A High Throughput FPGA-Based Floating Point Conjugate Gradient Implementation*

Antonio Roldao Lopes and George A. Constantinides

Electrical & Electronic Engineering,
Imperial College London,
London SW7 2BT, England
{aroldao,g.constantinides}@ic.ac.uk

Abstract. As Field Programmable Gate Arrays (FPGAs) have reached capacities beyond millions of equivalent gates, it becomes possible to accelerate floating-point scientific computing applications. One type of calculation that is commonplace in scientific computation is the solution of systems of linear equations. A method that has proven in software to be very efficient and robust for finding such solutions is the Conjugate Gradient algorithm. In this paper we present a parallel hardware Conjugate Gradient implementation. The implementation is particularly suited for accelerating multiple small to medium sized dense systems of linear equations. Through parallelization it is possible to convert the computation time per iteration for an order n matrix from $\Theta(n^2)$ cycles for a software implementation to $\Theta(n)$. I/O requirements are scalable and converge to a constant value with the increase of matrix order. Results on a VirtexII-6000 demonstrate sustained performance of 5 GFLOPS and projected results on a Virtex5-330 indicate sustained performance of 35 GFLOPS. The former result is comparable to high-end CPUs, whereas the latter represents a significant speedup.

1 Introduction

One area where Field Programmable Gate Arrays (FPGAs) are increasingly important is in the acceleration of scientific computations. Some important examples of these applications include genetics, robotics, computer graphics and optimization problems. This paper introduces some typical solutions for solving systems of linear equations, a basic and recurring sub-task in scientific computation, and goes on to detail the Conjugate Gradient (CG) method [1]. A parameterizable hardware implementation of this algorithm is outlined, a comparison with software is made and results reported.

Due to deep pipelining, our proposed implementation is particularly suitable for accelerating computations of multiple small to medium sized dense systems in parallel. An example of such application arises when solving large banded

* The authors would like to acknowledge the support of the EPSRC (Grant EP/C549481/1 and EP/E00024X/1) and the support of Dr. Eric Kerrigan.

R. Woods et al. (Eds.): ARC 2008, LNCS 4943, pp. 75–86, 2008.

linear systems using the parallel algorithm described in [2] or in Multiple-Input-Multiple-Output adaptive equalization [3]. These computations are widespread and include the numerical solution of partial differential equations used in optimal control problems [4].

The main contributions of this paper are thus:

- a parameterizable deeply pipelined hardware design for solving systems of linear equations,
- a detailed analysis of the Conjugate Gradient algorithm and its affinity for FPGA based implementation,
- a design capable of 5 GFLOPS on VirtexII, and projected results predicting that a sustained performance of 35 GFLOPS is possible for a Virtex5-330; a speedup of 5× compared to the *peak* theoretical performance of a Pentium IV running at 3 GHz, and an order of magnitude compared to the measured performance of this processor.

After discussing the relevant background in Section 2, we present an overview of the CG method in Section 3. Section 4 presents the proposed hardware design. Section 5 details resulting resource utilization, achievable throughput, I/O requirements, and comparison to a high performance CPU is made. Section 6 concludes the paper.

2 Background

Most scientific computation involves the solution of systems of linear equations. To address this problem there are some well studied and proven methods. These are divided into two main categories: direct, where the solution is given by evaluating a derived formula, and iterative where the solution is approximated based on previous results until a certain acceptable value is reached. Examples of direct methods include Cholesky factorization and Gaussian Elimination, while iterative methods include Generalized Minimal Residual Methods (GMRES) and the Conjugate Gradient Method that is described in this paper.

For large problems, matrix and vector operations can be computationally intensive and may require significant processing time. Nonetheless they can be accelerated by performing, whenever possible, parallel operations. As a result of advancements in FPGA density, massively parallel floating point computation has become feasible. Although there has been an increasing interest into the use of Field Programmable Gate Arrays to accelerate scientific computation, with recent supercomputers incorporating these devices [5,6], only very recently there has been research focused on developing hardware optimized linear algebra [7]. This has led to the study and comparison of the performance and precision against conventional microprocessors. The results forecast a very promising future for FPGAs by predicting that by the year 2009 they will be an order of magnitude faster in peak performance [8].

