An Integrated Reduction Technique for a Double Precision Accumulator

Krishna K. Nagar
Dept. of Computer Sci.and Engr.
University of South Carolina
Columbia, SC 29208 USA
nagar@cse.sc.edu

Yan Zhang
Dept. of Computer Sci. and Engr.
University of South Carolina
Columbia, SC 29208 USA
zhangy@cse.sc.edu

Jason D. Bakos
Dept. of Computer Sci.and Engr.
University of South Carolina
Columbia, SC 29208 USA
jbakos@cse.sc.edu

ABSTRACT

The accumulation operation, $A_{n+1} = A_n + X$, is perhaps one of the most fundamental and widely-used operations in numerical mathematics and digital signal processing. However, designing double-precision floating-point accumulators presents a unique set of challenges: double-precision addition is usually deeply pipelined and without special micro-architectural or data scheduling techniques, the data hazard that exists between A_{n+1} and A_n requires that each new value of X delivered to the accumulator wait for the latency of the adder. There have been several techniques proposed for alleviating this problem, but each carries significant overheads and/or restrictions on input characteristics. In this paper we present a design for a double precision accumulator that requires no timing overhead relative to the underlying add operation. We achieve this by integrating a coalescing reduction circuit within the low-level design of a base-converting floating-point adder. To demonstrate our accumulator design, we use it in a sparse matrix vector multiplication architecture, achieving a throughput of up to 3.7 GFLOPS.

Keywords

Categories and Subject Descriptors: C.1.3 [Processor Architectures]: Other Architecture Styles--Heterogeneous (hybrid) systems, C.5.4 [Computer System Implementation]: VLSI Systems, B.2.1 [Arithmetic and Logic Structures]: Design Styles—Pipeline

General Terms: Reconfigurable computing, high-performance computing, scientific computing, reduction, accumulator, double precision, IEEE 754

1. INTRODUCTION

The accumulation operation, depicted in Fig. 1, is required by any special-purpose architecture that performs a summation operation. When used as a component of a micro-architecture that supplies a new value to the accumulator every clock cycle, the designer cannot use a simple feedback-based accumulator circuit if the floating-point adder has a latency greater than one cycle, as this would prevent the adder from providing the current sum before the next value arrives. Also, in many applications the sequence of input values may belong to different accumulation sets and there may be no separation between values belonging to different sets.

Over the years, FPGA kernel designers have dealt with this problem using a number of different methods. Older designs used a static approach, where inputs were delivered to the accumulator in an ordering where the values belonging to different accumulation sets were interleaved according to the latency of the adder [1]. The disadvantage to this approach is that input data must be pre-processed and, depending on the characteristics of the input data, often led to low adder utilization.

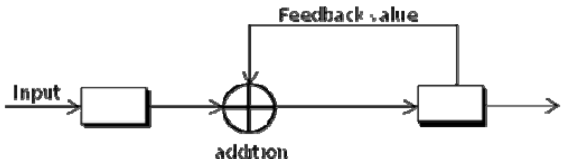

Fig. 1. Serially-delivered accumulator.

Table 1. Comparison with State of the Art for Double Precision Accumulation

Accumulator	Speed	Resources	Notes
Prasanna DSA [3]	142 MHz (Virtex 2 Pro)	2215 slices + 2 d.p. adders + 3 BRAMS	Based on 892-slice, 14-stage 170 MHz adders Out-of-order outputs
Prasanna SSA [3]	165 MHz (Virtex 2 Pro)	1804 slices + 1 d.p. adder + 6 BRAMS	
Gerards [5]	200 MHz (Virtex-4 LX 160)	3556 slices that includes 1 d.p. adder + 9 BRAMS + 3 DSP48	Based on 1220-LUT, 3-DSP48, 12-stage 324 MHz adder In-order outputs
This implementation	370 MHz (Virtex-5 LX sg-2)	998 slices + 3 DSP48	In-order outputs

2. PREVIOUS WORK

The current state-of-the-art is a series of dynamic approaches that are summarized in Table 1. Prasanna's group at the University of Southern California has written several seminal papers in this area [2,3,4]. Their first design was essentially a collapsed a binary adder tree organized as a linear series of adders. In this case the required number of adders is the log of the maximum number of expected input values to be accumulated. Each adder in the system has an exponentially lower utilization than the adder before it. This design also had a very long latency, had to be flushed between input sets, and the maximum input set size was fixed at design time.

