (%)
1	Crystk02	FEM Crystal	13965	968583	0.5
2	Crystk03	FEM Crystal	24695	1751178	0.29
3	stat96v1	linear programming	5995 x 197472	588798	0.05
4	nasasrb	Structure analysis	54870	2677324	0.09
5	raefsky4	Buckling problem	19779	1328611	0.34
6	ex11	3D steady flow	16614	1096948	0.4
7	rim	FEM fluid mechanics	22560	1014951	0.2
8	goodwin	FEM fluid mechanics	7320	324784	0.61
9	dbic1	linear programming	43200 x 226317	1081843	0.01
10	rail4284	Railways	4284 × 1092610	11279748	0.24

6.3 Comparison with microprocessors

In our design, the overhead mainly comes from the one clock cycle between continuous rows. It is decided by the total number of sub rows. The initialization time is for preloading sub vector X_0. Both of them can be exactly counted in simulation. The synchronization time is affected by the interface and API between host and FPGA chip. Our design needs few synchronization signals, such as "start", "complete", and start address of matrices/vectors. The synchronization time is neglected at this point. We test our design on matrices from different fields as shown in Table 4. All these matrices come from Tim Davis' Matrix Collection [20]. They are roughly ordered by increasing irregularity. The percentage of overheads in the test matrices is shown in Figure 9.

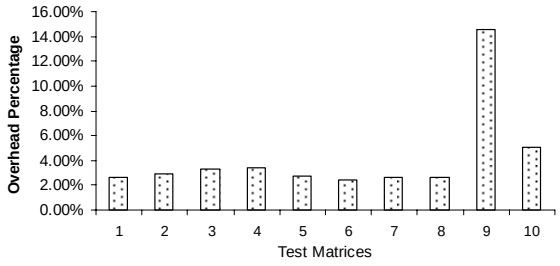

Figure 9: Overhead Percentage

We compare our design with microprocessors. Our design utilizes 8 PEs at 165 MHz frequency. The required memory bandwidth is 13.2 GB/s, which can be provided by current technology. For example, BenBLUE-V4 provides 16GB/s memory bandwidth [27]. We take a conservative performance estimation by

deducting 40% off the peak performance for control overhead of the high speed memory interface [5][10]. The achievable percentage of performance is shown in Figure 10.

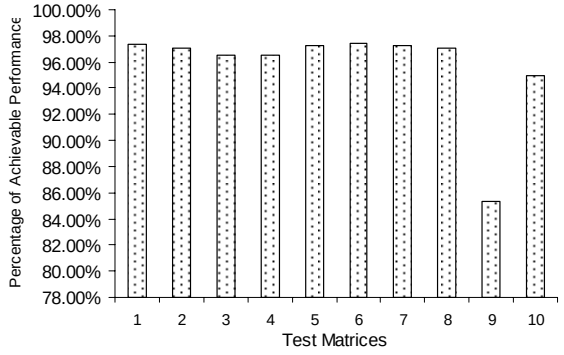

Figure 10: Percentage of Achievable Performance

For software performance on microprocessors, we use OSKI, which has achieved significant speedups by using techniques such as register and cache blocking [19]. The test machine is a dual 2.8GHz Intel Pentium 4 with 16KB L1, 512KB L2 Cache and 1GB memory.

The speedup of our design over the 2.8 GHz Pentium 4 is shown in Figure 11. Our design performs better then the Pentium 4 on matrices with irregular sparsity structures. This is because the overhead of our design depends on the number of nonzero elements per row of sub-matrices but is not affected by their sparsity structure.

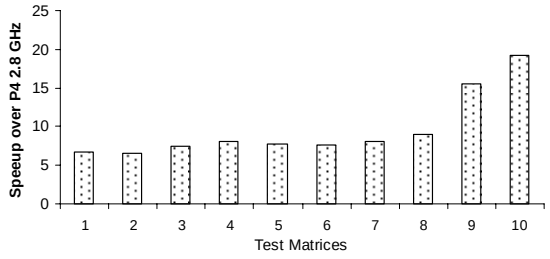

Figure 11: Speed Up over 2.8 GHz Pentium 4

7. Conclusions

We present an innovative SpMxV FPGA design with overall system performance addressed. First, we introduce an improvement for traditional BCRS, which results in lower I/O requirements and less overhead. Secondly, we propose an efficient multiplication accumulation circuit for pipelined floating points by taking advantage of the data flow. Compared to previous work, our design has high peak performance, low memory requirements, good scalability and does not need to modify the hardware for different matrices. Another contribution of this paper is the performance model for SpMxV on FPGAs, which considers the factors of computational ability, I/O, initialization time, synchronization time and overheads. Furthermore, we discuss the impact on performance from all these factors. The potential of using mixed data format for SpMxV is also explored. Our preliminary results show that, it results in higher frequency, lower I/O bandwidth and less resource requirements. Our future work includes completing high performance BLAS design on FPGAs and performance analysis.

