Optimising Memory Bandwidth Use for Matrix-Vector Multiplication in Iterative Methods

David Boland [1], George A. Constantinides [2]

Electrical and Electronic Engineering Department,
Imperial College London, London SW7 2AZ, UK
[1] `dpb03@ic.ac.uk` [2] `george.constantinides@ieee.org`

Abstract. Computing the solution to a system of linear equations is a fundamental problem in scientific computing, and its acceleration has drawn wide interest in the FPGA community [1–3]. One class of algorithms to solve these systems, iterative methods, has drawn particular interest, with recent literature showing large performance improvements over general purpose processors (GPPs). In several iterative methods, this performance gain is largely a result of parallelisation of the matrix-vector multiplication, an operation that occurs in many applications and hence has also been widely studied on FPGAs [4, 5]. However, whilst the performance of matrix-vector multiplication on FPGAs is generally I/O bound [4], the nature of iterative methods allows the use of on-chip memory buffers to increase the bandwidth, providing the potential for significantly more parallelism [6]. Unfortunately, existing approaches have generally only either been capable of solving large matrices with limited improvement over GPPs [4–6], or achieve high performance for relatively small matrices [2, 3]. This paper proposes hardware designs to take advantage of symmetrical and banded matrix structure, as well as methods to optimise the RAM use, in order to both increase the performance and retain this performance for larger order matrices.

1 Introduction

The large amount of resources available on modern FPGAs have made them suitable for accelerating floating point applications. The solution to a system of linear equations is a recurring sub-problem within many scientific computing problems [7], and hence there is considerable value in accelerating this operation. Iterative methods are one type of algorithm to solve a system of linear equations and recently studies have shown that by using FPGAs on these algorithms it is possible to achieve performance improvements of up to an order of magnitude over general purpose processors (GPPs) [2, 3].

The reason FPGAs are capable of accelerating iterative methods, such as the conjugate gradient and minimum residual (MINRES) algorithms [8], is that these algorithms often contain lots of inherent parallelism, the majority of which originates from a repeated matrix-vector multiplication. Furthermore, as it can be shown that in general this operation consumes the best part of the execution time of the algorithms [9], parallel execution of this operation significantly reduces the overall execution time. Unfortunately, highly parallel matrix-vector multiplication circuits require the use of the on-chip RAM to buffer data so as to provide the desired bandwidth, and hence the available RAM on the FPGA limits the maximum matrix order that can be implemented.

Given many problems in scientific computing result in large matrices, it is of interest to determine the extent to which this performance can be maintained for such matrices. To achieve this, this paper proposes hardware architectures for performing matrix-vector multiplication that can take advantage of banded matrix structure, symmetry within the matrix, or both. Banded matrices are sparse matrices of a specific structure such that all of the non-zero values lie within a specified bandwidth of the diagonal, and these arise in many problems, for example when solving partial differential equations [10]. Symmetric matrices are square matrices that are equal to its transpose, and these are of particular interest as both conjugate gradient and MINRES algorithms will only converge to a solution provided the input matrix is symmetric.

This paper will demonstrate that by exploiting these properties, it is possible to reduce the RAM requirements. Furthermore, as embedded RAMs on FPGAs have specific structures, this paper proposes an optimisation strategy using integer linear programming (ILP) in order to translate this reduced RAM requirement into the minimum use of embedded RAMs. Finally, it will be shown that by applying these strategies for saving RAM, it is possible to move the source of limitation on parallelism from RAMs to be a function of the look-up tables and dedicated multipliers which are used to construct floating point components. As a result, this work also goes on to describe parameterisable hardware architectures, depending upon matrix characteristics such as the bandsize and matrix order, which can scale to larger matrices and obtain as much parallelism as possible. The main contributions of this paper can be summarised as follows:

– Hardware architectures for banded matrices and symmetric matrices that can significantly extend the scalability to large order matrices and achieve higher degrees of parallelism,
– An optimisation strategy to reduce the number of embedded RAMs depending upon problem specification,
– Hardware architectures that can trade parallelism with FPGA resources to achieve greater scalability.

This paper begins with a survey of existing implementations of matrix-vector multiplication in Section 2, before describing our architectures in Section 3. Some results showing the benefit of this approach are then given in Section 4, before the work is concluded in Section 5.

2 Related Work

There has been a large amount of research into FPGA acceleration of floating point matrix-vector multiplication. The two main factors that distinguish these approaches are the method to store the matrix and how the on-chip RAM is utilised.

