An I/O Bandwidth-Sensitive Sparse Matrix-Vector Multiplication Engine on FPGAs

Song Sun, Madhu Monga, Phillip H. Jones, *Member, IEEE*, and Joseph Zambreno, *Member, IEEE*

***Abstract*—Sparse matrix-vector multiplication (SMVM) is a fundamental core of many high-performance computing applications, including information retrieval, medical imaging, and economic modeling. While the use of reconfigurable computing technology in a high-performance computing environment has shown recent promise in accelerating a wide variety of scientific applications, existing SMVM architectures on FPGA hardware have been limited in that they require either numerous pipeline stalls during computation (due to zero padding) or excessive input preprocessing during run-time. For large-scale sparse matrix scenarios, both of these shortcomings can result in unacceptable performance overheads, limiting the overall value of using FPGAs in a high-performance computing environment. In this paper, we present a scalable and efficient FPGA-based SMVM architecture which can handle arbitrary matrix sparsity patterns without excessive preprocessing or zero padding and can be dynamically expanded based on the available I/O bandwidth. Our experimental results using a commercial FPGA-based acceleration system demonstrate that our reconfigurable SMVM engine is highly efficient, with benchmark-dependent speedups over an optimized software implementation that range from $3.5\times$ to $6.5\times$ in terms of computation time.**

***Index Terms*—FPGA, reconfigurable computing, sparse matrix-vector multiplication.**

I. Introduction

FLOATING-POINT sparse matrix-vector multiplication (SMVM) plays a paramount role in many scientific and engineering applications, including image construction, economic modeling, industrial engineering, control system simulation and information retrieval [1], [2]. In solving large linear systems $y = Ax$ (where A is an $n \times n$ sparse matrix with n_z nonzero entries, and x and y are $1 \times n$ matrices) and eigenvalue (λ) problems $Ax = \lambda x$ using iterative methods, SMVM can be performed hundreds or even thousands of times on the same matrix. For example, Google's PageRank eigenvector problem is dominated by SMVM, where the size of the matrix (n) is of the order of billions [3]. A naive sequential solution of the PageRank problem using the power method would take days to converge. As the size of the datasets used in scientific and engineering applications (including Google PageRank) continue to grow rapidly, the runtime of SMVM is likely to continue to dominate these applications.

SMVM can be characterized as containing large amounts of floating-point computation, coupled with irregular memory access patterns [4]. Memory bandwidth has been a significant performance bottleneck in traditional microprocessor architecture. While multilevel cache hierarchies are successfully used to improve the memory throughput of data-intensive applications, the irregularity of SMVM memory access has led to limited performance scalability of these applications on traditional microprocessors [5], [6]. More recently, Graphics Processing Units (GPUs), in addition to being used for accelerating graphics rendering applications have been exploited for accelerating general-purpose computations [7]. These modern GPUs have a hybrid caching/memory hierarchy with various latencies and optimal access patterns. This architectural complexity (coupled with a highly parallel execution model) has made it a challenge to optimize an irregular data-intensive application such as SMVM for GPUs [4].

Several companies have leveraged the highly parallel and specialized computational fabric of modern FPGAs to develop FPGA-based high-performance computing systems (e.g., SRC [8], Cray [9], XtremeData [10], and Convey [11]). While large-capacity FPGAs have been successfully used in these platforms to accelerate important computational kernels, it is clear that improving on I/O throughput is necessary for accelerating FPGA-based data-intensive applications, including SMVM [12]. The main contribution of this paper is a new FPGA-based architecture for SMVM, which attempts to minimize memory access overhead without performing excessive preprocessing of the input matrix, as required in previous approaches. Our design efficiently keeps track of interrow computation in the SMVM pipeline, eliminating the need for zero padding (stalls), while adding little hardware overhead as compared to previous approaches.

The remainder of this paper is organized as follows. Section II describes related work in the general field of SMVM optimization, with examples of both hardware (e.g., FPGA) and software (e.g., CPU, GPU) implementations. The details of our architecture are presented in Section III, including the individual computational building blocks, how zero padding is eliminated, and an algorithm that generates the mapping table for an arbitrary I/O bandwidth budget. Performance evaluation and experimental results are presented next, with circuit implementation details followed by HW/SW platform integration in Section IV. Finally, the paper is concluded in Section V with a preview of future planned work in this area.

Manuscript received November 10, 2010; revised March 16, 2011; accepted June 06, 2011. Date of publication August 12, 2011; date of current version January 11, 2012. This paper was recommended by Associate Editor M.-D. Shieh.

The authors are with the Department of Electrical and Computer Engineering, Iowa State University, Ames, IA 50010 USA (e-mail: sunsong@iastate.edu; madhum@iastate.edu; phjones@iastate.edu; zambreno@iastate.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCSI.2011.2161389

II. Related Work

Due to its importance in scientific and engineering applications, a considerable amount of effort has been devoted to maximizing the performance of SMVM [13], [14]. The authors in [14] describe techniques that increase instruction-level parallelism on superscalar RISC processors. In [13], the balancing of distributed storage of nonzero elements among parallel processor arrays is investigated. More recently, several new optimizations for multicore platforms have been proposed in the research literature. In [1], the authors explore the potential advantages of explicit multicore programming. The emergence of GPU as a powerful multicore architecture in recent years has made it an attractive target for SMVM optimization. Several different approaches of SMVM on GPU have been presented, including [4], [15]–[18]. Some online toolkits and libraries for GPUs are also available [19], [20]. It is important to note that these GPU SMVM implementations achieve consistent speedups only for matrices with regular sparsity patterns.

FPGAs provide a relatively low-cost platform for parallelizing algorithms at the operand-level granularity. Consequently, a diverse selection of FPGA-based accelerators can be found in the research literature [21]–[26]. FPGAs have great potential in coping with the irregularity of SMVM due to their easily pipelined ability, inherent parallelism and configurable architecture. The design of SMVM in [2] employs double-precision floating-point multipliers and adders, and performs multiple floating-point and I/O operations in parallel. The results show that their design achieves over 350MFLOPS for all test matrices when the memory bandwidth is 8 GB/s. However, the performance of their design is greatly affected by the padding overhead. The smaller the overhead, the higher the performance. The implementation in [27] partitions the set of n dot products across multiple Processing Elements (PEs). A matrix mapping algorithm is critical in reducing the computation latency while minimizing the inter-PE communication. Also, since only on-chip BlockRAM (BRAM) is used to store the matrix, the matrix size is constrained by the BRAM. This preprocessing of the input matrix and vector would lead to potentially large overheads for very big matrices. As the performance speedup due to the use of FPGA technology is a function of the percentage of time spent in SMVM in the accelerated application, this preprocessing can quickly amortize the overall performance benefit. The design in [28] has pipeline stalls and depends less on the matrix structure compared with a software approach. An efficient SMVM computation of very large sparse Finite-Element matrices is proposed in [29].

III. Our Approach

A. Design Principle

Although not an optimization technique per se, to save storage space and improve performance, it is extremely common to store and process only the nonzero values in a sparse matrix. Our architecture assumes that the sparse matrix is stored in row-major order and aims at improving the performance of arbitrary sparsity patterns. One straightforward implementation is composed of a multiplier array (k multipliers) and a binary adder tree. In each clock cycle, the multipliers in the array multiply the nonzero elements in the same row in the sparse matrix with the elements in the vector. If the number of nonzeros of each row l_i is equal to k, this architecture is quite efficient. However, if l_i is smaller than k, there will be zero paddings; if l_i is larger than k, there will be pipeline stalls. The goal of our design is to remove these hurdles without overly complicating the architecture.

