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

David Boland and George A. Constantinides
Imperial College London

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 [Morris et al. 2006; Zhang et al. 2008; Zhuo and Prasanna 2006]. 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) [Lopes and Constantinides 2008]. 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 [Zhuo and Prasanna 2005; El-Kurdi et al. 2006]. However, whilst the performance of matrix-vector multiplication on FPGAs is generally I/O bound [Zhuo and Prasanna 2005], the nature of iterative methods allows the use of on-chip memory buffers to increase the bandwidth, providing the potential for significantly more parallelism [deLorimier and DeHon 2005]. Unfortunately, existing approaches have generally only either been capable of solving large matrices with limited improvement over GPPs [Zhuo and Prasanna 2005; El-Kurdi et al. 2006; deLorimier and DeHon 2005], or achieve high performance for relatively small matrices [Lopes and Constantinides 2008; Boland and Constantinides 2008]. 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.

Categories and Subject Descriptors: [**Design methodologies**]: ; [**Optimisation**]:

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Iterative Methods, Integer Linear Programming

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 [Heath 2001], 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 recent studies have shown that by accelerating them using FPGAs, it is possible to achieve performance gains of up to an order of magnitude over general purpose processors [Lopes and Constantinides 2008; Boland and Constantinides 2008].

D. Boland, Electrical and Electronic Engineering Department, Imperial College London,London SW7 2AZ,UK.

The reason FPGAs are capable of accelerating iterative methods, such as the conjugate gradient (CG) and minimum residual (MINRES) algorithms [Golub and Loan 1996], 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 [Hoekstra et al. 1992], 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 take advantage of banded matrix structure, matrix symmetry, 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 [Sewell 1988]. Symmetric matrices are square matrices equal to their own transpose, and these are of particular interest as both CG and MINRES algorithms will only converge to a solution provided the input matrix is symmetric.

This paper builds upon previous work which showed that by exploiting these properties, which reduce the RAM requirements, by using an optimisation strategy based on integer linear programming (ILP) that considers the specific structures of embedded RAMs on FPGAs, it is possible to reduce the resource usage [Boland and Constantinides 2010]. In this extended version, we add further simple architectures which provide greater flexibility to implement matrix-vector multiplication and allow a user greater control over the trade-off between parallelism and resources. Furthermore, we incorporate models of these structures into the same ILP framework, enabling us to determine the maximum parallelism given resource constraints. The main contributions of this paper are 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 and maximise the parallelism, according to 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.* [2006]. 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 to a 1.76 single precision GFLOPs peak on a Stratix S80 or a sustained 1.5 single precision GFLOPs.

The work by Morris *et al.* [2006] is slightly different, storing the matrix in a more traditional fashion, using Compressed Sparse Row (CSR) format [Barrett et al. 1994], which consists of all non-zero values of the matrix, an index to the column each 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.* [2005; 2007] 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 [2005] 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, and showed it is possible to improve the performance by using multiple FPGAs. However, it is also reported that this performance drops as communication time eventually outweighs computation time as the number of FPGAs increases, limiting the peak performance to 12 GFLOPs on 16 FPGAs.

The work by Lopes *et al.* [2008] and previous work by the authors [2008] looked to avoid these performance limitations by acknowledging that modern FPGAs have larger memories and the floating point support has improved, and hence focus on maximising the performance of the conjugate gradient algorithm and MINRES respectively, by storing dense matrices using the on-chip RAM on a single FPGA. 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, the former reporting up to 35 single precision GFLOPs for matrices of orders up to 58, the latter 53 single precision GFLOPs for matrices of orders up to 145.

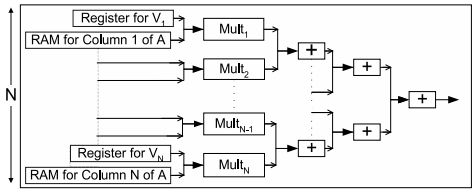


Fig. 1: Dot Product Circuit.

The work by Lopes *et al.* was extended for banded matrices to examine RAM savings [2008]; 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 that they could easily be plugged into an implementation of an iterative method such as [Lopes and Constantinides 2008] or [Boland and Constantinides 2008], which can achieve similar high levels of performance to these works whilst scaling to larger order matrices.