At one extreme is the work by El-Kurdi *et al.* [5]. This implements a streaming approach such that the 'stripes' containing the non-zeros for the matrix and the vector are held in off-chip RAM and streamed through a set of processing elements, with one processing element for each stripe to achieve the maximum parallelism. The advantage of this approach is that due to the streaming nature, it can operate on arbitrarily large matrices, provided there is sufficient off-chip RAM. The disadvantage of this approach is that the maximum number of stripes and hence the maximum parallelism is limited by the I/O bandwidth from a 1.76 single precision GFLOPs peak on a Stratix S80 to 1.5 single precision GFLOPs.

The work by Morris *et al.* [1] is slightly different, storing the matrix in a more traditional fashion, using Compressed Sparse Row (CSR) format [11], which consists of all non-zero values of the matrix, an index to the column the value lies in, and an index for when each new row begins. The hardware then must match several matrix values with their corresponding vector element and performs the multiplications in parallel, before accumulating the results. In comparison to the work by El-Kurdi *et al.*, it stores the vector on-chip to perform these parallel multiplications and this slightly improves the maximum performance. However, it is still limited by I/O, and storing these vectors on chip means its scalability depends on the available RAM to store these vectors. Further work by Zhuo *et al.* [4,12] examined in detail the floating point data hazards, improving the reduction circuits that accumulate these results to use less silicon, but the performance was still limited by I/O to be 2.88 single precision GFLOPs, or 2.16 GFLOPs in practical simulations on a Virtex2 Pro.

DeLorimier and DeHon [6] create an implementation which similarly targets sparse matrix-vector multiplication for matrices stored in CSR format, but it is specifically aimed at accelerating this function within iterative methods. This operation is special as the same matrix is used for every iteration, and hence this approach suggests loading the matrix into on-chip embedded RAM once, from which it can be re-used multiple times allowing much more parallelism as a result of the significantly higher memory bandwidth. Using this method, it achieved a performance of up to 1.5 sustained double precision GFLOPs on a Virtex2-6000. However, the maximum matrix size and performance in this work is limited by the available memory to store the matrix.

The work by Lopes *et al.* [2] and previous work by the authors [3] acknowledge that the more modern FPGAs have much larger memories and the floating point support has improved, and hence maximise the performance of the conjugate gradient algorithm and MINRES respectively, by storing on dense matrices using the on-chip RAM. With dense matrices, there is no need for matching vector elements, and hence matrix-vector multiplication can be achieved easily using a pipelined dot-product core consisting of a vector multiplier and adder-tree, as shown in Figure 1. In both works, this proved to be the major performance increase, with the latter reporting up to 53 sustained single precision GFLOPs, which could translate to approximately a factor of 10 performance increase over the peak theoretical performance of a Pentium 4, for matrices of orders up to 145.

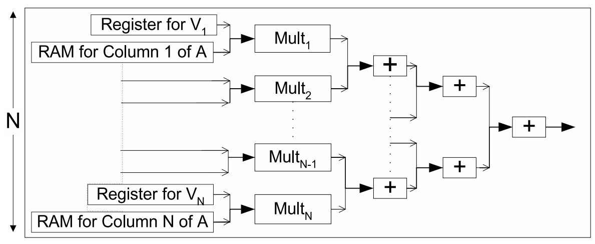


Fig. 1: Dot Product Circuit.

The work by Lopes *et al.* was then extended for banded matrices to examine RAM savings [13]; this allowed the maximum order to be extended from 92 in the dense case to 236 in the banded case for a thin band size of 5. However this work only implemented a basic architecture which performs parallel multiplication for the size of the band, but stores the entire vector in registers meaning that resource use still grows with matrix order, an approach which will be shown to be inefficient.

The aim of this work is to describe hardware architectures and RAMs configurations to perform the matrix-vector multiplication with the minimum hardware in such a way that they could easily be plugged into an implementation of an iterative method such as [2] or [3].

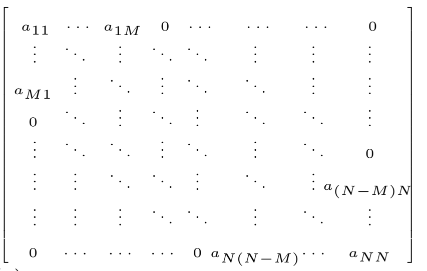
3 Performing Matrix-Vector Multiplication

This section describes simple modifications to the architecture as shown in Figure 1 to solve matrices with specific structures using a high level of parallelism. We begin with a detailed description of banded matrices before describing the hardware architectures and RAM configurations to implement matrix-vector multiplication for this type of matrix, discussing in detail how this same approach can be used to handle both thin and wide bands. We then describe how this approach can easily be extended to handle symmetric matrices, reducing the RAM requirements, before discussing our procedure to optimise the use of RAM and LUT resources on an FPGA given this RAM requirement. Finally, we discuss our approach to trade parallelism for scalability for larger matrices.

