AN FPGA-BASED IMPLEMENTATION OF THE MINRES ALGORITHM

David Boland, George A. Constantinides *

Electrical and Electronic Engineering Department,
Imperial College London
dpb03@ic.ac.uk, george.constantinides@ieee.org

ABSTRACT

Due to continuous improvements in the resources available on FPGAs, it is becoming increasingly possible to accelerate floating point algorithms. The solution of a system of linear equations forms the basis of many problems in engineering and science, but its calculation is highly time consuming. The minimum residual algorithm (MINRES) is one method to solve this problem, and is highly effective provided the matrix exhibits certain characteristics. This paper examines an IEEE 754 single precision floating point implementation of the MINRES algorithm on an FPGA. It demonstrates that through parallelisation and heavy pipelining of all floating point components it is possible to achieve a sustained performance of up to 53 GFLOPS on the Virtex5-330T. This compares favourably to other hardware implementations of floating point matrix inversion algorithms, and corresponds to an improvement of nearly an order of magnitude compared to a software implementation.

1. INTRODUCTION

The solution to a system of linear equations of the form $Ax = b$ (where A is an $N \times N$ matrix, while x and b are $N \times 1$ vectors) forms the basis of a large number of problems, most notably in the realm of scientific computing. In order to obtain accurate results, it is often desirable to solve such problems using floating point representation. As research has identified trends indicating the floating point performance of FPGAs will significantly exceed that of traditional processors [1], using an FPGA to generate an efficient solution to these problems is an important field of study.

There are two main families of methods to solve this type of linear algebra problem: direct methods and iterative methods. Direct methods find the solution in one shot, typically via some form of computationally intensive matrix factorization, whereas iterative methods refine a solution with each iteration. FPGA implementations for a variety of direct methods have been created, both in fixed point: Cholesky [2] and floating point; Gauss-Jordan [3], LU decomposition [4] and QR decomposition [5]. The salient features of these implementations are highlighted in Table 1.

However, it is the use of iterative methods to solve linear equations that is of growing interest within the field of FPGAs for these methods largely consist of multiply accumulate operations as opposed to division operations, which are often required in direct methods (for example any pivoting operation). It is well known that multiply-accumulate operations are highly suited to FPGAs due to their use in DSP type applications, whilst the suitability of performing this type of operation in floating point is justified by studies demonstrating efficient matrix multiplications using floating point [6, 7]. Furthermore, iterative methods initialized with a good initial guess will converge much faster and this is the case in many scientific computing problems, especially in optimisation [8]. For these reasons there has been growing interest in using one specific iterative method, the Conjugate Gradient algorithm, to solve these problems [9, 10]. However, it can be shown that the Conjugate Gradient algorithm breaks down if the A matrix is not symmetric positive definite, and hence a more general method is desirable.

This paper examines the MINRES algorithm [11], which is another iterative algorithm that is an efficient solver for cases where the A matrix is symmetric. The motivation for choosing this specific algorithm is that it strikes a good balance between complexity and generality in that many scientific computing problems could be mapped to problems with symmetric matrices that are not necessarily positive definite [8].

To the best knowledge of the authors, there has as yet been no implementation of the MINRES algorithm on an FPGA and therefore any direct comparison is not possible. However, compared to alternative hardware implementations of matrix inversion, this implementation achieves a significantly higher performance, as shown in Table 1. It should be noted that some other implementations can process larger matrix orders. However, this depends upon the exploitation of sparse structure in the matrices or through using off-chip RAM to store intermediate results. Sparse matrix solvers can handle much larger matrices as it is not necessary to hold or operate on any zeros in the matrix, but are less general by

*The authors would like to acknowledge the support of the EPSRC (Grants EP/C549481/1 and EP/E00024X/1)