Their next design was based on the notion of using a "coalescing" adder, which uses a feedback loop but has an output buffer and special control logic that sends the last two partial sums from the adder output back to the adder input and allows the adder to go through several passes to compute a final sum. Since this requires several cycles, they added a second adder such that while one adder coalesces the other adder begins reading the next set. This design required only two adders and an input FIFO of size $\alpha\lceil \lg\alpha + 1\rceil$ where α is the adder latency. However, the controller complexity required by this technique reduced its maximum speed by nearly 20% relative to the maximum speed of the floating-point adders that it was built around.

Their final design overcame this limitation and required only one adder but also required two memories of size α^2 and a control overhead speed reduction of about 3%.

An improved single-adder reduction architecture was later developed at the University of Twente [4]. This architecture reduced the memory requirement to $3\alpha + \alpha\lceil \lg \alpha \rceil + 2$ but its timing overheads imposed a 38.3% reduction in speed relative to the adder upon which it was based.

A team from UT-Knoxville and Oak Ridge National Laboratory recently developed another accumulator where inputs are striped across a bank of α adders [5]. The main disadvantage of this design was the number of adders that are required.

Each of these previous designs is based on instancing pre-made floating-point adders (usually generated with Xilinx Core Generator) into a top-level reduction architecture. In an alternative approach that is currently limited to single-precision, the adder itself is changed such that the de-normalization and significand addition step are designed to have a single cycle latency, making it possible to use a simple feedback-based accumulation technique. The other aspects of the adder, specifically those that deal with IEEE 754 housekeeping, need not be included in the feedback. In order to make this approach practical, the designer must minimize the latency across both the de-normalize (composed of a comparison and subtraction of the exponents and a shift of one of the significands) and the significand addition (an integer addition). Both Intel and a group from Princeton accomplished this by increasing the adder width while decreasing the width of the exponent comparison for the de-normalize step by converting the significand from base-2 to base-32 [6,7].

Our goal was to design a double precision accumulator that requires minimal memory and control logic such that its speed is limited only by the speed of the significand addition. Because double precision requires a wider exponent compare and significand addition, we pipeline this portion of the architecture instead performing it in a single stage. To solve the resultant data hazard, we apply a simplified reduction technique integrated within the adder design.

Sign	Exponent	Significand
63	62:52 (11 bits)	51:0 (52 bits)

$\underbrace{\hspace{10em}}_{\text{64 bits}}$

Fig. 2. IEEE 754 Double Precision Floating Point Format

3. ACCUMULATOR ARCHITECTURE

The IEEE 754 double precision format is shown in Fig. 2. The exponent field is 11 bits (encoded as an unsigned value with a "bias" of 1023) and the significand field is 52 bits (which represents the bits to the right of the decimal point if the mantissa with an implicit value of one to the left of the decimal point).

Our top-level accumulator architecture is shown in Fig. 3. As a research architecture, our current design does not recognize special IEEE 754 values such as *infinity* and *not-a-number*, does not perform rounding (i.e. it always rounds down), and does not support de-normalized values. We assume that the addition of these features will not significantly affect our results or conclusions.

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),

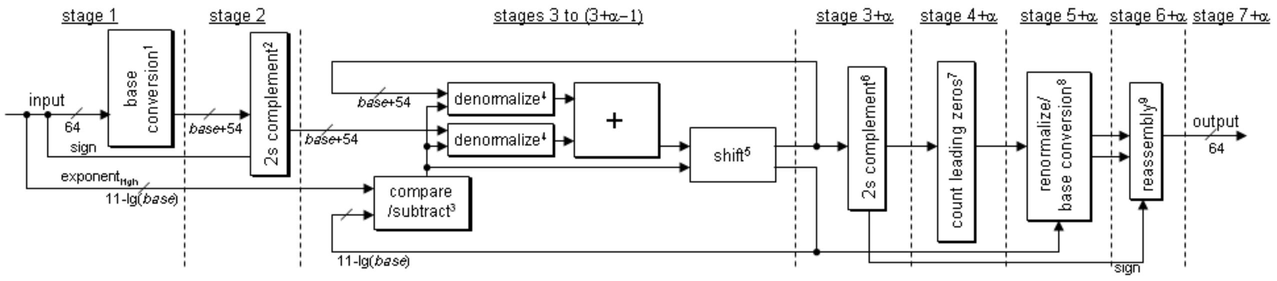

Fig. 3. Top-level design of accumulator architecture. Alpha represents the pipeline latency of addition datapath.