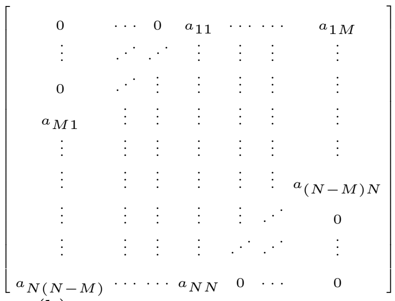
3.1 Matrix-Vector Multiplication for Banded Matrices

Banded matrices are matrices where all the non-zero elements lie within some known bandsize M from the main diagonal, as shown in Figure 2(a). As the location of the non-zeros is known *a priori*, simple structures can be used to hold these values such as Compressed Diagonal Storage (CDS) [11], shown in Figure 2(b).

Using CDS to store the matrix, all zeros that do not fall into the band are not stored. This corresponds to $(N-M)(N-M+1)$ saved elements. However, as is clear from Figure 2(b), there are still some zeros in this storage. These zeros do not reflect any in the original matrix, rather they reflect the fact that at the band ends there are no elements and hence zeros are added instead. This corresponds to a total of $M(M+1)$ additional zeros. This implies that if $2M-1 > N$, the amount of added zeros created from this redundancy could be greater than the number of zeros that are avoided by using this storage format. This section discusses these cases separately.



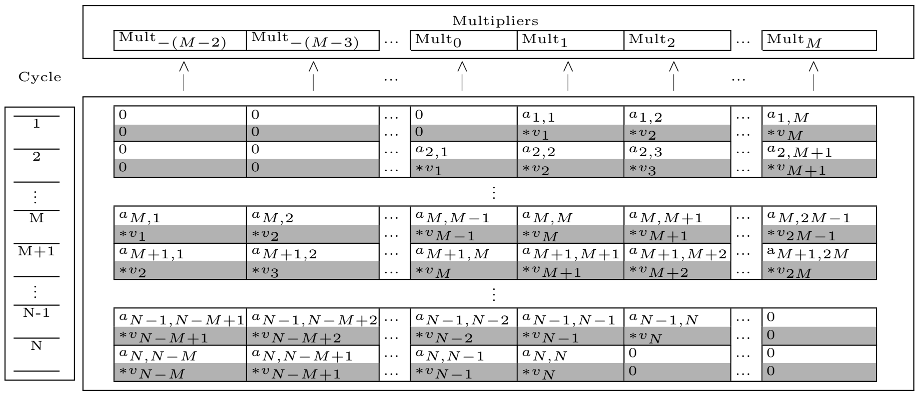
(a) Banded Matrix using Traditional Storage.



(b) Compressed Diagonal Storage.

Fig. 2: Methods to Store a Banded Matrix. Each column will be stored in a separate RAM, as shown in Figures 1 and 5.

Thin-bands ($2M-1 \le N$) In comparison to the method for dense matrices, the first difference is that instead of using N parallel multipliers, it is only necessary to perform parallel multiplications for the bandsize ($2M-1$), as the result of any other multiplications would be zero. The other slight complexity is that if the matrix is stored using CDS, the associated vector element for each RAM will change at each cycle. This is demonstrated in Figure 3 which shows the desired multiplications over time.



