Application Composition and Communication Optimization in Iterative Solvers using FPGAs

Abid Rafique
Department of Electrical and Electronic Engineering
Imperial College London
United Kingdom
a.rafique09@imperial.ac.uk

Nachiket Kapre
School of Computer Engineering
Nanyang Technological University
Singapore
nachiket@ntu.edu.sg

George A. Constantinides
Department of Electrical and Electronic Engineering
Imperial College London
United Kingdom
g.constantinides@imperial.ac.uk

Abstract*—We consider the problem of minimizing communication with off-chip memory and composition of multiple linear algebra kernels in iterative solvers for solving large-scale eigenvalue problems and linear systems of equations. While GPUs may offer higher throughput for individual kernels, overall application performance is limited by the inability to support on-chip sharing of data across kernels. In this paper, we show that higher on-chip memory capacity and superior on-chip communication bandwidth enables FPGAs to better support the composition of a sequence of kernels within these iterative solvers. We present a time-multiplexed FPGA architecture which exploits the on-chip capacity to store dependencies between kernels and high communication bandwidth to move data. We propose a resource-constrained framework to select the optimal value of an algorithmic parameter which provides the tradeoff between communication and computation cost for a particular FPGA. Using the Lanczos Method as a case study, we show how to minimize communication on FPGAs by this tight algorithm-architecture interaction and get superior performance over GPU despite of its ∼5× larger off-chip memory bandwidth and ∼2× greater peak single-precision floating-point performance.

***Keywords*-Communication-Avoiding Iterative Solvers, SpMV, Matrix Powers, FPGAs, GPUs**

I. Introduction

A high performance scientific computation operating on large datasets comprises two cost factors 1) *communication* cost of moving data within the memory hierarchy in sequential case or between processors in parallel case 2) computation cost due to *composition* of a sequence of fundamental linear algebra kernels. Take an example of solving a sparse $Ax = b$ using iterative solver like the Conjugate Gradient (CG) [**?**]. Each iteration involves moving the matrix A within the memory hierarchy and then launching a Sparse Matrix-Vector Multiply (SpMV) kernel and a sequence of vector-vector operations. Due to technology scaling, computation performance is increasing at a dramatic rate (flops/sec improves by 59% each year) whereas communication performance is also improving but at a much lower rate (DRAM latency improves by 5.5% and bandwidth improves by 23% each year) [**?**]. As a result of this wide gap, the performance of GPU and FPGAs is

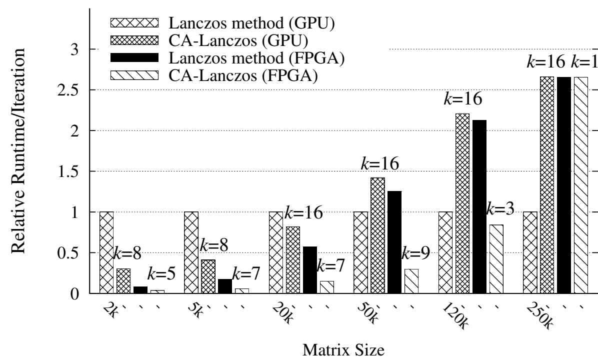


Figure 1. Single-Precision performance comparison of Communication-Avoiding Lanczos (CA-Lanczos) on GPU (Nvidia C2050 Fermi) and FPGA (Virtex6-SX475T) in terms of time/iteration. The Lanczos method from CUSP [**?**] is used as baseline and is run for 64 iterations. The input matrices are banded with a band size $b = 27$. k is an algorithmic parameter whose optimal value is shown at the top of each bar for CA-Lanczos.

bounded from above by their off-chip memory bandwidth, $e.g.$ in iterative solvers, with 2 flops per 4 bytes in SpMV, the maximum theoretical peak performance is 71 GFLOPs and 17 GFLOPs with a less than 7% and 4% efficiency respectively (See Table **??** for peak single-precision GFLOPs and off-chip memory bandwidth).

In order to bridge the gap between computation and communication performance, algorithmic innovations like communication-avoiding iterative solver [**?**] and communication-avoiding QR [**?**] are proposed which trade communication with redundant computation. There are two main challenges associated with this communication-avoiding approach 1) **how to compose the kernels** to keep the computation cost as low as possible 2) **how to select the optimal value of algorithmic parameter** which minimizes overall runtime by providing a tradeoff between computation and communication cost.

In this paper we present a systematic approach to compose multiple linear algebra kernels in communication-avoiding iterative solver and also propose a resource-constrained methodology to select the optimal value of the algorithmic parameter for FPGAs. Our approach is applicable to all scientific computations where the aim is to hide the low off-

Table I
ARCHITECTURAL FEATURES OF FPGA AND GPU.

