An Efficient Sparse Conjugate Gradient Solver Using a Beneš Permutation Network

Paul Grigoras, Gary Chow, Pavel Burovskiy and Wayne Luk
Department of Computing
Imperial College London
180 Queen's Gate, London SW7 2AZ, UK
Email: paul.grigoras09@imperial.ac.uk

***Abstract*—The conjugate gradient (CG) is one of the most widely used iterative methods for solving systems of linear equations. However, parallelizing CG for large sparse systems is difficult due to the inherent irregularity in memory access pattern. We propose a novel processor architecture for the sparse conjugate gradient method. The architecture consists of multiple processing elements and memory banks, and is able to compute efficiently both sparse matrix-vector multiplication, and other dense vector operations. A Beneš permutation network with an optimised control scheme is introduced to reduce memory bank conflicts without expensive logic. We describe a heuristics for offline scheduling, the effect of which is captured in a parametric model for estimating the performance of designs generated from our approach.**

I. Introduction

Solving systems of linear equations is an essential step in many scientific computing applications [1]. The sparsity structure of very large linear systems can often be used to reduce storage requirements. However, the irregular memory access pattern introduced by sparse storage formats can reduce computational efficiency. Methods for solving sparse linear systems include algorithms such as the conjugate gradient (CG), an instance of Krylov Subspace Iteration algorithms - cited as one of the most important classes of algorithms of the 20th century [2].

The conjugate gradient algorithm for solving sparse linear systems is difficult to accelerate on parallel platforms since *1)* the irregular memory access pattern makes the algorithm difficult to parallelise and *2)* the algorithm requires a mixture of both sparse and dense operations to be implemented efficiently.

We propose a solution to these challenges based on the following contributions:

- *a novel processor architecture utilising a Beneš permutation network* to improve the efficiency of concurrent access to on-chip memory; it supports both sparse and dense operations required by CG. (Section III)
- *efficient heuristics for scheduling CG operations* statically onto the proposed architecture. (Section IV)
- *an evaluation* of the effectiveness of the proposed architecture and heuristics for real-world problems on a commercial reconfigurable system. (Section V)

To the best of our knowledge, this is the first time the Beneš network is proposed for solving the multiple memory bank access problem in a processor for sparse linear algebra.

II. Background and Related Work

The conjugate gradient algorithm [3] constructs a set of mutually conjugate directions to form a basis of a linear space with respect to which it expands the solution vector $\vec{x}$. Iteratively, the new *search direction* $\vec{p}$ is chosen to minimise the *residual* $\vec{r}$ (the approximate error of a solution). The expansion of a solution vector in this new basis improves with every further basis vector $\vec{p}$, minimising the residual.

Algorithm 1 Conjugate gradient method.

```
1:  function ConjugateGradient(A, x, r)
2:      for i ≤ MaxIterations do
3:          A_p ← A p
4:          α ← rs_old / (p^T A_p)
5:          x ← x + α p
6:          r ← r − α A_p
7:          rs_new ← r^T r
8:          if rs_new < AbsoluteError^2 then
9:              return x
10:         end if
11:         p ← r + rs_new / rs_old × p
12:         rs_old ← rs_new
13:     end for
14: end function
```

CG implementations for dense systems [4] are easier to pipeline and achieve considerably higher memory efficiency due to the regular memory access pattern. On the other hand, sparse CG (where the system matrix **A** is sparse) is harder to parallelise due to the irregular access pattern and the mixture of both sparse and dense operations. The operation on line 3 of Algorithm 1 is a sparse matrix-vector multiplication (SpMV), but all other operations are dense vector-vector operations. Therefore, efficient SpMV implementations [5] are key to obtaining good performance but, as explained in Sections III and IV additional properties of the CG must be used to increase overall efficiency.

One such property, the time invariance of **A** (**A** is not updated during CG iterations) leads to an obvious optimisation used in [6]: copy **A** to the hardware accelerator only once for multiple iterations. We use this property to *statically schedule the entire algorithm* (Section IV). One limitation of [6] is the on-chip memory bottleneck which originates from the need to replicate (many times) the vector involved in SpMV to avoid bank conflicts. This is a common problem with many (and more recent implementations [7]) of sparse

matrix computations on FPGAs and is exactly what this paper addresses.

III. Architecture

To overcome the limitations of existing implementations for sparse CG we propose a new architecture using a permutation network to reduce bank conflicts without expensive resource usage. While this architecture benefits from a scalable communication network connected to the vector memory, the generation of appropriate control for this network requires careful design; an optimised control encoding will be introduced.

