Available online at www.sciencedirect.com
ScienceDirect
Computer Physics Communications 178 (2008) 558–570
Computer Physics Communications
www.elsevier.com/locate/cpc

FPGA architecture and implementation of sparse matrix–vector multiplication for the finite element method

Yousef Elkurdi, David Fernández *, Evgueni Souleimanov, Dennis Giannacopoulos, Warren J. Gross

Department of Electrical and Computer Engineering, McGill University, Montreal, Quebec, H3A 2A7, Canada

Received 24 July 2007; received in revised form 25 October 2007; accepted 29 November 2007

Available online 22 January 2008

Abstract

The Finite Element Method (FEM) is a computationally intensive scientific and engineering analysis tool that has diverse applications ranging from structural engineering to electromagnetic simulation. The trends in floating-point performance are moving in favor of Field-Programmable Gate Arrays (FPGAs), hence increasing interest has grown in the scientific community to exploit this technology. We present an architecture and implementation of an FPGA-based sparse matrix–vector multiplier (SMVM) for use in the iterative solution of large, sparse systems of equations arising from FEM applications. FEM matrices display specific sparsity patterns that can be exploited to improve the efficiency of hardware designs. Our architecture exploits FEM matrix sparsity structure to achieve a balance between performance and hardware resource requirements by relying on external SDRAM for data storage while utilizing the FPGAs computational resources in a stream-through systolic approach. The architecture is based on a pipelined linear array of processing elements (PEs) coupled with a hardware-oriented matrix striping algorithm and a partitioning scheme which enables it to process arbitrarily big matrices without changing the number of PEs in the architecture. Therefore, this architecture is only limited by the amount of external RAM available to the FPGA. The implemented SMVM-pipeline prototype contains 8 PEs and is clocked at 110 MHz obtaining a peak performance of 1.76 GFLOPS. For 8 GB/s of memory bandwidth typical of recent FPGA systems, this architecture can achieve 1.5 GFLOPS sustained performance. Using multiple instances of the pipeline, linear scaling of the peak and sustained performance can be achieved. Our stream-through architecture provides the added advantage of enabling an iterative implementation of the SMVM computation required by iterative solution techniques such as the conjugate gradient method, avoiding initialization time due to data loading and setup inside the FPGA internal memory.

PACS: 07.05.Bx; 47.11.Fg

Keywords: FPGA; SMVM; FEM

1. Introduction

Making good use of the reprogrammability feature of Field-Programmable Gate Arrays (FPGAs) motivates devising specialized algorithms and hardware designs optimized for accelerating computations for specific application areas. Finite Element Method (FEM) is a widely used engineering analysis tool based on obtaining a numerically approximate solution for a given mathematical model of a structure. The application of FEM requires the solution of a large system of linear equations at each iteration. The system of linear equations can be represented by the following equation:

$$A \times U = B, \tag{1}$$

where A is a $N \times N$ sparse matrix, U is a vector of dependent unknowns, and B is the residual vector. Since A can be large and sparse, iterative solvers are typically used to solve (1) due to their low storage requirements. N is also referred to as the *order* of the matrix A. The Conjugate Gradient (CG) method is

* Corresponding author.
E-mail addresses: yousef.el-kurdi@mail.mcgill.ca (Y. Elkurdi), david.fernandezbecerra@mail.mcgill.ca (D. Fernández), evgueni.souleimanov@mail.mcgill.ca (E. Souleimanov), dennis.giannacopoulos@mcgill.ca (D. Giannacopoulos), warren.gross@mcgill.ca (W.J. Gross).

0010-4655/$ – see front matter
doi:10.1016/j.cpc.2007.11.014

one of the most popular iterative methods used for solving large and sparse systems similar to (1). The dominant operation in each iteration of the CG algorithm is the Sparse-Matrix Vector Multiplication (SMVM) which is computed as follows:

$$Y^{(k)} = A \times X^{(k)}, \tag{2}$$

where (k) is the kth iteration of the CG algorithm, and X and Y are $N \times 1$ dense vectors.

FPGA are 2D arrays of regularly tiled reconfigurable logic blocks interconnected by a reconfigurable interconnection fabric. They also contain a variety of special purpose blocks such as internal memory, I/O blocks, and specialized arithmetic circuits. FPGAs can be used as reconfigurable processors that can exploit fine grained parallelism tailored to a specific application. FPGAs have been shown to outperform general-purpose CPUs in sustained floating-point performance [1]. Moreover, there is a great degree of pipelineability and parallelism inherent to the SMVM computation that can be exploited by FPGAs. SMVM computation involving FEM matrices requires large memory storage and bandwidth due to the large size of the FEM mesh. Many FPGA computing platforms such as the TM4 [2,3] and BEE2 [4] contain multiple external DRAM modules and provide large memory bandwidth and capacity.

On the other hand, cache based CPU architectures are not well-suited to large and sparse data sets. Large, sparse data sets increase the number of cache misses. The memory bandwidth problem limits the sustained performance of CPUs to a fraction their peak performance (less than 33% [5]). Constant increases in memory bandwidths have not alleviated this bottleneck, because the gap between it and microprocessor clock rates continues to grow [1]. Parallel systems divide data into smaller sets to overcome this bottleneck; however, space requirement, maintenance, power consumption, and complex programming and communication models limits this approach. Therefore, the parallel solution that FPGAs offer and their relative lower cost makes them a very promising alternative.

We have implemented a linear array architecture to compute SMVM for FEM applications. FEM matrices display specific sparsity patterns that can be exploited to improve the efficiency of hardware designs. Our design is aimed at matrices specific to the FEM application area. Typical FEM matrices have large dimensions (e.g., $N = 10^6$) which makes storing the matrix along with the X and Y vectors inside the FPGA prohibitive. Also, the cost of internal FPGA resources is much higher than external SDRAM which are available at commodity prices with increasing capacities and bandwidth. This means that using internal FPGA resources to store data statically instead of using them for communication and buffering purposes is cost-inefficient. Therefore, FEM matrix size scalability is an important concern.

Using a stream-through approach and implementing a new striping algorithm we designed an architecture that provides a balance between performance and available hardware resources, well suited for iterative implementations of the SMVM required by the CG method. The proposed architecture employs external SDRAM for data storage and due to its stream-through design it does not require initialization phases for data loading and setup as do other implementations [6,7].

2. SMVM-pipeline

The linear processor array architecture, SMVM-pipeline [8], possesses major properties such as, modularity, regularity, data locality, pipelineability and parallelism, all of which could be greatly exploited by the computation of sparse-matrix vector multiplication given that the sparse-matrix is carefully partitioned into special stripe formations. The SMVM-pipeline is shown in Fig. 1.

The SMVM-pipeline computes the overall SMVM operation in parallel by implementing a pipeline of Processing Elements (PEs) each containing a cascade of floating-point arithmetic units made of an adder and a multiplier. The matrix elements are grouped into specially organized stripes and fed to each PE from the top, while the X and Y vectors are fed into the pipeline horizontally from the same side. Therefore, each PE computes a partial summation of:

$$Y(i) \Leftarrow Y_{\text{prev}}(i) + X(j) \times A(i, j), \tag{3}$$

where $Y_{\text{prev}}(i)$ is the partially accumulated summation from a prior computation. The PE multiplies the stripe element $A(i, j)$ with $X(j)$ coming from the right side X stream and adds the result to $Y(i)$ coming from the right side Y stream. The PE then, sends off $X(j)$ and the new value of $Y(i)$ to the left queue streams to be processed by the subsequent PEs, if not required for further computations within the PE. This process is repeated with the next stripe element. The result of the multiplication is accumulated in the vector Y which must be initialized to zero at the start of the operation. To facilitate the PE process, each PE should contain counters, comparators along with a state machine in order to control the flow of data streams within each PE.

