2009 17th IEEE Symposium on Field Programmable Custom Computing Machines

Exploiting Matrix Symmetry to Improve FPGA-Accelerated Conjugate Gradient

Jason D. Bakos, Krishna K. Nagar
Department of Computer Science and Engineering
University of South Carolina
Columbia, SC USA
{jbakos, nagar}@cse.sc.edu

Abstract— **In this paper we describe a new approach for accelerating the Conjugate Gradient (CG) method using an FPGA co-processor. As in previous approaches, our co-processor performs a double-precision sparse matrix-vector multiplication. However, our implementation doubles the amount of computation per unit of input data by exploiting the symmetry of the input matrix and computing the upper and lower triangle of the input matrix in parallel. Using a Virtex-2 Pro 100 FPGA, we have achieved an observed computational throughput of 1155 MFLOPS.**

Keywords- FPGA, reconfigurable computing, high-performance computing, sparse matrix vector multiply, conjugate gradient

I. INTRODUCTION

Linear system solvers are used frequently in scientific computing. They are also computationally expensive and highly parallelizable, and this has made them popular targets for FPGA acceleration. Linear system solvers constitute the kernel computation of many partial differential equation (PDE) solvers [1], which are used for modeling many types of physical systems. Linear system solvers can be divided into two categories: direct, where the solution is computed by evaluating a derived formula, and iterative where the solution is approximated until a certain acceptable value is reached. Direct methods can only be feasibly used for solving small systems of equations. Nevertheless, there has been recent work to accelerate such methods, and these efforts are motivated by problems in electrical power distribution [2,3]. These solvers are usually limited to matrices of order less than 100. Iterative methods are used to solve larger systems of equations, but are only guaranteed to converge to a solution if the input matrix adheres to a set of characteristics that are specific to each method. The computationally intensive component of performing iterative methods is the matrix-vector multiplication, and one need only to implement this operation in hardware to effectively accelerate an iterative method. In most iterative methods, the matrix is invariant across iterations so it need only be transferred to the co-processor memory once per method invocation. As such, it is generally the case that the co-processor's memory bandwidth will determine the performance of the matrix-vector multiplication.

In this paper we describe our accelerator architecture for the double-precision Conjugate Gradient method for large input matrices. In order to reduce the effect that off-chip memory transfer capacity on the multiplier's throughput, we take advantage of the property of CG that requires the input matrix to be symmetric. This allows the multiplier to compute both the top triangle and bottom triangle of the input matrix in parallel. To our knowledge, this is the first sparse matrix-vector multiplier architecture that exploits matrix symmetry to nearly double the ratio of computation to communication and thus achievable computational throughput. This paper describes an actual, working implementation of this accelerator architecture, and our test results include all actual communication, system, and data encoding overheads.

II. BASE ARCHITECTURE

Our SMVM architecture is specifically designed to accelerate the conjugate gradient (CG) method. CG requires that the input matrix be symmetrical. When only the upper triangle and the diagonal is used to represent the matrix, the FPGA performs two multiplications for each non-zero matrix input value that is not on the diagonal. These products are used for accumulating the dot products for two distinct rows, which nearly doubles the amount of computation performed per unit of I/O. To take advantage of this symmetry, we have divided our SMVM architecture into two sections--one that performs computation for the upper triangle, including the diagonal, and the other that performs computation for the lower triangle excluding the diagonal. Each matrix value streamed into the accelerator is fanned-out to both sections. As a result, each dot product computed by the SMVM architecture is partially computed by both the upper and lower architectures, except for the first row. This arrangement produces two result vectors, for the upper and lower triangle, which are added by the host in software to produce the final vector product.

