A Scalable Sparse Matrix-Vector Multiplication Kernel for Energy-Efficient Sparse-Blas on FPGAs

Richard Dorrance
EE Department, UCLA
Los Angeles, CA 90095 USA
rdorrance@ucla.edu

Fengbo Ren
EE Department, UCLA
Los Angeles, CA 90095 USA
renfengbo@ucla.edu

Dejan Marković
EE Department, UCLA
Los Angeles, CA 90095 USA
dejan@ee.ucla.edu

ABSTRACT

Sparse Matrix-Vector Multiplication (SpMxV) is a widely used mathematical operation in many high-performance scientific and engineering applications. In recent years, tuned software libraries for multi-core microprocessors (CPUs) and graphics processing units (GPUs) have become the status quo for computing SpMxV. However, the computational throughput of these libraries for sparse matrices tends to be significantly lower than that of dense matrices, mostly due to the fact that the compression formats required to efficiently store sparse matrices mismatches traditional computing architectures. This paper describes an FPGA-based SpMxV kernel that is scalable to efficiently utilize the available memory bandwidth and computing resources. Benchmarking on a Virtex-5 SX95T FPGA demonstrates an average computational efficiency of 91.85%. The kernel achieves a peak computational efficiency of 99.8%, a >50x improvement over two Intel Core i7 processors (i7-2600 and i7-4770) and showing a >300x improvement over two NVIDA GPUs (GTX 660 and GTX Titan), when running the MKL and cuSPARSE sparse-BLAS libraries, respectively. In addition, the SpMxV FPGA kernel is able to achieve higher performance than its CPU and GPU counterparts, while using only 64 single-precision processing elements, with an overall 38-50x improvement in energy efficiency.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles—*Algorithms implemented in hardware*; G.1.3 [**Numerical Analysis**]: Numerical Linear Algebra—*Sparse, structured, and very large systems (direct and iterative methods)*

General Terms

Algorithms, performance

Keywords

SpMxV, sparse-BLAS, FPGA, CPU, GPU, energy-efficiency, computational efficiency, benchmarking

FPGA'14, February 26–28, 2014, Monterey, California, USA.

http://dx.doi.org/10.1145/2554688.2554785

1. INTRODUCTION

Sparse matrices arise in a wide variety of computational disciplines, including image reconstruction, circuit and economic modeling, industrial engineering, compressive sensing, neural networks, and algorithms for least squares and eigenvalue problems [1-3]. As such, Sparse Matrix-Vector Multiplication (SpMxV) is the main computational kernel that dominates the performance of many of the aforementioned applications. Unfortunately, the performance of SpMxV algorithms tends to be much lower than that of dense matrices, mostly due to the mismatch between the memory access patterns of sparse matrices and the compression formats required to efficiently store them [3,4].

Numerous efforts have been made to accelerate the performance of SpMxV on multi-core microprocessors (CPUs) [5,6] and graphics processing units (GPUs) over the years [7-9]. Recently, field-programmable gate arrays (FPGAs) have become an attractive option for accelerating SpMxV [1-4,10-14]. FPGAs have high floating-point performance, large amounts of on-chip memory, and an abundant number of high-speed I/O pins capable of providing large amounts of off-chip memory bandwidth. The flexible nature of FPGAs also allows architectural adaptations to satisfy the needs of different problems.

In this paper, we propose a scalable architecture for SpMxV with higher computational efficiency than traditional CPU/GPU-based approaches. Computational efficiency is a measure of the percentage of the total hardware resources available that are actively being used by an algorithm. An implementation of an algorithm with a higher computational efficiency will therefore be more energy-efficient. We leverage the structure of conventional sparse matrix compression formats for general sparse matrices in order to regularize their memory access patterns. The benchmarking results based on the FPGA implementation show that the proposed SpMxV kernel can reach significantly higher computational efficiency than state-of-the-art solutions using CPUs and GPUs, with more than a 50x and 300x improvement respectively. Even for very large, irregular, sparse matrices, our design can achieve performance comparable to that of dense matrices.

The remainder of the paper is organized as follows. Section 2 introduces SpMxV and discusses the inefficiencies present in existing software (powered by CPUs and GPUs) and hardware (FPGA) implementations. Section 3 details the proposed architecture to address these short comings. Benchmarking results on computational throughput and energy efficiency are presented in Section 4. Section 5 concludes the paper.

(a)
$$A = \begin{bmatrix} 1 & 4 & 0 & 0 & 0 \\ 0 & 2 & 3 & 0 & 0 \\ 5 & 0 & 0 & 7 & 8 \\ 0 & 0 & 9 & 0 & 6 \end{bmatrix}$$

(b)
$$COO \begin{cases} data = [1 \;\; 4 \;\; 2 \;\; 3 \;\; 5 \;\; 7 \;\; 8 \;\; 9 \;\; 6] \\ row = [0 \;\; 0 \;\; 1 \;\; 1 \;\; 2 \;\; 2 \;\; 2 \;\; 3 \;\; 3] \\ col = [0 \;\; 1 \;\; 1 \;\; 2 \;\; 0 \;\; 3 \;\; 4 \;\; 2 \;\; 4] \end{cases}$$

(c)
$$CSR \begin{cases} data = [1 \;\; 4 \;\; 2 \;\; 3 \;\; 5 \;\; 7 \;\; 8 \;\; 9 \;\; 6] \\ ptr = [0 \;\; 2 \;\; 4 \;\; 7 \;\; 9] \\ col = [0 \;\; 1 \;\; 1 \;\; 2 \;\; 0 \;\; 3 \;\; 4 \;\; 2 \;\; 4] \end{cases}$$

(d)
$$CSC \begin{cases} data = [1 \;\; 5 \;\; 4 \;\; 2 \;\; 3 \;\; 9 \;\; 7 \;\; 8 \;\; 6] \\ row = [0 \;\; 2 \;\; 0 \;\; 1 \;\; 1 \;\; 3 \;\; 2 \;\; 2 \;\; 3] \\ ptr = [0 \;\; 2 \;\; 4 \;\; 6 \;\; 7 \;\; 9] \end{cases}$$

Figure 1. The sparse matrix representation for (a) an example matrix *A* in the (b) COO, the (c) CSR, and the (d) CSC formats.

