A FLOATING-POINT SOLVER FOR BAND STRUCTURED LINEAR EQUATIONS *

Antonio R. Lopes[◇], *George A. Constantinides*[◇], *Eric C. Kerrigan*[†]

[◇]Electrical and Electronic Engineering Department, [†] Department of Aeronautics
Imperial College London
aroldao@ic.ac.uk, george.constantinides@ieee.org, e.kerrigan@imperial.ac.uk

ABSTRACT

Field Programmable Gate Arrays (FPGAs) have gradually been increasing their capacities and started to incorporate optimized coarse-grained modules such as BlockRAMs, multipliers, and even processors. These developments have extended their field of applications and one field that has been gaining significant interest is the acceleration of floating-point scientific computing. In this field, a recurring subtask is the solution of systems of linear equations. One well studied method that has proven to be very efficient in software and robust at finding such solutions is the Conjugate Gradient (CG) algorithm. In this paper we present a hardware CG method which takes advantage of the banded structure present in many common problems. With the flexibility provided by FPGAs, this implementation employs wide-parallelization to convert the per iteration computation time for an order n matrix with band width w from $\Theta(nw)$ clock cycles for a software implementation to $\Theta(n)$ in hardware. It also explores deep-pipelining so that solutions to P problems are produced every $\Theta(n)$ cycles opposed to every $\Theta(Pnw)$ cycles in software. Results demonstrate that performances up to 32 GFLOPs are achievable on a Virtex5-330T FPGA and a software comparison reports significant speed-ups in relation to high-end CPUs.

1. INTRODUCTION

In scientific computations one of the recurring problems is finding the solution of systems of linear equations. To find such solutions there are two main categories: direct methods, such as Gaussian elimination or the Cholesky Decomposition (where the solution is given by evaluating a derived formula) and iterative methods such as the Conjugate Gradient or the more generalized version GMRES. In this latter approach a solution is refined, based on previous results, until a certain acceptable value is reached.

*THE AUTHORS WOULD LIKE TO ACKNOWLEDGE THE SUPPORT OF THE EPSRC (GRANT EP/C549481/1 AND EP/E00024X/1)

As the size of a problems increases, matrix and vector operations become increasingly intensive in terms of computation 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 at the increased cost of resource utilization. Previous implementations have explored the benefits of using FPGAs. These implementations include a Cholesky approach that achieved a performance increase of 50% over software on a APEX EP20K15000E FPGA [1]; a Jacobi solver implementation on a Xilinx VirtexII Pro XC2VP50 which achieved a speedup of 1.3 to 36.8 relative to a high-end processor, depending on the matrix structure [2]; and two CG implementations. One of these implementations used the Logarithmic Number System (LNS) and reached up to 0.94 GFLOPS on a VirtexII-6000 [3], while the other used a rational number system representation and achieved 0.27 GFLOPS on a VirtexII Pro XC2VP4 [4]. Table 1 summarizes FPGA implementations of Conjugate Gradient method in terms of year of publication, number system, input problem structure, device and GFLOPS achieved.

This paper introduces the first Conjugate Gradient FPGA implementation to take advantage of a banded structure input matrix using IEEE floating point arithmetic standard [6]. The main contributions of this paper are thus:

- a parameterizable widely-parallel and deeply-pipelined hardware design for the solution of systems of diagonally banded linear equations,
- a study of resource utilization growth and performance scaling with matrix order n and band width w,
- results showing that a *sustained* performance of 32 GFLOPS is possible for a Virtex5-330T, translating to at least an order of magnitude speedup compared to the performance of an AMD Opteron 1220 CPU at 2.8 GHz.
- a comparison with previous implementations demonstrating how it is possible to process larger matrix orders by taking advantage of their structure.

Table 1. Previous FPGA-based Conjugate Gradient implementations.

Year	Reference	Number System	Device	GFLOPS	Structure	Max Order
2005	[3]	LNS	VirtexII-6000	1.1	*	*
2006	[4]	Rational	Virtex4-25	1.5	banded	3500
2007	[5]	FP single	VirtexII-6000	5	dense	16
2007	[5]	FP single	Virtex5-330	35	dense	58
2008	this	FP single	VirtexII-6000	8	banded	28
2008	this	FP single	Virtex5-330T	32	banded	236

* unspecified

A parameterizable hardware implementation of this algorithm is outlined, a comparison with software is made and results reported.

