A High Throughput FPGA-based Floating Point Conjugate Gradient Implementation for Dense Matrices [1]

ANTONIO ROLDAO and
GEORGE A. CONSTANTINIDES
{aroldao,g.constantinides}@ic.ac.uk
Imperial College London

Recent developments in the capacity of modern Field Programmable Gate Arrays (FPGAs) have significantly expanded their applications. One such field is the acceleration of scientific computation and 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 (CG) algorithm. In this paper we present a widely-parallel and deeply-pipelined hardware CG implementation, targeted at modern FPGA architectures. This implementation is particularly suited for accelerating multiple small-to-medium sized dense systems of linear equations and can be used as a stand alone solver or as building block to solve higher order systems. In this paper it is shown that through parallelization it is possible to convert the computation time per iteration for an order n matrix from $\Theta(n^2)$ clock cycles on a micro-processor to $\Theta(n)$ on a FPGA. Through deep-pipelining it is also possible to solve several problems in parallel and maximize both performance and efficiency. I/O requirements are shown to be scalable and convergent to a constant value with the increase of matrix order. Post place-and-route results on a readily available VirtexII-6000 demonstrate sustained performance of 5 GFLOPS, and results on a Virtex5-330 indicate sustained performance of 35 GFLOPS. A comparison with an optimized software implementation running on a high-end CPU, demonstrate that this FPGA implementation represents a significant speed-up of at least an order of magnitude.

1 Introduction

With the increase in density and embedding of optimized multiplier blocks, modern Field Programmable Gate Arrays (FPGAs) have become increasingly suited

[1] The authors gratefully acknowledge the support of the UK EPSRC (Grant EP/C549481/1 and EP/E00024X/1) and discussions with Dr. Eric Kerrigan.
Author's addresses: Electrical & Electronic Engineering, Imperial College, Exhibition Road, London SW7 2AZ, UK.

for accelerating scientific computations. Some important applications of these computations include genetics [1], robotics [2], medical imaging [3] and optimization problems [4].

This paper introduces some typical algorithms 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 [5]. A parameterizable hardware implementation of this algorithm is outlined, a comparison with software is made, and results are reported. Due to deep-pipelining, our implementation is particularly suited for accelerating computations of multiple small-to-medium sized dense systems in parallel. An example of such a requirement arises when solving large banded linear systems using the parallel algorithm described in [6] or in Multiple-Input-Multiple-Output adaptive equalization [7]. This implementation is also suited for generating approximate solutions to multiple systems of linear equations within a certain acceptable error or time constraint. A widely used example of such an application is given by the inner loop of the Truncated Newton Method [8]. These computations are widespread and include the numerical solution of partial differential equations used in optimal control problems [9], structural analysis, circuit analysis, and many other scientific problems.

The main contributions of this paper are thus:

- an FPGA-based parameterizable design for solving systems of linear equations efficiently by exploring wide-parallelism and deep-pipelining,
- a detailed analysis of the Conjugate Gradient algorithm and its affinity for FPGA based implementation,
- a quantification of performance, resource utilization, depth of the pipeline in terms of problems, I/O requirements,
- a design capable of 5 GFLOPS on a VirtexII-6000, and results demonstrating that a sustained performance of 35 GFLOPS is possible for a Virtex5-330 [10],
- a comparison with an Automatically Tuned Linear Algebra Software (ATLAS) program running on a high-end CPU.

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

2 Background

Most scientific computations involve 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. Notable examples of direct methods include the Gaussian Elimination, which can be applied to any type of matrix, and the Cholesky decomposition, which can only be applied to a symmetric and positive-definite matrix. The analogous iterative methods are the Generalized Minimal Residual methods (GMRES), for any type of matrix, and the Conjugate Gradient method, for symmetric and positive-definite matrices.

2.1 Architectures for Scientific Computation

Most methods of solving systems of linear equations involve matrix and vector operations which can be computationally intensive and may require significant processing time. Nonetheless these operations can be accelerated by performing, whenever possible, parallel operations. To explore this acceleration, a number of different hardware architectures have been investigated. These architectures include, Connection Machines [11], Cell Processors [12], Graphical Processing Units (GPUs) [13] and FPGAs [14]. A widely implemented comparative benchmark for floating-point computations is the General Matrix Multiply (GEMM) subroutine, part of the Basic Linear Algebra Subprograms (BLAS) library [15]. Table 1 compares the performance of this matrix-by-matrix multiplication operation on different hardware architectures.

Table 1. Floating-point matrix-by-matrix multiplication benchmark on different hardware architectures.

Year	Architectue	Reference	Device	Precision	GFLOPS
2004	GPU	[16]	Radeon X800XT	single	64
2005	FPGA	[17]	XC2VP125	double	16
2006	Cell	[18]	CBEA	double	15
2006	Clearspeed	[19]	CSX600	double	25
2008	CPU	[20]	Pentium4 (3.6GHz)	double	7
2008	GPU	[21]	Quadro FX 5600	single	120
2008	FPGA	[22]	3SE260	double	102

With the recent advancements in FPGAs density and architectures, massively-parallel and deeply-pipelined floating point computations have become feasible within an FPGA. Although there has been an increasing interest into the use of Field Programmable Gate Arrays to accelerate scientific computations, with the latest supercomputers incorporating these devices [23, 24], only very recently there has been research focused on developing FPGA optimized linear algebra [25]. This has led to the study and comparison of the performance and precision against conventional high-end CPUs and other architectures (Table 1).

A forecast from 2004 projected a very promising future, predicting that in the year 2009 these devices will be an order of magnitude faster in peak performance compared to traditional high-end CPUs [26]. Current work reports significant FPGA performance in line with Underwood's prediction [27][10].