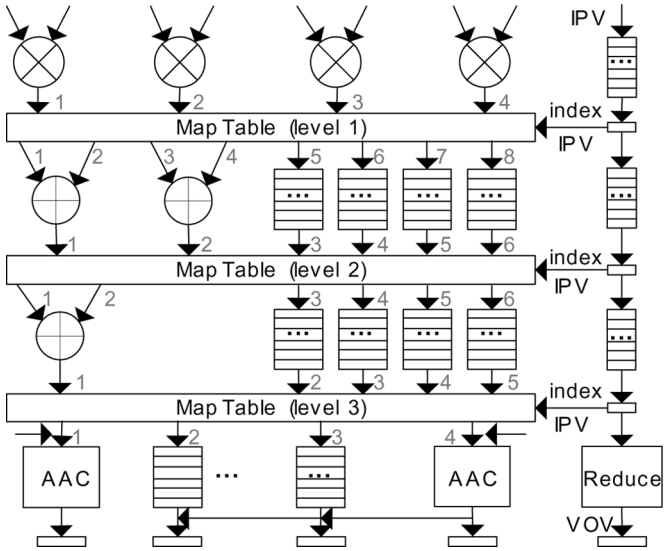


Fig. 1. Proposed SMVM architecture ($k = 4$).

B. Input Patterns

Suppose there are four ($k = 4$) multipliers in the architecture described above. The elements of the matrix are transferred to the system in one clock cycle, disregarding whether they are from the same row or consecutive rows. The nonzeros of a sparse matrix in row-major order can be considered as a stream of numbers which can be divided into a bulk of segments. Each segment has k numbers. The Input Pattern Vector (IPV) for each segment is defined as the following:

Definition 1 (Input Pattern Vector): The ith bit of a k-bit IPV is 1 if the ith number in a segment is the last nonzero of a row; otherwise, the ith bit of a k-bit IPV is 0.

Different IPVs can correspond to different structural configurations of the adder tree. In Fig. 2, the four inputs from the matrix are from four different rows when $\mathrm{IPV} = 1111/1110$. Thus, the four outputs from the multipliers need not be sent to the adder tree. The first input may have to be accumulated with the outputs of the previous clock cycle. The last input must be accumulated with the outputs of the next clock cycle when $\mathrm{IPV} = 1110$. In Fig. 2, the first three inputs must be added together and sent to the adder tree in the case of $\mathrm{IPV} = 0011/0010$ because they are from the same row. To assure the uniqueness of the adder tree structure, we require the addition operation to be left associative. This requirement does not negatively affect the computation accuracy due to the associativity of addition operation. It is easy to see that the number of adder trees with distinct structures is 2^{k-1} if the length of the IPV is k. An example of IPV generation is provided in Appendix A.

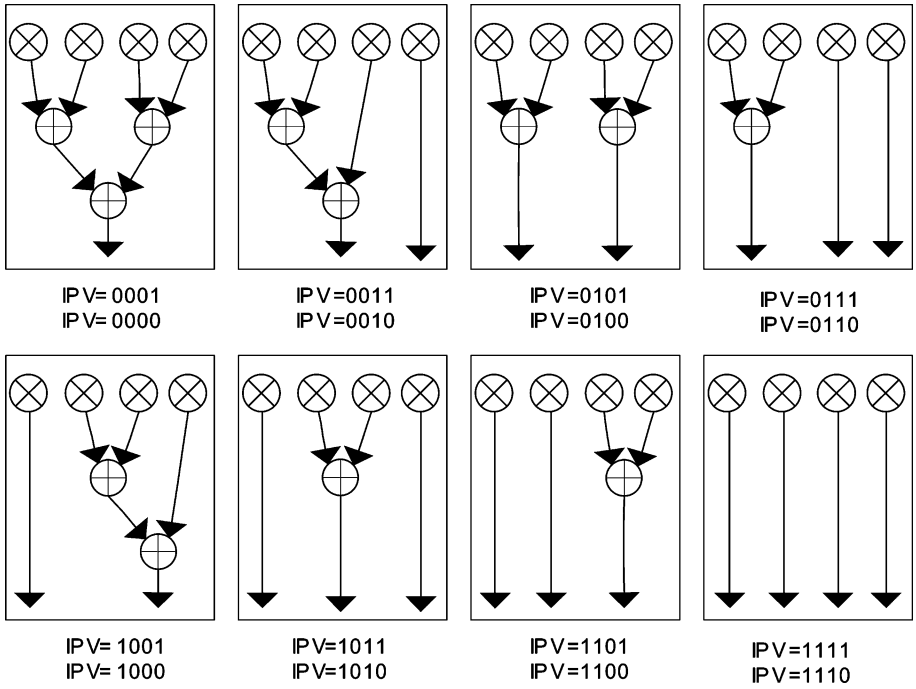


Fig. 2. Input pattern and tree structure ($k = 4$).

C. SMVM Architecture

Our architecture as shown in Fig. 1 is composed of a multiplier array, an adder tree, two adder accumulators (AACs), a map table, and register arrays. All components in the architecture are pipelined and run in parallel. The data flows top-down in a pipelined fashion. In each clock cycle, k numbers from the matrix are multiplied pairwise with k numbers from the vector. The number of multipliers (k) is a configurable parameter which can be any natural number. The connections between the outputs of multipliers and the inputs of adder tree are configured by looking up the map table on the fly. Similarly, the connections among the adder tree and register arrays, and between the adders/register arrays and accumulators, are determined by the map table during runtime.

D. Multipliers and Adders

The double-precision floating-point multipliers and adders used in the SMVM architecture are pipelined and have two inputs. In each clock cycle, k pairs of numbers from the sparse matrix and vector are fed into the inputs of the k multipliers. After some clock cycles, the k products are ready at the outputs of the multipliers. Based on the input pattern vector, some of the products are summed up together since they come from the same row of the input matrix. Since each adder has only two inputs, an adder tree is leveraged to sum more than two products. A product does not necessarily go through the adder tree if it is the only nonzero in a row during a specific clock cycle. For example, the first and second products will not be put into the adder tree if IPV $= 1101/1100$ in Fig. 2. In this scenario, these products are sent into register arrays between the map tables in Fig. 1, which have the same delay as the adders to guarantee they arrive at the next level at the same clock cycle as those sums of products. The register arrays are simply 64-bit FIFO queues. One 64-bit number (corresponding to an IEEE 754 double-precision floating-point value) is pipelined out of an array in each clock cycle. Due to the associativity of addition, $k-1$ adders are needed to sum up k numbers. Therefore, $k-1$ floating-point adders are enough to reduce no more than k products in our architecture. The adder tree has in total $\lceil log_2 k \rceil$ levels. The number of adders in each level is generated by the algorithm shown in Appendix B. The number of registers in the array in each level is k, which guarantees all products go through them if all the bits in the IPV are 1.

E. Adder Accumulator

When a row has more than k nonzeros, say n_z^i, the products have to be moved into the SMVM engine in $\lceil n_z^i/k \rceil$ clock cycles. The value of n_z^i is unpredictable in a large sparse matrix. Since floating-point adders are usually deeply pipelined to achieve a high clock frequency, and the subsums of each row are generated in different clock cycles, designing a floating-point adder accumulator is not trivial. Several AAC (also referred to as reduction circuit) architectures have been proposed in recent years [30]–[33]. The AAC in [34] can reduce an arbitrary number of groups with arbitrary size and does not require any preprocessing of the inputs. Consequently, it is suitable to handle the accumulation operation for SMVM with an irregular input matrix. The AAC accepts a data input of a group in each clock cycle. Once all the elements in a group go into the AAC, the sum of the group will be ready as an output after some number clock cycles. Fig. 3 is a timing diagram of the AAC computation model in [34]. The products for row A, B, C, and D are sent to the AAC sequentially. The number of pipeline stages of the AAC is two, and consequently, the sum of a row is ready two clock cycles after the last product is sent to the AAC.