2. SPARSE MATRIX-VECTOR MULTIPLICATION

SpMxV is a mathematical kernel that takes the form of:

$$y \leftarrow Ax, \tag{1}$$

where A is an M×N sparse matrix (the majority of the elements are zero), y is an M×1 vector, and x is an N×1 vector. More generally, SpMxV can be represented as:

$$y \leftarrow \alpha Ax + \beta, \tag{2}$$

where α and β are scalars.

The performance of sparse-matrix algorithms tends to be much lower than that of dense matrices due to two key factors: (1) the way the sparse matrix is represented in memory and (2) the computation architecture of the target platform.

2.1 Sparse Matrix Representation

There are a variety of ways to represent the sparse matrix for storage purposes. However, the few computationally efficient formats are restricted to highly structured matrices, such as diagonal or banded matrices. In this paper, we focus on boosting the efficiency of SpMxV for generic sparse matrices. Therefore, we only present general sparse storage schemes in this section.

Figure 1 illustrates a sample sparse matrix and three different schemes to represent it. The simplest storage scheme, shown in Fig. 1(b), is the coordinate (COO) format. The row indices, column indices, and values of the nonzero matrix entries are explicitly stored in 3 separate arrays: *row*, *col*, and *data*. The compressed sparse row (CSR) format (Fig. 1(c)) is the most commonly used sparse storage scheme, which also stores the column indices and nonzero values into the arrays: *col* and *data*.

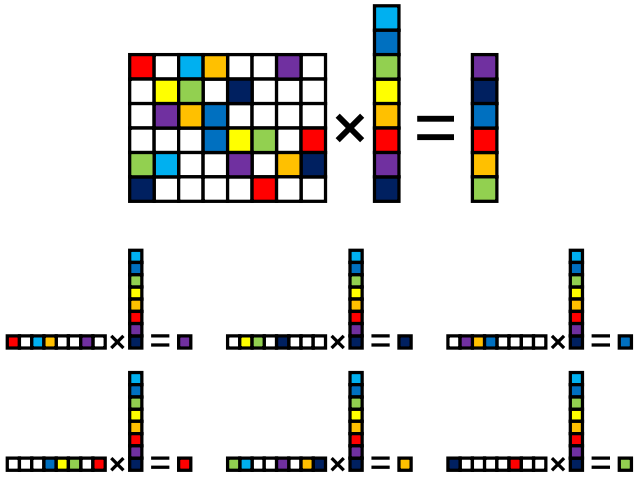

Figure 2. A graphical representation of how SpMxV is performed using the CSR format on CPUs and GPUs. Each element in *y* is calculated as the dot product between the appropriate row of *A* and the vector *x*.

Unlike the COO format, the row indices are not explicitly stored, but rather as an array of row pointers, *ptr*. The i^{th} element of *ptr* corresponds to the offset of the i^{th} row into the *col* and *data* arrays. For example, in Fig.1(c) the first element of *ptr* is 0, indicating that the first element in row 0 is 1 and is located in column 0; the second element of *ptr* is 2, indicating that the first element in row 1 is 2 and is located in column 1; the third element of *ptr* is 4, indicating that the first element in row 2 is 5 and is located in column 5; and so on. For an M×N matrix, *ptr* has M+1 elements in the CSR format, with the final element indicating the total number of nonzero entries in the matrix. The compressed sparse column (CSC) format, used in our SpMxV kernel, is a variation of the CSR format (Fig. 1(d)). Instead of storing the column indices and an array of row pointers, the CSC stores the row indices and an array of column pointers. For any matrix A, the CSR storage of A is exactly the same as the CSC storage of A^T.

2.2 Existing SpMxV Architectures

Specialized software libraries for solving dense and sparse linear algebra problems are very popular for high-performance computing. These libraries, such as MKL [5] for CPUs, and cuBLAS [7] and cuSPARSE [8] for GPUs, provide a standardized programming interface, with subroutines optimized for the target platform.

For SpMxV on CPUs and GPUs, the i^{th} element of y is typically calculated as the dot-product of the i^{th} row of A and the vector x (Fig. 2). This is because each computing core usually contains only a handful of general purpose registers and a single floating-point unit (FPU). Therefore, CSR is one of the most computationally efficient storage options for sparse matrices on CPUs and GPUs. It has the added benefit of being easily parallelizable: each computing core can be independently assigned a different value of y to calculate.

Improving the parallel performance of SpMxV via blocking (splitting up the matrix into several sub-matrices) and modifications to the CRS format is a very active area of study

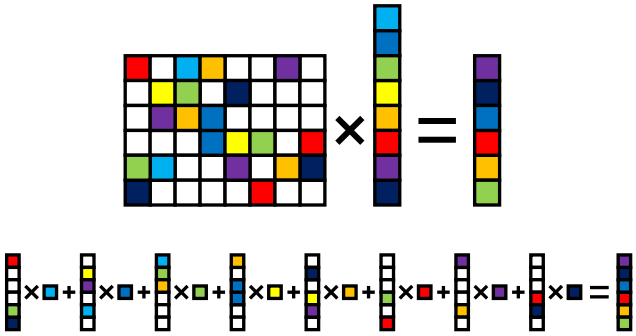

Figure 3. A graphical representation of how SpMxV is performed using the CSC format. The entire vector *y* is calculated as a series of vector additions of the columns of *A* weighted by the appropriate element from *x*.

[3,6,9]. Unfortunately, the use of CSR, and its variants, for SpMxV have several drawbacks on CPUs and GPUs that hurts its overall computational efficiency [3]:

(1) The SpMxV kernel is memory-bounded. CPUs and GPUs typically have much larger computational throughput than available memory bandwidth. This leads to a very low utilization rate for the computing resources, and subsequently, poor energy efficiency.

(2) The indirect (global) memory references for the vector *x* present in *col* adds uncertainty to the memory access pattern, ultimately delaying the computation. Each element of *col* must first be loaded from memory and added to the address of *x* as an offset. Only then can the correct value of *x* be loaded into the FPU for computation.

(3) Irregular memory access of vector *x* causes a large number of cache misses. In CPUs, this cache miss can add tens of cycles of latency. In GPUs, a cache miss can add hundreds of cycles of latency. GPUs typically try to hide these large latencies by interleaving dozens of threads on a single computational core. This works well for computation-bounded algorithms, but not memory-bounded algorithms like SpMxV.

(4) Short row lengths (i.e. very few nonzero elements per row) can cause serious performance degradation. When rows are short, the overhead associated with calculating each element of *y* becomes significant.

