A Hardware Acceleration Technique for Gradient Descent and Conjugate Gradient

David Kesler, Biplab Deka and Rakesh Kumar
Department of Electrical and Computer Engineering
University of Illinois at Urbana Champaign
dkesler2, deka2, rakeshk@illinois.edu

***Abstract*—Application Robustification, a promising approach for reducing processor power, converts applications into numerical optimization problems and solves them using gradient descent and conjugate gradient algorithms [1]. The improvement in robustness, however, comes at the expense of performance when compared to the baseline non-iterative versions of these applications. To mitigate the performance loss from robustification, we present the design of a hardware accelerator and corresponding software support that accelerate gradient descent and conjugate gradient based iterative implementation of applications. Unlike traditional accelerators, our design accelerates different types of linear algebra operations found in many algorithms and is capable of efficiently handling sparse matrices that arise in applications such as graph matching. We show that the proposed accelerator can provide significant speedups for iterative versions of several applications and that for some applications such as *least squares*, it can substantially improve the computation time as compared to the baseline non-iterative implementation.**

I. Introduction

Several aggressive techniques have recently been proposed to reduce processor power. One technique for reducing processor power consumption is to eliminate design level gaurdbands against worst-case variations [2]. The variation-induced errors in such processors can then be handled using a hardware-based mechanism. Recent work has shown that further savings in power can be achieved by using another approach that handles errors in software, at the algorithmic level [1]. This approach (*application robustification*) involves reformulating applications as stochastic optimization problems and solving them using iterative algorithms such as Gradient Descent (GD) or Conjugate Gradient (CG), which have inherent error tolerance.

Unfortunately, the performance of the iterative versions of many applications may be poor compared to their baseline non-iterative versions. For example, our analysis of the *bipartite graph matching* application from [1] reveals that for a graph with 128 vertices, the iterative algorithm needs 6.5 times the execution time of the baseline non-iterative version of the application to reach a 1% accuracy target in the results. Such high overheads may eliminate any potential energy benefits of handling errors in software. In this paper, to address the issue of high performance overhead of application robustification for several applications, we identify the requirements for accelerating GD and CG based iterative algorithms. Based on that, we present a hardware accelerator design, termed the *solver engine*, and the corresponding software support. Finally, we study the impact of this accelerator on the performance of iterative versions of applications such as bipartite graph matching, maxflow, least squares, and systems of sparse equations.

Compared to previous accelerator designs, our accelerator is better suited for CG and GD based iterative versions of applications due to two major reasons. First, iterative versions of many applications exhibit sparsity patterns that cannot be handled well by traditional accelerators. Our accelerator is supported by a novel sparse matrix format, called the Row Blocked Coordinate List (RBCOO) along with a static scheduling algorithm that allows efficient execution of operations on sparse matrices. Second, our accelerator is designed to accelerate all the different types of linear algebra operations found in the iterative versions of these applications. Traditional accelerators are generally designed to accelerate only one of these operations.

Our contributions in this paper are as follows:

- We present the requirements of an accelerator for accelerating gradient descent and conjugate gradient based iterative algorithms.
- We present a hardware accelerator that is designed according to these requirements. To the best of our knowledge, this is the first dedicated hardware accelerator that has the ability to accelerate multiple linear algebra operations found in these iterative algorithms and also efficiently accelerates sparse matrix operations with software support.
- We also present a novel sparse matrix format that allows efficient execution of sparse matrix operations found in these applications.
- We quantify the performance benefits from the proposed accelerator design and the corresponding software format. We show that the performance gains from using the accelerator makes the execution time of the iterative versions of some applications comparable to or better than that of the baseline versions. Thus it extends the scope of the numerical optimization based application robustification methodology beyond what was previously thought possible.

Note that while the focus of this work is application robustification, the proposed accelerator design and the corresponding software support will be useful for any application that has linear algebra operations, even when it consists of multiple types of operations and involves sparse matrices.The remainder of

978-1-4577-1213-5/11/$26.00

$$\text{while } ||\nabla f(x_i)||_2 > Threshold$$
$$x_{i+1} = x_i + \alpha \nabla f(x_i)$$

Fig. 1. The gradient descent algorithm

this paper is organized as follows. Section II discusses the gradient descent and conjugate gradient algorithms. Section III describes the architecture of the solver engine. Section IV describes the software support necessary for our solver engine including the format of row-blocked coordinate (RBCOO) list. Section V describes how the various linear algebra operations are accelerated in the architecture. Section VI describes the simulation infrastructure. Section VII describes the results of our experiments. Section VIII describes related work. Section IX concludes.

II. Background

In application robustification [1], applications are recast into their equivalent numerical forms that tolerate errors due to reduced gaurdbands in the processor. If x^* is the unknown solution to this numerical problem, then a cost function $f(x)$ is constructed so that $f(x)$ attains its minimum at x^*. Solving this problem then amounts to minimizing $f(x)$.