2.2 Previous FPGA Implementations

Some typical methods for solving systems of linear equations have already been implemented on FPGAs.

A Cholesky implementation demonstrated a performance increase by 1.99 times over software, for matrices of order 48, on a APEX EP20K1500E FPGA [28]. This implementation was based around a system that uses an asymmetric, shared-memory MIMD architecture and was built using two embedded Nios processors.

A Jacobi solver was implemented on a Xilinx VirtexII Pro XC2VP50 where performance estimates, which include both data transfer and execution time, show that this circuit provides a 1.3 times speedup with a large dense matrix, for a single iteration, when compared to a uniprocessor implementation. For a single iteration, a large sparse matrix Jacobi circuit could achieve an estimated speedup of 1.1 to 19.5, when compared to highly optimized uniprocessor implementations. Multiple iteration speedups ranged from 2.8 to 36.8. Sparse matrices having an irregular structure had the biggest speedups [29].

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 1.1 GFLOPS on a VirtexII-6000 [30]. The other uses a rational number representation and achieves 0.27 GFLOPS using a VirtexII Pro XC2VP4 [31] and projects that it will be able to sustain 15 GFLOPS on a Virtex4-55. In contrast, we present a widely-parallelised and deeply-pipelined Conjugate Gradient method using the IEEE 754 [32] single precision floating point number representation.

Due to the domination of the algorithm by inner-products, known to map well to FPGAs [14][25], CG is well suited, even for small dense systems. The FPGA allows the construction of a data-path specialised not only to the CG algorithm, but to the order of the matrix. Thus, for embedded applications, where the matrix order does not typically change on the fly [33], a very efficient data-path can be formed, minimizing control overheads. With this implementation 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 2 summarizes FPGA implementations of Conjugate Gradient method in terms of year of publication, number system, device and GFLOPS achieved.

Table 2. FPGA-based Conjugate Gradient implementations.

Year	Reference	Number System	Device	GFLOPS
2005	[31]	LNS	VirtexII-6000	1.1
2006	[30]	Rational	Virtex4-25	1.5
2008	this paper	FP single	VirtexII-6000	5
2008	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 $\mathbb{R}^n$) [5]. 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 Conjugate Gradient algorithm is a Krylov subspace method that [34] solves $Ax = b$ by repeatedly performing matrix-vector multiplications involving A. Starting with an initial guess, x_0, this algorithm consecutively produces an approximated solution x_k by minimizing the A-norm of the residual, given by

$||Ax_k - b||_A$ where k is the iteration number, and $||u||_A \triangleq u^T Au$.

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 descent in $J(x)$. $(cg1)$ and $(cg12)$.
2. Perform a line search to determine the best step length, α, in the 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. $(cg13)$.

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 [35].

3.2 Algorithm Example and Context

To exemplify this method, a system defined in (4) is solved below. For this example the solution is $x = [2\ 1]^T$. The corresponding quadratic function, given by (2), is shaped like paraboloid bowl and the solution is given at its lowest point.

$$\begin{bmatrix} 2 & 1 \\ 1 & 5 \end{bmatrix} x = \begin{bmatrix} 5 \\ 7 \end{bmatrix} \tag{4}$$

In Fig. 2 the starting point, which in this implementation is set as the origin by default, and subsequent iterations are illustrated. Each iteration arrow represents both the line search direction given by the descent direction, d, and the step length, α. Intermediate and final values of relevant variables are shown in Table 3 using single precision arithmetic [32]. In this example, the initial residual norm is $||Ax_0 - b||_2 = (5^2 + 7^2)^{\frac{1}{2}} \approx 8.6$. After one iteration this has decreased to $||Ax_1 - b||_2 \approx (1.553^2 + 1.110^2)^{\frac{1}{2}} \approx 1.9$, and after two iterations the residual has been reduced to a negligible level.

Table 3. Example iteration values.

Iteration	0	1	2
r	$\begin{pmatrix} 5 \\ 7 \end{pmatrix}$	$\begin{pmatrix} 1.553 \\ -1.110 \end{pmatrix}$	$\begin{pmatrix} 1.192 \times 10^{-7} \\ 8.345 \times 10^{-7} \end{pmatrix}$
d	$\begin{pmatrix} 5 \\ 7 \end{pmatrix}$	$\begin{pmatrix} 1.800 \\ -0.765 \end{pmatrix}$	$\begin{pmatrix} 1.192 \times 10^{-7} \\ 8.345 \times 10^{-7} \end{pmatrix}$
δ_{new}	74	3.644	7.105×10^{-13}
α	-	0.202	0.548
β	-	0.049	1.950×10^{-13}
x	$\begin{pmatrix} 0 \\ 0 \end{pmatrix}$	$\begin{pmatrix} 1.014 \\ 1.419 \end{pmatrix}$	$\begin{pmatrix} 2.000 \\ 1.000 \end{pmatrix}$

Example applications of Conjugate Gradient method include solution finding for problems that arise in optimal control systems. One such example that requires the repeated solution of multiple matrices of order 50 is given by the Citation Aircraft Model [33]. In the context of MIMO systems, the order of these matrices depends linearly on the number of antennas and the number of spatially and temporally independent sources, and is usually below 20 [7].