In the proposed architecture the matrix is stored as sparse and the intermediary vector results are stored as dense. As shown in Figure 1, the proposed architecture comprises the following elements:

1) a *shared vector memory* with multiple banks (VB_i) each capable of storing a subset of the elements of a dense vector (e.g. $\vec{p}$);
2) a *shared register file* for storing scalar values (e.g. $\alpha, rs_{old}, rs_{new}$);
3) multiple *processing elements* (PEs) which can perform arithmetic with operands from local memories, shared vector memory or the global register file (Figure 2);
4) a *permutation network* which enables vector memory banks to be accessed from any PE as long as no other PEs are accessing the same memory bank;
5) a *special processing element* (SPE) for operations that are required rarely (e.g. serial reduction of inputs, comparison of inputs with relative error threshold, scalar division);
6) an *adder tree* for reducing partial sums in the dot product operations.

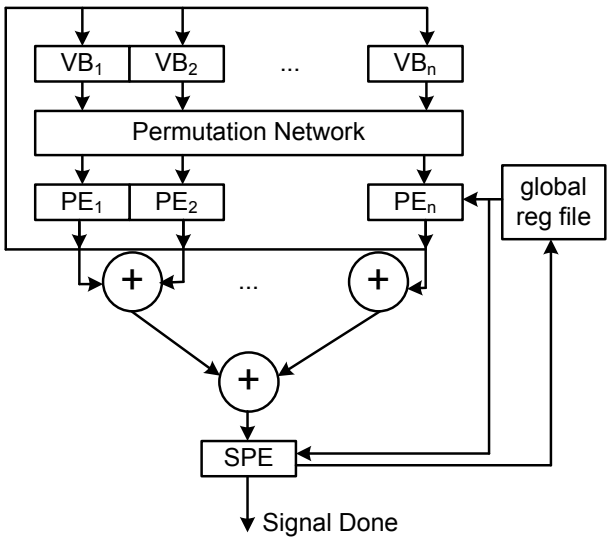

Fig. 1. Architecture of the proposed sparse CG solver.

The use of the permutation network allows more flexible access to the banks of the vector memory which results in better overall efficiency for the sparse matrix-vector multiplication. This helps reduce the impact of the irregular memory access pattern which is, as noted earlier, one of the greatest challenges in paralellising the CG method. The proposed architecture also achieves good performance for dense vector-vector operations (another important challenge) through careful PE design.

A. Shared Memories

The vector memory is used for storing $\vec{p}$ which is involved in the SpMV operation thus requiring random access. Elements of $\vec{p}$ are assigned to the N banks of the vector memory. At each cycle N values of $\vec{p}$ can be read by providing N separate read addresses. A particular permutation of these values is then sent to the N PEs in our design (one per PE). Because the vector memory is shared for all PEs, conflicts can occur when two PEs require elements of $\vec{p}$ which are stored in the same bank. We explain how we handle this in Section IV.

The register file is connected to every PE and the SPE and can broadcast its values to either (e.g. α required by the dense vector operations).

B. Processing Element

The majority of arithmetic operations required for CG are performed concurrently across multiple PEs. The architecture of a PE is shown in Figure 2.

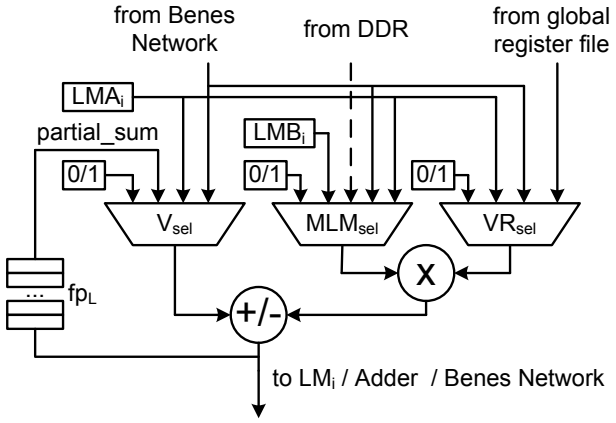

Fig. 2. Architecture of the Processing Element (PE).

Nonzero matrix values are read from off-chip DRAM. To minimise bank conflicts at runtime each PE has local storage for intermediate results — LMA_i, LMB_i. The two local memories are required to ensure the PE can simultaneously access all required operands on one clock cycle. Table I shows that all operators can be accessed concurrently with the proposed memory allocation. Note that the same value of $\vec{r}$ and $\vec{p}$ can be read more than once per cycle since LMA and the vector memory are connected to all three multiplexers.

Table II shows that the PE can perform all required operations without structural hazards when accessing functional units (multiplexers, multipliers, adders).