A. Gradient Descent

Gradient Descent [3], also known as the method of steepest descent, can be used to solve unconstrained optimization problems. The pseudocode for the basic algorithm is in Figure 1. Every iteration of the algorithm consists of calculating the gradient at the current position x, multiplying the gradient by a scaling factor, and adding the result to the current position. Under various assumptions of local convexity of $f(x)$, gradient descent converges to the true minimum. Also, the bulk of computation in Gradient Descent is computing the gradient. The algorithm is robust to errors in this computation, as long as the step size α is monotonically decreasing, errors are independent, and the magnitude of variance of the errors is bounded [1].

For some applications, the conversion can result in a constrained optimization problem. In such cases, the application is converted to an unconstrained optimization problem by converting the constraints into a numerical format defined by a matrix A and a vector b, and minimizing:

$$g(x) = f(x) + \lambda([Ax - b]_+)^2 \quad (1)$$

where $[.]_+$ is defined as max(., 0). λ is chosen to be a sufficiently large scaling factor. Essentially, each row of A defines a constraint on a subset of values of x by forcing the linear combination of the values in x to be less than the corresponding value of b. The gradient of $g(x)$ becomes

$$\nabla g(x) = \nabla f(x) + \lambda A^T [Ax - b]_+ \quad (2)$$

Depending on the optimization problem, A might be sparse or dense. Conversion of non-numeric applications to optimization problems typically results in sparse A matrices. Thus it is essential that an accelerator designed for gradient descent also supports computations involving sparse matrices.

B. Conjugate Gradient

Conjugate gradient [4] is an iterative numerical optimization algorithm which can be used to solve problems of the form $Ax = b$, where A is a NxN symmetric positive-definite matrix. For such problems, conjugate gradient is significantly more powerful than gradient descent as conjugate gradient is guaranteed to converge in at most N iterations. Conjugate gradient can also be used in cases where A is not a symmetric positive definite matrix by defining $A' = A^T A$ and $b' = A^T b$, and then solving $A'x = b'$.

III. Architecture of the Solver Engine

A. Motivation

As is evident from the description of gradient descent and conjugate gradient (Section II) methods, these algorithms contain multiple types of linear algebra operations. These include matrix-matrix multiplications, matrix-vector multiplications, vector operations, and dot product operations. Depending on the application, one of these operations dominate in these algorithms. However, other operations also have significant contributions to the execution time. For example, with a sparse matrix of size 150x150 and sparsity 0.05, we observed that 68.5% of the iteration time of conjugate gradient is spent on matrix-vector product operation, and 27.6% of it is spent on vector operations. The contributions from different types of operations in these applications changes with the sparsity of the matrices and their sizes.

Thus, the solver engine for CG and GD should be general enough to accelerate all of the above mentioned operations. Fortunately, at the lowest level, all these operations are composed of multiplications and additions. Also, the operations on different values can be performed in parallel. This motivates an accelerator design based on multiple processing elements (PE) that operate on individual values and are capable of performing floating point addition and multiplication. Also, to enable computation on data sets that are bigger than the on chip storage the design should use a blocked format for matrices where each matrix is divided into blocks and computation is performed one block at a time.

B. Solver Engine Architecture

The solver engine is integrated onto a general purpose CPU die as shown in Figure 2. It consists of an array of processing elements (PE) each of which consists of a pipelined floating point multiplier and adder in addition to local storage.

Computations on values corresponding to the same row of a matrix are assigned to a single PE. This allows the accelerator to complete computations corresponding to a row without the need for any shared global storage for communication among the multiple PEs. Also, as explained in Section IV-C, the scheduling algorithm assigns computations corresponding to multiple rows to each PE to hide the latency of the adder. Each row assigned to a PE uses the local storage to store its intermediate output during computation.

The solver engine presents a memory-mapped IO interface. A range of addresses corresponds to a set of registers in the

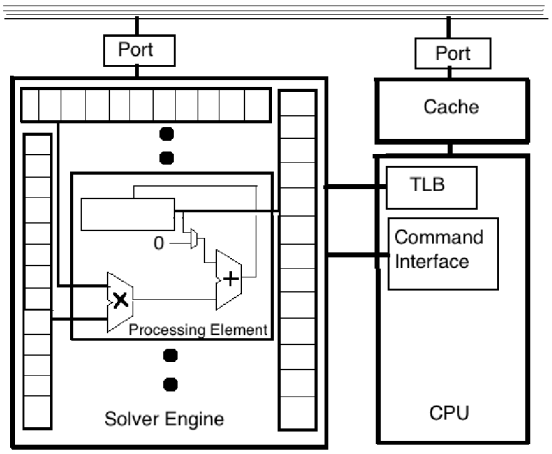

Fig. 2. A general purpose CPU augmented with the solver engine.

solver engine, while one address is used to receive commands. When a command is issued to the solver engine, it performs demand-driven memory accesses for the required data and begins computation while blocking all subsequent IO until it has finished the computation. The solver engine has access to the TLB on the core via the addition of a new read port. In the event that the solver engine suffers a miss in the TLB, the core processes the miss just as if it suffered the miss itself. Also, the solver engine does not include any caches as the computations were found to be memory bandwidth bound with low data reuse. To ensure coherence between the core's data cache and the solver engine, the data associated with a particular computation is flushed from the cache before beginning the computation on the solver engine using a targeted invalidate mechanism (a *flush-all* mechanism is also possible).

