Dataflow acceleration of Krylov subspace sparse banded problems

Pavel Burovskiy*, Stephen Girdlestone†, Craig Davies†, Spencer Sherwin‡, Wayne Luk*
*Department of Computing, Imperial College London
‡Department of Aeronautics, Imperial College London
†Maxeler Technologies
{p.burovskiy, s.sherwin, w.luk}@imperial.ac.uk, {girdlestone, craig}@maxeler.com

***Abstract*—Most of the efforts in the FPGA community related to sparse linear algebra focus on increasing the degree of internal parallelism in matrix-vector multiply kernels. We propose a parametrisable dataflow architecture presenting an alternative and complementary approach to support acceleration of banded sparse linear algebra problems which benefit from building a Krylov subspace. We use banded structure of a matrix A to overlap the computations $Ax, A^2x, \ldots, A^kx$ by building a pipeline of matrix-vector multiplication processing elements (PEs) each performing A^ix. Due to on-chip data locality, FLOPS rate sustainable by such pipeline scales linearly with k. Our approach enables trade-off between the number k of overlapped matrix power actions and the level of parallelism in a PE. We illustrate our approach for Google PageRank computation by power iteration for large banded single precision sparse matrices. Our design scales up to 32 sequential PEs with floating point accumulation and 80 PEs with fixed point accumulation on Stratix V D8 FPGA. With 80 single-pipe fixed point PEs clocked at 160Mhz, our design sustains 12.7 GFLOPS.**

***Index Terms*—Krylov subspace, matrix powers, iterative solvers, matrix exponentials, sparse matrix, SpMV, dataflow, performance model**

I. Introduction

It is well-known that the performance of sparse matrix-vector multiplication (SpMV) kernels is significantly limited by the available memory bandwidth. They sustain only a fraction of peak compute performance due to difficulty in data reuse and irregular data access patterns. As a result, sparse linear algebra designs in most cases under-utilise available compute capacity of contemporary CPUs, GPUs and FPGAs [2].

This problem is recognised by many researchers: Kestur et. al. [5] propose additional data compression utilising spare compute cycles. Similar activities exist on CPUs in a more generic setting [11]. Despite this, a substantial amount of work on sparse linear algebra is focused on saturating available memory bandwidth by increasing internal parallelism of an SpMV kernel, either in general purpose [4], [3], [9] or application specific [1], [2], [6] setting.

In this paper we propose a domain-specific approach to increase utilisation of compute capacity of reconfigurable hardware, in order to avoid bandwidth limitation becoming performance scaling bottleneck. We present a parametrisable architecture targeting banded sparse linear algebra problems which benefit from building a Krylov subspace:

$$\text{span}\{x, Ax, A^2x, \ldots, A^kx\}. \tag{1}$$

Such applications range from iterative linear and eigenvalue solvers and approximations to matrix exponentials to approximate computation of matrix inverse and even Google PageRank citation ranking by the power method [10].

The main contributions of this paper are as follows.

- A new parametrisable dataflow architecture to increase compute intensity of SpMV kernels beyond the bounds of available memory bandwidth, by exploiting mathematical structure of the problem. The architecture is presented in Section III.
- A performance and critial resource utilisation model in Section IV.
- An evaluation of the proposed architecture in Section V for a power iteration method, specialised to large unstructured sparse banded matrices, targeting Maxeler Maia card with an Altera Stratix V D8 FPGA. We measure its performance in order to verify our performance model, and compare it to a CPU implementation using OpenMP parallel SpMV kernel available from Intel MKL. Finally we estimate the performance for other possible choices of architecture parameters.

II. Background and related work

Google PageRank computation by the power method is an eigenvalue problem which consists of repeated matrix-vector multiplications aimed to converge to a dominant eigenvector, thus drawing a connection to Krylov subspace methods. With web sites linking mostly themselves and a few related neighbours, the page connectivity matrix should be sought as banded under band minimisation node reordering, at least locally.