Due to these drawbacks, CPUs and GPUs reach less than 5% of their theoretical peak processing throughput and utilize less than 50% of their available memory bandwidth for SpMxV [6,9].

Previous FPGA implementations have attempted to alleviate these inefficiencies by introducing several architectural changes. In some designs, several processing elements (PEs) work together to compute a single element of *y* in parallel [1,4,12]. These designs employ various reduction circuits in order to combine the intermediary results. Other designs have each PE calculate several elements of *y* in a sequential manner in order to mitigate the effect of short rows [2,10,12]. In both cases, the entirety of the *x* vector, or a large subsection (in the case of blocking), is buffered in on-chip Block RAM (BRAM) to reduce the effects of irregular memory accesses [1-4,10-13]. However, these prior implementations primarily focus on reducing the total number of

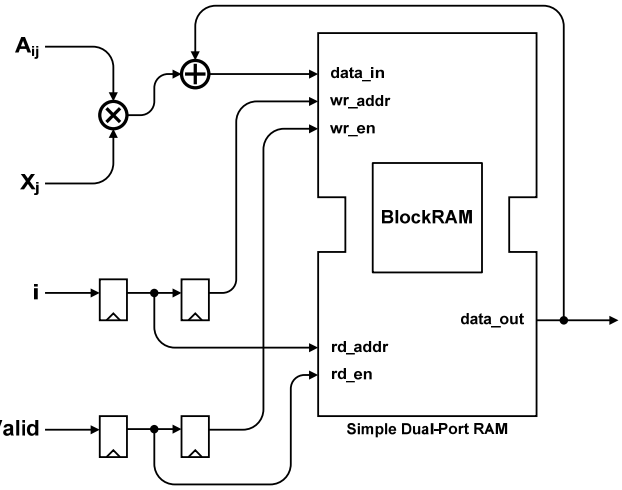

Figure 4. Schematic of a single PE using a simple dual-port RAM, floating-point adder, and a floating-point multiplier.

adders and their resource usage in the design. As such, they only average less than 50% of their theoretical peak processing throughput and memory bandwidth [1-4,10-14].

3. PROPOSED ARCHITECTURE

Our architecture abandons the idea of calculating each element of *y* separately as the row-wise dot product between *A* and *x*. Instead, the entirety of *y* is calculated as the column-wise vector additions of *A* weighted by each element of *x*, as shown in Fig. 3. Fundamentally, this allows us to directly address the major limitations present in the SpMxV algorithm when implemented on an FPGA:

(1) A dedicated co-processor allows for much better balancing of system resources. The number of processing elements (PEs) can be efficiently scaled to match the available memory bandwidth.

(2) What used to be indirect (global) memory references for *x* in *col* vector (for the CSR format) are now direct (local) memory references for *y* in the *row* vector. In other words, when a column of *A* is multiplied by an element of *x*, in the manner shown in Fig. 3, we know exactly which elements of *y* the partial product contributes to. This allows us to halve the number of require memory accesses, the largest bottleneck in the SpMxV algorithm.

(3) Memory access to the *x* vector is no longer irregular, but sequential. By using the CSC format to store *A*, both *A* and *x* can be placed in a large off-chip memory and sequentially streamed into the DSP co-processor (eliminating the time and energy overheads of a cache miss).

(4) Short row or column lengths have much less impact on the performance of SpMxV, since the PEs are rarely idled thanks to the balanced memory bandwidth and computing capability. However, performance is degraded as the memory bandwidth of *x* approaches that of *A* for extremely sparse matrices. In the rare case of $M \gg N$, the performance of CSC is no better than that of CSR.

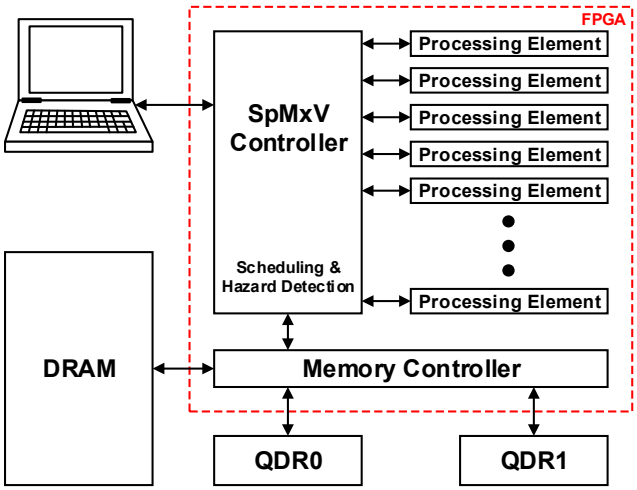

Figure 5. Top-level schematic of the SpMxV kernel, with 8 processing elements, running on the ROACH FPGA platform. The SpMxV kernel acts as a coprocessor for a networked computer running a MATLAB environment.

Computing the SpMxV column-wise also allows for an extremely simple PE (Fig. 4). Each PE contains a single-precision floating-point adder and multiplier, as well as a simple dual-port RAM. Each simple dual-port RAM utilizes the large amounts of BRAM resources available on FPGAs and can accommodate several hundred to several thousand elements of y. Figure 5 shows the overall experiment setup, in which the SpMxV kernel (implemented on an FPGA) serves as a co-processor attached to an external computer. A dedicated memory controller allows the elements of A and x to be continuously streamed into the FPGA, while the SpMxV controller's primary function is scheduling and hazard detection. Hazard detection avoids the conflict between two partial products that contribute to the same element of y overlapping due to the latency of the floating-point adder. If such a hazard is detected, we must either stall or provide alternative data to ensure that the result of y is calculated correctly.

3.1 Processing Element

As stated previously, each PE (Fig. 4) contains a single-precision floating-point adder and multiplier, as well as a simple dual-port RAM. To perform the SpMxV, each PE multiplies an element of the *data* vector (A_{ij}) and the corresponding element of the x vector (X_j) together. The resulting partial product is then added to address in the *row* vector (i), before being stored back into the BRAM of the dual-port RAM. Due to the latency of the multiplication and addition operations, a *Valid* signal is used to prevent data corruption due to hazards. Using this strategy, data can be continuously streamed into each PE (directly from the CSC format) with a small startup overhead latency equal to that of the adder and the multiplier.

Since each PE has its own working copy of the vector being computed, there are two possible strategies for assembling the final vector. The first option is to assign a subset of the x vector to each PE (i.e. blocking along the columns of A). Each PE computes a partial sum of the final vector and an adder tree (which can be built from the existing adders in each PE) is used to combine them at the end. Similar to prior FPGA implementations [1-4,10-13], this straightforward approach has several drawbacks.