The SpMV operation introduces the complication of accumulating partial sums. As discussed in Section IV, each matrix row is assigned to one PE and inner products are

TABLE I. MAPPING OF VECTORS TO MEMORIES SHOWS THERE IS NO STRUCTURAL READ HAZARD FOR CG STEP (ALGORITHM 1).

Op.	Mem.	Step Used	Concurrent With	Hazard
A	DDR	3	p	No
p	V	3, 4, 5, 11	A, Ap, x, p	No
r	LMA	6, 7, 11	Ap, r , p	No
x	LMA	5	p	No
Ap	LMB	4, 6	p, r	No

TABLE II. THE PE CAN PERFORM ALL OPERATIONS USING AT MOST ONE OF ALL UNITS, THUS NO STRUCTURAL HAZARDS ARE POSSIBLE.

Operation	Op. Units	Hazard
$\vec{A_p} \leftarrow \mathbf{A}\vec{p}$	MLM$_{sel}$, VR$_{sel}$, V$_{sel}$, add, mult	No
$\alpha \leftarrow \frac{rs_{old}}{(\vec{p}^T A_p)}$	MLM$_{sel}$, VR$_{sel}$, mult	No
$x \leftarrow x + \alpha\vec{p}$	V$_{sel}$, MLM$_{sel}$, VR$_{sel}$, add, mult	No
$r \leftarrow r - \alpha\vec{A_p}$	V$_{sel}$, MLM$_{sel}$, VR$_{sel}$, add, mult	No
$rs_{new} \leftarrow \vec{r}\vec{r}^T$	MLM$_{sel}$, VR$_{sel}$, mult	No
$r \leftarrow r - \alpha\vec{A_p}$	MLM$_{sel}$, V$_{sel}$ VR$_{sel}$, mult, sub	No
$p \leftarrow r + \frac{rs_{new}}{rs_{old}}\vec{p}$	V$_{sel}$, MLM$_{sel}$, VR$_{sel}$, add, mult	No

accumulated inside the PE. The floating point accumulator is highly pipelined to allow higher operating frequencies, but this introduces the complication of keeping the accumulator pipeline full and ensuring correctness of accumulated sums; this complication is addressed through the static scheduling strategy described in Section IV.

C. Beneš Permutation Network

During the matrix-vector multiplication step (Step 3, Algorithm 1), an element of $\vec{p}$ is required which might come from any of the V memory banks. Previous proposed architectures [6] use a crossbar to provide arbitrary access to the vector memory, but their scalability is limited by an asymptotic resource cost of $O(N^2)$ for an N bank vector memory. We propose to solve this problem by utilising a *Beneš network*.

Permutation networks [8]–[10] such as the Beneš network can be used to generate arbitrary permutations of input elements by recursively connecting sub-blocks that permute fewer and fewer elements. The base case is a block which permutes two elements based on a binary control signal. This divide and conquer approach results in lower resource usage of $O(Nlog(N))$, making the Beneš network more scalable. Beneš networks have been used on instruction level for performing arbitrary permutations among sub-words to accelerate different applications [11].

However, our solution must now generate control bits for each of the network's permutation block (on the order of $O(Nlog(N)$ control bits). Given the time invariance of **A** we are only required to generate such control bits once, which reduces the overall performance impact.

D. Special Processing Element

The special processing element (SPE) handles operations which are used comparatively rarely during CG iterations. Grouping all such operations in a separate processing element (which is not replicated) reduces the number of idle functional units, improving hardware utilisation. Operations performed by the SPE are *1)* serial reduction required for computing vector norm, *2)* scalar division and *3)* threshold convergence test. For serial reduction of N inputs we used a design with logarithmic number of adders in the latency of the floating point adder as described in [5].

E. Control Encoding

Control bits for the PEs, permutation network and other elements of our architecture are pre-computed on the CPU of the accelerator system and, due to time invariance of **A**, need only be generated once for all iterations of the CG. Table III gives an overview of the number of control bits required for an architecture with N_{PE} processing elements which supports square system matrices with up to N rows. The maximum size of each vector memory bank is $VB_{size} = k * N/N_{PE}$ where the meaning of k is explained in Section IV. One extra bit per bank is required to control the write enable signal. Write addresses into the vector memory are sequential and therefore can be generated by counter state machines.

TABLE III. CONTROL BITS REQUIRED PER CYCLE.