Also, for efficient storage of different data types used in different linear algebra operations, the solver engine contains four sets of registers, one each for scalar values, vectors, dense matrices, and sparse matrices. Scalar registers store a single double precision floating point number. Vector registers store a virtual pointer to memory pointing to the start of the vector and the length of the vector. Dense matrix registers store a pointer to memory as well as the width and height of the matrix. Sparse matrix registers contain memory pointers to point to each of the RBCOO arrays, the width and height of the matrix, and the length of both the val array and the $blkIdx$ array as described in Section IV-B.

The solver engine supports four different operations: matrix multiplication ($C = \pm AB$), matrix-vector multiplication ($c = \pm Ab \pm \beta d$), vector operations ($c = \pm \alpha b \pm \beta d$), and dot products ($c = \pm ab$). Sparse matrices are only supported for matrix-vector multiplication. Each instruction consists of a two bit op-code, the source and destination registers, and a few additional bits of information. The instruction has six 4-bit fields for the registers, one each for α, β, A, B, C, and d. Whether A, B, C, and d represent vectors or matrices is inferred from the instruction. Also, since matrix-vector multiplications are handled differently for sparse and dense matrices (as mentioned in Section V-3) , an additional bit is

$$\begin{bmatrix} 0 & 1 & 8 & 9 \\ 2 & 6 & 0 & 0 \\ 0 & 5 & 0 & 3 \\ 0 & 7 & 0 & 0 \end{bmatrix}$$

RBCOO

val: [1 2 6; 8 9; 5 7; 3]
relColIdx: [1 0 1; 0 1; 1 1; 1]
relRowIdx: [0 1 1; 0 0; 0 1; 0]
blockPtr: [0 3 5 7]
blockColIdx: [0 2 0 2]
blockRowPtr: [0 2]

CSR

val: [1 8 9 2 6 5 3 7]
columnIdx: [1 2 3 0 1 1 3 1]
rowPtr: [0 4 6 8]

Fig. 3. An example matrix encoded as CSR and as RBCOO with a 2x2 block size. Semicolons indicate boundaries between blocks for clarity. The RBCOO matrix has not been padded with zeros or reordered for scheduling on the hardware.

used to flag if A is sparse. This enables the solver engine to handle dense and sparse matrices differently during matrix-vector multiplication. There are two additional bits for each source register (A, B, and D) to indicate whether or not the operand should be clipped. Finally, we have two bits, one each to control the $\pm$ terms in the operations.

The details of how computations corresponding to the different types of operations mentioned above are executed on the solver engine are presented in Section V.

IV. Software Support for the Solver Engine

A. Motivation

As mentioned in Section III-B, computations from a particular row are assigned to a single PE in the solver engine. Thus, multiple PEs can be used each cycle if we can bring in values from multiple rows to the solver engine. Unfortunately, when sparse matrices are stored in popular formats such as Compressed Sparse Row (CSR) (or Compressed Sparse Column (CSC)), the values from the same row (or column) have to be stored consecutively and reordering of values for efficient utilization of the PEs is not possible. Therefore, sparse matrices that exhibit only a few non-zero values in one dimension (as is common in applications such as graph matching) cannot be efficiently scheduled without complex logic if stored using these formats. This motivates us to explore a novel format termed the *Row Blocked Coordinate List Format* (RBCOO) that allows us to arbitrarily reorder values within a block.

B. Row Blocked Coordinate List Format

We first partition a sparse matrix into multiple blocks of size S_jxS_j. The RBCOO matrix format then represents the matrix using six arrays of data. The first array, val, is an array which stores all the non-zero values in the matrix. The row and column indices of each value relative to the upper left corner of the block are stored in $relColIdx$ and $relRowIdx$. The remaining three arrays correspond to storing pointers to the start of each block in a CSR format. The $blockPtr$ array stores an index into val, $relColIdx$, and $relRowIdx$ which points to the start of a block. The block column indices are stored in $blockColIdx$. The ith value of $blockColIdx$ corresponds to the first column containing a non-zero value in the ith block. The $blockRowPtr$ array contains an index into the $blockPtr$

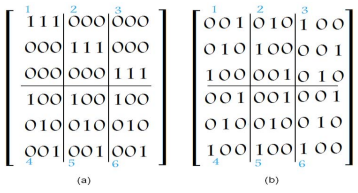

Fig. 4. The constraints for bipartite graph matching with 3 vertices in each set. (a) In the unshuffled form blocks 1, 2, and 3 can use only a single PE at a time and computation for these blocks take $3 * La$ cycles (b) After shuffling the columns, each block can use 3 PEs at a time and takes La cycles for computation

array, indicating the blocks that correspond to the start of a new row of blocks. The RBCOO format of a sparse matrix is shown in Figure 3, along with its representation in CSR format. Note that the RBCOO format allows values in a block to be reordered arbitrarily during scheduling (as explained in Section IV-C) as both the column and row indices are stored in separate arrays. This is not possible for the CSR format as values from a row are arranged consecutively and so their row indices are not stored in a separate array.