Efficient computation of sparse Krylov subspace problems is challenging due to the low arithmetic intensity of matrix-vector multiply kernels and the sparsity that prohibits usage of matrix-matrix multiply kernels. Although direct computation of the matrix power A^k can be done in $\log k$ matrix-matrix multiply steps, it may change the sparsity structure of matrix A, eventually leading to a much denser structure which negates performance benefits.

Instead, if one computes matrix actions in expression (1) as a sequence of independent matrix-vector products,

$$Ax^0 = x^1,\ Ax^1 = x^2, \ldots,\ Ax^{k-1} = x^k, \tag{2}$$

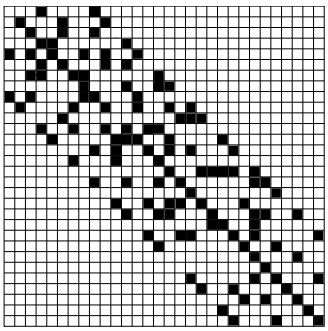

Fig. 1. Example of a matrix sparsity structure considered throughout the paper. Black squares represent nonzero elements. Matrix can be nonsymmetric.

for large matrix problems, memory traffic associated with fetching matrix data and vectors will grow linearly with k as $O(k\ n_{nnz})$ and $O(kn)$ accordingly, where n is a matrix rank and n_{nnz} is its number of nonzero entries. Thus, due to low data re-use, the execution becomes solely DRAM bound, resulting in k times increase in runtime. This effect can be observed on data sets exceeding the last level CPU cache size by a factor.

As an alternative, Rafique et. al. [7], [8] propose computing several matrix-vector products in parallel on FPGA for dense matrices with a tiny band. Their approach is to split the matrix and vectors into partitions and feed each partition to a complex processing element (PE), which computes all matrix powers on its own data. This results in overlapping computation fronts due to dependencies between data partitions (via shared parts of right hand side vectors), leading to either halo exchange between partitions, or redundant computations. As a result, there is a communication-computation trade-off which limits that computation to the tiny matrix bands and constraints scalability in terms of the parameter k.

In the next section we propose a more scalable approach to overlapping matrix power evaluations on reconfigurable hardware.

III. Krylov subspace architecture

We present a generic and parametrisable architecture of computing the basis of Krylov subspace (1), specialised to large, not necessarily symmetric sparse unstructured matrices whose nonzero entries lie in a relatively narrow band, centered along the diagonal (Fig. 1).

To deal with large matrix problems, both matrix data and vectors are stored in off-chip DRAM and streamed to the chip and back where necessary.

In order to avoid DRAM bandwidth limit becoming a scalability bottleneck, we maintain a shifting window into the matrix data on-chip and pass them from one SpMV processing element to another, offsetting matrix-vector product evaluations $Ax, A^2x, \ldots, A^kx$ in time.

With on-chip buffering, off-chip memory traffic is reduced by a factor $O(k)$ and overlapped computing fronts are avoidable; the matrix-vector multiply problem becomes an $O(n_{nnz})$ linear sweep through the matrix nonzero data and $O(n)$ sweep through the vectors, at the cost of increased on-chip memory utilisation.

Section IV will report that the matrix band has crucial impact on architecture scaling in terms of both resource utilisation and sustainable performance. There, we show that for bands small compared to the matrix rank, the offsetting costs are asymptotically negligible.

A. Processing element

The design of a processing element is a *parameter* to the Krylov subspace architecture. We accept any implementation of a processing element as part of our pipeline, as long as it satisfies the following requirements. Processing element implements row-major matrix-vector multiply but does not communicate to the DRAM itself. It accepts the following inputs at every cycle (Fig.2):

- the pair `(`x^i_j`, in_validity_bit)`, where x^i_j is an element of the dense vector x^i, if `in_validity_bit` is 1;
- the sparse matrix A in any row-major sparse matrix storage format;
- a global enable synchronisation signal, which indicates to a PE that the input data are valid and SpMV evaluation should start.