Table 1. Resource usage for the SpMxV kernel (64 PEs).

Resource	Used	Available	Percent
Registers	31,621	58,880	53.70%
LUTs	27,958	58,880	47.48%
BRAMs	160	244	65.57%
DSP48Es	320	640	50.00%

First, the reduction circuit adds a large amount of overhead in terms of latency and additional hardware (even if existing adders are used, more resources are needed for configurability). Second, by splitting up computation along the columns, we lose some of the sequential nature of the memory accesses we had gained with the CSC format. The memory accesses for each PE are still sequential, but globally the memory accesses for all PEs are irregular. To mitigate this, a more complicated memory controller is required to ensure a balanced load across all of the PEs. Third, if there are fewer columns than PEs, this approach is effectively no different than prior FPGA implementations.

The second option for computing the final vector is to assign a subset of rows of A to each PE (i.e. blocking along the rows of A). Each PE computes a subset of the final vector, which are then concatenated together at the end (requiring no additional latency). Additionally, this preserves the property of sequential memory accesses across all PEs, allowing for a much simpler memory controller (at the cost of a slightly more complicated SpMxV controller to handle additional scheduling and hazard detection). Finally, with minor modifications to the SpMxV and memory controllers, our SpMxV kernel can also support a sparse matrix dense matrix multiplication (SpMxM): each column of the dense matrix is assigned to a PE, which each PE computing a single column of the resulting matrix. Our SpMxV kernel uses this option, with the modifications to support SpMxM, in its implementation.

3.2 SpMxV and Memory Controller

The primary purposes of the SpMxV controller are scheduling and hazard detection. The memory controller acts as a slave to the SpMxV controller, ensuring a continuous stream of data into the PEs. Hazards arise when two or more partial products want to write to the same memory address of the dual-port RAM in a short period of time. Due to the latency of the floating-point adder in Fig. 4, the existing sum of the partial products in the dual-port RAM must be prefetched. If two partial products that contribute to the same term are allowed to proceeded, the second product will prefetch a sum that does not include the first product. The result is that the final sum of products will not include the first conflicting partial product. The SpMxV controller detects these hazards and corrects them in one of two ways. It first attempts to shuffle the partial products to increase their distance in time. If this is not possible, or would result in additional hazards, the SpMxV controller issues a stall command to the PE (by deasserting the *Valid* signal, and holding the values of A_{ij}, X_j, and i).

Figure 6 shows the stalling behavior for a single processing element performing SpMxV between the example A matrix from Fig. 1(a) and the vector $x = [0.1\ 0.2\ 0.3\ 0.4\ 0.5]^T$. In the example, the latency of the floating-point multiplier and adder are both 2

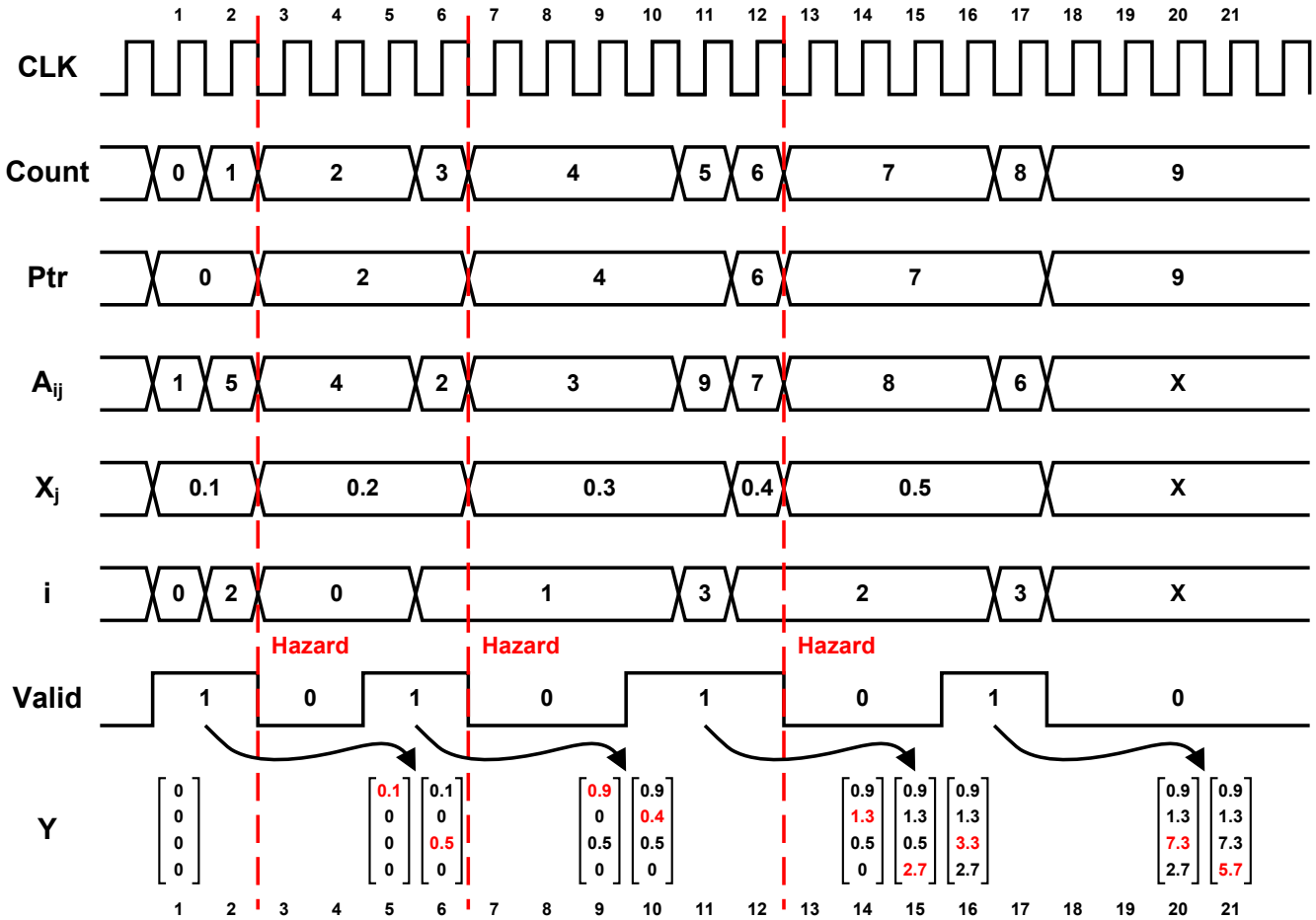

Figure 6. Timing diagram for calculating the SpMxV of the example *A* matrix from Fig. 1(a) and $x = [0.1\ 0.2\ 0.3\ 0.4\ 0.5]^T$ using a single PE. For this example, the floating-point adder and multiplier both have a latency of 2 clock cycles and the simple dual-port memory (*Y*) has a latency of 1 clock cycle.