2. HARDWARE IMPLEMENTATION

This implementation explores the banded structure of the input matrix and performs the CG required matrix-by-vector operation one row at a time. As a result, we trade-off full parallelism with resource reutilization to increase matrix orders. The main aspects under analysis are the achievable speed, latency and throughput. It is is also important to understand resource growth with input matrix order and band width, and determine the maximum matrix order attainable with this computation scheme. This resource utilization is reduced compared to a dense matrix, since this implementation only stores band elements for each row of the structured matrix. Due to the nature of this algorithm the data iterates through the hardware as depicted in Fig. 1. These iterations are used to spread the loading of problems and eliminate potential I/O bottlenecks as described in [5].

The modular floating point arithmetic units used in this design are provided by Xilinx Core Generator FP v3. With all units having maximum latency, a formula for number of clock cycles required per iteration of the CG method can be derived (1). This expression grows linearly with the matrix order due to the sequential block vector-by-vector operations required to implement the matrix-by-vector multiplication. There is also a logarithmic growth of cycles per iteration due to the vector-by-vector reduction tree as depicted schematically in Fig. 2. It is also possible to deduce the number of problems that can be processed simultaneously in a pipeline fashion. This formula (2) is equivalent to the number of clock cycles per iteration divided by the number of clock cycles required to input each problem, n, from the internal FIFOs. In order to have an integer number of problems in this pipeline and hence utilize the full instantiated FP units a FIFO, of size κ, has be inserted into the system in order to ensure that the number of clock cycles required per iteration is a multiple of n.

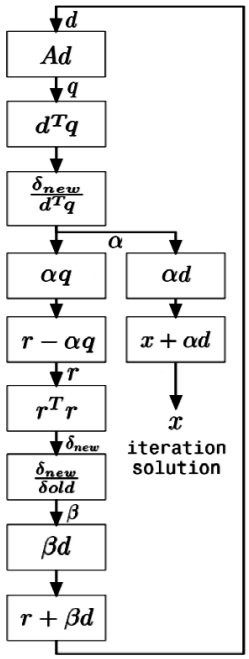

Fig. 1. Circuit data flow.

$$\frac{\text{Cycles}}{\text{Iteration}} = 29n + 12\lceil \log_2 w \rceil + 115 + \kappa \quad (1)$$

$$P_{max} = \text{Pipeline depth} = \frac{29n + 12\lceil \log_2 w \rceil + 115 + \kappa}{n} \quad (2)$$

Since the latency of the addition floating point module Lm is 12, this is the number of interleaved pipelined problems required to efficiently utilize the the vector-by-vector operation. In this scheme a simple feedback loop can be used to implement the required summation reduction tree as described in Fig 2. This property affects the clock cycles required per iteration (1) as well as the total number of pipelinable problems P_{max} given by (2).

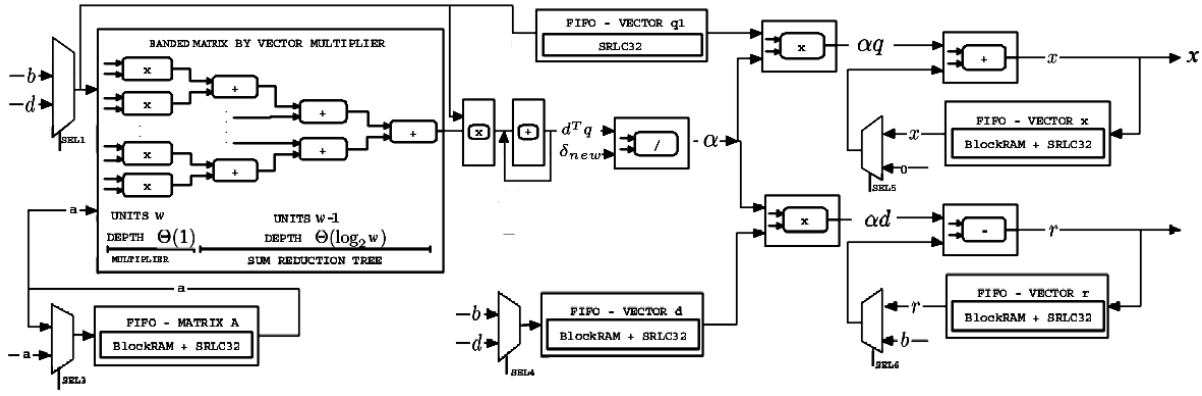

Fig. 2. Circuit data flow and schematic vector by vector multiplication.