Some typical methods for solution of linear equations finding have already been implemented on FPGAs. A Cholesky implementation has demonstrated a performance increase by 50% over software on a APEX EP20K1500E FPGA [9]. A Jacobi solver has been implemented on a Xilinx VirtexII Pro XC2VP50 and demonstrated a speedup of 1.3 to 36.8 relative to uniprocessor implementations, depending on the matrix structure [10]. There are also two papers that discuss an implementation of the Conjugate Gradient method: one uses a Logarithmic Number System (LNS) and achieves up to 0.94 GFLOPS on a VirtexII-6000 [11], the other uses a rational number representation and achieves 0.27 GFLOPS using a VirtexII Pro XC2VP4 [12] and projects that it will be able to sustain 1.5 GFLOPS with Virtex4-55. In contrast, we present a parallelised and deeply pipelined Conjugate Gradient method using the IEEE 754 [13] single precision floating point number representation. We are able to achieve approximately 5 GFLOPS on a readily available VirtexII-6000 and 35 GFLOPS on a high-spec Virtex5-330, for matrices of order 16 and 58 respectively.

Table 1 summarizes FPGA implementations of Conjugate Gradient method in terms of year of publication, number system, device and GFLOPS achieved.

Table 1. Previous FPGA-based Conjugate Gradient implementations

Year	Reference	Number System	Device	GFLOPS
2005	[11]	LNS	VirtexII-6000	0.94
2006	[12]	Rational	Virtex4-25	1.5
2007	this paper	FP single	VirtexII-6000	5
2007	this paper	FP single	Virtex5-330	35

3 Conjugate Gradient Method

The Conjugate Gradient Method is an iterative method for solving systems of linear equations of the form given in (1), where the n by n matrix A is symmetric (*i.e.*, $A^T = A$) and positive definite (*i.e.*, $x^T Ax > 0$ for all non-zero vectors x in $\mathbb{R}^n$) [1]. When matrix A is positive definite, the associated quadratic form given by $J(x)$, defined in (2), is convex. $J'(x)$, the differential of $J(x)$, is given in (3). Notice that setting $J'(x) = 0$ is identical to (1), hence the the solution to the linear system is equivalent to minimizing the quadratic function given in (2). This is the basic intuition of CG and other iterative algorithms.

$$Ax = b$$

$$\begin{bmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} = \begin{bmatrix} b_1 \\ b_2 \\ \vdots \\ b_n \end{bmatrix} \tag{1}$$

$$J(x) = \frac{1}{2}x^T Ax - b^T x \tag{2}$$

$$J'(x) = \begin{bmatrix} \frac{\partial}{\partial x_1}J(x) \\ \frac{\partial}{\partial x_2}J(x) \\ \vdots \\ \frac{\partial}{\partial x_n}J(x) \end{bmatrix} = Ax - b \tag{3}$$

3.1 Algorithm Description

The algorithm, described in Fig. 1, consists of two parts. The first is an initialization that produces a 'residual' or search direction. The second part iterates until the residual error is sufficiently small. The algorithm is intuitive and comprises of the following steps:

1. Determine a search direction, d, of steepest descent in $J(x)$. ($cg1$) and ($cg12$).
2. Perform a line search to determine the best step length, α, in the steepest descent direction. ($cg5$) and ($cg6$).
3. Generate the new solution by adding the vector d times the determined step length α to the current solution x and update the residual r. ($cg7$) and ($cg8$).
4. Iterate until the residual error is negligible. ($cg13$).

4 Implementation

4.1 Overview

The dataflow of this iterative algorithm is depicted in Fig. 2. The most computationally intensive operation is the matrix-by-vector multiplication in ($cg5$). To obtain scalable performance, the proposed design implements this computation by sequentially operating on each matrix row in turn; each constituent vector-by-vector multiplication, however, is fully unrolled and parallelised (see Fig. 3). We also use the same vector-by-vector unit for operations ($cg2$), ($cg6$) and ($cg10$). These operations are represented in the double lined boxes on dataflow diagram of Fig. 2. This vector-by-vector unit is fully pipelined, with a new vector introduced each cycle. As a result, this implementation is able to complete a conjugate gradient iteration every $n + 3$ cycles. This throughput is given by the vector-by-vector computational unit that has to compute for n cycles to perform the block matrix-by-vector operation and another 3 cycles to compute the remaining vector-by-vector operations ($cg2$), ($cg6$) and ($cg10$).