C. Scheduling

As mentioned earlier, the RBCOO format allows reordering of values within a block. This enables efficient use of multiple PEs even after assigning values from the same row to a single PE. In our implementation, this reordering is done in a scheduling step in the CPU after the conversion of the sparse matrix into the RBCOO format. Note that if L_a denotes the adder latency in cycles i.e. the number of pipelined stages in the adder, the scheduling algorithm has to make sure that values from the same row enter the PEs atleast L_a cycles apart. In order to hide this latency, values from multiple rows must be scheduled to the same PE.

Our implementation uses a greedy scheduling algorithm. Each cycle of scheduling consists of choosing p values, one for each PE. Row i of the input block is statically assigned to PE $i\%p$. For each PE, the rows assigned to that PE are examined and the row with the largest number of values remaining and that has not scheduled any values in the previous L_a cycles is chosen. If no row can be chosen (because this PE has scheduled all its non-zero values in the current block, or because all its remaining non-zero values are in rows which have been scheduled in the previous L_a cycles), the algorithm inserts a padded zero. We show in Section VII-C that the overhead of the extra padded zeros is small for large matrices.

Note that the number of rows assigned to a PE depends on the total storage available in the solver engine. To accumulate its result during computation, each row assigned to a PE needs local storage corresponding to the size of a single value. If storage corresponding to S_i values is present in the solver engine ($\frac{S_i}{p}$ in each PE), only $\frac{S_i}{p}$ rows can be scheduled to each PE.

D. Column Shuffling

Scheduling does well if the nonzero values in the matrices are evenly distributed among the blocks and also among the rows in each block. Unfortunately, for some applications, the constraint matrix is not well behaved and results in uneven distribution of non-zero values amongst the rows in a block. In such cases, shuffling the columns can provide better scheduling efficiency. An example of column shuffling is shown in Figure 4. The column numbers after shuffling is given by $(n * i)\%p$ where i is the original column number, p is the total number of columns and n is a prime number. Note that the values in the corresponding vector are also shuffled accordingly.

V. Implementation of Operations

Below we describe how computations corresponding to each of the operations mentioned in Section III-B are performed in the solver engine.

1) Vector Operations: During vector operations ($c = \alpha b \pm \beta d$ where α and β are scalars and b, c, and d are vectors), an Si sized block of b is brought into the engine and $\frac{Si}{p}$ values are assigned to each PE. The value of α is broadcast to the second input of all PEs . The output of the multiplier is summed with 0 and is stored in the local storage. Once Si values of αb have started being calculated, the corresponding values of d are brought in the solver engine while β is broadcast to all PEs. The adder in the PE adds the newly calculated βd with the previously stored αb.

2) Sparse Matrix-Vector Multiplication: The βd portion of the calculation of sparse matrix vector multiplications ($c = Ab \pm \beta d$ where A is a sparse matrix) is handled similarly as the first half of a vector operation. To perform the matrix-vector computation on a block of the sparse matrix, the solver engine loads the Sj values of b corresponding to the block. It then begins bringing in p shuffled tuples (value, column number, row number) of the block and assigns them to the appropriate PE. The column index is used to index into the buffered Sj values of b to be sent to the appropriate PE. The row index is buffered along with the value itself and passed to the PE. Each row assigned to a PE uses a different location of the local storage to store the adder outputs corresponding to its values. This output is used by the adder when it performs computation on the next values from this row in the future. When the solver has processed all values from the current block, it proceeds to the next block simply by loading a new Sj chunk of b.

3) Dense Matrix-Vector Multiplication: Dense matrix-vector multiplication proceeds similarly to sparse matrix-vector multiplication. The primary difference is that instead of arbitrarily indexing into the buffered Sj values of b, every PE accesses one particular value of b.

4) Dense Matrix-Matrix Multiplication: Dense matrix-matrix multiplication ($C = AB$) utilizes a blocked rank-one update algorithm [5]. Calculation is done on an Si sized block of C at a time, of height $i = p$ and $j = \frac{Si}{p}$. Thus, each PE computes a j sized row of C.

To calculate an ixj block in C (call the current block pC), i rows from A (call it pA) and j columns from B (call it pB) are needed. pC is initialized to all 0. The outer product of the kth column of pA and the kth row of pB is calculated. This results in an ixj panel which is summed with the current

value of pC. Once all N columns of pA and rows of pB are multiplied, the block is complete and can be output while work begins on the next block.

In order to perform the computation mentioned above, the solver engine reads in a column of $i = p$ values from pA and buffers them. The appropriate row from pB is streamed in one value at a time and broadcast to each PE. Once the entire row of j values from pB is read, computation begins on the next column of pA and the next row of pB.

5) Dot Product: Dot product operations are handled in three stages. In the first stage, p values from the first vector are passed to the PEs, along with the corresponding values from the second vector. It is followed by the next p values and so on till all values have been read in. The output of the adder in each PE is directly routed back into its input and local storage is not used. At the end of this stage, the multiplier has finished multiplying all values input into the PE. At this point, there are La values in each PE which need to be summed up.