Table 1. Comparison of Floating Point Matrix Inversion Methods

Method			Year	GFLOPS	Vs. Software	Max Order	Sparsity	Off-chip RAM	Device	Requirements of A
Direct	Gauss-Jordan	[3]	2006	N/A	4 ×	1700	Dense	Yes	Virtex II	Non-Singular
	LU	[4]	2006	2.6	6 ×	1000	Dense	Yes	Stratix II	Non-Singular
	QR	[5]	2008	35	N/A	12	Dense	No	Virtex5	Non-Singular
Iterative	Conjugate Gradient	[9]	2006	N/A	1.3 ×	2000	Sparse	Yes	2×Virtex II	Symmetric Positive Definite
	Conjugate Gradient	[10]	2008	35	5 ×	58	Dense	No	Virtex5	Symmetric Positive Definite
	Minres	this	2008	53	8.8 ×	145	Dense	No	Virtex5	Symmetric

definition. Using off-chip RAM to hold the intermediate results enables larger matrices to be held, but the I/O requirements to load onto the FPGA creates a bottle-neck (typically determined by the number of off-chip RAMs) upon performance and efficiency, typically leading to low speedups, as shown in Table 1 for [3] and [4]. The embedded memories on modern FPGA devices, however, now have the capacity to buffer large matrices on chip, a technique we exploit to break this bottleneck, at the cost of limiting the order of matrix to up to 145. This order of matrix is considered relatively large for *dense* problems and is sufficient for many applications which depend upon matrix inversion [12], and could be used as a building block for solving larger systems. Moreover, it is 2 to 12 times larger than previous dense on-chip solvers.

The main contributions of this paper are:

- A demonstration of the suitability of the MINRES algorithm for use on an FPGA,
- An analysis of the design decisions and trade-offs involved to create an optimum floating point implementation of the MINRES algorithm in hardware, including efficiency and pipeline depth,
- A design for solving multiple dense systems of linear equations in a pipeline for orders up to 145 using the MINRES algorithm, with results demonstrating a *sustained* performance, taking into account I/O overhead, of up to 53 GFLOPS.

This paper describes the MINRES algorithm and briefly highlights both the advantages and the additional complexities over its closest relative, the Conjugate Gradient algorithm, in Section 2. The hardware implementation is described in Section 3, along with its associated results in Section 4. Finally Section 5 concludes this paper.

2. MINRES ALGORITHM

The MINRES algorithm finds a (potentially approximate) solution x_k, to the system of equations $Ax = b$ (where A is an $N \times N$ matrix, x and b are $N \times 1$ vectors) by performing a minimisation of the residual $||b - Ax_k||_2$ in the two-norm over a Krylov Subspace [13]. It will generally converge to a very accurate solution without the need to calculate the entire subspace and hence the subspace is built iteratively, using the Lanczos process [11]. Overall, the pseudo code is described in Fig. 1, with the major sections of the algorithm highlighted.

The Conjugate Gradient Method can also be interpreted as an algorithm that makes use of the Lanczos process and therefore there are some similarities between the two methods [13]. For cases where it is desirable to compare hardware implementations of these two methods, it is important to highlight the two major differences in terms of hardware costs, both of which are a result of working with the two-norm. Firstly normalisation is required, resulting in square root operations. Secondly, it results in a three-term recurrence as opposed to the two-term recurrence in the Conjugate Gradient algorithm; this increases storage requirements. Thus the MINRES algorithm trades an increase in circuit complexity for the ability to solve a wider class of problems.

3. IMPLEMENTATION

The optimal hardware implementation is dependent upon a number of factors - number of resources, latency (in terms of cycles per iteration), throughput and efficiency (in terms of the amount of time resources are in use). The design described aims to achieve a good balance in terms of optimising these factors. The following sections detail the main considerations and potential trade-offs between these factors and justifies any major decisions. To aid description for this section, a diagram of the circuit is shown in Fig. 2.

3.1. Floating Point Units

Xilinx Core Generator was used to generate the floating point components for the circuit. This environment enables the user to trade latency for maximum clock frequency. Provided the pipeline remains full, due to the increased clock frequency, a component with a higher latency potentially has a higher throughput.