Device	Tech. (nm)	Die Area (nm^2)	Peak GFLOPs (single-precision)	Memory (On-Chip) RAM	Memory (On-Chip) Registers	Memory BW (On-Chip) RAM	Memory BW (On-Chip) Registers	Memory BW (Off-Chip)
Virtex6-SX475T	40	1806	450 [?]	4 MB	74 KB	5.4 TB/s	36 TB/s	34 GB/s [?]
Nvidia C2050 Fermi	40	529	1030	672 KB	1.7 MB	1.3 TB/s [?]	8 TB/s [?]	144 GB/s

chip memory bandwidth of the FPGAs ($\frac{1}{5}\times$ of the GPU) and to expose large on-chip memory ($2\times$) to share data across kernels and high on-chip communication as well as memory bandwidth ($4\times$) to saturate the floating-point cores. We give you a early preview of the results in Figure **??** where we compare the performance of the Lanczos method, an iterative solver and its variant Communication-Avoiding Lanczos (CA-Lanczos) [?] on GPU and FPGAs. We see three distinct regions.

- For small problems, CA-Lanczos is better than the standard Lanczos method on both GPU and FPGAs.
- For medium to large scale problems, composition of kernels on a GPU increases overall runtime due to off-chip sharing of data (See Section **??**). While performance of standard Lanczos method on FPGAs is worse than the GPU due to its relatively low off-chip memory bandwidth, CA-Lanczos on the other hand shows superior performance due to efficient composition and optimal selection of the algorithmic parameter.
- For extremely large problems, CA-Lanczos is worse on both GPU and the FPGAs due to high computation cost as large datasets are shared using off-chip memory.

We demonstrate that FPGAs are superior over GPU in composing kernels for a range of problem sizes (case 1 and 2 in the list above) where data can be shared across the kernels using on-chip memory (See Section **??**). The main contributions of this paper are

- A time-multiplexed architecture optimized for linear algebra which exploits high on-chip capacity and bandwidth of the FPGA to map all three kernels of communication-avoiding iterative solver.
- A resource-constrained methodology for selecting algorithmic parameter for a particular FPGA.
- A quantitative comparison between FPGA and GPU highlighting their architectural limitations.

II. CASE STUDY: LANCZOS METHOD

We take the Lanczos method to solve the extremal eigenvalue problem of large banded matrices as a case study. Such problems arise in semi-definite optimization solver [?] where we have to solve multiple of them in each iteration. The Lanczos method is at the heart of modern solvers like Conjugate Gradient or Generalized Minimum Residual method (GMRES) [?]. As a result, the results will be generally applicable for all Krylov subspace based iterative solvers [?]. We choose structured banded matrices for two reasons. First, they naturally arise in numerous scientific computations like stencils in solving partial differential equation (PDE) [?] and semi-definite optimization programs [?]. Secondly, computations on these matrices have been used as an architectural evaluation benchmark due to high parallelism and low computational intensity, offering opportunities to exploit on-chip parallelism and challenges with associated memory systems [?].

A. Lanczos Method

Given an $n \times n$ symmetric matrix A with band size b, the Lanczos method [?] applies orthogonal transformations to reduce it to a tridiagonal matrix T_i in an iterative manner

$$Q_i^T A Q_i = T_i \tag{1}$$

where i is the iteration count and $Q_i \in \mathbb{R}^{n \times i}$, $T_i \in \mathbb{R}^{i \times i}$. The eigenvalues of T_i are approximations to the eigenvalues of A with the extremal eigenvalues start converging first after a few iterations. For i iterations, SpMV kernel is launched i times followed by some vector-vector operations to build the Krylov subspace, $i.e.$ span(q, Aq, A^2q, ..., A^iq) [?]. As a result, in the sequential case, the matrix needs to be moved i times within the memory hierarchy making it a memory latency and memory bandwidth bound problem whereas in the parallel case with P processors, $\Omega(i \log P)$ messages are sent thus making it a network latency bound problem [?]. The Lanczos method is shown in Algorithm **??** with $k = 1$.

B. Communication-Avoiding Lanczos

Communication-Avoiding Lanczos (CA-Lanczos) [?] advances by k steps into the Lanczos method by generating k vectors in a single sweep as shown in Algorithm **??**. Compared to the Lanczos method, the communication cost is reduced by a factor of $O(k)$ as the matrix is fetched only once whereas storage and computation cost increases by a factor of $O(k)$. CA-Lanczos comprises three kernels, a matrix powers kernel (Line 2) replacing SpMV to generate k vectors, a Block Gram-Schmidt Orthogonalization (BGS) [?] kernel (Lines $3-4$) to orthogonalize with previous $k+1$ vectors and a QR factorization kernel (Lines $5-10$) to orthogonalize these k vectors with each other. The output of CA-Lanczos is used to form T_i. We now briefly discuss each kernel with its basic linear algebra blocks.

Algorithm 1 Communication-Avoiding Lanczos [?]

