FPGA vs. GPU for Sparse Matrix Vector Multiply

Yan Zhang[1], Yasser H. Shalabi[1], Rishabh Jain[2], Krishna K. Nagar[1], Jason D. Bakos[1]

Dept. of Computer Science and Engineering, Univ. of South Carolina
Columbia, SC 29208 USA
{zhangy,shalabi,nagar,jbakos}@engr.sc.edu[1]
rishabh.iitd.11@gmail.com[2]

Abstract—**Sparse matrix-vector multiplication (SpMV) is a common operation in numerical linear algebra and is the computational kernel of many scientific applications. It is one of the original and perhaps most studied targets for FPGA acceleration. Despite this, GPUs, which have only recently gained both general-purpose programmability and native support for double precision floating-point arithmetic, are viewed by some as a more effective platform for SpMV and similar linear algebra computations. In this paper, we present an analysis comparing an existing GPU SpMV implementation to our own, novel FPGA implementation. In this analysis, we describe the challenges faced by any SpMV implementation, the unique approaches to these challenges taken by both FPGA and GPU implementations, and their relative performance for SpMV.**

I. Introduction

FPGAs have been used as co-processors for scientific computing for many years. Recently, GPUs have begun to rapidly grow in this role. On the surface, GPUs seem to have several clear advantages over FPGAs: easier to program, less expensive in both hardware and development software, more ubiquitous, and have manufacturer-backed, standardized, and consistent programming abstractions, programming models, host interfaces, and system architectures.

GPUs are quickly becoming the dominant platforms for computational accelerators. GPUs have already demonstrated success in accelerating statistical phylogenetics [1], biological sequence alignment [2,3], image processing [4,5],network modelling [6], and computing cosmological data analysis [7].

Recent studies that have directly compared FPGAs and GPUs for scientific computing have concluded that GPUs do indeed outperform and are easier to program than FPGAs for several types of computations [8,9,10,11]. However, given that both technologies exploit parallelism using different techniques, it is clear that any comparisons between them must consider a particular type of computation.

One common computation that has not yet been used in a comparative study is sparse matrix-vector multiplication (SpMV). SpMV describes solving $y = Ax$, where y and x are vectors and A is a large matrix that is mostly composed of zero entries. SpMV is used as the kernel for many scientific applications, including those that include iterative linear system solvers (which is a kernel for solving many scientific problems such as approximating systems of partial differential equations).

Double-precision SpMV is a popular target for FPGA implementation because it is a notoriously difficult computation to accelerate. There are two reasons for this. First, the performance of any implementation is inherently dependent on memory bandwidth. Second, each element of the result vector must ultimately be computed by an accumulation of serially-delivered values, and the accumulation must be performed using a potentially deeply-pipelined double-precision floating-point adder. This creates a data hazard between each value to be accumulated and the previous value of the running sum.

In this paper, we describe a novel FPGA-based SpMV architecture that is built around our customized double-precision floating-point accumulator. Unlike FPGA application development, it is not possible to develop and integrate custom arithmetic units into GPU applications, so we use this as an opportunity to compare this SpMV architecture and its performance to an existing state-of-the-art implementation of a GPU-based SpMV kernel. We show that, despite the GPU's advantages in ease of programming, clock speed, and memory bandwidth, the GPU's inability to tailor the lower-level features of its microarchitecture limits its performance for certain types of computations.

II. Graphics Processor Units

Early implementations of non-graphics-based accelerators were achieved by mapping the desired general-purpose behaviors onto existing graphics APIs such as OpenGL and ActiveX. This technique was originally termed "General Purpose GPU (GPGPU)". However, this practice is no longer necessary, as general-purpose development environments and parallel programming models have recently become available for GPUs, including Compute Unified Device Architecture (CUDA), Stream (SDK), and OpenCL. In this section, we provide a brief overview of current-generation NVIDIA GPUs and the CUDA programming framework.

A. GPU Memory Hierarchy

NVIDIA's Tesla architecture, the GPU microarchitecture that we target in this paper, has a complex memory organization with multiple types of on-chip memories as well as off-chip, on-board DRAM. Each of these memories is optimized for specific access patterns, and one of the primary challenges when writing GPU code lies in efficiently mapping input, output, and state data to these memories, which for the most part must be manually performed by the program code.

The off-chip, on-board memory, called the "device memory," can be read and written by both the host and the GPU and its synchronization is handled by the execution framework. This memory is primarily used for sharing input and output data between the host and GPU, but can also be used as a backing store for the GPU to store state data. The GPU's on-chip memories are generally used for intermediate results and caching input data.

B. GPU Microarchitecture

An overview of the NVIDIA GPU microarchitecture is provided in the CUDA Programming Manual [12]. At the top level, the GPU consists of a set of "streaming multiprocessors (SM)". State-of-the-art GPUs have approximately 30 SMs (the GeForce GTX 260 card we use in our experiments has 27 as reported by the API but only 24 as reported in the programming manual).