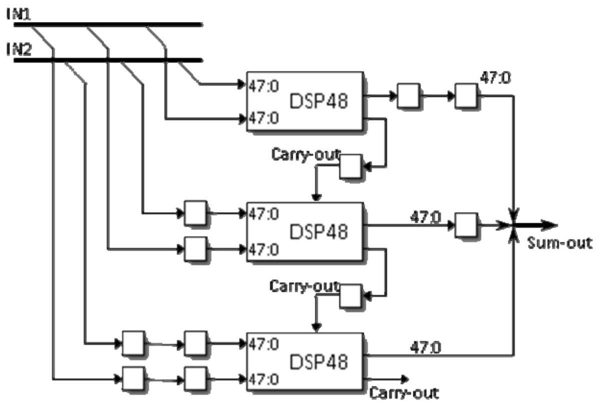

Fig. 4. Pipelined, cascaded DSP48 blocks forming a wide adder.

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 and significand addition begins. This is comprised of the following steps:

1. compare the high-order 11-(lg *b*) bits of both exponents, *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.

This series of steps involves sequential operations on both 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.

The first three columns of Table 2 show how the exponent comparison (and exponent subtraction) width versus adder width scale as the base is increased.

The single-precision accumulator designs from the literature perform these operations in one cycle. Further, they chose 32 as the base but do not provide analysis to justify why this value was chosen. Since our accumulator is double precision, we performed a synthesis-based analysis in order to choose this base value. We describe this analysis in the next section.

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.

4. COMPARATOR AND ADDER ANALYSIS

As demonstrated in Fig. 4, modern FPGAs contain hard DSP blocks that can be cascaded to create integer adders of arbitrary width. When pipeline registers are inserted between each adder in the carry chain, the entire adder array can operate at the same operating frequency as a single adder at the cost of one cycle of latency per adder that is added to the array.

In our accumulator design, the clock speed of the adder is an

Table 2: Synthesis results for the de-normalize and corresponding add operation for various base values. In each case, the adder is implemented using cascaded DSP48 blocks and is organized such that a single 48-bit add is performed each cycle to pipeline the carry chain as shown in Fig. 4. As a result, an *n*-bit add operation requires $\lceil n/48 \rceil$ cycles. For a Virtex-5 LX part with a -2 speed grade, this architecture allows an arbitrary-width adder to have a fixed operating frequency of 365.7 MHz.

Base	Add Width/ Exp. Width	# DSP48s/ Add Latency (cycles)	Denorm Latency (cycles)	Denorm Freq. (MHz)	Total Latency (cycles)
16	70 / 7	2 / 1	1	119.2	2
			2	196.6	3
			3	218.3	4
			4	201.3	5
32	86 / 6	2 / 1	1	246.1	2
			2	310.8	3
			3	341.9	4
			4	348.4	5
64	118 / 5	3 /2	1	368.4	3
			2	308.6	4
			3	402.3	5
			4	365.1	6
			5	365.1	7
128	182 / 4	4 / 3	1	372.6	4
			2	407.4	5
			3	518.2	6
			4	518.2	7
			5	518.2	8
256	310 /3	7 / 6	1	494.9	7
			2	518.2	8
			3	518.2	9
			4	518.2	10
			5	518.2	11

upper bound on the overall accumulator speed. In the case of a Virtex-5 LX with a speed grade of -2, this speed is 365.7 MHz. The challenge is to prevent any other aspect of the design from replacing the adder as the critical path and imposing a lower clock speed.

Since the number of adders can be varied, our strategy is to adjust the base value to reduce the logic latency of the de-normalize operation to reach equilibrium with the adder speed. To determine this, we performed a series of synthesis runs, where we synthesized the de-normalize operation corresponding with various bases, with the de-normalize operation itself pipelined over a range of cycles. For this analysis we used Synopsys Synplify Pro 2009.06.

As shown in Table 2. this analysis showed that a base-64 de-normalize over one cycle, which involves a 5 bit exponent comparison, reached a clock speed that matched that of the adder. Note that the synthesis results for base-64, unlike the other bases, show an inconsistent trend as pipeline depth is increased. This is a side-effect of the method by which the synthesis tool pipelines this operation. Also notice how the de-normalize clock speed reaches its maximum at 518.2 MHz.

From these results, we have selected a total de-normalize/add pipeline depth of 3 and a base values of 64 for carrying the accumulator design forward into the later steps of the design flow.

