IEEE International Symposium on Field-Programmable Custom Computing Machines

A Sparse Matrix Personality for the Convey HC-1

Krishna K. Nagar
Dept. of Computer Science and Engineering
University of South Carolina
Columbia, SC USA
nagar@email.sc.edu

Jason D. Bakos
Dept. of Computer Science and Engineering
University of South Carolina
Columbia, SC USA
jbakos@cse.sc.edu

Abstract— **In this paper we describe a double precision floating point sparse matrix-vector multiplier (SpMV) and its performance as implemented on a Convey HC-1 reconfigurable computer. The primary contributions of this work are a novel streaming reduction architecture for floating point accumulation, a novel on-chip cache optimized for streaming compressed sparse row (CSR) matrices, and end-to-end integration with the HC-1's system, programming model, and runtime environment. The design is composed of 32 parallel processing elements, each connected to the HC-1's coprocessor memory and each containing a streaming multiply-accumulator and local vector cache. When used on the HC-1, each PE has a peak throughput of 300 double precision MFLOP/s, giving a total peak throughput of 9.6 GFLOPS/s. For our test matrices, we demonstrate up to 40% of the peak performance and compare these results with results obtained using the CUSparse library on an NVIDIA Tesla S1070 GPU. In most cases our implementation exceeds the performance of the GPU.**

Keywords-floating point accumulation; reduction; reconfigurable computing; sparse matrix; SpMV

I. Introduction

Sparse Matrix Vector Multiplication (SpMV) describes solving $\boldsymbol{y} = \boldsymbol{Ax}$ where $\boldsymbol{y}$ and $\boldsymbol{x}$ are vectors and $\boldsymbol{A}$ is a large matrix populated mostly with zeroes. SpMV is frequently employed in scientific and engineering applications and is the kernel for iterative linear system solvers such as the conjugant gradient method [1].

Due to the sparseness of the matrix, it is often not practical to store every entry of the matrix in a traditional dense representation, so compressed sparse representations such as compressed sparse row (CSR) format are often used to represent the matrices [2]. The CSR format stores the non-zero elements in an array called **val**, the corresponding column numbers of an array called **col**, and the array indices of the first entry of each row in an array called **ptr**. **ptr** is terminated with the total number of non-zero entries.

For example, the second non-zero value in row 4 could be referenced as **val[ptr[4]+1]** and its corresponding column number in row 4 could be referenced as **col[ptr[4]+1]**. If the matrix has ***M*** rows and the array indices begin at 0, then **ptr[*M*]** stores the total number of non-zeros in the matrix. Multiplying a CSR matrix by a vector stored in an array called **vec** requires a row-wise multiply accumulate (MAC) operation for each matrix row:

sum = sum + val[*i*] x **vec[col[*i*]]**, where ***i*** iterates for each non-zero entry of the matrix.

As shown in these examples, CSR computations fundamentally require indirect addressing, which cannot be expressed in an affine loop and therefore are difficult to automatically optimize for SIMD and vector processors. In addition, SpMV architectures need only to perform two floating-point operations for each matrix value, yielding a computation/communication ratio of at best only two FLOPs per 12 bytes read (assuming a 64-bit value and a 32-bit column number) and this doesn't include references to input vector data. As such, performance is highly dependent on memory bandwidth. Since CSR data is stored sequentially, consecutive values can be read using overlapping outstanding requests from consecutive addresses, making it easy to maximize effective bandwidth. However, CSR stores values in consecutive memory locations in row-major order, so a third challenge for achieving high performance for SpMV comes from the need to accumulate values that are delivered in consecutive clock cycles into a deeply pipelined floating-point adder. This is a design challenge because subsequent additions on incoming values cannot be performed until the previous addition has completed. In order to overcome this hazard, static data scheduling or dynamic architectural methods must be employed.

As a result of these challenges, previous implementations of SpMV, both in special-purpose hardware and software, often suffer from low hardware utilization and developing new SpMV implementations remains an important area of study.

In this paper, we present an SpMV architecture based on our own novel streaming reduction circuit and specialized cache optimized for CSR data. In order to characterize our approach, we implemented this architecture on the Convey HC-1, a self-contained heterogeneous system containing a Xeon-based host and an FPGA-based co-processor board with four user programmable Virtex5-LX330 FPGAs. We compare the performance of our SpMV with the NVIDIA CUDA CUSPARSE library implementation running on an NVIDIA Tesla-S1070 GPU.

II. Previous Work

There has been much prior work in designing efficient FPGA-based SpMV architectures. The most novel aspect of individual SpMV implementations is often the approach taken in designing the floating-point accumulator.