The latency of one CG iteration is given by (4) where the linear growth comes from the row-by-row processing, the logarithmic growth comes from the addition tree in the inner-product computation, and the constants are due to the pipeline depths of the components. The discrepancy between a throughput of one iteration every $n + 3$ cycles and the latency given in (4)is used to our

Input :	Matrix A, Vector b,	
	Error tolerance ε	
Output :	x Such that $\|Ax - b\|_2 \leq \varepsilon \|b\|_2$	
$d \leftarrow b$		$(cg1)$
$r \leftarrow b$		$(cg2)$
$\delta_0 \leftarrow r^T r$		$(cg3)$
$\delta_{new} \leftarrow \delta_0$		$(cg4)$
do		
	$q \leftarrow Ad$	$(cg5)$
	$\alpha \leftarrow \frac{\delta_{new}}{d^T q}$	$(cg6)$
	$x \leftarrow x + \alpha d$	$(cg7)$
	$r \leftarrow r - \alpha q$	$(cg8)$
	$\delta_{old} \leftarrow \delta_{new}$	$(cg9)$
	$\delta_{new} \leftarrow r^T r$	$(cg10)$
	$\beta \leftarrow \frac{\delta_{new}}{\delta old}$	$(cg11)$
	$d \leftarrow r + \beta d$	$(cg12)$
while $\delta_{new} > \varepsilon^2 \delta_0$		$(cg13)$

Fig. 1. Conjugate Gradient Algorithm [14]

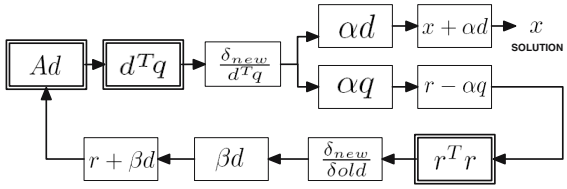

Fig. 2. Circuit data flow diagram. Single boxed operations are implemented using a single floating point unit each. Double boxed operations are implemented on the single matrix/vector by vector module that requires $2n - 1$ FP units.

advantage, by using the slack to operate on multiple different matrix/vector pairs in a pipelined fashion. The total number of linear systems that can be processed simultaneously by the pipeline is therefore given by (5), a $\Theta(1)$ function that converges to 7 for large n as shown in Fig. 4. Note that in order to continuously process problems every $n + 3$ cycles, a constant κ is introduced into (4) so that the number of clocks per iteration is a multiple of $n + 3$. This is implemented through the addition of a FIFO at the output of last operation $(cg12)$. This guarantees the new value of d is output at the start of a new iteration in $(cg5)$ ensuring full pipeline utilization.

One of the major advantages of the proposed row-based scheme is its scalable FPGA I/O requirements, eliminating I/O bottlenecks. The conjugate gradient algorithm completes in n iterations under infinite precision, and $\Omega(n)$ iterations under finite precision [1] [15]. Since one iteration is completed by our design every $\Theta(n)$ cycles and to find the solution for this system under its finite precision we

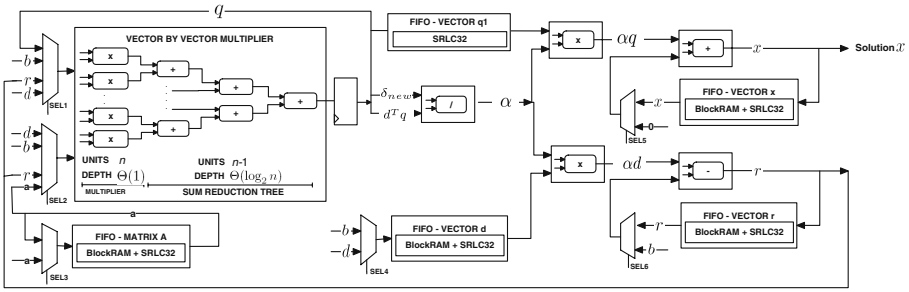

Fig. 3. Partial circuit schematic displaying the vector-by-vector module, two of constant by vector multiplications, a vector by vector summation, vector by vector subtraction and FIFOs. Some of these FIFOs use a combination of Xilinx SRLC32 primitives and BlockRAMs and store various vectors including A matrices in the row-by-row form.