5. REDUCTION CIRCUIT

The reduction circuit must reconcile the data hazard created by the three cycle latency of de-normalize/add step, since each input to this step depends on the most recent output. Note that any of the previously designed reduction circuits from the literature would fulfil this requirement. However, these previous reduction circuits were designed for much longer pipelines (i.e. an entire floating-point adder pipeline as opposed to only the de-normalize/add pipeline). In this case, we only need a reduction circuit to operate over a three-stage pipeline, which gives us the opportunity to design a reduction circuit that has significantly less resource and control overhead than previous designs.

5.1 Design Goals

The design goal for our reduction circuit design is to make its implementation simple in order to not impose timing overhead on the overall adder pipeline. In other words, the addition of the logic required to transform an adder into an accumulator should not shift the critical path away from the adder itself (i.e. the accumulator should operate at the same speed as the adder on which it is based).

In previous work in reduction circuit design, the control and memory overheads required scale with the pipeline depth. Our goal is for the memory requirement to remain constant and only the control logic to scale with the pipeline depth.

5.2 Operating Principle

After a sufficient number of clock cycles have passed reducing a single input set, 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.

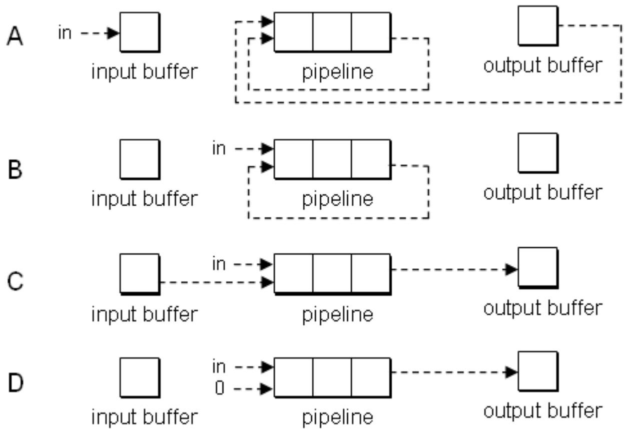

Fig. 5. Configuration states for the reduction circuit.

As shown in Fig. 5, 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.

As shown in Table 3 for a pipeline depth of α = 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. Note that α1, α2, and α3 represent the three partial sums from the previous 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.

Table 3. Example of the Reduction Circuit Operating Over a Pipeline of Depth 3.

Clock cycle	Accum. input	Input buffer	Adder pipeline			Output buffer	Note
0							Configuration B (Steady-state)
1	B1		α3	α2	α1		Configuration D Set A complete, adder pipeline full
2	B2		B1+0	α3	α2	α1	Configuration A
3	B3	B2	α1+ α2	B1+0	α3		Configuration C
4	B4		B2+B3	α1+ α2	B1+0	α3	Configuration B
5	B5		B1+B4	B2+B3	α1+ α2	α3	Configuration A
6	B6	B5	α1+ α2+ α3	B1+B4	B2+B3		Configuration B
7	B7	B5	B2+B3+B6	α1+ α2+ α3	B1+B4		Configuration B
8	B8	B5	B1+B4+B7	B2+B3+B6	α1+ α2+ α3		Configuration C Set A accumulation complete, use this cycle to clear input buffer
9	B9/C1		B5+B8	B1+B4+B7	B2+B3+B6		Configuration B/D Earliest valid cycle for input set C to begin

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

5.3 Minimum Set Size

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. This is the trade-off for the ability to use simple reduction logic. However, since the reduction used in our accumulator only spans the de-normalize and integer add portion of the floating-point adder, we are able to limit the reduction to a shallow pipeline.

Including the 3 stage de-normalize/add pipeline, the top-level accumulator pipeline is 10 stages. However, when determining the overall latency of the accumulator, the latency of the reduction operation must also be included. Since the reduction operation requires 8 cycles to compute the final sum after the last input value of each set, the total latency of the accumulator is 7 cycles for the base conversion and IEEE 754 overhead plus 8 cycles for the reduction, totalling 15 cycles.

6. CASE STUDY: SPARSE MATRIX-VECTOR MULTIPLY

Sparse matrix-vector multiplication (SpMV) is a common operation in numerical linear algebra and is the computational kernel of many scientific applications. These include those that contain an iterative linear system solver, which itself is a kernel for solving many scientific problems such as approximating systems of partial differential equations. SpMV is one of the original and perhaps most studied targets for FPGA acceleration.