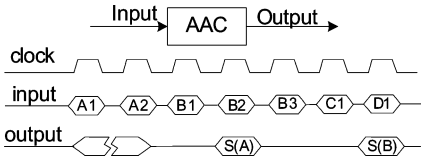


Fig. 3. Timing diagram of AAC computation model.

In the extreme case, the AAC acts as a FIFO queue if each row only includes one product.

Taking AAC as a building block renders neatness and compactness to the SMVM architecture. For some patterns in Fig. 2, the sum of products of a row in current clock cycle t_i should be accumulated with the sum of products of the same row in the (t_i-1)th and/or the (t_i+1)th clock cycle. In a clock cycle, the products may come from several rows. Only the sum of products in the first and last row in the current clock cycle need be accumulated with products in adjacent clock cycles. The products in the other rows are already reduced into one sum by the register-adder tree. Therefore, only two AACs should be used to accumulate the sum of products generated by the register-adder tree. The left AAC accumulates the sum of products of the first row in the current clock cycle with the sum of products of the last row in the previous clock cycle. In this case, the first row in the current clock cycle and the last row in the previous clock cycle are the same. The right AAC accumulates the sum of products of the last row in the current clock cycle and the sum products of the first row in the next clock cycle. In this case, the last row in the current clock cycle and the first row in the next clock cycle are the same.

For example, if we have $\mathrm{IPV} = 1000$ in clock cycle t_{i-1} and $\mathrm{IPV} = 1010$ in clock cycle t_i, the sum of the last three products in clock cycle t_{i-1} should be accumulated with the first product in clock cycle t_i. The second and third products in clock cycle t_i come from the same row and are reduced into one sum by the register-adder tree which should not be accumulated with any other products. The last product in clock cycle t_i should be accumulated with the first sum of products in clock cycle t_{i+1}. If there is only one row in a clock cycle, the singular sum of products reduced by the register-adder tree is sent to the left AAC. There are at most k reduced sums in each clock cycle if the k products come from k different rows. Hence, another $k-2$ register arrays are placed in the same level with the two AACs. The inputs to these register arrays are the reduced sums of rows which are resolved in the register-adder tree. If the products in the previous clock cycle are from at least two rows and the last row has products in both previous and current clock cycle, the positions of the two AACs must be switched in the current clock cycle (illustrated as the horizontal arrow for AAC inputs in Fig. 1). This assures the sum of products of a row are sent to the same AAC for accumulation.

F. IPV Reduction

The structure of the adder tree is entirely determined by the IPV. On the other hand, the register-adder tree structure is dynamically generated at run-time to fully utilize the configurable ability of the FPGA. Therefore, the IPV is an ideal source to code and index the map table and is critical to generate valid outputs. The process of IPV reduction and mapping to register-adder tree is illustrated in Fig. 4. The IPV is checked bit by bit from left to right. If the current bit is 0, it is paired off with the next bit. Each bit is only in one pair. The pattern of a pair is either (0, 0) or (0, 1) which is reduced to 0 or 1 and sent to the next level. Those bits which are not in any pairs are sent to the next level directly. This reduction procedure is repeated in each level and executed $\lceil log_2 k \rceil$ times. The length of the IPV decreases or remains unchanged during the reduction process. Initially, each bit in the IPV corresponds to a multiplier in the SMVM circuit. If a bit in the IPV is reduced with another adjacent bit, the product of the corresponding multiplier is summed up with its adjacent product; otherwise, the product is sent to the register array. The number of valid outputs in the register adder tree is equal to the length of the reduced IPV in the same level. The structure of the register-adder tree is a bijection of the IPV. The isomorphism between them is the proof of correctness of the map table generator discussed later. The Reduced IPV (RIPV) has the same number of 1 s as the original IPV with all set bits "pushed" to the left.

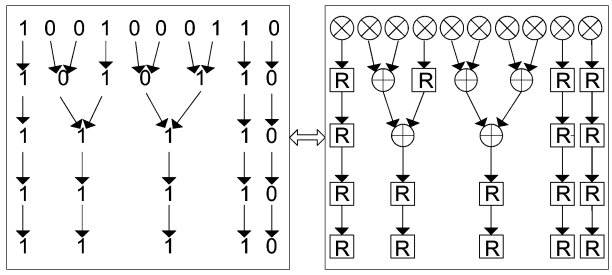


Fig. 4. IPV Reduction and tree structure ($k = 10, \mathrm{IPV} = 1001000110$).

G. Map Table

The map table is a 2-D array and provides connections between the inputs and outputs of adjacent levels during run-time. For an architecture with k multipliers, k numbers from the matrix are multiplied with k numbers from the vector in each clock cycle. At the same time, a k-bit IPV is transferred to the register array pipeline. The map table entries used in each level and clock cycle are solely determined by the integer value of the IPV. That is, the integer values of the IPVs are used to index the map table. The register arrays for IPVs shown in the rightmost column of Fig. 1 are k-bit FIFO queues.

In order to create the map table, the inputs and outputs of each level are assigned an index as shown in Fig. 1. The inputs and outputs of the adders are numbered sequentially ahead of the register arrays. For each level, the index starts from 1 instead of 0, since 0 is reserved for those inputs which are unconnected. Table I shows the map table in the case of $k = 4$. The first level is the mapping between the multipliers and the register-adder tree. The second level is the mapping inside the register-adder tree. The third level represents the mapping between the register-adder tree and the register-AAC group. Each input in a specific level l whose column index is i and row index is $\widehat{ipv}$ in the table has a value $\widehat{T}[\widehat{ipv}][i]$. This indicates that the ith input of the map table in level l should be connected to the $\widehat{T}[\widehat{ipv}][i]$th output of the map table in level l when the IPV of level l in the

TABLE I
MAP TABLE ($k = 4$)

		Level 1								Level 2						Level 3			
IPV	**Index**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**1**	**2**	**3**	**4**	**5**	**6**	**1**	**2**	**3**	**4**
000X	**0**	1	2	3	4	0	0	0	0	1	2	0	0	0	0	1	0	0	0
001X	**1**	1	2	0	0	3	4	0	0	1	3	4	0	0	0	1	0	0	2
010X	**2**	1	2	3	4	0	0	0	0	0	0	1	2	0	0	2	0	0	3
011X	**3**	1	2	0	0	3	4	0	0	0	0	1	3	4	0	2	3	0	4
100X	**4**	2	3	0	0	1	4	0	0	1	4	3	0	0	0	2	0	0	1
101X	**5**	2	3	0	0	1	4	0	0	0	0	3	1	4	0	2	3	0	4
110X	**6**	3	4	0	0	1	2	0	0	0	0	3	4	1	0	2	3	0	4
111X	**7**	0	0	0	0	1	2	3	4	0	0	3	4	5	6	2	3	4	5

current clock cycle is equal to $\widehat{ipv}$. If $\widehat{T}[\widehat{ipv}][i]$ is equal to 0, the input is not connected to any interface in the current clock cycle and the input value of it is 0. Notice that the last output to the last level is always sent to the right AAC. The rationale behind this is that if the outputs in the last level are from different rows, the output from the last row may need to accumulated with the sum of the products in the same row of the next clock cycle. If there is only one row, the sum of the products reduced by the register-adder tree is sent to the left AAC.

The map table generation is closely related to the IPV reduction. The map table generator checks the IPVs bit by bit from left to right. Based on the action of reduction, it puts the outputs of the previous level into the inputs of adders or register arrays. We used a software implementation of Alg. 2 (listed in Appendix) to generate the map table. The output of it for $k = 4$ is similar to Table I. The output table is then manually hard-coded into the hardware description language as a 2-D array which is synthesized and put into distributed memory of FPGA chips For an architecture with k multipliers, the map table is solely determined by the value of k and is independent of the characteristics of matrices or vectors. Note that the software algorithm is not implemented as part of the architecture and has no impact on the run-time performance—map table generation is a one-time task that is generated offline for a given value of k.