Each SM contains an on-chip, programmer-controlled static memory called the *shared memory*, since this memory space is shared by all the threads running on the SM. This memory size is 16K x 32 bits and is organized in sixteen banks.

Each SM also contains two hardware-maintained, read-only caches that are hardware managed and therefore behave as a traditional memory cache to the off-chip *texture memory* and the *constant memory*. These caches are optimized for 3D rendering computations but are available for use for other types of computations.

Within each SM, there are eight scalar processor (SP) cores. Each SP contains its own register file that is managed by the compiler. There are 16,384 32-bit registers per SM (2,048 per SP). Organized this way, our GPU is effectively a 27 x 8 = 216-core, single-chip ("manycore") multiprocessor.

Each SM contains an issue unit that issues a single instruction across all eight SPs, and each of these can execute four copies of the instruction in parallel. Each SM is thus capable of issuing a single instruction (but with different operands) from up to 32 threads in true parallel. However, this is only possible when all the threads are executing the same instruction. In other words, it requires that all 32 threads to agree on a common execution path (i.e. no divergent control behavior). A grouping of threads that are grouped for this reason is referred to as a *warp*.

This approach is referred to as the SIMT (single-instruction, multiple-thread) technique. SIMT is similar to SIMD (single-instruction, multiple data), with the key difference being that SIMT abstracts away the data width from the programmer. However, the SIMT architecture exposes a weakness of the SM architecture, as the SIMT architecture only reaches full capacity of 32 parallel threads only when the control flow of all threads is identical (i.e. only the input data differs between the threads). In cases where threads follow divergent control paths through if-statements or loops, the threads must be serialized. This results in warps having less than 32 active threads. In the case, the warp must be issued multiple times having disjoint sets of active threads. For example, the warp may be issued once with threads 0-15 active, again with threads 16-23 active, and again with threads 24-31 active. This serialization hinders performance. As such, the SIMT architecture is optimized for data-parallel, control independent computations.

While threads are parallelized within warps and multiple warps are executed in parallel, each individual warp is not parallelized. Specifically, only one instance, at most, of any warp may be active in the execution pipeline at any time. This allows the SM architecture to exclude hardware for detecting data hazards among registers. The programming manual also implies that the SM does not perform any type of dynamic scheduling, out-of-order instruction execution, register renaming, branch prediction, or speculative execution. Presumably the lack of these features, which are present in nearly any modern microprocessor, constitutes the trade-off that allows each SM to contain more registers and functional units than modern microprocessors.

C. CUDA Programming Model

In this paper we use CUDA as the GPU development environment. The CUDA programming model is a 2-level hierarchy that defines groupings that encompass both programs and data. The finest granularity for parallelization is the *thread*, and up to 512 threads can be grouped within a *block*. All threads within a block share the on-chip shared memory, texture cache, and constant cache. The shared memory can be used for inter-thread communication and synchronization primitives are available.

Each block is assigned to a single SM. Multiple blocks can be assigned the same SM at the same time, although the maximum number of threads per SM is 1024 so a single SM can only support two maximum-sized blocks. There can be more blocks than SMs to execute them, but in this case block execution is serialized over the available SMs (the GPU does not support context switching). This gives the CUDA programming model transparent scalability. Blocks are organized into a 1, 2, or 3-dimensional *grid*, which is intended to be organized to match the structure of the input data. The maximum grid size is limited to 65535 blocks.

D. GPU Utilization and Throughput Metrics

GPU performance can be measured by the CUDA profiler using two metrics. The first is *occupancy*, which is the ratio of active warps to the maximum number of active warps per SM. For our GPU, the maximum number of active warps per SM is 16. Each SM can execute at most 32 active warps, equalling at most 1024 threads, An occupancy of 1 is desirable, but it is limited by the number of registers required, the amount of shared memory required, and instruction count required by the threads. Occupancy only measures the number of active warps per SM, but since the warps themselves can either be fully active or partially active, occupancy is not an accurate indicator of SM utilization or ratio of maximum to actual instruction throughput.

Instruction throughput ratio is another metric that measures the ratio of achieved instruction rate to peak instruction rate. Since each warp can have at most one instance of itself in the execution pipeline, if all warps are active, no instructions can be issued until active instructions complete. This situation causes the instruction throughput ratio to fall below one. This is commonly caused when memory latency cannot be

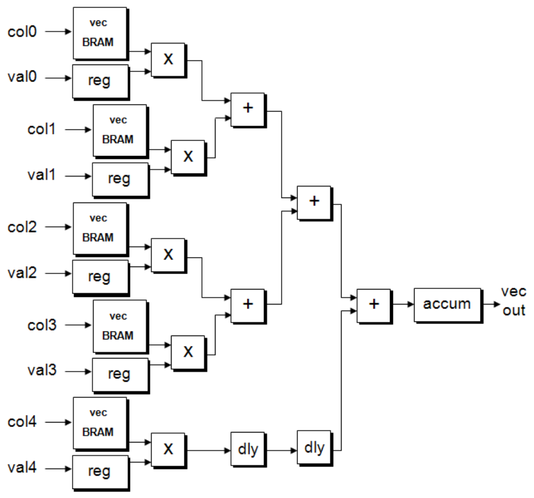


Fig. 1. FPGA-based SpMV architecture.

overlapped with arithmetic instructions, by bank conflicts on shared memory, and by inactive threads within a warp due to thread divergence.