3. Striping

FEM matrices display specific sparsity patterns that can be exploited to improve the efficiency of hardware designs. Our approach aims first at organizing the data of the finite element sparse-matrix by taking advantage of its banded sparsity structure and groups it into stripes which will facilitate the use of the

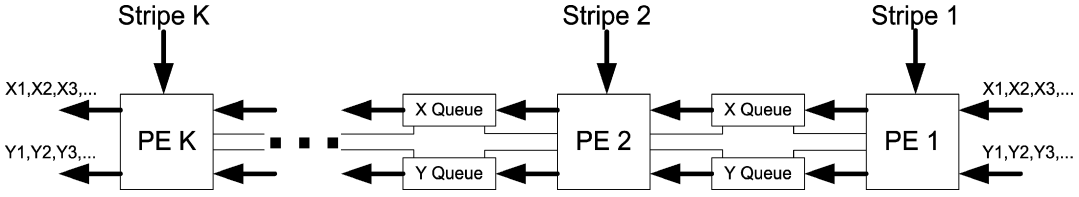

Fig. 1. SMVM-pipeline.

SMVM-pipeline efficiently. Although FEM matrices have been used to demonstrate the effectiveness of the proposed design, it might also provide similar benefits to other methods that generate large sparse systems such as finite differences and spectral differences [9,10].

3.1. Previous work

Different forms of striping schemes have been developed in previous work. These different schemes try to meet either one of two major objectives. Firstly, producing the least number of stripes that cover the sparse matrix. Secondly, to facilitate better utilization of the parallelism features of the SMVM-pipeline.

One of the most basic striping formations is the non-zero diagonals of the sparse-matrix. Straight-diagonal stripes are most suitable for banded sparse-matrices, and the number of stripes formed is closely equal to the bandwidth of the sparse matrix. This stripe formation requires the simplest hardware implementation since they are completely systolic on the PE-pipeline [11]. However, diagonal stripes form the highest number of stripes among the other striping algorithms. Since for most applications the matrix is usually much sparser than what is represented by its diagonals, the straight-diagonal stripes will result in a poor hardware utilization of the SMVM-pipeline.

A scheme that produces the lowest possible number of stripes is presented in [12]. The stripes formed in this method take the shape of staircases. While this stripe formation can be used with the SMVM-pipeline it has the major disadvantage that it will prevent the implementation of a pipelineable floating-point adder unit which greatly impedes the overall clock speed and throughput of the SMVM-pipeline. This is because the staircase stripe can have multiple elements having the same row index value which will create a feedback path around the adder unit.

Melhem introduces a jagged-diagonal striping scheme presented in [13]. At the expense of producing more stripes than the staircase stripe formation, it does not prevent the implementation of pipelined floating-point adders. However, we have found that Melhem's scheme does not produce the least number of pipelineable stripes which indicates that there is a wasted efficiency when used with the SMVM-pipeline. Nevertheless, Melhem's work on adapting the jagged-diagonal stripes for matrices specific to the Finite Element (FE) application area [14] is the cornerstone of our design approach. By using specific numbering techniques, Melhem shows that a FE matrix can be represented with $O(\pi)$ number of jagged-diagonal stripes independent of the matrix size N. This fact allows us to design a scalable architecture that uses a fixed set of PEs to compute on the matrix stripes as the size of the problem grows, without demanding a corresponding increase in hardware requirements.

3.2. Pipelineable stripes

We use our own stripe formation, pipelineable stripes [15], which produces stripes that allow the implementation of pipelined floating-point arithmetic units. In fact, the number of pipelineable stripes, μ, is always upper bounded by the number of jagged-diagonals the same matrix can form. That is:

$$\mu \leqslant \pi, \tag{4}$$

pipelineable stripes are defined by the following definition of strict-rows increasing order stripes.

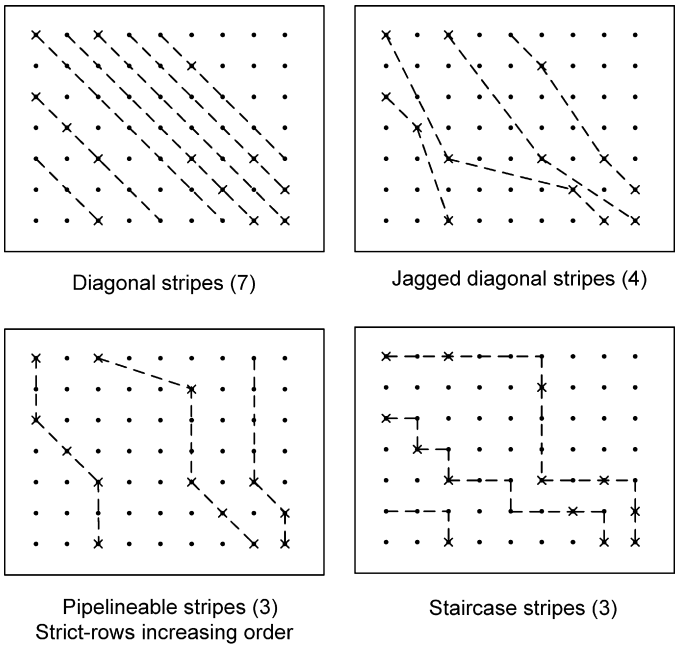

Fig. 2. Stripe formation.

Definition 1. Stripe $S = \{A(i_1, j_1), A(i_2, j_2), A(i_3, j_3), \ldots, A(i_k, j_k)\}$ is said to be *strict-rows increasing*, if the rows and columns of every two adjacent elements $A(i_{k-1}, j_{k-1})$ and $A(i_k, j_k)$ in S are related by $i_{k-1} < i_k$ and $j_{k-1} \leqslant j_k$.

As will be shown later, efficiency of the SMVM-pipeline increases by reducing the number of stripes representing the same matrix. But, it should be noted that scaling is not actually limited by the number of stripes. Further scaling can be achieved by creating simple matrix partitions and implementing multiple SMVM-pipelines for each partition. In fact, using our striping scheme along with our matrix partitioning scheme will yield linear scaling bounded only by available memory bandwidth. Fig. 2 presents a visual comparison of the different striping formations.

Stripes may be padded by inserting zero elements in order to make each stripe contain exactly one element from each row. Padding greatly simplifies the PE design by streamlining the data flow in the SMVM-pipeline. A major advantage of our pipelineable stripes is that they can be systematically padded. This means that it is possible to design a simple two stage pipelined circuit that generates and inserts the pads between the stripe elements. In other words, the pads are generated inside the FPGA and do not need to be stored with the stripes in external memory, which conserves both memory storage and bandwidth.

In order for the PEs in the SMVM-pipeline to process data more concurrently and to minimize the required X-queue sizes,

the PE pipeline should receive stripes ordered in a higher to lower relationship. A stripe higher relationship ($\leqslant_h$) is defined by the follow property:

Definition 2. For stripe $S_1 = \{A(i_k, j_k)\}$ and stripe $S_2 = \{A(l_p, m_p)\}$, S_2 is said to be *higher* than S_1 ($S_2 \geqslant_h S_1$) if $l_p \geqslant i_k \Rightarrow m_p \geqslant j_k, \forall k, p$.

Therefore, we can define a totally ordered property on a group of stripes Ψ as follows:

Definition 3. A stripe formation group $\Psi = \{S_1, S_2, \ldots, S_\mu\}$ is said to be *totally ordered* iff each stripe S_n in Ψ is increasing and every two adjacent stripes S_{n-1}, and S_n are related by $(S_{n-1} \leqslant_h S_n)$.

At this point, the minimum bound on the X-queue size can be defined for totally ordered stripes. This minimum queue size bound should prevent the SMVM-Pipeline from possibly going into a deadlock because of X-queue overflow. To facilitate this, we define the largest horizontal separation (LHS) value between ordered stripes as follows:

Definition 4. For stripe $S_1 = \{A(i_k, j_k)\}$ and stripe $S_2 = \{A(l_p, m_p)\}$, if $(S_1 <_h S_2)$, then the *largest horizontal separation* (LHS) is computed by:

$$\mathrm{LHS} = \min_{i_k \geqslant l_p} (m_p - j_k - 1) \quad \forall k, p. \tag{5}$$

The X-queue depth (X_{QD}) between any two PEs is lower bounded by LHS as follows:

$$X_{\mathrm{QD}}(S_1, S_2) \geqslant \mathrm{LHS}(S_1, S_2) + c \quad \text{when } S_1 \leqslant_h S_2, \tag{6}$$

where c is a small integer constant reflecting the latency response of the queue status signals. For example, if the queue status response requires one clock cycle then $c = 1$.