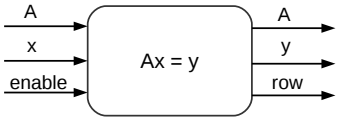

Fig. 2. The schematic representation of a Processing Element (PE)

The operation of the processing element can be summarised as follows:

- it reads p matrix elements per cycle, $p \geq 1$ and outputs them every cycle unchanged;
- it maintains a shifting window of width $w \geq b$ to the input vector in order to support banded matrix with band b for any matrix rank;
- it updates its shifting window once `in_validity_bit` is 1.
- in a cycle, it internally processes nonzero entries from one matrix row.

It has the following outputs:

- the pair `(`y^i_j`, out_validity_bit)` at every cycle and sets `out_validity_bit` to 1 whenever y^i_j, j-th element of the vector y^i, is valid;
- the number of the row it is currently processing;
- p matrix elements in the same storage format;
- a global stall synchronization signal, indicating internal data conflict.

The level of internal parallelism p corresponds to the number of matrix elements processed per cycle, and is the most important performance parameter of the processing element.

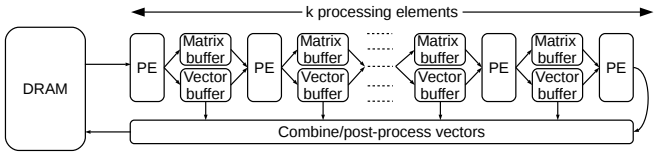

Fig. 3. Schematic diagram of a Krylov subspace architecture

B. Krylov subspace pipeline

We propose to build a Krylov subspace computation as a k-stage pipeline of processing elements (PEs). Each PE performs a single SpMV computation. The matrix A and the input vector x^0 are streamed from DRAM only once as inputs to the first PE. Then each PE passes the same matrix and vector x^i to the *next* PE locally on the chip, thus avoiding extensive communication to DRAM. The output of the pipeline is either the last vector x^k or some subset of vectors $\{x^i, i \in [1..k]\}$ (see discussion below in this section); the output vectors are sent to DRAM as the pipeline output.

There are a number of conceptual and implementation challenges of such pipelining which we describe below.

1) Data dependency: The elements of vector x^i should be computed before $Ax^i = x^{i+1}$ evaluation begins. In case of non-banded matrix A one needs to *complete* the previous computation $Ax^{i-1} = x^i$. The result is a consecutive evaluation of SpMVs, strictly one after another. As above, this leads to run times scaling linearly with k.

2) Using matrix structure to reduce run times: For the *banded* matrix A with bandwidth b, one may *overlap* the next SpMV computation with the previous: its banded structure implies that only a portion of a vector column is accessed for every matrix row. In the beginning, the computation $Ax^i = x^{i+1}$ needs at most b elements from the vector x^i since there are only b nonzero matrix entries within the band. Once the first b elements in x^i are ready, the *next* PE may start evaluating $Ax^i = x^{i+1}$. Thus, we take the banded structure of matrix A into account and overlap consecutive matrix-vector multiplications. Note the next PE may start its work earlier if PEs process matrix storage in row-major ordering.

3) Minimising off-chip matrix data traffic: In the beginning of *the next* matrix-vector computation, a processing element needs to access the first few matrix elements (or, the first matrix element) while *the current* computation has already progressed further into the matrix storage. To avoid repeated fetches of matrix data from DRAM, we keep a portion of matrix data local to the chip as a shifting window into the matrix storage.

In order to support k PEs, the architecture maintains $k-1$ buffers implementing shifting windows into the matrix and interleaves them with processing elements. Once a PE makes use of a matrix element from *the current* shifting window, it writes that element to *the next* matrix buffer. Therefore, matrix data are read from DRAM only once and then reused by all k matrix-vector computations. Here we benefit from a custom memory buffering and data movement strategy, a feature unsupported by commodity hardware.