III. Sparse Matrix-Vector Multiply (SpMV)

Both the FPGA and GPU SpMV implementations described in this paper use the Compressed Sparse Row (CSR) format. 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 in the matrix, arranged in an order starting with the upper-left of the matrix and continuing column-wise left-to-right and then row-wise from the top to bottom. The *ptr* array stores the indices within *val* and *col* where each row begins, terminated with a value that contains the total length of *val* and *col* (i.e. *ptr*[0]=0, *ptr*[4] = the index within *val*/*col* where row 4 begins, *ptr*[*nr*]=*nz*, where *nr* = the number of rows and *nz* = the number of non-zero values).

CPU, FPGA, and GPU implementations of sparse matrix-vector multiply achieve only a fraction of the peak floating-point throughput [13,14]. It is normally the case that sparse matrices are too large to fit in on-chip memory and thus must be read from an off-chip source. Also each storage element of the matrix (a double-precision floating-point value and corresponding column number) is used only for a single multiply-accumulate operation (two floating-point operations). This high ratio of communication to computation makes the overall computation highly dependent on memory bandwidth.

The CSR format allows the matrix data to be read sequentially and thus its access pattern is known *a priori*. As such, the memory access latency can be hidden by making multiple outstanding requests for matrix data and "streaming" the matrix data into the SpMV co-processor. The vector data, on the other hand, is randomly accessed. Its access pattern depends on the sequence of incoming matrix value column numbers. However, since repeated references can be made to a single entry, the vector data has the potential for temporal locality. It is normally the case that the vector is small enough to be stored in on-chip memory. In most cases, multiple copies of the vector are stored on-chip to exploit on-chip memory parallelism.

There is also a challenge for implementing the dot-product computation. Since it is not feasible to perform all scalar multiplications required for each dot product in parallel, there is an unavoidable requirement to perform an accumulation of serially-delivered floating-point values. Since each addition performed during the accumulation depends on the result of the previous addition, the latency of the adders creates a data hazard.

IV. FPGA SpMV Architectures

In this section we describe previous work in FPGA-based SpMV implementations, which are summarized in Table 1. There is a significant amount of literature in this area, but we only highlight a few significant examples.

deLorimier and DeHon designed an SpMV architecture that worked around the accumulator data hazard by forcing the host to statically schedule and zero-pad the input matrix values such that the rows are interleaved [15]. However, this scheduling adds significant input data overhead that makes the architecture's performance highly dependent on the structure of the matrix. The peak performance of their architecture on a Xilinx Virtex-II 6000 was 2240 MFLOPS and on average they expect this architecture to achieve 66% of its peak, although these results do not include the computational.

Prasanna's group at the University of Southern California was one of the first to design an SpMV architecture based on the design shown in Figure 1 [16]. Prasanna's design was notable for being the first to incorporate a specialized "reduction circuit" to solve the dot product accumulation problem dynamically, without needing to perform static data scheduling. However, this early reduction circuit had several; drawbacks, such as the requirement for it to be flushed between matrix rows and a maximum set size. They estimated that their design would achieve approximately 45% of its peak performance on average. Prasanna later developed two improved reduction architectures called the double- and single-strided adders (DSA, SSA) that solved many of the problems of earlier accumulator design [17]. These new architectures required significantly less adders, did not limit the maximum number of values that can be accumulated and did not need to be flushed between data sets. However, the performance of each of these reduction circuits was limited by their extensive control and memory overhead as compared to performance of the floating-point adder upon which they were based.

V. GPU SpMV Architectures

To date there have been two primary contributions to the development of double-precision sparse matrix-vector multipliers for NVIDIA GPUs. The first was developed at NVIDIA Research and was noted for supporting a wide range of matrix storage representations, including DIA, ELL, HYB, and CSR [18]. The second was developed by a group at Ohio State, apparently on behalf of IBM [14]. Their

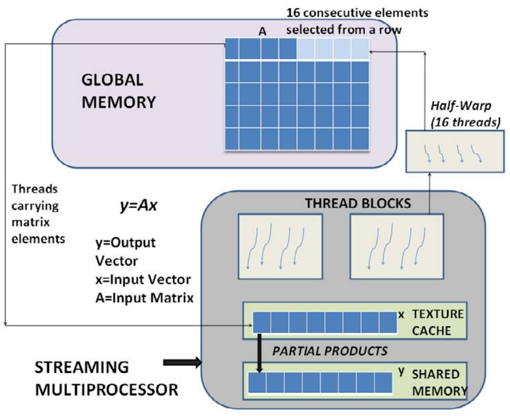


Fig 2. Memory organization of the GPU-based SpMV implementation.

implementation was built on top of NVIDIA's SpMV CSR kernel and added several optimizations, mostly involving memory management.