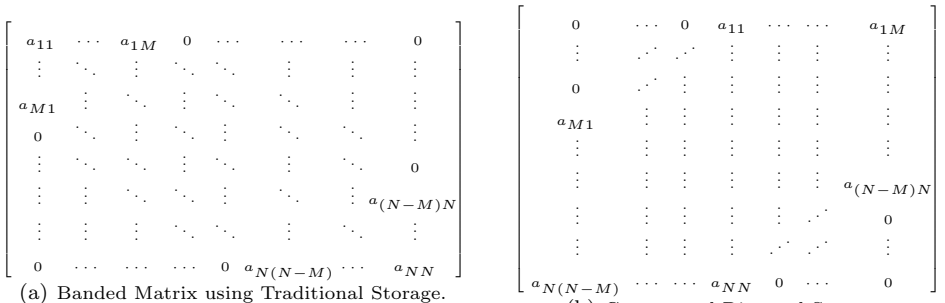
3. PERFORMING MATRIX-VECTOR MULTIPLICATION

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) [Barrett et al. 1994], shown in Figure 2(b). This is preferable to other storage schemes such as compressed sparse row [Barrett et al. 1994] because these schemes store indices of the locations of the non-zero elements, which is a waste of RAM use as this redundant information.

Using CDS to store the matrix, all zeros that do not fall into the band are not stored, saving $(N-M)(N-M+1)$ 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. These total $M(M+1)$ 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.

3.1.1 *Thin-bands ($2M-1 \leq N$).* 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.

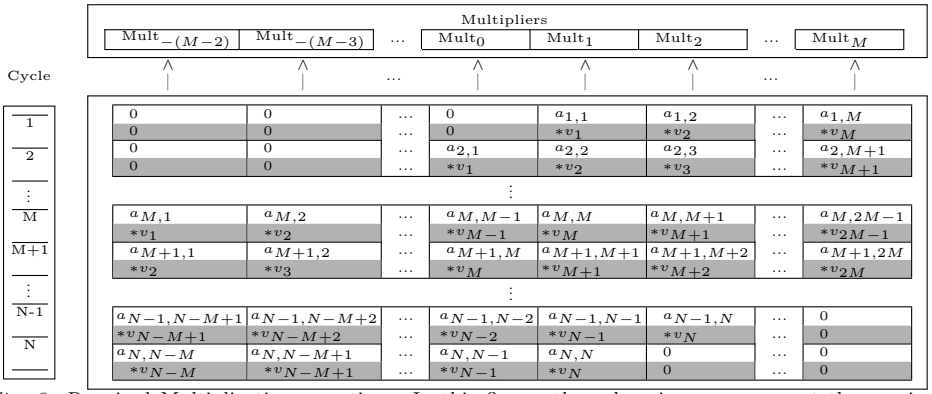
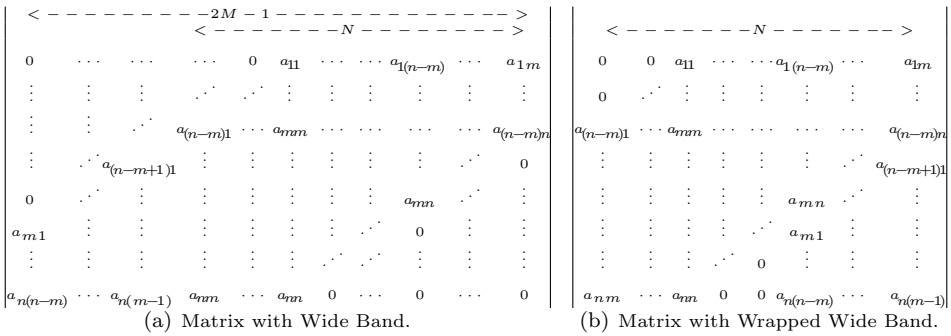


Fig. 3: Required Multiplication over time. In this figure, the values in grey represent the required vector elements, whilst the values in white represent the required matrix elements from RAM. Any 0 value refers to a multiplication that need not be performed.

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.

3.1.2 *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.

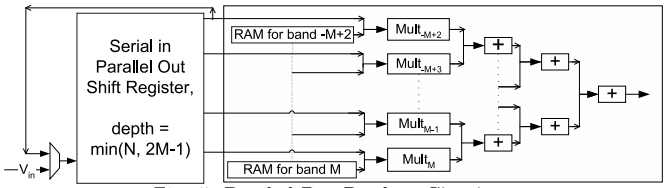


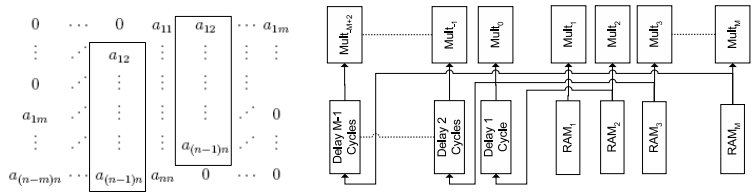
Fig. 5: Banded Dot Product Circuits.