978-0-7695-4301-7/11 $26.00 © 2011 IEEE
DOI 10.1109/FCCM.2011.60

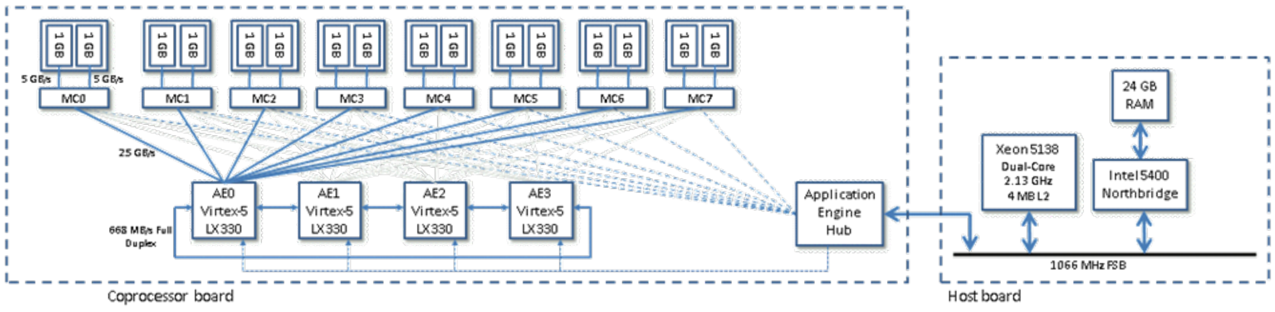

Figure 1. The HC-1 coprocessor board. Four application engines connect to eight memory controllers through a full crossbar.

Historically, there have been two basic approaches for designing high-performance double precision accumulators. The first approach is to statically schedule the input data in order to interleave values and partial sums from different rows, such that consecutive values belonging to each row are delivered to the accumulator--which is designed as a simple feedback adder--at a period corresponding to the pipeline latency of the adder. This still allows the adder to accept a new value every clock cycle while avoiding the accumulation data hazard among values in the same accumulation set (matrix row). Unfortunately, this method requires a large up-front cost in scheduling input data and is not practical for large data sets.

An early example of this approach was the work of deLorimier and DeHon [3]. Their scheduling technique leads to the architecture's performance being highly dependent on the structure of the matrix, although on average they were able to achieve 66% of the peak performance in their simulation-based studies.

The second approach is to use a dynamic reduction technique that dynamically selects each input or partial sum to send into the adder--dynamically managing the progress of each active accumulation set using a controller (i.e. dynamically scheduling the inputs). For the latter case, these approaches can be divided into two types depending on whether they use a single adder or multiple adders.

An early example using the dynamic reduction technique was from Prasanna's group at the University of Southern California [4]. In their earliest work, they used a linear array of adders to create a flattened binary adder tree where each adder in the array was utilized at half the rate of the previous adder in the array. This required multiple adders with exponentially decreasing utilization, had a fixed maximum set size, and required stalls between matrix rows.

A similar implementation from UT-Knoxville and Oak Ridge National Laboratory used a similar approach but with a parallel—as opposed to a linear--array of n adders, where n was the adder depth [5]. This implementation striped each consecutive input across each adder in turn, achieving a fixed utilization of $1/n$ for each adder.

Prasanna's group later developed two improved reduction circuits, called the double- and single-strided adders (DSA, SSA), that solved many of the problems of earlier accumulator design [6]. These new architectures required only two and one adders, respectively. In addition, they did not limit the maximum number of values that can be accumulated and did not need to be stalled between accumulation sets. However, these designs required a relatively large amount of buffer memory and extremely complex control logic which limited their clock speed.

An improved single-adder streaming reduction architecture was later developed at the University of Twente [7]. This design is the current state-of-the-art, as it requires less memory and less complex control than Prassanna's SSA design. In this paper we describe a new streaming reduction technique that requires even less memory and simpler control logic than this design.

In each of the above discussed work, pre-made adders (usually generated with Xilinx Core Generator) have been used as the core of the accumulator. Another approach is to modify the adder itself such that the de-normalization and significand addition steps have a single cycle latency, which makes it possible to use a feedback without scheduling. To minimize the latency of denormalize portion, which includes an exponent comparison and a shift of one of the significands, both inputs are base-converted to reduce the width of exponent while increasing the width of the mantissa [8]. This reduces the latency of the denormalize while increasing the adder width. Since wide adders can be achieved cheaply with carry-chained DSP48 components, these steps can sometimes be performed in one cycle. This technique is best suited for single precision operands but can be extended to double precision as well [9]. However, in general this approach requires an unacceptably long clock period.

III. Background: Convey HC-1

At Supercomputing 2009, Convey Computer unveiled the production version of the HC-1, their contribution to the space of socket-based reconfigurable computers. The HC-1 is unique in several ways. Unlike in-socket coprocessors from Nallatech [10], DRC [11], and XtremeData [12]—all of which are confined to a footprint matching the size of the socket--Convey uses a mezzanine connector to bring the front side bus (FSB) interface to a large coprocessor board roughly the size of an ATX motherboard. This coprocessor board is housed in a 1U chassis that is fused to the top of another 1U chassis containing the host motherboard.