H. Outputs

The AACs and $k-2$ register arrays in Fig. 1 send the sums of products to the k output registers in each clock cycle. However, there will not be k valid outputs from k different rows unless the first $k-1$ bits of the corresponding IPV are all ones. Some outputs are just partial sums of products from different rows. In this sense, we need a k-bit Valid Output Vector (VOV) to indicate the valid outputs in each clock cycle. If the ith bit of the VOV is 1, the ith output register has a valid output; otherwise, its output is a partial sum of a row.

The number of ones remains unchanged during the IPV reduction. From the definition of the IPV, the ith bit of a IPV is 1 if the ith number is the last nonzero in a row. That implies that all the nonzeros of the ith row have been transferred into the system. After the reduction of the register-adder tree with or without the accumulation of AACs, the sum of products of that row is ready at an output register. In other words, the number of valid outputs in a clock cycle is equal to the number of ones in the corresponding IPV. Consequently, the IPV has a close relationship with the VOV. Nevertheless, the positions of the valid outputs in the output registers may not be the same as those indicated in IPV, due to the reduction in the register-adder tree and accumulators. Based on Alg. 2, the products are sent to the left operation units in the architecture as often as possible except that the final product is sent to the right AAC. Therefore, one can expect that all the valid outputs are packed into the left side of the output register arrays except the last one. If we shift the output of the right AAC to the left position adjacent to a valid output, the Reduced IPV (RIPV) in the last level can be used as the VOV. In our architecture, we generate the VOV by extending the RIPV to k bits filled with zeros if the number of ones is less than k. The output of the right AAC is sent to the $(i+1)$th output register if the number of ones in the current IPV is i. In this way, there are k outputs from the output registers and a k-bit VOV in each clock cycle. If the ith bit of the VOV is 1, the output of the ith output register is a valid sum for a row. The reduce module in Fig. 1 generates the VOV. Notice that the IPV reduction for VOV here is different from the IPV reduction in the map table generation. The former is performed during run-time for each IPV input while the latter is processed off-line in Alg 2 .

IV. PERFORMANCE EVALUATION

The performance of the computing platform as a whole system determines the speed of SMVM. SMVM is a typical data-intensive application where the processing requirement scales linearly with data size. IO bandwidth is a usually major performance constraint for data-intensive computing. However, the IO bandwidth and pin count requirement of the FPGA chip is practically determined by design strategy and where the original data is stored. For example, the computation power of our SMVM engine can be fully utilized and the required IO bandwidth is negligible if the data is stored in on-chip memory. The required number of pins can be greatly reduced if the SMVM module is wrapped in other modules.

Consequently, a memory-management methodology that would improve the effectiveness of our approach is to partition large sparse matrices into blocks of smaller size which can fit into the architecture. Similarly, the loading time cannot be ignored if the original data is stored in external storage (e.g., hard disk, network storage). Our SMVM architecture can be considered as an engine with k processing elements (PEs). Each PE processes one element from the input matrix in one clock cycle. If the whole matrix is partitioned into n_z/k blocks, all elements in one block can be processed in one clock cycle, with the blocks loaded into the engine and processed sequentially. This is also referred to as the *locally parallel globally*

TABLE II
COMPARISON OF SMVM HARDWARE ARCHITECTURES

	Multiplier	Adder	AAC	FLOPS	Required Overhead
[2]	k	$k-1$	1	$2kf/(1+\alpha)$	Zero padding
[27]	k	k	0	$2kf \times 66\%$	Static data scheduling, zero padding
[28]	k	k	1	$< 2kf$	Pipeline stalling, zero padding
[29]	k	k	0	$2kf \times u$	Strip elements reordering, only for FE matrices
[36]	k	k	k	$< 2kf$	Data preprocessing, dynamic scheduling, zero padding
This Work	k	$k-1$	2	$2kf$	IPV generation

sequential(LPGS) method in [35]. If the whole SMVM engine is considered as one PE and multiple FPGA chips are used, then *locally sequential globallly parallel (LSGP)* methods [35] can be used. In the LSGP approach, a large matrix can be divided into multiple blocks where each block contains a continuous sequence of rows. Our planned future work in porting this architecture to the Convey HC-1 platform [11] will adopt the LSGP method. A discussion of these and other partition schemes can be found in [2], [27].

A. Architectural Comparison

Floating point Operations Per Second (FLOPS) is a common measure of computing performance in high-performance computing environments. To compute $Y = AX$, each nonzero element of A requires two floating-point operations. The FLOPS are computed by the following equations where n_{op} is the total number of floating-point operations.

$$\text{FLOPS} = \frac{n_{op}}{t_{\text{total}}} \approx \frac{2n_z}{\frac{n_z}{k} \times t_{\text{clock}}} = 2k \times f_{\text{clock}}. \qquad (1)$$

Independent of the IO bandwidth and data source, the comparison of different hardware SMVM architectures is shown in Table II, where f is the clock frequency, α is the zero-padding overhead [2], and $u \in [0,1]$ is a utilization factor [29]. Note that the IPVs are generated only once for each sparse matrix given a specific k. In the scenario of computing $Y = A^m X$, for $m \gg 1$, the IPVs are generated once for matrix A. For each iteration, the IPVs are sent with the data to the FPGA. This is a one-time overhead, as compared to the other architectures in Table II for which the overhead recurs across all iterations.

B. Demonstration in Reconfigurable System

1) Experimental Setup: The XD2000i from XtremeData [10] is an example of a cutting-edge system that provides a complete platform to deploy high-performance computing solutions with FPGAs. It greatly reduces the development effort of integrating the software and hardware components. The XD2000i consists of a Dual Xeon motherboard with one Intel Xeon processor and two Altera Stratix III EP3SE260 FPGAs in the other socket as shown in Fig. 5. The Xeon CPU has 4 GB system memory and four cores running at 1.6 GHz which communicate with the FPGAs through the Intel Memory Controller Hub (MCH). The MCH is connected to the bridge FPGA via the 1067 M Front Side Bus (FSB) interface. The FSB with 8.5 GBytes/s peak communication bandwidth is the highest performing, lowest latency bus in this generation of Intel platform [37]. The communication behavior between the MCH and FPGAs is nailed down to the cycle accuracy level.

The clock speed of the FPGAs is hardwired to 100 MHz by the platform. We used the Altera MegaWizard Plug-In Manager to customize the IEEE 754 double-precision floating-point multipliers and adders in our implementation. The multipliers were configured to use dedicated multiplier circuitry (DSPs). Our SMVM engine is integrated into one of the application FPGAs with other hardware IP components provided by XtremeData. Specifically, the SMVM engine module is wrapped in an $afu\_loopback$ module which is again wrapped in an app_core module provided by the XtremeData XD2000i platform. These modules serve as the intermediator between the SMVM engine and the bridge FPGA. The design was simulated, synthesized, placed and routed using Altera Quartus II 8.1.

During the run-time, the CPU reads the entries in the input sparse matrices and vectors from disk and prepares a data buffer in system memory. The size of the buffer for transmission is no larger than 4 MB, which is an XD2000i constraint. Once the data buffer is full, it is sent to the FPGA chip directly using a send-request command. Only one Send operation may be active at a time. The Receive command corresponding to the previous Send command must complete before the next Send command is submitted. However, the Send request may return immediately so that CPU can perform other operations before sending the Receive request. The computation in hardware is overlapped with the fetching data operation in software. A data buffer of known size is received back from the FPGA hardware and is stored in system memory [10]. We calculate the size of the receiving data buffer by counting the number of rows which have at least one nonzero in the sending data buffer. Based on our experimental results, the bandwidth available for sending data from CPU to FPGA and back to CPU during run-time is 1.035 GBytes/s in each direction, which is solely a function of the XtremeData architecture and drivers. All inputs are set to zeros to fill the idle clock cycles during which no data has arrived.

