A HARDWARE ACCELERATION TECHNIQUE FOR GRADIENT DESCENT AND CONJUGATE GRADIENT

BY

DAVID ROBERT KESLER

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Adviser:

Assistant Professor Rakesh Kumar

ABSTRACT

Gradient descent, conjugate gradient, and other iterative algorithms are a powerful class of algorithms; however, they can take a long time for convergence. Baseline accelerator designs feature insufficient coverage of operations and do not work well on the problems we target. In this thesis we present a novel hardware architecture for accelerating gradient descent and other similar algorithms. To support this architecture, we also present a sparse matrix-vector storage format, and software support for utilizing the format, so that it can be efficiently mapped onto hardware which is also well suited for dense operations. We show that the accelerator design outperforms similar designs which target only the most dominant operation of a given algorithm, providing substantial energy and performance benefits. We further show that the accelerator can be reasonably implemented on a general purpose CPU with small area overhead.

ACKNOWLEDGMENTS

I would like to thank my adviser, Professor Rakesh Kumar, for his guidance, his understanding, his advice, and his ceaseless efforts to push me to improve myself. I would like to thank Joseph Sloan for laying the groundwork for this work and for his continued input over the course of its execution. I would also like to thank Nicolas Zea and John Sartori for their guidance and support during my time at the University of Illinois. Lastly I would like to thank my parents for their unwavering dedication.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

Gradient descent, conjugate gradient, and other numerical optimization algorithms constitute a powerful class of applications which are heavily used in scientific and numerical computing. Due to the heavily numeric nature of these algorithms, they can take advantage of specialized hardware which can provide both performance and energy benefits. Traditionally this has been exploited via the use of reconfigurable hardware such as FPGAs or the use of hardware designed for highly parallel floating-point computation such as GPUs. While these approaches have been very promising, further substantial energy gains can be made through the use of hardware specifically designed to support these numerical optimizations. Furthermore, traditional accelerators lack adaptability in terms of the operations they can execute and the sparsity patterns of sparse matrices they can effectively handle. (Of particular interest are recent investigations into the robustness benefits of transforming non-numeric algorithms into numerical optimization problems which results in matrices which are ill-suited to traditional hardware acceleration techniques [1].)

We propose a novel architecture, termed the "solver engine," which is capable of executing a number of linear algebra operations to support numerical algorithms utilizing dedicated low power hardware. Because it is designed to be integrated onto a die along with a general purpose CPU, it will be more area limited than FPGA based accelerators. Thus it must work with buffer space which comprises an acceptably small amount of die area. The solver engine design will also be fixed at fabrication time, unlike an FPGA. As such, it must be able to support a variety of algorithms and operations since it cannot be reconfigured on demand.

To support this architecture, we use a novel sparse matrix-vector multiplication format, row blocked coordinate list, along with a static scheduling algorithm which provides several benefits. Firstly, it enables the efficient exe-

cution of sparse matrix-vector multiplication with matrices which have sparsity patterns which cannot be handled by traditional accelerators. Secondly, it is able to be easily executed on hardware which is also able to efficiently execute dense matrix and vector operations. This allows it to execute a variety of operations while sharing the majority of its hardware (including all the floating point hardware) without the need for reconfigurable elements. Finally, it addresses several general flaws in traditional accelerators such as the use of shared, centralized structures.

The remainder of the thesis is organized as follows. Chapter 2 examines gradient descent and conjugate gradient, describes the requirements that those algorithms place on the solver engine, and motivates the need to support a variety of operations. Chapter 3 describes the system architecture of a CPU with a solver engine integrated onto the chip, the architectural layout of the solver engine itself, and the algorithms it uses to perform each of its operations. Chapter 4 lays out the software support necessary for the use of the solver engine, including the format of row-blocked coordinate list and the way in which sparse matrices are statically scheduled on the solver engine. Chapter 5 demonstrates how the specific characteristics of the solver engine used in our experiments were derived, describes the simulation infrastructure, and examines the benchmarks used in our experiments. Chapter 6 describes the results of our experiments. Chapter 7 describes related work in sparse matrix formats and hardware linear algebra accelerators. Chapter 8 concludes.

CHAPTER 2

BACKGROUND AND MOTIVATION

Conjugate gradient [2], gradient descent [3], and other iterative optimization algorithms are a powerful class of algorithms. Of particular interest is their inherent error tolerance [1]. All iterative optimization algorithms work, in general, by defining a function $f(x)$ to be minimized, choosing a starting location for x, iteratively choosing a direction, and then moving x along the chosen direction with a certain step size. It is this behavior that provides their innate error tolerance, because as long as the errors are bounded, the algorithms can still make forward progress each iteration (technically, regression is possible in some iterations as long as the net progress is towards the correct solution). The presence of errors does have negative impact on the speed of convergence. Also, errors may weaken the guarantee of convergence for algorithms such as conjugate gradient.

In this chapter, we discuss the construction of the gradient-based iterative optimization algorithms and motivate the design requirements of the *solver engine*, the hardware accelerator for such algorithms.

2.1 Gradient Descent

Also known as the method of steepest descent, gradient descent can be used to solve arbitrary unconstrained linear programs. The goal is to minimize a function $f(x)$ by moving in the direction opposite and proportional to the gradient, $\nabla f(x)$. The pseudocode for the basic algorithm is in Figure 2.1. 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. The algorithm can also be run for a fixed number of iterations, but there is no theoretical guarantee on the accuracy of the result.

$$\text{while } ||\nabla f(x)||_2 > Threshold$$
$$x = x + \alpha \nabla f(x)$$

Figure 2.1: The gradient descent algorithm.

Linear programming is p-complete and is, therefore, capable of solving a large class of applications [4]. Applications can be converted into linear programs by defining a cost function $c(x)$ and a set of constraints such that $c(x)$ will be minimized when x is the correct solution. In order to solve this linear program, we need to convert it from a constrained optimization problem to an unconstrained optimization problem. We do this by converting the constraints into a numerical format defined by a matrix A and a vector b, and minimizing

$$f(x) = c(x) + \lambda([Ax - b]_+)^2 \tag{2.1}$$

where $[.]_+$ is defined as $\max(., 0)$ and 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 f becomes

$$\nabla f(x) = \nabla c(x) + \lambda A^T [Ax - b]_+ \tag{2.2}$$

As will be clear in later discussion, the performance of the solver engine depends greatly on the sparsity of matrix A. Formulations of numeric applications yield A that may be dense or sparse. Converting non-numeric applications into linear programs typically results in sparse A matrices.

Without errors, gradient descent-based algorithms are guaranteed to converge to the correct answer as long as $f(x)$ is convex. The function $f(x)$ can often be made convex even for non-numeric applications by constraining the problem only partially. For example, we know that a correct solution vector x for the graph matching problem will result in every value x_i within x being either 0 or 1. Therefore, our constraint may simply require that $x_i \geq 0$ and $x_i \leq 1$. Since we can determine from the problem formulation that f will be minimized only if $x_i = 0$ or $x_i = 1$ for all i, we will eventually converge to a correct solution. With errors, convergence can still be guaranteed for gradient descent-based problems as long as the step size α is monotonically

decreasing, errors are independent, and the magnitude of the variance of the errors is bounded [1].

Several optimizations are possible for gradient descent. An optimization of particular interest exploits the regularity of the constraints. Constraints that require that all elements of x must be above (or below) a certain value are fairly common. When expressed in matrix form, such constraints result in sections of A which are uniform and diagonal. The structure in A can then be exploited to optimize performance/memory bandwidth. For example, $A^T[Ax-b]_+$ (Equation 2.2) can be transformed into $A_1^T[A_1x-b_1]_+ + A_2^T[A_2x-b_2]_+$ where A is a vertical concatenation of A_1 and A_2, and b is a vertical concatenation of b_1 and b_2. Now, if A_1 is diagonal, and A_1 and b_1 are both uniform (which they may be for several problems because of the nature of their constraints), we can replace A_1 and b_1 by a scalar each. This may result in significantly lower memory bandwidth for computing $A^T[Ax-b]_+$. Similarly, if b_2 were uniform, we could simply use a single scalar instead of a vector.

More details on gradient descent can be found in [3].

2.2 Conjugate Gradient

Conjugate gradient is an iterative numerical optimization algorithm which can be used to solve problems of the form $Ax = b$, where A is an 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 N iterations. Conjugate gradient can also be used in cases where A is not a symmetric positive definite matrix by defining $A' = A^TA$ and $b' = A^Tb$, and then solving $A'x = b'$. The resulting vector x minimizes $||Ax-b||_2$ (note that this is the goal of the least squares problem).

Figure 2.2 displays the pseudocode for conjugate gradient. While conjugate gradient is more powerful than gradient descent for a class of problems, it is also less error tolerant as computation in an iteration is not based on the current position of x. Thus, unlike gradient descent, errors affecting the current position of x cannot get corrected in future iterations.

More details on conjugate gradient can be found in [2].

$$
\begin{aligned}
&r(0) = b - Ax \\
&p(0) = r(0) \\
&t = 0 \\
&\text{while } t < A.width \\
&\qquad \alpha(t) = \frac{r(t)^T r(t)}{p(t)^T Ap(t)} \\
&\qquad x(t+1) = x(t) + \alpha(t)p(t) \\
&\qquad r(t+1) = r(t) - \alpha(t)Ap(t) \\
&\qquad \beta(t) = \frac{r(t+1)^T r(t+1)}{r(t)^T r(t)} \\
&\qquad p(t+1) = r(t+1) + \beta(t)p(t) \\
&\qquad t = t + 1
\end{aligned}
$$

Figure 2.2: The conjugate gradient algorithm.

2.3 Design Requirements of a Solver Engine

Typical hardware accelerator designs focus on accelerating one operation. For example, in a work on accelerating dense matrix operations, Zhuo and Prasanna present four different designs for four different operations: dot product, matrix-matrix multiplication, matrix-vector multiplication, and LU decomposition [5]. Similarly, [6] contains a hardware design which only accelerates sparse matrix-vector multiplication and nothing else. (More detailed examinations of these works can be found in Section 7.2).

Our target algorithms feature a number of different operations, even within a single algorithm. Consider gradient descent-based algorithms. Such algorithms often deal with sparse matrices due to the nature of the constraints, especially for programs that are traditionally non-numeric (e.g., bipartite graph matching). As such, sparse matrix multiplications need to be supported in any accelerator architecture for such algorithms. However, such algorithms are also heavy in vector operations due to the sparsity of the matrices. So, the accelerator may need to support vector operations as well. In fact, vector operations take approximately 60.6% of the execution time that the sparse matrix-vector multiplication does for graph matching. Even if the matrix is dense, the vector operations may still comprise a significant fraction of the execution time depending on the shape of the matrix.

As another example, conjugate gradient iterations feature matrix-vector multiplication, vector operations, and dot products. If the matrix is sparse,

then the complexity of matrix-vector multiplication may be on par with the vector operations as the matrix may only contain $O(N)$ non-zero values. With a random sparse matrix of size 150x150 with sparsity .05, we observed that 27.6% of the iteration time is spent on vector operations, while 68.5% of the time is spent on the matrix-vector product. If the matrix is dense, on the other hand, the matrix-vector product will dominate since its computational complexity will be $O(N^2)$, while the vector operations will have an $O(N)$ overhead. In fact, with a dense 150x150 matrix, the matrix-vector product takes 93.5% of the execution time of the entire iteration.

In the case that A is not a symmetric positive definite matrix, $A^TAx = A^Tb$ needs to be solved. If A is an MxN matrix, computing A^TA will have a computational complexity of $O(MN^2)$. Alternately, we can perform two matrix-vector products per iteration (first calculating Ax, then multiplying the resulting vector by A^T). This results in a computational complexity of $O(MN)$ per iteration for N iterations. Despite the fact that the computational complexities are the same, if $M > N$, it is much better to perform the matrix-matrix multiplication since matrix-matrix multiplication is much more memory efficient than matrix-vector multiplication. If the matrix-matrix multiplication is performed, every iteration will be dominated by a matrix-vector multiplication involving A^TA, an NxN matrix. This means that each iteration will feature $O(N^2)$ complexity, and with N iterations the total complexity of the iterative portion of the computation will be $O(N^3)$. The ratio of work between the matrix-matrix multiplication and the iterative computation will be determined by the ratio of M to N. Therefore, if M is not substantially larger than N, the matrix-vector product will contribute a significant enough fraction of the execution time to warrant acceleration.