Multipliers
Cycle

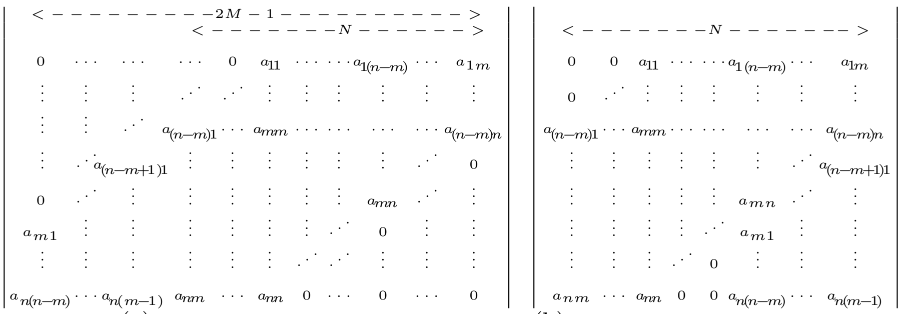
Cycle	$\text{Mult}_{-(M-2)}$	$\text{Mult}_{-(M-3)}$	…	Mult_0	Mult_1	Mult_2	…	Mult_M
1	0	0	…	0	$a_{1,1}$	$a_{1,2}$	…	$a_{1,M}$
	0	0	…	0	$*v_1$	$*v_2$	…	$*v_M$
2	0	0	…	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$	…	$a_{2,M+1}$
	0	0	…	$*v_1$	$*v_2$	$*v_3$	…	$*v_{M+1}$
⋮				⋮				
M	$a_{M,1}$	$a_{M,2}$	…	$a_{M,M-1}$	$a_{M,M}$	$a_{M,M+1}$	…	$a_{M,2M-1}$
	$*v_1$	$*v_2$	…	$*v_{M-1}$	$*v_M$	$*v_{M+1}$	…	$*v_{2M-1}$
M+1	$a_{M+1,1}$	$a_{M+1,2}$	…	$a_{M+1,M}$	$a_{M+1,M+1}$	$a_{M+1,M+2}$	…	$a_{M+1,2M}$
	$*v_2$	$*v_3$	…	$*v_M$	$*v_{M+1}$	$*v_{M+2}$	…	$*v_{2M}$
⋮				⋮				
N-1	$a_{N-1,N-M+1}$	$a_{N-1,N-M+2}$	…	$a_{N-1,N-2}$	$a_{N-1,N-1}$	$a_{N-1,N}$	…	0
	$*v_{N-M+1}$	$*v_{N-M+2}$	…	$*v_{N-2}$	$*v_{N-1}$	$*v_N$	…	0
N	$a_{N,N-M}$	$a_{N,N-M+1}$	…	$a_{N,N-1}$	$a_{N,N}$	0	…	0
	$*v_{N-M}$	$*v_{N-M+1}$	…	$*v_{N-1}$	$*v_N$	0	…	0

Fig. 3: Required Multiplication over time. In this figure, the values in grey represent the required vector elements, whilst the values in white represent the required matrix elements from RAM. Any 0 value refers to a multiplication that need not be performed.

However, from Figure 3, it should be clear that the required vector element for each multiplier is simply shifted once per clock cycle. This would require little additional hardware in comparison to Figure 1, which uses a vector of registers, as the shift could be achieved using a serial-in-parallel-out shift-register. Furthermore, this shift register need only be of size $2M-1$, as opposed to a vector of N registers.

Wide-bands ($2M-1 > N$) There are two issues when using wide bands. The first is the excessive storage, as mentioned above, the other is that when using a banded matrix, the number of parallel multiplication is equal to $2M-1$, but if $2M - 1 > N$, this would mean the number of multipliers is greater than the size of the vector, and hence any such multiplications would correspond to a multiplication by zero.

As a result, in order to minimise resources, the number of parallel multipliers should be restricted to N. To map this to the RAMs, the proposed solution to 'wrap' the data in the RAM around N columns, as shown in Figure 4. The vector can also easily be 'wrapped' by feeding the output of the final output of the shift register back into the input, and adding a multiplexer to choose between this input and the vector input, this is shown in Figure 5. The control for this multiplexer is simple and requires little additional hardware.



(a) Matrix with Wide Band. (b) Matrix with Wrapped Wide Band.

Fig. 4: Wrapped Wide Bands.

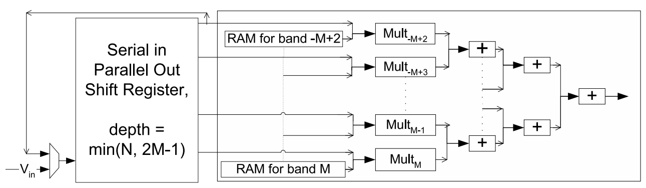


Fig. 5: Banded Dot Product Circuits.

Whilst using N columns of RAM to store the matrix appears to be no better than a dense implementation, there are two main benefits to this wrapping approach. The first is that it allows the same hardware to be used for both cases; the second is that, excluding the dense case, using the optimisation process described in Section 3.3, it is possible to save some RAM.