Figure 2 shows the memory organization used in their implementation. The matrix was stored in device (off-chip) memory, the input vector is stored in texture memory (and cached in each SM's on-chip texture cache), and the output vector is stored in the shared memory of each SM.

To achieve the highest possible off-chip memory bandwidth, memory references must be organized carefully in the GPU code. Simultaneous memory accesses by the threads in a half-warp may be *coalesced* into single aligned memory transactions of 32, 64, or 128 bytes. To ensure this, the programmer must use the thread ID as an offset within the array index to guarantee address alignment.

In the SpMV implementation, each matrix row is assigned among a group of sixteen threads (one half warp). In other words, each thread is assigned one sixteenth of the non-zero elements in a row. The matrix data is pre-processed by the host such that each row is zero-padded to force the number of non-zero elements per row to be a multiple of sixteen. Each thread exits after processing its portion of the row.

Each group of sixteen threads can read 64 or 128 bytes in a single memory load transaction. When reading from the *val* array, each thread receives eight bytes (one double precision value) each, and when reading the corresponding *col* array, each thread receives 4 bytes (one integer) each. 512 threads are assigned to each block (the maximum allowable). This allows each block to process 32 rows. The number of blocks is set to total number of rows divided by 32 rounded up.

In order to determine GPU utilization, we ran a set of test matrices through the GPU SpMV implementation using the GPU profiler and recorded the occupancy and instruction throughput ratio for each matrix. The input vectors are randomly generated.

Table 3 lists the matrices used in this analysis and GPU instruction throughput utilization for each. Each of the matrices were obtained from Matrix Market [19] and the University of Florida Sparse Matrix Collection [20].

The occupancy achieved by the GPU SpMV for all the matrices was one, meaning that each thread used a sufficiently small amount of registers and shared memory that each SM was capable of executing the maximum number of threads possible. Surprisingly, the instruction throughput ratio is relatively constant across all matrices, ranging from 0.799 to 0.886. However, the GPU's off-chip memory bandwidth and performance in GFLOPS is correlated to the average number of non-zero elements per row. As such, there is a performance penalty associated with threads having low iteration counts. The GPU code computes GFLOPS by dividing the total number of non-zero elements by two (since each element must be multiplied and accumulated) and dividing it by the kernel execution time.

VI. FPGA SpMV Architecture

Our SpMV architecture is shown in Figure 3 and is built around our novel double precision accumulator architecture. The architecture is based on instancing parallel dot product modules, each of which includes a copy of the vector in BRAM, a multiplier, and an accumulator. In this configuration, each accumulator will perform a dot product between the input vector and each row of the input matrix. For this to work, all values from each matrix row must be mapped to the same accumulator.

We chose this organization rather than the one depicted in Figure 1 for two reasons. First, the accumulator architecture, which we describe below, has a minimum set size of eight, which gives us a minimum number of non-zero values per row as eight. In the original organization shown in Figure 1, the minimum number of non-zero values per row would instead be the product of eight and the number of multipliers (40 in this case). Second, in this new configuration each accumulator functions independently, which allows the architecture to easily be scaled up for FPGA boards that provide more memory bandwidth than ours as long as there

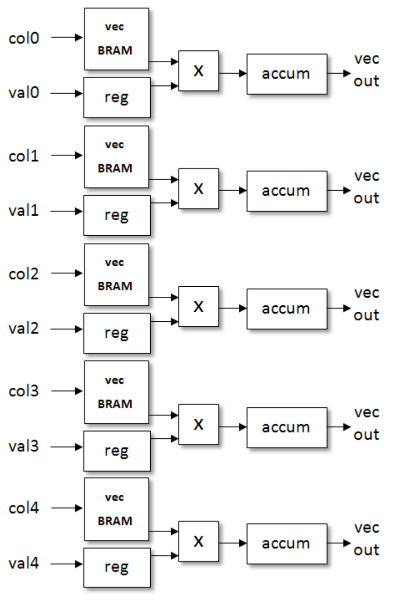
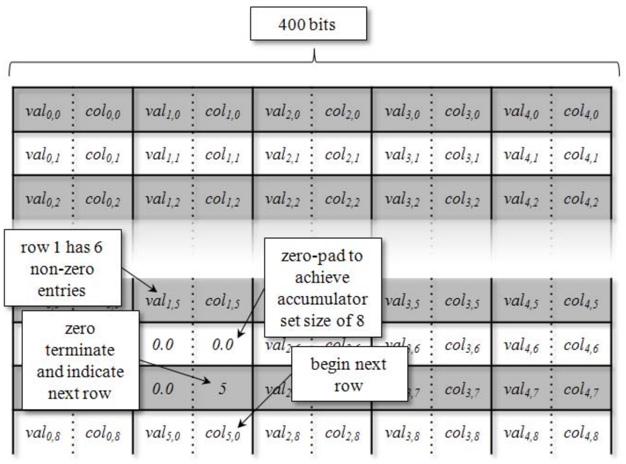


Fig. 3. SpMV Architecture. The FPGA can read five matrix values and their corresponding column values per cycle. The FPGA associates a copy of the input vector, a multiplier, and an accumulator with each.