4 Implementation

4.1 Overview

The dataflow of the algorithm is depicted in Fig. 3. The most computationally intensive operation is the matrix-by-vector multiplication in ($cg5$). To obtain scalable performance, the 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. 4). We also use the same vector-by-vector unit for operations ($cg3$), ($cg6$) and ($cg10$). These operations are represented in the double lined boxes in Fig. 3. This vector-by-vector unit is fully pipelined, with a new vector being introduced each clock cycle. As a result, this implementation is able to complete a conjugate gradient iteration

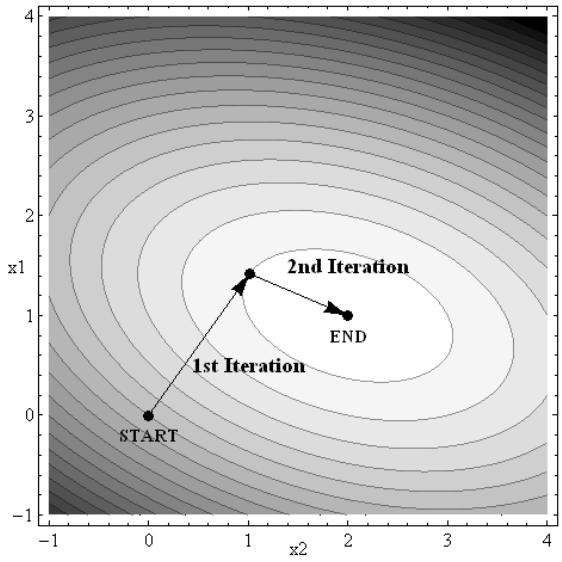

Fig. 2. Contours of constant value of $J(x)$, start and end points of x and iteration steps.

every $n + 3$ cycles. This throughput is given by the vector-by-vector computational unit, which has to compute for n cycles to perform the 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 (5) 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 (5) is used to our advantage, by using the slack to operate on multiple different matrix/vector pairs in a round-robin pipelined fashion. The total number of linear systems that can be processed simultaneously by the pipeline is therefore given by (6), a $\Theta(1)$ function that converges to 8 for large n as shown in Fig. 5. Note that in order to continuously process problems every $n + 3$ cycles, a constant κ is introduced into (5) 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 the final operation $(cg12)$. This guarantees the new value of d is output at the start of a new iteration in $(cg5)$ ensuring that an integer number of problems can be stored in the pipeline.

One of the major advantages of the employed row-based scheme is its scalable

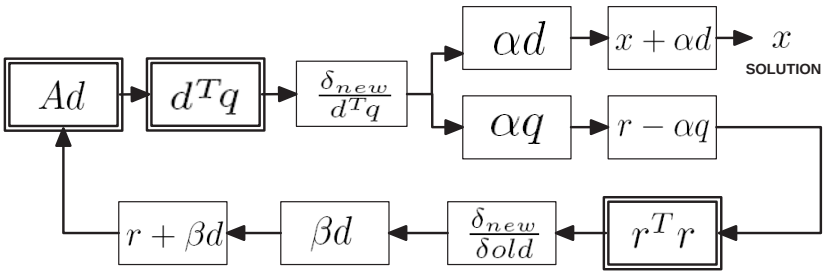

Fig. 3. Circuit data flow diagram. Single boxed operations are implemented using a single floating point unit each. Double boxed operations are implemented on the single matrix/vector-by-vector module that requires $2n - 1$ FP computational units.

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 [5] [36]. 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 require at least n iterations, 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.

$$\text{Clocks per Iteration}(n) = 7n + 36\lceil \log_2 n \rceil + 127 + \kappa \tag{5}$$

$$\text{Pipeline Depth}(n) = \frac{7n + 36\lceil \log_2 n \rceil + 127 + \kappa}{n + 3} \tag{6}$$

4.2 Performance

With Xilinx Core Generator Floating Point v3 units it is possible to tradeoff latency with maximum clock frequency [37]. For the Virtex5-330 [38], individual floating point cores were synthesized as described in Table 4, using Xilinx ISE version 9.1i. In order to optimize for throughput, modules with the highest latency were selected. From Table 4, the maximum frequency achievable is 364MHz limited by the SUM/SUB module. In practice, when included with the other logic, this falls to 287MHz on the Virtex5-330 (and 126MHz on the VirtexII-6000).

Since this implementation does not have every floating point computational module in operation for the entire iteration of the CG method, two performance formulas were deduced. One describes the peak performance (7) 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 (8). This second formula corresponds