3.2 Matrix-Vector Multiplication for Symmetric Matrices

With symmetric matrices, it is only necessary to either store the lower or upper diagonal matrix. Interestingly, extending CDS (Figure 2(b)) to only store the symmetric portion is straightforward: all that needs to be done is to remove the columns that only hold the redundant data, *i.e.* all the columns to the left of that holding the diagonal. However, whilst this reduces the RAM requirements, in order to use the same architecture (Figure 5), one must emulate the behaviour of the extra RAMs used for band storage. Interestingly, the organisation of the RAMs in CDS makes it quite simple to achieve this. Observing Figure 2(b), the values that have been removed in the symmetric storage are simply seen to be a delayed version of other columns, the required delays shown in Figure 6.

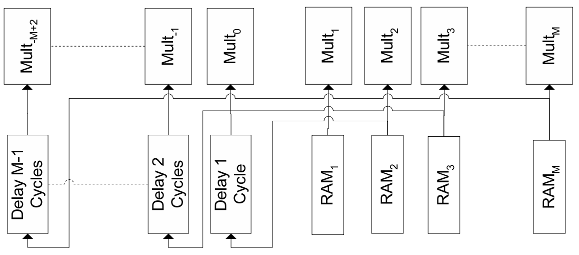


Fig. 6: Symmetric Shift Register.

Implementing delays for symmetry on FPGAs Using FPGAs, there are three potential methods to create this delay. The simple method would be to use FIFOs made up of either shift registers or RAMs. The problem with using this method is that if the delay is large, these FIFOs may also become large and this may use a lot of resources.

Alternatively, some FPGAs have embedded RAMs which can implement true dual-port memory [14]. In this case, one port could access the current value, and the other port select the delayed value, meaning the delay could then be implemented simply by using a delayed counter which would require minimal additional circuitry. However, there are more subtle issues when using embedded RAMs. Xilinx BRAMs on a Virtex 5 are 36KBit, and can be configured in one of two ways: as 2-18KBit Block RAMs implementing simple dual-port RAM; or one single 36KBit true dual-port RAM [14]. This implies that by using the Block RAMs in true dual-port fashion, the amount of flexibility of the RAMs is reduced. Viewing this in another way, the likelihood of a large portion of an embedded RAM being empty is heavily increased, and this can reduce the number of RAMs available and impact the potential parallelism.

The final choice is that if the delay required for symmetry is greater than the size of RAM needed to store a column, then the same RAM can be used to also feed the multiplier for the symmetrical delay without requiring a second port, as only one of the two multipliers will require input data at any given time. Given these three options, determining the optimum use of resources is described in the following section.

3.3 Minimising RAM use

The amount of RAM required to store the matrix is dependent upon the the size of matrix N, the bandsize M, and the number of LUTs on the FPGA the user is willing to allow to be used in the place of embedded RAMs. In order to optimise the configuration, this section proposes an integer linear programming (ILP) formulation, which can be solved by many existing solvers, such as CPLEX [15]. A high-level description of this ILP is given in Figure 7(a), with the formal ILP problem given in Figure 7(b). This section discusses how this formal ILP is obtained.

Notation As shown in Figure 2(b), the matrix columns in CDS contain trailing zeros. It is not necessary to store them, and hence the RAM requirement for each column decreases. In contrast, as shown in Figure 6, the symmetric delay

Given input matrices of order N and bandsize M for P problems,
min: *Overall RAM Use*

subject to:
Matrix Memory Constraints
Symmetric Delay Constraints

Available Register Constraints

(a) ILP Problem General Description.

$$\textbf{min:} \sum_{i=1}^{M} 2x_i + y_{1i} + y_{2i}$$

subject to:

$$\forall i, 2Bx_i + By_{1i} + z_{1i} \geq r_i$$
$$\forall i, 2Bx_i + By_{2i} + z_{2i} \geq s_i$$
$$\sum_{i=1}^{M} z_{1i} + \sum_{i=1}^{M} z_{2i} \leq R$$
$$\forall i, x_i, y_{1i}, y_{2i}, z_{1i}, z_{2i} \in \mathbb{Z}$$

(b) Formal ILP Problem.

Fig. 7: Minimising RAM using ILP.

required increases for each column. As each column has different requirements, the variable i is used as an index for the M columns, with $i = 1$ being the column containing the diagonals. For the ILP, the values for the RAMs required and symmetric delay required for column i can then be denoted as r_i, and s_i respectively. The maximum capacity of the BRAMs in terms of the number of words they can store is denoted as B, and the number of registers allocated by the user as an alternative to embedded RAMs as R; the choice of R will typically be the number of unused registers, and hence will vary depending upon the type of FPGA and the number of resources used for other functions in a given hardware implementation.