4) Avoiding data hazards: Shifting windows into the matrix should be non-overlapping. Otherwise, the *next* PE completes the work on matrix elements it owns earlier than the *the current* PE shifts down the matrix. As a consequence, the *next* PE starts addressing the elements of its left hand side vector before they are ready, resulting in errors.

Generally, there is no guarantee of even data distribution within the matrix and thus shifting windows may start overlapping at some moment of computation.

To address this problem, it is necessary to either stall *the next* PE dynamically, or offset PEs in time sufficiently to avoid the problem. The second solution is less preferable since it increases the size of all matrix windows and thus requires more BRAMs.

5) Avoiding data loss: Shifting windows into matrix data needs to have no gaps between each other, otherwise some matrix data may miss the on-chip buffers and be effectively lost between matrix computations. Thus, either every PE is active processing their matrix elements and passing them further within the pipeline, or a global stall needs to happen to avoid data loss.

6) Minimising on-chip memory utilisation: The amount of on-chip memory spent on matrix buffers is proportional to the time offsets between the operation of neighbouring PEs. The shorter the delay between adjacent processing elements, the less on-chip memory is required for buffering.

7) Minimising off-chip vector data traffic: We buffer intermediate vectors on-chip in the same way as matrix data: while *the previous* processing element computes some further parts of its output vector, some *past data* from that vector are used by the successive PE. The same issues of data hazards and loss apply to vector buffers as well.

Buffering vectors eliminates the need for *reading* vectors $x^i, i > 0$ from DRAM, but not necessarily eliminates *writing*: further usage of intermediate vectors is an application-specific choice. There are three possibilities:

- only the final vector x^k is required. In this case there is no need to send first $k-1$ vectors to DRAM;
- the application implements post-processing of intermediate vectors. If such post-processing is done by another compute kernel on-chip, there is no off-chip vector data traffic associated with *this compute kernel*;
- all or some vectors are required. Then we buffer vectors between processing elements *and* send them to DRAM. In this case, the scalability of an architecture is limited by the DRAM bandwidth.

If an application does not need to store or post-process intermediate vectors, there will be no arrows from vector buffers in Fig. 3.

C. Parameter space

Our architecture offers opportunities for design space exploration:

1) Internal design of a processing element: The processing element can be implemented in various ways. It may have different levels of internal parallelism p. It can achieve a given level of parallelism in different ways. It can support floating point or fixed point accumulation. The resource utilisation of a single processing element is also a parameter.

2) Performance: There are two performance parameters: the internal parallellism of a processing element p, and the number of matrix powers k. The first parameter enables increasing DRAM bandwidth utilisation, while the second makes the Krylov subspace pipeline deaper and thus increases compute intensity for the same level of bandwidth utilisation.

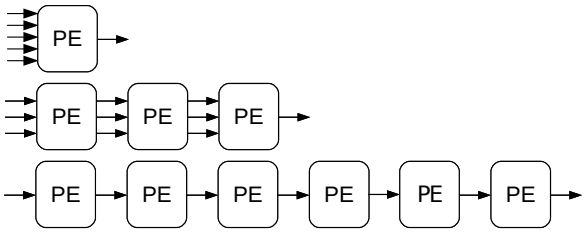

Fig. 4. Possible performance choices: one highly parallel PE, a pipeline of a few moderately parallel PEs, and a deep pipeline of sequential PEs

3) Frequency of stalls: The probability of Krylov subspace pipeline stalls is inversely proportional to the capacity of matrix and vector buffers. The lower excessive capacity of these buffers, the more probable global pipeline stall becomes.

4) Application-specific choices: Whether an application needs to post-process all or some of the vectors $x^1, ..., x^k$?

IV. Performance model

Let us assume that a stall due to data irregularities is a rare event, and the distribution of the number of nonzero entries in each row is close to uniform.