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 3.4, 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, we would like to use the same architecture (Figure 5) because we believe it to have minimal control and allowing a high clock frequency, since it consists of only a shift register, RAMs and a reduction tree, and also to allow any architectures which improve these components of our circuit to be easily incorporated into our design. However, in order to achieve this, 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: as highlighted in Figure 6(a), the values to

the left of the diagonal can be seen as a delayed version of other columns, meaning these columns can be emulated using the required delays shown in Figure 6(b).



(a) Symmetric Matrix. (b) Symmetric Shift Register.

Fig. 6: Using Delays to Emulate Symmetry.

3.2.1 *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. 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. As it is not possible to describe an optimisation strategy for all FPGA architectures, in this work we wish to demonstrate how one could make use of this additional functionality within the same optimisation framework, and we have chosen to use Virtex 5 family as a case study for this aspect; we believe that our framework could easily be modified when targeting a different architecture. However, there are more subtle issues when using Xilinx 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 [Xilinx 2010]. 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 Trading Performance with Slices

When implementing a matrix-vector multiplication circuit, ideally, the amount of parallelism should be limited by the resource constraints, which in turn will be dependent upon the chosen FPGA device and any other operations. In this section, we discuss two simple methods to trade the level of parallelism in order to make

best use of available resources: increase the parallelism by performing parallel dot products or reduce the resource usage by performing dot products in stages.

3.3.1 *Performing parallel dot products.* In matrix-vector multiplication, it is possible to perform $\alpha \leq N$ parallel dot products for the multiple rows of the matrix. Whilst creating α parallel vectors of multipliers and adder reduction trees is a trivial task, the memory storage is not. The reason for this is that whilst it is necessary to have RAMs feeding each dot product circuit, it is undesirable to simply replicate the memory structure, for this would store a large amount of redundant data. Instead it is best to re-organise the data so that the RAMs only store the data for the relevant dot product circuit. One simple method to do this would be to split a matrix vertically into α subsections and repeat the circuit. However, this is undesirable for two reasons: firstly, the delay circuitry for symmetry will then be repeated α times, which would waste hardware; secondly, to perform all operations in parallel, it would be necessary to have the entire vector available, which means that for a very large matrix with a thin band ($N \gg M$), there would be little gain compared to the approach specified in Figure 5 which begins performing dot products once the first band of the vector is available. Instead we wish to compute parallel dot products for α consecutive rows. We also note that this parallelism reduces the size of each RAM by α by spreading the RAM across several circuits. However, by doing this, we create dependencies between the various dot product circuits when implementing the symmetric delay; some example circuits which demonstrate how this is achieved are shown in Figures 7 and 8.

In order to understand how wiring and delay circuits are created, let us first label i as the circuit index, where $0 \leq i \leq \alpha - 1$, and j as the RAM column index, where $-M+2 \leq j \leq M$. Now firstly, it should be clear from Figure 6(b) that because the RAM storing the diagonal ($j = 1$) is not re-used for symmetry, the first symmetric delay column ($j = 0$) will always comes from the second RAM column ($j = 2$), with a delay of one cycle, and the second symmetric delay column ($j = -1$) will always comes from the third RAM column ($j = 3$), with a delay of two cycles. We can extend this for all the symmetric delay columns, which are in the range ($-M + 2 \leq j \leq 0$), to say they will delay data from the RAM column $2 - j$, with a delay of $1 - j$ cycles. Secondly, we must consider that the idea of the symmetric delay columns in Figure 6(b) is to store the data used in a previous multiplication, but when multiplying consecutive rows in different circuits in parallel, storing data from a previous multiplication corresponds to taking the data directly from the RAM of a previous circuit, and a delay of one cycle now corresponds to data from α multiplications back. This means that for a given circuit i, the first symmetric delay column ($j = 0$) can be taken directly from circuit $i - 1$ and the second symmetric delay column ($j = -1$) can be taken directly from circuit $i - 2$, whilst the symmetric delay column $j = -\alpha$ can be taken from circuit i with a delay of one cycle and the symmetric delay column $j = -\alpha - 1$ can be taken from circuit $i - 1$ with a delay of one cycle. In general, because the symmetric delay column j of any circuit i, as stated earlier, requires data from $1 - j$ previous multiplications, the data for this column will come from circuit $i - (1 - j) \mod \alpha$, with a delay of $\lceil (1 - j)/\alpha \rceil$ cycles.