CHAPTER 3

A SOLVER ENGINE BASED SYSTEM ARCHITECTURE FOR ACCELERATING GRADIENT DESCENT AND CONJUGATE GRADIENT

In this chapter, we examine the hardware support for accelerating gradient descent and conjugate gradient. We first examine the overall system level layout, showing how the solver engine is integrated onto a general purpose CPU die. We also define the interface used to communicate with the solver engine. We then examine the layout of individual processing elements and describe how they are used to perform each of the solver engine's target operations.

3.1 System Layout

The solver engine is designed to be integrated onto a general purpose CPU die. Figure 3.1 shows the system level layout of a single core augmented with the solver engine. The memory interface of the core has been modified to capture commands which should be routed to the solver engine. We have two choices for connecting the solver engine to the on-chip interconnect. We can give the solver engine its own interconnect port (and thus it would be treated as an independent core from the point of view of the memory system), but this may result in increased interconnect complexity due to the addition of more nodes. Alternatively, the solver engine and the core could share one port. However, sharing the port means that either non-trivial arbitration between the core and the solver engine must be added or only one of the two may be active at a time.

For now, we assume the solver engine can be given its own unique connection to the chip interconnect. In addition, the solver engine needs access to the TLB via the addition of a new read port. Ideally we would store physical addresses to avoid the need for the TLB; however, large matrices or vectors may span multiple pages. Since we cannot guarantee that contiguous virtual

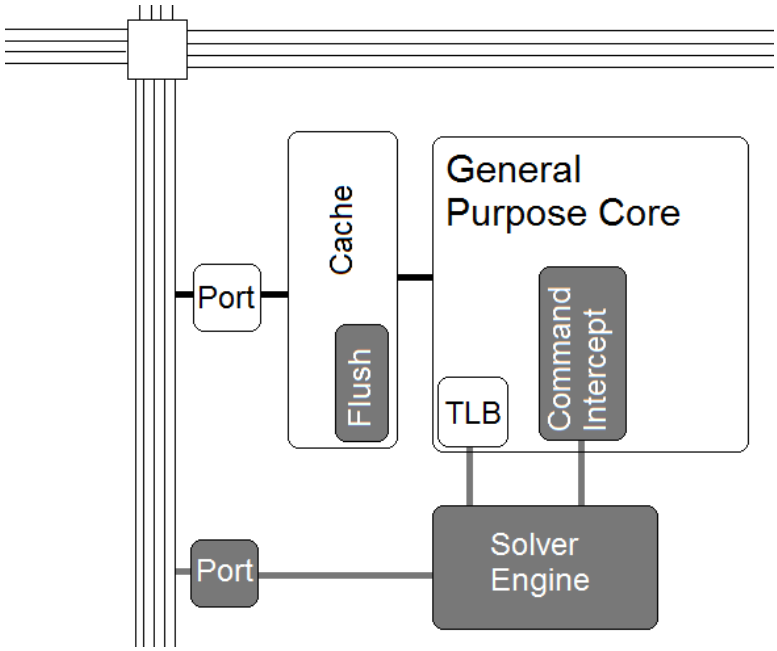

Figure 3.1: A diagram of a general purpose CPU augmented with the solver engine. New components are in gray.

pages are also contiguous in physical memory, we need to store virtual addresses and translate every access. In the event that the solver engine suffers a miss in the TLB, the core will need to process the miss just as if it suffered the miss itself. Ideally, the miss latency will be hidden by the solver engine prefetching far enough in advance, either naturally or by deliberately probing ahead to force page misses.

Because the solver engine will be interacting with a standard core which will have caches, we need to ensure coherence between the caches and the solver engine. We have two choices here: integrate the solver engine with the coherence protocol, or flush the caches manually to maintain coherence. The challenge of integrating the solver engine into the coherence protocol is that the solver engine does not cache data. As such, it needs to be able to read the data from the cache without being marked as a sharer of the data anywhere. Similarly when it writes data, the data needs to be invalidated everywhere, rather than the solver being marked as the owner of modified data. And since the solver engine cannot respond to coherence requests, we need to ensure that no CPU is attempting to touch the data during the operation of the solver engine.

Flushing the data is more straightforward, although we still need to ensure that no CPU accesses the operand data while the solver is executing. Obviously flushing the entire cache system is not ideal, since we will flush a substantial amount of data which is not related to the solver engine's operation. Instead, we would like to be able to flush only the data associated with the matrices and vectors actually being used. We thus assume a targeted invalidate of some kind will be available. Even if it requires some overhead we only need to flush once (right at the beginning of the algorithm after we have set up the matrices and vectors but before we start computation), so we assume the cost is negligible.

We should note that the fact that we are not taking advantage of the data being cached on chip does not negatively affect the performance of the solver engine. The solver engine is designed assuming it will be memory bandwidth limited. As such, even if higher effective bandwidth were available (due to the data being cached), we could not take advantage of it. This does not pose a problem as reasonable problem sizes will not fit within the cache and will thus be bandwidth-bound anyway.

There is one final potential pitfall due to cache coherence due to false sharing. If a matrix or vector starts or ends mid cache-line, there may be unrelated data on the remainder of the cache line. If this is the case, we cannot assure proper coherence. Standard cores may need to access and/or modify that data, violating our otherwise enforceable condition that no CPU access the operand data when a computation is in progress. Furthermore, when the solver engine is writing data back to memory, we want to avoid the overhead of needing to read-modify-write as this both increases memory bandwidth and requires additional hardware. As such, we assume that all matrices and vectors start at cache-line boundaries and are padded to cache-line length. Given the large size of the matrices and vectors we will be operating on, this overhead is negligible.

3.2 Engine Interface

The solver engine presents a memory-mapped IO interface. A range of addresses correspond to a set of registers inside the solver engine, while one address is used to send commands. When a command is sent via the com-

mand address, the solver engine begins immediately computing the command, blocking all subsequent IO until it has finished the computation.

The solver engine contains four sets of registers, for scalar values, vectors, dense matrices, and sparse matrices. (We could also have one set of registers and flag what kind of register it is, but different types of registers need different amounts of storage.) Scalar registers hold a single, double precision, floating point number. Vector registers hold a virtual pointer into memory pointing to the start of the vector as well as the length of the vector. Dense matrix registers similarly contain a pointer to memory as well as the width and height of the matrix. Sparse matrices 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.

The solver engine is capable of performing 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 in matrix-vector multiplication. Each instruction consists of the 2-bit op-code, the source and destination registers, and a few additional bits of information. The instruction has 6 4-bit fields for the registers, one each for α, β, A, B, C, and D. (Note that not all operands are used in every operation, in which case the field should contain 0.) Whether A, B, C, and D represent vectors or matrices is inferred from the instruction, while an additional bit is used to flag if A is sparse.

There are two additional bits for each source register (A, B, and D) to indicate whether or not the operand should be clipped. "00" indicates no clipping. "01" indicates that all values in the operand less than 0 should be treated as 0, while "10" does the same thing for values greater than 0. "11" is invalid. The reason the solver engine supports these operations, as well as the logic required, is described in Section 3.3.6. Finally, we need two bits, one each to control the $\pm$ terms in the operations.

3.3 Solver Engine Architecture

The backbone of the solver engine is made up of an array of processing elements (PE). The general layout can be seen in Figure 3.2. Each processing element consists of a pipelined floating point multiplier, adder, and a small

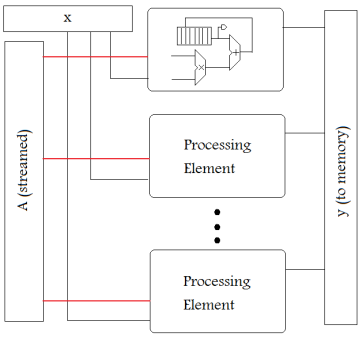

Figure 3.2: The general layout of the base architecture of the solver engine.

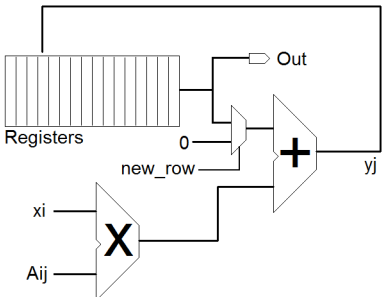

Figure 3.3: The general layout of a processing element. Wires are labeled as for matrix-vector multiplication.

amount of local storage to accumulate values in. The layout of the basic PE can be see in Figure 3.3. Defining an operation consists of supplying the inputs to the multiplier of the PE, controlling which value in the local storage the multiplication will be summed with, and controlling when the local memory will be output. The following parameters define the size and characteristics of the solver engine. The number of processing elements is p, La refers to the latency of the adder, Si is the number of output values which can be calculated at a time (and so each PE has $\frac{Si}{p}$ registers), and Sj is the maximum block width in sparse matrix vector multiply.

3.3.1 Vector Operations

The easiest operations to support are vector operations, i.e., operations of the form $c = \alpha b \pm \beta d$ where α and β are scalars and b, c, and d are vectors. In this case, an Si sized block of b is streamed into the engine with one value

assigned to each PE. The second input of the PEs is α which is broadcast to all PEs. The output of the multiplier is summed with 0 and the output of the adder is stored into local storage. Once Si values of αb have started being calculated, the corresponding values of d are streamed in while β is broadcast to the second PE input. The adder in the PE adds the newly calculated βd with the previously stored αb.

Computation for consecutive blocks of Si output values can be overlapped with no latency since αb is summed with 0 rather than the contents of local storage. As such, at that point we can output the values in local storage corresponding to the previous Si calculated values.

3.3.2 Sparse Matrix-Vector Multiplication

The most important operation the solver engine handles is of the form $c = Ab \pm \beta d$ where A is a sparse matrix. The βd portion of the calculation is handled similarly to the first half of a vector operation where Si values of d are streamed in, multiplied by β, and stored. The solver engine then continues the computation by multiplying one row of blocks in A by b to calculate Si output values.

To perform the actual matrix-vector computation, the solver loads the next Sj values of b starting at the next block's starting column. The solver then begins streaming in value, column, row tuples. A total of p values are read in at a time, and one is passed out to each processor. 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 into the PE. At the output of the multiplier, the row index is used to read the appropriate value from the PE's local storage to accumulate with. It is then used at the output of the adder to write back into the local store. 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. (This chunk can actually be preloaded while the previous block is calculating, so there is no delay between blocks.) In order to handle padded zeros, we flag zero outputs of the multiplier, but go ahead and read the value out of the local storage indicated by the row index. We propagate the flag through the adder so that when the (now non-zero) value reaches the output of the adder, we can

disable writing that value to the local storage. Because each PE can only read and write its local storage, all values associated with a row of A should be sent to the same PE.

In between rows of A, we need to ensure that all $\frac{Si}{p}$ values stored in each PE are output. Just like with vector operations, we can output those values while calculating βd. In the event that we only want the solver engine to calculate Ab without adding a vector, we still need to emulate the addition to ensure the values get flushed out of the PEs and the local store in each PE gets initialized to zero. This is done by detecting that $\beta = 0$ and ensuring that both inputs to the PE are 0 during this time (instead of retrieving d from memory). This overhead is small when compared to the actual matrix-vector multiplication ($\frac{Si}{p}$ cycles) so there is no incentive to add complexity to avoid it.

With a large number of processing elements all needing to read the buffered portion of b, it may become too complex. If this is the case, the b buffer can be replicated so that only a subset of processors read from one buffer. Since the buffers are being preloaded during the previous block's computation, replicating the data will not affect the critical path and so will not present a substantial burden.

3.3.3 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, we know that every PE will access every value of b in order. As such we can simply broadcast the relevant value of b to each PE. We also no longer have the row indices to index into the PE's local storage. However, again because of the regular nature of the dense matrix, we can simply generate the row indices deterministically.