Require: $A \in \mathbb{R}^{n\times n}$, $q_0 \in \mathbb{R}^n$ with $\|q_0\|_2 = 1$, $\overline{Q}_1 \in \mathbb{R}^{n\times(k+1)} = 0$
$R_i \in \mathbb{R}^{k\times k}$, $\overline{R}_i \in \mathbb{R}^{k+1\times k}$, $Q_i \in \mathbb{R}^{n\times k}$
1: **for** $i = 0$ to $\frac{i_{max}-1}{k}$ **do**
2: $Qi \leftarrow [A^k q_{ki}, A^{k-1} q_{ki}, \ldots, A q_{ki}]$ —Matrix Powers—
3: $\overline{R}_i \leftarrow \overline{Q}_i^T Q_i$ — BGS1 —
4: $Q_i \leftarrow Q_i - \overline{Q}_i \overline{R}_i$ — BGS2 —

—QR Factorization—
5: **for** $l = 1$ to k **do**
6: $r_{ll} \leftarrow \|q_l\|_2$ —entry at row l and col l of R_i—
7: $q_l \leftarrow \frac{q_l}{r_{ll}}$ —q_l is the lth col of Q_i—
8: **for** $m = l+1$ to k **do**
9: $r_{lm} \leftarrow q_l^T q_m$
10: $q_m \leftarrow q_m - r_{lm} q_l$
11: **end for**
12: **end for**
13: $\overline{Q}_{i+1} \leftarrow [Q_i \; q_{ki}]$
14: $q_{k(i+1)} \leftarrow Q_i(1{:}n, k)$
15: **end for**
16: **return** R_i and $\overline{R_i}$

1) Matrix Powers Kernel: The basic idea is to partition the matrix into blocks and perform k SpMVs on blocks without fetching the block again in the sequential case and performing redundant computation to avoid communication with other processors in the parallel case [?]. In our previous work [?], we also propose a hybrid algorithm shown in Figure **??** which matches the strengths of FPGAs. We load large blocks sequentially but perform computations on sub-blocks in parallel without performing redundant computations but only at the end of block. The computation on each vertex in this graph is $x^T y$ ($x, y \in \mathbb{R}^b$) shown by the number of edges going into each vertex.

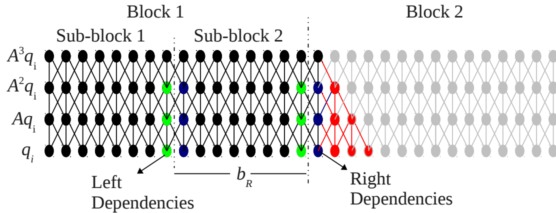


Figure 2. Compute graph of hybrid matrix powers kernel with $k = 3$ operating on matrix A of size $n = 32$ and band size $b = 3$ (tridiagonal). q_i is a vector of length n. All sub-blocks of size $b_R \times b$ are computed in parallel. Left and right dependencies are exchanged after each level in the graph. Red vertices show redundant computation.

2) Block Gram-Schmidt Orthogonalization: Gram-Schmidt Orthogonalization [?] is an approach where a vector x is orthogonalized with vector y such that $x^T y = 0$. Block Gram-Schmidt Orthogonalization (BGS) [?] performs the same computation but for matrices as shown in Lines 3−4 of Algorithm **??**. The computations involved are $x^T y$ and $\alpha x + y$ on Line 3 and 4 respectively with $x, y \in \mathbb{R}^n$.

3) QR Factorization: The k vectors compose a tall-skinny matrix on Line 4 of Algorithm **??** and QR factorization of this matrix is required using some numerically stable method. We use Modified Gram-Schmidt Orthogonalization (MGS) [?] due to its parallel potential and its stability which is proved for iterative solvers [?]. Like BGS, the computations involved are $x^T y$ (Line 6 and 9) and $\alpha x + y$ (Line 7 and 10) with $x, y \in \mathbb{R}^n$. We summarize the basic linear algebra blocks for all three kernels in Table **??**.

Table II
BASIC LINEAR ALGEBRA BLOCKS FOR CA-LANCZOS.

Kernel	Basic Linear Algebra Block
Matrix Powers	$z \leftarrow x^T y$ (x, y$\in \mathbb{R}^b$)
BGS	$z \leftarrow x^T y$ and $y \leftarrow \alpha x + y$ (x, y$\in \mathbb{R}^n$)
QR Factorization	$z \leftarrow x^T y$ and $y \leftarrow \alpha x + y$ (x, y$\in \mathbb{R}^n$)

III. RELATED WORK

A. Communication Optimization

The communication problem in scientific comptuations has historical roots in the *memory wall* [?]. Efforts have been made in the scientific computing community to formulate algorithmic innovations to avoid communication with memory [?] [?]. Demmel *et al.* [?] were the first to trade communication with redundant computation for communication-inensive iterative solvers. Using Communication-Avoiding GMRES (CA-GMRES), they show a 4.3× speedup over GMRES for banded matrices and up to 2.3× for general sparse matrices on an 8-core Intel Clovertown [?]. In FPGA-based iterative solvers, so far the focus is to maximize the use of on-chip memory to load the largest possible matrix at once in order to avoid off-chip memory access. The seminal work is by Boland *et al.* [?] who present an Integer Linear Programming (ILP) framework to optimally utilize the on-chip memory for symmetric banded matrices. However their approach is restricted to small matrix sizes ($n = 8$k with band size $b = 20$ projected to Virtex6-SX475T).