2.1. Resource utilization

Instantiated resources are optimized and are kept in operation throughout each iteration, avoiding any pipeline stalls. This implementation requires a number of floating point computational units that grows linearly with matrix band width, w, and is given by two times this band width plus thirteen. Theoretical storage utilization grows as $\Theta(nw)$ due to the need to store nw elements for each input matrix A. To store these values, in a FIFO, a mixture of embedded BlockRAMs and SRLC32 primitives are used [7]. 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. These discrete configurations explain the BlockRAM utilization jumps in Fig. 3. This figure plots the resource usage as a function of the matrix order when the band width is equal to five, $w = 5$. Results showed that for the extreme case where band width is equal to its matrix order, $w = n$, the Virtex5-330T can accommodate matrix orders up to 92. For band widths smaller than 92 the limiting resources are the available REGisters. When this is not the case, the limiting resources becomes the DSP48 embedded blocks. Best fit resource usage functions for DSP48Es, LUTs, and REGisters as a function of matrix order and band width are described in (3), (4), and (5) respectively.

$$\text{DSP48Es}(n, w) = 2w + 8 \tag{3}$$

$$\text{kLUTs}(n, w) = 676 + 36n + 87w \tag{4}$$

$$\text{kREGs}(n, w) = 1007 + 91n + 108w \tag{5}$$

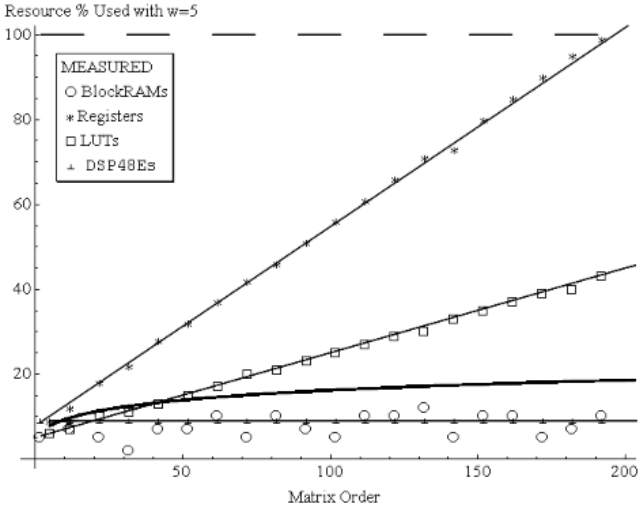

Fig. 3. BlockRAMs, REGisters, DSP48s, and LUTs resource utilization with matrix order for the Virtex5-330T. Light lines represent the best fit based on the synthesis reports for band width equivalent to five, $w = 5$. The dark line represent the upper bound of BlockRAM utilization.

2.2. Software comparison and discussion

The peak FLOPS performance (6) is given by the total number of active floating point units, which is a function of the matrix band width, w, and the maximum achievable frequency.

$$\text{FLOPS} = FP_{units} \times \text{FPGA\_freq} \tag{6}$$

Acceleration relative to software is provided by pipelining and parallelization. An important speedup is provided by the matrix-by-vector operation, as depicted in Fig. 2. This operation which requires $n(2w - 1)$ sequential operations in software, is reduced to the latency of the multipli-