Element	Symbol	Number of Bits
Permutation Network	B_{PN}	$N_{PE}log_2N_{PE}$
Processing Element Control	B_{PE}	4
Special Processing Element	B_{SPE}	2
Register File	B_{RF}	4
Vector Memory	B_{VM}	$N_{PE} * (\lceil log_2(VB_{size}) \rceil)$

According to Figure 2 each PE requires 13 control bits: 12 bits for the two 5-input and one 6-input multiplexers and one bit for add/subtract selection. However, since the total number of distinct operations performed by the PE is less than 16, we can minimise this to just *4 control bits* per PE. The SPE requires two bits to select between the three operations it can perform. The register file requires two bits to select between broadcasting α, rs_{old}, rs_{old}/rs_{new} and two bits for write enable (corresponding to disabled, update α, update rs_{new}).

Thus the number of control bits required per cycle is:

$$B_{Control} = B_{PN} + B_{PE} + B_{SPE} + B_{RF} + B_{VM} \quad (1)$$

Some control bits can be generated on the FPGA, to reduce the DRAM transfer overhead. For example, write and read addresses for the LMA and LMB could be replaced by on-chip state machines since their values can be fully determined based on cycle count and operation stage. In addition, many control bits will not change for hundreds, possibly thousands of instructions. For example, the control bits for the register file will not change while computing the SpMV or from steps 4 to 6, the control bits of the PEs will also not change

while computing an SpMV (with the exception of the case where a stall is required) etc. We expect that compressing the control bits (e.g. via run-length encoding) could lead to much improved results in practice but leave such optimisation opportunities as future work.

This encoding leads to a *specific sparse* storage format, in which we encode a matrix as a tuple of (nonzero values, control bits), as opposed to the traditional Compressed Sparse Row (CSR).

F. Operation

On the whole, the operations performed by the proposed architecture to support *one iteration* of CG are:

1) Fetch nonzeros and control bits from memory
2) Run corresponding operations:
   a) *SpMV* - parallel on all the PEs
   b) *Dot product* (PEs in parallel for inner-products, adder tree and SPE for reduction)
   c) *Compute and update PE memories* from steps 4, 5 (dense add, subtract, multiply)
   d) Compute rs_{new} (dense dot product)
3) Check value of Signal Done and stop if high
4) Compute new value of $\vec{p}$
5) Copy $\vec{p}$ to vector memory for next iteration

IV. Scheduling

In this section, we introduce heuristics for mapping the irregular SpMV operations onto our proposed architecture. The mapping procedure can be divided into 2 parts: allocating elements of $\vec{p}$ into different memory banks (**index allocation**), and allocating the actual SpMV operations into each PE (**SpMV allocation**). Although it is possible to develop heuristics to achieve the goal by considering both index and SpMV allocation at the same time, such heuristics will be time consuming and hence they are considered separately. The two static scheduling heuristics are computed on the CPU of the accelerator system.

A. Index allocation

In our architecture, each vector memory bank stores a subset of the elements of $\vec{p}$ for the SpMV operation. This arrangement can generate conflicts when multiple PEs require access to elements which are stored in the same bank. To minimise the idle cycles resulting from bank conflicts during the SpMV, elements of $\vec{p}$ should be allocated to different memory banks such that each memory bank will be accessed the same number of times in each SpMV iteration. To achieve this goal, we define the workload (wl) of index i as the number of non-zeros in the column i of matrix $\mathbf{A}$,

$$wl_i = nnz(\mathbf{A}_{i,*}) \tag{2}$$

By defining the set of indices allocated to memory bank k as id_k, the workload of memory bank k can be defined as,

$$bankload_k = \sum wl_i \ \ (i \in id_k) \tag{3}$$

and the goal of the index allocation heuristic is to minimise the following cost function f,

$$f = max(bankload_k) - min(bankload_k) \tag{4}$$

Although we might achieve a reduced cost function by allocating a different number of indices to different banks, we restrict our heuristic to allocate the same number of indices to each bank so that the resulting design is more uniform. The heuristic performs the allocations in batches of N_{PE} indices iteratively. In every iteration, we select N_{PE} unallocated indices and sort them with their workload wl_i. We also sort the memory bank with their bank workload $bankload_k$, and allocate exactly one index to each bank such that the memory bank currently having lowest workload will get the index with highest workload. The bank workload $bankload_k$ will be updated and next iteration will continue until all indices are allocated. Figure 3 shows how indices are allocated for even $bankload$.

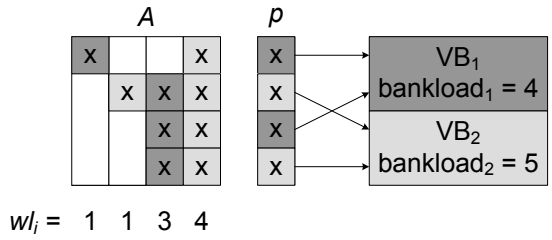

Fig. 3. Example of index allocation.