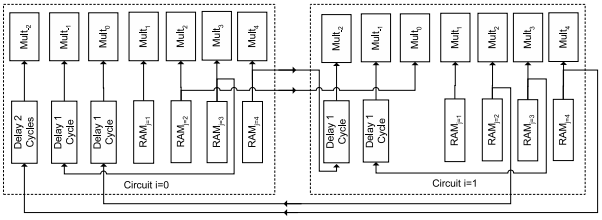


Fig. 7: $\alpha = 2$ Parallel Dot Product Circuits of Bandsize 4.

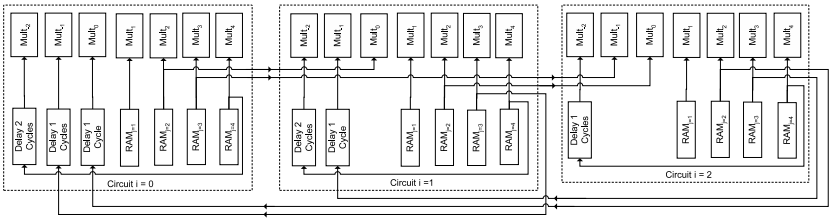


Fig. 8: $\alpha = 3$ Parallel Dot Product Circuits of Bandsize 4.

3.3.2 *Performing dot products in stages.* When creating a dot product circuit, the number of parallel multiplications grow according to $\min(N, 2M-1)$ and the number of adders in the reduction tree grows according to $\min(N, 2M-1)-1$. This means that depending upon the available hardware, it may be necessary to restrict the parallelism to save slices, and we note that the significant reduction in memory use that this optimisation strategy 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 can easily become the limiting factor. We can reduce the amount of multipliers and adders used by performing partial multiplications of a row of size $\lceil \min(N, 2M-1)/\beta \rceil$, where β is an integer, and use a reduction circuit, several of which are discussed in [Zhuo et al. 2007], to sum these partial multiplications. We note that by doing this, we reduce the number of required RAMs by β, but increase their size by β. The circuit required to achieve this is relatively simple, with the inputs for each multiplier controlled by a multiplexer. As an example, the circuit required for the case of $\beta = 2$, 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 9. For different choices of β, only the reduction circuit would change, many of which are discussed in [Zhuo et al. 2007].

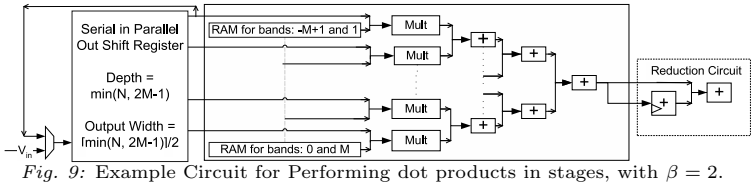


Fig. 9: Example Circuit for Performing dot products in stages, with $\beta = 2$.

3.3.3 *Performing dot products in stages in parallel.* Finally, it is worth noting that to obtain greater control over the trade off between parallelism and resources, it is possible to combine both of these methods. The reason this could be important can be demonstrated with a simple example. Suppose one was trying to perform dense matrix-vector multiplication for a matrix of order $N = 120$, but the number

of slices limited the maximum number of floating point multipliers and adders each to 80. Obviously, one could not perform full dot product, as there is insufficient resources. Instead, it is possible to perform a partial dot-product by choosing $\beta = 2$, which would use 60 multipliers and 59 adders. This would work, but it fails to make the best use of resources, for alternatively we could perform $\alpha = 2$ dot-products in $\beta = 3$ stages in parallel, each partial dot-product circuit would use 40 multipliers and 39 adders, and an extra adder would be needed for the reduction circuit, using almost all the available resources.

3.4 Maximising Matrix-Vector Performance

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 slices on the FPGA the user is willing to allow to be used in the place of embedded RAMs, whilst the amount of parallelism is also restricted by the number of LUTs and DSPs (or embedded multipliers) on the FPGA the user is willing to allow to be used to create the circuit, as well as the number of required RAMs to feed the circuit. In our circuits, we have implemented single precision floating point operators using Xilinx Coregen [Xilinx 2010], with each multiplier using a single DSP. 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 [Ilog, Inc. 2009]. The ILP is shown in Figure 10, this section discusses how this formal ILP is obtained.