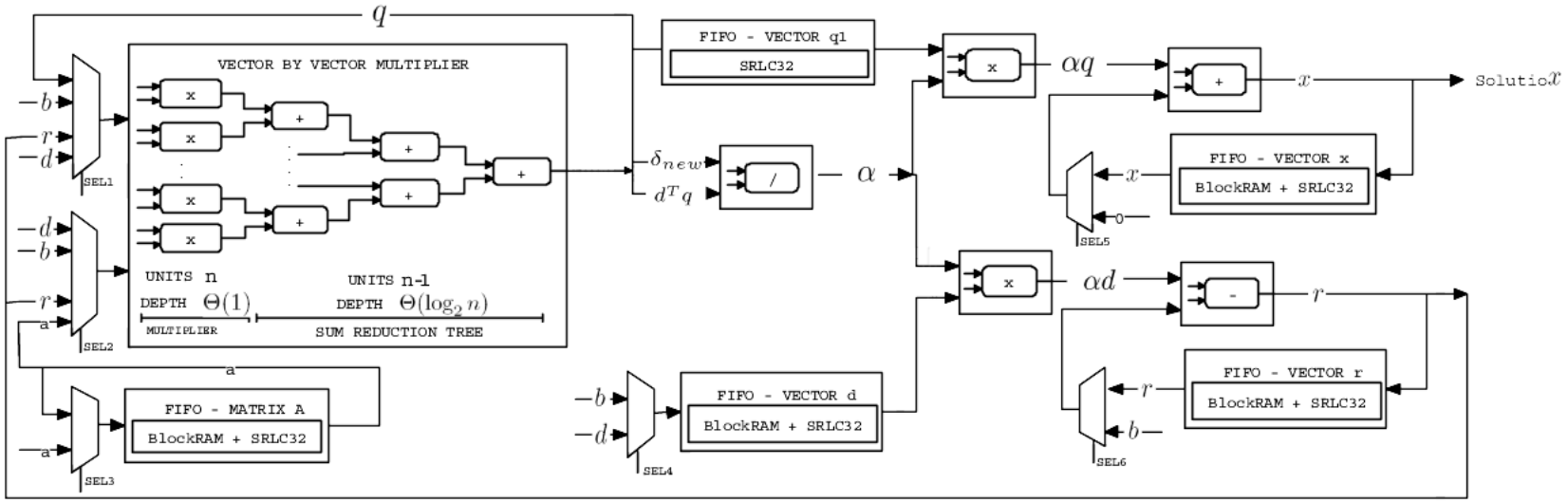

Fig. 4. Partial circuit schematic displaying the vector-by-vector operation module, two constant-by-vector multiplication modules, a vector-by-vector summation module, a vector-by-vector subtraction module and storage FIFOs. Some of these FIFOs use a combination of Xilinx SRLC32 primitives and BlockRAMs and store various vectors including A matrices in a row-by-row form.

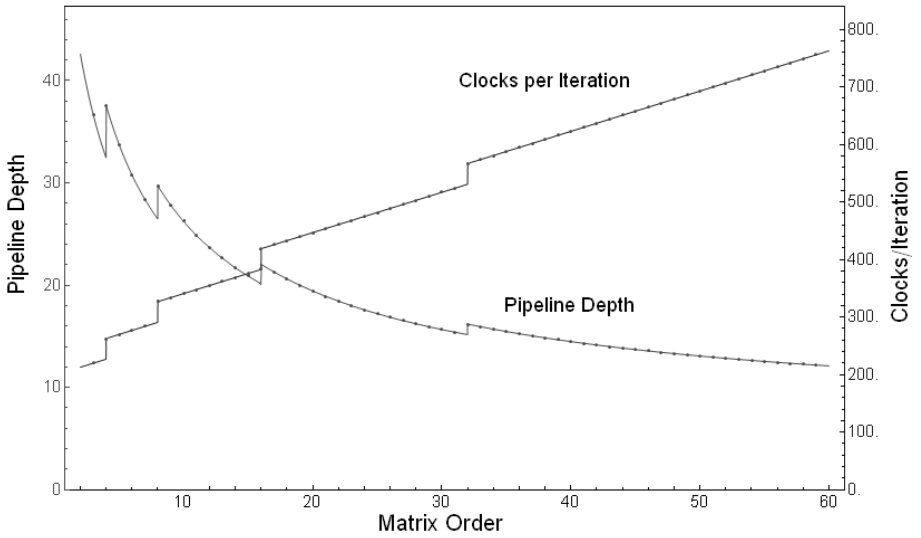

Fig. 5. The Pipeline Depth curve represents the number of problems that can be present simultaneously on the FPGA. The Clocks per Iteration curve represents the number of clock cycles required by each problem to complete an iteration. Both these lines are a function of matrix order.

to sustained performance, and accounts for the idle time of floating point units involving vector operations that only function for n cycles out of every $n + 3$ cycles.

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

$$\text{FLOPS Sustained}(n) = \frac{2n(n + 5)}{n + 3} \times \text{MaxFreq} \tag{8}$$

Fig. 6 plots the peak and sustained GFLOPS performances as a function of matrix order n and pipeline depth. The dark bold line represents the peak GFLOPS and it takes into account only the short period when all the Floating Point units are in simultaneous operation. The light lines represent the sustained performance which is given by the number of operations performed per iteration divided by time required by each iteration. For the Sustained^{full} line, the number of problems in the pipeline is given by (6).

Table 4. Latency, resource and max frequency for Xilinx Core Generator Floating Point v3.0 units using Xilinx ISE 9.1i on the Virtex5-330.

	Latency	LUT Slices	REG Slices	Max Freq (MHz)
SUM/SUB	0	416	0	53
	2	416	63	152
	5	432	240	242
	8	407	418	285
	12	447	573	364
DIV	0	755	0	16
	2	731	100	18
	5	763	224	66
	8	766	368	113
	28	766	1383	390
MULT	0	689	1	13
	2	818	143	157
	5	627	519	224
	8	689	627	366

5 Results

5.1 Resource utilization

Reported resources utilization was generated using Xilinx ISE 9.1i tool-chain. These resources are consumed by the instantiation of floating point computational units, FIFO storage structures and control logic. This CG implementation employs a total number of floating point computational units as detailed in Table 5.

Table 5. 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})	$2n+7$

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

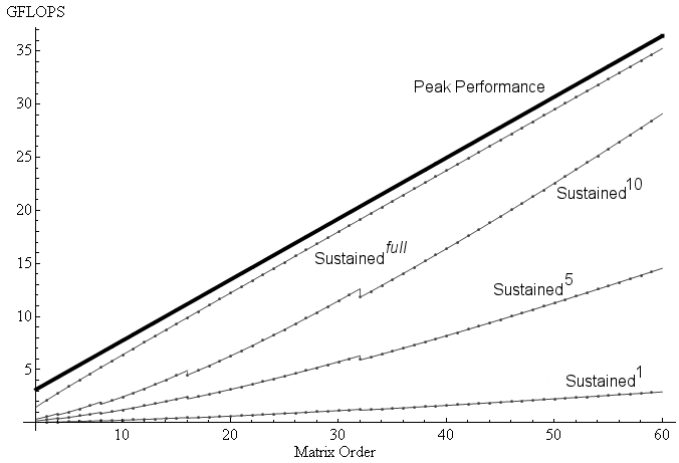

Fig. 6. Peak and sustained performance as a function of matrix order n and the number of pipelined problems.