We designed our SSpVxM core using custom VHDL and targeted our Annapolis Micro Systems Wild Star II Pro computing card with its Virtex-II Pro 2 100 FPGA. Before computation begins, the matrix is transferred using DMA from the host memory into the FPGA card's onboard SRAM, which is composed of six banks of 36-bit wide DDR2 SRAM modules that can theoretically deliver a new word every 5 ns for a total of 5.15 GB/s of memory bandwidth. We map the memory banks onto two 64-bit double-precision matrix values and their corresponding 16-bit column identifiers, utilizing 160 of the 216 bits. Our double-precision adders and multipliers (generated with Xilinx Core Generator) are limited to 148 MHz in the context of our design, allowing our design to utilize

978-0-7695-3716-0/09 $25.00 © 2009 IEEE
DOI 10.1109/FCCM.2009.44

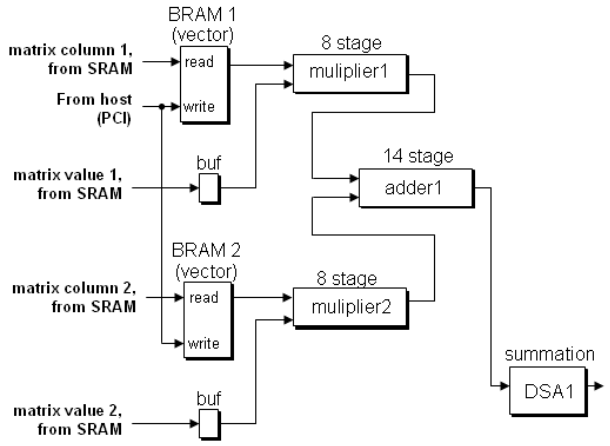

Fig 1. Upper Triangle Architecture

approximately 3 GB/s of the total theoretical memory bandwidth.

In our design we use a slightly modified version of the Compressed Row Storage (CRS) format The CRS 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 indices within *val* and *col* where each row begins, terminated with a value that contains the size of *val* and *col*. Instead of using the *ptr* array, our design assumes the end-of-row information is encoded within the *val* and *col* arrays using zero-termination. We do this for practical reasons as described below. Since our accelerator is designed specifically for symmetric matrices, we assume that only the non-zero matrix values on the diagonal and within the upper triangle of the matrix are represented in the matrix input data.

The purpose of the upper triangle architecture was to compute the partial dot products for only the diagonal and upper triangle of the input matrix:

$$result_{upper}(i) = \sum_{j=ptr(i)}^{j<ptr(i+1)} val(j) \cdot vec(col(j)),$$

where *i* is the row for which the partial dot product is being computed.

Our upper triangle architecture is shown in Figure 1. A copy of the vector is stored on-chip in a two-port 64-bit block RAM for each multiplier. Both of these BRAMs are directly written in parallel by the host using a DMA transaction. Incoming matrix values read from the on-board SRAM are paired with a corresponding value from the vector and sent into a multiplier. Each pair of products are added and their sum is sent into a reduction circuit for accumulation.

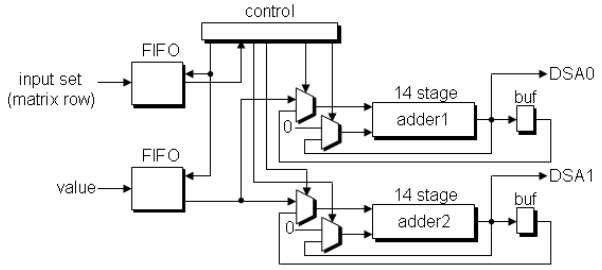

Fig. 2. DSA Reduction Circuit Design from [4].

A. Reduction Circuit Design

Our reduction circuit design is the double-strided adder (DSA) from [4] and shown in Figure 2. The DSA has two adders that independently operate in one of three states: *fill*, *steady state*, or *coalesce*. In the fill state, a new value from the FIFO is added to zero, sending individual inputs into the adder pipeline. This is performed until the adder pipeline fills or until input values for the current input set are exhausted. In the case where the adder pipeline fills and additional input values for the current input set are waiting in the FIFO (i.e. the number of input values exceeds the number of adder pipeline stages), the adder switches to steady-state mode. In this state, subsequent input values are added to the sums being fed back from the adder output. After all input values for the current input set are exhausted, the adder switches to the coalesce state. In this state, the adder does not consume any input values from the FIFO. Instead, the two most recent non-zero values produced by the adder pipeline are fed back into the adder. When two non-zeros values are not available, zeros are routed into the adder instead. The adder stays in this state until all the sums in the adder pipeline have been coalesced into a final output sum.