It is possible to maximise the amount of time the pipeline is full by multiplexing several independent problems into the device. As a result, it was chosen to set all floating point components to work to their maximum latency, for an implementation with a high throughput that could potentially operate on multiple problems simultaneously would generally be more useful than a small reduction in latency for a single

% Initialisation

$v_0 = 0$; $v_1 = b - Ax_0$

$\beta_1 = ||v_1||_2$

$\eta = \beta_1$

$\gamma_0 = 1$; $\gamma_1 = 1$

$\sigma_0 = 0$; $\sigma_1 = 0$

$w_0 = 0$; $w_{-1} = 0$

$i = 1$

while $\eta > \varepsilon$

% Calculate Lanczos Vectors

$v_i = \frac{v_i}{\beta_i}$

$\alpha = v_i^T A v_i$

$v_{i+1} = Av_i - \alpha v_i - \beta v_{i-1}$

$\beta_{i+1} = ||v_{i+1}||_2$

% Calculate QR Factors

$\delta = \gamma_i \alpha_i - \gamma_{i-1}\sigma_i\beta_i$

$\rho_1 = \sqrt{\delta^2 + \beta_{i+1}^2}$

$\rho_2 = \sigma_i\alpha_i + \gamma_{i-1}\gamma_i\beta_i$

$\rho_3 = \sigma_{i-1}\beta_i$

% Calculate New Givens Rotations

$\gamma_{i+1} = \frac{\delta}{\rho_1}$

$\sigma_{i+1} = \frac{\beta}{\rho_1}$

% Update Solution

$w_i = \frac{v_i - \rho_3 w_{i-2} - \rho_2 w_{i-1}}{\rho_1}$

$x_i = x_{i-1} + \gamma_{i+1}\eta w_i$

$\eta = \sigma_{i+1}\eta$

$i = i + 1$

end

Fig. 1. MINRES Algorithm [14].

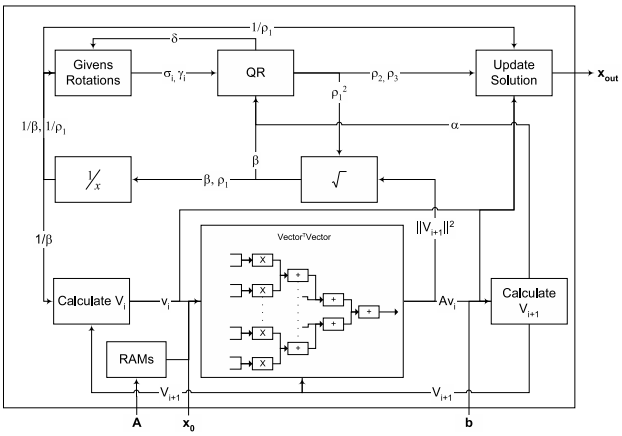

Fig. 2. Circuit Data Flow.

problem. To this end, it was assumed that in all situations there would be sufficient problems available to the circuit to fill the pipeline. This assumption is valid in many problems requiring matrix inversion, for example for the control community [12]. The number of independent problems required to keep the pipeline busy is discussed in Section 3.4, where it is shown that it approaches 4 for large problems.

3.2. Datapath

In order to ensure the clock frequency is as close as possible to the maximum available for the floating point components, it was decided whenever possible to use a dedicated component to perform each operation as this minimises wiring and multiplexers between the floating point components. The exception to this is for expensive operations. The square root and division operators consume a large number of resources, whilst as seen in the pseudo code (Fig. 1), they are only used during four operations. Furthermore, if the vector division is calculated by a single inversion (to compute $1/\beta$ and $1/\rho_1$) followed by multiplication by the result, these two operators are only used twice per iteration per problem. Therefore it is both possible and desirable to re-use these components.