8. Acknowledgement

This project is supported by the University of Tennessee Science Alliance and the ORNL Laboratory Director's Research and Development program. We also would like to thank Richard Barrett of ORNL for useful discussion on sparse matrices.

9. References

[1] 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.

[2] Storaasli. "Performance of NASA Equation Solvers on Computational Mechanics Applications", *34th AIAA Structures, Structural Dynamics and Materials Conference*, April, 1996.

[3] Pinar and M. T. Heath. "Improving Performance of Sparse Matrix-Vector Multiplication". *Supercomputing*, November 1999.

[4] E.-J. Im, K. A. Yelick. "Optimizing Sparse Matrix Computations for Register Reuse in Sparsity". *International Conference on Computational Science*, 2001.

[5] K. D. Underwood. "FPGAs vs. CPUs: Trends in peak floating-point performance". *ACM International Symposium on Field Programmable Gate Arrays*, February 2004.

[6] Y. Bi, G.D. Peterson, L. Warren, and R. Harrison. "Hardware Acceleration of Parallel Lagged-Fibonacci Pseudo Random Number Generation". *ERSA*, June 2006.

[7] R. Scrofano and V. K. Prasanna. "Computing Lennard-Jones Potentials and Forces with Reconfigurable Hardware". *ERSA*, June 2004.

[8] J.L. Tripp, A.A. Hanson, M. Gokhale, and H.S. Mortveit. "Partitioning Hardware and Software for

Reconfigurable Supercomputing Applications: A Case Study". *Supercomputing*, November 2005.

[9] K. Underwood, S. Hemmert, and C. Ulmer. "Architectures and APIs: Assessing Requirements for Delivering FPGA Performance to Applications". *Supercomputing*, November 2006.

[10] L. Zhuo and V. K. Prasanna. "Sparse matrix-vector multiplication on FPGAs". *2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, pages 63–74, February, 2005.

[11] M. deLorimier and A. DeHon. "Floating-Point Sparse Matrix-Vector Multiply for FPGAs". *International Symposium on Field Programmable Gate Arrays*, February, 2005.

[12] Y. El-kurdi, W. J. Gross, and D. Giannacopoulos. "Sparse Matrix-Vector Multiplication for Finite Element Method Matrices on FPGAs". *2006 IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2006.

[13] J. Langou, J. Langou, P. Luszczek, J. Kurzak, A. Buttari, and J. Dongarra. "Exploiting the performance of 32 bit floating point arithmetic in obtaining 64 bit accuracy". *Supercomputing*, November, 2006.

[14] R. Barrett, Templates for the solution of Linear Systems: Building Blocks for Iterative methods, 2nd Edition. *SLAM*, Philadelphia, PA, 1994.

[15] Cray Inc. www.cray.com

[16] Digilent Inc. www.digilentinc.com

[17] Xilinx Inc. http://www.xilinx.com

[18] H. A. ElGindy, and Y. L. Shue. "On Sparse Matrix-Vector Multiplication with FPGA-based System". *10th IEEE Symposium on Field-Programmable Custom Computing Machines*, April 2002.

[19] R. Vuduc, J. Demmel, K. Yelick. "OSKI: A library of automatically tuned sparse matrix kernels". SciDAC 2005, *Journal of Physics*: Conference Series, June 2005.

[20] T. Davis, University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices, NA Digest, 92(42), October 16, 1994, NA Digest, 96(28), July 23, 1996, and NA Digest, 97(23), June 7, 1997.

[21] Storaasli, "Compute Faster without CPUs: Engineering Applications on NASA's FPGA-based Hypercomputers", *Technical Symposium on Reconfigurable Computing with FPGAs*, Manchester UK, February 2005.

[22] J. Sobieski and O.O. Storaasli. "Computing at the Speed of Thought," *Aerospace America*, Oct. 2004, pp. 35-38.

[23] Storaasli, "Engineering Applications on NASA's FPGA-based Hypercomputer", *MAPLD*, September, 2004.

[24] Storaasli, "Computing Faster without CPUs: Scientific Applications on a Reconfigurable, FPGA-based Hypercomputer." *Military and Aerospace Programmable Logic Devices (MAPLD) Conference*, September, 2003.

[25] Storaasli, R. C. Singleterry, and S. Brown. "Scientific Computations on a NASA Reconfigurable Hypercomputer." *5th Military and Aerospace Programmable Logic Devices (MAPLD) Conference*. September, 2002.

[26] J. Sun, G. Peterson, O.O. Storaasli, "Sparse Matrix-vector Multiplication Design on FPGAs", *FCCM*, April, 2007.

[27] Nallatech Inc. http://www.nallatech.com.