The design of the coprocessor board is depicted in Figure 1. There are four user-programmable Virtex-5 LX 330s, which Convey calls "application engines (AEs)". Convey refers to a particular configuration of these FPGAs as a "personality". Convey licenses their own personalities and corresponding compilers, development tools, simulators, debuggers, and application libraries. Currently, this includes three soft-core vector processors and a Smith-Waterman sequence alignment personality. Convey has not yet developed a personality that is specifically designed for sparse matrix computations, nor do they currently provide a sparse BLAS library targeted to one of their vector personalities.

The four AEs are each connected to eight independent memory controllers through a full crossbar. Each memory controller is implemented on its own FPGA and is connected to two Convey-designed scatter-gather DIMM modules. Each AE has a 2.5 GB/s link to each memory controller, and each SGDIMM has a 5 GB/s link to its corresponding memory controller. As a result, the effective memory bandwidth of the AEs is dependent on their memory access pattern. For example, each AE can achieve a theoretical peak bandwidth of 20 GB/s when striding across a minimum of eight of the sixteen DIMMs across eight different memory controllers, but this bandwidth could drop if two other AEs attempt to read from the same set of eight DIMMs since this may saturate the 5 GB/s DIMM-memory controller links. All four AEs can achieve an aggregate bandwidth of 80 GB/s when used together assuming a uniformly distributed access pattern across all sixteen DIMMs. For double-precision CSR SpMV with 32-bit column indices, this gives a peak performance of 80 GB/s / (12 bytes/2 FLOPs) = ~13 GFLOPs/s.

The most unique property of the HC-1 is that the coprocessor memory is fully coherent with the host memory. The coherence is implemented using the snoopy coherence mechanism built into the Intel FSB protocol. This creates a common global virtual address space that both the host and coprocessor share. As in classical snoopy coherence protocols, each virtual memory address in both the host and coprocessor local memory may be in an invalid, exclusive, or shared state. Shared memory locations are guaranteed have identical contents in both the host and coprocessor memory. Exclusive locations in one memory are invalid in the other, and represent locations that have been written but not yet read in (and thus automatically propagated to) the other memory. Whenever a virtual memory location transitions from exclusive to shared, the contents of the memory are updated in the requestor's local memory. The coherence mechanism is transparent to the user and removes the need for explicit DMA negotiations and transactions (required for PCI-based coprocessors).

The coprocessor board contains two FPGAs that together form the "application engine hub (AEH)". One of these FPGAs serves as the coprocessor board's interface to the FSB, maintains the snoopy memory coherence protocol and manages the page table for the coprocessor memory. This FPGA is actually mounted to the mezzanine connector. The second AEH FPGA contains the "scalar processor", a soft-core processor that implements the base Convey instruction set. The scalar processor is a substantial general-purpose processor architecture and features such as out-of-order execution, branch predication, register renaming, sliding register windows, and a virtualized register set. The scalar processor plays a significant role on the coprocessor because it is the mechanism by which the host invokes computations on the AEs. In Convey's programming model, the AEs act as coprocessors to the scalar processor as they implement custom instructions, while the scalar processor acts as coprocessor for the host CPU.

When using the Convey Personality Development Kit (PDK), code for the scalar processor is generally written by hand in Convey's own scalar processor assembly language. After assembly, the scalar processor code is linked into the host executable in a linker section named "ctext". On execution, scalar processor routines can be invoked from the host code by the blocking and non-blocking versions of the "copcall" API functions.

The scalar processor is connected to each AE via a point-to-point link, and uses this link to dispatch instructions to the AEs. Examples of such instructions include move instructions for exchanging data between scalar processor registers and AE registers, as well as the custom AE instructions, a set of 32 "dummy" instructions that can be used to invoke user-defined behaviors on the AEs. Through the dispatch interface on the AE, logic on the AEs can also trigger exceptions and implement memory synchronization behaviors.

Designing custom personalities requires the use of the Convey PDK. The PDK is physically comprised of a set of makefiles to support simulation and synthesis design flows, a set of Verilog support and interface files, a set of simulation models for all the non-programmable components of the coprocessor board (such as the memory controllers and memory modules), and a PLI-based interface to allow the host code to interface with a behavioral HDL simulator such as Modelsim.

Developing with the PDK involves working within a Convey-supplied wrapper that gives the user logic access to instruction dispatches from the scalar processor, access to all eight memory controllers, access to the coprocessor's management processor for debugging support, and access to the AE-to-AE links. However, the wrapper requires fairly substantial resource overheads: 66 out of the 288 18Kb BRAMS and approximately 10% of the slices on each FPGA. Convey supplies a fixed 150 MHz clock to the user logic on each FPGA.

