ICCM2012
November 25-27, 2012, Gold Coast, Australia
www.ICCM-2012.org

Solving Tri-Diagonal Linear Systems using Field Programmable Gate Arrays

D. J.Warne*[1,2], N. A. Kelson[1], R. F. Hayward[2]
[1]*High Performance Computing and Research Support Group*
Queensland University of Technology, Brisbane, Queensland, Australia
[2]*School of Electrical Engineering and Computer Science*
Queensland University of Technology, Brisbane, Queensland, Australia
**Corresponding author: david.warne@qut.edu.au*

Abstract

In this paper, we present the outcomes of a project on the exploration of the use of Field Programmable Gate Arrays (FPGAs) as co-processors for scientific computation. We designed a custom circuit for the pipelined solving of multiple tri-diagonal linear systems. The design is well suited for applications that require many independent tri-diagonal system solves, such as finite difference methods for solving PDEs or applications utilising cubic spline interpolation. The selected solver algorithm was the Tri-Diagonal Matrix Algorithm (TDMA or Thomas Algorithm). Our solver supports user specified precision thought the use of a custom floating point VHDL library supporting addition, subtraction, multiplication and division. The variable precision TDMA solver was tested for correctness in simulation mode. The TDMA pipeline was tested successfully in hardware using a simplified solver model. The details of implementation, the limitations, and future work are also discussed.

Keywords: FPGA, Matrix Factorisation, Hardware Acceleration, RASC, Tri-Diagonal Matrix Algorithm, Reconfigurable Computing

Introduction

Tri-diagonal linear systems arise in many areas of computational science which involve models of mechanics. For example, some partial differential equations (PDEs) result in a tri-diagonal system under discretisation (Mallik, 2001). In the context of computational fluid dynamics (CFD), tri-diagonal systems arise from a system of PDEs which need to be solved many times per simulation time-step. Repeated solving of tri-diagonal systems represents a large percentage of the total runtime of such simulations (Oliveira, 2008).

Reconfigurable hardware such as field programmable gate arrays (FPGAs) can be used as co-processors for high performance computing applications. A user application can reconfigure an FPGA at runtime with a custom circuit designed to achieve hardware acceleration for an application specific task. This can improve application performance considerably with very little increase in power consumption (Sundararajan, 2010).

In this paper, we discuss our experiences in using an FPGA as a co-processor for scientific applications. As a test case, we implemented a tri-diagonal linear system solver for FPGAs. The

design of our solver pipeline allows a user application to stream a new tri-diagonal system to the co-processor before a solution to previous system in complete; this feature is unique to our design. The design takes advantage of the fine and coarse grain parallelism available on an FPGA. The solver utilises our custom floating point arithmetic circuits that can be set to a user specified precision. In simulation, a speedup of ~2x over a naïve C implementation is seen (~70x speedup if data transfer overhead is ignored).

We have designed our tri-diagonal solver co-processor for applications which require many independent (relatively small) tri-diagonal systems to be solved. Such applications would arise from some numerical models defined on regular lattices (including CFD application). Applications involving cubic spline interpolation could also benefit since this operation can be formulated as solving a tri-diagonal system (Press, 1992).

Background

Reconfigurable Computing

Reconfigurable computing devices are integrated circuits (ICs) in which the user has some control over the data path at runtime (Azarian, 2009). These devices provide a platform for the design of accelerators which are specific to an application, possibly even being reconfigured for different tasks throughout application execution.

Though reconfigurable computing devices have historically been used for computationally light tasks (due to low potential computing densities for their cost), recent advances in technology have seen these devices become more applicable to high performance computing (HPC) systems (Zhuo, 2008). Particularly as power utilisation (i.e. FLOPS/Watt) is becoming an increasing concern for the HPC community (Lee, 2010).