these values a mixture of embedded BlockRAMs and SRLC32 primitives are used [38]. This mixture depends on the length of the FIFO. When this length is 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. 7 depicts post place-and-route resource utilization as a function of the matrix order, using Xilinx ISE version 9.1i. 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 quadratic, however for the lengths in the range of our implementation, this growth is at most $n \log n$. This is due to the need of assembling n FIFOs for the storage of the A matrix in a row-by-row configuration. Each of these FIFOs stores matrix elements of the same column. Each FIFO require $\log n$ BlockRAMs, since multiple BlockRAMs may be needed to fulfil a desired length, due to discrete lengths available as powers of 2.

For the Virtex5-330, resources are saturated for matrices orders above 58 having depleted all BlockRAMs. Best fit resource usage function for DSP48Es, LUTs, REGisters, and BlockRAMs usage as a function of matrix order are described in (9), (10), (11) and (12) respectively. BlockRAMs usage varies significantly from the best fit, because they are used in conjunction with SRLC32s, as explained previously.

$$\text{DSP48Es}(n) = 2n + 2 \tag{9}$$

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

$$\text{REG Slices}(n) = 3007n + 6446 \tag{11}$$

$$\text{BlockRAMs}(n) = 12.2n \log_2 n - 21 \tag{12}$$

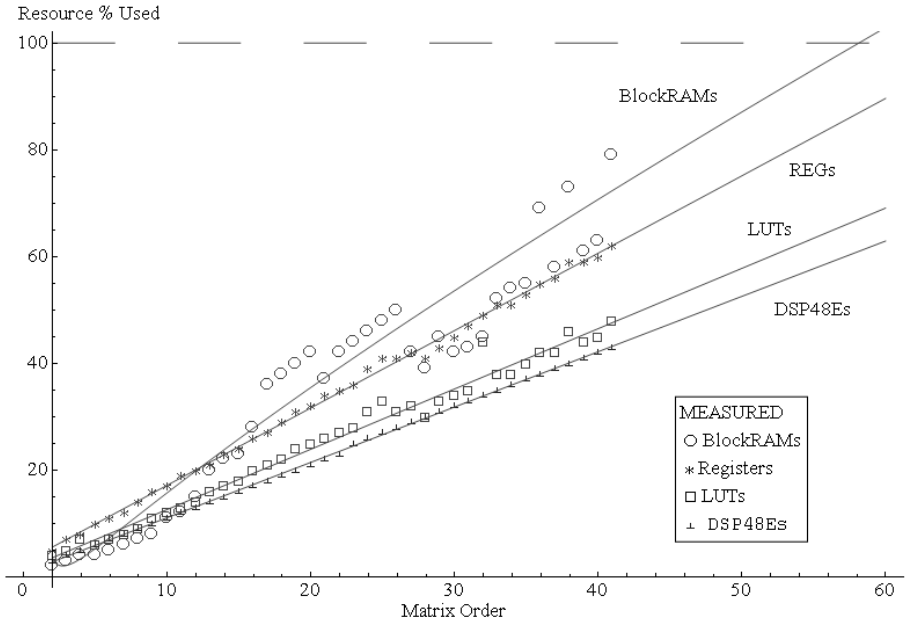

Fig. 7. BlockRAMs, REGisters and LUTs resource utilization with matrix order for the Virtex5-330. Light lines represent the best fit based on the post placement-and-route reports of Look-Up-Tables, REGisters, BlockRAMs and DSP48Es usage.

5.2 Software comparison and discussion

To effectively compare this work with software on a high-end CPU, we have coded two equivalent Conjugate Gradient algorithms in ANSI-C. The first program, CPU_{direct}, is a direct implementation, while the second, CPU_{atlas}, was optimized using Basic Linear Algebra Subprograms (BLAS) functions from the leading edge Automatically Tuned Linear Algebra Software (ATLAS) library [39]. Both these programs were compiled with GNU gcc compiler version 4.2.3. The machine and architecture targeted for these software implementations is a Sun Ultra 20 M2, which comprises of a AMD Opteron 1220 CPU at 2.8 GHz with 4GB (2x2GB)

of RAM, running Gentoo amd64 Linux, which was considered the world's fastest single socket x86 system on the floating point suite [40].

We benchmark the time required per iteration using the POSIX standard function, *clock_gettime*. This function is called just before the start of the iterative code and immediately after it has run for n iterations, and takes into consideration the sampling delay.

Fig. 8 illustrates the performance of the CPU using both the direct and the ATLAS CG implementation. In this figure it is possible to observe that performance increases with matrix order n for both implementations. For the CPU_{direct} program, its performance reaches a plateau around 0.2 GFLOPS for matrix orders above 50. The ATLAS optimized program, CPU_{altas}, peaks at 2.4 GFLOPS for matrix orders around 450, and decreases its performance to stabilize just under 2 GFLOPS, for matrix orders above 600. The direct implementation is faster than the ATLAS optimized code for low matrix orders due to the elimination of function-call and ATLAS data structure overheads. As the matrix order increases, these overheads reduce as a proportion of execution time, and at the same time the degree of instruction-level parallelism available for extraction by the superscalar processor increases, resulting in an improvement in floating-point performance. The direct implementation levels off at only 0.2 GFLOPS mainly because the memory access pattern has not been optimized for the cache, unlike in the ATLAS implementation. The dip in performance of ATLAS for large matrix orders corresponds to an increase in L1 cache misses for these data structures.