The pipelineable stripe formation can easily be adapted to generating stripes having an upper-bound on the LHS between them. This can be done, by dissecting the matrix along cut-lines. The cut-lines could either be along columns or along diagonals, but since we are dealing with finite-element matrices that have mostly diagonal sparsity structure, diagonal cut-lines are more appropriate.

We use our striping algorithm to generate pipelineable stripes from existing test matrices for the purpose of validating the SMVM-pipeline design. However, pipelineable stripes should be used as a mean of storing FE matrices required to be used by iterative solvers. In fact, the striping storage format does not require any more memory space or bandwidth than the existing row major, column major, or coordinate formats [13].

4. SMVM-system

A complete system has been implemented to analyze the performance of the SMVM-pipeline and its operations using our striping method. We used the TM4 FPGA development board [2,3] to implement our design. The TM4 contains four Altera Stratix EP1S80 FPGAs each connected to two SDRAM interfaces. The TM4 board also contains a host PC as a PCI add-on card on the TM4 board. We used the TM4 during an early stage of its development, and only one of the FPGA chips was active with only one SDRAM interface, limiting both our computational capacity and memory bandwidth. However, we have implemented our design as parameterized VHDL entities allowing us to easily port it to the full TM4 system or to any other development board.

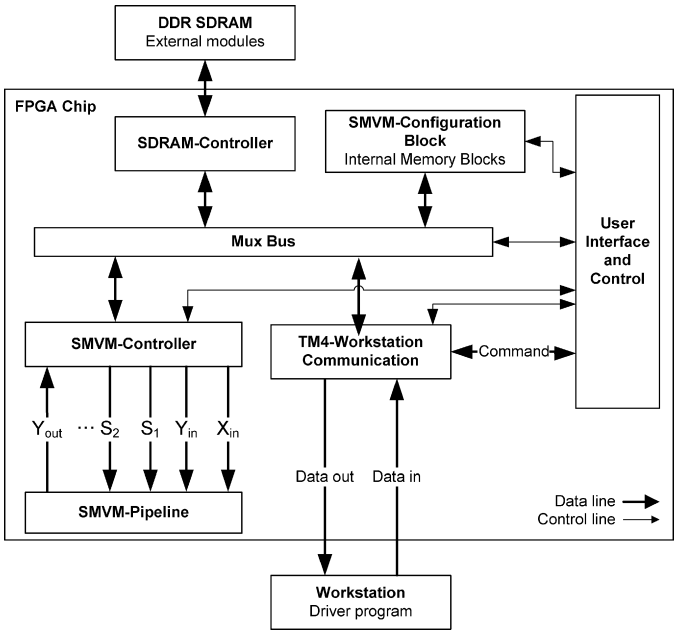

Fig. 3. SMVM system.

Fig. 3 shows the high-level functional building block of the SMVM-system implemented on one of the four TM4 FPGAs. The SMVM-system is implemented using the VHDL hardware description language with the exception of the driver program which is written in C and is executed on the Linux host workstation. The VHDL code is compiled and downloaded to the TM4 using the Altera Quartus software. In this section, we present the underlying design principles used in implementing the SMVM-system components.

4.1. SMVM-pipeline

The SMVM-pipeline is where the SMVM computation is performed. It receives its input from the input queues which in turn are fed data from SDRAM by the SMVM-controller. Fig. 4 shows the configuration of the SMVM-pipeline and its main components which are made of FIFO queues, stripe padding circuits, and the processing elements (PEs).

4.1.1. FIFO queues

The data flow in the SMVM-pipeline is controlled by FIFO queues. Queue components can be implemented using internal dual-ported SRAM memory blocks which are well distributed throughout the Stratix FPGA chip. The queues in the SMVM-system have been generated using the Altera Quartus design software which produces fully parameterized queues

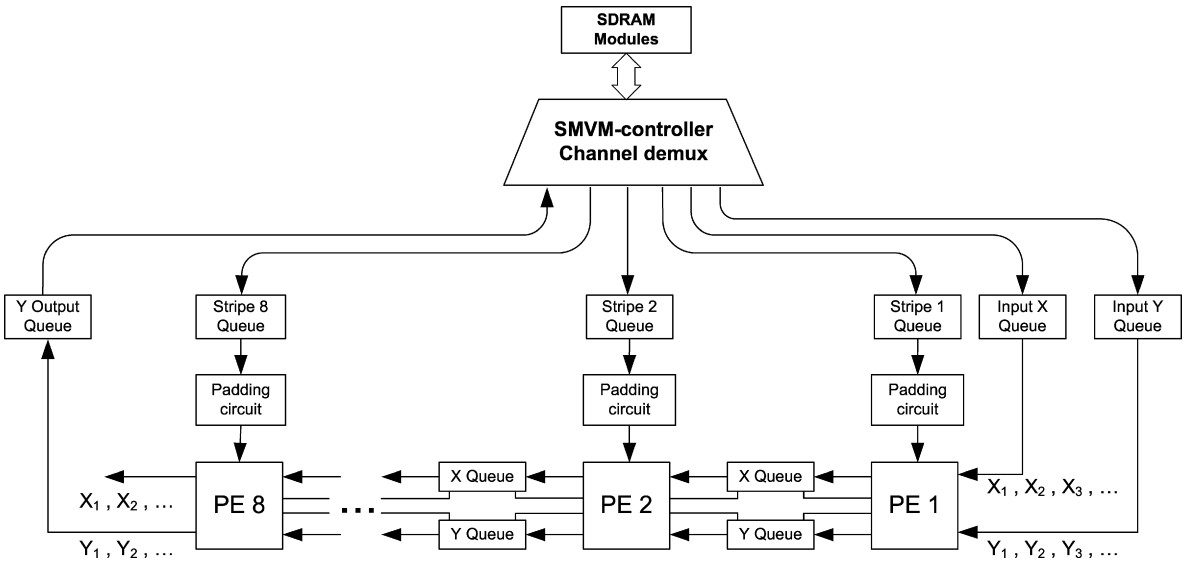

Fig. 4. PE-pipeline.

that are specifically optimized for the Altera device architecture. Queues have a convenient control interface which avoids the complexity of dealing with explicit memory addressing, and hence simplify the design of the PE control.

Two main queue types have been used in the SMVM-system, dual-clock queues and common-clock queues. In dual-clock queues, the input and the output ports can operate on unrelated clocks. Dual-clock queues are used for the input and output queues connected between the SMVM-pipeline and the SMVM-controller. This is to enable the SMVM-controller and the SMVM-pipeline to operate in different clock domains. Also, dual-clock queues are ideal for managing the bursty nature of data flowing in the input and output channels of the SMVM-controller from the external SDRAM modules. Since all the SMVM-pipeline components operate in the same clock domain, common-clock queues can be used to queue the X and Y values in the links between PEs inside the SMVM-pipeline.

4.1.2. Padding circuit

The padding circuit generates the zero pads that may exist between any two stripe elements as specified by the padded pipelineable stripes specification. The padding circuit has a two-stage register pipeline. The circuit also produces an additional signal, *vertical*, which indicates whether the next stripe element index has the same value or not. This signal is used by the PE's control to streamline both the load data and compute operations in order to maximize throughput.

4.1.3. Processing element (PE)

Fig. 5 shows the hardware design of the PE. The PE is designed to compute the partial results of SMVM for fully padded pipelineable stripes. The main components of the PE are a floating-point multiply-add arithmetic unit, a 24-bit counter, an asynchronous 24-bit comparator, and a finite-state-machine (FSM). The multiply-add unit performs the arithmetic operation on each stripe value. The counter and the comparator are used to control the X stream traffic. Finally, the FSM controls the operation and the timing of the PE components.