IV. DATA FORMAT

We designed our SpMV kernels in 8157 lines of hand-written VHDL. In order to simplify the SpMV controller design, we use a slightly modified version of the CSR format in order to eliminate the use of the **ptr** array. As described above, the CSR format stores a matrix in three arrays, **val**, **col**, and **ptr**. **val** and **col** contain the value and corresponding column number for each non-zero value, arranged in a raster order starting with the upper-left and continuing column-wise left-to-right and then row-wise from the top to bottom. The **ptr** array stores the indexes within **val** and **col** where each row begins, terminated with a value that contains the

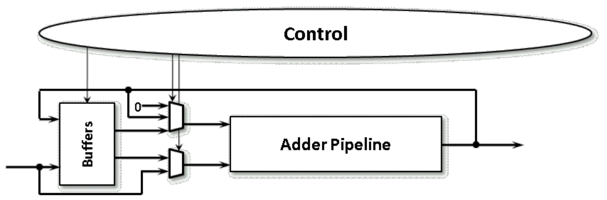

Figure 2. The accumulator is designed by adding control logic around the adder pipeline.

size of **val** and **col**. Instead of using the **ptr** array, we encode the end-of-row information within the **val** and **col** arrays using zero termination. Thus, to mark the termination of a row, we use 0 for both the **val** and **col** values. This increases the length of these arrays by the number of matrix rows and requires pre-processing of the matrix data. However, for applications such as iterative system solvers that invoke SpMV iteratively using an invariant matrix, this preprocessing step would only be a one-time upfront cost.

V. Reduction Circuit Design

Our goal was to develop a floating-point accumulator suited to double-precision CSR-based SpMV using only one double-precision floating-point adder coupled to external buffering and control to dynamically schedule the inputs to the adder. While designing the accumulator, we make the following assumptions:

(1) input values are delivered serially, one per cycle,

(2) output order need not match the arrival order of accumulation sets,

(3) the accumulation sets are contiguous, meaning that the values from different accumulation sets are not inter-mixed, and

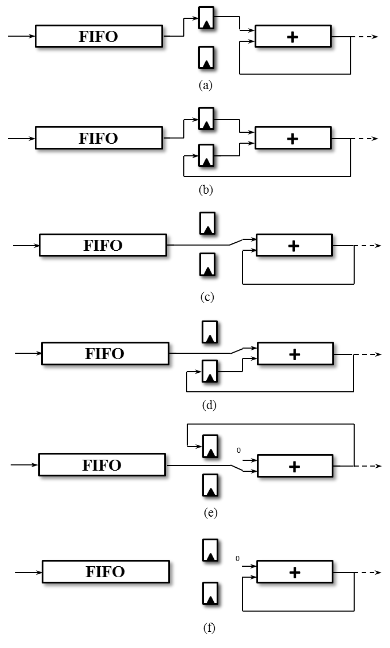

Figure 3. Data routing for rules 1 (a) through 5 (e), as well as the special case for rule 5 (f).

(4) the size of each accumulation set is variable and is not known *a priori*.

As Figure 2 depicts, the general idea is to add both control logic--in the form of comparators, counters, and buffers--around a single adder in order to form a dynamically scheduled accumulator. More specifically, the accumulator architecture consists of a set of data paths that allow input values and the adder output to be delivered into the adder or buffered based on their corresponding accumulation set ID and the state of the system. In this case, the set ID represents the matrix row for the purpose of computing a dot product. Data paths are established by the control unit according to five basic rules.

A. Data Path Rules

The rules which govern the inputs to the pipeline and inputs to the buffers are as follows:

Rule 1: Combine the adder output with a buffered value. Buffer the incoming value.

Rule 2: Combine two buffered values. Buffer the incoming value. Buffer the adder output (if necessary).

Rule 3: Combine the incoming value with the adder output.

Rule 4: Combine the incoming value with a buffered value. Buffer the adder output (if necessary).

Rule 5: Combine the incoming value with 0 to the adder pipeline. Buffer the adder output (if necessary).

In order to describe the rules in a more concise manner, we represent the incoming input value to the accumulator as ***input.value*** and ***input.set***, buffer *n* as ***buf$_n$.value*** and ***buf$_n$.set***, the value emerging from the adder pipeline as ***adderOut.value*** and ***adderOut.set***, the inputs to the adder pipeline ***addIn$_1$*** and ***addIn$_2$*** and the reduced accumulated sum as ***result.value*** and ***result.set***. Also, we represent the number of partial sums belonging to set *s* as ***numActive(s)***.

Using this notation, we re-describe the rules below, in descending order of priority.

Rule 1:

```
$if\,\exists n: buf_n.set = adderOut.set\ then\ \{$
$addIn_1 := adderOut, addIn_2 := buf_n$
$if\ input.valid\ then\ \{buf_n := input\}\}$
```

Rule 2:

```
$if\,\exists i,j: buf_i.set = buf_j.set\ then\ \{$
$addIn_1 := buf_i, addIn_2 := buf_j$
$if\ input.valid\ then\ \{buf_i := input\}$
$if\,numActive(adderOut.set) = 1\ then\{$
$result := adderOut\}$
$else\{buf_j := adderOut\}\}$
```

Rule 3:

```
$if\,input.valid\ then\ \{$
$if\ input.set = adderOut.set\ then\{$
$addIn_1 := input$
$addIn_2 := adderOut\}\}$
```