Field programmable gate arrays (FPGAs) are a type of programmable logic device that can be used for reconfigurable computing. FPGAs consist of an array of configurable logic blocks (CLBs), and block RAM (BRAM) which can all be connected through programmable interconnects (Azarian, 2009; Compton, 2002). CLBs consist of a number of look-up tables (LUTs), multiplexers and logic gates which can be configured by the user to implement custom logic operations. The output from one CLB could be connected to the input of another CLB or stored in BRAM. This architecture provides a platform for massive scale parallelism, which is a main attraction for use in HPC (Sundararajan, 2010).

The target system for our design was SGI's reconfigurable application-specific computing (RASC) RC100 blade (SGI, 2007) operating with an SGI Altix 4700 server. The RC100 contains two Xilinx Virtex-4 LX200 FPGAs. Each of these FPGAs is connected to five 8MB QDR SRAM DIMMs. A host application can copy data to these SRAMs via the NUMAlink using SGI RASC API function calls. For our application two SRAMs per FPGA are dedicated for input and two for output. Each FPGA may therefore read and write 128 bits per clock cycle.

Tri-diagonal Linear Systems

Linear algebra operations such as matrix factorisation are the backbone of many scientific applications (Datta, 2010). As such, it is not surprising that a great deal of reconfigurable HPC research is dedicated to the accelerating these tasks. Efficient LU decomposition designs have been shown to achieve performance increases over a traditional CPU-based system (Johnson, 2008; Wu,

2012). For BLAS level 1 and 2 operations GPUs have been shown to achieve high performance, but at a much higher energy cost than FPGA-based solutions (Kestur, 2010).

Tri-diagonal systems arise in many areas of science and engineering, often through the application of finite difference methods to boundary value problems (Fischer, 1969; Usmani, 1994; Mallik, 2001). For our research, we investigated the design of an FPGA-based solution for solving multiple tri-diagonal linear systems. The algorithm we applied was the Thomas algorithm, commonly referred to as the tri-diagonal matrix algorithm (TDMA). Oliveira et. al. (2008) designed a TDMA co-processor for a CFD application; however their solution was very specific to that particular CFD model. We proposed a general TDMA co-processor. Our co-processor was designed to pipeline multiple tri-diagonal system, thus maximising throughput.

The flexibility of an FPGA platform allows one to deviate from standard floating point specifications. This provides more opportunity for performance gains if a particular application does not require full single or double precision floating point, as less of the FPGAs logic fabric is utilised by the floating point operations. We have designed a library for floating point arithmetic where the individual bit-widths of the exponent and mantissa can be set by the user.

Pipelined TDMA Solver Design

TDMA Implementation

TDMA is a special case of LU-decomposition, in which the coefficient matrix is a banded matrix with a bandwidth of 1. That is, the system has the following form:

$$\begin{aligned} &a_{11}x_1 + a_{12}x_2 = b_1 \\ &a_{i(i-1)}x_{i-1} + a_{ii}x_i + a_{i(i+1)}x_{i+1} = b_i,\ i \in [2,3,...,n-1] \\ &a_{n(n-1)}x_{n-1} + a_{nn}x_n = b_n \end{aligned} \tag{1}$$

Typically Eq (1) is stored as four arrays of n elements,

$$\begin{aligned} \mathbf{U} &= \left[0, a_{12}, a_{23}, ..., a_{(n-1)n}\right] \\ \mathbf{D} &= [a_{11}, a_{22}, a_{33}, ..., a_{nn}] \\ \mathbf{L} &= \left[a_{21}, a_{32}, ..., a_{n(n-1)}, 0\right] \\ \mathbf{x} &= [b_1, b_2, b_3, ..., b_n] \end{aligned} \tag{2}$$

where **L**, **U**, and **D** store the lower, upper, and diagonal bands of the coefficient matrix respectively. The array **x** will store the right hand side vector, but as the algorithm progresses it will be updated with the solution vector.

Using the data structures from Eq (2), the TDMA algorithm can be expressed as a factorisation/forward substitution step followed by a backward substitution step (see Fig 1).