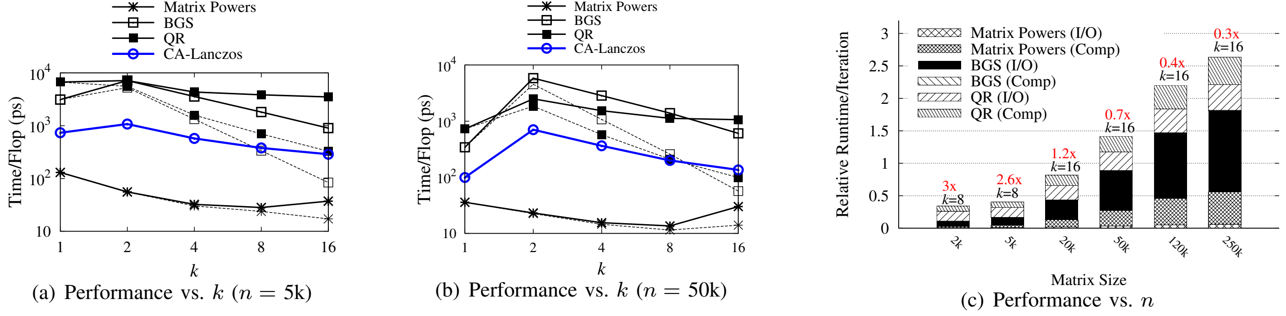
In this paper, we target more general case of large-scale problems where the matrix needs to be accessed from the off-chip memory in each iteration. We combine the latest advancements in communication-avoiding linear algebra and the efficient composition power of FPGAs to minimze the overall runtime in iterative solvers. We optimally trade communication with computation by selecting algorithmic parameter k using a resource-constrained framework.

B. Composition

In FPGAs, composition of kernels can be done in three different ways,

- Fully spatial architecture for each kernel [?].
- A dynamically reconfigurable architecture using full FPGA area for each kernel [?].
- A unified architecture with time-multiplexed scheduling of different kernels.

We use the third approach by designing a high throughput architecture for the primitive linear algebra operations identified in Table **??** and then launch all the kernels in a time-multiplexed fashion. In this way, we avoid the inefficiency of the first approach due to non-overlapped



(a) Performance vs. k (n = 5k) (b) Performance vs. k (n = 50k) (c) Performance vs. n

Figure 3. CA-Lanczos Peformance Analysis on GPU. In Figure **??** and **??**, the dashed lines represent the cost for doing useful operations only . The speedup over the Lanczos method (k = 1) is shown at the top of the bar in Figure **??** for a range of matrices with band size $b = 27$.

kernels and reconfiguration overhead of the second approach (in milliseconds). As our proposed architecture is based on primitive linear algebra operations, it is flexible to support any scientific computation besides iterative solvers. We use on-chip memory to share data across kernels and the only communication involved is to access matrix A from off-chip memory. We compare our work with GPU which also computes kernels in a time-multiplexed fashion but it lacks support of data sharing across kernels using on-chip memory. We show that this results in high computation cost leading to poor performance as compared to FPGAs for a range of problem and band sizes.

IV. Minimizing Communication For GPU

A. Composition and Optimization of Kernels

We briefly discuss CA-Lanczos on GPU. For the Lanczos method, we tailor Conjugate Gradient (CG) implementation from CUSP [**?**] which is an optimized sparse library for GPU. For CA-Lanczos shown in Algorithm **??**, we use a highly optimized parallel matrix powers kernel (Line 2) from our previous work [**?**]. BGS involves multiplication of a short-wide matrix with a tall-skinny matrix (Line 3). `cublassgemm` routine from CuBLAS library suffers from kernel overhead and short vector effects [**?**] for matrices of this aspect ratio as the block sizes are optimized for square matrices. We tune the block size of an open source `magmablas_sgemm` routine from MAGMA library and get a 4× speedup over `cublassgemm`. The QR factorization (Lines 5−12) involves a tall-skinny matrix which involves more communication than any other aspect ratio and therefore we tune a communication-avoiding TSQR routine for GPUs [**?**].

B. Performance Analysis

We use $\frac{time}{flop}$ as the metric to see whether we get any performance improvement using communication-avoiding approach on GPUs. In Figure **??** and **??**, we vary k and show the performance considering useful operations (minus the redundant computation) as well as actual operations. In both problems, we observe a reduction in time for the matrix powers kernel until $k = 8$ and then it starts increasing due to redundant computation which grows quadratically with k [**?**]. On the other hand, we observe two things with BGS and QR kernels. First, they perform more redundant computation as shown by the marked difference between actual and useful performance curves. Secondly, with increasing problem size they start to dominate overall time mitigating the benefits of cost reduction with the matrix powers kernel. We perform source code instrumentation on all these kernels to measure their communication and computation time as shown in Figure **??**. The k vectors generated by the matrix powers kernel are stored in global memory due to low on-chip capacity and since BGS and QR operate on these vectors, the communication (I/O) cost becomes the dominating factor due to low arithmetic intensity in these kernels. Hence, composition of kernels on GPU is inefficient due to this off-chip sharing of data across the kernels. We, therefore, see up to 3×−0.3× speedup over standard Lanczos method from small to large problems.