One catch with dense matrix-vector multiplication is that we actually want to read the data in column-major format. This requires that the matrix either be stored in column-major format or that we transpose the matrix on the fly. We examine the cost of dynamically transposing A in Section 3.3.6.

3.3.4 Dense Matrix-Matrix Multiplication

Dense matrix-matrix multiplication utilizes a blocked rank-one update algorithm to perform $C = AB$. Calculation is done on one block of the matrix C at a time. Each block is of height i and width j. Since the block must fit into the available Si words of storage, $i * j = Si$. In our solver engine we use $i = p$ and $j = \frac{Si}{p}$. We could use $i > p$, but this requires more complex control and more buffering of values. Thus in our design, each PE is responsible for a j sized row of C.

To calculate an ixj block in C (call the current block pC), we need i rows from A (call it pA) and j columns from B (call it pB). pC is initialized to all 0. Then we calculate the outer product of the kth column of pA and the kth row of pB, which 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 panel is complete and can be output while work begins on the next panel.

In order to actually perform this computation, the solver engine reads in a column of i values from pA and buffers them. (Since in the solver engine $i = p$, we only need two registers per PE: one for the current column and one to preload the next column.) The appropriate row from pB is streamed in one value at a time and broadcast to each PE. Once we have read the entire row of j values from pB, we move to the next column of pA and the next row of pB.

To handle the transition between blocks of C, we rely on the fact that for the first outer product for a panel it is being summed with 0. Thus, at the point we would normally read from local storage in the PE to send to the adder, we can instead send that value from the previous panel out of the PE. This way there is no latency between blocks. Again, like with dense matrix-vector multiplication, A needs to be in column major format. B, meanwhile, needs to be in row-major format. We will examine the implications of this in Section 3.3.6.

3.3.5 Dot Product

Supporting dot products requires the most additional hardware since, unlike the other operations, we cannot assign each PE an independent set of work. Just as with vector operations, we pass out p values of the first vector per

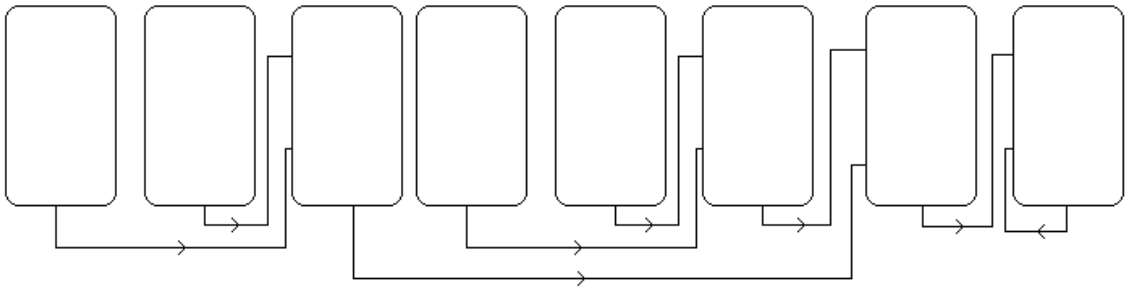

Figure 3.4: The flow of values in the adder tree. The output of the final PE loops back into itself to finish the summation.

cycle, one to each processor. However unlike vector operations, rather than broadcast a single value to all processing elements for the second operand, we also pass out the corresponding p values of the second vector instead. We continuously stream in these values until both vectors have been read in. Meanwhile, the output of the adder in each PE is directly routed back into its input rather than written into local storage.

The first stage of the dot product finishes when 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. We perform this summation by overlaying an adder tree on top of the processing elements. In other words, the output of each adder is routed to one of the inputs of the adder above it in the adder tree. Now, with p PEs, we can only create an adder tree with $p-1$ PEs. The output of the top of the adder tree is routed to the multiplier-side input of the one remaining PE's adder. This final PE's output remains looped back to itself just as in the first stage. As computation continues, the final PE will eventually sum all the remaining values until La values remain, all in the final PE's pipeline. An illustration of this adder tree can be seen in Figure 3.4.

At this point, we transition to the third stage of the computation. A special purpose control unit controls the adder writing to the PE's local storage, delaying until the next valid value is at the output of the adder, then reading it back out to sum with the most recent output. An illustration of this process can be seen in Figure 3.5. Because the control unit need only be present in one PE, and because the adder latency La will be very small, this final control unit need not be very complex.

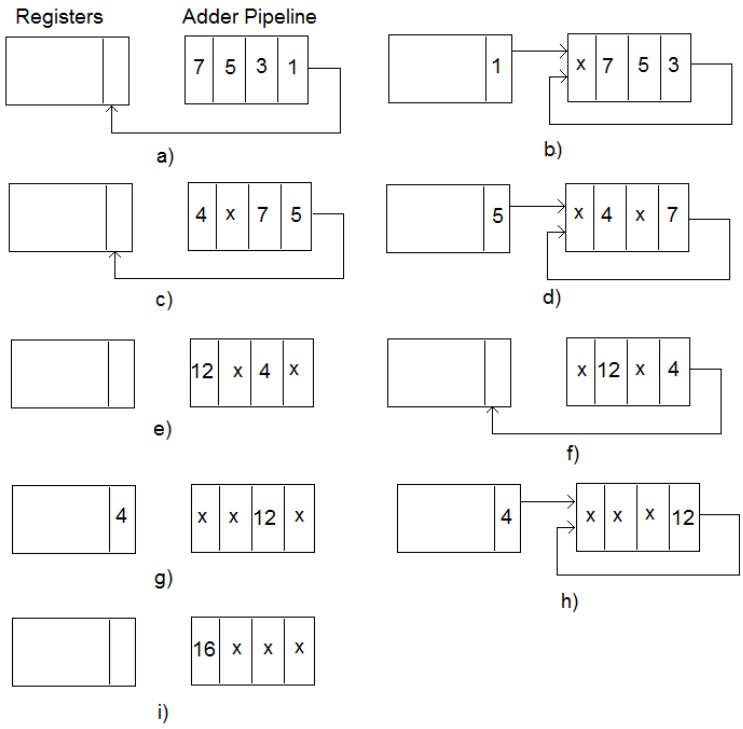

Figure 3.5: A view of the adder pipeline and registers of the final PE in the dot product tree assuming an adder latency of 4 cycles.

While the dot product process may seem rather logically complex, it does not require substantial additional hardware and still performs very efficiently. The adder tree can be implemented by the addition of a single multiplexer per PE. As for the performance, the first stage will require $\frac{N}{p}$ cycles, where N is the length of each vector. The second stage will require La cycles per level of the adder tree, which will be $log_2 p$ levels tall, for $Lalog_2 p$ total cycles. In the final stage, we halve the number of remaining values every La cycles and start with La cycles, so it will take $La(1 + log_2 La)$ cycles. The first stage will clearly dominate for large N, as it would for any hardware design to calculate dot products.

3.3.6 Additional Support

As noted in the discussion of dense matrix-vector and matrix-matrix multiplication, we either need to store some matrices in column major format or we need to transpose row-major matrices on the fly. Assuming a read from

memory returns a line of W words, we would need a total of $2 * W * Si$ registers. The solver engine would perform Si reads to fill half the registers, and then read the values out in column major format. In order to hide the latency of doing this buffering, we would also need to be loading the next Si lines at the same time. Assuming 64-byte lines and $Si = 256$, we would need 32 kB of storage total to perform this transposition.

This does cost a fairly substantial amount of area, more than the rest of the solver engine with 32 PEs; however, because it is low utilization, it does not cost a substantial amount of power relative to the PEs themselves. As such we would consider it an acceptable cost in a general purpose accelerator. However given our target applications, there is not sufficient justification for its use. In practice, only conjugate gradient uses dense matrices. Conjugate gradient calculates $A^T A$, so it requires A in row-major format and A^T in column-major format. Since these two are equivalent, we only need one copy of A. The further matrix-vector multiplications are with a symmetric matrix which obviously would never require transposition.

Gradient descent often needs to calculate $[Ax - b]_+$, where $[.]_+$ means $max(., 0)$. (This is used to enforce one-sided constraints, where we only want the penalty to apply if $Ax - b > 0$.) This can be done with a single 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. This can be trivially modified to also support the opposite operation, $min(., 0)$.

3.3.7 Complexity of Supporting Multiple Operations

While supporting multiple operations will undoubtedly provide positive performance benefits, they must be weighed against the additional complexity of supporting these operations.

Table 3.1 summarizes the components of the architecture and which operations utilize them. (Buffers are referred to with the labels in Figure 3.2.) Clearly, most operations need most of the components to be present. More importantly, except for the multiplexers to overlay the adder tree on the PEs, there are no major components which are only necessary due to the support of multiple operations. A is always necessary so that values can be

Table 3.1: The operations of the system and which components they utilize

Unit	Used By
A	All
x	Dense MM, SpMV, Dense MV
y	All
Floating Point Multiplier	All
Floating Point Adder	All
PE Registers	All but Dot Product
Adder Tree	Dot Product

preloaded to be passed out to the PEs simultaneously. The x buffer is always necessary in matrix operations to store the current row of the matrix B (for matrix-matrix multiplication) or the current segment of x (for matrix-vector multiplication). The y buffer is always necessary to store the output while it is written off chip so that the PEs can continue processing. The adder tree will contribute some additional logic in the amount of one additional multiplexer per PE. There will also be some additional complexity in the control logic since the control unit must support generating control signals for multiple operations; however, the vast majority of the area of the accelerator will be taken up by the floating point units and buffers. Thus any additional control complexity should be minor relative to the overall area of the accelerator.

CHAPTER 4

SOFTWARE SUPPORT FOR THE SOLVER ENGINE

In this chapter, we examine the software support necessary for efficient use of the solver engine described in Chapter 3. First, we define a new sparse matrix storage format, the Row Blocked Coordinate List (RBCOO). Sparse matrix-vector multiplication requires that the x vector be buffered on chip for reasonable performance, and a blocked format ensures that only a segment of x will be accessed at any given time. This allows us to buffer only the active segments of x, as otherwise we would be limited to problem sizes where x was small enough to fit on the chip in its entirety. Blocked formats also allow for a lower memory footprint by storing lower precision relative indices for each value, and only storing full precision absolute indices for each block [7]. Next, we describe scheduling values within the RBCOO format, which consists of reordering non-zero values and adding padding zeros such that values are presented to the correct processing element in an order which does not cause a data hazard due to the adder pipeline. We also describe two further optimizations which can be used to improve the performance of the RBCOO format on the solver engine.

4.1 Row Blocked Coordinate List Format

Compressed sparse row (CSR) or compressed sparse column (CSC) are fairly common sparse matrix formats. However they can be somewhat complex to execute in parallel when streaming in the values as only one row (for CSR) or column (CSC) can be scheduled at a time (without complex logic). This is particularly a problem as the sparse matrices seen in gradient descent often have very few values in one dimension. For example, bipartite graph matching (which will be described in detail in Chapter 5) features n values in one dimension and 2 values in the other for problem size n. This will result

in extremely poor behavior as a number of zeros will need to be used to pad the computation, drastically limiting the effective speedup.

Instead, we store each subblock as a coordinate list (COO). We can then arbitrarily reorder the values within the block. Given p processing elements, every p contiguous values represent one cycle of values, each being assigned to a different processing element. The values are ordered so that they will be statically scheduled in such a way as to prevent data hazards and so that there is no need for global storage that all processing elements need to be able to write to.

The RBCOO matrix format itself consists of six arrays of data. The first array, *val*, is an array which stores all the non-zero values in the matrix (as well as any zeros added for padding purposes). The row and column indices of each value relative to the upper left corner of the block are stored in $relColIdx$ and $relRowIdx$. Because these indices are relative and we know the maximum block size, we do not need full 32-bit integers. We actually only need 8-bit integers if blocks are limited to 256x256.

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 $blockPtr$ array, indicating that that block is the start of a new row of blocks. An example matrix is shown in Figure 4.1, along with the corresponding CSR matrix.