require at least n iterations, the the data transfer bandwidth required is a $\Theta(1)$ function, *i.e.* approaches a constant for large n. Section 5 quantifies this I/O requirement for synthesized designs, and shows it to be well within PCI-express bandwidth limitations.

$$\text{Clocks per Iteration}(n) = 7n + 36\lceil \log_2 n \rceil + 127 + \kappa \tag{4}$$

$$\text{Pipeline Depth}(n) = \frac{7n + 36\lceil \log_2 n \rceil + 127 + \kappa}{n + 3} \tag{5}$$

4.2 Performance

In order to optimize for throughput, floating point modules with the highest latency were selected from the Xilinx Coregen Library. From results gathered, the maximum frequency achievable on the Virtex5-330 is 364MHz limited by the divider. In practice, when included with the other logic, this falls to 287MHz on the Virtex5-330 (and 126MHz for the VirtexII-6000).

Since this implementation does not have every floating point computational module in operation for the entire iteration of the CG method, as described in Section 4.3, two performance formulas were deduced. One describes the peak performance (6) when all the modules are in operation simultaneously (*e.g.* at the start of a $n + 3$ period when the pipeline is full) and the other counts the number of operations per iteration divided by clocks per iteration (7). This second formula accounts for the idle time of floating point units involving vector operations that only function for n cycles, every $n + 3$ cycles.

$$\text{FLOPS Peak}(n) = (2n + 7) \times \text{MaxFreq} \tag{6}$$

$$\text{FLOPS Sustained}(n) = \frac{2n(n + 5)}{n + 3} \times \text{MaxFreq} \tag{7}$$

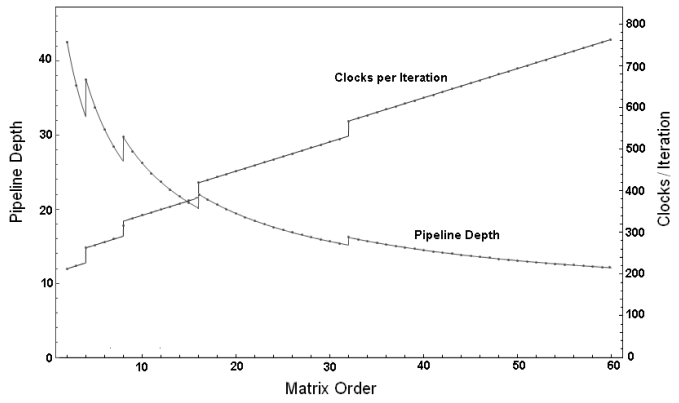

Fig. 4. Pipeline depth and required cycles per iteration with matrix order

4.3 Floating Point Unit Operations

Due to the iterative nature of the algorithm it is possible to reutilize all allocated floating point units. A single module is instantiated to perform all vector-by-vector multiplications, n times in ($cg5$), and once in ($cg2$), ($cg6$) and ($cg10$). This module implements the double boxed computations in Fig. 2, and operates continually processing and outputting new values every $n+3$ cycles. The remaining vector operations, in order to output a solution every $n+3$ cycles, require a single floating point unit each. These units operate for n (+ latency) clock cycles before a vector solution is output. After this solution is output, each unit is idle for 3 cycles to compensate for the matrix and vector by vector operation module delay. In brief, every constant by vector multiplication and vector by vector addition/subtraction operates for n cycles every $n+3$ cycles. The divisions in ($cg6$) and ($cg12$) only operate for 1 (+ latency) clock cycle every $n+3$ cycles. The total number of floating point units is detailed in Table 2.