The floating-point multiply–add unit performs the arithmetic operation. The unit is a cascade of two individual units which are a multiplier and an adder obtained from a library of parameterized floating-point cores from Northeastern University [16]. We modified these units for single precision arithmetic and increased their pipeline depth by one stage in order to obtain a slightly higher clock rate. The three inputs X, A, and *Y_input* should simultaneously be available on the input of the multiply–add unit, therefore the *Y_input* is initially delayed by cascaded registers in order to match the latency of the multiply unit. The multiply–add unit contains a synchronization signal that greatly simplifies the control of data flow. When the correct X, Y and A are at the input of the multiply–add unit, the FSM asserts the *ready* signal, otherwise this signal should be kept de-asserted. The *ready* signal propagates through a cascade of 1-bit registers which finally becomes the *done* signal on the output. These cascaded registers must be equal in depths to the pipeline latency of the multiply–add unit. The *done* signal is connected to the *write_en* signal of the left Y queue which, when asserted, captures the correct output from the multiply–add unit. In order to guarantee the correctness of this configuration, the full status of the right Y queue should not be the completely full indication, rather it should be the almost full indication leaving enough empty words equal to the number of the pipeline stages of the floating-point unit plus one. That is, to give a chance to the multiply–add unit to flush its internal data without losing it due to queue overflow. The additional one comes from the fact that the FSM will streamline the load and compute operations to maximize throughput as will be described later. Also, this configuration provides flexibility by allowing the floating-point units to be easily replaced by more efficient ones that may have a longer pipeline.

The control of the flow and the data association of the three streams of traffic is performed by the FSM. There are no guarantees on the availability of regular data flow for any stream.

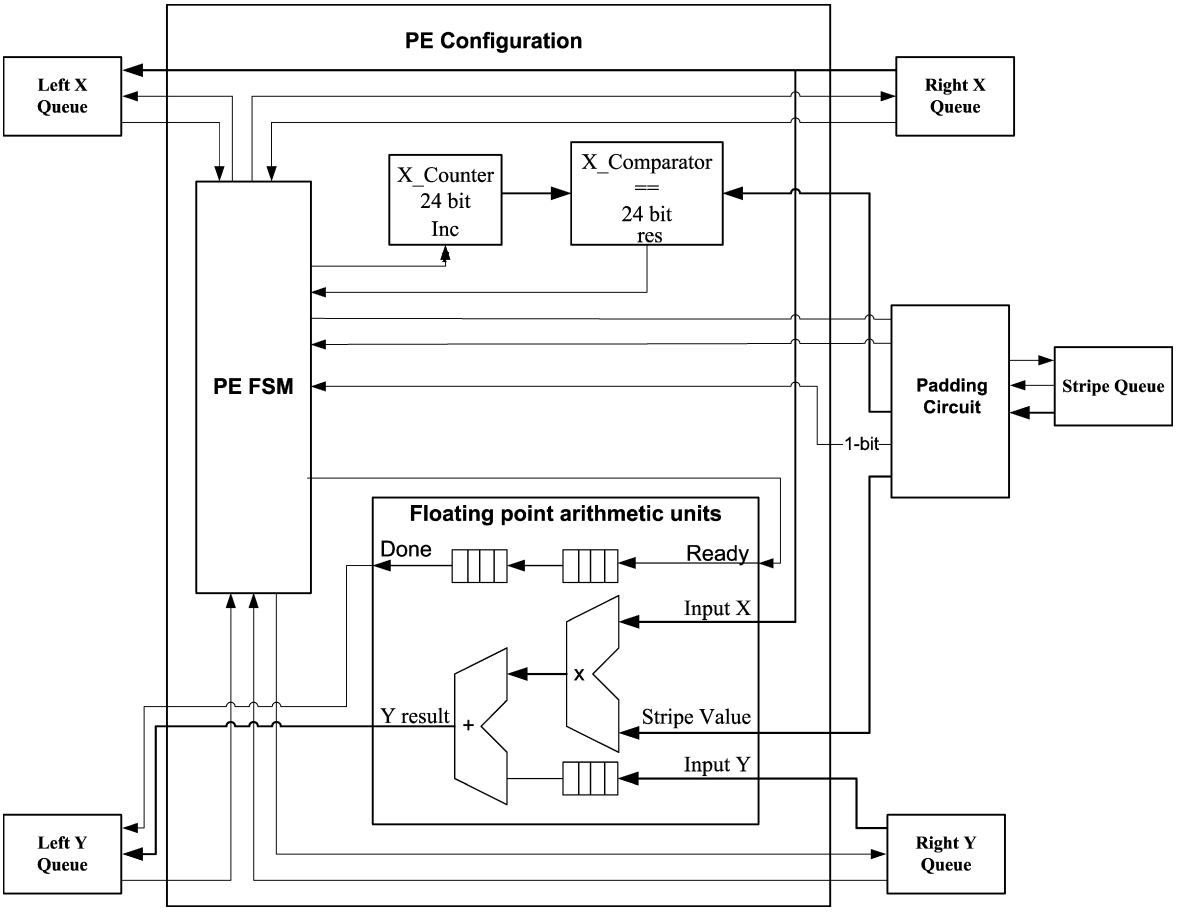

Fig. 5. PE hardware design.

Therefore, we would like to obtain a flexible PE-pipeline design that can adapt to any SDRAM bandwidth available to the SMVM-system. The three streams of traffic are assumed to be fully irregular. The X and Y streams are considered to be flowing only if both the left queue is not empty and the corresponding right queue is not full. The stripe stream is considered flowing if the status of its queue is not empty.

The FSM design will be critical in obtaining a maximum data throughput, that is we need to accomplish the most possible computational throughput. The tasks for input request, index association and computation needs to be streamlined concurrently. Our FSM design accomplishes these objectives. This is also made possible by the use of the *vertical* signal which is produced during stripe padding by the padding circuit. Using this signal, the FSM can make an early decision on whether to the keep the current X value to be used for the next stripe value or dispose of it by obtaining a new X value.

4.2. SMVM-controller

The SMVM-controller controls the SDRAM-controller which is a core component that is provided by the TM4 support package and is designed to provide an easy interface to the DDR SDRAM modules. The controller maximizes the access efficiency of the DDR SDRAM bandwidth by providing block transfers of consecutive data. This form of data transfer is ideal for SMVM applications especially after using our striping formation. Since, the individual stripes along with the X and Y vectors are stored in the form of contiguous data in SDRAM, it is possible to access them in the form of bursts of consecutive blocks forming multiplexed streaming data channels.

There are two main functions of the SMVM-controller, the first is to execute the sequence of operations required to complete the SMVM computation, and the second is to effectively demultiplex the SDRAM bandwidth into 11 channels to keep feeding the SMVM-pipeline with the data it needs for the SMVM computation. The 11 channels are composed of 8 channels of stripes feeding the PEs and another 3 channels which are for the X vector, the Y input vector and lastly for the resulting Y output vector.

At the start of the operation, the SDRAM addresses for the eight stripes for the current phase along with the X and Y addresses are retrieved from the SMVM-configuration block. Next, the PE-pipeline and its internal queues are reset, therefore discarding any data remaining in the queues from a previous phase computation. The reset function will put the FSMs of each PE into an initial state in order to be ready for the new phase computation. The next sequence in the operation cycle is to start the channel demux unit. The channel demux unit works on supplying the different PEs in the PE-pipeline with the required data from the SDRAM modules using the SDRAM controller. Finally, the phase counter is incremented

and a decision is made on whether to start a new phase or to exit when all the phases are completed. This operation cycle is repeated a number of times equal to the number of phases entered in the SMVM-configuration block during the SMVM setup stage. Each phase runs a computation for eight stripes.

An effective demultiplexing of the SDRAM bandwidth into individual channels in order to supply data to the SMVM-pipeline is a critical function. The speed of SMVM computation can be very dependent on the quantity of available SDRAM bandwidth, therefore it is important to utilize the SDRAM bandwidth efficiently in order to obtain good performance results. Another source of difficulty in acquiring an efficient bandwidth utilization comes from the irregularity of data transfers required by each channel. This stems from the sparse nature of the matrix which makes the PEs require stripe data at different rates. In other words, the bandwidth requirements for each individual channel changes dynamically during SMVM computation. This means that the channel demux unit has to allocate SDRAM bandwidth dynamically for each channel as its data drainage rate varies.