4.2 Scheduling

In its unscheduled and unpadded format, RBCOO will feature a smaller memory footprint than the corresponding CSR matrix due to the use of relative indices. We could save even more memory by storing the blocks in CSR format while still using relative indices. However, the choice of using a coordinate list was made because the additional memory saved is not substantial enough to make up for the fact that we can no longer arbitrarily reorder values within the block. We rely on this ability to reorder values (and to pad with zeros) to efficiently utilize all processing elements.

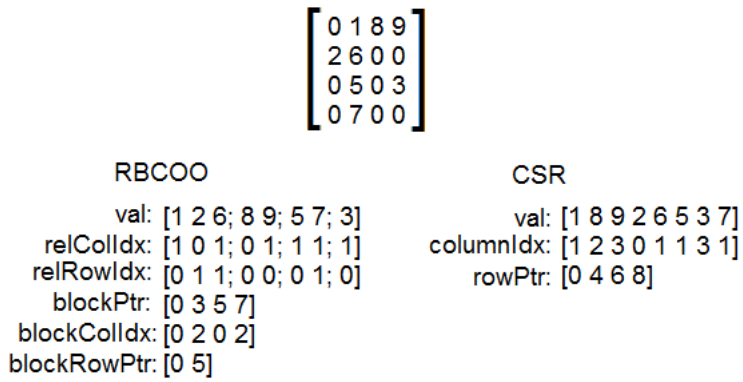

Figure 4.1: An example matrix encoded as CSR and as RBCOO with a 2x2 block size. Semicolons indicate boundaries between blocks for clarity. The RBCOO matrix has not been padded with zeros or reordered for scheduling on the hardware.

The computation will progress through an entire row of blocks before progressing to the next row, meaning that only a portion of the output vector is active at any given time. Each processing element will be in charge of a subset of the active block of the output vector. Thus values for a given row of the current block need to always be handled by the same processing element. (By doing this we avoid a globally written structure or any need for inter-processing element communication.) Because the floating point units will be pipelined, there is a potential data hazard. If a new value from a row of A is used before a previous value from the same row has exited the adder pipeline, a data hazard will occur because both will try to accumulate with the same register in local storage. Thus the scheduling algorithm also needs to ensure that at least La cycles will pass between references to the same row.

We currently use a greedy scheduling algorithm outlined in Figure 4.2. Each "cycle" of scheduling consists of choosing p values, one for each processing element. For each PE we examine the rows assigned to that PE and choose the row with the largest number of values remaining that has not been scheduled in the previous La cycles. If no row can be chosen (because this processing element 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 too recently scheduled), the algorithm inserts a padded zero. As we will see in Section 6.4, the overhead of the extra padded zeros is small given

```
for each sub-block:
  while unscheduled non-zero values remain in current sub-block:
    for each processing element PE:
      R = all rows assigned to PE not scheduled in previous La cycles
      if R empty:
        schedule padded 0
      else:
        choose row in R with maximum remaining non-zero values
        schedule next value of chosen row
```

Figure 4.2: Pseudocode for scheduling the RBCOO matrix to run on the solver engine. Scheduling consists of choosing the order of values and padding with zeros when necessary. La refers to the latency of the adder.

large enough, well behaved matrices.

In addition to the overhead added by padding zeros, we need to worry about the computational complexity of the scheduling. Typically with sparse matrices, the matrix is created using a format which allows for flexible addition of values such as a list-of-lists matrix (which comprises a linked list of rows, each of which is a linked list of value, column pairs). The matrix is then converted into a format which is more suitable to computation such as CSR, or in our case RBCOO. The scheduling algorithm is applied during this conversion when copying a list-of-lists matrix to an RBCOO matrix. Converting from list-of-lists to CSR is $O(nnz)$ where nnz is the number of nonzero values.

Assuming the matrix is well-formed and contains roughly the same number of non-zeros per block and the same number of non-zeros per row per block, each block will consist of $\frac{nnz}{numBlocks}$ non-zero values. Each scheduling pass will schedule p values and requires examining all S rows in the block. Thus the complexity of scheduling each block will be roughly $O(\frac{nnz}{numBlocks*p} * S)$, or $O(nnz * \frac{S}{p}) = O(nnz)$ for the entire scheduling process. While the constant factor is fairly substantial, for large enough problem sizes the conversion overhead is acceptably small relative to the actual computation.

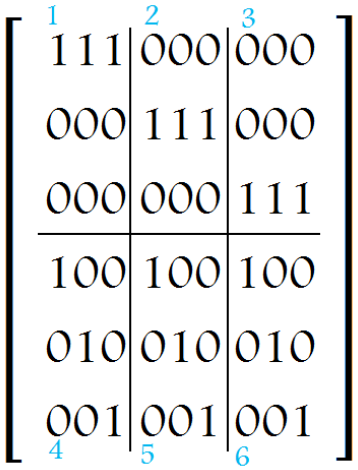

Figure 4.3: The constraints for bipartite graph matching with 3 vertices in each set. Blocks 1, 2, and 3 can use only a single PE at a time.

4.3 Column Shuffling

As noted above, scheduling does well if the matrices are well behaved, i.e., if in each block, each processor has roughly the same number of non-zeros. Unfortunately, for some applications, the constraint matrix is not well behaved due to the patterns the constraints form in the matrix. Figure 4.3 shows the constraint matrix for graph matching on a fully connected 3,3 graph. It is divided into blocks assuming a block size of 3 and 3 processing elements; however, the same phenomenon occurs with larger block sizes and input sets.

Blocks 4, 5, and 6 all would schedule well. Each processing element handles its one nonzero value in the block in the first cycle and the computation would proceed to the next block. However blocks 1, 2, and 3 all contain non-zeros only for one processing element. This means that the computation for each block would require $3 * La$ cycles, requiring the other processing elements to execute on padded zeros.

This results in unacceptable overheads when converting into RBCOO format. However, we can counter this by employing column shuffling. When the matrix is being built, we shuffle the column indices as $col_{new} = q * col_{old} \% colWidth + 1$. Figure 4.4 shows the same matrix with shuffled columns. As can be seen, the matrix will schedule much better than if it were unshuffled. The only additional thing that must be done is that b must also have its values shuffled in the same manner. This shuffling requires $O(nnz)$ com-

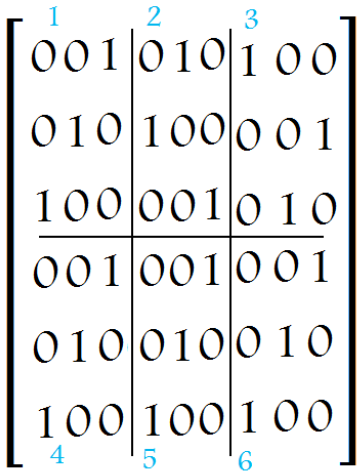

Figure 4.4: The constraints for bipartite graph matching with 3 vertices when columns have been shuffled to achieve better scheduling with $q = 11$. Each block can use 3 PEs at a time.

plexity since it needs to be applied once to each value in A, so it does not increase the computational complexity of building the list-of-lists matrix.

4.4 Row Indices

Previously, we described how the row indices are relative to the start of the block, and thus we can save memory by only utilizing enough bits to span the maximum block height of 256 rows. One weakness of this approach is that if we want to increase the number of PEs in the solver engine but keep the block height fixed, each PE will be assigned fewer rows. This not only negatively impacts the scheduling flexibility (resulting in more padded zeros), but sets a hard limit on the maximum number of PEs which can be utilized.

To mitigate this problem, we can take advantage of the fact that row i is statically assigned to PE $i\%p$. This means that we need not store the bits that determine which PE a value is being assigned to. For example, given 8 bit row indices and 32 PEs, the low 5 bits of the row index will be identical for all rows assigned to the same PE. We can instead store the 6-13th bits of the row position, allowing us to utilize a maximum block height of $2^{13} = 8192$ instead of $2^8 = 256$. Essentially, instead of the entire solver engine being able to address 256 rows at a time, each PE can address 256 rows at a time even

if we scale the number of PEs. In practice, using this technique may result in utilizing too much storage if the maximum possible block size is chosen. However we can simply choose not to shift the row indices up as high as possible, instead only shifting them up until we get the desired maximum block height.

CHAPTER 5

METHODOLOGY

In this section, we first derive the design characteristics of the solver engine such as the number of processing elements and frequency. We then describe our simulation methodology and infrastructure used to generate our results. We also examine each of the example benchmarks and explain how they are formulated.

5.1 Synthesis Results

To get baseline power and area estimates, we synthesized an unpipelined 64 bit floating point adder and multiplier in 45 nm using the Nangate 45 nm Open Cell Library using an extremely small clock period to determine the maximum frequency. The multiplier was capable of reaching a critical path length of 15.67 ns, so 16 ns was chosen as the desired clock period. We resynthesized with a 16 ns clock period and the resulting statistics are summarized in Tables 5.1 and 5.2. (The synthesis results have 3 64-bit registers: two input registers for the two operands and one output register for the output of the unit.) We also derived statistics for a single 64-bit register.

We use these values to derive area and power characteristics for a design point with a given number of processing elements p, frequency f (which in turn determines the adder latency L_a), and the total number of rows the solver engine can be working on at a time, S_i. We derive the area and power estimates from three primary sources, the processing elements themselves (consisting of p adders, p multipliers, and S_i total registers of storage), the buffers for x during sparse matrix-vector multiplication (consisting of S_j registers), and the output buffers which hold the result of the computation for a given set of rows before they are written back to memory (consisting of S_i registers). The resulting area and power numbers do neglect the control

Table 5.1: The power characterization of a 64 bit floating point adder and multiplier at the 45 nm technology node

Unit	Dynamic Power (μW)	Leakage Power (μW)
Multiplier	822	213
Adder	152.93	37.8

Table 5.2: The area characterization of a 64 bit floating point adder and multiplier at the 45 nm technology node

Unit	Combinational Area (μm^2)	Register Area (μm^2)
Multiplier	25033	1253
Adder	4906	1223

unit, TLB port, chip interconnect port, and routing; however, the components included will account for the majority of the area and power of the design.

5.2 Size of Engine

We have three variables we can set in our design: the number of processing elements p, the frequency f (which implicitly determines the adder latency L_a), and the number of rows per block in our matrix format, S_i. (Refer to Section 4.4 for an explanation of why we can vary S_i.) There are two obvious restrictions on the overall design. The first is that it should not consume too much power and the second is that it should not take up too much area. However memory bandwidth can end up being an even more stringent restriction. Matrix vector multiplication and vector operations are memory bound operations, requiring at least one value per processing element per cycle.

Given that each PE will need to consume one double precision value per cycle, we can estimate the necessary memory bandwidth as $8 * p * f = 8 * p * f_0 * L_a$, where f_0 is the frequency of the unpipelined floating point units. We assume the amount of memory bandwidth available to be 32 GB/s, which is in line with current top-of-the-line desktop processors. Given that $f_0 = 62.5$

Table 5.3: The area of various design points of the solver engine in mm^2

$S_i \backslash p$	64	32	16	8
256	1.69	1.12	0.83	0.68
512	1.91	1.33	1.04	0.9
1024	2.34	1.76	1.47	1.33
2048	3.2	2.63	2.34	2.19

Table 5.4: The power consumption of various design points of the solver engine in W

$S_i \backslash p$	64	32	16	8
256	0.1	0.12	0.16	0.24
512	0.1	0.12	0.16	0.25
1024	0.1	0.13	0.17	0.27
2048	0.11	0.14	0.19	0.31

MHz from Section 5.1, we derive the restriction $8 * 62.5$ MHz $*p * L_a < 32$ GB/s, or $p * L_a < 64$. Also, given that we have 8 bits per PE to address rows, we can restrict $S_i < 256 * p$.

We now have a tradeoff between many low latency processing elements and fewer higher latency processing elements. Increasing the number of processing elements and decreasing frequency increases area and decreases energy consumption for the same memory bandwidth (disregarding padded zeros due to scheduling). It also allows for a larger S_i since each PE can address up to 256 rows. This helps scheduling efficiency at the cost of area. However, it turns out that having fewer higher latency processing elements itself also improves scheduling efficiency even when maintaining the same number of rows per PE because there are fewer PEs to pad zeros for when only a few PEs have non-zero values remaining. Tables 5.3, 5.4, and 5.5 summarize the area, power, and scheduling overhead of the design space. (The scheduling overhead was calculated for a random 2048x2048 matrix with sparsity .052.) We also show the performance of the various design points (solving the randomly generated matrix with conjugate gradient) in Table 5.6. The performance results show interesting behavior as diagonals of equal performance appear, representing a range of area, energy points for the same performance.