B. SpMV allocation

After the index allocation, the next step of our static scheduling process is to map each sparse vector-vector multiplication onto our proposed architecture. Algorithm 2 shows the SpMV allocation heuristic: in each iteration, an element of the row vector being computed by a PE will be picked to update the partial product with $a_{i,j} \times p_i$ if p_i can be found on a memory bank unused in this iteration. If none of the elements in the current computing row vector can be scheduled (i.e. bank conflict), a stall operation will be used (i.e. partial product plus zero).

Algorithm 2 SpMV allocation heuristic.

```
R = set of unallocated row vector in A sorted descendingly with # elements, C_k = set of element in a row vector to be scheduled to PE k currently, U = set of unused memory banks in this iteration.
while R ≠ ∅ or C_k ≠ ∅ do
    U ← {1, 2, ...N_PE}
    for k = 1 to N_PE do
        if C_k = ∅ then
            C_k ← elements of first vector of R
        end if
        remove an element from C_k such that its column index is found in bank j, j ∈ U, remove j from U
    end for
end while
```

Another constraint of the SpMV scheduling is the latency of the partial sum accumulation loop. Due to the pipeline

latency of the floating point adder, the partial sum can only be added once every fp_L cycles. Hence, we divide the rows of $\mathbf{A}$ into fp_L groups having roughly the same number of nonzeros (i.e. workload). fp_L independent runs of algorithm 2 are used to schedule each group. The time required to finish the entire SpMV is thus the maximum required iteration among all fp_L scheduling.

C. Vector duplication

As follows from Algorithm 2, the efficiency of the SpMV is significantly affected by the number of stall operations due to bank conflicts. To reduce such efficiency degradation, we duplicate and rotate the vector $\vec{p}$ in the V memory bank so that each memory bank holds $(N_{dup} \times N)/N_{PE}$ elements of $\vec{p}$ rather then N/N_{PE} elements where N_{dup} is the number of duplications. Although this arrangement reduces the maximum dimension of the vector we can handle given the same amount of memory, experimental results show that 2 to 3 duplications are enough for us to achieve good efficiency in the SpMV step for most of our real-world benchmarks.

V. Evaluation

Since the performance of Krylov Subspace methods is *highly dependent* on the systems, we use system matrices from real-life applications obtained from the *University of Florida Sparse Matrix Collection* [12] as our benchmarks. All matrices used are symmetric and positive definite, which is a prerequisite of the CG method. We analyse matrices whose dimension is larger than 10K elements, since for smaller sizes a direct solver approach would be better suited.

A. CPU, GPU and FPGA Solvers

We implement the proposed architecture on a Maxeler Maia accelerator card with an Altera Stratix V D8 chip, 48 GB of on-board memory and Infiniband connection to a Xeon E5-2640.

Table IV shows that the maximum size of the architecture which can be built on the D8 is bounded by the BRAM usage of the PE's local memories, followed by that of the vector memory. We note that the BRAM usage of the adder tree corresponds to additional buffering introduced by MaxCompiler. Additional resources (not shown) are used for the memory controller and input/output buffering, resulting in full utilization of the available BRAM resources.

TABLE IV. Resource usage for all components.

Element	FF %	LUT %	BRAM %	DSP %
Permutation Network	0.02	5.57	0.78	0.00
Vector Memory	5.38	3.72	14.96	0.00
Adder Tree	14.30	7.87	12.47	0.00
Special Processing Element	0.64	0.38	0.70	0.00
128 Processing Elements	29.20	20.43	44.88	6.52
Total	49.54	37.97	73.78	6.52

Therefore the largest possbile design on a commercially available FPGA has 128 single precision processing elements (each with 844 element deep local memories) and a vector memory of 128 banks, each 563 elements deep. Both sizes enable us to fit matrices up to 36K elements with duplication or matrices up to 72K elements without duplication.

The static scheduling heuristics described in Section IV and Algorithm 2 are implemented in carefully optimised multithreaded C++ routines using the Boost library. These routines run once for the entire conjugate gradient solver of Algorithm 1.

We compare our approach with optimised CPU and GPU solvers by measuring the average time per iteration for CPU and GPU. For both solvers we use an $AbsoluteError$ (Algorithm 1, line 8) of 1E-6, which is considered sufficiently accurate for many real-life applications.