$\mathbf{L}(1) \Leftarrow \mathbf{L}(1) \div \mathbf{D}(1)$
for i in 2 to n
$\mathbf{D}(i) \Leftarrow \mathbf{D}(i) - \mathbf{L}(i-1) \times \mathbf{U}(i)$
$\mathbf{L}(i) \Leftarrow \mathbf{L}(i) \div \mathbf{D}(i)$
$\mathbf{x}(i) \Leftarrow \mathbf{x}(i) - \mathbf{L}(i-1) \times \mathbf{x}(i-1)$
end

$\mathbf{x}(n) \Leftarrow \mathbf{x}(n) \div \mathbf{D}(n)$
for i in n-1 to 1
$\mathbf{x}(i) \Leftarrow \mathbf{x}(i) - \mathbf{x}(i+1) \times \mathbf{U}(i+1)$
$\mathbf{x}(i) \Leftarrow \mathbf{x}(i) \div \mathbf{D}(i)$
end

Figure 1. Factorisation and forward substitution (left) and backward substitution (right)

We designed a hardware module for both these steps which implement the inner portion of the loop. In each case, a new row of the matrix is loaded into the module at the rising edge of each clock cycle. Values of the new row that will be required in the next iteration are stored in internal registers. The data flow diagrams of these modules are some in Fig 2 and Fig 3 respectively.

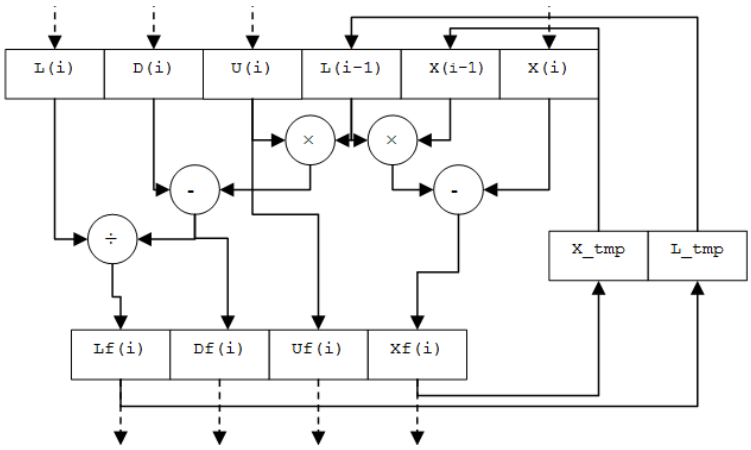

Figure 2. Factorisation and forward substitution module

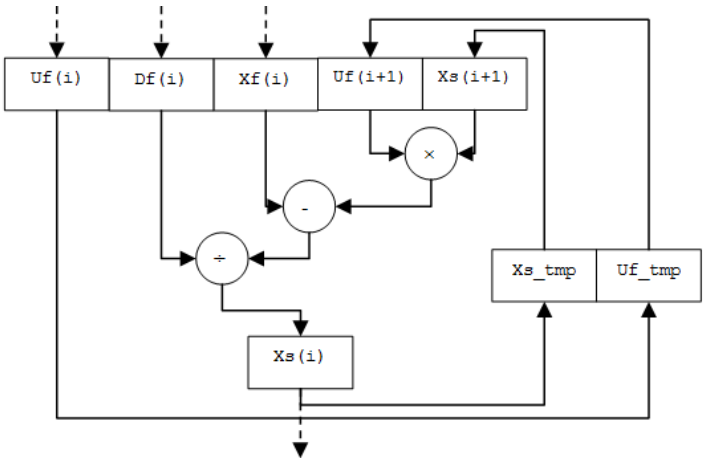

Figure 3. Backward substitution module

Because of the flexibility of the FPGA platform, we are able to perform independent operations in the same clock cycle.

Variable Precision Floating Point

One key feature of our solver design is that is utilises VHDL generic statement to allow implementation at variable precisions. Our floating point designs can be set to user specified bit

widths for the exponent and mantissa, and then re-synthesised for that specific application. Although in practice IEEE-754 32-bit or 64-bit interchange formats are used, it may be that a specific application does not require such precision (or requires more).