Table 5.5: The percentage overhead of padded zeros due to scheduling for various design points

$S_i \backslash p$	64	32	16	8
256	27.49	18.92	12.67	7.1
512	19.97	14.02	8.68	5.37
1024	16.53	9.39	5.94	3.91
2048	11.19	6.67	4.15	2.23

Table 5.6: The execution time (in μs) of one iteration of conjugate gradient on a 2048x2048 sparse matrix with sparsity 0.52 for varying design points of the solver engine

$S_i \backslash p$	64	32	16	8
256	106.27	94.12	87.76	84
512	96.32	87.67	83.14	80.61
1024	89.22	83.88	80.43	77.94
2048	84.26	79.98	77.78	76.49

From this data, we can derive the optimal design point for our system. For our experiments, we choose the design point with the lowest area * power * performance product, which happens to be $p = 16$, $f = 250$ MHz, $L_a = 4$ cycles, and $S_i = 256$. The design features an area of .83 mm^2 and a power consumption of 0.16 W, well below a general purpose processor. For real designs, decisions can be made depending on energy/power, area, performance, or a combination of the three, but these exact decisions are beyond the scope of this work. A summary of the characteristics of the solver engine can be found in Table 5.7.

5.3 Simulation Infrastructure and Methodology

In order to perform experiments, we need to measure the performance of both the standard core as well as the solver engine itself. In order to measure the performance of the general purpose core, we ran our applications on M5 in system emulation mode. We augmented M5 to perform a purely functional simulation of the solver engine and enabled utilizing the solver engine by

Table 5.7: The characteristics of the solver engine

Frequency	250 MHz
Processing Elements	16
Adder/Multiplier Latency	4
S_i	256
S_j	256
Memory Bandwidth	32 GB/s

Table 5.8: The characteristics of the general purpose core

Frequency	3.33 GHz
Issue/Execution Width	4
L1 Cache (I/D)	32 kB/64 kB
L2 Cache	2 MB

adding in syscalls which performed the requested operations instantaneously. Just as the solver engine avoids caches, the functional solver engine simulator retrieves data directly from memory. We further augmented M5 to allow the core to force the write back and invalidation of data so that data being read by the solver engine is coherent, just as a real system utilizing the solver engine would need to. The characteristics of the general purpose out of order core simulated are in Table 5.8.

Because we implemented only a functional simulator of the solver engine itself, we can only measure the performance of the core using M5. In order to measure the performance of the solver engine itself, we wrote analysis routines with the same interface as the corresponding algorithms, gradient descent or conjugate gradient. When called by the benchmarks themselves, the analysis routines convert any sparse matrices into RBCOO format to measure the overhead of the padded zeros, then measure the number of cycles the solver engine is used per call and per iteration. These measurements ignore memory latency as the solver engine has been designed such that it overlaps computation with memory access. (There will be an initial delay when a computation begins, but the total computation time is much greater than the initial delay.) The analysis tool *does* take memory bandwidth into account as it is the limiting factor on any operation but matrix multiplication.

A summary of the characteristics of the solver engine is in Table 5.7.

In order to examine the robustness of our applications, we create an error model which can be applied by both the functional units in M5's CPU models and by the functional solver engine simulator. The time between errors is determined by an exponential distribution. If an error occurs on a particular computation, mantissa bits are flipped at random. Each mantissa bit flips with an independent probability which is chosen such that the expected number of bits flipped per error is approximately 1. (Specifically, each mantissa bit in a double precision number flips with probability .02, and each bit in a single precision number flips with probability .04.) The error model does not guarantee that at least one bit flips, so it is possible that even though an error "occurs," no bits are actually flipped. This happens with probability $.98^{52} = .35$. All error rates or number of operations per error seen from here on are adjusted to account for this.

5.4 Benchmarks

5.4.1 Bipartite Graph Matching

Bipartite graph matching involves finding the set of weighted edges in a bipartite graph such that every vertex is adjacent to at most one edge in the set and the total weight of the set is maximized. We use a complete bipartite graph with equal nodes in each half. The Hungarian algorithm is an $O(V^2E)$ algorithm and is implemented by the OpenCV calcEMD2() call which is utilized as the baseline non-robust algorithm [8].

Gradient descent is transformed into a constrained optimization problem by treating the search variable X as a VxV matrix such that x_{ij} is 1 if the edge between the ith node in one vertex set and the jth node in the other set is in the maximum matching. Otherwise X_{ij} is 0. Thus we want to minimize $-c_{ij} * x_{ij}$, where c_{ij} is the weight of the corresponding edge. We also need to apply the constraints so that each vertex is adjacent to at most one edge. This is done by requiring that every row in X contain at most one 1 and every column contain at most one 1.

In order to further transform the problem into an unconstrained optimization problem, we must assemble our constraint matrix A and vector b. We

x01	x02	x03	x10	x12	x13	x20	x21	x23	x30	x31	x32
1	0	0	-1	-1	-1	0	1	0	0	1	0
0	1	0	0	1	0	-1	-1	-1	0	0	1

Figure 5.1: The constraints for maxflow with 4 nodes. The label x23 indicates the column is associated with the edge from node 2 to node 3. Each row is enforcing the constraint that the total flow entering a node equals the total flow exiting a node. Nodes 0 and 3 are the source and sink respectively, so there are no constraints on the total flow entering and leaving the nodes.

need two sets of constraints. First we must constrain $x_{ij} >= 0$. Second, we must constrain $\sum_{i=0}^{V} x_{ij} <= 1 \forall j$ and $\sum_{j=0}^{V} x_{ij} \forall i$. This results in a sparse matrix with $O(V^2)$ nonzero entries. An example for $V = 3$ can be seen in Figure 4.3. The variable X which we are optimizing for also contains V^2 entries, and so the complexity for a single iteration of gradient descent on this problem is $O(V^2)$.

5.4.2 Maxflow

Given a weighted directional graph, the maximum flow problem attempts to assign flows to each edge such that the total flow is maximized. The flow in any given edge may not exceed the weight of the edge, and except for two specially designated nodes (the source and the sink), the total flow entering a node must be equal to the total flow exiting a node. The source allows an infinite amount of flow to exit and the sink allows an infinite amount of flow to enter.

In order to solve maxflow with gradient descent, we want to minimize $-\sum_{i,j} x_{ij}$, where x_{ij} is the flow through the edge from vertex i to vertex j. We subject X to three constraints, $x_{ij} \geq 0 \forall i, j$, $x_{ij} \leq c_{ij} \forall i, j$ (where c_{ij} is the maximum capacity of the edge), and $\sum_i x_{ij} = \sum_i x_{ji}$ for all nodes j except the source and sink. An example constraints matrix for 4 vertices can be seen in Figure 5.1. (The uniform constraints are not shown since they are not implemented as a matrix.) The first two sets of constraints require $O(E)$ nonzero values total while the last set of constraints requires $O(V)$

constraints with $O(V)$ non-zero elements each. Since $E = O(V^2)$ for dense graphs, the total complexity of a single iteration of gradient descent will be $O(V^2)$.

We use the Edmonds-Karp algorithm as the baseline non-robust algorithm. Edmonds-Karp has an $O(VE^2)$ complexity.

5.4.3 Least Squares

The least squares problem is a common optimization problem where, given an overdetermined set of equations on a set of variables x, we wish to find the value of x which minimizes $||Ax - b||_2$. Each row of A corresponds to an equation, each column corresponds to a variable in x, and we expect substantially more equations than variables. In our experiments, A and b are randomly generated. We utilize Cholesky decomposition in our baseline non-robust version of least squares by solving $A^TAx = A^Tb$. We then use conjugate gradient to solve the same problem and compare the residuals of both approaches. Overall, both algorithms spend most of their time on computing A^TA, since given MxN matrices A^TA will be $O(N^2M)$. A^TA itself will be an NxN matrix, requiring an $O(N^2)$ matrix-vector multiplication each iteration for N iterations. Since $M > N$ in typical least squares problems, the matrix multiplication dominates. However if M and N scale at the same rate, increasing problem size will keep the ratio of work between matrix-matrix and matrix-vector operations the same.

5.4.4 System of Sparse Equations

Numerically solving sets of partial differential equations often results in needing to solve $Ax = b$ where A is symmetric, positive definite, and sparse simply due to the nature of the problem being solved. To simulate this, we randomly generate a symmetric positive definite sparse matrix A with sparsity between .05 and .06 and solve it. Unlike with least squares, there is no need to compute A^TA, so most computation is spent on the matrix-vector operation in each iteration of CG. Also, unlike least squares, we have found that Cholesky decomposition performs poorly due to fill-in caused by the factorization. Thus we consider the baseline to be CG itself. So while we gain no robustness

due to algorithmic transformation, we still show the energy and performance benefits of the solver engine in sparse matrix dependent algorithms.

CHAPTER 6

RESULTS

6.1 Performance

In this chapter, we evaluate the performance and energy consumption of the iterative algorithms when executed on a general purpose CPU (SW), when executed on a general purpose CPU with support for the operation which comprises the majority of the execution time (DOM), and when all supportable operations are executed on the solver engine itself (ALL). Figure 6.1 shows the execution time of a single iteration of gradient descent solving graph matching, normalized to the execution time of a single iteration with no acceleration. We see that accelerating only the matrix-vector multiplication does provide substantial speedup, but there is clearly room for improvement by accelerating all operations.

Figure 6.2 shows performance results for gradient descent solving maxflow, again normalized to a single iteration with no acceleration. Again, as expected, accelerating only sparse matrix-vector multiplication does provide the most speedup, decreasing execution time by about a third. However, accelerating vector operations decreases execution time by another third.

The performance results for using conjugate gradient to solve systems of sparse equations is in Figure 6.3. Here, accelerating sparse matrix-vector multiplication cuts the execution time for a single iteration approximately in half. Further accelerating vector operations continues to provide benefit, although it is less pronounced than in the gradient descent based algorithms. This is partially because Amdahl's law is coming into play with these smaller vector sizes due to the non-accelerable overhead of conjugate gradient.

Least squares is different from the three previous benchmarks in that it is dominated by dense matrix-matrix multiplication. Figure 6.4 shows the execution time of the non-iterative computation (the matrix-matrix multipli-

cation), the iterative computation (matrix-vector multiplication and vector operations), and the total execution time (assuming the maximum required iterations). The iterative computation is not shown for DOM since it is not affected by accelerating matrix-matrix multiplication. All three are normalized relative to the same computation done with no acceleration at all. Accelerating only matrix-matrix multiplication provides around an order of magnitude decrease in execution time at larger data sizes. However, this acceleration is so effective that the execution time of the iterative portion of the computation now makes up a substantial amount of the execution time. Thus it is worthwhile to further accelerate the remaining operations. Note that this is NOT an artifact of operating on small data sets. The matrix-matrix multiplication is $O(MN^2)$ while the iterative matrix-vector multiplication is $O(N^3)$. Assuming M and N scale at the same rate (as it does in this work where $M = 100N$), the ratio of matrix-matrix multiplication to matrix-vector multiplication will be roughly the same.

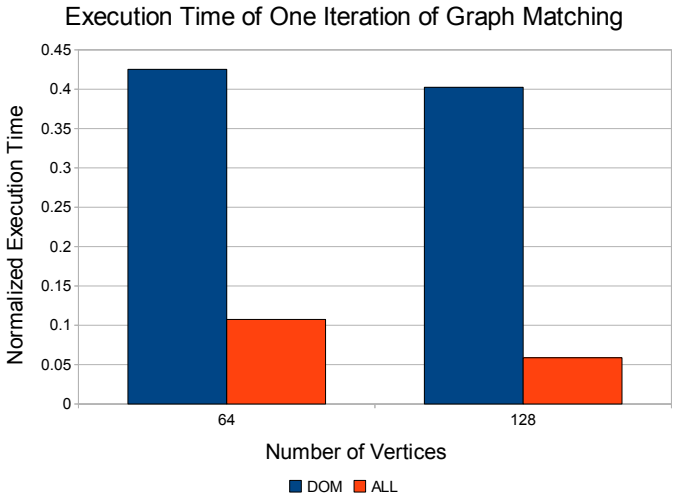

Figure 6.1: Execution time of a single iteration of gradient descent solving graph matching, normalized to the execution time of a single iteration without acceleration.