SpMV describes solving ***y*** = ***Ax***, where ***y*** and ***x*** are vectors and ***A*** is a large matrix that is mostly composed of zero entries. Double-precision SpMV is a popular target for FPGA implementation because it is a notoriously difficult computation to effectively accelerate. There are two reasons for this. First, the performance of any implementation is inherently dependent on the memory bandwidth for reading the matrix.

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 section we describe our own SpMV architecture that is based on our accumulator design. In this case, the most substantial challenge is to determine how to overcome the minimum set size limitation of the accumulator.

6.1 Matrix Format

Our SpMV implementations described in this paper uses 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, 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*

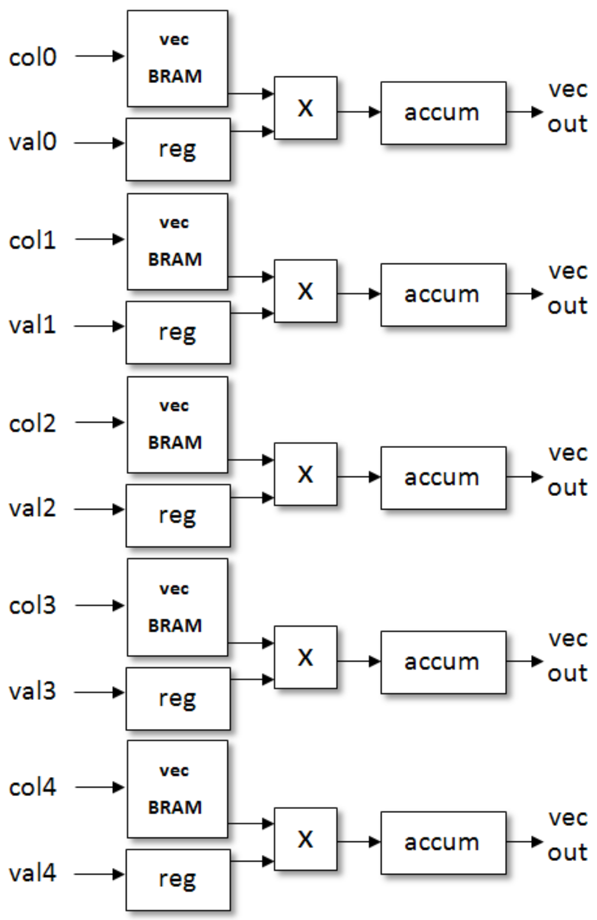

Fig. 6. SpMV Architecture. The FPGA can read five matrix values and their corresponding column values per cycle. The FPGA associates a copy of the input vector, a multiplier, and an accumulator with each.

where each row begins, terminated with a value that contains the size 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).

6.2 Microarchitecture

Our SpMV architecture is shown in Fig. 6. Our FPGA card 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, and we can thus read five value/column pairs per cycle (using 400 bits). For each of these, we associate an on-chip copy of the input vector in block RAM, 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. In this case, 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 are sufficient resources on the FPGA. However, in order to guarantee this data mapping, the matrix data must be scheduled.

6.3 Matrix 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 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.

Fig. 7 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.

6.4 Performance Results

Table 4 lists a set of matrices that we use in our analysis. Each of the matrices were obtained from Matrix Market [8] and the University of Florida Sparse Matrix Collection [9].

The table shows the overheads required by both the zero padding and zero termination. The utilization column expresses this overhead as a ratio between the actual number of non-zero values and the number of entries used after zero padding and zero termination.

Since the design runs at 370 MHz, the architecture will perform five double precision multiplications and five double precision additions per cycle, giving a peak throughput of 3700 MFLOPS. However, since no useful work is performed for zero