The GPU version uses an optimised CG solver from the Cusp framework, based on the CUDA SDK 5.0 and the Thrust framework for parallel algorithms. We compile the design using the CUDA Compiler with `-O3 -use_fast-math -m64` optimisation flags and run it on a Tesla C2070 card (448 Cores and 5375MB of DDR3 memory). All measurements do not include the time to transfer data over PCIe to the GPU's on-board DRAM. We use the CSR format for the sparse matrix.

The CPU benchmark uses an optimised double precision CG solver based on Intel MKL Sparse BLAS routines [13]. MKL does not provide a single precision Conjugate Gradient routine and as such we use symmetric CSR stoarge in the CPU implemntation to offset the memory overhead introduced by double precision storage format. The symmetric storage almost halves the data size, making symmetric double matrix comparable to non-symmetric double storage. However for each nonzero entry two vector updates must be performed and memory traffic (due to vector element transfer) is not directly halved since additional performance degradation may result from reduced locality due to the double precision vector format. We use both double precision Conjugate Graident solver (`dc_get`, `dc`) and symmetric sparse matrix vector mutliplication (`mkl_dcsrsymv`). The CPU solver is compiled using the Intel C Compiler 12.1.4, with optimisation flags `-O3 -m64` and run on 6 core Xeon X5650 with HyperThreading (12 threads total), with peak memory bandwidth of 32GB/s.

B. Performance Model for FPGA Implementation

We estimate the execution time for one iteration of the Conjugate Gradient on the proposed architecture using the following performance model which analyses both DRAM transfer and compute time. We assume the sparse matrix and control bits are stored in DRAM memory on the accelerator and do not include overhead of PCIe transfer in our model. Parameters used in our estimation are summarised in Table V.

1) Transfer Time: Peak DRAM bandwidth on the tested device is 65GB/s:

$$DRAM_b = 65 * 1024^3 * 8$$

The number of bits per cycle is therefore:

$$BpC = DRAM_b/\mathrm{F}$$

The number of data bits to transfer is given by:

$$DB = N_{nnzs} * D_{width}$$

TABLE V. PARAMETERS USED TO ESTIMATE THE EXECUTION TIME OF ONE CG ITERATION ON THE PROPOSED ARCHITECTURE.

Parameter	Description	Value
N	Matrix Dimension	Matrix Specific
N_{nnzs}	Number of nonzeros	Matrix Specific
F	Clock Frequency	150 MHz
$DRAM_b$	DRAM Bandwidth	65 GB / s
N_{PE}	Number of PEs	128
D_{width}	Data width	32 bits
$fpAdd_l$	Floating point adder latency	16 cycles
$fpSub_l$	Floating point subtractor latency	16 cycles
$fpMul_l$	Floating point multiplier latency	30 cycles
$fpDiv_l$	Floating point divider latency	30 cycles

The number of control bits is given by Equation 1:

$$CB = B_{PN} + B_{PE} + B_{SPE} + B_{RF} + B_{VM}$$

The total FPGA transfer time is then:

$$T_{transfer} = (DB + CB)/DRAM_b$$

2) Compute Time: The time for transferring data and control bits during the SpMV operation is:

$$C_{SpMVTransfer} = (C_{SpMV} * CB + DB)/BpC$$

where $C_{SpMVMax}$ is determined using the scheduling heuristic and takes into account conflicts in memory banks.

Accounting for duplication as explained in Section IV-C, the time required to update the V memory with the new value of $\vec{p}$ (where k is the number of duplications):

$$C_{copy} = N/N_{PE} * k$$

Cycle count to perform the other operations required by the CG algorithm accounts for two floating point divisions:

$$C_{div} = 2 * fpDiv_l$$

two vector dot products:

$$C_{dot} = 2 * (log_2(N_{PE}) + N/N_{PE} + fpMult_l + fpAdd_l)$$

and three vector operations (2 additions, 1 subtraction, each pre-multiplied with a constant):

$$C_{vop} = 3 * (N/N_{PE} + fpMult_l) + fpAdd_l + fpSub_l$$

The total number of cycles required for other operations is therefore:

$$C_{other} = C_{vop} + C_{div} + C_{dot}$$

The total number of cycles required for one iteration of the CG is therefore:

$$C_T = max(C_{SpMVTransfer}, C_{SpMV}) + C_{copy} + C_{other}$$

This gives the total compute time:

$$T_{compute} = C_T/F$$

Therefore the total FPGA execution time is:

$$T_{total} = max(T_{compute}, T_{transfer})$$

C. Results

Table VI shows that for larger matrices the proposed architecture could be up to 3.6 times faster than the CPU and up to 2.6 times faster than the GPU. For smaller matrices, the conjugate gradient method converges quickly and the overheads for GPU initialisation are significant and workload is not high enough to saturate all cores. Hence our FPGA solver and CPU solver are considerably faster than GPU solver for the smaller matrices in our benchmark.