A. Speedup vs sequential FPGA implementations

The Krylov subspace pipeline aims to overlap SpVM computes, and due to data dependencies, it adds time offsets between consecutive SpMV computations. Therefore, the break-even point between speedup and slowdown takes place when computing all overlapped SpMV together with all offsetting overhead equals the time for evaluating k matrix powers by k independent SpMV evaluations without time offsetting:

$$T_{\mathrm{PE}} + (k-1)T_{\mathrm{off}} = kT_{\mathrm{PE}},$$

which simplifies to $T_{\mathrm{PE}} = T_{\mathrm{off}}$.

Processing matrix data by a single PE takes $T_{\mathrm{PE}} = \lceil \frac{n_{nnz}}{p} \rceil$ cycles. The minimal offsetting of two consecutive SpMV evaluations is given by the number of cycles for computing b elements of a shared vector. Since each matrix row has at most b nonzero entries, the upper bound for $T_{\mathrm{off}} = b\lceil \frac{b}{p} \rceil$. Thus, the condition $T_{\mathrm{PE}} = T_{\mathrm{off}}$ takes the form

$$\left\lceil \frac{n_{nnz}}{p} \right\rceil = b \left\lceil \frac{b}{p} \right\rceil,$$

or, approximately, $b \approx \sqrt{n_{nnz}}$. Therefore, the proposed architecture leads to slowdown with respect to a sequential SpMV for matrix bandwidths larger than $\sqrt{n_{nnz}}$.

The speedup factor is the ratio of run times in sequential and parallel cases:

$$S_p = \frac{kT_{\mathrm{PE}}}{T_{\mathrm{PE}} + (k-1)T_{\mathrm{off}}} = \frac{k\left\lceil \frac{n_{nnz}}{p} \right\rceil}{\left\lceil \frac{n_{nnz}}{p} \right\rceil + (k-1)b\left\lceil \frac{b}{p} \right\rceil} \to k$$

as long as $b \ll \sqrt{n_{nnz}}$ and $n_{nnz} \to \infty$.

In other words, the speedup factor S_p tends to k in the limit of large matrices with small bandwidth.

TABLE I
Estimated variations of speedup factor. Sequential SpMV

n_{nnz}	10^6				10^6			10^7
b	100	100	100	100	512	512	512	512
k	2	4	16	32	2	4	32	32
S_p	1.98	3.88	13.91	24.43	1.58	2.23	3.51	17.65

Note that S_p is an *additional* speedup factor relative to a baseline performance of a p-parallel SpMV kernel. Table I shows how S_p scales for some realistic matrix ranks, bandwidths and numbers of matrix powers.

B. Predicting performance: runtime and FLOPS

Here we show how the performance model predicts the run times and FLOPS rate of a Krylov subspace architecture. Table II shows the expected run time and FLOPS metric scaling for the 100Mhz design with $k = 32$ and a sequential SpMV kernel.

TABLE II
Estimated performance for p=1, k=32, $n_{nnz} = 10^6$ and $b = 10^2$ and clock frequency 100Mhz

	1 SpMV	32 SpMVs	Krylov subspace pipeline
Run time, s	0.01	0.32	0.0131
GFLOPS	0.07633	0.07633	2.44

The figures presented in the table are estimated as follows. If a sample design with $k = 32$ matrix powers is clocked at 100Mhz, the processing time for the matrix with $n_{nnz} = 10^6$ and $b = 10^2$ is estimated to be ${}^{10^6+31\times10^4}/_{100\times10^6} = 0.0131$ second. Although a single SpMV kernel should sustain just ${}^{10^6}/0.0131 \times 10^6 = 76.33$ MFLOPS, the projected FLOPS rate for a design with 32 matrix powers is estimated to be ${}^{32\times10^6}/0.0131 \times 10^9 = 2.44$ GFLOPS.