Rule 4:

```
$if\ input.valid\ then\ \{$
  $if \exists n{:}\ buf_n.set = input.set\ then\ \{$
    $addIn_1{:} = input$
    $addIn_2{:} = buf_n$
    $if numActive(adderOut.set) = 1\ then\ \{$
      $result{:} = adderOut\}$
    $else\{buf_n{:} = adderOut\}\}\}$
```

Rule 5:

```
$if\ input.valid\ then\ \{$
  $addIn_1 \coloneqq input$
  $addIn_2 \coloneqq 0$
  $if numActive(adderOut.set) = 1\ then\ \{$
    $result{:} = adderOut\}\ else\ \{$
    $if\ \exists n{:}\ buf_n.valid = 0\ then\ \{$
      $buf_n \coloneqq adderOut\}\ else\ \{error\}\}\}$
$else\ \{$
  $addIn_1 \coloneqq AdderOut$
  $addIn_2 \coloneqq 0\}$
```

Figure 3 shows various configurations of the reduction circuit: (a) the output of the pipeline belongs to the same set as a buffered value; (b) two buffered values belong to the same set (c) the incoming value and adder output belong to the same set; (d) the incoming value and buffered value belong to the same set; (e) the incoming value does not match the set of the pipeline output or any of the buffered values; (f) there's no incoming value.

B. *Tracking Set IDs*

As shown in Figure 4, in order to determine when a set ID has been reduced (accumulated) into a single value, we use three small dual-ported memories, each with a corresponding counter connected to the write port. Together, these memories keep track of the number of active values belonging to each set ID in each cycle, i.e. *numActive()*.

The write port of each memory is used to increment or decrement the current value in the corresponding memory location. The write port of one memory is connected to *input.set* and always increments the value associated with this set ID corresponding to the incoming value.

The write port of the second memory is connected to *adderIn.set* and always decrements the value associated with this set ID whenever two values from this set enter the adder. This occurs under all rules except for 5, since each of these rules implement a reduction operation.

The write port of the third memory is connected to *adderOut.set* and always decrements the value associated with this set ID whenever the number of active values for this set ID reaches one. In other words, this counter is used to decrement the number of active values for a set at the time when the set is reduced to single value and subsequently ejected from the reduction circuit.

The read port of each memory is connected to *adderOut.set*, and outputs the current counter value for the set ID that is currently at the output of the adder. These three values are added to produce the actual number of active values for this set ID. When the sum is one, the controller signals that the set ID has been completely reduced. When this occurs, the set ID and corresponding sum is output from the reduction circuit.

C. *Double Precision Streaming Multiply-Accumulator Design*

The core of each PE is a double precision streaming multiply-accumulator (MAC). The MAC consists of the reduction circuit-based accumulator fed by a double-precision multiplier.

During initialization, the host provides each PE with a workload, consisting of an initial row number, a starting address in the matrix memory, and the total number of values to multiply. The MAC keeps track of which row is currently being processed by incrementing the row number each time a zero termination is read. The MAC includes a FIFO for buffering incoming products between the multiplier and accumulator.

D. *Processing Element Design*

The processing element (PE) design is shown in Figure 5. It consists of a vector cache, a shifter, the multiply-accumulator as described above, a controller, and a result FIFO.

The role of the PE is to load matrix values and stream them into the multiply-accumulator. In order to achieve maximum memory bandwidth, the PE must load matrix data in parallel—across all eight memory interfaces—and serialize it before streaming it into the multiplier. The HC-1's memory interfaces respond to outstanding load requests in a randomized order. Also, each memory interface is only capable of addressing one eighth of the address space, and—due to the way the HC-1 partitions its address space among the memory controllers—the PE must read at least four consecutive words from each memory controller in order to read a contiguous block of addresses from the memory. To address both of these issues, a small on chip matrix cache is used to buffer incoming matrix data. A global, 64KB cache on each AE is subdivided into eight segments (one for each PE). The cache is organized as 32 x 512 x 32 BRAMs. Each segment holds 672 matrix values and their corresponding column number ("val-col pairs").

Matrix data is read from the cache in blocks of 42 val-col pairs, loaded in parallel into a shift register, and then each val-col pair is shifted out serially. Since the matrix cache is shared among all PEs, only one PE can read from it at any time. As such, access is arbitrated using fixed priority according to PE ID number. After each PE has consumed all sixteen blocks of cached matrix data held for it in the matrix cache, the PE sends a request signal to the top-level global memory controller, which then reads a new segment into the matrix cache. During a cache miss, this line size insures that there are an equal number of reads from each SGDIMM.

The incoming stream of column numbers is used to index the input vector to obtain the value to be multiplied with the matrix value. Many FPGA-based SpMV implementations in

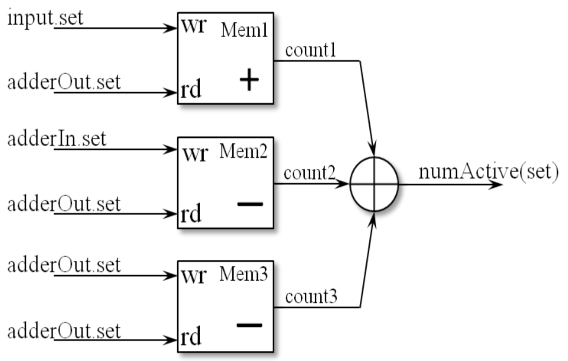

Figure 4. Design for tracking the number of active partial sums in an accumulation set.