There are three choices that to store matrix elements and to implement the delays: true dual-port RAM, simple dual-port RAM or shift-registers, and as described in Section 3.2, true dual-port RAM can both store matrix elements and implement symmetric delay, whereas separate simple dual-port RAM or shift-registers are needed to implement this delay. To simplify the notation, for each column i, integer variables which represent the number of true dual-port RAMs, simple dual-port RAMs, simple dual-port RAMs used for delay for symmetry, shift-registers and shift-registers used for delay for symmetry, are denoted as $x_i, y_{1i}, y_{2i}, z_{1i}, z_{2i}$ respectively. As the RAMs and shift registers are physical components, these must be integer variables, making this an ILP.

Objective function The aim is to minimise the RAM use, so the objective function is a summation of the variables for the various RAMs. However, as the true dual-port RAMs are twice the size of the simple dual-port RAMs, the cost for all x_i variables is twice that of y_{1i} and y_{2i}.

Matrix Memory and Symmetric Delay Constraints The three types of storage must satisfy the matrix memory and symmetric delay constraints for each column. The complexity in this approach is that, as mentioned in Section 3.2, the true-dual port rams contribute to both the symmetric delay and matrix memory constraints, and thus this same variable appears in both inequalities.

It should be noted that the values r_i and s_i must be calculated prior to implementing the ILP. As i increases, the RAM requirement for each column of a matrix incrementally decreases, and hence the memory requirement for r_i is given by $(N-i+1)$. Furthermore, in previous works [2,3,6] it has been highlighted that due to the deep pipelines in floating point operators on FPGAs, in order to maintain high sustained performance it was necessary to perform matrix-vector multiplication on many different problems in a pipeline, and each of these problems would have to be stored in RAM. This can easily be incorporated into the model by modifying the memory requirement for P problems to be $P(N-i+1)$. In contrast, as i increases, the symmetric delay requirement for each column increases incrementally, but for delays, it is no longer necessary to store multiple problems, and hence s_i can be found to be $i-1$. Also, as mentioned in Section 3.2, if $i > N$, then there is no need for a separate RAM to implement the extra delay, and hence s_i is given by $i-1$ if $i \leq N$ and 0 if $i > N$.

Finally, one should note that by replacing symmetric delay constraints with extra memory constraints, it is possible to use this same ILP for banded matrices.

Available Register Constraints A final constraint is added to allow a user to trade BRAM with registers. This is likely to be problem dependent, determined by the FPGA resources used elsewhere and the total available resources.

3.4 Trading Performance with Slices

It will be shown in Section 4 that the significant reduction in memory use that this method provides implies that it would no longer be the memory available that limits the maximum matrix order of the matrix-vector multiplication, rather the slices and multipliers become the limiting factor.

The reason for the growth in slices is that the number of floating point units required to perform the parallel multiplication grows according to $\min(N, 2M-1)$. The solution would be to perform partial multiplications of size $\lceil \min(N, 2M-1)/\alpha \rceil$, where α is an integer, and use a reduction circuit, several of which are discussed in [12], to sum these partial multiplications. Any problems caused by the ceiling function are avoided by extra multiplications by zero. As it is desirable to achieve as much parallelism as possible, α should be as small as possible. The circuit required for the case of $\alpha = 2$, which would perform half the multiplication of the first half of the row during odd cycle and the second during even cycles, is shown in Figure 8. For different choices of α, only the reduction circuit would change, many of which are discussed in [12].

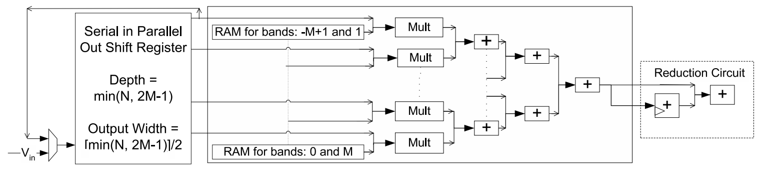

Serial in Parallel Out Shift Register
Depth = min(N, 2M-1)
Output Width = ⌈min(N, 2M-1)⌉/2
V_in
RAM for bands: -M+1 and 1
RAM for bands: 0 and M
Mult
Mult
Mult
Mult
+
+
+
+
+
Reduction Circuit
+
+

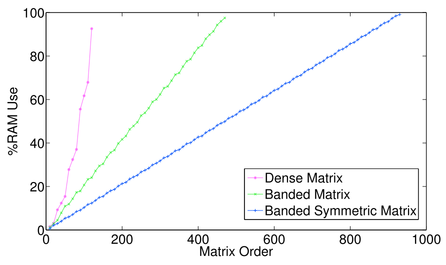
Fig. 8: Example Parallel Circuit for Large Dense Matrices.