400 bits
row 1 has 6 non-zero entries
zero-pad to achieve accumulator set size of 8
zero terminate and indicate next row
begin next row

Fig. 4. Matrix data scheduling technique. In this case, 400-bit "packets," that are read each cycle, are composed of five "slots"—one for each accumulator.

are sufficient resources on the FPGA. However, in order to guarantee this data mapping, the matrix data must be scheduled.

Our FPGA card, the Annapolis Micro Systems WildStar-II Pro, has a 432-bit interface to its on board SRAM (six banks of 36-bit wide DDR2 SRAM). Using 16-bit column values requires 80 bits per value/column pair, thus our SpMV architecture can read five value/column pairs per cycle (using 400 of the 432 bits). This is equivalent to a memory bandwidth of 5 GB/s at a 100 MHz clock and 10 GB/s at a 200 MHz clock. Note that this is approximately six times less bandwidth than our GeForce GTX 260 card and 8-16% of its 1.24 GHz clock rate (note the GPU was manufactured on a 65 nm process, the FPGA on 0.13 μm process).

A. Data Scheduling

The matrix data must be pre-processed and scheduled before being sent to the FPGA card's on-board memory. In the hardware design, each of the five multipliers keep track of which matrix row is currently being processed, and they are initialized with rows 0, 1, 2, 3, and 4. Non-zero values from each matrix row are scheduled to be sent to a single multiplier until all the non-zero values from the row are exhausted. At this point, there is a zero termination where the value is set to 0.0 and the column value is used to specify which row will scheduled to appear next for that multiplier.

Since our accumulator requires a minimum set size of eight non-zero values, any rows that have less than this number must be zero-padded. Zero padding must also be used near the end of the matrix data when there are less than five rows that still contain non-zero values in the matrix data. Pad values have a value of 0.0 and a column value of 0.

Figure 4 shows an example of a matrix data file. As shown, the matrix data is constructed as a sequence of 400-bit packets, corresponding to the FPGA's memory interface width. Each packet contains five slots, one for each accumulator. In the figure, the first subscripted value represents to the row number. The second subscripted value counts each non-zero value in the row.

B. Accumulator Architecture

Our top-level accumulator architecture is shown in Figure 5. As shown in the figure, the first two stages are used to condition the incoming value. The base conversion step (box 1) converts the incoming value from base 2 to an arbitrary base, which is set as a "generic" parameter in our VHDL. For base *b*, this step performs the following:

1. adds a 1-bit to the left-hand side of the 52-bit significand value (the implied leading digit to the left of the decimal point),
2. shifts the significand value to the left by the value stored in the low order lg *b* bits in the exponent field (note that this effectively adds *b* - 1 bits to the width of the significand),
3. strips the lower lg *b* bits from the exponent, and
4. adds a zero sign bit and zero carry-out bit ("00") to the left side of the resultant (53 + *b* - 1)-bit base-*b* significand value, resulting in (54 + *b*) total bits.

The next stage performs an arithmetic negation of this value if the original sign bit was set to one (box 2).

The third stage is where the de-normalize (box 4) and significand addition begins. This is comprised of the following steps:

1. compare the high-order 11-(lg *b*) bits of *exp1* and *exp2* (corresponding to base-*b* significands *sig1* and *sig2*),
2. if *exp1* > *exp2*, shift *sig2* to the right by *b**(*exp1*-*exp2*) bits, else shift *sig1* to the right by *b**(*exp2*-*exp1*) bits,
3. add the resultant *sig1* and *sig2*, and
4. if the addition caused the carry out bit to be set to one, add one to max(*exp1*,*exp2*) and shift the sum *b* bits to the right (box 5).

This series of steps involves sequential operations on both

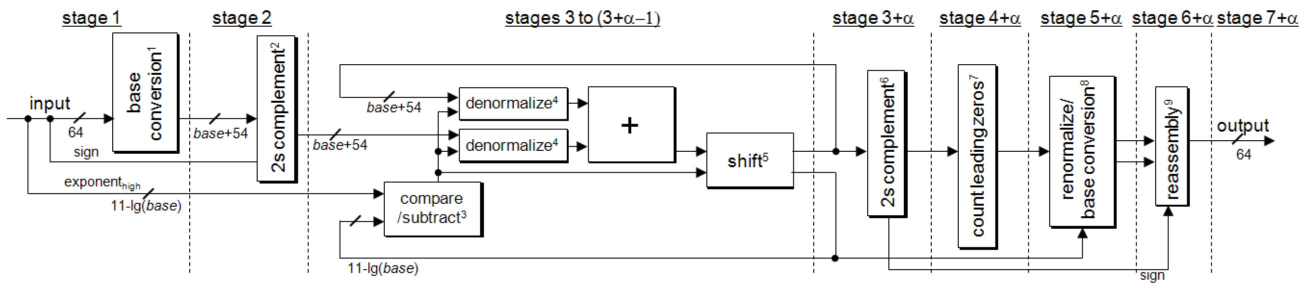


Fig. 5. Top-level design of accumulator architecture. Alpha represents the pipeline latency of de-normalize/addition datapath.

the high order bits of the original exponent and the base-converted significands. Larger values of *b* will result in lower latency exponent operations but a wider and thus higher latency integer addition, while lower values of *b* will result in wider and thus higher latency exponent operations and lower latency integer addition.