For the second stage, hardware support is needed in the form of an adder tree arrangement among the adders in various PEs in the solver engine. The output of each adder is routed to one of the inputs of the adder above it in the adder tree. At the end of this stage La values remain, all in the final PE's adder pipeline. In the third stage of the computation these values are added in the final PE which has extra control logic to add appropriate delays during this computation.

Note that the additional hardware support required by the dot-product operation has very low overhead. The adder tree is implemented by the addition of a single multiplexer in each PE. The extra control logic for the third stage needs to be present in only one PE and it is not very complex as the adder latency is small (L_a= 4 in our final design presented in Section VI-A).

6) Additional Hardware Support: As described in Section II-A, calculations of the form $[Ax - b]_+$, where $[.]_+$ means $max(., 0)$ may arise in gradient descent based computations. This is supported by having an additional multiplexer per PE which examines the sign bit of the incoming value. If the sign bit is negative, the multiplexer passes 0; otherwise it passes the value through.

VI. Methodology

A. Synthesis Results

We assume that the memory bandwidth available to the solver engine is 32 GB/s which is in line with the current top of the line desktop processors. As is common in parallel designs, this bandwidth restriction creates a tradeoff between the number of processing elements, p and the frequency of operation, f. In our solver engine design, f, in turn, determines the adder latency in cycles, L_a, and the total number of values that needs to be stored in the solver engine, S_i.

Tables I, summarize the area, power, scheduling overhead, and execution time of one iteration of conjugate gradient corresponding to different designs synthesized using the Nangate 45nm Open Cell Library. Note that diagonals of equal performance appear representing a range of area, power options for the same performance. For subsequent evaluations, we choose the design point (details in Table II) with the lowest area×power×performance product. Note that this design has an area of $0.83mm^2$ and a power consumption of $0.16W$ which is well below that of a general purpose CPU core.

S_i \ p	64		32		16		8	
256	1.69	27.49	1.12	18.92	0.83	12.67	0.68	7.1
	0.1	106.27	0.12	94.12	0.16	87.76	0.24	84
512	1.91	19.97	1.33	14.02	1.04	8.68	0.9	5.37
	0.1	96.32	0.12	87.67	0.16	83.14	0.25	80.61
1024	2.34	16.53	1.76	9.39	1.47	5.94	1.33	3.91
	0.1	89.22	0.13	83.88	0.17	80.43	0.27	77.94
2048	3.2	11.19	2.63	6.67	2.34	4.15	2.19	2.23
	0.11	84.26	0.14	79.98	0.19	77.78	0.31	76.49

TABLE I
AREA IN mm^2 (TOP LEFT), PERCENTAGE OVERHEAD OF PADDED ZEROS (TOP RIGHT), POWER CONSUMPTION IN W (BOTTOM LEFT), AND EXECUTION TIME IN μS OF ONE ITERATION OF CG ON A SPARSE MATRIX OF SIZE 2048 AND SPARSITY 0.52 (BOTTOM RIGHT) FOR DIFFERENT DESIGN POINTS.

CPU		Solver Engine	
Frequency	3.33GHz	Frequency	250MHz
L1 I Cache	32kB	Processing Elements	16
L1 D Cache	64kB	Adder/Multiplier Latency	4
L2 Cache	2MB	S_i, S_j	256
Execution Width	4	S_j	256
Issue Width	4	Memory Bandwidth	32GB/s

TABLE II
CHARACTERISTICS OF THE CPU AND SOLVER ENGINE USED IN OUR EVALUATIONS

B. Simulation Infrastructure and Methodology

In our experiments, the performance of the CPU core was evaluated by running benchmarks on the M5 simulator in system emulation mode. The characteristics of the general purpose out of order core are in Table II. M5 was augmented to perform functional simulation of the solver engine and syscalls were added which performed the requested operations instantaneously. The functional solver engine simulator retrieves data directly from memory. The M5 simulator was also augmented to allow write back and invalidation of data present in the CPU's data cache.

In order to measure the performance of the solver engine, analysis routines were used that had the same interface as the corresponding algorithms, gradient descent or conjugate gradient. When called by the benchmarks, the analysis routines convert any sparse matrices into RBCOO format and measure the overhead of the padded zeros and the number of cycles the solver engine is used per call and per iteration.

C. Benchmarks

Bipartite graph matching, maxflow, least squares, and system of sparse equations (as described in [1]) are used as benchmarks for our experiments. Gradient Descent is used as the solver for graph matching and maxflow whereas conjugate gradient is used for least squares and systems of sparse equations.