Fig. 6 presents a functional view of the channel demux unit. The SRAM buffers of the SDRAM controller are partitioned into segments, where each segment is allocated for each channel. The segment size should be at least equal to the SDRAM transfer burst size which is chosen to be equal to 8 blocks (16×72 bits). Each SDRAM transaction transfers 8 blocks of stripe data into its allotted segment in the SRAM buffer, then once the transaction is completed this data is transferred from the SRAM buffer to the input queue of the PE-pipeline. In order to meet the objective of dynamic bandwidth allocation for each channel, the data transfer function is split into two functions operating in parallel, which are the SMVM-pipeline interface, and the SDRAM transaction interface. The SMVM-pipeline interface is responsible for transferring data for each channel from its segment in the SRAM buffers to the appropriate input queue of the SMVM-pipeline. The SDRAM transaction interface is responsible for initiating the SDRAM transactions to update

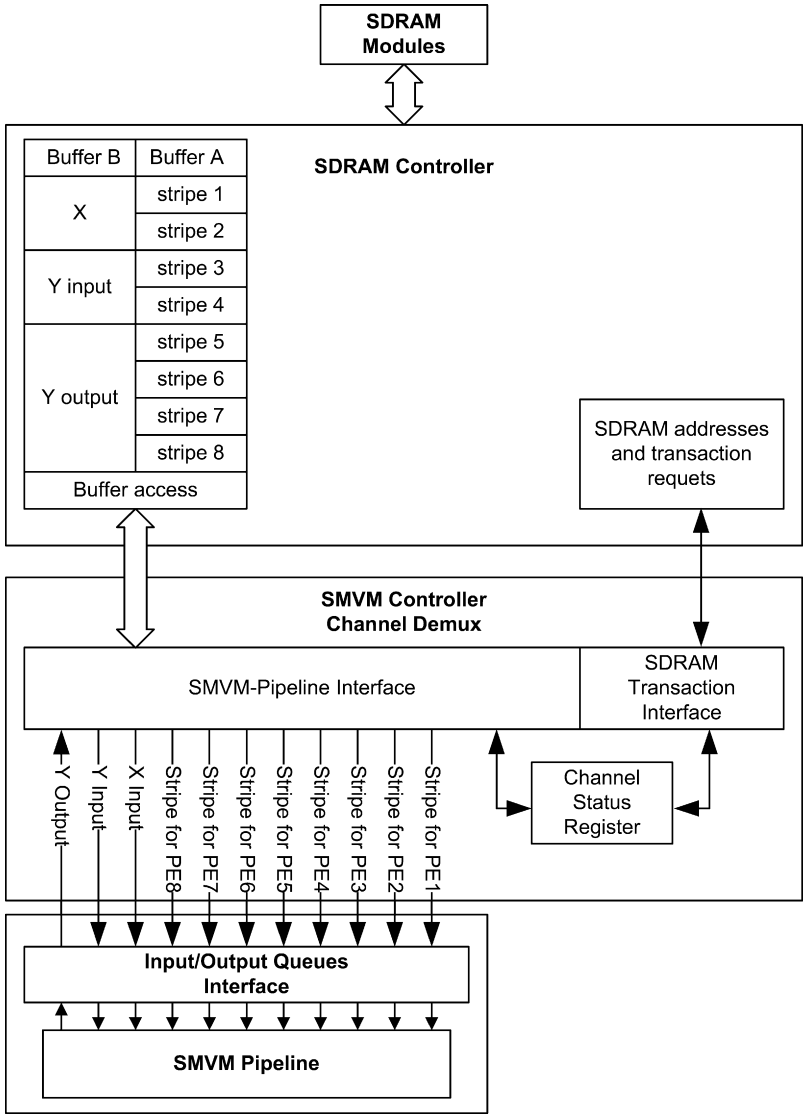

Fig. 6. SMVM-controller channel demux.

Table 1
SMVM-system timing zones clock rates

Clock description	Clock rate (MHz)	Type	Based on
TM4_clk_0	66	TM4 pin	–
TM4_clk_1	66	TM4 pin	–
Zone 1	133	PLL_1 output	TM4_clk_0
Zone 2	66	PLL_2 output	TM4_clk_0
Zone 3	133	PLL_2 output	TM4_clk_0
Zone 4	66–110	PLL_3 output	TM4_clk_1

each segment in the SRAM buffer once its data is used by the SMVM-pipeline interface. In order to identify whether a channel segment requires an update or not, a status bit is maintained for each channel in a channel status register.

Initially, the SDRAM transaction interface updates a segment in the SRAM buffer it sets the flag bit in channel status register corresponding for the updated segment's channel. The SMVM-pipeline interface unloads the SRAM segment into the input queue when two conditions are asserted, the first is when the status bit for that channel is set and the second is when the input queue is not full. When the SMVM-pipeline interface finishes unloading an SRAM segment, it resets the status bit corresponding to the segment's channel. The SDRAM interface does not update a channel's segment unless its status bit is reset.

This parallel operation of loading and unloading of the different segments in the SRAM buffers leads to efficient use of the SDRAM bandwidth when feeding the PE-pipeline channels processing data at varying rates. One last note to make is with regard to the input queues' full status signal. The full status of the input queues is not asserted when the queue is absolutely full rather it is asserted when the number of words in the queue reaches (maximum capacity—burst size). The burst size is 8×16 words for the stripe queues and 8×32 words for the X and Y queues. This is in order to avoid queue overflow and hence loss of data.

4.3. SMVM-system clock domains

The timing of the SMVM-system is divided into four isolated zones. This timing configuration on one hand helps better utilize the place-and-route resources and on another helps in obtaining the highest possible clock rates that produces a high memory bandwidth and a high computational throughput. For example, it is possible to clock the master FSM and the TM4-workstation communication circuits at lower clock rates since, these components do not impact the speed of computation. On the other hand, the SMVM-controller and SDRAM-controller need to be clocked at the highest clock rates since these components impact the memory bandwidth of the overall system. The TM4 provides two independent user programmable clock sources that are used to derive the different timing zones of the SMVM-system. Table 1 shows the clocking dependencies of the SMVM-system timing zones.

Transferring data and control signals between different clock zones can incur metastability which significantly corrupts data and cause malfunctions. In order to eliminate metastability effects on clock zone crossings, we adopted the Altera design guidelines for transferring data between clock domains. This involves the utilization of dual-clock FIFOs, handshaking circuitry, and cascaded registers.

Table 2
SMVM-system resource utilization of a single Stratix EP1S80

Resource	Utilized	Utilization
Logic elements	23,887 of 79,040	30%
Memory bits	3,072,253 of 7,427,520	41%
DSP block 9-bit elements	64 of 176	36%
PLLs	3 of 12	25%
DLLs	1 of 2	50%

4.4. FPGA resource utilization

Table 2 shows the post place-and-route resource utilization of an SMVM-system implemented on a single Stratix EP1S80 FPGA on the TM4. The M4K internal SRAM block has been utilized by the pipeline's X and Y queues which are more suitable than the larger MRAM blocks due to a single clock cycle latency as opposed to the two clock latency of the MRAM block. Our architecture does not demand a large amount of resources either in terms of computational logic or internal memory blocks. As we will demonstrate later in Section 6, the limited FPGA resource utilization of 30% logic and 40% internal memory is adequate to support a wide range of FEM matrix sizes, including very large matrices, since the number of stripes in the FEM matrix is independent of its dimension N (size) [14]. In other words, the FEM matrix size is only limited by SDRAM memory and not FPGA resources, given that it is striped using our striping scheme.