Table 1 shows how the exponent comparison (and exponent subtraction) width versus adder width scale as the base is increased. The remaining stages are used to re-condition the base-converted sum into IEEE 754 format. Box 6 computes the absolute value of the sum. In the next stage, box 7 counts the number of leading zeros. In the next stage, box 8 uses this information to shift the significand and adjust the exponent in order to convert the significant back to base 2 and then to re-normalize. The last stage repackages the value into IEEE 754 format.

Since the de-normalize/add step is the critical logic path in the accumulator design, the trade-off between the exponent comparator width and the adder width will give different levels of circuit performance as the base is varied. In order to determine the base that provides the highest performance, we synthesized versions of the design shown in Figure 5 over a range of base values.

For each base value, we also added delays to the outputs of the de-normalize/add stage and enabled retiming and pipelining in the synthesizer to give it the ability to further pipeline the normalize/add step. In other words, we added cycles of latency to the de-normalize/add step to improving the overall pipeline speed of the accumulator. Note that in addition to pipelining the exponent comparison, exponent subtraction, and significand addition, there are also two shifters involved in this step (de-normalize and renormalize in the case of a carry-out).

In this analysis, the versions of the design having a de-normalize/add latency greater than one are not functionally correct without the addition of the reduction features described later in this paper. However, since this analysis is for timing only, and because the reduction features will require only minimal timing overhead, we do not include them in this analysis. Our analysis included base values ranging from 4 to 512. We used Synplify Pro 8.8.0.4 as the synthesis tool and targeted our in-house Virtex-2 Pro 100 FPGA. From these results, we have selected a pipeline depth of 3 and base values of 32 to 128 for carrying the accumulator design forward into the later steps of the design flow.

C. *Reduction Circuit*

The de-normalize/add step is pipelined over three stages and thus creates a data hazard. To solve this, we designed a novel reduction circuit that is specifically tailored to operate over this shallow pipeline, which allows the reduction algorithm to be simplified and thus not impose control overhead on the performance of the adder. The trade-off required for the simplified implementation is a minimum set size, but since it is computed as a function of the adder latency it doesn't present a significant problem. For a three cycle latency the minimum set size is eight, which is small enough to be manageable for this application.

Under normal operation, the reduction circuit operates in steady-state mode where it routes the current input value and the output of the pipeline back into the input of the pipeline. In this operating state, the pipeline contains α partial sums, where α is the pipeline depth. When there is a change in input set, the pipeline must take a series of actions to coalesce these partial sums while still accepting values from the next input set.

As shown in Figure 6, our reduction circuit design requires a single input buffer and a single output buffer. The inputs to the pipeline can be routed according to the following four different configurations:

- **Configuration A:** buffer the incoming value, route the buffered output value and the output currently being produced by the pipeline back into the pipeline. For a pipeline depth of α, this must occur once for every internal node of a binary tree having α leaves, equalling α - 1 occurrences. To ensure that the buffer depth may be limited to one, the value in the input buffer must be consumed (using configuration C) once between each instance of configuration A.
- **Configuration B:** add the incoming value with the value currently being produced by the pipeline. This is the "steady-state" configuration, and is used when accumulating the current input set into α partial sums.
- **Configuration C:** add the buffered input value with the incoming input value. This occurs during cycles when the output of the pipeline need not re-enter the pipeline. This includes the cycles where the pipeline output is buffered (which must occur once before the architecture enters configuration A) and the cycles where an input set is reduced to a final sum (which occurs once per input set).
- **Configuration D:** add the incoming value with zero. This only occurs one time per input set, prior to the first time an input is buffered.

For a pipeline depth of $\alpha = 3$, starting with the first cycle where the incoming value belongs to a new input set, the controller will instruct the reduction circuit to cycle through a deterministic series of configuration changes for the following eight cycles. This sequence of configurations will reduce the previous input set to a single sum while continuing to accept values from the new input set. The required controller can be implemented as a single 9-state FSM, where all state transitions are unconditional except for the condition when the next input set differs from the current input set. This is detected by comparing the input set from stage 3 and stage 2 in the top-level accumulator pipeline. Starting with the cycle immediately prior to an input set change (i.e. row number change), the controller cycles through the sequence B, D, A, C, B, A, B, B, C, B/D (depending if a new set enters this cycle).