V. Minimizing Communication For FPGAs

FPGAs have relatively low off-chip memory bandwidth but a large on-chip capacity and high on-chip memory bandwidth as shown in Table **??**. So how we can use this on-chip capacity and memory bandwidth?

A. On-Chip Memory Driven Data Partitioning

We divide CA-Lanczos into three possible scenarios based on the size of matrix A ($n{\times}b$ stored in Compressed Diagonal Storage (CDS) format) and the Lanczos vectors ($\overline{Q}_i \in \mathbb{R}^{n\times(k+1)}$ and $Q_i \in \mathbb{R}^{n\times k}$).

1) $n \cdot b$ is small : Matrix and vectors are stored on-chip.
2) $n \cdot k \ll n \cdot b$: Matrix is stored off-chip whereas the vectors are stored on-chip.
3) $n \cdot k$ is large : Matrix and vectors are stored off-chip.

We show the range of matrices that can be solved with these three scenarios in Figure. **??**.

As the parameter k influences the decision where to store the data, we therefore select it carefully to optimize performance (See Section **??**). From Figure **??** we observe that there is a wide range of matrices where we can keep

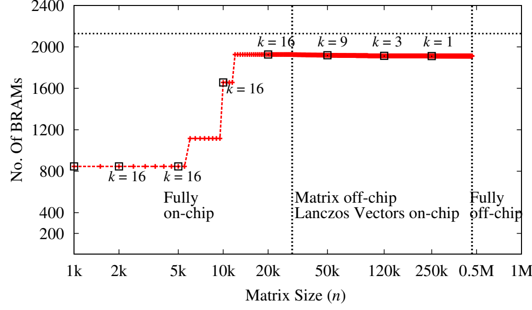


Figure 4. Three distinct scenarios for CA-Lanczos on Virtex6-SX475T (2128 18Kb BRAMs) for problems with band size $b = 27$. The maximum possible value of k is shown for some matrix sizes.

either both the matrix as well as the Lanczos vectors on-chip or only the Lanczos vectors on-chip. In both cases, we eliminate communication with off-chip memory during BGS and QR factorization as they operate on the Lanczos vectors. As the on-chip capacity of new FPGA devices continues to grow (∼2× on Virtex7), the range of matrices where Lanczos vectors can be stored on-chip will be pushed further.

B. Time-Multiplexed FPGA Implementation of CA-Lanczos

CA-Lanczos comprises three kernels, matrix powers, BGS and QR factorization. A fully spatial architecture is not the design choice as the FPGA area gets wasted during different phases. We can populate the FPGA with each kernel and using dynamic reconfiguration we can schedule different kernels but that comes with reconfiguration overhead which is of the same order as the whole application time itself. We design a unified architecture for basic linear algebra blocks identified in Table **??** and then schedule all these kernels in a time-multiplexed fashion. In this way, we re-use logic to enhance the compute capacity of FPGAs for each kernel.

1) Basic Linear Algebra Subroutine (BLAS) Circuit: GPU organizes everything in a simple data-parallel fashion ideally suitable for data-parallel $\alpha x + y$ but inefficient for reduction operations like $x^T y$. We show the performance of these operations on GPU and FPGA in Figure **??** (Assuming x and y are stored in global memory of GPU as in Section **??** and in on-chip memory of FPGAs as shown in Section **??**). How do we achieve this performance? Using superior communication and on-chip memory bandwidth of the FPGAs, we design a high throughput architecture for these compute patterns in Figure **??** which we denote as Basic Linear Algebra Subroutine (BLAS) circuit.

For x and y vectors of length N, both $\alpha x + y$ and $x^T y$ have a sequential latency of $O(N)$ cycles. Our proposed architecture has $O(1)$ cycles latency for $\alpha x + y$ and performs $x^T y$ in $O(\log N)$ cycles. This is highly efficient architecture with an initiation interval of one clock cycle, *i.e.* new set of inputs can be applied at every clock cycle.

2) DataPath: We arrange the BLAS circuits into a large tree reduction architecture as shown in Figure **??**. We aim

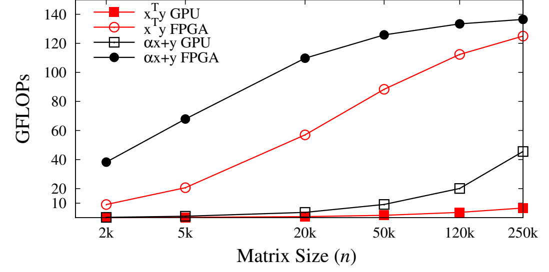


Figure 5. $\mathrm{x}^T\mathrm{y}$ and $\alpha\mathrm{x} + \mathrm{y}$ on GPU and FPGA.