C. Resource utilisation

Since we buffer matrix and vector data on-chip, the capacity of the on-chip BRAM becomes a limiting factor to design scalability. Let us assume that the matrix uses a generic single

precision CSR sparse storage. The CSR format encodes every nonzero entry as a tuple {`vals, column_idx`}, both 32 bit wide, along with $n+1$ 32-bit indices providing variable count of nonzeros in each row. We will also assume that the maximum number of nonzero entries per row r is less than the band, $r \leq b$. Since we need to store b rows with up to r nonzero entries, one buffer storage for matrix data between 2 processing elements estimated as $32(2br+b)$ bits. We also need to store b vector entries in a vector buffer. Assuming single precision format, we have another $32b$ bits of on-chip storage. In total, buffering requirement for the design is estimated to be

$$M = (k-1)\left(2 \times 32 \times (b+br)\right)$$

bits. For example, a design with $k = 80$, $b = 128$ and $r = 64$ would require

$$M_{\text{sample}} = \frac{79 \times 64 \times (128 + 128 \times 64)}{8 \times 1024^2} = 5.01\text{MB}$$

of BRAM storage, the capacity available in contemporary high end FPGAs. There is no doubt that M is a lower theoretical bound to a BRAM resource utilisation of a real application.

Note that matrix nonzero entries provide dominating contribution to resource utilisation. A consequence of this estimate is that some degree of internal parallelism of PEs comes for 'free': PEs may have up to r copies of shifting windows into their column vectors (for processing $p \leq r$ matrix elements per cycle without bank conflicts) before these redundant buffers start making contribution to BRAM utilisation equal to a storage requirement of matrix buffers.

Finally, an application requiring a combine/post-processing unit as part of a Krylov subspace pipeline additionally needs to maintain an on-chip shifting window of an appropriate size. For example, a linear combination unit computing vector $y = \sum_i \alpha_i x^i$ needs to accept contributions from all k PEs, and different PEs contribute to the same element y_j at different moments of time. Thus, the width of its window should be proportional to k: $M_{\text{y}} = 32kb$.

V. Evaluation

A. Implementation

We demonstrate our approach by implementing a benchmark power iteration solver on a Maxeler Maia Dataflow Engine (DFE), an Altera 5SGSD8 FPGA-based acceleration board with 48GB of on-board DRAM memory, connected to a 2.5Ghz Xeon E5-2640 server with 12 physical CPU cores.

We implement a simple processing element with minimal resource utilisation in order to demonstrate the scalability of Krylov subspace architecture in terms of parameter k. Our PE processes only one matrix element per cycle from a large single precision CSR matrix. Also, PE assumes the input matrix does not have entries leaving band region. Violation of this assumption results in incorrect computation.

For the band $b = 128$ and $r = 64$ a *floating point* accumulator with an adder tree reduction circuit becomes constraint by LUT resources: with $k = 32$ PEs in the pipeline, the LUT utilisation is approaching 93%. Alternatively, a *fixed point* accumulator (with inputs still being single precision floats) allows the pipeline depth for the same b and r to increase up to $k = 80$, utilising 84% (2160) BRAMs and just 29% of LUTs. Note that using 2160 BRAMs in the Altera chip corresponds to 5.27MB of BRAM buffers, close to the performance model prediction. In either case, an FPGA pipeline is clocked at 160Mhz.

B. Experimental results

We compare our designs to an equivalent CPU version based on OpenMP parallel MKL routine `mkl_cspblas_scsrgemv`. We compile our code using Intel C Compiler 12.1.4 with flags `-O3 -m64 -mtune=native -march=native`.

For the measurements, we generate 3 synthetic sparse matrices of in-memory size 12MB, 1239MB and 2021MB to test cache-local and out-of-cache CPU performance and compare it to the FPGA runtime. The matrix parameters are presented in Table III. For every matrix, its r values are evenly distributed within the band.