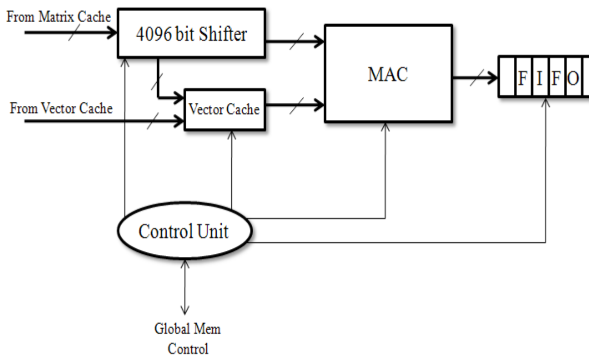

Figure 5. PE design.

the literature assume that a copy of the entire input vector for each multiplier can be stored chip or use blocking techniques to perform the SpMV over multiple passes of the input matrix. Since this architecture is designed for multiplying large matrices, we assume that the entire input vector will not fit within on-chip memory for each PE. As such, we designed a vector cache for each PE to hold a subset of the input vector. The non-values in many sparse matrices exhibit spatial locality, as values are often clustered in nearly columns. To take advantage of this, we implemented the cache as a traditional four-line direct mapped cache where each line holds 2048 consecutive double-precision values from the vector. In total the cache holds 8192 double-precision values. As with the matrix cache, during a cache miss there are an equal number of reads from each SGDIMM on the coprocessor memory.

The vector cache is local to each PE and vector data can thus be duplicated across the FPGA, depending on how the workload is distributed among the PEs. The top-level memory controller can only service one vector miss or matrix request at any time.

Each time the MAC computes a dot product value, the value and its corresponding set ID (i.e. row ID) are written into the PE's result FIFO. The global memory controller monitors the state of each PE's result FIFO and writes any pending results to coprocessor memory as soon as any pending matrix or vector requests have completed. The coprocessor memory address for each write is computed by adding the set ID, multiplied by eight, to the vector base address, which is written into all PEs by the host prior to execution. Result values are written to coprocessor memory using a fixed priority according to PE ID number. Result writes are given priority over cache misses, since they generally only require one cycle to complete while cache misses take substantially longer to service.

Our top-level design is shown in Figure 6. As shown, the shared matrix cache, each PE, and the result FIFOs all share access to the global memory controller. PE requests for vector and matrix cache misses, as well as result write requests, are serialized and arbitrated in one pool using a fixed-priority scheme according to PE ID number.

VI. Implementation on Convey HC-1

In this section we describe specific issues related to our personality design.

A. Resource Requirements

As described in Section 3, the HC-1 contains four user-programmable FPGAs called the application engines (AEs). Each AE has an interface to eight memory controllers and each memory controller interface has two 64-bit wide ports on the system clock, corresponding to each clock edge for the DDR memory.

TABLE I. Resource Utilization

Application Engine	Slices	BRAM	DSP48E
4 PE per AE	26,055 / 51,840 (50%)	146 / 288 (51%)	48 / 192 (25%)
8 PE per AE	38,225 / 51,840 (73%)	210 / 288 (73%)	96 / 192 (50%)

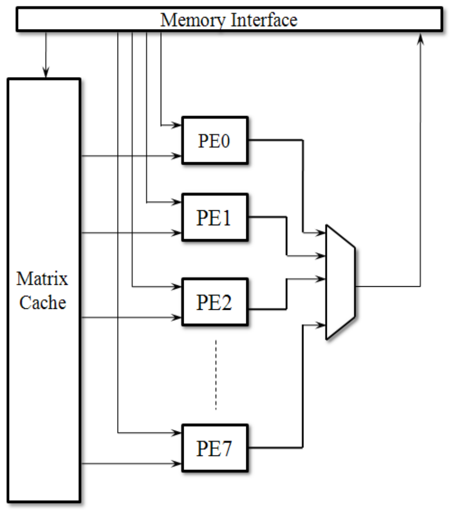

Figure 6. Top-level design.