3.3. Parallelisation

It is clear from the pseudo code (Fig. 1) that the calculation of the Lanczos vectors is independent of the operations to perform the QR decomposition, Givens Rotations and updating the solution. Therefore it is possible for all these parts of the circuit to work in parallel, and this reduces the overall latency to be that of the Lanczos iteration.

Theoretically, it is also possible to parallelise matrix and vector operations, however, the limited number of resources on an FPGA mean that for large N it is not possible to parallelise every operation. The operation of highest computational complexity is the *Matrix* $\times$ *Vector* multiplication within the Lanczos iteration. Though a dedicated component to perform this calculation would significantly reduce the latency of the circuit, the resource usage would scale heavily with N ($\Theta(N^2)$ in terms of multipliers and adders) and also the I/O requirements for such an implementation would quickly exceed the capabilities of the FPGA, making it highly unscalable. Instead it was chosen to overlap the *Matrix* $\times$ *Vector* operation in a pipelined fashion within a dedicated *Vector*^{T}*Vector* circuit. This involves a dedicated vector multiplier and an adder sum tree (as shown in Fig. 2), which has a cost of N multipliers and $N-1$ adders, but reduces the latency to be $\Theta(N)$ instead of $\Theta(N^2)$. In comparison, if one were to parallelise the other vector operations, it would only remove a constant latency at a cost of an increased use of N operators. Furthermore, this V^TV circuit is re-used when calculating the norm, saving resources.

Using this parallelism, the total number of floating point components is given in equation (1) and the overall latency of the circuit is given by equation (2), where P is the number of independent problems to be stored in the pipeline. Referring to (2), the factor $3N$ is a result of N cycles needed to perform the Matrix-Vector product in the pipeline as described above, as well as $2N$ cycles for the two series to parallel conversions (for v_i and v_{i+1}) which are inputted to this circuit (Fig. 2); the factor $c_1\lceil \log_2 N \rceil$ is a result of the summation tree in the V^TV circuit (Fig. 2); and the factor P is a result of the re-use of the V^TV circuit for the norm meaning that for one cycle per problem it will not be performing a *Matrix* $\times$ *Vector* computation. The values c_1 and c_2 are constants representing the latency of the other operations.

$$\text{Number of Floating Point Operators } = 2N + 26. \quad (1)$$

$$\text{Total Latency (cycles) } = 3N + c_1\lceil \log_2 N \rceil + P + c_2. \quad (2)$$

3.4. Pipelining

As mentioned in Section 3.1, in order to maximise the efficiency of the circuit, the pipelines in the floating point components must continually be as full as possible, and this can be achieved by multiplexing P problems into the system. Due to the high resource usage of the V^TV circuit (N multipliers and $N-1$ adders), in order to maintain a high efficiency, the minimum pipeline depth is chosen such that this component is always in operation.

Using the V^TV circuit described in Section 3.3, for P problems this circuit will be in operation for $PN+P$ cycles. Thus an effective way to determine the minimum pipeline depth is to match $PN + P$ with the latency for one iteration (equation (2)), for this ensures the V^TV circuit will be operating on other problems until it is again needed for the subsequent iteration of the first problem. Using this method, the pipeline depth is given by equation (3). It should be clear from equation (3) that the depth of the pipeline tends to the value 4 as N tends to infinity, implying for large matrices only a small number of independent problems are required to keep the pipeline busy. The depth of the minimum pipeline found by this method is shown in Fig. 3.

$$\text{Pipeline Depth } (P) = \left\lceil \frac{3N + c_1\lceil \log_2 N \rceil + c_2}{N} \right\rceil. \quad (3)$$

A graph illustrating the efficiency of the circuit using this minimum pipeline is shown in Fig. 4. This demonstrates that in all situations a high efficiency (above 70%) is achieved and the efficiency increases with matrix order,

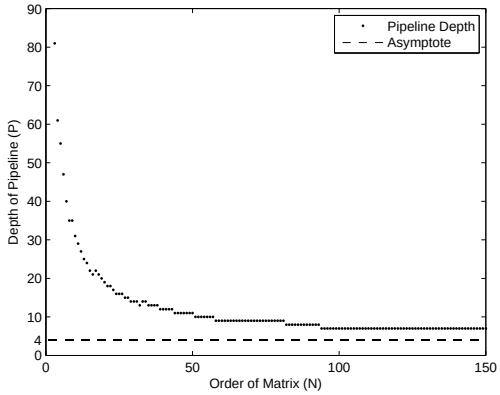

Fig. 3. Plot of Pipeline Depth for increasing Matrix Order. This is the minimum number of problems required for the V^TV circuit to always be in full operation.