TABLE III
Benchmark matrix problems

problem	1	2	3
rank	5×10^5	5×10^6	10^7
band	127	127	127
r	31	31	5
nnz	1.55×10^6	1.55×10^8	2.5×10^8
fits to	L3	out-of-cache	out-of-cache
size, MB	12MB	1239MB	2021MB
FPGA, ms	10.75	970.1	1563.8
CPU, 12T, ms	17.57	3149.73	5221.42
speedup, $k = 80$	1.63	3.24	3.33

In order to estimate the performance of a power iteration computation on a CPU, we consecutively run `mkl_cspblas_scsrgemv` routine k times using $k+1$ different vectors, evaluating Equation (2). We allocate their storage independently, thus drawing no guarantee of their contiguous memory arrangement. Initially, we run this procedure a few times as a warm-up precaution; we then measure the execution duration of a power iteration repeated 10 times, and calculate an average duration of a single power iteration procedure. Finally, we repeat the above measurements with `MKL_NUM_THREADS` equal to 1, 2, 4, 6 and 12 in order to test the multi-core scaling of performance.

On FPGA, we first transfer matrix data to the DFE's local DRAM and then measure 10 runs of a Krylov subspace pipeline which evaluates k matrix-vector multiplications in one go. We then calculate an average duration of power iteration pipeline execution.

To alleviate runtime deviations on both CPU and FPGA, we re-run time measurements 3 times and choose the lowest value.

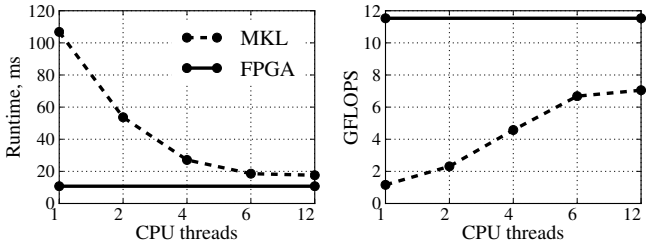

Fig. 5. Performance of L3 cache problem, $k = 80$, 12MB matrix dataset

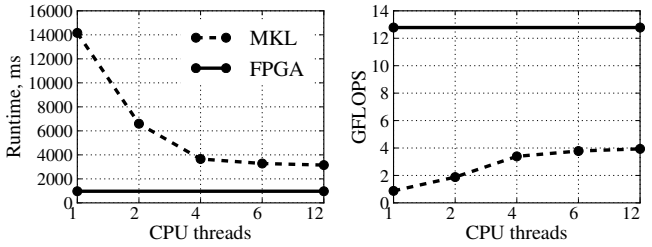

Fig. 6. Performance of large problem, $k = 80$, 1239MB matrix dataset

The results of a design with $k = 80$ PEs processing our test problems are presented in Fig. 5 and Fig. 6. For the L3 cache local and out-of-cache problems FPGA delivers 1.6 to 3.3 speedup versus 12 OpenMP thread running on 2 socket system accordingly. We stress this is achieved by an internally sequential SpMV design which dramatically under-utilises DRAM throughput capacity. Since spare chip resources for this design allow for more computation to happen, there is a space for further acceleration, *e.g.* using p-parallel SpMV designs.

For the comparison, the Fig. 7 presents similar measurements for $k = 32$ matrix power problem with the same large matrix of rank 5×10^6. Here, the FPGA design with a floating point accumulator is used. For this setting, the FPGA run time *stays the same* up to a measurement error as for a problem with $k = 80$, although it performs $\approx {}^{32}/_{80} = 0.4$ times less compute work, thus sustaining 5.1 GFLOPS instead of 12.7 GFLOPS for $k = 80$. CPU evaluation shows the opposite trend: the FLOPS rate stays the same while runtime scales with factor 0.4: it takes 5663 ms for $k = 32$ and 14162 ms for $k = 80$.

It is worth pointing out the relevance of out-of-cache execution on multi-core CPUs. For our large test problems a cluster with about 100 12-core CPUs is required to fit our matrix (split into parts) to its last level caches, despite of just 1.8 times acceleration. Indeed, we measure about 7 GFLOPS sustained with 12 core execution of L3 local problem, while an out-of-cache problem sustains about 4 GFLOPS, which yields 1.8 time faster execution of the L3 local problem. Note we do not account for any network communication and halo overheads.