TABLE II. EXPERIMENTAL RESULTS, NVIDIA CUSPARSE ON TESLA T10 GPU VS. CONVEY HC-1

Matrix	Application	r * c	nz	nz/row	CUSPARSE GFLOPs/s	HC-1 16 PE GFLOPs/s	HC-1 32 PE GFLOPs/s
dw8192	Electromagnetics	8192*8192	41746	5.10	0.49	1.71	1.65
t2d_q9	Structural	9801*9801	87025	8.88	0.94	2.07	2.48
epb1	Thermal	14734*14734	95053	6.45	0.80	2.18	2.56
raefsky1	Computational fluid dynamics	3242*3242	294276	90.77	2.59	2.94	3.85
psmigr_2	Economics	3140*3140	540022	171.98	2.83	2.84	3.94
torso2	2D model of a torso	115967*115967	1033473	8.91	3.00	1.06	1.17

When designing an on-chip cache, in order to take advantage of all available off-chip memory bandwidth, enough BRAMs must be instanced to consume 8 * 2 * 64 = 1024 bits in a single clock cycle. At minimum BRAM depth (512), this yields a minimum of sixteen 512x36 BRAMs for each vector cache and the matrix cache, requiring 144 BRAMs to support the fully populated (eight PE) AE design. Since the Convey-designed memory controller interfaces themselves require an overhead of 66 BRAMs.

Table 1 summarizes the resource usage of our AE design. As shown, BRAM and slice usage scale almost identically, while DSP usage is low in comparison.

B. *Parallelizing SpMV*

We parallelize the SpMV operation as follows. Since all matrix rows can be processed independently, we assign each PE an equal workload by dividing the set of matrix rows into equal-sized sets and distributing each set to the PEs across all four AEs. To do this, the host sends a matrix base address to each PE that specifies the starting location in the matrix data, as well as a starting row number and the number of rows to process before completion. This requires that matrix rows may not cross PE boundaries, which we ensure in the host by inserting zero padding where needed.

VII. GPU SPMV

GPUs have become a popular platform for accelerating scientific applications, particularly data parallel floating-point kernels such as computational fluid dynamics [13], ODE/PDE-based simulation [14], and medical imaging [15]. However, due to the challenges with working with compressed sparse matrices as described in Section I, achieving high performance on GPU architectures for CSR-formatted sparse matrix arithmetic remains an open problem for which efforts are still in progress by both NVIDIA and third parties [16, 17].

In addition, there is a still a substantial gap between single and double precision floating point performance, even on current generation GPUs (although this gap appears to be closing over subsequent generations of GPU architectures). According to NVIDIA, there was a 10X performance gap between single and double precision performance on Tesla GPUs and a 5X performance gap on Fermi GPUs [18].

GPUs have approximately 50% more memory bandwidth per GPU than the combined memory bandwidth of all four of the HC-1's AEs. This gives it an upper performance bound of near 20 GFLOPS/s. However, as shown in the results, the GPU achieves substantially lower ratio of this peak performance than the FPGA architecture.

We used NVIDIA's CUDA CUSPARSE library on a Tesla S1070 to measure GPU performance of our test matrices [19]. CUSPARSE supports sparse versions of basic linear algebra subroutines (BLAS) for a variety of sparse matrix representations, including CSR. We used CUSPARSE, as opposed to other third-party CUDA-based sparse matrix libraries, since CUSPARSE is the official *de facto* library endorsed by NVIDIA. Although the Tesla-S1070 consists of four GPUs, CUSPARSE can only to run on a single GPU even when multiple GPUs are available to the host.

VIII. EXPERIMENTAL RESULTS

We chose a set of test matrices from Matrix Market [20] and the University of Florida Matrix Collection [21]. We chose the matrices to cover a wide range of matrix orders, total number of nonzero entries, and average number of nonzero values per matrix row (sparseness).

The test matrices are summarized in Table 2, along with the corresponding throughput achieved with CUSPARSE and the HC-1. Throughput is computed as 2**nz* / (*execution time*), where the execution time is measured without including transfer time between host memory and coprocessor memory.

As shown in Table 2, the HC-1 generally outperformed the Tesla. However, the HC-1's performance doesn't scale well from four PEs/AE to eight PEs/AE, which is caused by memory interface contention and controller overhead. For one matrix, dw8192, scaling up the number of PEs slightly reduced performance.

IX. CONCLUSIONS AND FUTURE WORK

In this paper we described our CSR sparse matrix-vector multiplier and its implementation on the Convey HC-1 reconfigurable computer. The contributions of this paper include a new streaming reduction circuit design and an on-chip memory architecture optimized for CSR-formatted sparse matrix data. Our results show performance that exceeds that of the Tesla GPU. While the Tesla is a previous-generation architecture, so is the Virtex-5 LX FPGAs on which the HC-1 is based.

In the future, we plan to improve several limitations in our current design, described below.

(1) At this time, the top-level memory controller is only capable of servicing a single cache miss at a time. In other words, each cache miss must wait until the current cache

miss is complete and there is no overlap when servicing cache misses. References to off-chip coprocessor memory have long latency but the HC-1 supports a large number of outstanding requests. As such, it is possible to overlap requests for multiple cache misses, which would reduce the miss time.

(2) The HC-1's memory channels are independently controlled, and each has an independent request FIFO that has feedback signals to stall the requestor when the reference request FIFO becomes full. In our current design, the entire memory controller stalls when any of the stall signals becomes asserted. By decoupling the requests going out on the channels and only stalling the requests on the specific channels that request a stall, we may be able to further improve achieved memory throughput.