The performance of our system implemented on the XtremeData XD2000i platform is compared with SparseLib++ [38], which is a widely-used general-purpose software implementation and an object oriented C++ library for sparse matrix computations. It is designed for portability and high performance across a wide class of machine architectures [39]. The SparseLib++ is built upon the Sparse BLAS which provides high-performance sparse matrix-vector kernels that can be application or platform-specific [40].

2) Parameter Design: The design is characterized by several parameters. The depth of the pipelines in the multipliers, adders and AACs is 11, 14, and 48, respectively. Increasing these parameters would achieve a higher clock frequency at the expense of greater resource usage and initial latency. The number of multipliers, adders and register arrays, the width of IPV and the

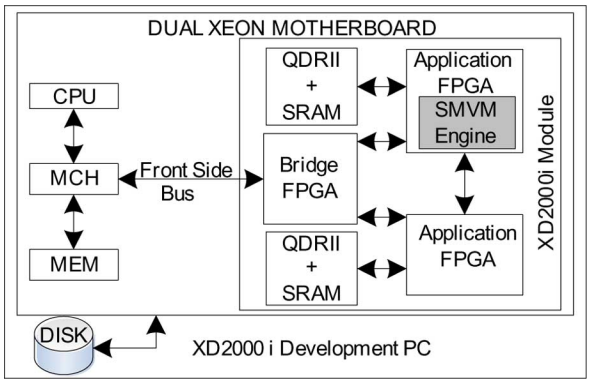


Fig. 5. XtremeData XD2000i architecture with SMVM engine.

size of the map-table increase with k. Depending on the design strategy and data source, the value of k is usually determined by several factors including IO bandwidth, clock speed, pin count of the chip, and FPGA logic resources. However, the pin count of the FPGA chip has no impact on k since the SMVM engine must be wrapped in other hardware modules in the XD2000i platform. The on-chip memory such as BRAMs or distributed memory usually runs at the same clock frequency as the logic and the bit-widths are large enough. If the data is originally stored in the on-chip memory, the number of multipliers k is only limited by the logic resource in FPGAs, allowing the SMVM performance to grow linearly with the value of k. When the data source is off-chip, the IO bandwidth constrains the value of k in this platform. In our SMVM architecture, the number of input bits in each clock cycle is $2k \times 64 + k$ if double-precision floating-point data is used. The calculation of k based on peak IO bandwidth is consequently

$$k \approx \frac{\text{IO bandwidth}}{129 \times f_{\text{clock}}} = \left\lceil \frac{\frac{8.5\text{GBytes}}{s}}{129\text{bits} \times 100\text{MHz}} \right\rceil = 6. \quad (2)$$

Based on the values of clock frequency and IO bandwidth shown in Section IV-B-I, $k = 6$ is enough to fully utilize the peak bandwidth of the FSB. Hence, using more multipliers will not provide higher performance given that communication bandwidth would become the bottleneck. Only one application FPGA is used since the two FPGA chips share the same communication channel. In Table III, the computation time for FPGA ranges from $k = 3$ to $k = 12$ (the design complexity exceeded the capabilities of the Quartus FPGA synthesis tool for $k > 12$). The SMVM architecture uses 62% of the total logic in the Stratix III EP3SE260 FPGA when $k = 12$. The components in the whole XD2000i system which incorporates the SMVM architecture into the FPGA consume 14% of the total available FPGA LUT resources.

3) Experimental Results: In order to have a fair comparison, the SparseLib++ software also runs on the XtremeData 2000i platform using one CPU of the Intel Xeon subsystem. Table III characterizes the sparse matrices from the University of Florida Sparse Matrix Collection [41]. The matrices are stored as MatrixMarket Coordinate format (shown in Appendix A). These matrices have different sparsity patterns. The nonzeros in `ex25`, `mcfe`, `jpwh_991`, `cdde2` are confined to a diagonal band. The

TABLE III
SMVM CHARACTERISITICS ON MATRICES

Name	Size	Nonzeros	t_{read}	t_{trans}	t_{ipv}
`str_600`	363×363	3279	6459	61	36
`cdde2`	961×961	4681	10741	87	44
`bp_1600`	822×822	4841	9387	90	63
`jpwh_991`	991×991	6027	9660	112	78
`mcfe`	765×765	24382	41630	452	171
`ex25`	848×848	24612	54897	457	173
`tomography`	500×500	28726	60897	533	200

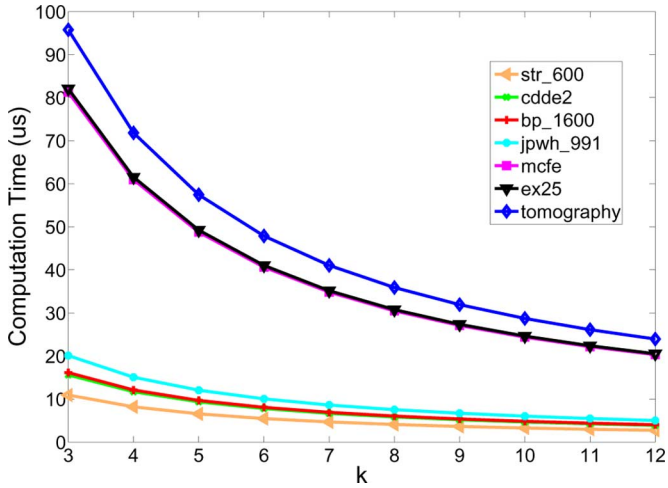


Fig. 6. FPGA computation time as a function of k.

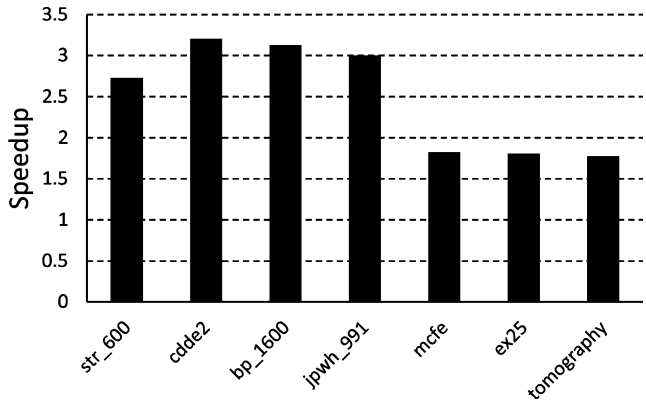


Fig. 7. Speedup of FPGA computation over CPU (for $k = 6$).

sparsity patterns of tomography, `bp_1600` and `str_600` are not obvious. The values in the vectors are generated randomly and saved in a file on disk. The FPGA computation time for different benchmarks is shown in Fig. 6. The computation speedup of FPGA over CPU (the running time ratio of CPU to FPGA) is depicted in Fig. 7. Note that our design can run at higher clock frequencies in the Stratix III EP3SE260 FPGA, which would require linearly less time for computation.