VII. Results

A. Performance

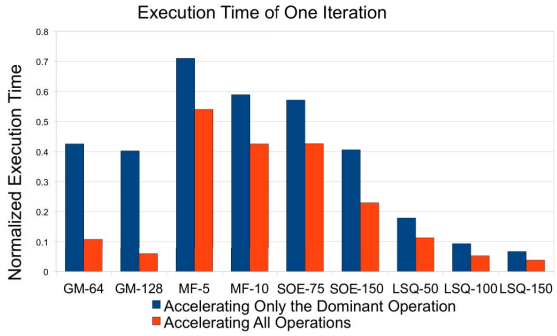

Fig. 5. Execution time of a single iteration of accelerated versions of applications (normalized to the execution time of a single iteration without acceleration) for graph matching with 64 and 128 vertices, maxflow with 5 and 10 vertices, system of equations with sparse matrices of size 75 and 150, and least squares with matrix sizes of 50, 100 and 150.

We evaluated the time taken for a single iteration of the iterative versions of the applications for different levels of acceleration. Figure 5 shows the execution time of a single iteration when only the dominant operation is executed and when all supportable operations are executed on the solver engine (normalized to their execution on a CPU with no acceleration).

We see that the solver engine successfully reduces the computation time of the iterative versions of all targeted applications. Further, additional gains are seen in all applications by accelerating all operations as opposed to accelerating only the dominant operation. Thus, the emphasis on accelerating all types of linear algebra operations present in gradient descent and conjugate gradient, while designing the solver engine, is justified. Graph matching and least squares achieve more than 96% reduction in execution time.

B. Comparison to Traditional Algorithms

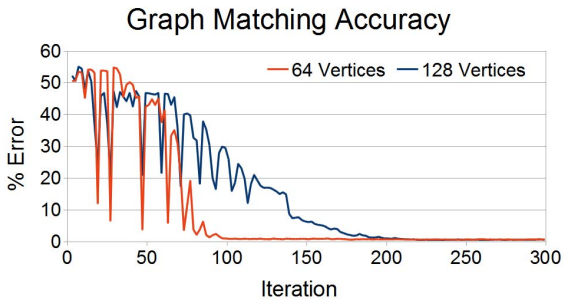

Fig. 6. The error in the match weight of graph matching during gradient descent iterations.

We also evaluated the performance of the iterative versions of graph matching, maxflow, and least squares with various levels of acceleration against their respective baseline implementations.

Figure 6 shows the error in the total matching weight as the number of iterations of graph matching was increased while accelerating all types of operations. We find that the error settles to 1% error in around 100 iterations with 64 vertices whereas it takes around 200 iterations to reach similar levels of accuracy with 128 vertices. In our experiments, we further observed that the algorithm needs orders of magnitude increase in the number of iterations to appreciably increase accuracy beyond this level.

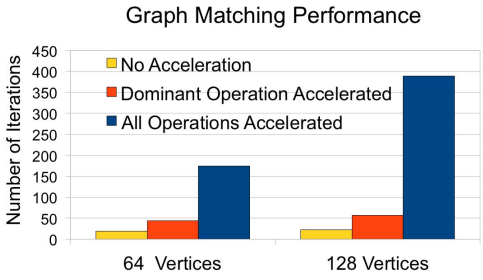

Fig. 7. The number of iterations of gradient descent completed during the execution time of the baseline algorithm for graph matching.

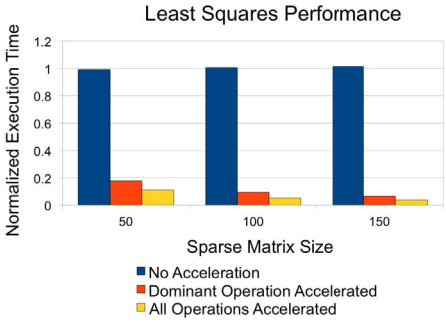

Fig. 8. Normalized execution time of least squares with varying levels of acceleration

To compare the performance of the iterative implementation of graph matching with the baseline, we show the number of iterations of graph matching that can execute in the same time as that of the baseline in Figure 7. We see that it is possible to execute around 170 and 380 iterations for 64 and 128 vertices respectively. Thus the computation time of the iterative version of graph matching on the solver engine with a 1% error target is significantly lower (41.2% and 47.3% lower for 64 and 128 vertices respectively) than that of the baseline non-iterative implementation.

Figure 8 shows the execution time of solving least squares using conjugate gradient with varying levels of accelerator support (normalized to the execution time of the baseline implementation based on Cholesky decomposition). We see that the execution time of the unaccelerated version of gradient descent is almost equal to that of the baseline. However, we are able to complete the computation in significantly lower (89%, 94.8%, and 96.2% lower for matrices of size 50, 100 and 150 respectively) time as compared to the baseline by accelerating all operations using the solver engine.

As shown in Figure 5, the solver engine successfully accelerates the iterative version of maxflow. However, we found that this acceleration is not sufficient to reach performance

levels of the baseline version.

The above results show that we do not necessarily have to sacrifice performance in order to utilize the potential energy benefits of application robustification.

C. Software Support Overhead

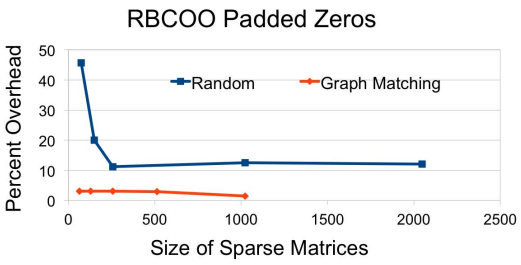

Fig. 9. Storage overhead of padded zeros when scheduling random sparse matrices of sparsity 0.05 and the sparse matrices used by graph matching (with column shuffling enabled).