Table 6 compares the performance of the direct and optimized software implementations with the FPGA using a Virtex5-330. The results demonstrate that performance is dependent on matrix order n but that speedups of at least an order of magnitude have been achieved.

Acceleration relative to software is provided by pipelining and parallelization of matrix/vector-by-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 core and n is the matrix order. In the case where several vectors need to be multiplied, they can be pipelined and a result provided 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. 9, which compares the processing time, for each CG iteration, on the FPGA and the CPU. 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

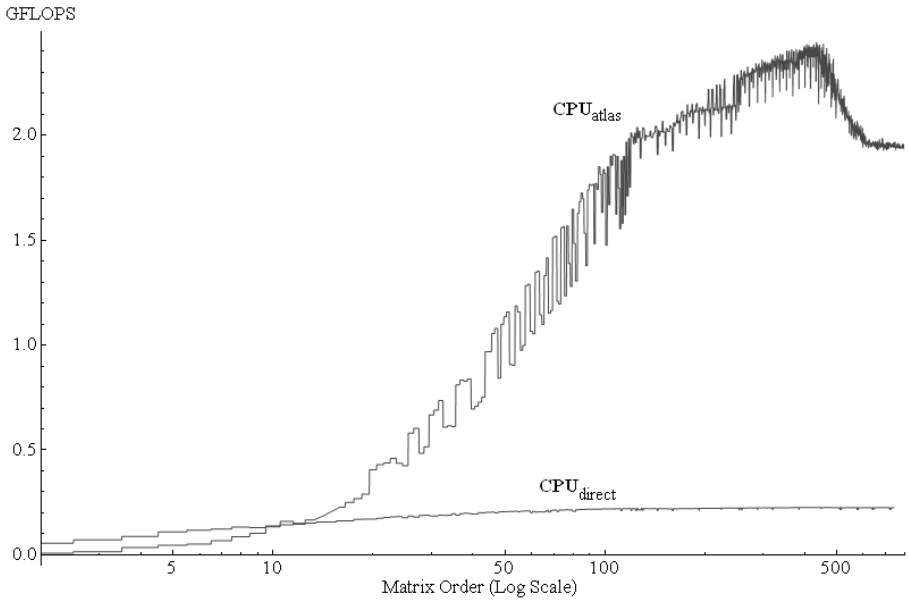

Fig. 8. Measured performance as a function of matrix order, for a direct ANSI-C implementation and an ATLAS optimized Conjugate Gradient method. Both measurements were made on a AMD Opteron 1220 CPU.

Table 6. Performance comparison for both the DIRECT and ATLAS optimized programs, running on a AMD Opteron 1220 CPU, with this FPGA hardware implementation using a Virtex5-330. FPGA vs CPU speed-up values are given for two extreme cases: when the FPGA has a single problem loaded on its pipeline and when this pipeline is full.

	DIRECT	ATLAS	FPGA^{single}		FPGA^{full}	
n	MFLOPS	MFLOPS	MFLOPS	SPEEDUP	MFLOPS	SPEEDUP
2	56	8	36	0.6×	1548	27×
5	110	46	90	0.8×	3060	28×
8	123	86	181	1.3×	5430	44×
10	138	135	227	1.7×	6129	44×
20	170	406	604	1.5×	12080	30×
30	185	667	1116	1.7×	17856	27×
40	195	696	1603	2.3×	24045	35×
50	206	1135	2216	2.0×	31024	27×
58	209	1285	2734	2.1×	35542	28×

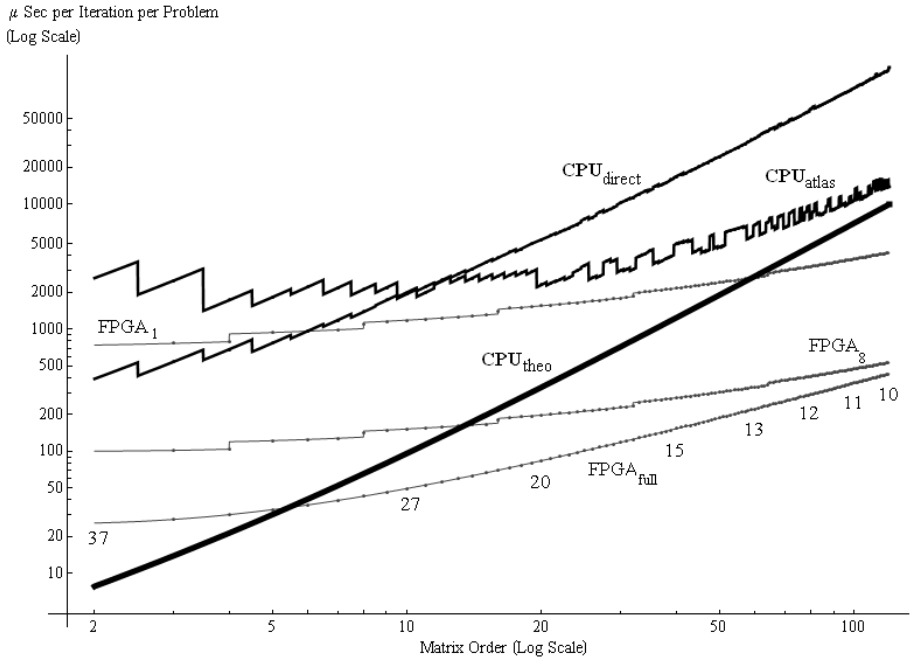

Fig. 9. Iteration time required for solving a number of CG problems as a function of matrix order on a CPU and FPGA. The bold lines represent a high-end CPU. The CPU_{theo} line depicts and ideal CPU running at 2.8 GHz. The CPU_{direct} line plots direct ANSI-C implementation while the CPU_{atlas} line represents the time required by the ATLAS optimized Conjugate Gradient software. Both these measurements are based on an AMD Opteron 1220 CPU. 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 (6).