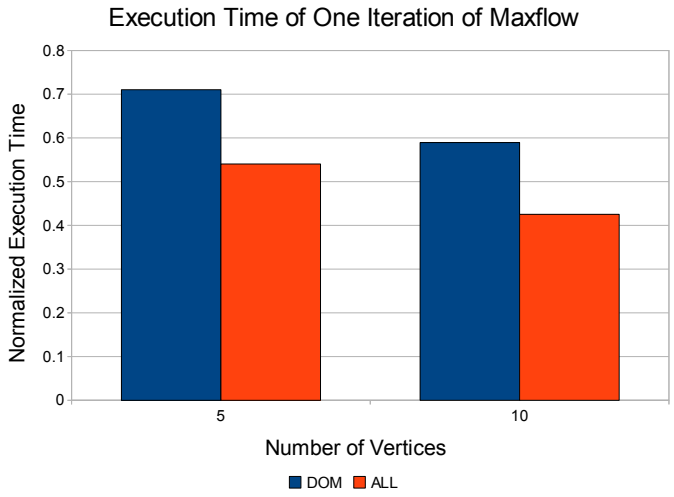

Figure 6.2: Execution time of a single iteration of gradient descent solving maxflow, normalized to the execution time of a single iteration without acceleration.

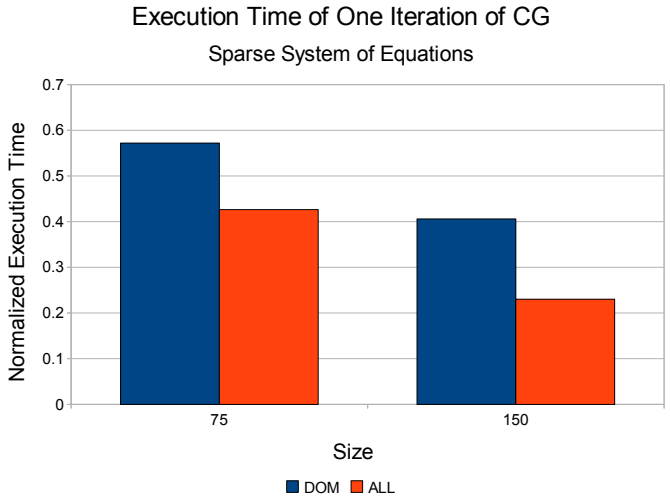

Figure 6.3: Execution time of a single iteration of conjugate gradient with a sparse matrix. Time is normalized to the same computation without acceleration.

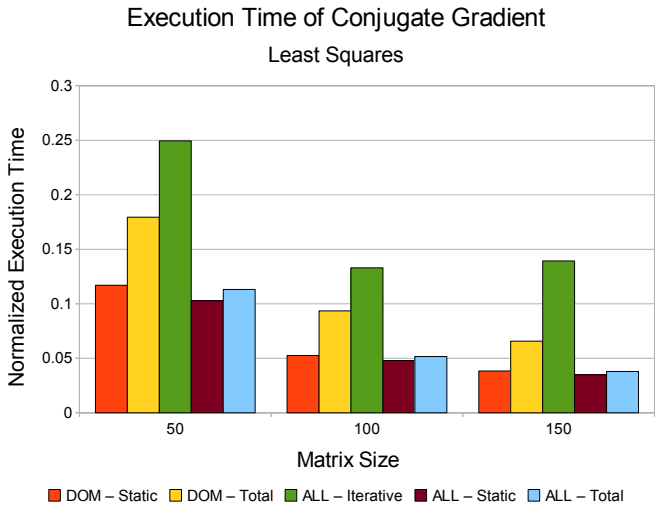

Figure 6.4: Execution time of the static computation, iterative computation, and total computation of conjugate gradient solving least squares. Times are normalized to the same computation without acceleration.

6.2 Comparison to Traditional Algorithms

The use of gradient descent to solve non-numerical optimization problems is a new idea which has been shown to potentially show error resilience benefits [1]. However, full convergence to the absolute correct answer is typically not practical due to the long execution time. We can get approximate answers, however, by running for a fixed number of iterations. Thus we want to compare the behavior of gradient descent on the solver engine to the traditional algorithm used to solve these problems.

Figure 6.5 shows the number of iterations of graph matching which equal the execution time of the baseline algorithm for no acceleration, matrix-vector acceleration only, and full acceleration. We can clearly see the number of iterations increase at least linearly with the input size when all operations are accelerated. Figure 6.6 shows the error in the total matching weight when run for a range of iterations. We see that with two sets of 64 vertices, within 100 iterations we settle to around a 1% error. This falls within our desired

limit of 168 iterations. Further iterations help very little, requiring orders of magnitude more iterations to appreciably increase accuracy. We see a similar trend with sets of 128 vertices, only taking around 200 iterations to reach similar levels of accuracy.

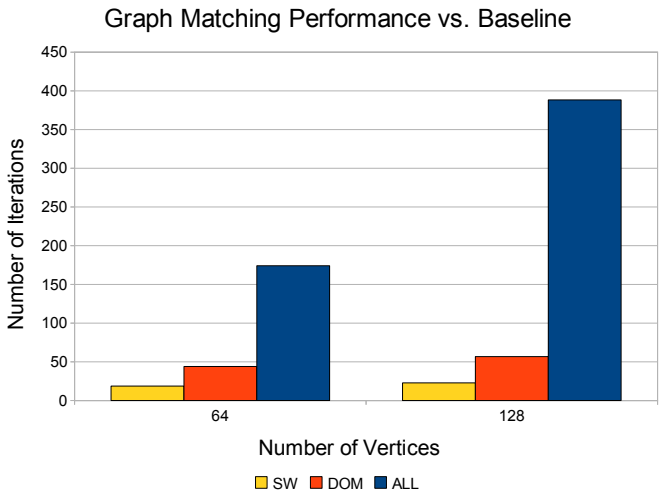

Figure 6.5: The number of iterations of gradient descent required to equal the execution time of the baseline algorithm for graph matching.

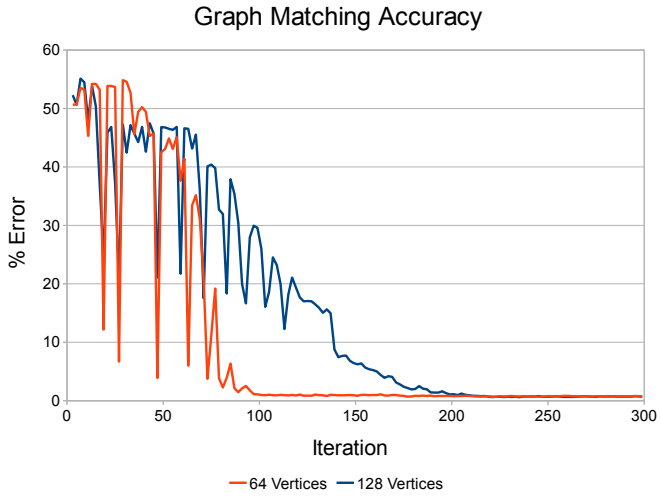

Figure 6.6: The error in the match weight returned by gradient descent compared to the correct maximum match weight.

Figure 6.7 shows the number of iterations of gradient descent needed to equal the execution time of the baseline algorithm for maxflow. We see that acceleration does not help as much as it does with graph matching, just more than doubling the number of iterations which will match the baseline's execution time from 8 to 18.5. We examine the accuracy of maxflow in Figure 6.8. We ultimately get within 3% error in the maximum flow in about 1500 iterations with 5 vertices and around 13% error in 5000 iterations with 10 vertices. We could stop gradient descent sooner and still get reasonable error; however, the problem with maxflow is that we must take into consideration both error in maximum flow as well as whether or not any flow violations have occurred (i.e. whether a node has more flow entering than leaving or vice versa). Figure 6.9 examines the relationship between flow violation and iterations. Unlike in graph matching, we cannot round to a valid (if sub-optimal) solution. Instead we must execute enough iterations that there is acceptably low flow violation. This seems to indicate that problems where a valid answer cannot be inferred easily from the results may not be suitable for gradient descent computation.

Similar to the use of gradient descent, the use of conjugate gradient to solve least squares is typically not optimal (although it can be close to traditional

approaches such as our baseline Cholesky decomposition). However it is close enough that we can run for the maximum number of iterations possible with CG and still show energy benefits. In fact, we are capable of showing even performance benefits due to the use of the solver engine. Figure 6.10 shows the execution time of the baseline algorithm, Cholesky decomposition and back substitution, alongside the execution time for CG with varying levels of accelerator support. We can clearly see that the accelerated CG offers real speedup benefits relative to Cholesky decomposition. CG has an extremely small error relative to Cholesky: $1.46 * 10^{-16}$ and $2.84 * 10^{-16}$ for 50 and 100 variables respectively.

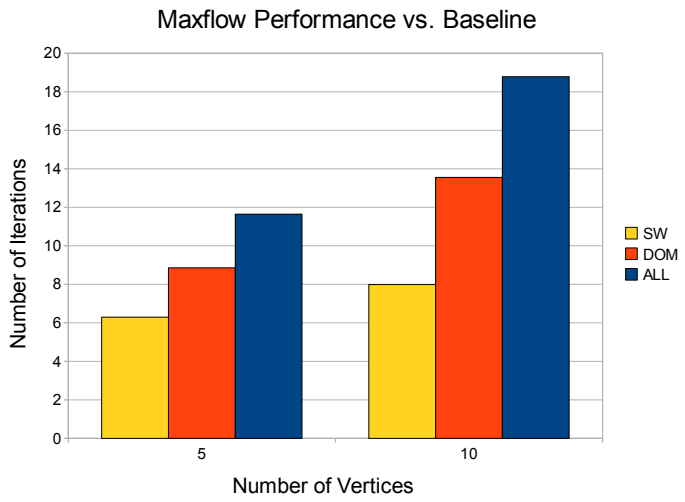

Figure 6.7: The number of iterations of gradient descent required to equal the execution time of the baseline algorithm for maxflow.

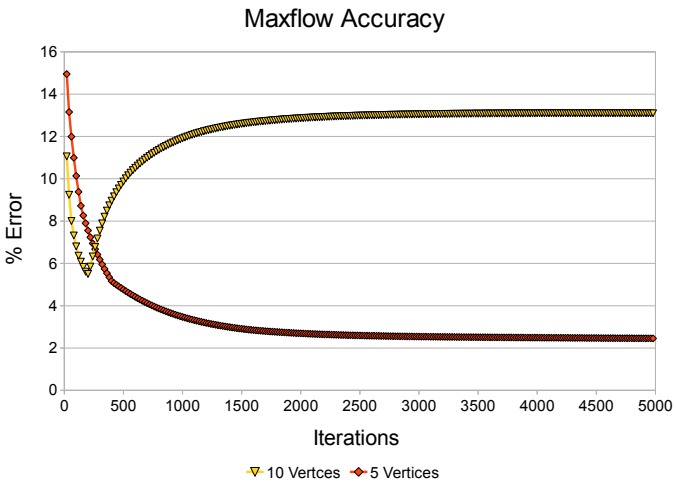

Figure 6.8: The error of the maximum flow returned by gradient descent compared to the correct maximum flow.