Our design is minimalistic, in that all rounding is simplified with truncation and no special cases such as NANs and Infinities handled. This allows for more floating point units to be placed on the FPGA, resulting in a higher compute density. Due to numerical errors introduced through truncation, the application has to be able to operate given the accuracy constraints, which are discussed in more detail in the Results section.

The Solver Pipeline

For a single linear system, the dependence the backward substitution step has on the factorisation/forward substitution step cannot be removed. The factorisation/forward substitution loop must completely execute before the backward substitution can begin. If we naively assume that a user cannot load a new system onto the FPGA co-processor until the previous system is solved then we require c_{total} clock cycles given by

$$c_{total} = 2 \times N \times M \,, \tag{3}$$

where M is the number of systems and N is the number of equations per system (assumed to be constant for the sake of simplicity.).

In the naive approach resulting in Eq (3) the factorisation/forward substitution loop sits idle while the backward substitution completes. This results in a waste of logic fabric. Instead, we have designed a solver pipeline that allows an application to load as soon as the factorisation/forward substitution step has completed on the previous system, as shown in Fig 4. This provides coarse gain parallelism which effectively cuts the number of cycles required in half. We have our new equation for $c_{total.}$,

$$c_{total} = N \times (1 + M) \,. \tag{4}$$

This has an improvement of $(M-1) \times N$ fewer cycles. However, it does require that two complete systems are stored on the FPGA board. In our case, all remaining logic fabric was used to implement two banks of registers. These two register banks are connected to the factorisation/forward substitution module output via a de-multiplexer (DEMUX), and to the backward substitution module input via a multiplexer (MUX).

Factorise Module	M=1 R=1	M=1 R=2	.....	M=1 R=N	M=2 R=1	M=2 R=2	.....	M=2 R=N	M=3 R=1	.....	
BSub Module	.....	.....	.....	.....	M=1 R=N	M=1 R=N-1	.....	M=1 R=1	M=2 R=N	.....	
Cycle	1	2	.....	N	N+1	N+2	.....	2N	2N+1	.....	

Figure 4. Coarse grain parallelism due to pipelining

When the factorisation/forward substitution output is routed to one register bank, the other register bank is routed to the backward substitution input. The MUX and DEMUX are switched when an "end-of-system" symbol is encountered (i.e., $i = N$). The data flow for this design is shown in Fig 5.

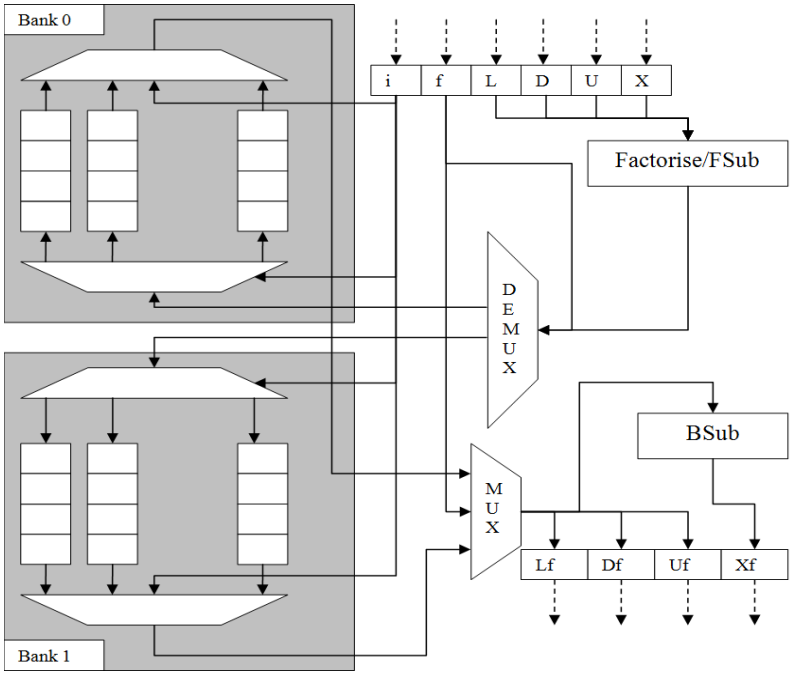

Figure 5. Full TDMA pipeline

In Fig 5, the input data is read from the dedicated input SRAM, and output is written to the output SRAM. The user host application simply writes to and reads from these SRAM via SGI API functions.