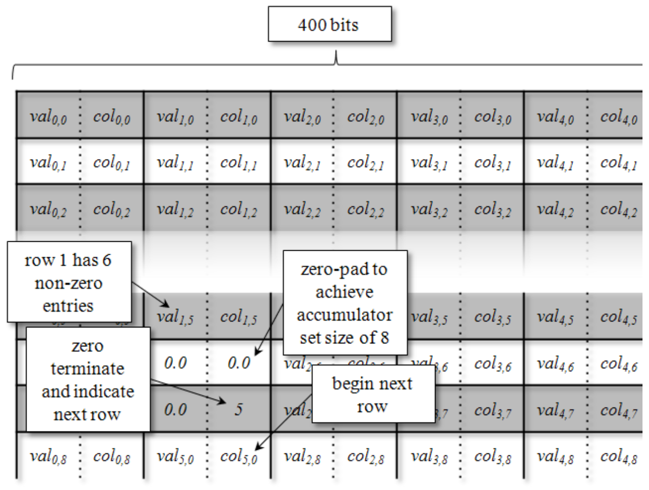
400 bits
$val_{0,0}$ $col_{0,0}$ $val_{1,0}$ $col_{1,0}$ $val_{2,0}$ $col_{2,0}$ $val_{3,0}$ $col_{3,0}$ $val_{4,0}$ $col_{4,0}$
$val_{0,1}$ $col_{0,1}$ $val_{1,1}$ $col_{1,1}$ $val_{2,1}$ $col_{2,1}$ $val_{3,1}$ $col_{3,1}$ $val_{4,1}$ $col_{4,1}$
$val_{0,2}$ $col_{0,2}$ $val_{1,2}$ $col_{1,2}$ $val_{2,2}$ $col_{2,2}$ $val_{3,2}$ $col_{3,2}$ $val_{4,2}$ $col_{4,2}$
$val_{1,5}$ $col_{1,5}$ $val_{3,5}$ $col_{3,5}$ $val_{4,5}$ $col_{4,5}$
0.0 0.0 $col_{3,6}$ $val_{4,6}$ $col_{4,6}$
0.0 5 $col_{3,7}$ $val_{4,7}$ $col_{4,7}$
$val_{0,8}$ $col_{0,8}$ $val_{5,0}$ $col_{5,0}$ $val_{2,8}$ $col_{2,8}$ $val_{3,8}$ $col_{3,8}$ $val_{4,8}$ $col_{4,8}$
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. 7. Matrix data scheduling technique. In this case, 400-bit "packets," that are read each cycle, are composed of five "slots"—one for each accumulator.

entires (due to zero termination and zero padding), the effective throughput is determined by the data utilization.

Due to the accumulator's limitation of minium input set size, data utilization is affected by the third column, "Average nonzero value per row". For matrices with an average number of nonzero values per row higher than the minium required by the accumulator (8 in this case), the data utilization is relatively close to 1 (0.87-0.98). For matrices with an average number of nonzero values per row less than the minium input set size, data utilization is much less due to zero padding.

7. CONCLUSION

In this paper we describe a new design technique for high performance, low resource double precision accumulators. Our approach combines elements from two previous techniques. The first technique is to use base-converted adders to reduce the comparison complexity of the de-normalize operation through the use of a wider addition operation. This allowed the accumulator feedback to be constructed over only a portion of the double precision adder. Thus the reduction circuit can be built around a shallower pipeline, which in turn allowed the reduction circuit itself to require less overhead than previous designs.

We demonstrated our accumulator in a sparse matrix-vector multiplier architecture. When used with a data scheduling technique, our architecture achieved between 50% and 98% of its peak performance, depending on the density of the input matrix.

In our future work, we will enhance the accumulator to overcome the minimum set size limitation, perform a more thorough performance analysis, and expand the features of the accumulator to implement a full IEEE 754 implementation.

8. ACKNOWLEDGMENTS

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

9. REFERENCES

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

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

[3] 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.

[4] 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.

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

[6] 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).

[7] 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.

Table 4. Test Matrices and Performance Results

Matrix	Order/ dimensions	Ave. non-zeroes per row	Number of non-zero entries	Entries after zero-padding	Entries after zero termination	Number of 400-bit packets	Data Utilization	MFLOPS at 370MHz
TSOPF_RS_b162_c3	15374	40	610299	633231	648625	129725	0.941	3482
E40r1000	17281	32	553562	553562	570855	114171	0.970	3589
Simon/olafu	16146	32	1015156	1015156	1031330	206266	0.984	3641
Garon/garon2	13535	29	373235	373243	386800	77360	0.965	3571
Mallya/lhr11c	10964	21	233741	256014	267020	53404	0.875	3236
Hollinger/ mark3jac020sc	9129	6	52883	72608	81835	16367	0.646	2390
Bai/dw8192	8192	5	41746	57360	65575	13115	0.637	2357
YCheng/psse1	14318x11028	4	57376	100960	115295	23059	0.498	1843
GHS_indef/ ncvxqp1	12111	3	73963	99412	111535	22307	0.663	2453

[8] Z. Luo, M. Martonosi, "Accelerating Pipelined Integer and Floating Point Accumulations in Configurable Hardware with Delayed Addition Techniques," IEEE Transactions on Computers, Vol. 49 No. 3 March 2000.

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

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