For an adder with α pipeline stages, an input set of *n* values requires $\alpha\lceil \log_2 \alpha + 1 \rceil - 2$ cycles to coalesce when $n \geq \alpha$, but requires $\alpha\lceil \log_2 n + 1 \rceil - \left(2^{\lceil \log_2 n \rceil} - n\right)$ cycles when $n < \alpha$. For example, for our 14-cycle adders, this means that matrix rows that have at least 28 values per row will require 68 cycles to coalesce, since 2 input values are included with each DSA input. When an adder is coalescing it does not accept new input values from the FIFO, but as long as the other adder is not also in the coalesce state it will take over accepting new values. This is guaranteed if the size of each input set is greater than or equal to the number of coalesce cycles required for the previous read input set. However, if this is not the case, there will be cycles where both adders are in the coalesce state, causing the input FIFO to grow. Since the DSA is only capable of reading one value at a time, there is no time where the size of the FIFO will decrease until the core stops reading new matrix values from the on-board memory. As a result, input matrices having rows with less than 136 non-zero values have the potential to casue the FIFO to overflow. For these cases, we designed a throttler circuit that forces the SSpVxM core to stop reading new matrix values when the FIFO size reaches 90% and resumes when the size reaches 10% (our DSA input FIFO holds 1024 values).

If the DSA throttles the SSpVxM core during operation, this effectively reduces our memory bandwidth and thus the performance of the core. However, this behavior depends on the characteristics of the input matrix, i.e. it will only affect small input matrices or large matrices that are very sparse,