5. Performance

Due to the regularity of data flow in the SMVM-pipeline architecture, its performance can be easily formulated and projected for further scalability analysis. First, we start by defining a utilization factor $\mathcal{U} \in [0, 1]$ for a given matrix which indicates how much of the peak SMVM-pipeline performance can be utilized. In general $\mathcal{U}$ can be computed using the following formula:

$$\mathcal{U} = \frac{T_{\text{comp}}}{T_{\text{total}}}, \tag{7}$$

where T_{comp} is the number of clock cycles the SMVM-pipeline is performing useful computation involving non-zero elements only and T_{total} is the total number of clock cycles required by the SMVM-pipeline to complete the computation. If the SMVM-pipeline is performing a useful computation each clock cycle, as in the case of complete stripes where each stripe contains a non-zero element on each row, the utilization in this case would be approximately 100%. Therefore, the utilization for the SMVM-pipeline depends on the sparsity structure and the striping scheme used for a given matrix. If the number of PEs that can be implemented in the SMVM-pipeline for a certain FPGA

chip is less than the number of stripes the matrix may have, the computation needs to be performed in multiple phases where each phase processes as many stripes as there are available PEs. The Y vector is initialized to zero for the first phase only. Each phase will sequentially accumulate a partial result on the Y vector. When the last phase is done, the Y vector will have the final SMVM result. We define a set of computational phases ($\mathcal{P}$) as follows:

$$\mathcal{P} = \left\{1, 2, \ldots, \left\lceil \frac{\mu}{\mathrm{PE}_n} \right\rceil \right\},$$

where PE_n is the number of PEs in the SMVM-pipeline. We define $\mathcal{P}_{\max}$ as the total number of required phases and is equal to $\lceil \mu/\mathrm{PE}_n \rceil$. $\mathrm{LHS}_{\mathcal{P}}$ is the LHS between stripes within each computational phase, and the maximum LHS for all phases is found by:

$$\mathrm{LHS}_{\max} = \max\{\mathrm{LHS}_{\mathcal{P}}\}, \quad \forall_{\mathcal{P}}.$$

Now, the total number of clock cycles required to finish a computation is determined by:

$$\begin{aligned} T_{\mathrm{total}} &= \sum_{\mathcal{P}} N + \mathrm{LHS}_{\mathcal{P}} \\ &\leqslant \mathcal{P}_{\max} \times (N + \mathrm{LHS}_{\max}). \end{aligned} \tag{8}$$

The average number of computational clock cycles for each phase is found by:

$$T_{\mathrm{comp}} = \mathcal{P}_{\max} \times \mathcal{D}, \tag{9}$$

where $\mathcal{D}$ is the stripe density which is found by:

$$\mathcal{D} = \frac{\text{total number of non-zeros}}{\mathrm{PE}_n \times \mathcal{P}_{\max}}. \tag{10}$$

In other words, $\mathcal{D}$ describes the average number of non-zero elements per stripe. Substituting (8) and (9) in (7) we obtain:

$$\mathcal{U} \geqslant \frac{\mathcal{D}}{N + \mathrm{LHS}_{\max}}. \tag{11}$$

Considering that LHS is limited for FEM matrices [14] and in many cases is small compared to N, $\mathrm{LHS}_{\max}$ can be ignored. We can see from (10) and (11) that the utilization increases as the stripe density increases which is done by reducing the number of stripes representing the same matrix. This optimization is done by virtue of our special striping scheme which is found to produce a very low number of pipelineable stripes for FEM matrices.

The computational performance of the SMVM-pipeline measured in MFLOPS for the unlimited memory bandwidth condition can be found by:

$$\mathrm{MFLOPS}_{\mathrm{comp}} = \mathcal{U} \times \mathrm{MFLOPS}_{\mathrm{peak}}, \tag{12}$$

where $\mathrm{MFLOPS}_{\mathrm{peak}}$ is the maximum computational rate of the SMVM-pipeline. Since each stripe value requires 2 floating-point operations to process, $\mathrm{MFLOPS}_{\mathrm{peak}}$ is determined by:

$$\mathrm{MFLOPS}_{\mathrm{peak}} = 2 \times \mathrm{PE}_n \times (\text{clock rate in MHz}).$$

In order to find the actual MFLOPS performance of the SMVM-pipeline we need to consider the amount of memory bandwidth available for the SMVM-pipeline. Since our design of the SMVM-controller effectively distributes the available SDRAM bandwidth between all the SMVM-pipeline ports dynamically, we can formulate each port throughput as follows:

$$\mathrm{BW} = 3 \times P_v + \mathrm{PE}_n \times \mathcal{U} \times P_s, \tag{13}$$

where BW is the total available SDRAM bandwidth, P_v is the port throughput for each of the X, Y_{input} and Y_{output} vectors, and P_s is the port throughput for each stripe. Since we are using 32-bit IEEE single precision floating-point arithmetic and our SDRAM modules have a word width of 72 bits, each SDRAM word can store a single stripe value along with its index or two X, Y vector values. Therefore, the relationship between P_v and P_s can be written as:

$$P_s = 2P_v. \tag{14}$$

If a different memory organization or double precision arithmetic is used, relationship (14) should be modified. After substituting (14) into (13) and rearranging we obtain:

$$P_v = \frac{\mathrm{BW}}{(3 + 2 \times \mathrm{PE}_n \times \mathcal{U})}. \tag{15}$$

When BW is measured in 32-bit words per second, P_v becomes the available bandwidth for the output result that can be produced by the SMVM-pipeline when the computational capacity is unlimited. Therefore, the performance of the SMVM-pipeline under limited memory bandwidth is found by:

$$\mathrm{MFLOPS}_{\mathrm{BW}} = 2 \times \mathrm{PE}_n \times P_v \times \mathcal{U}. \tag{16}$$

Therefore, the actual performance of the SMVM-pipeline $\mathrm{MFLOPS}_{\mathrm{smvm}}$ can be bounded by either computation or memory bandwidth as follows:

$$\mathrm{MFLOPS}_{\mathrm{smvm}} = \min(\mathrm{MFLOPS}_{\mathrm{BW}}, \mathrm{MFLOPS}_{\mathrm{comp}}). \tag{17}$$

It should be noted that (17) should be applied on each computational phase separately in order to get an accurate performance measurement. That is because the actual utilization varies, which may drive the performance to be either computation or bandwidth bound within each phase. However, for simplicity of calculation, an overall average utilization value on all phases can be used (17) which will produce a close approximation of the performance.

6. Results

We have selected FE test matrices from the Matrix Market website [17] that represent a range of utilization factors from 20% to 80%. The matrix selection corresponds to the fact that our architecture focuses on accelerating regularly banded sparse matrices that arise when using FEM in electromagnetic problems. The test matrices are listed in Table 3 and ordered according to increasing utilization factors. We striped the matrices using our striping algorithm which generates banded pipelineable stripes. The processing time for striping did not exceed 2 minutes for the largest matrix. The striping needs to be done only once at the beginning of the CG algorithm. However,

Table 3
Matrix market FE test matrices

Matrix number	Matrix name	Matrix order	Non-zeros count	Banded stripes	Maximum LHS	Number of partitions	Effective utilization
1	dwt_1007	1007	8575	24	1972	1	17.74%
2	fidapm13	3549	70994	64	450	1	27.74%
3	fidap011	16614	1.09E+06	328	9678	12	29.06%
4	fidapm10	3046	53344	48	278	1	33.35%
5	cavity11	2597	71601	64	504	1	36.08%
6	fidap013	2568	75628	72	330	1	36.25%
7	fidap010	2410	54816	56	226	1	37.13%
8	cavity10	2597	76171	64	504	1	38.38%
9	e20r0000	4241	131412	72	462	1	38.81%
10	e20r5000	4241	131430	72	462	1	38.81%
11	cavity17	4562	131735	64	654	1	39.46%
12	fidapm09	4683	93836	48	234	1	39.71%
13	cavity16	4562	137887	64	654	1	41.31%
14	fidapm15	9287	96099	24	244	1	42.01%
15	e30r0000	9661	305794	64	682	1	46.20%
16	e30r5000	9661	306002	64	682	1	46.23%
17	fidapm29	13668	183394	32	566	4	46.51%
18	e40r0000	17281	553216	64	902	1	47.54%
19	e40r5000	17281	553562	64	902	1	47.57%
20	fidap009	3363	99397	56	170	1	50.24%
21	fidap029	2870	23754	16	84	1	50.26%
22	mhd3200a	3200	68026	40	58	1	52.20%
23	mhd4800a	4800	102355	40	58	1	52.62%
24	mhd1280a	1280	48061	64	116	1	53.62%
25	fidap015	6867	96421	24	134	1	57.39%
26	s1rmt3m1	5489	217651	48	382	1	77.23%
27	s3rmt3m1	5489	217669	48	382	1	77.24%
28	s2rmt3m1	5489	217681	48	382	1	77.24%
29	s1rmq4m1	5489	262411	56	382	1	79.81%
30	s3rmq4m1	5489	262943	56	382	1	79.98%
31	s2rmq4m1	5489	263351	56	382	1	80.10%
32	s3dkq4m2	90449	4.43E+06	56	1228	1	86.24%

stripes can also be used as a native storage format which is generated by the FE application avoiding the initial latency due to conversion to the stripe format before SMVM computation. The memory storage requirements for stripes do not exceed the coordinate format used by the Matrix Market website nor the popular CSR representation (used by other architectures such as [6,7]).