Results

In this section, we discuss the results of our analysis of our TDMA FPGA co-processor. We particularly focus on the accuracy and performance aspects of the design.

Accuracy

There is a trade-off between speed and accuracy, and accuracy is particularly important for us to discuss since we used our own custom floating point units. We compared the average and maximum relative errors of our floating point operation with the same operations applied in a standard C program. To make this a meaningful comparison with IEEE-754 32-bit operations, the exponent and mantissa bit widths in our design were configured to be the same as specified in IEEE-754 (i.e. 8 and 23 bits respectively). We evaluated the accuracy of each operation for 180 randomly generated operand pairs. The results (see Table 1.) indicate that even with the minimalistic approach we have taken the maximum error seen is approximately 6e-07 for division.

Table 1. Floating Point Error for 180 Random Operand Pairs

Floating Point Operation	Relative Error	
	Mean	Maximum
+	3.01e-08	2.91e-07
−	2.40e-08	4.36e-07
×	4.44e-08	1.12e-07
÷	3.33e-08	6.21e-07

While the worse performance was for division, this could be improved by operating in a slightly higher precision internally, or adding more iterations of Goldschmidt's algorithm (Goldberg, 2007).

The accuracy of the TDMA pipeline was not as good as that of the individual floating point operations, but it would still be acceptable for many applications (see Table 2).

Table 2. Overall TDMA Pipeline Error

Relative Error	
Mean	Maximum
1.79e-05	1.79e-04

Simulation

FPGA vendors (e.g. Xilinx) provide tools for engineers to simulate the behaviour and performance before implementing the design on the FPGA hardware itself. We compared the theoretical performance against an ANSI C TDMA implementation. Comparisons were made for both the Intel C compiler (icc) and the GNU C compiler (gcc). Comparisons of performance for 16 MB of 5x5 tri-diagonal systems are shown in Table 3.

Table 3. Performance CPU Vs FPGA Simulation

Compiler and Flags	Runtime	Speedup
icc -O0	375 ms	560x
gcc -O0	142 ms	212x
gcc -O3	50 ms	75x
gcc -O2	48 ms	72x
icc -O2	7 ms	10x
icc -O3	3 ms	4.5x
FPGA simulation	0.671 ms	1x

We have applied deep data paths in our design. As a result, the board needs to be clocked very low (< 5 MHz). Despite this, our tests in simulation displayed speedup of around 560x over C code compiled with no optimisation flags and around 4.5x over the highest optimisation level. Considering that the C code executes on a CPU core clocked in the GHz (Intel X5650), this is theoretically quite good. As discussed in the following section, these theoretical results were not indicative of the real performance, however, further improvements could be made to our design to increase the minimum clock period and improve the real performance.

Implementation

Our design required a clock frequency that was lower than the minimum setting of the RC100 Blade (which is 5 MHz). Clearly our design needs to be refined further. However, for the purposes of comparing theoretical performance with actual runtimes, we used a very cut-down division module (as this module had the deepest data path, and was responsible for the lower clock frequency constraint). Surprisingly the actual performance in hardware was significantly poorer than that of the simulations prediction. The execution took around 590 ms! This is slower than all of

the CPU runs. Certainly some degradation in performance was expected due to memory transfer over heads, but our estimates predicted this to be around 20 ms as a maximum.

On further investigation we have found that the performance of the RC100 blade is significantly affected by its memory transfer settings from host to FPGA SRAM (Mitra, 2006; Wielgosz, 2009). It may be possible that there our system was not configured to allow for maximum throughput. Further tests could not be made as our Altix 4700 server was been decommissioned soon after we ran our hardware test. However, our design is modular enough that it would not be difficult to implement on an alternate platform. A future goal for this project is to re-implement the memory interface design to operate with our Nallatech PCIe-280 Virtex-5 board. This will also provide us with more logic fabric and faster data transfer rates.