Table 2. Floating Point units used in this implementation

Operation	FP units
Matrix/Vector by Vector Multiplier	$2n-1$
Constant by Vector Multiplier	3
Vector by Vector Summation	2
Vector by Vector Subtraction	1
Floating Point Divider	2
Total (FP_{units})	$2n+7$

5 Results

5.1 Resource Utilization

Theoretical floating point resource utilization grows as $\Theta(n)$. However for this method to be efficient the coefficients of each problem to be solved need to be stored or generated within the FPGA. This requires a storage that grows with $\Theta(n^2)$ ($\Theta(n^2)$ for one problem, with $\Theta(1)$ problems in the pipeline). To store these values a mixture of embedded BlockRAMs [16] and SRLC32 primitives are used. This mixture depends on the length of the FIFO. When this length is equal to or below 64, they are implemented solely using SRLC32 primitives. When above 64, they are implemented by combining BlockRAMs and SRLC32 primitives for efficiency. This is due to the fact that Xilinx Coregen BlockRAM FIFOs are only available in sizes of 2^n with $n > 3$; thus SRLC32 primitives are used to take up any slack. Fig. 5 depicts resource usage as a function of the matrix order. Growth of each resource is approximated linearly as predicted, with the exception of BlockRAMs that are also used for matrix storage. The usage of these BlockRAMs is asymptotically quadractic, however for the depths in the range of our implementation, this growth is at most $O(n \log n)$, since the BlockRAMs are deep enough to implement the required FIFOs and a further $O(\log n)$ units can be used to help fill the slack mentioned earlier.

For the Virtex5-330, resources are saturated for matrices orders above 58 having depleted all BlockRAMs. Best fit resource usage for REGisters, LUTs, BlockRAMs and DSP48Es usage as a function of matrix order is described in (9), (10), (11) and (8) respectively. BlockRAMs usage varies significantly, from the best fit, because they are used in conjunction with SRLC32s, as explained previously.

$$\text{DSP48Es}(n) = 2n + 2 \tag{8}$$

$$\text{REG Slices}(n) = 3007n + 6446 \tag{9}$$

$$\text{LUT Slices}(n) = 2361n + 3426 \tag{10}$$

$$\text{BlockRAMs}(n) = 25n \log_2 n - 2n - 3 \log_2 n - 39 \tag{11}$$

5.2 Software Comparison and Discussion

Acceleration relative to software is provided by pipelining and parallelization of vector operations. In this implementation considerable speedup is due to the block module that performs a fully parallelized vector by vector multiplication. Each of these operations requires $2n - 1$ sequential operations in software while in hardware they can be reduced to $Lm + Ls\lceil \log_2 n \rceil$ cycles for a single problem, where Lm is the latency of the multiplication core, Ls the latency of the addition

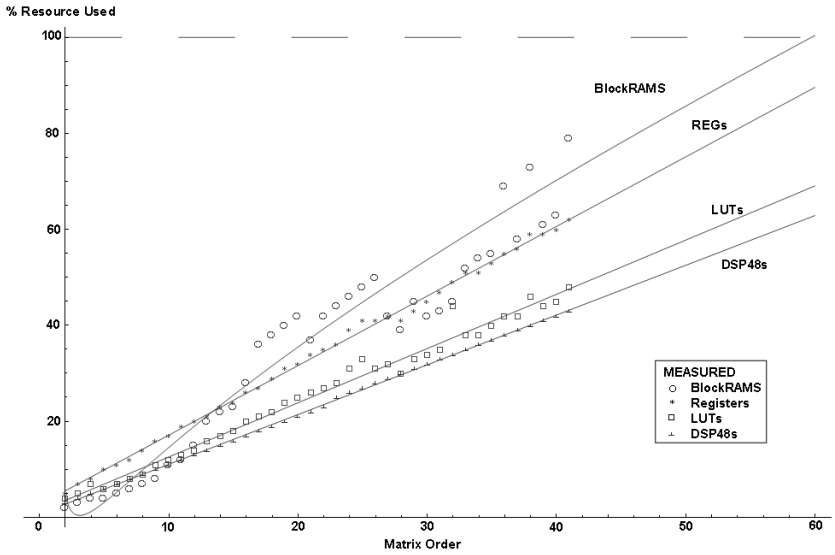

Fig. 5. BlockRAMs, REGisters and LUTs resource utilization with matrix order for the Virtex5-330. Lines represent the best fit based on the synthesis reports of Look-Up-Tables, REGisters, BlockRAMs and DSP48Es usage.