The implemented SMVM-pipeline contains 8 PEs and is clocked at 110 MHz obtaining a peak performance of 1.76 GFLOPS. However, the SMVM-pipeline performance was found to be limited by SDRAM bandwidth. This is due to the fact that we used the TM4 board at an early stage of its development which supported only one SDRAM interface and a single FPGA. Because of the reduced number of memory ports available for the SMVM-pipeline, the effective memory bandwidth supplied to the SMVM-pipeline became approximately 0.55 GB/s. Future upgrades will focus on resolving this issue and allow the SMVM-pipeline to use more of the TM4 potential when completed, which is capable of supplying 16 GB/s of SDRAM bandwidth for four FPGAs [2]. Other recent development systems provide 8–12 GB/s [4,7].

Fig. 7 shows the MFLOPS results for the test matrices along with expected average results modeled by (17). Matrix 1 showed higher performance compared to the expected results due to its low utilization value relative to its size which made it less bounded by bandwidth compared to other matrices. The projected performance is shown in the graph as the memory bandwidth varies from 0.55 to 8 GB/s. This figure also shows the significant impact of the bandwidth on performance, especially for matrices with high utilization factors. One can see that a 14.5 times gain in performance is achieved when bandwidth varies from 0.55 GB/s to 8 GB/s for a utilization factor

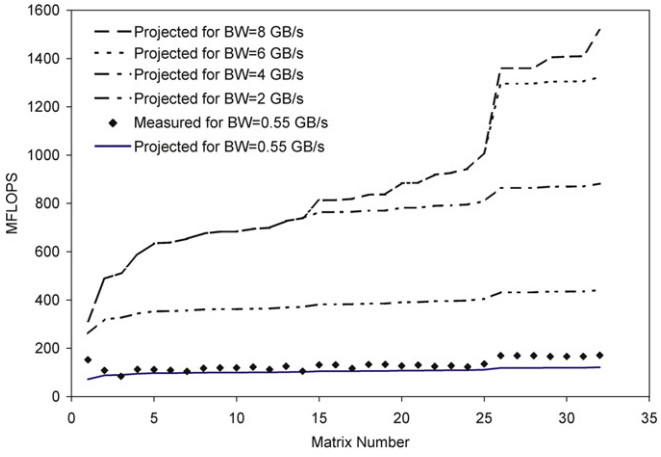

Fig. 7. MFLOPS performance as the memory bandwidth varies.

Table 4
SMVM performance comparison

Processor	MHz	Peak MFLOPS	SMVM MFLOPS	Utilization	Ref.
Pentium 4	1500	3000	425	14.29%	[7]
Power 4	1300	5200	805	16.67%	[7]
Sun Ultra 3	900	1800	108	6.25%	[7]
Itanium	800	3200	345	10.00%	[7]
Itanium 2	900	3600	1200	33.33%	[7]
V2 6000-4	140	2240	1500	66.67%	[7]
V2-Pro 70	165	2880	576–2160	20–75%	[6]
Stratix S80 [this work]	110	1760	312–1520	17.74–86.4%	–

of 86.24%. This performance increase is in the same proportion of the change in bandwidth. It is also possible to observe that BW dominates the performance of our system over the matrix utilization factor that only achieves a maximum of 4.8 times performance enhancement when varying the utilization factor from 17.74% to 86.24%. Note that Matrix 32 is the largest which contains $\cong 4.4 \times 10^6$ non-zeros and has efficiency on the SMVM-pipeline of 86.4%. For large FE matrices the peak performance is reached as the memory bottleneck is resolved.

Next we compare our performance results to other computing platforms in Table 4. The MFLOPS performance of our 8-PE design under the assumption of 8 GB/s memory bandwidth ranges from 312 MFLOPS to 1520 MFLOPS, depending on the utilization factor. It is key to note that the utilization depends strongly on the choice of mesh numbering technique [14], and that these matrices were not mesh numbered taking the utilization on our pipeline under our striping scheme into account (they were chosen from a readily-available open-source matrix collection for convenience). Results in [14] indicate that appropriate mesh numbering taking the stripe structure into account, produces a limited number of stripes for larger sparse matrices, increasing the overall utilization as discussed in Section 5. These performance measurements show that our design achieves 1520 MFLOPS (reaching almost its peak performance) which offers similar computation capacity to an 8 leaf tree of [6] and 8 PEs of [7] (the same number of multiplications and additions for [7] while [6] does more additions). These other FPGA implementation achieve 1500 MFLOPS (for [7]) and 2160 MFLOPS (for [6] with a bandwidth of 14.4 GB/s) while all FPGA designs manage better sustained throughput outperforming the best general purpose CPU implementation as shown in Table 4.

Although our design has the second best performance its resource usage scales much better than [6] and [7]. This is due to the stream-through architecture implemented which enables it to operate on arbitrarily big matrices, being limited only by the external SDRAM (available in big quantities when compared to internal memory). Moreover, our design uses a fixed amount of internal RAM per PE to store the input/output data in a cache manner, scaling much better than the other FPGA designs. In fact, when using 8 PEs our architecture consumes only 41% (3,072,253 of 7,427,520) of the Block Ram and 30% (23,887 of 79,040 LUTs) of logic resources available which offers sufficient space to increase the pipeline. On the other hand, [6] and [7] are limited by the maximum size of the internal memory available. This is an important issue since this internal memory is found in limited quantities on current FPGAs and it is a highly used resource. The design in [6] uses internal memory to store the X vector in each leaf node of its dot-product tree architecture (which does not scale well) while the non-zero elements of the A matrix are fetched from external memory. On the other hand, [7] uses internal memory to store the A matrix and the X vector as well as the buffers it requires for its double ring communication scheme, which imposes even greater scalability constrains in this design. This is the reason why both designs [6,7] resort to multiple FPGA implementations when dealing with bigger matrices.

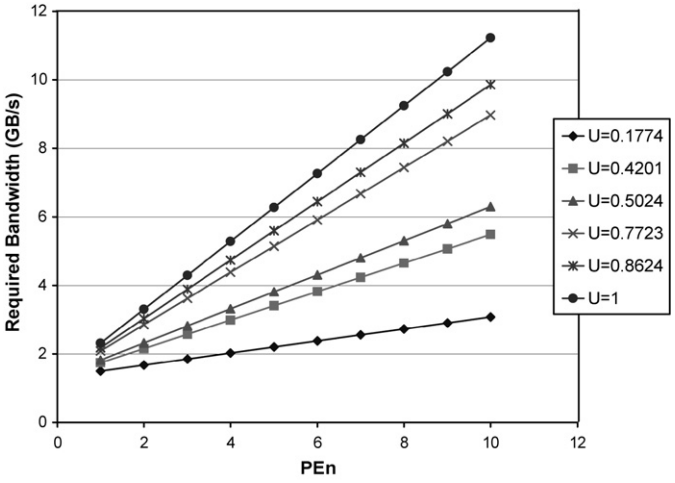

Fig. 8. Bandwidth scaling when varying the number of PE for different Utilization Factors (U).