clock cycles. Additionally, the simple dual-port memory (*Y*) has a latency of 1 clock cycle. In cycle 2, a hazard is detected due to the proximity of the partial product of 1×0.1 and the partial product of 4×0.2. We must stall for 2 cycles—by deasserting *Valid* and holding the values of A_{ij}, X_j, and *i*—to ensure that partial product of 1×0.1 is added to *Y* before continuing. If co-processor had not stalled, the partial product of 4×0.2 would have added itself to the current value of *Y*, 0, producing an incorrect final value of 0.8 (instead of 0.9). Computation resumes in cycle 5, after the hazard has passed. Additional hazards are detected in cycles 6 and 12, with each hazard resulting in 3 clock cycles of stalling.

4. PERFORMANCE EVALUATION

The SpMxV kernel was evaluated on the open-source academic research platform "ROACH" (Reconfigurable Open Architecture Computing Hardware) [15]. The ROACH platform is equipped with a Virtex-5 SX95T FPGA for DSP applications, a PowerPC running Linux, two 36Mb QDRII+ SRAMs, and 2GB of DDR2 SDRAM for a combined peak memory bandwidth of 35.74 GB/s. The PowerPC allows a computer running MATLAB to interface with "software registers," BRAMs, and FIFOs on the FPGA, as well as to load data in and out of the board-level QDRs and DRAM (Fig. 5).

Table 1 show the total resource usage of the SpMxV kernel implemented with 64 PEs using a single-precision floating-point format. The SX95T FPGA can support up to 96 single-precision PE (using 98.4% of the available BRAM), but is ultimately limited by our available memory bandwidth. The CSC *A* matrix is stored in the SDRAM, while the *x* and *y* vectors are stored in the two QDRs. The QDRs are accessed in parallel to effectively create a single memory with twice the bit width. The ROACH board can operate up to 150MHz (limited by the QDR memory controller), resulting in a peak performance of 19.2 GFLOP/s with a thermal design power (TDP) of 25W.

4.1 Comparison to CPUs and GPUs

We use a collection of 10 unstructured matrices used by both Williams et al. [6] and Bell et al. [9] in our performance benchmarking study. Table 2 details the size and overall sparsity structure of each matrix. All of the matrices are publically available online from the University of Florida Sparse Matrix Collection [16]. For comparison, the same benchmarks are run on both a 64-bit Linux machine and a 64-bit Windows machine.

The Linux machine has 16GB of memory and an Intel Core i7-2600 processor (4 physical cores with hyper-threading, for a total of 8 virtual cores), using the MKL sparse-BLAS library [5]. The Core i7-2600 processor has a peak memory bandwidth of 21GB/s

Table 2. Summary of unstructured matrices used for benchmarking performance (publically available from [16]).

Matrix	Rows	Columns	Nonzeros	Nonzeros/Column	Density
Dense	2,000	2,000	4,000,000	2000.00	100.00000%
Protein	36,417	36,417	4,344,765	119.31	0.32761%
FEM/Spheres	83,334	83,334	6,010,480	72.13	0.08655%
FEM/Cantilever	62,451	62,451	4,007,383	64.17	0.10275%
Wind Tunnel	217,918	217,918	11,524,432	52.88	0.02427%
FEM/Harbor	46,835	46,835	2,374,001	50.69	0.10823%
QCD	49,152	49,152	1,916,928	39.00	0.07935%
FEM/Ship	140,874	140,874	3,568,176	25.33	0.01798%
Economics	206,500	206,500	1,273,389	6.17	0.00299%
FEM/Accelerator	121,192	121,192	2,624,331	21.65	0.01787%

with a peak performance of 108.8GFLOP/s and a TDP of 95W [17]. The benchmarks were also run on an NVIDIA GeForce GTX 660 graphics card (960 CUDA cores), installed on the same Linux machine, using the cuSPARSE library [8]. The GPU has a peak memory bandwidth of 144.2GB/s and a peak performance of 1881.6GFLOP/s with a TDP of 140W [18].

The Windows machine has 32GB of memory and an Intel Core i7-4770 processor (4 physical cores with hyper-threading, for a total of 8 virtual cores), using the MKL sparse-BLAS library [5]. The Core i7-4770 processor has a peak memory bandwidth of 25.6GB/s and a peak performance of 217.6GFLOP/s with a TDP of 84W [19]. The benchmarks were also run on an NVIDIA GeForce GTX Titan graphics card (2688 CUDA cores), installed on the same Windows machine, using the cuSPARSE library [8]. The GPU has a peak memory bandwidth of 288.4GB/s and a peak performance of 4,500GFLOP/s with a TDP of 250W [20].

Figure 7 compares the raw computational performance (in GFLOP/s) of the CPU, GPU, and FPGA SpMxV kernels for all of the matrices tested. SpMxV on the two CPUs showed a performance drop of 20-50% compared to dense matrices, while the two GPUs showed a performance drop of 30-60%. Figure 8 compares the computational efficiency of the CPU, GPU, and FPGA SpMxV kernels for all of the matrices tested. For a memory bound algorithm like SpMxV, the computational efficiency is strongly determined by the memory hierarchy (i.e. the cache structure and size). The computational efficiency is calculated as the ratio of the measured SpMxV performance, in GFLOP/s, over the theoretical peak GFLOP/s achievable for each platform.

The Core i7-2600 and Core i7-4770 processors achieved an average performance of 2.01 and 4.59GFLOP/s, respectively, across all 10 test matrices. The resulting computational efficiencies were 1.84% and 2.11%. Overall, the computational efficiency of both CPUs was 1-2% for all of the test matrices. The Core i7-4770 processor was able to achieve a 2.28x speedup over the Core i7-2600, despite only having 22% more memory bandwidth, due to its more efficient memory accesses with its larger vector processing cores.