Conclusions

We have presented our work in designing a custom variable precision TDMA co-processor for HPC applications. It is clear that more work needs to be done to get the design to a similar performance to that of the simulation predictions. However, if this performance could be achieved then this would provide a very low-powered accelerator solution to HPC applications requiring the solutions of multiple tri-diagonal linear systems (e.g., CFD, and spline fitting). Our future work will focus on further optimisation of our floating point modules, and the targeting of a more modern FPGA platform.

References

Azarian, A. and Ahmandi, M. (2009), Reconfigurable computing architechture survey and introduction. in *ICCSIT 2009*, Beijing, pp. 269-274.

Compton, K. and Hauck, S. (2002), Reconfiguable computing: a survey of systems and software. *ACM Computing Surveys*, 3, pp. 171-210.

Datta, B. N. (2010), *Numerical Linear Algebra and Application* (2 ed.): SIAM, Philadelphia, Pa.

Fischer, C. F., and Usmani, R. A. (1969), Properties of some tridiagonal matrices and their application to boundary value problems. *SIAM J. Numer. Anal.*, 6(1), pp. 127-142.

Goldberg, R., Even, G., and Seidel, P.-M. (2007), An FPGA implementation of pipelined multiplicative division with IEEE Rounding. in *FCCM 2007*, pp. 185-194.

Johnson, J., Chagnon, T., Vachranukunkiet, P., Nagvajara, P., and Nwankpa, C. (2008), Sparse LU Decomposition using FPGA. in *PARA'08*.

Kestur, S., Davis, J. D., and Williams, O. (2010), BLAS Comparison on FPGA, CPU and GPU. in *ISVLSI 2010*, pp. 288-293.

Lee, J., Sun, J., Peterson, G. D., Harrison, R. J., and Hinde, R. J. (2010), Power-aware Performance of Mixed Precision Linear Solvers for FPGAs and GPGPUs. in *SAAHPC'10*.

Mallik, R, K. (2001), The inverse of a tridiagonal matrix. *Linear Algebra and its Applications*, 325, pp. 109-139

Mitra, A., Yao, G., and Najjar, W. (2006), Performance Analysis of SGI RASC RC100 Blade on 1D DWT. in *Reconfigurable Systems Summer Institute 2007 at UIUC*, Urbana-Champaign, IL.

Oliveira, F., Santos, C. S., Castro, F. A., and Alves, J. C. (2008), A Custom Processor for TDMA Solver on CFD Application. in *ARC '08*, pp. 63-74.

Press, W. H., Flannery, J. D., Teukolsky, S. A., and Vetterling, W. T. (1992), *Numerical Recipes in FORTRAN: The art of Scientific Computation* (2 ed.): Cambridge University Press, Cambridge, England, pp. 107-109

SGI (2007), *Reconfigurable Application-Specific Computing User's Guide*: Silicon Graphics, Inc.

Sundararajan, P. (2010), High Performance Computiing Using FPGAs. *Xilinx White Paper*, pp. 1-15.

Usmani, R. A. (1994), Inversion of Jacobi's Tridiagonal Matrix. *Computers Math. Applic.*, 27(8), pp. 59-66

Wielgosz, M., Jamro, E., and Wiatr, K. (2009), Accelerating Calculations on the RASC Platform: A Case Study of the Exponential Function. in *ARC '0*, pp. 306-311.

Wu, G., Dou, Y., Sun, J., and Peterson, G. D. (2012), A High Performance and Memory Efficient LU Decomposer on FPGAs. *IEEE Transactions on Computers*, 61, pp. 366-378.

Zhuo, L. and Prasanna, V. K. (2008), Scalable Hybrid Designs for Linear Algebra on Reconfigurable Computing Systems. *IEEE Transactions on Computers*, 57, pp. 1661-1675.