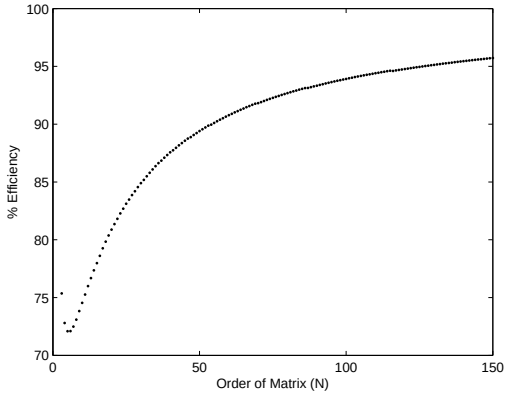

Fig. 4. Plot of percentage Efficiency for increasing Matrix Order using the Pipeline Depth (3).

tending to 100%. This growth in efficiency is due to the number of operators for the V^TV circuit increasing with N, whilst the number of operators working on vectors and scalars remains constant (equation (1)). This implies for large N, the number of operators is dominated by the V^TV circuit, and this will always be in operation by design. Such a performance is highly unlikely to occur in any software implementation due to various delays such as cache misses. Indeed the efficiency of the order of 40 to 60 % is common in software, even for a highly optimized implementation, as shown in [15].

It should be noted that the efficiency reaches a minimum for $N = 6$; below this order, the total latency (equation (2)) is small and hence any operators that work serially on vectors are used relatively efficiently.

3.5. I/O Considerations

The major consideration with regard to I/O is to ensure the V^TV circuit will continually have input data. As a result of using single precision floating point representation (requiring 32 bits) and the limited off chip I/O bandwidth in typical FPGA computing platforms, all elements of the A matrix cannot be loaded in parallel. Instead the A matrix is held in on-chip RAM (as shown in Fig. 2), organised as a parallel bank of RAMs, each storing a column of the matrix for P problems.

The A matrix for a given problem is re-used during each iteration and hence the I/O requirement is determined by the need to be able to load the set of A matrices into the FPGA for the next set of P problems within the time period to solve the first P problems. It is important to note that this method requires the RAMs to be twice as large as necessary for any single iteration (half of the RAM loads the next set of data whilst the other half is in use).

It can be shown that the maximum number of iterations for a given problem to reach a solution is N, but the method will generally converge before this worst case. Denoting the number of iterations executed as I (where $I \leq N$) and considering the latency for one iteration, after matching for pipeline depth (Section 3.4), to be $PN + P$; then the total time available to load the data is $I(PN + P)$. The total amount of data transferred is the A matrices, the two vectors b and x_0, and the final output vector x_{out} for P problems; a total of $P(N^2 + 3N)$ elements. Thus the I/O requirement is given by equation (4a).

$$\text{I/O Req} = \frac{P(N^2 + 3N)}{I(PN + P)} \text{words/cycle.} \tag{4a}$$

$$\approx \frac{N}{I} \text{words/cycle.} \tag{4b}$$

$$= 1.1\frac{N}{I} \text{GBytes/s.} \tag{4c}$$

In order to consider this in terms of available I/O technology, this is also shown as Bytes/second (4c), using the clock frequency (Section 4.2). While I is data dependent in general, this I/O bandwidth is, in our experiments, well below that provided by typical FPGA computing platforms, such as PCI-express (8 GBytes/s).