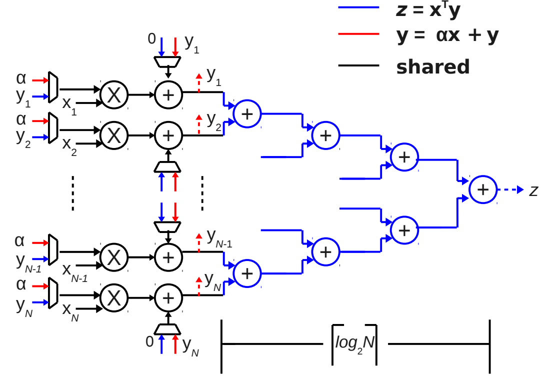


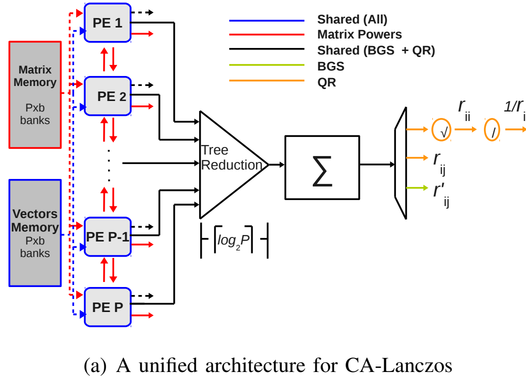
Figure 6. A BLAS Circuit for $z \leftarrow \mathrm{x}^T\mathrm{y}$ and $\mathrm{y} \leftarrow \alpha\mathrm{x} + \mathrm{y}$ where x, y$\in \mathbb{R}^N$. The dotted arrow links show outputs.

to minimize the time required in doing reduction operation which is dominant in all the kernels. Each sub-tree operating on vectors of length b is denoted as a processing element (PE) with its internal architecture shown in Figure **??**. In order to provide the nearest neighbor communication required in mapping the matrix powers kernel compute graph shown in Figure **??**, we also include left and right FIFOs each of length $\frac{b-1}{2}$ in each PE. The data from P PEs is reduced by a adder reduction tree and finally accumulated to support arbitrary $x^T y$ operation (x, y$\in \mathbb{R}^n$). We use dedicated square root and divide units used in QR factorization. The total number of floating-point units are given in Table **??**.

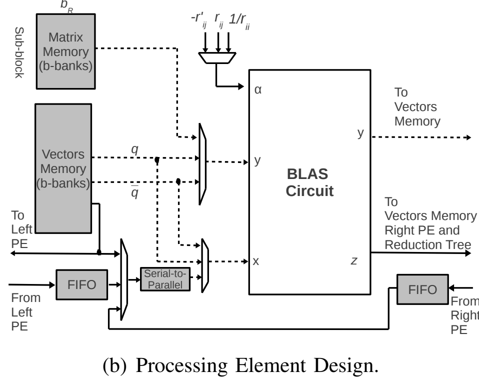
Table III
FLOATING-POINT UNITS FOR CA-LANCZOS.

Floating-Point Unit	Total Number	Latency
Add	$P(2b-1)+P+5$	11
Mult	Pb	8
Div	1	27
Sqrt	1	27

3) Memory Subsystem: We distribute our on-chip memory across our data-path to provide the necessary bandwidth to saturate the floating-point units. The input matrix A is partitioned into sub-blocks where each sub-block is stored in a *Matrix Memory*. The total number of blocks is $N_b = \lceil \frac{n}{Pb_R} \rceil$ where P is the number of PEs and b_R is the number of rows in each sub-block. We discuss in Section **??** how to select b_R. *Matrix Memory* as well as the *Vectors Memory* (for



(a) A unified architecture for CA-Lanczos



(b) Processing Element Design.

Figure 7. A time-multiplexed architecture for CA-Lanczos. The dotted lines show vector links of length b whereas solid line represents scalars.

Lanczos vectors) is distributed across P PEs with each PE accessing them as a bank of width b as shown in Figure **??**. The memory used in single-precision implementation of CA-Lanczos in terms of BRAMs (18kbit each) is given by

$$\textit{Matrix Memory} = Pb\left\lceil \frac{32b_R}{18 \times 1024} \right\rceil \quad (2)$$

$$\textit{Vectors Memory} = 2Pb\left\lceil \frac{32\left\lceil \frac{n}{Pb}\right\rceil (k+1)}{18 \times 1024} \right\rceil \quad (3)$$

$$\textit{FIFOs} = 2P\left\lceil \frac{32(b-1)}{2 \times 18 \times 1024} \right\rceil \quad (4)$$

C. Compute Schedule

The architecture shown in Figure **??** implements CA-Lanczos in a time-multiplexed fashion. We show the compute schedule for CA-Lanczos in Algorithm **??** and highlight the blocks used in each kernel in Figure **??**.