The largest drops in performance for the CPUs were recorded using the sparsest matrices: FEM/Ships, Economics, and FEM/Accelerator. These matrices had a significantly higher rate of cache misses due to their large size and overall sparsity. The relativity small number of nonzero elements per row (especially for the Economics matrix) also added significant overhead by having to flush the pipeline more often.

Similarly, the GTX 660 and GTX Titan GPUs achieved an average performance of 5.79 and 14.86 GFLOP/s, respectively across all 10 matrices. The resulting computational efficiencies were 0.31% and 0.33%. Overall, the computational efficiency of both GPUs was between 0.2-0.5% for all of the test matrices. The GTX Titan had an average speed up of 2.57x over the GTX 660 GPU, which is consistent with the GTX Titan having 2x the memory bandwidth and 2.39x the number of processing cores as the GTX 660. Because individual process threads are organized differently on the GPU, each CUDA core had to be flushed far less often than the CPU for the FEM/Ships, Economics, and FEM/Accelerator matrices, leading to more even performance.

On the Linux test setup, the GTX 660 had an average speedup of 2.93x over the i7-2600 processor, and on the Windows setup, the GTX Titan had a 3.23x speedup over the i7-4770 processor. Despite the roughly 3x performance increase by the GPUs (Fig. 6), the CPUs are about 6x more computationally efficient than the GPUs (Fig. 7). The GPUs use 100-300x more processing cores to achieve a total, theoretical peak performance roughly 20x greater than that of the CPU, but only have about 8x more memory bandwidth. The cache structure of a GPU is smaller and has higher latency that of a CPU [18,19]. GPUs are designed to mask random memory accesses for computationally intensive algorithms, leading to much larger penalties in efficiency for cache misses when compared to a CPU.

The SpMxV kernel running on the Virtex-5 SX95T FPGA achieved an average performance of 17.64GFLOP/s, for a computational efficiency of 91.85%, across all 10 matrices. The Dense matrix achieved a peak performance of 19.16GFLOP/s for a computational efficiency of 99.8%. This performance represents an average speedup of 9.55x and 4.18x over the i7-2600 and i7-4770 CPUs and a 3.31x and 1.28x speedup over the GTX 660 and GTX Titan GPUs. Moreover, the computational efficiency of the FPGA SpMxV kernel had an average improvement of 54x and 322x over the CPUs and GPUs, respectively.

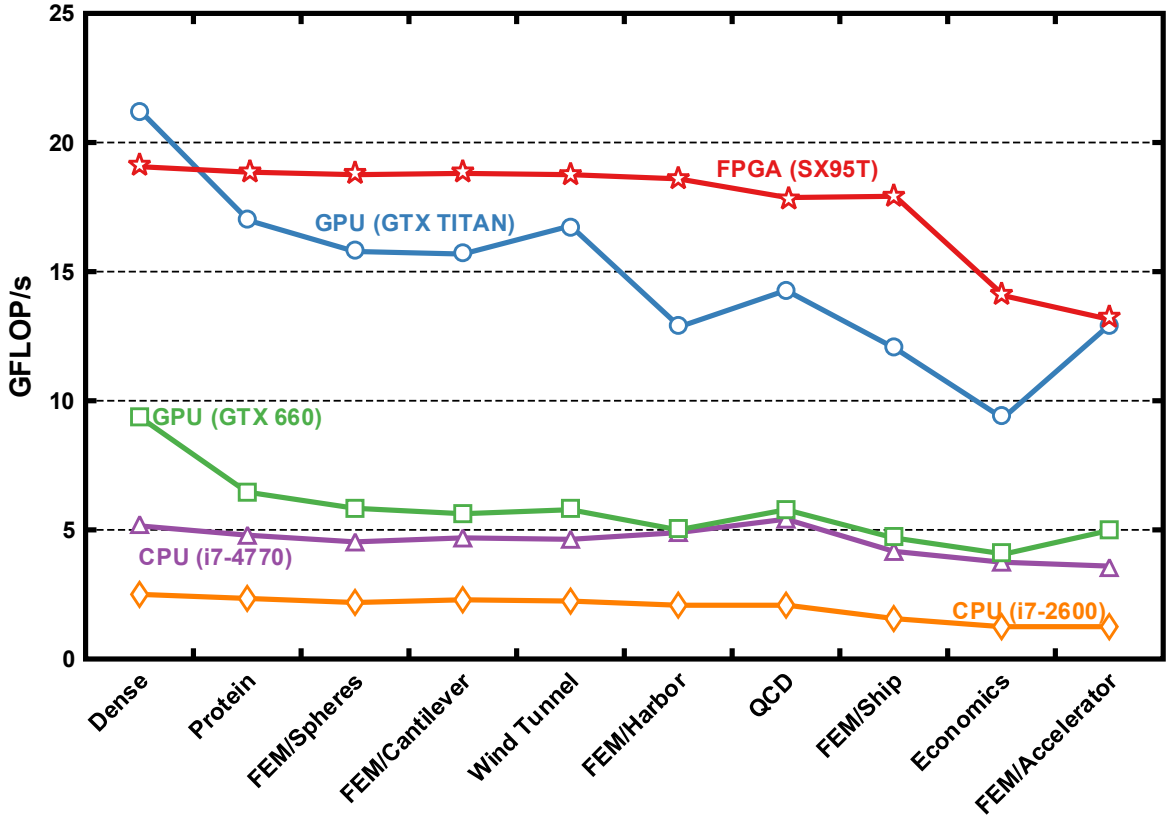

Figure 7. Raw computational performance of the CPU, GPU, and FPGA SpMxV kernels.