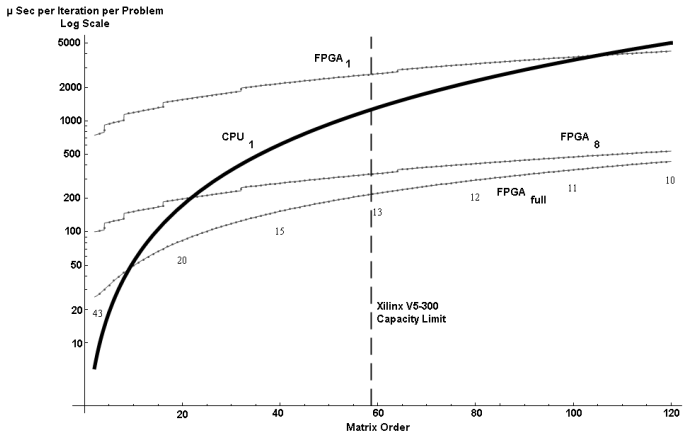

Fig. 6. Iteration time required for solving a number of CG problems as a function of matrix order on a CPU and FPGA. The bold line represents a high end CPU operating at 6 GFLOPS [17]. The remaining lines show the FPGA Virtex5-330 implementation with a single problem in the pipeline, with 8 problems and a fully loaded pipeline. This last line also depicts the number of problems in the pipeline for that matrix order in accordance with (5).

core and n, the matrix order. In the case where several vectors need to be multiplied, they can be pipelined and a result provided for every clock cycle at the initial cost of filling the pipeline.

The overall speedup given by the combination of parallelization and pipelining is illustrated in Fig. 6, which compares the processing time, for each CG iteration, on an FPGA and on a Pentium IV running at 3.0 GHz. Three lines are shown for the FPGA implementation: one representing the pipeline containing only a single problem, another intermediate line showing the pipeline with 8 problems, and a third line representing a full pipeline. Below this last line, the number of problems pipelined and being concurrently solved, given by (5), is shown for every 20 matrix orders. The dark bold line represents the CPU at its best case scenario of peak theoretical performance at 6 GFLOPs [17]. Comparing the FPGA with a full pipeline and a CPU, it is possible to observe that the FPGA is faster than the CPU for orders greater than 10. For a single problem in the FPGA pipeline, the CPU becomes slower than the FPGA for matrix orders above 105. With the intermediate FPGA line showing the time required to process 8 pipelined problems it is possible to observe its convergence to the FPGA full line as demonstrated in (5). Thus with only eight parallel problems, FPGA superiority is clearly established, even for low matrix orders.

5.3 Input/Ouput Considerations

As input, this method requires a matrix A and a vector b to be introduced. As a output, it requires the solution vector x, which, under finite precision, is generated after at least n iterations [15]. This translates to the need to transfer $32(n^2 + 2n)$ bits per problem for a total number of problems given by (5). This transfer can occur over a period given by at least n times the clocks per iteration (4) because this is the time it takes to generate a solution and start a new problem. Combining these data we can deduce the a minimum bit rate as given in (12). For the Virtex5-330 limit matrix order of 58, running at 287MHz this, translates to a data rate requirement of 1.1GB/s, a value well within the operation range of PCI-Express [18].

$$\text{I/O Bits per Clock Cycle} = 32 - \frac{32}{n+3} \tag{12}$$

6 Conclusions

This paper describes a Conjugate Gradient implementation. It analyzes its resource utilization growth with matrix order, peak performance achievable, compares this performance with a high end processor and demonstrates that this method exhibits high performance with scalable I/O requirements.

It is demonstrated that multiple dense problems of matrix order 16 can be solved in parallel with a sustained floating point performance of 5 GFLOPS, for the VirtexII-6000 and multiple dense matrices of order 58, with a sustained floating point performance of 35 GFLOPS, for the Virtex5-330. Multiple parallel

solutions of these orders are required, for example, in Multiple-Input-Multiple-Output communication systems using adaptive quantization [3] and in solving large banded matrices using the algorithm described in [2]. These banded systems arise in a number of problems including optimal control systems [4].