Given an input matrix of order N and bandsize M for P problems, and user input constants X, Y, Z (defined in 3.4.1)

$$\textbf{max:} \quad Z\alpha - (1-Z)\sum_{i=1}^{MY}(2\rho_i + \sigma_{1i} + \sigma_{2i}) \qquad \textit{(Maximise Performance)}$$

subject to:

$$\forall i \forall j, 2B\rho_{ij} + B\sigma_{1ij} + \tau_{1ij} + L\sum_{k=1}^{j}(\nu_j - 1) \geq Y\beta P(N - i + 1)/X \qquad \textit{(Matrix Memory Constraints)}$$

$$\forall i \in \{1,...,N\} \forall j, 2B\rho_{ij} + B\sigma_{2ij} + \tau_{2ij} + L\sum_{k=1}^{j}(\nu_j - 1) \geq Y\beta(i-1)/X \qquad \textit{(Symmetric Delay Constraints)}$$

$$\sum_{i=1}^{MY}(2\rho_i + \sigma_{1i} + \sigma_{2i}) \leq R \qquad \textit{(Available RAM Constraints)}$$

$$K_1\sum_{j=1}^{\alpha}(\tau_{1ij} + \tau_{2ij}) + K_2\iota + K_3\kappa \leq S \qquad \textit{(Available Slice Constraints)}$$

$$K_4\iota + K_5\kappa \leq D \qquad \textit{(Available DSP Constraints)}$$

$$\iota - \alpha\min(N, 2M-1)/Y = -1 \qquad \textit{(Required Adder Constraint)}$$

$$\kappa - \alpha\min(N, 2M-1)/Y = 0 \qquad \textit{(Required Multiplier Constraint)}$$

$$\forall j, \alpha + L\nu_j + L\sum_{k=1;k\neq j}^{X}(1-\nu_j) \geq j, \quad \sum_{j=2}^{X}\nu_j = X - 1 \qquad \textit{(Parallelism Constraint 1)}$$

$$\forall j, \alpha + L\nu_j + L\sum_{k=1;k\neq j}^{X}(1-\nu_j) \leq j \qquad \textit{(Parallelism Constraint 2)}$$

$$\forall j, \beta + L\nu_j + L\sum_{k=1;k\neq j}^{X}(1-\nu_j) \geq X/j \qquad \textit{(Parallelism Constraint 3)}$$

$$\forall i \forall j, \rho_{ij}, \sigma_{1ij}, \sigma_{2ij}, \tau_{1ij}, \tau_{2ij} \in \mathbb{Z}, \quad \iota, \kappa \in \mathbb{Z}, \quad \alpha \in \mathbb{Z},$$
$$\forall j, \nu_j \in \{0,1\}, \quad i \in \{1,...,MY\}, \quad j \in \{1,...,X\}$$

Fig. 10: Maximising Performance using ILP.

3.4.1 *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 changes. As a result, the variable i is used as an index for the M columns, with $i = 1$ being the column containing the diagonals. Similarly, as a result of the fact that, as mentioned in Section 3.3, the depth of each RAM will also vary depending upon our choice of the size of the partial dot product and whether it

is shared across parallel circuits, the variable j as an index for each parallel circuit. To obtain control over the size of the ILP, we also add input constants X and Y for the maximum number of the parallel circuits and smallest partial dot products respectively, with the variables α and β are used to determine the chosen number of parallel circuits and the size of the required RAMs, whilst the binary variables ν_j are used conjunction with a large value L for 'big-M formulation' [Winston 2003] which represents whether a parallel circuit is used.

There are three choices 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 3.2.1, 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 column i of circuit j, we use 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 $\rho_{ij}, \sigma_{1ij}, \sigma_{2ij}, \tau_{1ij}, \tau_{2ij}$ respectively. As the RAMs and shift registers are physical components, these must be integer variables, making this an ILP.

The remaining input constants are $R, S, D, B, K_1, K_2, K_3, K_4, K_5$ and integer constraints are the variables ι, κ. The constants R, S, D represent the maximum number of BRAMs, slices and DSPs on the target device, whilst the rest are used to translate these integer variables into the actual components: the maximum capacity of the BRAMs in terms of the number of words they can store is denoted as B, K_1 represents the number of slices to create a one cycle delay, K_2, K_3 represent the required number of slices to create an adder or multiplier and K_4, K_5 representing the required number of DSPs for these components. If it is possible to create several different adders and multipliers which trade use of DSPs and slices, extra variables for could be added to ι and κ, as well as their associated constants.