The floating-point data is stored in scientific notation format. The t_{read} in Table III is the time for reading the matrix and vector from the disk into the system memory. The size of the file is determined by the number of digits in the mantissa which explains t_{read} for `cdde2` is larger than that of `bp_1600` even though `cdde2` has less nonzeros. The t_{trans} is the time for transferring the data from the system memory to the FPGA chips. The IPV generation time by software is t_{ipv}. The time is measured in

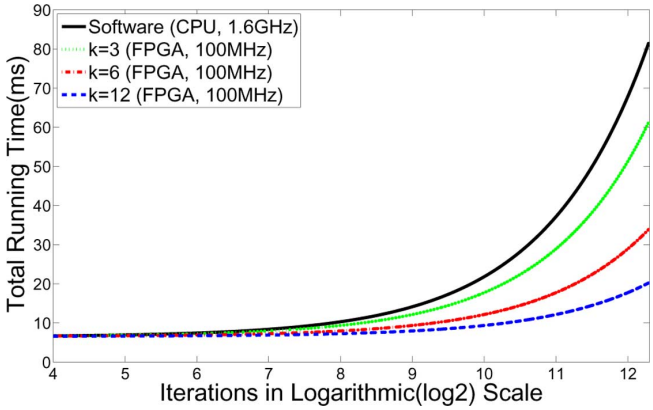


Fig. 8. Total running time comparison for `str_600`.

microseconds. For a sparse matrix, SparseLib++ reads the matrix and vector from disk to the system memory before sending them to the CPU for computation. Therefore, the preprocessing time is t_{read}. To use the SMVM engine in FPGAs, the data in the system memory buffer along with the generated IPVs is sent to the FPGA chips. The preprocessing time is $t_{read}+t_{trans}+t_{ipv}$.

To cope with the deficiency in input communication bandwidth, a finite state machine (FSM) in the $afu\_loopback$ module implemented in VHDL is used to feed data into the SMVM engine. In each clock cycle, the FSM collects valid input data from the $write\_data$ bus and stores it into the input buffer of SMVM engine. If the buffer is full (with k pairs of data and an IPV), the data in it is sent to the SMVM engine. Otherwise, the inputs of SMVM engine are all zeros and the FSM stays in the data collecting state. From Fig. 3, the computation time is much smaller than the preprocessing time in computing $Y = A^m X$ for $m = 1$. Fig. 8 compares the total running time of CPU and FPGA increasing with m (the number of SMVM iterations) when to compute $Y = A^m X$. If m is small, the CPU system is even faster than the FPGA system because of the proprocessing overhead. Nevertheless, the running time is dominated by the computation time when SMVM is heavily used in practical computation.

It should be noted that assuming the input matrix is stored continuously on-chip, the scale of the problem (the number of nonzeros) is constrained by the available on-chip memory. For example, in Altera Stratix III EP3SE260, two M144 K memory blocks with 4 k × 64 (depth × width) are used to store the matrix and vector. When $k = 6$, 40 M144 K blocks are combined to serve as IO storage. Since there are in total 48 M144 K blocks, the number of nonzeros in the matrix should be no more than $4k$. Partitioning large matrices for iterative computation in FPGAs is a planned aspect of our future work. In summary, the SMVM architecture as implemented on XD2000i platform has an advantage over the traditional approach in system level when the source data is stored in the on-chip memory of FPGAs or the SMVM is iteratively used (such as the power method for computing $Y = A^m X$, for $m \gg 1$).

V. Conclusion and Future Work

An expendable and high performance sparse matrix-vector multiplication architecture is proposed in this paper. The paths in which the data flows through the architecture are dynamically determined by the map table during runtime. The architecture can deal with sparse matrices with arbitrary size and sparsity pattern. It eliminates zero-paddings and pipeline stalls with the introduction of IPVs. With enough communication bandwidth and hardware resources, the performance of the architecture grows linearly with the number of multipliers, k, which is a configurable parameter in the architecture. We implemented the architecture on the XtremeData2000i reconfigurable platform. Compared with an existing optimized software implementation, our design is significantly faster in computation. However, our architecture is not limited by a specific platform. In our planned future work, the SMVM architecture will be ported to a new platform (the Convey HC-1 [11]), that is better capable of taking advantage of multiple FPGAs and independent memory controllers.

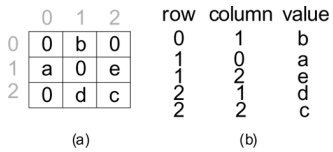


Fig. 9. Example of sparse matrix representation format. (a) Matrix. (b) MatrixMarket coordinate format.

Appendix A
Sparse Matrix Storage Format and IPV Generation

Our architecture is independent of the matrix storage format as long as the data is stored in row-major order. For the demonstration in Section IV-B, we use the MatrixMarket Coordinate format which is used to represent general sparse matrices. The coordinates and values are explicitly given for each nonzero element in a matrix. An example is shown in Fig. 9.

The generation of IPVs is straightforward according to the definition. To send b, a, e, d to our architecture with $k = 4$ in the same clock cycle, the k-bit IPV is 1010. Based on the sparse matrix storage format, IPVs can be attached as meta-data if the value of k is known for the architecture to be used. Alternately, IPVs can be generated during the computation which will introduce some computation overhead.

Appendix B
Adder Number Generation Algorithm

Algorithm 1: Generating the Adder Number in Each Level

Input: the number of multipliers, k

Output: the number of adders in each level, $\widehat{N}[\,]$

$K \leftarrow k$;

for $i = 1$ to $\lceil log_2 k \rceil$ **do**

$\widehat{N}[i] \leftarrow K/2$;

$K \leftarrow K/2 + K\%2$;

end for

The adder tree has $\lceil log_2 k \rceil$ levels. The number of adders in each level is determined when all the inputs are from the same row in a particular clock cycle. In the case the inputs are from

different rows, register arrays must be used in some level. Suppose all the k inputs are from the same row in a clock cycle. After some clock cycles, there are k outputs from the k multipliers. These k outputs serve as the k inputs to the adders in the first level of register-adder tree where they are sent to $k/2$ adders. If k is an odd integer, one of the outputs is sent to the register array. After some clock cycles, the number of intermediate results sent to the register-adder tree in the next level is $k/2+k\%2$. The same principle is applied to each level as shown in Alg. 1.

APPENDIX C
MAP TABLE GENERATION ALGORITHM

Algorithm 2: Map Table Generation Algorithm

Input: the number of multipliers, k

Output: the map table, $\widehat{T}[\,][\,]$

for $ipv = 0$ to $2^{k-1} - 1$ **do**

Convert ipv to a binary vector $\widehat{ipv}$;

$\widehat{B1} \leftarrow \widehat{ipv}$; $\widehat{B2} \leftarrow \emptyset$; $\widehat{S1} \leftarrow \{1 \cdots k\}$; $\widehat{S2} \leftarrow \emptyset$;

for $level = 1$ to $\lceil log_2 k \rceil$ **do**

$i_{\widehat{B1}}, i_{\widehat{B2}} \leftarrow 0$; $i_a^{out} \leftarrow 1$; $i_r^{out} \leftarrow \widehat{N}[level] + 1$;

$i_a^{in} \leftarrow i_s^{in}$; $i_r^{in} \leftarrow i_s^{in} + 2\widehat{N}[level]$;

while $i_{\widehat{B1}} < l_{\widehat{B1}}$ **do**

if $level = \lceil log_2 k \rceil$ and $i_{\widehat{B1}} = l_{\widehat{B1}} - 1$ **then**

$\widehat{T}[ipv][3k - 2 + k\lceil log_2 k \rceil] \leftarrow \widehat{S1}[i_{\widehat{B1}}]$;

$\widehat{S2}[i_{\widehat{B2}}] \leftarrow k$; $\widehat{B2}[i_{\widehat{B2}}] \leftarrow \widehat{B1}[i_{\widehat{B1}}]$;

$i_r^{out}, i_r^{in}, i_{\widehat{B1}}, i_{\widehat{B2}} \leftarrow increase\ 1$;

else if $\widehat{B1}[i_{\widehat{B1}}] = 1$ or $i_{\widehat{B1}} = l_{\widehat{B1}} - 1$ **then**

$\widehat{T}[ipv][i_r^{in}] \leftarrow \widehat{S1}[i_{\widehat{B1}}]$;