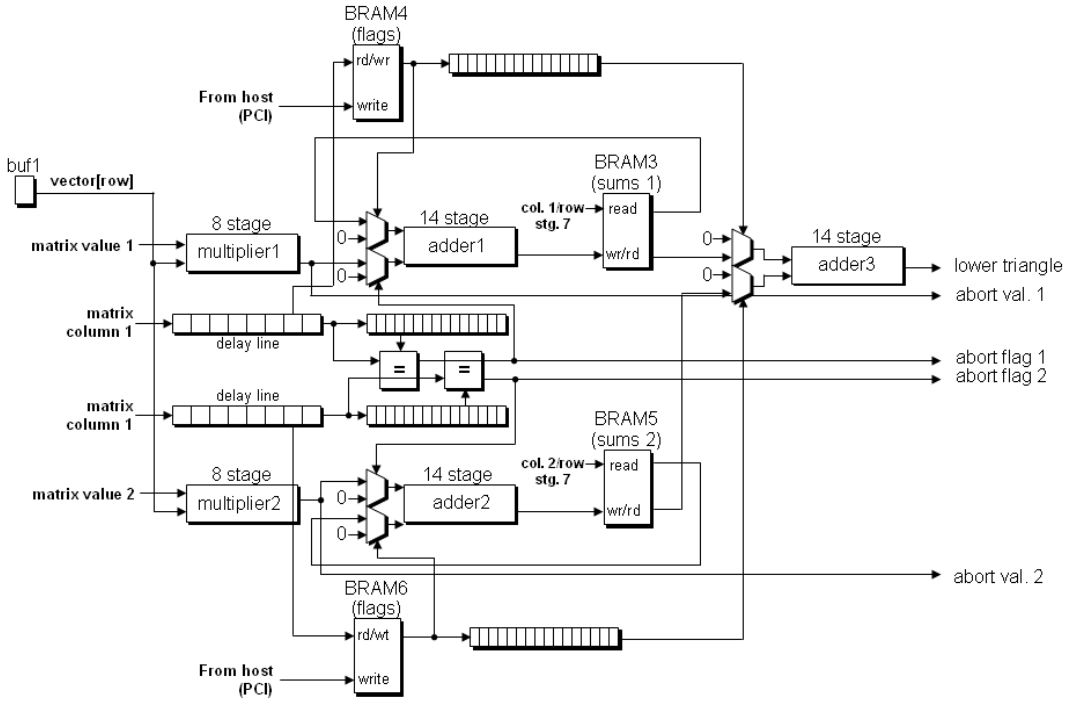

Fig 3. Lower triangle architecture. This design multiplies each incoming matrix value by the vector element corresponding to the current matrix row. These products are accumulated into a BRAM for each multiplier. If a RAW data hazard is detected in either accumulation circuit, the product is "aborted," by being sent to the host and accumulated in software. After each matrix row, the values accumulated in each BRAM are added and sent to the host.

having less than 136 non-zero values per row on average on the upper triangle.

B. Lower Triangle Architecture

The primary contribution of this paper is the lower triangle architecture, which is shown in Figure 3. Computing the vector contributed by the lower triangle in parallel with computing the vector contributed by the upper triangle allows two additional multiplies without requiring any additional memory bandwidth.

The lower triangle architecture treats the row number of each incoming value as its column number. Since the incoming matrix values are grouped by row, the corresponding vector value can be read from one of the on-chip vector BRAMs and stored prior to each matrix row being streamed from on-board memory into the FPGA (shown as BUF1 in Fig. 3). This saves BRAM utilization, as the lower triangle architecture need not include an on-chip copy of the vector--the vector value for each row can be read during the zero-termination for the previous row. This allows for a substantial savings in on-chip memory, but does require that we zero-terminate the values from each row as well as including a leading zero as the first value in the matrix storage.

The stream of products produced by the two lower triangle multipliers must be accumulated into the result location referenced by the column numbers of the incoming matrix values (i.e.the column number is now treated as a row number). As a result, the values to be accumulated will not arrive in a contiguous stream, preventing us from using a reduction circuit. Instead, we use a BRAM for each multiplier to keep track of the accumulated values for each entry in the result vector for the lower triangle, along with a traditional feedback-based adder (BRAM3 and BRAM5 in Fig. 3). There are several complications that arise from this arrangement. The first is the initialization. While BRAMs can be initialized when the FPGA is configured, they cannot be reset after each matrix-vector multiply. To solve this, we use set minimal-width BRAMs (16 bits) as flags to mark whether a given result vector entry has previously been written (BRAM4 and BRAM6 in Fig. 3). The entries of the flag BRAMs are reset by the host as it writes the input vector and are set to one when the first result vector entry is accumulated. These flags control the input to the feedback accumulator.

The feedback-based accumulator is also subject to data hazards caused by the adder latency. This occurs when two input matrix values with the same column number are not separated by the latency of the accumulator adders. To address this problem, we designed a circuit that detects these data hazards, i.e. when a product emerging from a multiplier is to be added to an accumulated value that is currently in the adder pipeline. This happens when two values in the matrix having the same column are not separated by at least $n\alpha$ entries in the matrix memory, where n is the number of multipliers and α is the combined latency of the adder and accumulator memory. This corresponds to 30 in our design. When this happens, the product is "ejected" from the accelerator and instead read by the host. We refer to this as an "abort value". Abort values are added into the final result vector by the host. These data hazards are detected by a 15-stage shift registers for each adder that keeps track of which accumulator entry is to be updated by

the value currently in the adder pipeline. If there is a match with the column associated with the value emerging from the multiplier, the value is aborted and sent to the host.

III. Experimental Setup

We implemented our architecture on a Virtex-2 Pro 100 FPGA on our Annapolis Micro Systems WildStar II-Pro platform. At peak, our architecture performs four double-precision multiplies and six double-precision adds per cycle at 148 MHz (1480 MFLOPS). It is relatively trivial to scale this architecture to accommodate platforms with higher memory bandwidth. We tested the co-processor using a set of real symmetric positive definite matrices (requirements imposed by the CG method) obtained from Matrix Market [5]. We also added a randomly generated, fully populated order 1000 matrix for comparison. This matrix cannot be solved with CG.