The problem with this circuit is that parallelism is reduced by approximately a factor of α and hence the maximum performance of the circuit will similarly decrease. The counter side is that the number of slices will decrease by approximately the same factor as the number of multipliers and size of the reduction tree is reduced. This method could therefore be used for any general circuit to trade resources in terms of DSPs and registers for scalability.

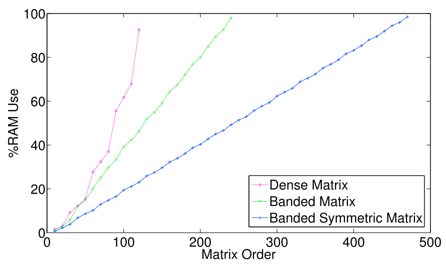
4 Results

4.1 RAM Use

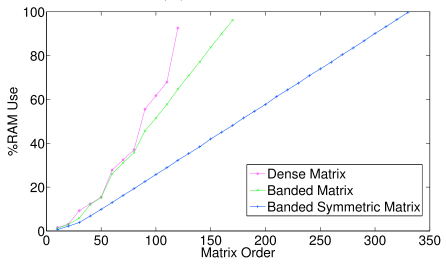
The main benefit of this work is that it significantly reduces the RAM use. Figure 9 consists of four graphs showing the percentage of embedded RAMs of a Virtex 5 LX330T that are required to hold banded matrices of warying widths. In these examples, the number of pipelined problems has been set to $P = 20$, whilst the number of registers that can be used instead of RAMs has been set to be equal to the size of one simple dual-port RAM, which corresponds to approximately 0.5% of the slices of the FPGA.



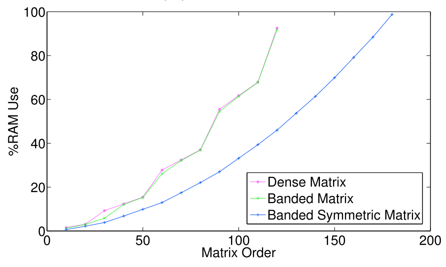
(a) Band=20.



(b) Band=40.



(c) Band=60.



(d) Dense.

Fig. 9: RAM Use.

The greater scalability of this approach is clearly shown for the thinnest bandsize, when $M = 20$, which demonstrates that a large amount resources can be saved in comparison to the basic method storing a dense matrix, the maximum matrix order can be extended from 120 to 470 in the banded case and 930 in the symmetric banded case. It should be noted that this bandwidth would, at the maximum, require 39 parallel floating point multiplications, and hence could not be fed using off-chip RAM. As the bandsize increases, though this difference gets smaller, it is still significant. However, it is interesting to note that the difference between the dense and banded case decreases much faster than the difference between the dense and the symmetric case. The reason for this is that the symmetric delay is only a function of N, whereas storing the band instead of implementing this delay is a function of N and P. It is also worth noting that in the graph where there is a wide band of $M = 60$, there is indeed still RAM savings using the banded format as opposed to storing it in the dense format, as mentioned in section 3.1.

4.2 Parallelism

The other claims of this work are that by reducing the amount of RAM used, it is possible to obtain greater parallelism, and this parallelism becomes limited by the registers and DSPs and hence it becomes necessary to trade parallelism for scalability. In order to demonstrate this, Figure 10 compares the resource use, post place and route, of matrix vector multiplication for a dense matrix of increasing order, using the architecture from Figure 1 that was used in previous works and this approach. The number of problems has been set to $P = 14$ for this is what was used for the largest matrix in [3], the previous work which claims the highest performance.