As mentioned in Section IV-D, sparse matrices in graph matching are highly irregular. Therefore, the RBCOO format suffers significant storage overheads due to extra zeroes padded for scheduling. These overheads were found to be 690% and 1490% for graphs with 64 and 128 vertices. Column shuffling lowers the overhead to 3.1% for both cases. This translates to a 14% decrease in the number of padded zeros relative to a CSR implementation and thus, justifies the need for the RBCOO format to achieve better scheduling of sparse matrices on the solver engine.

We also examined a set of sparse matrices with sparsities between .05 and .06 and calculated the zero padding overhead. Figure 9 shows the results of this test for a range of matrix sizes, from 75x75 to 2048x2048. We see that the overhead can be substantial at small matrix sizes. However, larger matrices provide more flexibility in scheduling and result in just over 10% padded zeros.

The overhead of scheduling for the RBCOO format on the CPU was also evaluated. For graph matching, scheduling takes 3.04 ms and 18.07 ms for 64 and 128 vertices. This is equal to the execution time of the baseline algorithm for 64 nodes (3.37 ms), and around 60% of the execution time with 128 nodes (30.6 ms). Thus, the scheduling overhead is a concern for small data sets as it lowers the performance gains achieved by accelerating the iterative version of graph matching. Note that it might be possible to lower this overhead by using other scheduling algorithms.

VIII. Related Work

A. Sparse Matrix Formats

Hierarchical sparse matrix (HiSM) is a matrix format designed for execution on machines with vector instructions [6]. HiSM stores explicit pointers to point to blocks which may be in arbitrary locations in memory whereas RBCOO stores one contiguous array of non-zero elements and uses indices to point to the start of blocks.. Additionally, blocks in RBCOO may start at arbitrary indices while blocks in HiSM always start at fixed intervals. This can potentially reduce the number of blocks since a pattern of non-zeros which may only be n blocks wide may not align evenly with the block boundaries.

Jagged Diagonal Storage (JDS) is a matrix format which features efficient sparse matrix-vector multiplication on vector processors [7]. The ordering created by JDS is identical to the ordering caused by RBCOO scheduling if the number of rows is equal to the height of the matrix. RBCOO will result in better scheduling, however, if the matrix is taller. In JDS, an entire column must be read, using one value from each row, before the next value in a row may be used. RBCOO only requires an interleaving of accesses to the same row equal to the adder latency.

Row Blocked CSR (RBCSR) is a matrix format similar to RBCOO, differing only in that the submatrices are stored in traditional CSR format instead of as a coordinate list [8]. In RBCSR the entire submatrix is buffered on the accelerator hardware. RBCOO on the other hand, was designed so that the submatrix could be streamed in without buffering due to the limited amount of memory available for buffering on a CPU. The downside of RBCOO is that it will have a larger memory footprint due to need for storage of explicit row indices for each value.

B. Accelerators

The FPGA based SPMV design presented in [9] is unsuitable for our purpose because of two main reasons. First, it requires that A and b be stored in their entirety on chip. While this is acceptable on an FPGA, it is not suitable for our design where chip area is at a premium. Second, it requires that each row have at least p non-zero values or the dot product unit will be largely empty resulting in highly sub-optimal performance on matrices common in our problems such as graph matching which feature a small constant number of non-zero values per row.

The FPGA based accelerator design presented in [8], while similar to the one presented in this paper, is not suitable for our purposes for several reasons. First, it requires the entire submatrix be buffered on chip. Secondly, for efficient operation, it requires there to be more than L_a values per row of the submatrix. Lastly, it requires the use of a centralized adder tree and result BRAM. This not only presents a high cost (due to the complex routing and arbitration logic required), but can potentially become a bottleneck to scalability. Another Sparse Matrix Vector Multiplication (SMVM) implementation for FPGAs was presented in [10].Unlike our implementation, the implementation in [10] idoes not support other linear algebra operations.

GPUs generally outperform CPUs on sparse matrix vector multiplication, largely due to their higher peak memory bandwidth compared to general purpose CPUs [11], [12]. However they do not make the most efficient use of peak floating point capacity. For example single precision floating

point performance on a GTX280 was determined to be 36 GFLOP/s, well below the GPUs peak of 933 GFLOP/s [11]. The accelerator design present here, by comparison, is capable of achieving a much higher fraction of peak performance. This, combined with its inherent low power design, will result in much higher performance per watt.

In contrast to sparse matrix operations, dense matrix multiplication and matrix vector multiplication map well onto specialized hardware and GPUs. The largest fraction of work has focused on matrix multiplication on FPGAs [13], [14], [15], [16], [17], [18]. Other works have also considered matrix-vector multiplication and dot products [19] on FPGAs, or matrix multiplication on GPUs [20]. However, these designs cannot be used for the sparse matrices that arise in the applications targeted in this work.