$\widehat{S2}[i_{\widehat{B2}}] \leftarrow i_r^{out}$; $\widehat{B2}[i_{\widehat{B2}}] \leftarrow \widehat{B1}[i_{\widehat{B1}}]$;

$i_r^{out}, i_r^{in}, i_{\widehat{B1}}, i_{\widehat{B2}} \leftarrow increase\ 1$;

else

$\widehat{T}[ipv][i_a^{in}] \leftarrow \widehat{S1}[i_{\widehat{B1}}]$;

$\widehat{T}[ipv][i_a^{in} + 1] \leftarrow \widehat{S1}[i_{\widehat{B1}} + 1]$;

$\widehat{S2}[i_{\widehat{B2}}] \leftarrow i_a^{out}$; $\widehat{B2}[i_{\widehat{B2}}] \leftarrow \widehat{B1}[i_{\widehat{B1}} + 1]$;

$i_a^{in}, i_{\widehat{B1}} \leftarrow increase\ 2$; $i_a^{out}, i_{\widehat{B2}} \leftarrow increase\ 1$;

end if

end while

$i_s^{in} \leftarrow i_s^{in} + 2N[level] + k$;

$\widehat{B1} \leftarrow \widehat{B2}$; $l_{\widehat{B1}} \leftarrow i_{\widehat{B2}}$; $\widehat{S1} \leftarrow \widehat{S2}$;

end for

end for

The software algorithm of the map table generator is shown in Alg. 2 which iterates on different values of the IPV. The number of rows in the table is 2^{k-1} where each row corresponds to a distinct IPV. For each given IPV, the algorithm produces the entries of the map table level by level. In general, the number of inputs in level i are $2 \times a_i + k$, where a_i is the number of adders in level i. There are in total $k - 1$ adders and $\lceil log_2 k \rceil$ levels, while each level has k register arrays. The last level which is a register-AAC group always has k inputs. By adding up the number of inputs in each level, there are in total $2 \times (k-1) + k \times \lceil log_2 k \rceil + k$ inputs which is also the number of columns in the map table.

For each level, $\widehat{B1}$ is the IPV which is reduced in the current level; $\widehat{B2}$ is the IPV generated by reducing $\widehat{B1}$ in the current level. $\widehat{S1}$ is the output index vector of the previous level while $\widehat{S2}$ is the output index vector of the current level. If the index of the ith output in the previous level is j, then $\widehat{S1}[i] = j$. The indices are the numbers assigned to the adder outputs and register arrays outputs in Fig. 1. Recall that the adders and register arrays are numbered sequentially. However, the register arrays are possibly used ahead of the adders during the reduction of IPVs. Hence, we need $\widehat{S}$ to record the output index sequence. Obviously, the value of $\widehat{S}$ of the first level is the same as the serial number assigned to the multipliers, i.e., $\widehat{S1} = \{1, 2, \ldots, k\}$. i_a^{out} and i_r^{out} are the index counters of those numbers assigned to the outputs of the adders and register arrays respectively. $i_{\widehat{B}}$ is the index counter for $\widehat{B}$ and $\widehat{S}$. In Fig. 1, a unique number is assigned to each input of the adders and register arrays in a level. i_a^{in} and i_r^{in} are the input index counters of those numbers for the adders and register arrays respectively. i_s^{in} is the index of the map table for each ipv. $\widehat{N}[i]$ is the number of adders in level i.

An IPV is reduced in a $while$ loop as shown in Alg. 2. As byproducts of the $while$ loop, a reduced IPV $\widehat{B2}$ and output index vector of the current level $\widehat{S2}$ are generated. There are three scenarios while reducing IPVs. In the first scenario, the current index is the last input of the last level. The last output in previous level is sent to $\widehat{T}[ipv][2 \times (k-1) + k \times \lceil log_2 k \rceil + k]$, i.e., the input of the right AAC. Otherwise, if the current bit in $\widehat{B1}$ can not be reduced or is the last bit, the current indexed output of previous level is sent to the current indexed register, i.e., $\widehat{T}[ipv][i_r^{in}] \leftarrow \widehat{S1}[i_{\widehat{B1}}]$. If the current bit should be reduced with the next bit in $\widehat{B}$, as shown in the last scenario, the corresponding two outputs in the previous level should be sent to the two inputs of an adder in the current level. Meanwhile, the two bits $\widehat{B1}[i_{\widehat{B1}}]$ and $\widehat{B1}[i_{\widehat{B1}} + 1]$ in the current IPV are reduced into $\widehat{B2}[i_{\widehat{B2}}]$ in the next IPV. The value of the newly reduced bit $\widehat{B2}[i_{\widehat{B2}}]$ is equal to $\widehat{B1}[i_{\widehat{B1}} + 1]$ since $\widehat{B1}[i_{\widehat{B1}}]$ is equal to 0.

The maximum value of an entry is $k + \lfloor k/2 \rfloor$ which is the number of outputs of the first level in the register-adder tree. Therefore, $\lceil log_2(k + \lfloor k/2 \rfloor) \rceil$ bits are employed to store each entry. The number of bits to store a map table is $2^{k-1} \times (2 \times (k-1) + k \times \lceil log_2 k \rceil + k) \times \lceil log_2(k + \lfloor k/2 \rfloor) \rceil$. The number of inputs in each level is always no less than the number of outputs in the previous level.

REFERENCES

[1] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel, "Optimization of sparse matrix-vector multiplication on emerging multicore platforms," presented at the Supercomputing (SC), 2007.

[2] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," presented at the Int. Symp. Field Programmable Gate Arrays (FPGA), Feb. 2005.

[3] , A. N. Langville and C. D. Meyer, Eds., *Google's PageRank and Beyond: The Science of Search Engine Rankings*. Princeton, NJ: Princeton Univ. Press, 2006.

[4] M. M. Baskaran and R. Bordawekar, Optimizing Sparse Matrix-Vector Multiplication on GPUs, 2009, IBM Research Report, Tech. Rep. RC24704.

[5] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," presented at the Supercomputing (SC), 1999.

[6] E.-J. Im and K. Yelick, "Optimizing sparse matrix computations for register reuse in SPARSITY," presented at the Int. Conf. Computational Science, 2001.

[7] General-Purpose Computation on Graphics Hardware [Online]. Available: http://www.gpgpu.org

[8] SRC Computers, LLC [Online]. Available: http://www.srccomp.com

[9] Cray Inc. [Online]. Available: http://www.cray.com

[10] XD2000i Development System User Handbook, XtremeData Inc. [Online]. Available: http://www.xtremedata.com

[11] Convey Computer Corporation [Online]. Available: http://www.conveycomputer.com

[12] K. Underwood, K. S. Hemmert, and C. Ulmer, "Architectures and APIs: Assessing requirements for delivering FPGA performance to applications," presented at the Supercomputing (SC) 2006.

[13] A. T. Ogielski and W. Aiello, "Sparse matrix computations on parallel processor arrays," *SIAM J. Sci. Comput.*, vol. 14, pp. 519–530, 1993.

[14] S. Toledo, "Improving memory-system performance of sparse matrix-vector multiplication," *IBM J. Res. Develop.*, 1997.

[15] N. Bell and M. Garland, Efficient Sparse Matrix-Vector Multiplication on CUDA, NVIDIA Corporation, 2008, Tech. Rep. NVR-2008-004.

[16] N. Bell and M. Garland, "Implementing sparse matrix-vector multiplication on throughput-oriented processors," presented at the Supercomputing (SC), Aug. 2009.

[17] A. Monakov and A. Avetisyan, "Implementing blocked sparse matrix-vector multiplication on NVIDIA GPUs," presented at the Int. Symp. Systems, Architectures, Modeling and Simulation (SAMOS), 2009.