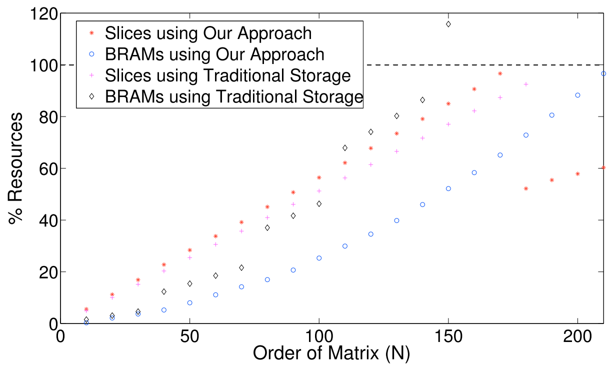


Fig. 10: Resource Use

This graph demonstrates several points of interest. Firstly it shows that this work approaches the maximum performance achievable on an FPGA for matrix-vector multiplication in IEEE single precision floating point, in that it uses almost all of the slices and DSPs, with most of these being used as floating point components. This is unlike the method for traditional storage, in which case the RAM limits the maximum possible parallelism; it is clear that the RAM required for traditional storage exceeds that available for a matrix order of 150. It is also interesting that the transition for RAMs is much smoother for our method than for using traditional storage; this is a result of the fact our approach reduces the probability RAMs are empty, as mentioned in Section 3.2. In addition, this graph demonstrates the value of reducing the parallelism to increase the scalability. Finally, one should note that whilst there is still limited scalability, this is due to the focus on a dense matrix with a large number of pipelined problems. However, this is nonetheless a valuable test as this component could easily be plugged into the iterative solvers in [2, 3] to improve their performance and scalability.

5 Conclusion

Overall, this work has described how to create a parameterisable circuit to implement matrix-vector multiplication that could be plugged into existing hardware implementations of iterative methods. Furthermore, it has shown that by taking into account symmetry and banded matrices, only simple hardware changes to an implementation of matrix-vector multiplication circuit using a pipelined dot-product circuit, along with an optimisation strategy for RAM use, are required to significantly improve both the scalability and performance of the circuit.

Finally, one should note that any algorithm containing matrix-vector multiplication which is suitable for on-chip buffering of data could use this circuit, whilst the contributions to reduce RAM use on FPGAs could be applied to any circuit that stores banded or symmetric matrices on-chip.

References

1. G. R. Morris, V. K. Prasanna, and R. D. Anderson, "A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer," in *Proc. 14th IEEE Symp. Field-Programmable Custom Computing Machines*, 2006, pp. 3–12.
2. A. R. Lopes and G. A. Constantinides, "A high throughput FPGA-based floating point conjugate gradient implementation," in *Proc. Applied Reconfigurable Recomputing*, 2008, pp. 75–86.
3. D. Boland and G. Constantinides, "An FPGA-based implementation of the MINRES algorithm," *Field Programmable Logic and Applications, 2008. FPL 2008. International Conference on*, pp. 379–384, Sept. 2008.
4. L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proc. ACM/SIGDA 13th Int. Symp. on Field-Programmable Gate Arrays.* New York, NY, USA: ACM, 2005, pp. 63–74.
5. Y. El-Kurdi, W. J. Gross, and D. Giannacopoulos, "Sparse matrix-vector multiplication for finite element method matrices on FPGAs," *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, vol. 0, pp. 293–294, 2006.
6. M. deLorimier and A. DeHon, "Floating-point sparse matrix-vector multiply for FPGAs," in *Proc. ACM/SIGDA 13th Int. Symp. on Field-Programmable Gate Arrays.* New York, NY, USA: ACM, 2005, pp. 75–85.
7. M. T. Heath, *Scientific Computing.* McGraw-Hill Higher Education, 2001.
8. G. H. Golub and C. F. V. Loan, *Matrix computations (3rd ed.).* Baltimore, MD, USA: Johns Hopkins University Press, 1996.
9. A. G. Hoekstra, P. Sloot, W. Hoffmann, and L. Hertzberger, "Time complexity of a parallel conjugate gradient solver for light scattering simulations: Theory and spmd implementation," Tech. Rep., 1992.
10. G. Sewell, *The numerical solution of ordinary and partial differential equations.* San Diego, CA, USA: Academic Press Professional, Inc., 1988.
11. R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition.* Philadelphia, PA: SIAM, 1994.
12. L. Zhuo, G. R. Morris, and V. K. Prasanna, "High-performance reduction circuits using deeply pipelined operators on FPGAs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 18, no. 10, pp. 1377–1392, 2007.
13. A. Lopes, G. Constantinides, and E. C. Kerrigan, "A floating-point solver for band structured linear equations," *Proc. Int. Conf Field Programmable Technology*, pp. 353–356, 2008.
14. Xilinx, *Virtex-5 FPGA User Guide.*
15. Ilog, Inc., "Solver cplex," 2009, http://www.ilog.fr/products/cplex/ (accessed 02 November 2009). [Online]. Available: http://www.ilog.fr/products/cplex/