IX. Conclusions

We address the issue of poor performance of iterative versions of applications during application robustification, by presenting a novel accelerator design that is integrated on a CPU die with an acceptable area and power overhead. This accelerator targets all types of linear algebra operations that constitute the algorithms used in application robustification, namely, gradient descent and conjugate gradient.

Since sparse matrices are common in iterative versions of applications, our design is supported by a novel storage format (RBCOO) for sparse matrices that makes it possible to use the same hardware for efficient sparse matrix operations. We show that the storage overhead of this format is acceptable for applications such as graph matching.

Our results show that significant performance benefits can be achieved compared to traditional accelerators which target single linear algebra operation. Applications such as graph matching and least squares showed more than 96% reduction in their execution times after acceleration. The benefits of acceleration were enough to lower the computation time of the iterative versions of some applications compared to their baseline versions. Thus, the proposed accelerator design makes it possible to derive energy benefits of application robustification from a larger set of applications than was previously thought possible. Also, while the focus of this work is application robustification, the proposed accelerator design and the corresponding software support will be useful for any application that has linear algebra operations, even when it consists of multiple types of operations and involves sparse matrices.

X. Acknowledgements

The authors would like to thank Joseph Sloan and the anonymous referees for their valuable feedback. This work was supported in part by NSF and GSRC.

References

[1] J. Sloan, D. Kesler, R. Kumar, and A. Rahimi, "A numerical optimization-based methodology for application robustification: Transforming applications for error tolerance," in *40th IEEE/IFIP International Conference on Dependable Systems and Networks, 2010*, July 2010.

[2] T. Austin, D. Blaauw, T. Mudge, and K. Flautner, "Making typical silicon matter with razor," *Computer*, vol. 37, pp. 57–65, 2004.

[3] J. A. Snyman, *Practical Mathematical Optimization: An Introduction to Basic Optimization Theory and Classical and New Gradient-Based Algorithms*. Springer Publishing, 2005.

[4] M. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *J. Research Natl Bureau of Standards*, vol. 49, no. 6, 1952.

[5] V. B. Y. Kumar, S. Joshi, S. B. Patkar, and H. Narayanan, "Fpga based high performance double-precision matrix multiplication," in *VLSID '09: Proceedings of the 2009 22nd International Conference on VLSI Design*. Washington, DC, USA: IEEE Computer Society, 2009, pp. 341–346.

[6] P. Stathis, S. Vassiliadis, and S. Cotofana, "A hierarchical sparse matrix storage format for vector processors," in *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, 22-26 2003, p. 8 pp.

[7] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. D. Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Philadelphia, PA: SIAM, 1994.

[8] J. Sun, G. Peterson, and O. Storaasli, "Sparse matrix-vector multiplication design on fpgas," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, 23-25 2007, pp. 349 –352.

[9] G. R. Morris and V. K. Prasanna, "Sparse matrix computations on reconfigurable hardware," *Computer*, vol. 40, no. 3, pp. 58–64, 2007.

[10] M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for fpgas," in *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, 2005.

[11] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, NVIDIA Technical Report NVR-2008-004, Dec. 2008.

[12] M. M. Baskaran and R. Bordawekar, "Optimizing sparse matrix-vector multiplication on gpus," IBM, IBM Research Report RC24704, Apr. 2009.

[13] L. Zhuo and V. K. Prasanna, "Scalable and modular algorithms for floating-point matrix multiplication on reconfigurable computing systems," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 4, pp. 433–448, 2007.

[14] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point fpga matrix multiplication," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*. New York, NY, USA: ACM, 2005, pp. 86–95.

[15] S. Rousseaux, D. Hubaux, P. Guisset, and J.-D. Legat, "A high performance fpga-based accelerator for blas library implementation," in *RSSI'07: Proceedings of the Third Annual Reconfigurable Systems Summer Institute*, July 2007.

[16] N. Dave, K. Fleming, M. King, M. Pellauer, and M. Vijayaraghavan, "Hardware acceleration of matrix multiplication on a xilinx fpga," in *MEMOCODE '07: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 97–100.

[17] S. M. Qasim, S. A. Abbasi, and B. A. Almashary, "Hardware realization of matrix multiplication using field programmable gate array," in *MASAUM Journal of Computing*, vol. 1, August 2009, pp. 21–25.

[18] C. Sajish, Y. Abhyankar, S. Ghotgalkar, and K. Venkates, "Floating point matrix multiplication on a reconfigurable computing system," in *Proceedings of the International Conference on High Performance Computing and Applications*. Springer Berlin Heidelberg, 2005, pp. 113–122.

[19] L. Zhuo and V. K. Prasanna, "High-performance designs for linear algebra operations on reconfigurable hardware," *IEEE Trans. Comput.*, vol. 57, no. 8, pp. 1057–1071, 2008.

[20] S. Barrachina, M. Castillo, F. Igual, R. Mayo, and E. Quintana-Orti, "Evaluation and tuning of the level 3 cublas for graphics processors," in *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, april 2008, pp. 1 –8.