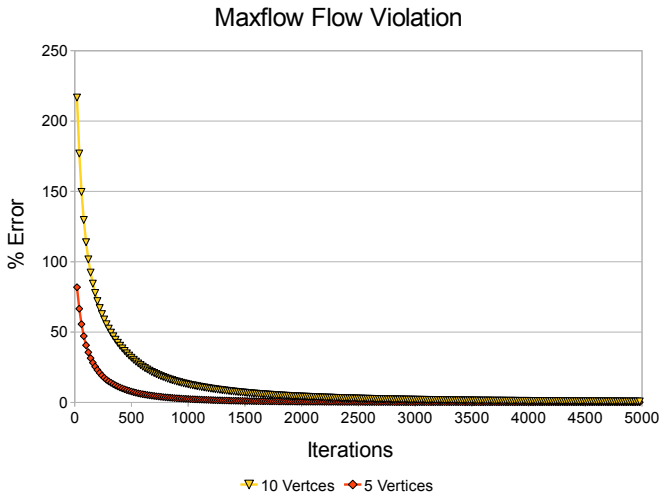

Figure 6.9: The ratio of total amount of flow violating the problem constraints to the correct maximum flow after gradient descent.

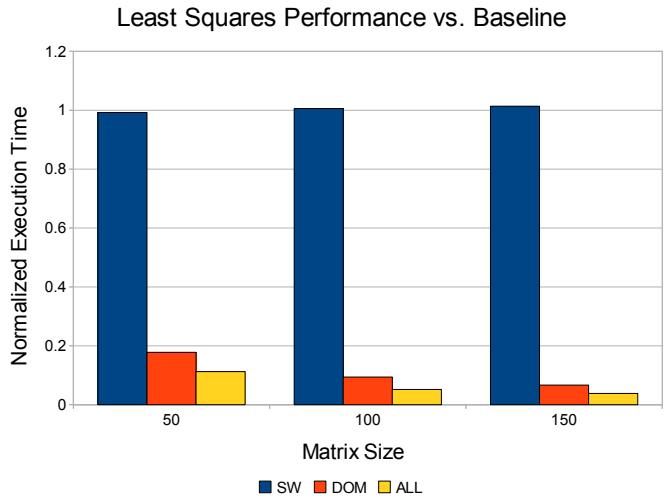

Figure 6.10: The execution time of least squares using conjugate gradient normalized to the execution time of using Choleskey decomposition.

6.3 Error Tolerance

One of the reasons executing problems as gradient descent is attractive is the innate error tolerance. As such, we would like to compare the results of both the baseline and the optimization algorithms in the presence of floating point errors. We executed graph matching with a range of fault rates, from .0067 to .33. The results are in Figures 6.11 and 6.12 for 64 and 128 vertices respectively. Even with an fault rate of .33 we achieve 4.25% and 5.25% error in 300 iterations for 64 and 128 vertices. The baseline proved to be unstable in the presence of floating point errors, causing segmentation faults. At lower error rates we did see some successful completions with low error in the match weight, but crashes were simply too common to collect any consistent results.

The gradient descent version of maxflow does extremely poorly in the presence of faults, unable to guarantee either low error or minimal flow violation. The accuracy and flow violation of the baseline algorithm and the gradient descent version (run for 5000 iterations) with 10 vertices is shown in Figure 6.13 for a range of error rates. At high error rates the baseline output does have 3% flow violation (i.e. the ratio of total flow violating the constraints to total correct maximum flow); however, at lower error rates it has flow violation comparable to the flow violation seen from gradient descent with no errors. In any case, the error and flow violation is substantially lower than that of the gradient descent version.

The error in least squares can vary by orders of magnitude, so averages are not particularly informative. Instead, we "bin" the output error of each data set by order of magnitude of the error, from $< 10^{-16}$ to $> 10^{2}$. Figure 6.14 shows the results for least squares run with 100 iterations of CG and the baseline Cholesky decomposition over a range of error rates. Because conjugate gradient is less error tolerant, we use floating point error rates which are much lower than with gradient descent. We discovered that Cholesky decomposition can actually result in arithmetic faults due to the square root operation if a value is negative due to floating point errors, so we modified it such that if it attempts to take the square root of a negative number it instead simply uses the value 1. Without this modification the faults will prevent Cholesky decomposition from completing consistently enough to measure error.

For solving a system of sparse equations, we do not compare the result to a baseline algorithm run without errors. However, we measure the absolute error by calculating $||Ax - b||_2$ and binning that similarly to least squares with ranges from 10^{-24} to 10^{6}. Figure 6.15 shows the results for the same range of errors as least squares on a 150x150 sparse matrix.

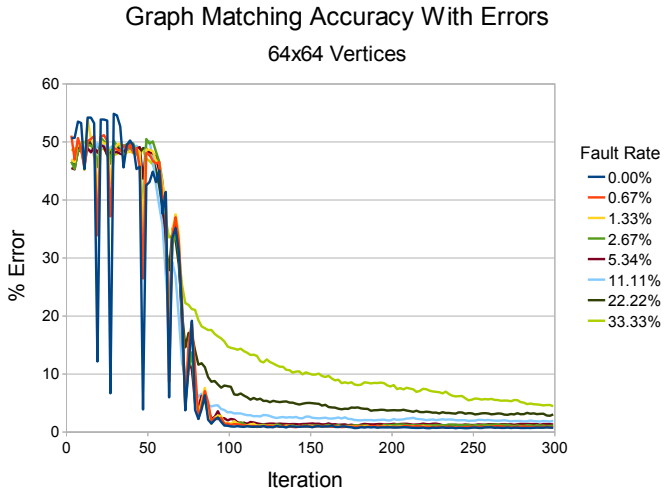

Figure 6.11: The error in the match weight for sets of 64 vertices calculated by gradient descent for a range of error rates.

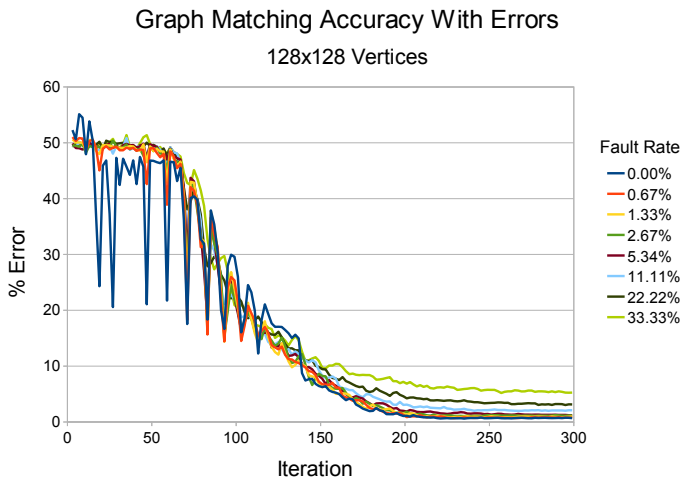

Figure 6.12: The error in the match weight for sets of 128 vertices calculated by gradient descent for a range of error rates.

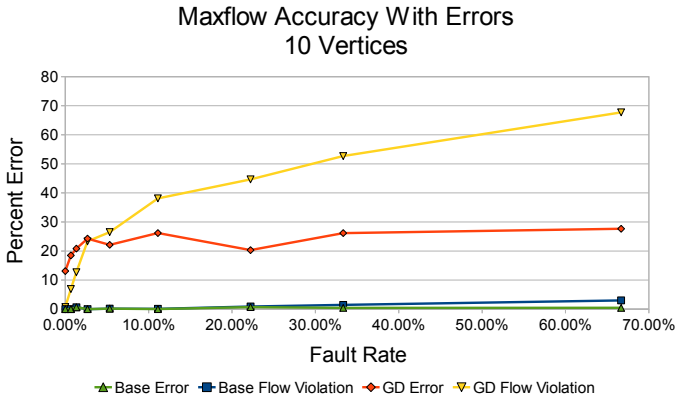

Figure 6.13: The error and flow violation in the result of the baseline algorithm and gradient descent after 5000 iterations for maxflow with 10 vertices.

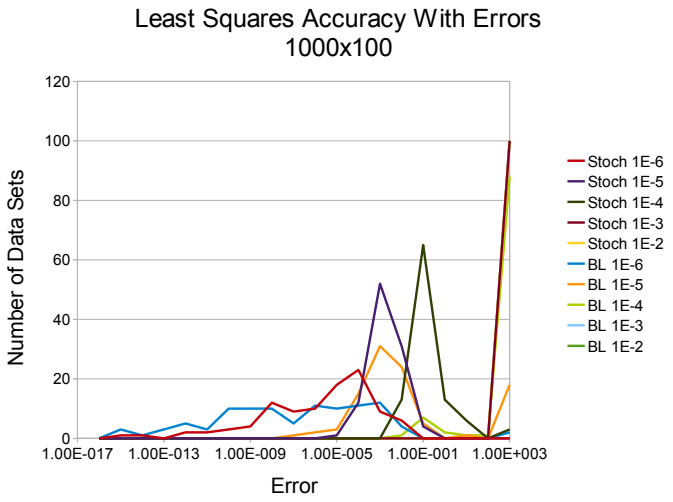

Figure 6.14: Frequency graph of the errors in the solution returned by 100 runs of CG and Cholesky decomposition relative to the correct answer calculated by Cholesky decomposition with no errors. The algorithms were used to solve the least squares problems for a random dense 1000x100 matrix.

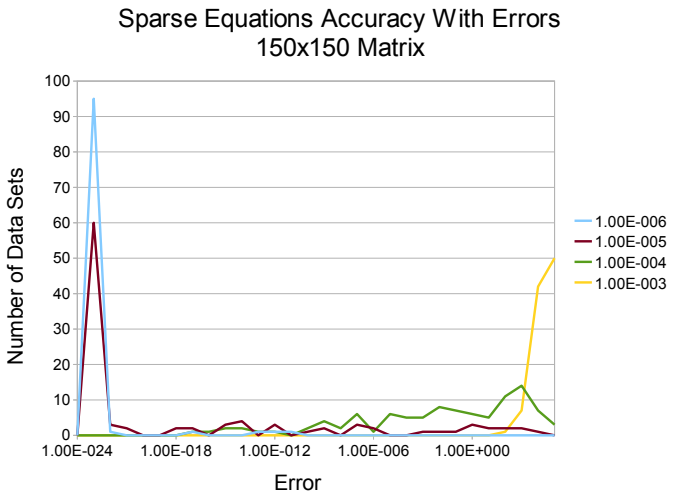

Figure 6.15: Frequency graph of $||Ax - b||_2$ for the solution returned by 100 runs of CG with errors for matrices of size 150x150.

6.4 Software Support Overhead

There are two factors to consider when determining if the overhead of using RBCOO is acceptable. First, the amount of padded zeros must be sufficiently low. Second, the overhead of converting a matrix into RBCOO format must be low enough that it does not wipe out the gains achieved by accelerating gradient descent and conjugate gradient.

Converting the sparse matrix used in graph matching into RBCOO format naively actually results in substantial overhead of 690% and 1490% extra values for 64 and 128 vertices. However if we shuffle the columns as per Section 4.3, we see a much lower overhead of 3.125% for both. This translates into a nearly 14% *decrease* in total memory usage relative to a CSR implementation.

We also consider randomly generated sparse matrices. We generated a set of sparse matrices with sparsities between .05 and .06 (the same way we generated the sparse matrices in solving systems of sparse equations) and calculated the overhead in terms of number of extra values. Figure 6.16 shows the results of this test for a range of matrix sizes, from 75x75 to 2048x2048. We see that at small matrix sizes the overhead can be substantial; however, larger matrix sizes provide more flexibility in scheduling and result in just over 10% padded zeros.

We also measure the execution time of performing the scheduling itself. 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). For the randomly generated sparse matrices, scheduling a 75x75 matrix takes 0.137 ms compared to the 0.143 ms needed to use it for (unaccelerated) conjugate gradient. With a 150x150 matrix, the execution times are 0.56 ms for scheduling versus 0.86 ms for conjugate gradient. Clearly the overheads for the smaller data sizes would be unacceptable. However the overhead of scheduling does not increase as quickly as the execution time of the actual computation. Thus for larger data sizes, we expect the scheduling overhead to be acceptable. (Note also that the scheduling algorithm is still a naive implementation, and a more optimized version may fare even better).