Algorithm 2 Compute Schedule

Matrix Powers Kernel
Step 1 : Load a block of matrix A and divide it into sub-blocks to be stored in *Matrix Memory* of each PE.
Step 2 : Configure each PE to compute $z \leftarrow x^T y$ (x, y$\in \mathbb{R}^b$). Launch a operation every clock cycle to compute all b_R vertices in each sub-block of Figure **??**. Store the results in *Vectors Memory* (Q_i). Compute k levels of the compute graph in the same fashion. Go to Step 1 until all blocks $N_b = \lceil \frac{n}{Pb_R} \rceil$ of A are not finished.
Block Gram-Schmidt Orthogonalization
Step 3 : In each PE, load components of vectors b at a time from Q_i and $\overline{Q}_i$ shown as q and $\overline{q}$ respectively in Figure **??**. Compute dot product and accumulate the results from P PEs.
Step 4 : Go to Step 3 and load next b components $\beta = \lceil \frac{n}{Pb} \rceil$ times until we reach at the end of the vectors. Compute all dot products at Line 2 of Algorithm **??**. Save the accumulator output in $\overline{R}_i$.
Step 5 : Compute Line 3 of Algorithm **??** by configuring each PE to compute $y \leftarrow \alpha x + y$. Here α is $-\overline{r}_{ij}$ $i.e$ entry of matrix $\overline{R}_i$ scanned as column-major order. x is the vector from $\overline{Q}_i$ whereas y is from Q_i.
QR Factorization
Step 6 : Compute QR factorization like BGS.

D. Performance Model

We build an analytical model for overall latency which we will use in a resource-constrained framework to select an optimal value of k. The latencies of single-precision floating point multiplier, adder, divider and sqrt and accumulator are denoted by l_M, l_A, l_D , l_S and l_{acc} respectively. We use Xilinx Coregen for these operators and their latencies are given in Table **??**. Using the compute schedule in Algorithm **??**, we show the latencies (in cycles) of all the kernels in Table **??**.

Table IV
CA-LANCZOS FPGA ANALYTICAL PERFORMANCE MODEL.

Kernel	Latency
Matrix Powers	$l_{BC} = l_M + l_A + l_A\lceil\log_2 b\rceil$ $l_{MP} = N_b k(b + l_{BC} + b_R - 1)$
BGS	$l_{BGS,1} = (k^2+1)(l_{BC} + \beta + l_{red})$ $l_{BGS,2} = (k^2+1)(l_M + l_A + \beta)$ $\beta = \lceil\frac{n}{Pb}\rceil$, $l_{red} = \lceil\log_2 P\rceil + l_{acc}$
QR Factorization	$l_{QR,1} = (k+1)(l_{BC} + l_{red} + l_M$ $+l_A + 2\beta + l_S + l_D)$ (Outer Loop) $l_{QR,2} = k\frac{(k+1)}{2}(l_{BC} + l_{red} + l_M$ $+l_A + 2\beta)$ (Inner Loop)
CA-Lanczos	$L = l_{MP} + l_{BGS,1} + l_{BGS,2}$ $+l_{QR,1} + l_{QR,2}$

E. Resource-Constrained Framework

In order to select k which trades communication with computation and gives optimal performance, we develop the following resource-constrained framework.

- Find the maximum number P of PEs that can be synthesized within the FPGA for a given band size b.
- Find the memory bandwidth required to saturate these P PEs. Partition the available on-chip memory such a way that Lanczos vectors can fit on-chip.
- Pick k and b_R based on the following constrained optimization problem

$$\min_{k,P,b_R} \frac{L(k,P,b_R)}{k(2nb+7n)}$$

subject to

$$\begin{aligned} M(P,k,b_R) &\leq \text{FPGA}_{BRAMs} \\ R(P) &\leq \text{FPGA}_{Logic} \\ k &\leq 16 \\ k &\leq 2\frac{b_R}{b-1} \end{aligned} \quad (5)$$

Referring to Equation (**??**), $\frac{time(cycles)}{flop}$ is our objective we want to minimize. $L(k,P,b_R)$ corresponds to the total compute latency per CA-Lanczos iteration shown in Table **??** and $k(2nb+7n)$ is the total number of flops in k iterations of

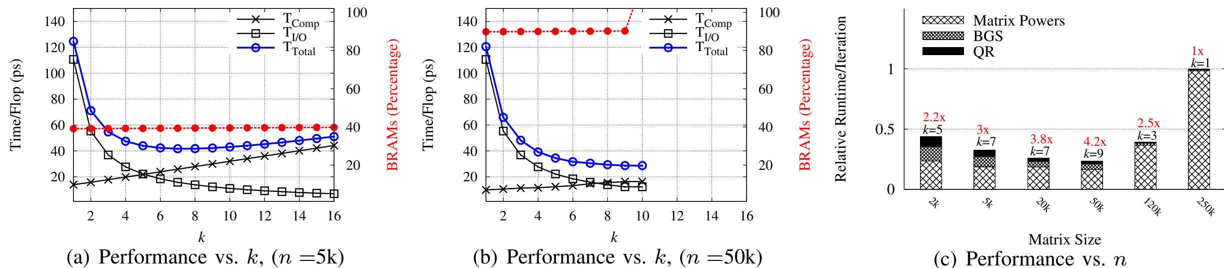


Figure 8. CA-Lanczos Performance Analysis on FPGA. The value of k is selected using the resource-constrained framework in Figure **??** and Figure **??**. The speedup over the Lanczos method ($k = 1$) on FPGA is shown in Figure **??** for a range of matrices with the band size $b = 27$.