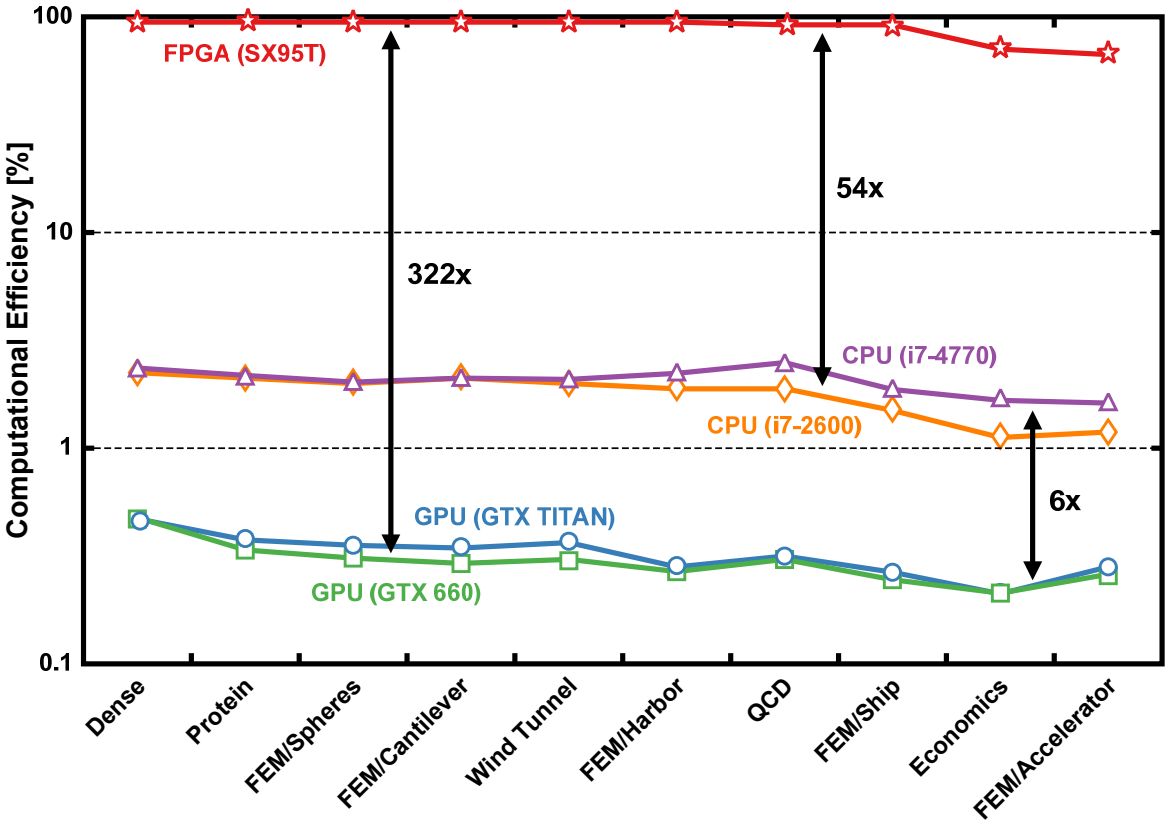

Figure 8. Computational efficiency of the CPU, GPU, and FPGA SpMxV kernels.

Table 3. Comparison of FPGA SpMxV Architectures

	[1]	[2]	[11]	[12]	[13]	This Work
FPGA	Virtex-5 LX155T	Stratix-III EP3SE260	Virtex-II Pro 70	Virtex-II Pro 100	Virtex-II 6000	Virtex-5 SX95T
Frequency [MHz]	100	100	200	170	95	150
Memory Bandwidth [GB/s]	6.5	8.5	8	8.5	1.6	35.74
Number of PE	16	6	4	5	3	64
Peak Performance [GFLOP/s]	3.2	1.2	1.6	1.7	0.57	19.2
Matrix Format	CVBV†	COO	CSR	CSR	SPAR*	CSC
Sparse Test Matrix Density						
MIN-MAX [%]	0.01-5.48	0.51-11.49	0.04-4.17	0.04-0.39	0.01-1.10	0.003-0.33
Average [%]	1.41	3.34	0.87	0.16	--	0.09
Computational Efficiency						
MIN-MAX [%]	1-7	5-7	20-79	50-98.4	1-74	69-99.8
Average [%]	4.48	5.63	42.6	79.4	55.6	91.9

†Variant of CSR. *Variant of CSC.

The average power consumption of the i7-2600 and i7-4770 processors was measured to be 77.2W and 66.3W, respectively. The resulting power efficiencies are 26MFLOP/s/W and 69MFLOP/s/W. Similarly, the average power of the GTX 660 and GTX Titan were measured to be 99W and 163W, respectively. The resulting power efficiencies are 58MFLOP/s/W and 91MFLOP/s/W. The worst case power of the SX95T FPGA was measured to be 5.1W, resulting in a power efficiency of 3,460 MFLOP/s/W. This represents more than a 50x and 38x improvement in energy efficiency over the CPU and GPU implementations, respectively.

4.2 Comparison to Existing FPGA Art

In Table 3, we compare our proposed architecture to several published SpMxV FPGA architectures. The architectures can be categorized into 3 distinct groups. The first is to either re-encode, reorder, or preprocess the sparse matrix in such a way as to reduce data hazards [1,2]. Kestur et al. [1] re-encode the matrix into the Compressed Variable-Length Bit Vector (CVBV) format (a variation of the CSR format) on the fly to reduce the required memory bandwidth. However, re-encoding the matrix insures a significant overhead penalty, resulting in marginal efficiency improvements over CPU and GPU implementations (only 3-10x). Sun et al. [2] preprocess the matrix to reorganize and optimize the datapath to eliminate hazards. After preprocessing, the matrices achieve a computational efficiency of 96-99% on the FPGA. Unfortunately, the preprocessing overhead (datapath optimization, FPGA reconfiguration, and buffering the matrix on BRAM) is about 20 times greater than that of the SpMxV calculation. This results in an effective computational efficiency of only 5-7%.

The second approach is to use several PE (each with its own working copy of x) in parallel with a reduction circuit or adder tree to accumulate a single element of y [11,12]. In Zhuo et al. [11], the partial product of 4 multipliers are added together via a tree of 3 adders. The resulting sum is then fed into a novel reduction circuit (accumulator) that handles the potential read-after-write data hazards. The drawback of this approach is that it requires zero padding to achieve a minimum row size. Combined with blocking along the columns of A, the design is sensitive to the sparsity structure of the matrix. The computational efficiency ranges from 20% to 79%, performing particularly poorly for extremely sparse matrices (<0.1% density). Zhang et al. [12] improves upon this design by having each PE calculate a different element of y. Their accumulator requires a minimum row length of 8, zero padding when necessary, but can switch between any of several different rows if it encounters a data hazard. This roughly doubles the computational efficiency of the design as compared to Zhuo et al. [11]. However, it still performs quite poorly for extremely sparse matrices.