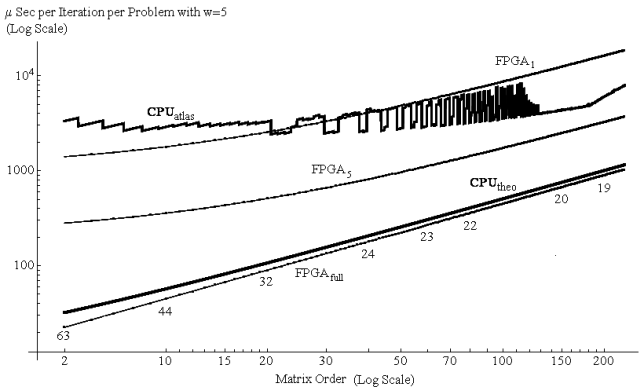

Fig. 4. Iteration time per CG problem on a CPU and FPGA with $w = 5$. The bold line describes a high-end theoretical peak CPU performance and a measured ALTAS implementation, and three lighter lines show the FPGA Virtex5-330T implementation using a full pipeline ($P = P_{max}$), the pipeline with 5 problems, and with a single problem.

cation core, which is given by a constant plus the latency of the adder tree plus $n - 1$ cycles. The performance improvement between the FPGA and CPU is is given by (7). In this formula, the numerator comprises of the FPGA's operating frequency, FPGA_freq, times the number of problems on the pipeline, P, times the number of clock cycles required for a software iteration and a denominator that is comprised of the CPU's operating frequency, CPU_freq, times the clocks per iteration on the FPGA. This is assuming the CPU is able to process a floating point operation every clock cycle, which is an overly optimistic view on CPUs. When the FPGA's pipeline is full, with P_{max} problems given by (2), this formula is reduced to (8).

$$\text{Speedup}_P = \frac{\text{FPGA\_freq} \times (2wn + 4w + 5n) \times P}{\text{CPU\_freq} \times (29n + 12\lceil \log_2 w \rceil + 115)} \quad (7)$$

$$\text{Speedup}_{full} = \frac{\text{FPGA\_freq} \times (2wn + 4w + 5n)}{\text{CPU\_freq} \times n} \quad (8)$$

For the case where $w = 5$, Fig. 4 compares the growth in processing time, for each CG iteration, on an FPGA and on a AMD Opteron 2.8 GHz. One bold line represents the top theoretical performance of the high end CPU performing a FP operation every clock cycle while the other represents the same CPU running an ATLAS optimized implementation of the CG algorithm [8]. Three lines are shown for the FPGA implementation: one representing a full pipeline, other for 5 problems in this pipeline and another for a single problem. It is possible to verify that the performance of the FPGA, with pipeline full, is superior to the CPU theoretical maximum, even for the low parallelism provided by $w = 5$. For the case where there is only one problem in the FPGA's pipeline, performance is matched and surpassed for matrix orders above a certain threshold, which itself varies with the degree to which the pipeline is filled.

3. CONCLUSIONS

This paper has described a deep-pipeline and wide-parallel block Conjugate Gradient FPGA implementation using the IEEE single precision floating point standard for problems with a banded input matrix. It was demonstrated that matrix bands up to 92 are achievable, with peak floating point performance above 32 GFLOPS for the Virtex5-330T device. With the lower capacity and readily available Virtex2-6000, it is possible to process matrices with band widths up to 28 and reach the floating point performance of 8 GFLOPS. Taking advantage of hardware parallelization, the required processing time for a single iteration is reduced from $\Theta(nw)$ to $\Theta(n)$, at the cost of increasing hardware utilization from $\Theta(1)$ to $\Theta(w)$. With this parallelization combined with pipelining it is possible to outperform a high-end CPU. In the case of the higher density FPGA, this performance is improved by at least an order of magnitude. This implementation also takes advantage of the iterative nature of the CG algorithm in order to spread the data transfer, which then becomes well within the FPGA's I/O capacity.

4. REFERENCES

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] A. Roldao and G. A. Constantinides, "A High Throughput FPGA-based Floating Point Conjugate Gradient Implementation," in *Proc. Applied Reconfigurable Computing*, 2008, pp. 75–86.

[6] IEEE, "754 Standard for Binary Floating-Point Arithmetic," http://grouper.ieee.org/groups/754/, Accessed on 18/03/2007.

[7] Xilinx, "DS100 (v3.0) Virtex5 Family Overview," http://www.xilinx.com/, Accessed on 1/03/2007.

[8] ATLAS, "Automatically Tuned Linear Algebra Software," http://math-atlas.sourceforge.net/, Accessed on 20/04/2008.