Table 1 summarizes our results for a single matrix-vector multiply for each test matrix. As shown in the table, the performance of the co-processor increases with the average number of non-zero values per row for the upper triangle is increased. This is due to the DSA throttling the input speed when its FIFO fills. The "result throughput" indicates the amount of DMA capacity that was used to send the result data to the host memory. This value depends on the percentage of products computed by the lower triangle architecture that are aborted due to a data hazard. This percentage is shown in the last column. As far as we know, none of these multiplies required more result DMA capacity than was available.

Our test matrix is the only matrix that has at least 136 non-zero values per row on average, and thus represents an upper bound for our expected performance. Because of its high density, it also causes the least number of aborted products due to data hazards.

IV. Conclusions

We presented a sparse matrix-vector multiplier design for accelerating the Conjugate Gradient method that exploits matrix symmetry to achiever higher computational parallelism. We have shown that the performance of the design depends heavily on the reduction circuit and its behavior when reducing input sets that are smaller than its latency. In our future work we will replace the DSA reduction circuit with one that exhibits higher adder utilization for matrices with a low average number of non-zero values per row.

References

[1] R. L. Burden, J. D. Faires, "Numerical Analysis 6/e," Brooks/Cole Publishers.

[2] S. Haridas, S. Ziavras, "FPGA Implementation of a Cholesky Algorithm for a Shared-Memory Multiprocessor Architecture," Journal of Parallel Algorithms and Applications, Vol. 19, No. 6, pp 411-426, Dec. 2004.

[3] X. Wang, S. Ziavras, "Parallel Direct Solution of Linear Equations on FPGA-Based Machines," Proc. International Parallel and Distributed Processing Symposium (IPDPS'03).

[4] 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.

[5] NIST Matrix Market, http://math.nist.gov/MatrixMarket, January 2009.

TABLE 1. PERFORMANCE RESULTS FOR MATRIX-VECTOR MULTIPLY. COLUMN "AVERAGE n_z/ROW UPPER TRIANGLE" LISTS THE AVERAGE NUMBER OF NON-ZERO VALUES PER ROW ON THE UPPER TRIANGLE. LOWER VALUES FOR THIS LEAD TO THE DSA THROTTLING THE INPUT, EFFECTIVELY REDUCING THE MEMORY BANDWIDTH.

Matrix	average n_z/row upper triangle	Software time	Software MFLOPS	Co-processor time	Co-processor MFLOPS	Result throughput	% products on upper triangle aborted
BCSSTK10	10.7	294 µs	154 MFLOPS	216 µs	209 MFLOPS	235 MB/s	20%
BCSSTK13	21.4	1923 µs	88 MFLOPS	492 µs	345 MFLOPS	208 MB/s	11%
BCSSTK15	15.4	1430 µs	168 MFLOPS	895 µs	268 MFLOPS	268 MB/s	22%
BCSSTK16	30.2	8812 µs	66 MFLOPS	1343 µs	436 MFLOPS	128 MB/s	4%
BCSSTK17	20.0	5055 µs	172 MFLOPS	2649 µs	329 MFLOPS	181 MB/s	9%
BCSSTK18	6.7	2320 µs	133 MFLOPS	1716 µs	181 MFLOPS	242 MB/s	15%
BCSSTK25	8.7	7844 µs	66 MFLOPS	2571 µs	203 MFLOPS	258 MB/s	20%
S3RMT3M3	19.8	4726 µs	87 MFLOPS	1350 µs	307 MFLOPS	140 MB/s	5%
S2RMT3M1	20.3	2537 µs	172 MFLOPS	1380 µs	316 MFLOPS	149 MB/s	6%
S2RMQ4M1	24.5	9451 µs	55 MFLOPS	1497 µs	352 MFLOPS	245 MB/s	16%
BIGMATRIX	500.5	16168 µs	123 MFLOPS	1731 µs	1155 MFLOPS	14 MB/s	.02%