3.4.2 *Objective function.* The aim of the ILP is to maximise the performance, and minimise the RAM use. The first objective involves creating the largest number of parallel multiplications, given by α. For the second objective, the RAM use is a summation of the variables for the various RAMs whilst noting that as the true dual-port RAMs are twice the size of the simple dual-port RAMs, the cost for all ρ_{ij} variables is twice that of σ_{1ij} and σ_{2ij}. In order to convert both goals into maximisations, the latter term is negated. Finally, as both objectives will compete for slices, multipliers and RAMs, the variable Z is added as an input variable to allow a user to favour one goal over the other.

3.4.3 *Matrix Memory and Symmetric Delay Constraints.* The three types of storage must satisfy the matrix memory and symmetric delay constraints for each column i and circuit j. The big-M formulation in this constraint ensures that if parallel circuit j is not used, there will also be no RAM constraint. The complexity in this approach is that, as mentioned in Section 3.2.1, the Xilinx true-dual port RAMs contribute to both the symmetric delay and matrix memory constraints, and thus this same variable appears in both inequalities.

The RAM requirement for each column of a matrix incrementally decreases, and hence the memory requirement could be given by $(N - i + 1)$. However, in previous works [deLorimier and DeHon 2005; Lopes and Constantinides 2008; Boland and

Constantinides 2008] 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 the symmetric delay requirement is generally given by $i-1$. However, as mentioned in Section 3.2.1, if $i > N$, then there is no need for a separate RAM to implement the extra delay, and hence the symmetric delay requirement is $i-1$ 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.

3.4.4 *Available RAM/Slice/DSP Constraints.* These constraints translate the integer variables into resource constraints which ensure the implementation will fit on an FPGA.

3.4.5 *Required Adder/Multiplier Constraints.* These constraints ensure that there are sufficient multipliers and adders to implement the relevant level of parallelism, noting that the required number of adders is one less than the number of multipliers.

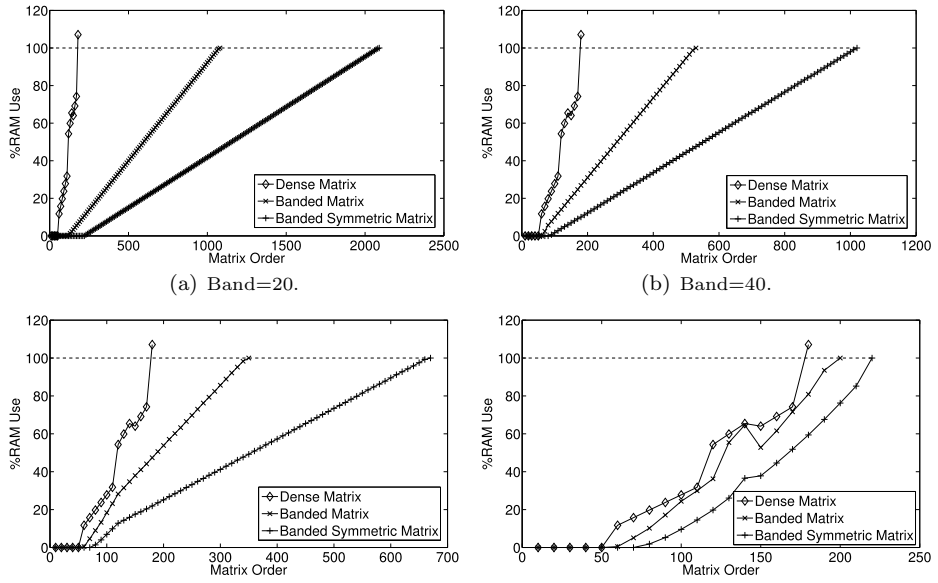
3.4.6 *Parallelism Constraints.* There is an inverse relationship between the size of the RAMs and the number of circuits, and this cannot be directly represented in an ILP. However, as we have restricted the maximum amount of parallelism and minimum dot product size, there is a distinct number of configurations and thus we can use big-M constraints to ensure that for any level of parallelism (α), the required RAM size (β) satisfies this inverse relationship.