The final approach is to use the method outlined is Section 3 to stream CSC matrix data through several PEs. Gregg et al. [13] use a variant of CSC called the sparse matrix architecture and representation (SPAR) format. In the SPAR format, *row* and *ptr* are combined into a single vector with zero padding introduced at the start of each column of the *data* vector. Each PE only buffers a small portion of the y vector (32 elements vs. 3,456 elements in our design) in a local cache. A y-cache miss requires a write-back to high latency DRAM, incurring a 109 clock cycle penalty. With such a small cache, the design is very highly sensitive to the sparsity structure of the matrix. Computational efficiency ranges from as low as 1% to as high as 74%.

Our architecture improves upon Gregg et al. [13] by buffering much larger sections of the y vector (removing the need for a costly write-back scheme) and the elimination of unnecessary stalling due to zero padding (due to the SPAR format). Our design is able to handle matrix densities below 0.01% (10x sparser than prior FPGA designs) with computational efficiencies as high as 99.8%. Our drop in performance (and efficiency) for the Economics matrix on the ROACH platform is due to the very small number of nonzeroes per column, causing the x vector to need more memory bandwidth than the QDRs can deliver. This results in about 25% of the PEs being idled during any given cycle. The FEM/Accelerator matrix, while similar in density to the FEM/Ship matrix, alternates between being extremely dense and extremely sparse along its columns, causing about 30% of the PEs to be idled during any given clock cycle. This particular issue can be alleviated by buffering the x vector on-chip like previous

FPGA designs. We estimate that this would increase the sustained (average) computational efficiency from 91.85% to 98.29%.

5. CONCLUSIONS

This paper describes a SpMxV kernel using a CSC sparse-matrix format, and demonstrates its computational efficiency using an FPGA. The efficiency advantage of the kernel results from transforming irregular random memory accesses into a regularized stream of serial memory accesses. The benchmarking results show that the proposed architecture achieves a peak computational efficiency of 99.8% when performing SpMxV, which is over 54 and 322 times more efficient than an Intel Core i7-4770 processor and over an NVIDA GTX Titan GPU performing the same tasks, respectively. Implemented on a Virtex-5 SX95T FPGA, our design is able to achieve higher performance than its CPU and GPU counterparts running optimized sparse-BLAS software libraries, while only using 64 single-precision processing elements, with a 38-50x improvement in energy efficiency.

6. ACKNOWLEDGMENTS

The authors would like to thank Yuta Toriyama and Fang-Li Yuan of UCLA for their helpful discussions.

7. REFERENCES

[1] S. Kestur, J.D. Davis, and E.S. Chung, "Towards a Universal FPGA Matrix-Vector Multiplication Architecture," *Int. Symp. Field-Programmable Custom Comp. Mach. (FCCM 2012)*, pp. 9–16, May 2012.

[2] S. Sun, M. Monga, P.H. Jones, and J. Zambreno, "An I/O Bandwidth-Sensitive Sparse Matrix-Vector Multiplication Engine on FPGAs," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 59, no. 1, pp. 113–123, Jan. 2012.

[3] G. Goumas, K. Kourtis, N. Anastopoulos, V. Karakasis, and N. Koziris, "Understanding the Performance of Sparse Matrix-Vector Multiplication," *Euromicro Conf. Parallel, Distributed and Network-Based Process. (PDP 2008)*, pp. 283–292, Feb. 2008.

[4] J. Sun, G. Peterson, and O. Storaasli, "Mapping Sparse Matrix-Vector Multiplication on FPGAs," *Reconfigurable Systems Summer Institute (RSSI 2007)*, July 2007.

[5] "Intel Math Kernel library." [Online]. Available: http://software.intel.com/en-us/intel-mkl

[6] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demme, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," in *Proc. ACM/IEEE Conf. Supercomputing (SC 2007)*, pp.1–12, Nov. 2007.

[7] "Nvidia cuBLAS." [Online]. Available: http://developer.nvidia.com/cublas

[8] "Nvidia cuSPARSE." [Online]. Available: http://developer.nvidia.com/cusparse

[9] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," in *Proc. ACM/IEEE Conf. Supercomputing (SC 2009)*, pp. 18:1–18:11, Nov. 2009.

[10] G. Kuzmanov and M. Taouil, "Reconfigurable sparse/dense matrix-vector multiplier," *Int. Conf. Field-Programmable Tech. (FPT 2009)*, pp. 483–488, Dec. 2009.

[11] L. Zhuo and V.K. Prasanna, "Sparse Matrix-Vector multiplication on FPGAs," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays (FPGA '05)*, pp. 63-74, Feb. 2005.

[12] Yan Zhang, Y.H. Shalabi, R. Jain, K.K. Nagar, and J.D. Bakos, "FPGA vs. GPU for sparse matrix vector multiply," *Int. Conf. Field-Programmable Tech. (FPT 2009)*, pp. 255–262, Dec. 2009.

[13] D. Gregg, C. McSweeney, C. McElroy, F. Connor, S. McGettrick, D. Moloney, and D. Geraghty, "FPGA Based Sparse Matrix Vector Multiplication using Commodity DRAM Memory," *Int. Conf. Field Programmable Logic Applicat. (FPL 2007)*, pp. 786-791, Aug. 2007.

[14] C.Y. Lin, H. K.-H. So, and P.H.-W. Leong, "A Model for Matrix Multiplication Performance on FPGAs," *Int. Conf. Field Programmable Logic Applicat. (FPL 2011)*, pp.305–310, Sept. 2011.

[15] "ROACH." [Online]. Available: https://casper.berkeley.edu/wiki/ROACH

[16] T. A. Davis and Y. Hu., "The university of Florida sparse matrix collection.," *ACM Trans. Math. Softw.*, vol. 38, no. 1, pp. 1:1–1:25, Dec. 2011.

[17] P. Gepner, D. L. Fraser, and V. Gamayunov, "Evaluation of the 3rd generation Intel Core Processor focusing on HPC applications," *Int. Conf. Parallel Distrib. Process. Techn. Applicat. (PDPTA 2012)*, pp. 818–823, July 2009.

[18] "NVIDIA GeForce GTX 680: The fastest, most efficient GPU ever built." [Online]. Available: http://www.geforce.com/Active/en_US/en_US/pdf/GeForce-GTX-680-Whitepaper-FINAL.pdf

[19] "Intel® Core™ i7-4770 Processor." [Online]. Available: http://ark.intel.com/products/75122/Intel-Core-i7-4770-Processor-8M-Cache-up-to-3_90-GHz

[20] "Introducing the GeForce GTX TITAN." [Online]. Available: http://www.geforce.com/whats-new/articles/introducing-the-geforce-gtx-titan