For the lower density VirtexII-6000 FPGA, the GFLOPS are between peak theoretical performance and the measured performance of a Intel Pentium IV 3.0 GHz CPU, based on a Linpack benchmark [17]. Taking advantage of hardware parallelization, the required latency for a single iteration is reduced from $\Theta(n^2)$ to $\Theta(n)$, at the cost of increasing hardware computational utilization from $\Theta(1)$ to $\Theta(n)$.Since generating each solution requires at least n iterations under finite precision [1] and each iteration requires $n + 3$ clock cycles, this design exhibits scalable I/O transfer rates that converge to a constant number, as matrix order n increases. Hence, its inherent properties make it exceptionally suited for FPGA implementation.

Future work will be focused on the direct solution of structured systems originating in [4] and matrix sparsity will also be exploited to accelerate the solutions of especial cases.

References

1. Hestenes, M., Stiefel, E.: Methods of conjugate gradients for solving linear systems. Journal of Research of the National Bureau of Standards 49(6), 409–436 (1952)
2. Wright, S.: Parallel Algorithms for Banded Linear Systems. SIAM Journal on Scientific and Statistical Computing 12(4), 824–842 (1991)
3. Biglieri, E., Calderbank, R., Constantinides, A., Goldsmith, A., Paulraj, A.: MIMO Wireless Communications. Cambridge University Press, Cambridge (2007)
4. Wright, S.: Interior Point Methods for Optimal Control of Discrete Time Systems. Journal of Optimization Theory and Applications 77(1), 161–187 (1993)
5. Cray, XD1 Datasheet (2005) (Accessed on 2/03/2007) `http://www.cray.com/downloads/Cray-_XD1_Datasheet.pdf`
6. SGI, RASC RC100 Blade (2006) (Accessed on 2/03/2007) `http://www.sgi.com/-pdfs/3920.pdf`
7. Zhuo, L., Prasanna, V.K.: High Performance Linear Algebra Operations on Reconfigurable Systems. In: Proc. of SuperComputing, pp. 12–18 (2005)
8. Underwood, K.: FPGAs vs. CPUs: Trends in Peak Floating-Point Performance. In: Proc. ACM. Int. Symp. on Field-Programmable Gate Arrays, pp. 171–180 (2004)
9. Haridas, S., Ziavras, S.: FPGA Implementation of a Cholesky Algorithm for a Shared-Memory Multiprocessor Architecture. Journal of Parallel Algorithms and Applications 19(6), 411–226 (2004)
10. Morris, G., Prasanna, V.: An FPGA-Based Floating-Point Jacobi Iterative Solver. In: Proc. of the 8th International Symposium on Parallel Architectures, Algorithms and Networks, pp. 420–427 (2005)
11. Maslennikow, V.L.O., Sergyienko, A.: FPGA Implementation of the Conjugate Gradient Method. In: Proc. Parallel Processing and Applied Mathematics, pp. 526–533 (2005)
12. Callanan, A.N.O., Gregg, D., Peardon, M.: High Performance Scientific Computing Using FPGAs with IEEE Floating Point and Logarithmic Arithmetic For Lattice QCD. In: Proc. of Field Programmable Logic and Applications, pp. 29–35 (2006)

13. IEEE, 754 Standard for Binary Floating-Point Arithmetic (1985) (Accessed on 18/03/2007), http://grouper.ieee.org/groups/754/
14. Shewchuk, J.: An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, Edition $1\frac{1}{4}$ (2003) (Accessed on 28/02/2007), http://www.cs.cmu.edu/~jrs/+jrspapers.html#cg
15. Meurant, G.: The Lanczos and Conjugate Gradient Algorithms from theory to Finite Precision Computation, SIAM, 323–324 (2006)
16. Xilinx, DS100 (v3.0) Virtex5 Family Overview - LX , LXT, and SXT Platforms (2007) (Accessed on 1/03/2007), http://direct.xilinx.com/bvdocs/publications/ds100.pdf
17. Dongarra, J.: Performance of Various Computers Using Standard Linear Equations Software (2007) (Accessed on 15/03/2007), http://www.netlib.org/benchmark/performance.ps
18. Bhatt, A.: PCI-Express - Creating a Third Generation I/O Interconnect (2007) (Accessed on 19/06/2007), http://www.intel.com/technology/pciexpress/devnet/docs/WhatisPCIExpress.pdf