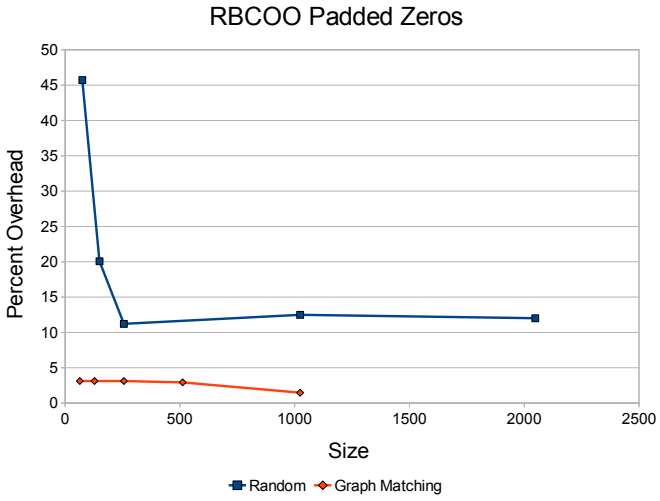

Figure 6.16: Overhead of padded zeros added when scheduling random sparse matrices of sparsity .05 and the sparse matrices used by graph matching (with column shuffling enabled). Size refers to the size of the matrix for the random sparse matrices and to the number of vertices for graph matching.

CHAPTER 7

RELATED WORK

7.1 Sparse Matrix Formats

Hierarchical sparse matrix (HiSM) is a matrix format designed for execution on machines with vector instructions [7]. It is a blocked format which allows it to store only relative indices for non-zero elements, saving a substantial amount of storage. It is also intended to eliminate indexed loads within blocks. The designers also propose some vector processor extensions to accelerate the performance of operations using hierarchical sparse matrices. The primary difference between HiSM and RBCOO is that RBCOO stores one contiguous array of non-zero elements and uses indices to point to the start of blocks. HiSM stores explicit pointers to point to blocks which may be in arbitrary locations in memory. 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 [9]. First, all non-zero values are shifted left and their column indices are stored. Then the rows are sorted by the number of non-zero values in them and the values and column indices are stored in column major format. In addition, arrays need to be stored indicating the initial row of each permuted row and indices to the columns. In a vector processor, each column can be read and used in a multiply-accumulate since each value is from a unique row. 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, consuming one value from each row, before the next value in a row may be consumed. RBCOO only requires an interleaving of accesses to the same row equal to the adder latency.

Row Blocked CSR (RBCSR) is a matrix format most similar to RBCOO, differing only in that the submatrices are stored in traditional CSR format instead of as a coordinate list [6]. It too is designed for execution on special purpose accelerator hardware; however, it was designed for the entire submatrix to be buffered dynamically distributed row by row to processing elements. RBCOO was specifically designed so that it could be streamed in without buffering due to the limited amount of memory available for buffering on a CPU. The downside to this is that RBCOO will have a larger memory footprint due to needing to store explicit row indices for each coordinate (as opposed to an index used to mark the start of rows in CSR).

7.2 Accelerators

Morris and Prasanna describe an FPGA based SPMV design based primarily on a dot product unit [10]. The matrix A and the vector x are buffered on their FPGA chip in its entirety. Each cycle, a vector of p values from a single row of A (and the corresponding values from x) is passed into a dot product unit consisting of p multipliers and an adder tree. The partial dot products output by the dot product unit are passed into an accumulator which draws from an L_axL_a array of values buffered on chip to accumulate with. This width of this array is necessary because there may be up to L_a values corresponding to a given row in the accumulator pipeline at a time. When a row has been reduced to L_a (or fewer) partial dot products, it enters an adder tree which finishes the reduction and outputs the row's final value.

This design features two fundamental flaws which make it unsuitable for use in the solver engine. First, it requires that A and x be stored in their entirety on chip. This restricts the maximum problem sizes which can be accelerated. While this restriction may be acceptable on an FPGA which has a comparatively large amount of storage, it is not acceptable on a CPU where area is at a premium. Furthermore, it requires that each row have at least p non-zero values or the dot product unit will be largely empty. This will result in extremely sub-optimal performance on matrices common in our

problems which may feature a small constant number of non-zero values per row. (Graph matching, for example, features only two non-zeros per row when multiplying with the transposed constraints matrix.)

DeLorimier and DeHon describe an FPGA based hardware accelerator designed to compute $A^i x$ using a CSR representation of A [11]. The design works by statically scheduling a set of rows of A onto each PE, then statically scheduling the rows onto each PE, interleaving accesses to the same row by a factor of L_a. After a single SPMV execution, the output vectors are passed to PEs requiring them for the next multiplication via an on-chip ring interconnect. The idea of static scheduling was highly influential on the sparse matrix-vector design used in the solver engine. However, the design does require that the matrix and input and output vector be stored on chip in their entirety. Like in other designs, this is an extremely limiting feature which makes it unsuitable for the solver engine.

Sun et al. propose an FPGA design for a hardware sparse matrix accelerator which utilizes a row blocked CSR format [6]. The accelerator fully buffers each submatrix and dynamically assigns rows to each PE. Each PE reduces an entire row of the submatrix down to L_a values, then receives a new row from the centralized matrix manager. A centralized result controller further reduces the L_a values output by each PE using an adder tree and adds it to the row's running total stored in the result BRAM by previous submatrices in the same row.

This accelerator design, while similar to the one presented in this thesis, has several features which make it undesirable for our purposes. First, it requires the entire submatrix be buffered so it can be dynamically distributed to processing elements. This would necessitate the use of an unacceptably large amount of on-chip storage or the use of small submatrices (which will have negative effects on performance due to the poorer ability to distribute small submatrices to processing elements). Secondly, for efficient operation, it requires there to be more than L_a values per row of the submatrix. The first L_a values of each row pass through the multiplier and enter the accumulator which accumulates them with 0. After the accumulator's pipeline fills up it can route its output back to its input, performing useful operations. However, if there are few values in a particular row of the submatrix, the accumulator is active without performing any useful work. As previously noted, rows with very few values are not only probable, but extremely common in some

problems. 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.

Much work has been done to enable efficient execution of sparse matrix-vector multiplication on GPU's [12, 13]. GPUs generally outperform CPUs on sparse matrix vector multiplication, largely due to their higher peak memory bandwidth compared to general purpose CPUs. However, they do not make the most efficient use of peak floating point capacity. For example, Bell and Garland's highest reported single-precision floating-point performance on a GTX280 was 36 GFLOP/s, well below the GPUs peak of 933 [12]. The solver engine, 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.

Attarde proposes an FPGA based SPMV design which is explicitly designed for matrices which cannot be buffered on chip in their entirety [14]. The design exploits the fact that the matrices expected to be used will feature clusters of dense submatrices and so proposes handling clusters of dense submatrices separately from the remaining highly sparse non-zero values. The relatively dense blocks are stored in Blocked-Column-Row format. The format is similar to RBCOO in that each element of a submatrix stores both row and column indices relative to the start of the submatrix; however, some additional information is stored for operation in the accelerator design. Also, similar to the SPMV design used in the solver engine, the matrix is pre-processed to divide the matrix into submatrices and those submatrices are scheduled onto processing elements.

During execution, values in the dense block are passed out to each PE to be operated on. Each PE stores a copy of the output vector for the current set of rows and accumulates computations into it. When all submatrices in a set of rows are finished, the output vectors of each PE are transferred out and summed using an adder tree to get the final piece of the output vector corresponding to those rows. While the design performs favorably, it relies on matrices with dense subblocks to exploit. While this may be a valid strategy in general computations, the problems we are targeting with the solver engine do feature pathological sparsity patterns.

In contrast to sparse matrix operations, dense matrix multiplication and

matrix vector multiplication map well onto specialized hardware and GPUs. The dense matrix multiplication design proposed by Kumar et al., which is based on the rank one update algorithm, heavily inspired the design of the solver engine [15]. The largest fraction of work has focused specifically on matrix multiplication on FPGAs [16, 17, 18, 19, 20, 21]. Other works have also considered matrix-vector multiplication and dot products [22] on FPGAS, or matrix multiplication on GPUs [23]. In fact, Nvidia has released a CUDA based library of BLAS implementations called CUBLAS to encourage the use of GPUs for dense matrix operations [24].

CHAPTER 8

CONCLUSION

The solver engine described in this thesis represents a novel and effective solution for the acceleration of numeric optimization algorithms. Because it is intended to be integrated onto a traditional CPU die, it solves several problems that traditional sparse matrix accelerator designs do not face. It is able to work efficiently with low amounts of on-chip buffer space. It is also able to accelerate a number of different operations for a number of different algorithms because it cannot be reconfigured for each desired algorithm as FPGA based accelerators can. Furthermore, it addresses several general flaws in the design of traditional accelerators. It avoids the use of shared, centralized structures which can limit the scalability of designs. It is also capable of accelerating sparse matrices with sizes and sparsity patterns which might not be effectively handled by other designs.

Many features of the solver engine are dependent on the use of a novel sparse matrix storage format, Row Blocked Coordinate List. This novel storage format is efficiently schedulable on the same accelerator hardware which handles dense matrix and vector operations. It incurs less than 15% overhead in terms of extra padding zeros on random matrices of sparsity 0.05 and less than 5% overhead on the matrix used for graph matching. The latter is particularly interesting as the sparsity pattern of the graph matching matrix is pathological to several accelerator designs.

We derived expected performance characteristics for an example solver engine with 16 processing elements operating at 250 MHz with 32 GB/s of memory bandwidth. We used a modified version of M5 to profile the behavior of the general purpose processor and calculated the behavior of the solver engine itself. We examined four benchmarks, bipartite graph matching, maxflow, least squares, and solving systems of sparse equations. We show a 16x speedup for graph matching, a 2.4x speedup for maxflow, a 25x speedup on least squares, and a 4.3x speedup on solving systems of sparse

equations. These results demonstrate that the solver engine is indeed competitive with traditional accelerator designs despite the additional challenges it solves.

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, pp. 161–170.

[2] M. Hestenes and E. Stiefel, "Methods of conjugate gradients for solving linear systems," *J. Res. Nat. Bur. Stand.*, vol. 49, no. 6, pp. 409–436, 1952.

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

[4] D. P. Dobkin and S. P. Reiss, "The complexity of linear programming," *Theor. Comp. Sci.*, vol. 11, no. 1, pp. 1–18, 1980.

[5] L. Zhuo and V. K. Prasanna, "High performance linear algebra operations on reconfigurable systems," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, 2005, p. 2.

[6] J. Sun, G. Peterson, and O. Storaasli, "Sparse matrix-vector multiplication design on FPGAs," in *Proceedings of the 15th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007, pp. 349–352.

[7] P. Stathis, S. Vassiliadis, and S. Cotofana, "A hierarchical sparse matrix storage format for vector processors," in *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003, p. 61.

[8] "The OpenCV library," Aug. 2010. [Online]. Available: http://opencv.willowgarage.com/documentation/index.html

[9] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. Philadelphia, PA: SIAM, 1994.

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

[11] M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *FPGA '05: Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays*, 2005, pp. 75–85.

[12] N. Bell and M. Garland, "Efficient sparse matrix-vector multiplication on CUDA," NVIDIA Corporation, Santa Clara, CA, USA, Tech. Rep. NVR-2008-004, Dec. 2008.

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

[14] S. Attarde, "Accelerating double precision sparse matrix vector multiplication on FPGAs," M.S. thesis, India Institute of Technology-Bombay, 2010.

[15] 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*, 2009, pp. 341–346.

[16] 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.

[17] 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*, 2005, pp. 86–95.

[18] 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, pp. 1–10.

[19] 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*, 2007, pp. 97–100.

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

[21] 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*, 2005, pp. 113–122.

[22] 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.

[23] S. Barrachina, M. Castillo, F. Igual, R. Mayo, and E. Quintana-Orti, "Evaluation and tuning of the level 3 CUBLAS for graphics processors," in *Proceedings of the International Parallel and Distributed Processing Symposium*, April 2008, pp. 1–8.

[24] *CUDA CUBLAS Library*, NVIDIA, Santa Clara, CA, USA, Aug. 2010.