representing a full pipeline. Below this last line, the number of problems being concurrently solved, in the pipeline, is given by (6). Two dark lines represent the measured CPU performance for a direct ANSI-C as well an ATLAS optimized software implementation. A third darker line represents the ideal case where a software implementation is performing at the CPUs peak theoretical capacity of 5.6 GFLOPS (2× clock frequency). Comparing the FPGA with a full pipeline and theoretical CPU, it is possible to observe that the FPGA is faster than the CPU for orders greater than 5. For a single problem in the FPGA pipeline, the theoretical CPU becomes slower than the FPGA for matrix orders above 60. 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 (6). Thus with only eight parallel problems, FPGA superiority is clearly established, even for low matrix orders. Comparison with both measured CPU implementations, the FPGA has demonstrated superiority even if only one problem is being processed, for matrix orders above 6.

5.3 Input/Ouput Considerations

As input, this method requires a matrix A and a vector b to be introduced. As output, it requires the solution vector x, which, under finite precision, is generated after at least n iterations [36]. This translates to the need of transferring $32(n^2 + 2n)$ bits per problem for a total number of problems given by (6). This transfer can occur over a period given by at least n times the clocks per iteration (5) because this is the time it takes to generate a solution and start a new problem. Combining these values we can deduce the minimum bit rate as given in (13). With the Virtex5-330 design solving problems of order 58 and running at 287MHz, this requirement translates to a data rate requirement of 1.1GB/s. This value is well within the operation range of PCI-Express [41].

$$\text{I/O Bits per Clock Cycle} = 32 - \frac{32}{n+3} \tag{13}$$

6 Conclusions

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

The implementation targets multiple medium-to-small dense systems, and may also be used when the exact solution (to within machine precision) may not be required, through early termination. An example of such a case arises in the inner loop of a truncated Newton method. While the FLOP count of direct and iterative solvers may indicate a preference for direct methods if an exact solution is required on small matrices, iterative and direct methods have different

opportunities to extract fine grain parallelism and pipelining.

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 [7] and in solving large banded matrices using the algorithm described in [6]. These banded systems arise in a number of problems including optimal control systems [9].

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 typically requires at least n iterations under finite precision [5] 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, this CG implementation is exceptionally suited for FPGAs.

This work outlined that with an effective use of parallelism, pipelining, number system and data-path, FPGAs can greatly outperform the top theoretical performance of high-end CPUs. The FPGA superiority is further emphasised when considering the typical CPU cache misses and pipeline stalls, as demonstrated in Section 5.3 with two CG software implementations. Results for this implementation, using the Virtex5-330, represented a superior performance of at least an order of magnitude comparing to a high-performance CPU.

Future work will be focused on the solution of structured systems originating in [9] and matrix sparsity will also be exploited to accelerate the solutions of special cases. Problem preconditioning will also be explored in order to optimize computation time.

References

1. I. Pournara, C. Bouganis and G. Constantinides, "FPGA-Accelerated Reconstruction of Gene Regulatory Networks," Proc. of Field Programmable Logic, 2005, pp. 323–328.
2. V. Bonato, R. Peron, D. Wolf, J. Holanda, E. Marques and J. Cardoso, "An FPGA Implementation for a Kalman Filter with Application to Mobile Robotics." Proc. Symposium on Industrial Embedded Systems, 2007, pp. 148–155.
3. O. Dandekar, W. Plishker, S. Bhattacharyya and R. Shekhar, "Multiobjective Optimization of FPGA-Based Medical Image Registration." Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines, 2008, p. (to appear).
4. S. Bayliss, C. Bouganis and G. Constantinides, "An FPGA Implementation of the Simplex Algorithm." Proc. International Conference on Field Programmable Technology, 2006, pp. 49–56.