4. RESULTS

4.1. Resource usage

The circuit was placed and routed, targeted to the Virtex5 LX 330T. Fig. 5 shows the resource use in terms of DSP48Es, slices and BRAMs. The growth of slices and DSP48Es with matrix size is highly linear. This is to be expected, for the growth in floating point units is linear (equation (1)), and this design is dominated by floating point components.

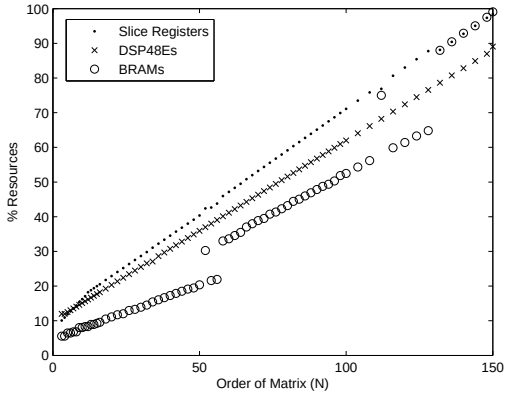

Fig. 5. Plot of percentage Resource Usage on a Virtex 5 LX 330T for increasing Matrix Order.

The BRAM usage, as seen in Fig. 5, grows in a piecewise linear fashion, with occasional jumps. This is a result of storing the N columns of the A matrix for P problems (which translates to N RAMs each storing PN elements) dominating the BRAM use. Thus linear growth is caused by the number of columns increasing with N, whilst the large jumps occur when PN exceeds the physical sizes of the BRAM. Together, this corresponds to a quadratic growth asymptotically, but for orders up to 145, this is not significant and is only seen as three jumps. The reason the BRAM usage is not monotonically increasing with matrix order is due to the decreasing pipeline depth (3) reducing the number of A matrices that must be stored.

4.2. Performance

The maximum clock frequency reported after place and route is approximately 250 MHz for small matrix orders ($N \leq 16$), after which the speed slowly degrades with N, approximately in a linear fashion, to about 170MHz for the largest matrix orders. This high performance is likely to be a result of the simple datapath as described in Section 3.2, combined with the deep pipelining allowing a large degree of retiming freedom, whilst the degredation is simply likely to be a result of the increased size of the circuit requiring increased wiring. Given this frequency, the maximum matrix order of 145, the number of floating point operators given in equation (1) and the efficiency of 96% (Fig. 4), it is possible for this circuit to achieve a sustained performance of 53 GFLOPS.

As has been demonstrated in Sections 3.3 and 3.4 this hardware implementation involves significant parallelism to reduce the latency of the iteration, and also works upon multiple problems in a pipelined fashion. In order to quantify this improvement, the performance of the hardware is com-

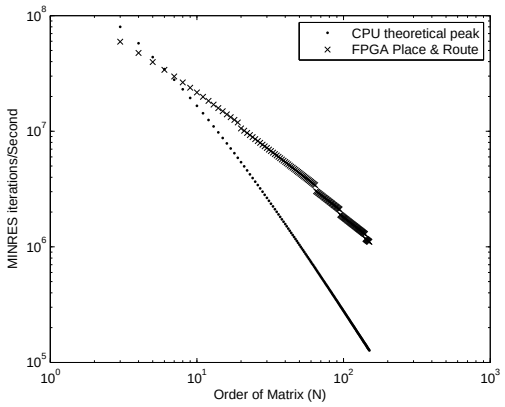

Fig. 6. Comparison of Hardware and Software Performance.

pared to the peak theoretical performance of a software implementation [15]. The performance metric is MINRES iterations per second.