Routing can be performed with a 2-input mux before the first input and a 3-input mux before the second input to the de-normalize/add pipeline. Note that each input value consists of a (54 + *b*)-bit significand and a (11-lg *b*)-bit upper exponent

TABLE I
EXPONENT COMPARISON WIDTH VS. ADDER WIDTH

Base	Exponent Compare	Adder Width
2	11	54
32	6	86
64	5	118
128	4	182
256	3	310
512	2	566
1024	1	1078
2048	0	2102

value. The controller also raises the data_valid flag to indicate the output sum is valid for each input set.

The reduction algorithm described above has a latency that inherently requires a minimum set size in order to allow for the coalesce process for the previous input set to finish before the current set ends. For a pipeline depth of α, the minimum set size is $\alpha\lceil \lg\alpha + 1\rceil - 1$ cycles, since after each α-cycle pass, there are half the number of partial sums in the pipeline. As shown in the example above, the minimum set size for α=3 is 8, while for α=4 is 11. Note that for deeper pipelines, the minimum set size imposed by this reduction algorithm is prohibitive.

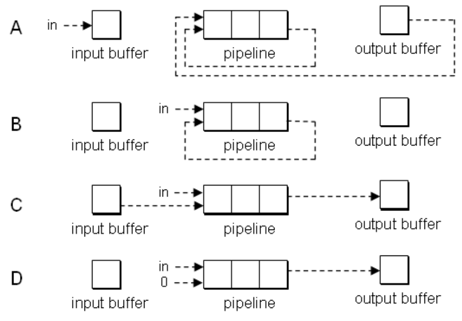


Fig. 6. Configuration states for the reduction circuit.

VII. EXPERIMENTAL RESULTS

Table 2 shows the performance results of both the GPU and FPGA SpMV implementations. For the FPGA results, we compute the GFLOPS using the same method as the GPU implementation, where the execution time is divided by the number of non-zero values from the matrix multiplied by 2. The utilization column indicates the number of non-zero matrix values stored in the input data divided by the total number of entries, showing the level of zero-padding required by each matrix when encoded into the data format required by the FPGA implementation.

Our FPGA SpMV implementation operates at 170 MHz on our Annapolis Micro Systems WILSTAR 2 Pro card. The clock speed is limited by the maximum operating speed of the

TABLE II
TEST MATRICES AND PERFORMANCE RESULTS

Matrix				GPU Performance Metrics			FPGA Performance Metrics			
Matrix	Order/ dimensions	n_z	Ave. n_z/row	Inst. Thrghpt.	Mem. Bndwdth.	GFLOPS	Data Utilization @actual bandwidth	GFLOPS at 170 MHz (8.5 GB/s)	Data Utilization @adjusted bandwidth	Adj. GFLOPS
TSOPF_RS_b162_c3	15374	610299	40	0.799	58.00 GB/s	10.08	0.941	1.60	0.940 @51.0 GB/s (x6)	9.59
E40r1000	17281	553562	32	0.859	57.03 GB/s	8.76	0.970	1.65	0.968 @51.0 GB/s (x6)	9.87
Simon/olafu	16146	1015156	32	0.824	52.58 GB/s	8.52	0.984	1.67	0.983 @51.0 GB/s (x6)	10.03
Garon/garon2	13535	373235	29	0.814	49.16 GB/s	7.18	0.965	1.64	0.963 @42.5 GB/s (x5)	8.18
Mallya/lhr11c	10964	233741	21	0.839	40.23 GB/s	5.10	0.875	1.49	0.873 @34 GB/s (x4)	5.94
Hollinger/ mark3jac020sc	9129	52883	6	0.842	26.64 GB/s	1.58	0.646	1.10	0.643 @25.5 GB/s (x3)	3.28
Bai/dw8192	8192	41746	5	0.827	25.68 GB/s	1.28	0.637	1.08	0.635 @25.5 GB/s (x3)	3.24
YCheng/psse1	14318 x 11028	57376	4	0.875	27.66 GB/s	1.24	0.498	0.85	0.497 @25.5 GB/s (x3)	2.54
GHS_indef/ ncvxqp1	12111	73963	3	0.886	27.08 GB/s	0.98	0.663	1.13	0.662 @25.5 GB/s (x3)	3.38

accumulators (note that the Virtex-2 Pro does not contain any hardware adders like the Virtex-4 and later—as such, the adders in our implementation are implemented are LUT-based). This gives a maximum theoretical throughput of 1.7 GFLOPS. The actual time required by the FPGA implementation depends on the level of encoding overhead required by the data encoding format, which depends on the structure of the matrix.

Our FPGA card has significantly less memory bandwidth than our GPU card, but this is due to limitations of the FPGA's system board and not of the FPGA (i.e. our board only devotes 396 of the Virtex-2 Pro's 1164 user I/O pins for its six SRAM banks and the on-board SRAM modules themselves are relatively low speed). As a result, for each test we also provide a theoretical result where the performance of the FPGA architecture is scaled by the memory bandwidth achieved by the GPU for each test. Note that in this scaling, we do not account for the possibility that the scaled architecture will not fit in the available FPGA resources or if there enough user-defined pins on the FPGA to support the scaled bandwidth amount (although our architecture and data encoding format can be trivially scaled across multiple FPGAs).