As we will show in the next section, the peak and achievable MFLOPS can be linearly scaled by partitioning the matrix and using multiple pipelines. This is made possible by the low hardware requirements of a single pipeline.

7. Scaling

Scaling can be performed in either two ways, one is increasing the number of PEs in the pipeline until equal to the number of stripes the matrix forms, and the other is horizontally partitioning the matrix and assigning each partition to a SMVM-pipeline. The choice of either depends on the hardware configuration of the development board in terms of number of FPGAs and memory bandwidth available to each FPGA. Fig. 8 shows the effect on the required bandwidth as we increase the number of PE in the SMVM for a given Utilization Factor on one FPGA. This is especially true when the Utilization Factor is high, so in order to keep streaming data through the SMVM-pipeline higher bandwidths are required. Hence, to exploit the fine-grained parallelism it is necessary to solve the bandwidth limitations. In general, tradeoffs must be met to use a number of PE per FPGA so the required BW does not exceed the available BW, then the peak performance may be reached. On the other hand, Fig. 9 shows a partitioning scheme for 3 pipelines reducing the total time to at most a third of the initial time. In other words, scaling provides linear speedups. This is a direct

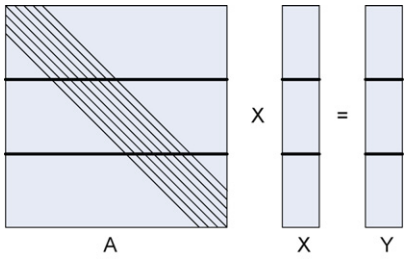

Fig. 9. Matrix partitioning.

Table 5
Matrix partitioning effect on utilization

Matrix name	Matrix order	Non-zeros count	Single partition utilization	Number of partitions	Partitioned utilization
fidap011	16614	1.09E06	12.66%	12	29.06%
fidapm29	13668	183394	40.26%	4	46.51%

result of the regularity of our architecture that uses local interconnect between the PEs in the pipeline which does not produce communication overhead.

The stripe formation is not impacted by the horizontal matrix partitioning. In fact, the utilization of the SMVM-pipeline may increase due to partitioning because the number of stripes within each partition will either be equal or less than the number of stripes for the whole matrix. Therefore utilization will increase as long as the separation between stripes within a phase is limited compared to the number of rows partitioned. A typical partition for FE matrices should contain at least 1000 to 2000 rows. Table 5 shows the partitioning effect on utilization factors for two large matrices.

Finally, it is interesting to observe that the architecture in [7] incurs limitations when resorting to scaling both in resource utilization and in communication issues (due to its double ring design). On the other hand, [6] uses a similar row-block partitioning scheme in order to scale the design to compute on large matrices but it does not solve its Block-Ram issue.

8. Future work

Our current SMVM-system prototype will be upgraded in order to utilize the four FPGAs on the TM4 when they become available. It was shown that our architecture scales efficiently given there is available FPGA resource as well as SDRAM bandwidth. This is made possible by virtue of the local interconnects between PEs inherent to the linear array processor architecture which eliminates communication overhead typical of multiprocessing platforms.

The clock speed of the SMVM-pipeline will also be increased in future upgrades from its current value of 110 MHz. This can be performed mainly in two ways. Firstly, we can replace our floating-point units with ones that are more deeply pipelined and have higher clock speeds. Secondly, we can perform custom place-and-route of the SMVM-pipeline on the FPGA in order to produce better timing results. This is in order to exploit the modularity and the locality of interconnects in the SMVM-pipeline.

9. Conclusion

In this paper we analyzed the acceleration of SMVM computation on FPGAs for regularly banded sparse matrices that arise when using FEM in electromagnetic problems, and demonstrated an SMVM-pipeline architecture that can obtain high performance for very large sparse FEM matrices. Using our striping scheme and due to the sparsity pattern of FEM matrices we were able to compute on arbitrarily big matrices with a fixed number of PEs, limited only by the amount of external SDRAM memory available to the FPGA. Moreover, because of our low resource usage we are able to increase the size of our pipelined processing elements, hence our throughput, when we are not I/O bound. We also show that as a result of the regularity and the modular nature of this architecture, the SMVM-pipeline produces linear speedups when scaled by implementing multiple SMVM-pipelines. Hence, our design achieves a balanced goal between size scalability, regularity and performance. Finally, our stream-through type architecture provides the added advantage of enabling an iterative implementation of the SMVM computation for the CG method by avoiding initialization time due to data loading and setup inside the FPGA internal memory.

Acknowledgement

The authors would like to thank the TM4 team at the University of Toronto, especially Joshua Fender, David Galloway, and Prof. Jonathan Rose, for their guidance and support with the TM4 system. We would also like to express our appreciation to the reviewers for their useful comments and suggestions which have helped to enhance the quality of our work.

References

[1] K.D. Underwood, K.S. Hemmert, Closing the gap: CPU and FPGA trends in sustainable floating-point BLAS performance, in: FCCM'04: Proceedings of the 12th Annual IEEE Symposium on Field Programmable Custom Computing Machines, IEEE Computer Society, Washington, DC, USA, 2004, pp. 219–228.

[2] J. Fender, An FPGA-based hardware development system with multi-gigabyte memory capacity and high bandwidth, Master's thesis, University of Toronto, January 2005.

[3] The Transmogrifier-4 project, http://www.eecg.utoronto.ca/~tm4/, University of Toronto, 2005.

[4] BEE2: A multi-purpose computing platform, http://bwrc.eecs.berkeley.edu/Research/BEE/BEE2/, Berkeley Wireless Research Center, University of California, 2004.

[5] V.E. Taylor, A. Ranade, D.G. Messerschmitt, SPAR: a new architecture for large finite element computations, IEEE Transactions on Computers 44 (4) (1995) 531–545.

[6] L. Zhuo, V.K. Prasanna, Sparse matrix–vector multiplication on FPGAs, in: FPGA'05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays, ACM Press, New York, NY, USA, 2005, pp. 63–74.

[7] M. DeLorimier, A. DeHon, Floating-point sparse matrix–vector multiply for FPGAs, in: FPGA'05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays, ACM Press, New York, NY, USA, 2005, pp. 75–85.

[8] Y. El-Kurdi, W.J. Gross, D. Giannacopoulos, Sparse matrix–vector multiplication for finite element method matrices on FPGAs, in: FCCM'06:

14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, Napa, CA, 2006, pp. 293–294.
[9] D.K. Jordan, D.A. Mazziotti, Spectral differences in real-space electronic structure calculations, Journal of Chemical Physics 120 (2) (2004) 574–578.
[10] D.K. Jordan, D.A. Mazziotti, Comparison of two genres for linear scaling in density functional theory: Purification and density matrix minimization methods, Journal of Chemical Physics 122 (2005) 084114.
[11] S.Y. Kung, VLSI Array Processors, Information and System Science Series, Prentice-Hall, Inc., Englewood Cliffs, NJ, 1988.
[12] L.S. Heath, S.V. Pemmaraju, C.J. Ribbens, Processor-efficient sparse matrix–vector multiplication, Computers and Mathematics with Applications 48 (2004) 589–608.
[13] R. Melhem, Parallel solution of linear systems with striped sparse matrices, Parallel Computing 6 (2) (1988) 165–184.
[14] R. Melhem, Determination of stripe structure for finite element matrices, SIAM Journal on Numerical Analysis 24 (6) (1987) 1419–1433.
[15] Y. El-Kurdi, D. Giannacopoulos, W.J. Gross, Hardware acceleration for finite-element electromagnetics: Efficient sparse matrix floating-point computations with FPGAs, IEEE Transactions on Magnetics 43 (4) (2007) 1525–1528.
[16] P. Belanovic, M. Leeser, A library of parameterized floating point modules and their use, in: Proceedings of the International Conference on Field-Programmable Logic and Applications, 2002, pp. 657–666.
[17] Matrix market, http://math.nist.gov/MatrixMarket/, maintained by: National Institute of Standards and Technology (NIST), June 2004.