VI. Conclusion

We present a novel scheme for accelerating Krylov subspace sparse problems on reconfigurable hardware, specialised to the relatively small bandwidths. We provide an analytic performance model that may assist *a priori* estimations about whether a given problem benefits from the proposed acceration method.

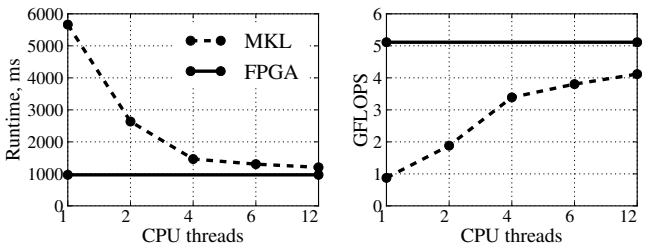

Fig. 7. Performance of large problem, $k = 32$, 1239MB matrix dataset

Our technique provides a complementary acceleration strategy to the standard efforts aiming to saturate the available DRAM bandwidth. Our approach is shown to be bound by silicon resource, enabling it to scale with increase of BRAM capacity in future FPGA generations. As both DRAM bandwidth and LUT resources are not fully utilised with this approach, there is scope for further optimisations.

Acknowledgment

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] R. G. Melhem, "Parallel Solution of Linear Systems with Striped, Sparse Matrices", Parallel Computing, vol 6, no 2, pp. 165-184 (1988).

[2] Y. Zhang, Y. Shalabi, R. Jain, K. K. Nagar, J. D. Bakos, "FPGA vs. GPU for Sparse Matrix Vector Multiply", Proc. IEEE International Conference on Field-Programmable Technology, 2009.

[3] K. K. Nagar, J. D. Bakos, "A Sparse Matrix Personality for the Convey HC-1," Proc. 19th Annual IEEE International Conference on Field-Programmable Custom Computing Machines, 2011.

[4] L. Zhuo and V. K. Prasanna, Sparse Matrix-Vector Multiplication on FPGAs, The 13th ACM International Symposium on Field-Programmable Gate Arrays, 2005.

[5] S. Kestur, J. D. Davis, E. S. Chung, "Towards a Universal FPGA Matrix-Vector Multiplication Architecture", in Proc. of the IEEE Intl Symp on Field Programmable Custom Computing Machines, 2012.

[6] Y. El-Kurdi, D. Fernández, E. Souleimanov, D. Giannacopoulos, W. J. Gross: "FPGA architecture and implementation of sparse matrix-vector multiplication for the finite element method". Computer Physics Communications 178(8): 558-570 (2008).

[7] A. Rafique, G. Constantinides, N. Kapre, "Communication Optimization of Iterative Sparse Matrix-Vector Multiply on GPUs and FPGAs", IEEE Transactions on Parallel and Distributed Systems, 2013.

[8] A. Rafique, N. Kapre and G. Constantinides, "Application Composition and Communication Optimization of Iterative Solvers using FPGAs", International Symposium on Field Programmable Custom Computing Machines, 2013.

[9] Y. Shan, T. Wu, Y. Wang, B. Wang, Z. Wang, N. Xu, and H. Yang, "FPGA and GPU implementation of Large Scale SpMV", in Application Specific Processors (SASP), 2010 IEEE 8th Symposium on. IEEE, 2010, pp. 64–70.

[10] L. Page, S. Brin, R. Motwani, and T. Winograd, "The PageRank citation ranking: Bringing order to the web", tech. rep., Stanford Digital Library Technologies Project, 1998.

[11] F. Alted, "Why modern cpus are starving and what can be done about it", Computing in Science and Eng., vol. 12, no. 2, pp. 6871, Mar. 2010.