4. RESULTS

4.1 RAM Use

The main benefit of this work is that it significantly reduces the RAM use. We demonstrate this by using our ILP to optimise the RAM use for four banded matrices of varying widths. Figure 11 consists of four graphs showing the percentage of embedded RAMs of a Virtex 5 LX330T that are required to hold these matrices using traditional dense storage, and by using band storage or symmetric band storage when optimised using our ILP. In these examples, the number of pipelined problems has been set to $P = 10$ so as to be comparable to previous works [Boland and Constantinides 2008], and we set $Z = 0$ to focus on optimising RAM use. In most of the test cases, we have set $X = 1$ and $Y = 1$ so as to perform comparisons with the traditional method of storing a dense matrix. However, for cases where it is necessary to reduce the parallelism in order to satisfy the slice constraints of the FPGA, we allow $Y = 2$. These choices also ensure a short run-time for the ILP of a few seconds. Finally, in order to perform a fair comparison with the dense implementation, we allow the slices on the FPGA that are not used for the dot-product circuit to store matrix elements, a procedure which is performed automatically in our optimisation framework for the banded and symmetric banded cases.

The greater scalability of this approach is clear for the thinnest bandsize, when



(a) Band=20. (b) Band=40.

(c) Band=60. (d) Dense.

Fig. 11: RAM Use.

$M = 20$, where the large amount resources saved in comparison to the basic method storing a dense matrix, extends the maximum matrix order from 170 to almost 1000 in the banded case and 2000 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.

It is also interesting to see 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. However, it is clear from the graphs, there are indeed still RAM savings using the banded format as opposed to storing it in the dense format, even where there is a wide band of $M = 60$, as mentioned in section 3.1.2.

Finally it is interesting to see how our approach smoothes all the transitions between the discrete RAM sizes. It is clear in the dense case that there are distinct jumps which are a result of needing a larger RAM size, which is likely to be largely empty, as mentioned in Section 3.2.1, to hold each column of the matrix. In our approach, as we optimise the use of RAMs and shift registers, these sudden jumps generally do not occur. There are however two exceptions to this, the first is for small matrices when it is possible to store the entire matrix using shift registers instead of RAMs, the second is that whenever the parallelism is decreased, there is a sudden increase in slices that could be used as shift registers. However, this second effect could easily be controlled by the choices of X and Y.

4.2 Performance

As well as RAM savings, this work can also be used to both achieve greater parallelism, and continue to achieve greater parallelism for higher order matrices. In

order to demonstrate this, we again set $P = 10$, then focus on parallelism by setting $Z = 1$, whilst to ensure a fair degree of flexibility to choose the best circuit, we set $Y = 10$ and $X = \min(N, 2M - 1)Y$, which allows for potential for a fully parallel matrix-vector multiplication, provided there are sufficient resources, whilst still maintaining the run-time of the ILP to be only a few seconds. Figure 12 compares the maximum performance achievable for a dense matrix using a traditional dot-product approach as in Figure 1, our optimisation strategy applied to a dense matrix, and our approach applied to a matrix with a wide band of 80. All designs were successfully placed and routed to a target frequency of 150 MHz, and in this figure, we have plotted the GFLOPs assuming the circuit to be fully utilised, which is possible in a good design, as shown in [Boland and Constantinides 2008].

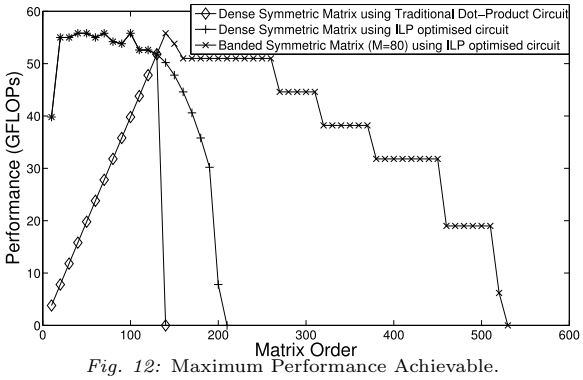


Fig. 12: Maximum Performance Achievable.