However, the FPGA solver cannot significantly outperform the CPU solver in the `Pres_Poisson` benchmark because its performance gains in the SpMV kernel and other vector operations are offset by the initial static scheduling overhead. For larger matrices, the CG solvers require more iterations to converge, and our FPGA solver outperforms both CPU and GPU solvers in these benchmarks (Figure 4).

TABLE VI. AVERAGE TIME PER CONJUGATE GRADIENT ITERATIONS AND SPEEDUP VERSUS CPU (**S CPU**) AND GPU (**S GPU**).

Matrix	Size	CPU (ms)	GPU (ms)	FPGA (ms)	S CPU	S GPU
Pres_Poisson	14822	0.25	0.64	0.2146	1.14	2.98
olafu	16146	0.37	0.66	0.1078	3.44	6.11
nd6k	18000	2.53	1.34	0.9852	2.57	1.36
smt	25710	0.92	1.03	0.5029	1.82	2.04
thread	29736	1.47	1.22	0.4640	3.17	2.64
ship_001	34290	1.81	1.27	0.4921	3.68	2.57
nd12k	36000	5.59	2.27	1.7698	3.16	1.28
cant	62451	1.75	1.16	0.4578	3.82	2.54
crankseg_1	52804	4.97	2.04	1.5604	3.18	1.31
crankseg_2	63838	6.53	2.60	1.8566	3.52	1.40

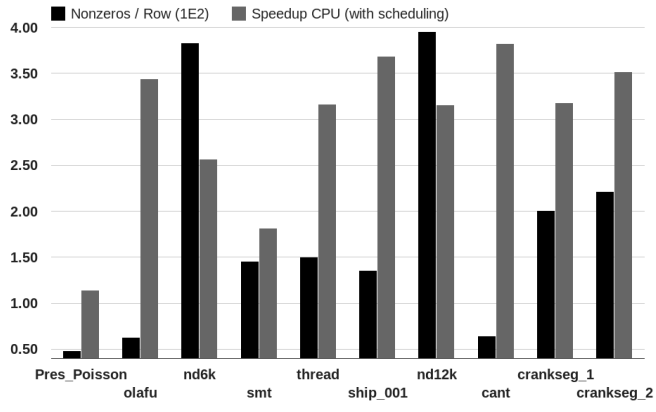

Fig. 4. FPGA solver speedup (compared with CPU) and average number of nonzeros per row for the benchmark set.

Figure 5 shows that increasing the number of duplications does not substantially improve the efficiency achieved. This removes the need for duplication for larger matrices providing the opportunity for a run-time trade-off between the size of

the matrix supported and the efficiency of the scheduling. The efficiency achieved without duplication is still high enough, so that we observe a good speedup for large matrices such as `crankseg_1, cant, crankseg_2` for which the corresponding conjugate vector $\vec{p}$ would otherwise be too large to fit in the vector memory.

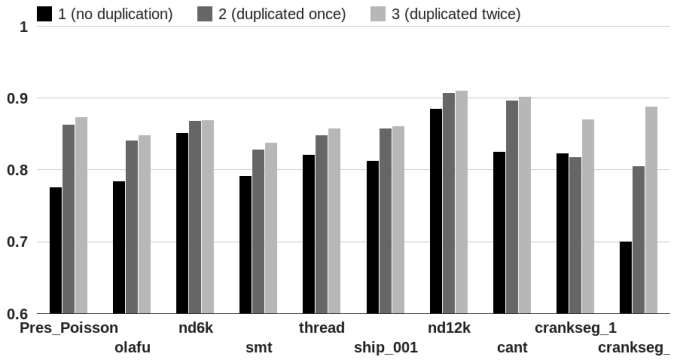

Fig. 5. Efficiency of SpMV in proposed FPGA solver for different number of vector duplication N_{dup}.

Figure 6 shows that for the larger matrices in our benchmark the scheduling overhead is small compared to the total execution time (maximum of transfer time and compute time). With the exception of the `Pres_Poisson` benchmark, the static scheduling accounts for at most 25.10% of the average execution time per conjugate gradient iteration.