We do, however, consider the effect that scaling will have on matrix encoding efficiency, since increasing the packet size will decrease the utilization. This occurs as a result of additional zero padding that is necessary near the end of the matrix data when the number of remaining rows is less than the number of slots in the scaled packet.

While these theoretical results provide a more fair comparison between the FPGA and GPU, there are still biased toward the GPU. This is because, as shown in the table, the bandwidth achieved by the GPU depends on the number of non-zero entries of the input matrix. This is presumably due to the fact that smaller matrices generate less threads and thus less warps to cover memory latency of the other warps. This leads to lower average instruction throughput (due to the load instructions), which results is lower effective memory bandwidth. On the other hand, the FPGA implementation will always use all of its available memory bandwidth reading the formatted matrix file regardless of the size of the input matrix. Even so, even when given equivalent memory bandwidth to the GPU's effective bandwidth for each matrix, the FPGA exceeds the GPU implementation in all but one of the tests.

VIII. CONCLUSIONS

In this paper we describe and compare the implementation and performance of a GPU and FPGA SpMV kernel. The GPU greatly outperforms the FPGA, but the FPGA co-processor board is severely disadvantaged by low memory bandwidth as compared to the GPU co-processor board. The FPGA exceeds the performance to the GPU when its memory bandwidth is artificially scaled to match the bandwidth achieved by the GPU. This is primarily due to the ability to design a customized accumulator architecture for the FPGA implementation, allowing it to make more efficient use of the memory bandwidth.

IX. ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant Nos. CCF-0844951 and CCF-0915608.

REFERENCES

[1] M. A. Suchard, A. Rambaut, "Many-Core Algorithms for Statistical Phylogenetics," Bioinformatics Vol. 25 no. 11 2009, pages 1370–1376.

[2] W. Liu, B. Schmidt, G. Voss, W. Muller-Wittig, "Streaming Algorithms for Biological Sequence Alignment on GPUs," IEEE Trans. on Parallel and Distributed Systems, Vol. 18, No. 9, Sept. 2007.

[3] W. Liu, B. Schmidt, G. Voss, A. Schroder, W. Muller-Wittig, "Bio-Sequence Database Scanning on a GPU," Proc. 20th International Symposium on Parallel and Distributed Processing (IPDPS 2006), Apr. 25-29, 2006.

[4] J. L. T. Cornwall, O. Beckmann, P. H. J. Kelly, "Automatically Translating a General Purpose C++ Image Processing Library for GPUs," Proc. 20th International Symposium on Parallel and Distributed Processing (IPDPS 2006), Apr. 25-29, 2006.

[5] M. Gong, R. Yang, "Image-gradient-guided Real-time Stereo on Graphics Hardware," Proc. 5th International Conference on 3D Digital Imaging and Modeling (3DIM05), June 13-16, 2005.

[6] Z. Xu, R. Bagrodia, "GPU-accelerated Evaluation Platform for High Fidelity Network Modeling," Proc. 21st International Workshop on Principles of Advanced and Distributed Simulation (PADS 07), June 12-15, 2007.

[7] D. W. Roeh, V. V. Kindratenko, R. J. Brunner, "Accelerating cosmological data analysis with graphics processors", Proc. 2nd Workshop on General Purpose Processing on Graphics Processing Units, 2009.

[8] S. Che, J. Li, J. W. Sheaffer, K. Skadron, and J. Lach. "Accelerating Compute Intensive Applications with GPUs and FPGAs." In Proceedings of the IEEE Symposium on Application Specific Processors (SASP), June 2008. (pdf)

[9] Z.K. Baker, M.B. Gokhale, J.L. Tripp, "Matched Filter Computation on FPGA, Cell and GPU," Proc. 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2007. FCCM 2007.

[10] X. Tian, K. Benkrid, "High Performance Quasi-Monte Carlo Financial Simulation: FPGA vs. GPP vs. GPU," ACM Transactions on Reconfigurable Technology and Systems, to appear.

[11] J. Chase, B. Nelson, J. Bodily, Z. Wei, D.-J. Lee, "Real-Time Optical Flow Calculations on FPGA and GPU Architectures: A Comparison Study," Proc. 16th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2009. FCCM 2009.

[12] NVIDIA CUDA Programming Guide 2.2, http://www.nvidia.com.

[13] Jason D. Bakos, Krishna K. Nagar, "Exploiting Matrix Symmetry to Improve FPGA-Accelerated Conjugate Gradient," 17th Annual IEEE International Symposium on Field Programmable Custom Computing Machines, April 5-8, 2009.

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

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

[16] L. Zhou, V. K. Prasanna, "Sparse Matrix-Vector Multiplication on FPGAs," Proc. 133h ACM/SIGDA Symposium on Field-Programmable Gate Arrays (FPGA 2005).

[17] 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.

[18] N. Bell, M. Garland, "Efficient Sparse Matrix-Vector Multiplication on CUDA", NVIDIA Technical Report NVR-2008-004, December 2008.

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

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