5. M. Hestenes and E. Stiefel, "Methods of Conjugate Gradients for Solving Linear Systems," *Journal of Research of the National Bureau of Standards*, vol. 49, no. 6, pp. 409–436, Dec. 1952.
6. S. Wright, "Parallel Algorithms for Banded Linear Systems," *SIAM Journal on Scientific and Statistical Computing*, vol. 12, no. 4, pp. 824–842, July 1991.
7. E. Biglieri, R. Calderbank, A. Constantinides, A. Goldsmith, A. Paulraj, *MIMO Wireless Communications*. Cambridge Press, UK, 2007.
8. C. Kelley and E. Sachs, "Truncated newton methods for optimization with inaccurate functions and gradients," in *SIAM Journal on Optimization*, 1999, pp. 43–55.
9. S. Wright, "Interior Point Methods for Optimal Control of Discrete Time Systems," *Journal of Optimization Theory and Applications*, vol. 77, no. 1, pp. 161–187, Apr. 1993.
10. A. Roldao and G. Constantinides, "High Throughput FPGA-based Floating Point Conjugate Gradient Implementation," Proc. Applied Reconfigurable Computing, 2008, pp. 75–86.
11. M. Grote and H Simon, "Parallel preconditioning and approximation inverses on the Connection Machine," in *Proc. on Scalable High Performance Computing Conference*, 1992, pp. 76–83.
12. J. Kurzak, A. Buttari and J. Dongarra, "Solving Systems of Linear Equations on the CELL Processor Using Cholesky Factorization," in *IEEE Trans on Parallel and Distributed Systems*, 2008, pp. 1175–1186.
13. N. Fujimoto, "Faster matrix-vector multiplication on GeForce 8800GTX," in *IEEE Int. Symp. on Parallel and Distributed Systems*, 2008, pp. 1 – 8.
14. M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *IEEE Int. Symp. on Field Programmable Gate Arrays*, 2005, pp. 75–85.
15. Netlib, "Basic Linear Algebra Subprograms," http://www.netlib.org/blas/, Accessed on 22/08/2008, 2008.
16. K. Fatahalian, J. Sugerman and P. Hanrahan, "Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication," Proc on Graphics Hardware, 2004.
17. Y. Dou, S. Vassiliadis, G.K. Kuzmanov, G.N. Gaydadjiev, "64-bit Floating-Point FPGA Matrix Multiplication," in *Proc. on Int. Symp. on Field Programmable Gate Arrays*, 2005, pp. 86–95.
18. ——, "The potential of the cell processor for scientific computing," in *Proc. on 3rd Conference on Computing Frontiers*, 2006, pp. 9–20.
19. Clearspeed, "CSX600 Product Brief," http://www.clearspeed.com/-docs/resources/CSX600_Product_Brief.pdf, Accessed on 22/08/2008, 2006.
20. K. Goto and R. Geijn, "Anatomy of High-Performance Matrix Multiplication," in *ACM Trans. Math. Softw*, 2008, pp. 12:1–12:25.
21. S. Tomov, "GPUs for HPC - NVIDIA's Compute Unified Device Architecture," http://www.cs.utk.edu/ dongarra/WEB-PAGES/SPRING-2008/Lect09_GPU.pdf, Accessed on 24/08/2008, 2008.
22. M. Langhammer, "RSSI - 2008 - Foundation of FPGA Acceleration," http://www.rssi2008.org/proceedings/industry/Altera.pdf, Accessed on 23/08/2008 , 2004.
23. Cray, "XD1 Datasheet," http://www.cray.com/downloads/Cray-_XD1_Datasheet.pdf, Accessed on 2/03/2007, 2005.
24. SGI, "RASC RC100 Blade," http://www.sgi.com/- pdfs/3920.pdf, Accessed on 2/03/2007, 2006.

25. L. Zhuo and V. K. Prasanna, "High Performance Linear Algebra Operations on Reconfigurable Systems," in *Proc. of SuperComputing*, 2005, pp. 12–18.
26. K. Underwood, "FPGAs vs. CPUs: Trends in Peak Floating-Point Performance," in *Proc. ACM. Int. Symp. on Field-Programmable Gate Arrays*, 2004, pp. 171–180.
27. M. Langhammer, "Floating Point Datapath Synthesis for FPGAs," in *IEEE Int. Conf. on Field Programmable Logic and Applications*, 2008, pp. 355–360.
28. S. Haridas and S. Ziavras, "FPGA Implementation of a Cholesky Algorithm for a Shared-Memory Multiprocessor Architecture," *Journal of Parallel Algorithms and Applications*, vol. 19, no. 6, pp. 411–226, Dec. 2004.
29. G. Morris and V. Prasanna, "An FPGA-Based Floating-Point Jacobi Iterative Solver," in *Proc. of the 8th International Symposium on Parallel Architectures, Algorithms and Networks*, 2005, pp. 420–427.
30. A. N. O. Callanan, D. Gregg and M. Peardon, "High Performance Scientific Computing Using FPGAs with IEEE Floating Point and Logarithmic Arithmetic For Lattice QCD," in *Proc. Field Programmable Logic and Applications*, 2006, pp. 29–35.
31. V. L. O. Maslennikow and A. Sergyienko, "FPGA Implementation of the Conjugate Gradient Method," in *Proc. Parallel Processing and Applied Mathematics*, 2005, pp. 526–533.
32. IEEE, "754 Standard for Binary Floating-Point Arithmetic," http://grouper.ieee.org/groups/754/, Accessed on 18/03/2007, 1985.
33. M. He and K-V Ling, "Model Predictive Control on a Chip," in *Proc. of Int. Conf. on Control and Automation*, 2005, pp. 43–55.
34. G. Golub and F. Van-Loan, *Matrix Computations*. The Johns Hopkins University Press, 1996, p. 53.
35. J. Shewchuk, "An Introduction to the Conjugate Gradient Method Without the Agonizing Pain, Edition $1\frac{1}{4}$," `http://www.cs.cmu.edu/~jrs/`jrspapers.html#cg, Accessed on 28/02/2007, 2003.
36. G. Meurant, *The Lanczos and Conjugate Gradient Algorithms from theory to Finite Precision Computation*. SIAM, 2006, pp. 323–324.
37. Xilinx, "Core Generator Floating Point v3," http://www.xilinx.com/bvdocs/ipcenter/data_sheet/floating_point_ds335.pdf, 2006.
38. ——, "DS100 (v3.0) Virtex5 Family Overview - LX , LXT, and SXT Platforms," http://direct.xilinx.com/bvdocs/ publications/ds100.pdf, Accessed on 1/03/2007, 2007.
39. ATLAS, "Automatically Tuned Linear Algebra Software," http://math-atlas.sourceforge.net/, Accessed on 20/04/2008, 2008.
40. S. P. E. Corporation, "Floating Point Component of SPEC CPU2000 Benchmarks," http://www.spec.org/cpu2000/results/cpu2000.html, Accessed on 28/04/2008, 2008.
41. A. Bhatt, "PCI-Express - Creating a Third Generation I/O Interconnect," http://www.intel.com/technology/- pciexpress/devnet/docs/WhatisPCIExpress.pdf, Accessed on 19/06/2007, 2007.