(3) During startup, several PEs may request identical cache lines, since the current memory controller does not support broadcasts to multiple PEs. By implementing broadcasted vector data requests, we can decrease the number of duplicated bulk off-chip memory loads.

We also plan to perform a more formal mathematical characterization of our reduction circuit design in order to bound the number of required buffers, input FIFO depth, number of active set counters, counter width, and row ID width as a function of adder pipeline depth and minimum accumulation set size.

In addition, there are many design trade-offs that may be explored for this design. Due to memory contention for the off-chip memory interfaces by the PEs, it may be possible to achieve higher performance by instancing fewer PEs but a higher amount of on-chip vector cache and/or matrix cache for each PE. Other degrees of freedom include number of vector cache lines and associativity. We plan to use modeling techniques to explore this design space to find the optimal number of PEs versus cache resources to achieve higher PE utilization across various matrix characteristics. We also plan to add performance counters to the design in order to instrument and measure the overheads required by the memory system. Finally, we plan to extend this design to support more sparse BLAS routines such as sparse matrix-matrix multiply.

REFERENCES

[1] Llyod N.Trefetthen and David Bau,III "Numerical Linear Algebra". Society for Industrial and Applied Mathematics.

[2] Intel, "Sparse Matrix Storage Formats", http://software.intel.com/sites/products/documentation/hpc/mkl/webhelp/appendices/mkl_appA_SMSF.html

[3] M. deLorimier, A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," Proc. 13th ACM/SIGDA Symposium on Field-Programmable Gate Arrays (FPGA 2005).

[4] L. Zhou, V. K. Prasanna, "Sparse Matrix-Vector Multiplication on FPGAs," Proc. 13th ACM/SIGDA Symposium on Field-Programmable Gate Arrays (FPGA 2005).

[5] J. Sun, G. Peterson, O. Storaasli, "Sparse Matrix-Vector Multiplication Design for FPGAs," Proc. 15th IEEE International Symposium on Field Programmable Computing Machines (FCCM 2007).

[6] L.Zhuo, V. K. Prasanna, "High-Performance Reduction Circuits Using Deeply Pipelined Operators on FPGAs," IEEE Trans. Parallel and Dist. Sys., Vol. 18, No. 10, October 2007.

[7] M. Gerards, "Streaming Reduction Circuit for Sparse Matrix Vector Multiplication in FPGAs". Master Thesis, University of Twente, The Netherlands, August 15, 2008.

[8] S. R. Vangal, Y. V. Hoskote, N. Y. Borkar, A. Alvandpour, "A 6.2-GFlops Floating-Point Multiply-Accumulator With Conditional Normalization," IEEE Journal of Solid-State Circuits, Vol. 41, No. 10, Oct. 2006.

[9] Krishna.K. Nagar, Jason D. Bakos, "A High-Performance Double Precision Accumulator," IEEE International Conference on Field-Programmable Technology, Dec. 9-11, 2009.

[10] Nallatech, "Intel Xeon FSB FPGA Accelerator Module," http://www.nallatech.com/Intel-Xeon-FSB-Socket-Fillers/fsb-development-systems.html.

[11] DRC Computer, "DRC Reconfigurable Processor Units (RPU)," http://www.drccomputer.com/drc/modules.html.

[12] XtremeData Inc., "XD2000i™ FPGA In-Socket Accelerator for Intel FSB," http://www.xtremedata.com/products/accelerators/in-socket-accelerator/xd2000i.

[13] A. Antoniou et al, "Acceleration of a Finite-Difference WENO Scheme for Large-Scale Simulations on Many-Core Architectures". 48th AIAA Aerospace Sciences Meeting, Orlando, Florida, January 2010.

[14] D. Sato et al, "Acceleration of cardiac tissue simulation with graphic processing units,". Medical and Biological Engineering and Computing, Volume 47, Number 9, 1011-1015, DOI: 10.1007/s11517-009-0514-4.

[15] M Roberts et al, "A Work-Effcient GPU Algorithm for Level Set Segmentation". Proc SIGGRAPH '10, ACM Special Interest Group on Computer Graphics and Interactive Techniques, July 2010.

[16] M. M. Baskaran; R. Bordawekar, "Optimizing Sparse Matrix-Vector Multiplication on GPUs," IBM Technical Report RC24704. 2008.

[17] N. Bell and M. Garland, "Implementing Sparse Matrix-Vector Multiplication on Throughput-Oriented Processor". Proc. Supercomputing '09 (SC09), November 2009.

[18] NVIDIA Corportation, "Whitepaper: NVIDIA' s Next Generation CUDA Compute Architecture: Fermi," http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf.

[19] NVIDIA Corportation, "CUSPARSE User Guide - CUDE CUSPARSE Library. PG-05329-032_V01". August 2010.

[20] Matrix Market, http://math.nist.gov/MatrixMarket.

[21] The University of Florida Sparse Matrix Collection, http://www.cise.ufl.edu/research/sparse/matrices.

.