The software model is based upon the peak theoretical floating point performance of a Pentium IV running at 3.0 GHz (6 GFLOPS) [15], applied to the number of floating point operations given in equation (5) which is found by a simple operation count of the algorithm described in Fig. 1. The hardware model assumes a pipeline depth given by equation (3) and an operational frequency of given by the place and route results.

$$\text{\#Floating Point Operations} = 2N^2 + 15N + 14. \quad (5)$$

A comparison of these two models is shown in Fig. 6. This shows that as a result of the parallelism reducing the latency, the performance is greater than a software implementation even for orders as low as $N = 7$. Due to the increased efficiency and parallelism, this performance improvement grows to almost an order of magnitude over the peak theoretical performance of a software implementation.

5. CONCLUSION

This paper has demonstrated that the MINRES algorithm can be used effectively as a means to solve a system of linear equations in hardware. It has discussed in detail several design decisions and described how significant parallelism has been used to reduce the latency of the circuit by a factor of N and through pipelining it is possible to achieve an efficiency which will tend to 100%, with values of 96% achieved in practice. Finally, it has described a circuit using these considerations that exhibits a sustained performance of 53 GFLOPs, which is superior to previous work and predicts a performance improvement of nearly an order of magnitude compared to the peak theoretical software implementation.

6. REFERENCES

[1] K. Underwood, "FPGAs vs. CPUs: trends in peak floating-point performance," in *Proc. Int. Symp. Field Programmable Gate Arrays*, 2004, pp. 171–180.

[2] P. Salmela, A. Happonen, A. Burian, and J. Takala, "Several approaches to fixed-point implementation of matrix inversion," *Proc. Int. Symp. Signals, Circuits and Systems*, vol. 2, pp. 497–500, July 2005.

[3] G. de Matos and H. Neto, "On reconfigurable architectures for efficient matrix inversion," *Proc. Int. Conf. Field Programmable Logic and Applications*, pp. 369–374, Aug. 2006.

[4] K. Turkington, K. Masselos, G. Constantinides, and P. Leong, "FPGA based acceleration of the LINPACK benchmark: A high level code transformation approach," *Proc. Int. Conf. Field Programmable Logic and Applications*, pp. 375–380, Aug. 2006.

[5] X. Wang and M. Leeser, "Efficient FPGA implementation of QR decomposition using a systolic array architecture," in *Proc. 16th Int. Symp. Field Programmable Gate Arrays*, 2008, p. 260.

[6] L. Zhuo and V. K. Prasanna, "High performance linear algebra operations on reconfigurable systems," in *Proc. ACM/IEEE Conf. Supercomputing*. Washington, DC, USA: IEEE Computer Society, 2005, p. 2.

[7] Y. Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev, "64-bit floating-point FPGA matrix multiplication," in *Proc. 13th Int. Symp. Field-programmable gate arrays*, 2005, pp. 86–95.

[8] J. Nocedal and S. J. Wright, *Numerical Optimization*. New York, USA: Springer, 1999.

[9] G. R. Morris, V. K. Prasanna, and R. D. Anderson, "A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer," in *Proc. 14th IEEE Symp. Field-Programmable Custom Computing Machines*, 2006, pp. 3–12.

[10] A. Lopes and G. Constantinides, "A high throughput FPGA-based floating point conjugate gradient implementation," *to appear in Proc. Applied Reconfigurable Computing*, 2008.

[11] C. C. Paige and M. A. Saunders, "Solution of sparse indefinite systems of linear equations," *SIAM*, vol. 12, no. 4, pp. 617–629, Sept 1975.

[12] K. V. Ling, J. M. Maciejowski, and W. B. F., "Multiplexed model predictive control," *Proc. 16th IFAC World Congress*, July 2005.

[13] G. H. Golub and C. F. V. Loan, *Matrix computations (3rd ed.)*. Baltimore, MD, USA: Johns Hopkins University Press, 1996.

[14] B. Fisher, *Polynomial Based Iteration Methods for Symmetric Linear Systems*. Baltimore, MD, USA: Wiley, Teubner, 1996.

[15] J. J. Dongarra, "Performance of various computers using standard linear equations software." [Online]. Available: www.netlib.org/benchmark/performance.pdf, Accessed on 10/03/2008.