[18] Garland and Michael, "Sparse matrix computations on manycore GPUs," presented at the Design Automation Conf. (DAC), 2008.

[19] Sparse Matrix-Vector Multiplication Toolkit for Graphics Processing Units, IBM., 2009 [Online]. Available: http://www.alphaworks.ibm.com/tech/spmv4gpu

[20] Cudpp: Cuda Data Parallel Primitives Library, Nvidia [Online]. Available: http://gpgpu.org/developer/cudpp

[21] O. Nibouche, S. Boussakta, and M. Darnell, "Pipeline architectures for radix-2 new mersenne number transform," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 56, no. 8, pp. 1668–1680, Aug. 2009.

[22] J. Liu, Y. V. Zakharov, and B. Weaver, "Architecture and FPGA design of dichotomous coordinate descent algorithms," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 56, no. 11, pp. 2425–2438, Nov. 2009.

[23] M.-D. Shieh, J.-H. Chen, W.-C. Lin, and H.-H. Wu, "A new algorithm for high-speed modular multiplication design," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 56, no. 9, pp. 2009–2019, Sep. 2009.

[24] Y. K. Choi, K. You, J. Choi, and W. Sung, "A real-time FPGA-based 20000-word speech recognizer with optimized DRAM access," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 57, no. 8, pp. 2119–2131, Aug. 2010.

[25] C. Zhang, C. Wang, and M. O. Ahmad, "A pipelined VLSI architecture for high-speed computation of the 1-D discrete wavelet transform," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 57, no. 10, pp. 2729–2740, Oct. 2010.

[26] X. Chen, J. Kang, S. Lin, and V. Akella, "Memory system optimization for FPGA-based implementation of quasi-cyclic LDPC codes decoders," *IEEE Trans. Circuits Syst. I, Reg. Papers*, vol. 58, no. 1, pp. 98–111, Jan. 2011.

[27] M. deLorimier and A. Dehon, "Floating-point sparse matrix-vector multiply for FPGAs," presented at the Int. Symp. Field Programmable Gate Arrays (FPGA), Feb. 2005.

[28] J. Sun, G. Peterson, and O. Storaasli, "Sparse matrix-vector multiplication design on FPGAs," presented at the Int. Symp. Field-Programmable Custom Computing Machines, Apr. 2007.

[29] Y. El-Kurdi, D. Giannacopoulos, and W. J. Gross, "Hardware acceleration for finite-element electromagnetics: Efficient sparse matrix floating-point computations with FPGAs," *IEEE Trans. Magn.*, vol. 43, no. 4, pp. 1525–1528, Apr. 2007.

[30] L. Zhuo, G. Morris, and V. Prasanna, "High-performance reduction circuits using deeply pipelined operators on FPGAs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 10, pp. 1377–1392, Oct. 2007.

[31] P. Zicari, S. Perri, P. Corsonello, and G. Cocorullo, "An optimized adder accumulator for high speed macs," in *Proc. Int. Conf. ASIC (ASICON)*, Oct. 2005, pp. 757–760.

[32] F. de Dinechin, B. Pasca, O. Cret, and R. Tudoran, "An FPGA-specific approach to floating-point accumulation and sum-of-products," in *Proc. Int. Conf. Field-Programmable Technology (FPT)*, 2008, pp. 33–40.

[33] C. He, G. Qin, M. Lu, and W. Zhao, "Group-alignment based accurate floating-point summation on FPGAs," in *Proc. Int. Conf. Engineering of Reconfigurable Systems and Algorithms (ERSA)*, 2006, pp. 136–142.

[34] S. Sun and J. Zambreno, "A floating-point accumulator on FPGAs," presented at the Int. Conf. Field-Programmable Technology (FPT'09), Dec. 2009.

[35] , S. Y. Kung, Ed., *VLSI Array Processors*. Upper Saddle River, NJ: Prentice-Hall, 1988.

[36] Y. Zhang, Y. Shalabi, R. Jain, K. Nagar, and J. Bakos, "FPGA vs. GPU for sparse matrix vector multiply," in *Proc. Int. Conf. Field-Programmable Technology (FPT)*, 2009, pp. 255–262.

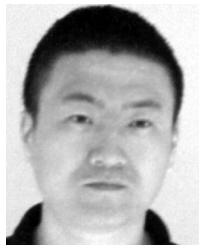
[37] L. Ling, N. Oliver, C. Bhushan, W. Qigang, A. Chen, S. Wenbo, Y. Zhihong, A. Sheiman, I. McCallum, J. Grecco, H. Mitchel, L. Dong, and P. Gupta, "High-performance, energy-efficient platforms using in-socket FPGA accelerators," in *Proc. Int. Symp. Field Programmable Gate Arrays (FPGA)*, 2009, pp. 261–264.

[38] R. Pozo, K. Remington, and A. Lumsdaine, Sparselib++ [Online]. Available: math.nist.gov/sparselib++/

[39] J. Dongarra, A. Lumsdaine, X. Niu, R. Pozo, and K. Remington, *Sparse Matrix Libraries in C++ for High Performance Architectures,* 1994.

[40] I. S. Duff, M. A. Heroux, and R. Pozo, The Sparse BLAS CERFACS, Technical Report TR/PA/01/24, 2001.

[41] T. Davis and Y. Hu, The University of Florida Sparse Matrix Collection [Online]. Available: www.cise.ufl.edu/research/sparse/matrices



Song Sun received the B.S. degree in computer science and engineering from the Beijing Information Technology Institute, China, in 1998, and the M.S. degree from the University of Science and Technology of China in 2001. He is currently a research assistant pursuing the Ph.D. degree in the Reconfigurable Computing Lab, Iowa State University, Ames. His research interests include high-performance reconfigurable computing and embedded systems.

Madhu Monga received the B.S. degree in electronics and communication from Kurukshetra University in 2003 and the M.S. degree in computer engineering from Iowa State University, Ames, in 2010.

She was a research assistant in the Reconfigurable Computing Lab, Iowa State University, until 2010, and worked on accelerating real-time simulation of dynamic systems. Her research interests include high-performance reconfigurable computing and embedded systems.

Phillip H. Jones (M'06) received the B.S. and M.S. degrees in electrical engineering from the University of Illinois at Urbana-Champaign, Urbana, in 1999 and 2002, respectively, and the Ph.D. degree in computer engineering from Washington University, St. Louis, MO, in 2008.

Currently, he is an Assistant Professor in the Department of Electrical and Computer Engineering, Iowa State University, Ames, where he has been since 2008. His research interests are in adaptive computing systems, reconfigurable hardware, embedded systems, and hardware architectures for application-specific acceleration.

Dr. Jones received Intel Corporation sponsored Graduate Engineering Minority (GEM) Fellowships from 1999–2000 and from 2003–2004, and the best paper award from the IEEE International Conference on VLSI Design in 2007.

Joseph Zambreno (M'02) received the B.S. degree (Hons.) in computer engineering in 2001, the M.S. degree in electrical and computer engineering in 2002, and the Ph.D. degree in electrical and computer engineering in 2006 from Northwestern University, Evanston, IL.

Currently, he is an Assistant Professor in the Department of Electrical and Computer Engineering, Iowa State University, Ames, where he has been since 2006. His research interests include computer architecture, compilers, embedded systems, and hardware/software co-design, with a focus on run-time reconfigurable architectures and compiler techniques for software protection.

Dr. Zambreno was a recipient of a the National Science Foundation Graduate Research Fellowship, a Northwestern University Graduate School Fellowship, a Walter P. Murphy Fellowship, and the Electrical Engineering and Computer Science Department Best Dissertation Award for his Ph.D. dissertation "Compiler and Architectural Approaches to Software Protection and Security".