Figure 12 shows our approach achieves 55 single precision GFLOPs for small matrix orders by performing the dot product of several rows in parallel by making use of almost all the slices available on the device, with the exceptions for the matrix order of 10 where the maximum parallelism is a fully parallel matrix-vector multiplication, where the slight jumps in the graph which are due to the choices of X and Y that only allow for certain discrete levels of parallelism, and maintains this high performance for larger matrix orders by performing parallel dot products in stages, in conjunction with the RAM optimisation, before gradually dropping with larger matrix orders due to the RAM requirements allowing less room for parallelism. In comparison to previous work mentioned in the background section, the greatest performance was achieved by the MINRES solver with a traditional dot-product circuit providing the majority of the performance [Boland and Constantinides 2008], and this only achieved 53 GFLOPs when solving the largest matrix order of 145. Furthermore, Figure 12 shows that in general, any approach using a the traditional full dot-product approach can only approach the performance of our optimised circuit for the maximum matrix order of 170, and cannot scale to larger orders to achieve greater performance due to insufficient memory, shown earlier in Figure 11(d).

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, simple hardware changes to an implementation of matrix-vector multiplication circuit using a pipelined dot-product circuit, along with an optimisation strategy for both RAM use and performance, can 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

Barrett, R., Berry, M., Chan, T. F., Demmel, J., Donato, J., Dongarra, J., Eijkhout, V., Pozo, R., Romine, C., and der Vorst, H. V. 1994. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods, 2nd Edition*. SIAM, Philadelphia, PA.

Boland, D. and Constantinides, G. 2008. An FPGA-based implementation of the MINRES algorithm. *Proc. Int. Conf. Field Programmable Logic and Applications*, 379–384.

Boland, D. and Constantinides, G. 2010. Optimising memory bandwidth use for matrix-vector multiplication in iterative methods. *Proc. Int. Symp. Applied Reconfigurable Computing*, 169–181.

deLorimier, M. and DeHon, A. 2005. Floating-point sparse matrix-vector multiply for FPGAs. In *Proc. ACM/SIGDA 13th Int. Symp. on Field-Programmable Gate Arrays*. ACM, New York, NY, USA, 75–85.

El-Kurdi, Y., Gross, W. J., and Giannacopoulos, D. 2006. Sparse matrix-vector multiplication for finite element method matrices on FPGAs. *Proc. Int. Symp. Field-Programmable Custom Computing Machines*, 293–294.

Golub, G. H. and Loan, C. F. V. 1996. *Matrix computations (3rd ed.)*. Johns Hopkins University Press, Baltimore, MD, USA.

Heath, M. T. 2001. *Scientific Computing*. McGraw-Hill Higher Education.

Hoekstra, A. G., Sloot, P., Hoffmann, W., and Hertzberger, L. 1992. Time complexity of a parallel conjugate gradient solver for light scattering simulations: Theory and spmd implementation. Tech. rep.

Ilog, Inc. 2009. Solver cplex. `http://www.ilog.fr/products/cplex/` (accessed 02 November 2009).

Lopes, A., Constantinides, G., and Kerrigan, E. C. 2008. A floating-point solver for band structured linear equations. *Proc. Int. Conf Field Programmable Technology*, 353–356.

Lopes, A. R. and Constantinides, G. A. 2008. A high throughput FPGA-based floating point conjugate gradient implementation. In *Proc. Applied Reconfigurable Recomputing*. 75–86.

Morris, G. R., Prasanna, V. K., and Anderson, R. D. 2006. A hybrid approach for mapping conjugate gradient onto an FPGA-augmented reconfigurable supercomputer. In *Proc. 14th IEEE Symp. Field-Programmable Custom Computing Machines*. 3–12.

Sewell, G. 1988. *The numerical solution of ordinary and partial differential equations*. Academic Press Professional, Inc., San Diego, CA, USA.

Winston, W. L. 2003. *Introduction to Mathematical Programming: Applications and Algorithms*. Duxbury Resource Center.

Xilinx. 2010. *Virtex-5 FPGA User Guide*.

Zhang, W., Betz, V., and Rose, J. 2008. Portable and scalable FPGA-based acceleration of a direct linear system solver. *Proc. Int. Conf Field Programmable Technology*, 17 –24.

Zhuo, L., Morris, G. R., and Prasanna, V. K. 2007. High-performance reduction circuits using deeply pipelined operators on FPGAs. *IEEE Trans. Parallel Distrib. Syst. 18,* 10, 1377–1392.

Zhuo, L. and Prasanna, V. K. 2005. Sparse matrix-vector multiplication on FPGAs. In *Proc. Int. Symp. on Field-Programmable Gate Arrays*. ACM, New York, NY, USA, 63–74.

Zhuo, L. and Prasanna, V. K. 2006. High-performance and parameterized matrix factorization on fpgas. *Proc. Int. Conf. Field Programmable Logic and Applications*, 1 –6.