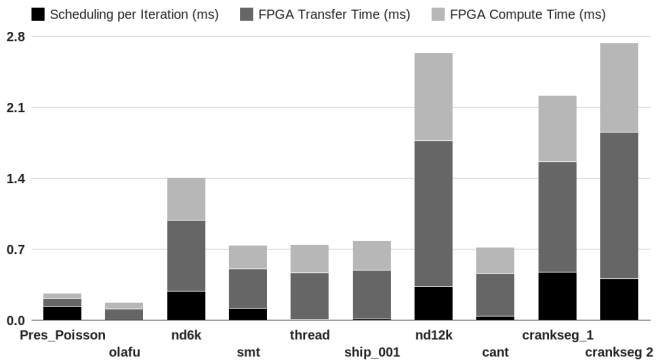

Fig. 6. Scheduling, transfer and compute time for a CG iteration on the proposed architecture.

Based on the evaluation we can conclude that our approach is better applicable to larger systems where acceleration is most needed.

VI. Conclusion

We introduce a novel architecture for an efficient conjugate gradient solver on reconfigurable accelerator system which can achieve speedups of up to 3.6 times over an optimised multithreaded CPU. This uses a Beneš permutation network and a simple but effective static scheduling heuristic to allow better and cheaper conflict resolution for sparse matrix vector multiplication operations. Since it supports all required operations our approach can be easily extended to cover other Krylov Subspace methods such as the biconjugate gradient [14]. Such extensions are left as future work along with many possible optimisations such as exploiting the symmetry of matrices to increase performance [15], development of more optimised scheduling heuristics [16], compression of control bits and storage formats [17] and use of fixed point arithmetic [18].

Acknowledgement

This work is supported in part by the European Union Seventh Framework Programme under grant agreement number 257906, 287804 and 318521, by the UK EPSRC, by the Maxeler University Programme, by the HiPEAC NoE, by Altera, and by Xilinx.

References

[1] G. Dahlquist and Å. Björck, *Numerical Methods*. Dover Publications, 2003.

[2] J. Dongarra and F. Sullivan, "Guest editors introduction: the top 10 algorithms," *Computing in Science & Engineering*, vol. 2, no. 1, pp. 22–23, 2000.

[3] J. Nocedal and S. J. Wright, *Conjugate gradient methods*. Springer, 2006.

[4] A. R. Lopes and G. A. Constantinides, "A high throughput FPGA-based floating point conjugate gradient implementation," in *Proc. ARC*, 2008, pp. 75–86.

[5] L. Zhuo and V. K. Prasanna, "Sparse matrix-vector multiplication on FPGAs," in *Proc. FPGA*. ACM, 2005, pp. 63–74.

[6] 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. FCCM*, 2006, pp. 3–12.

[7] R. Dorrance, F. Ren, and D. Marković, "A Scalable Sparse Matrix-vector Multiplication Kernel for Energy-efficient Sparse-blas on FPGAs," in *Proc. FPGA*, 2014, pp. 161–170.

[8] V. E. Beneš, *Mathematical theory of connecting networks and telephone traffic*. Academic Pr, 1965, vol. 17.

[9] A. D. Ruslanov and J. R. Johnson, "An FPGA Implementation of Bene" Permutation Networks," in *Proc. FPGA*, 2004, pp. 245–245.

[10] A. Waksman, "A permutation network," *Journal of the ACM*, vol. 15, no. 1, pp. 159–163, 1968.

[11] C. H. Ho, W. Luk, J. Szefer, and R. Lee, "Tuning instruction customisation for reconfigurable system-on-chip," in *Proc. SOCC*, 2009, pp. 61–64.

[12] T. A. Davis and Y. Hu, "The University of Florida sparse matrix collection," *ACM Transactions on Mathematical Software*, vol. 38, no. 1, p. 1, 2011.

[13] J. Dongarra, V. Eijkhout, and A. Kalhan, "Reverse communication interface for linear algebra templates for iterative methods," *UT, CS-95-291, May*, 1995.

[14] R. Fletcher, "Conjugate gradient methods for indefinite systems," in *Numerical Analysis*. Springer, 1976, pp. 73–89.

[15] J. D. Bakos and K. K. Nagar, "Exploiting matrix symmetry to improve FPGA-accelerated conjugate gradient," in *Proc. FCCM*, 2009, pp. 223–226.

[16] D. Nassimi and S. Sahni, "Parallel algorithms to set up the Benes permutation network," *IEEE Transactions on Computers*, vol. 100, no. 2, pp. 148–154, 1982.

[17] S. Kestur, J. D. Davis, and E. S. Chung, "Towards a universal FPGA matrix-vector multiplication architecture," in *Proc. FCCM*, 2012, pp. 9–16.

[18] J. L. Jerez, G. A. Constantinides, and E. C. Kerrigan, "Fixed point lanczos: Sustaining TFLOP-equivalent performance in FPGAs for scientific computing," in *Proc. FCCM*, 2012, pp. 53–60.