the Lanczos method. $M(P, k, b_R)$ is the number of BRAMs required and $R(P)$ is a vector containing the number of resources in terms of LUTs, FFs and DSP48Es from Table **??**. $k \leq 16$ is an algorithmic constraint due to loss of orthogonality in CA-Lanczos [**?**]. The last constraint is to allow the partition size large enough to ensure only nearest neighbor communication at the boundaries of blocks in the matrix powers kernel shown in Figure **??**.

VI. Experimental Setup

The experimental setup comprises Virtex6-SX475T FPGA and an Nvidia C2050 Fermi device with their architectural features shown in Table **??**. We use $\frac{time}{iteration}$ as our metric relative to the Lanczos method on GPU which is used as a baseline. We use banded matrices with common band sizes that commonly arise in stencil computation in practical applications. We use Xilinx Coregen single-precision floating-point cores for our hardware operators. Our placed and routed design has an operating frequency of 258 MHz. We show the resource utilization for different band sizes in Table **??**. We maximize the use of DSP48Es for high performance. We do not show BRAMs here because they also depend on the value of k and n. We use 50% of the maximum possible I/O bandwidth of the FPGA in order to compute the communication cost.

Table V
FPGA Resource Utilization.

Band Size	DSP48Es (%)	LUTs (%)	FFs (%)	PEs
3	99.6	70.3	41.3	95
9	97.5	68.8	40.4	31
27	94.3	66.6	39.0	10

VII. Evaluation

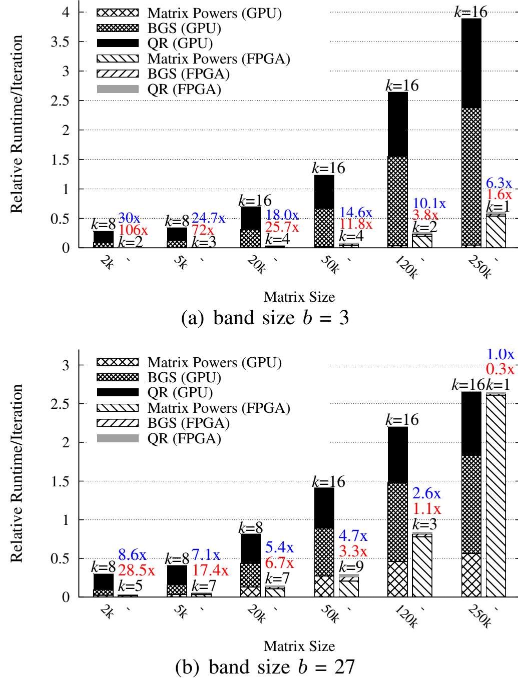
We first evaluate the impact of k on FPGA performance and then compare the results of CA-Lanczos on FPGA and GPU.

A. Impact of k on FPGA Performance

We show $\frac{time}{flop}$ for computation (T_{Comp}) and communication ($T_{I/O}$) in Figure **??**. We select the value of k which minimizes total time, *i.e.* $T_{Total} = T_{Comp} + T_{I/O}$. Ideally, we should see a reduction in this cost by k as the matrix is fetched only once to generate k vectors. However from Figure **??**, we see the total cost decreases until $k = 7$ and then it increases due to computation cost ($O(nk^2)$ work in BGS and QR). We see a minima at $k = 7$ where we get a $\sim 3\times$ performance improvement over the standard Lanczos method ($k = 1$). In Figure **??**, the value of k is restricted due to BRAMs as beyond $k = 9$ the vectors can no more be stored on-chip. By picking the value of k using this framework, we optimally trade communication with redundant computation to minimize the overall cost. As a result, for a range of problem sizes, we get $1\times - 4.2\times$ speedup over the FPGA-based standard Lanczos method shown in Figure **??**.

B. Performance Comparison with GPU

We compare the performance of our proposed design with GPU in Figure **??** for $b = 3$ and $b = 27$ showing runtime breakdown of each kernel. From these results, we find out that the matrix powers kernel on GPU is efficient as compared to FPGAs because of $\sim 5\times$ larger off-chip bandwidth to fetch the matrix A. However, the communication-avoiding approach is not as useful on GPU as it is on FPGA due to two reasons. First the composition of kernels require sharing data *i.e.* vectors through global memory. Even if they can be stored on-chip for small problem sizes, they are distributed across all SMs. The communication between different SMs is required in reduction operations involved in BGS and QR, and as this communication is only possible through global shared memory, it therefore increases communication cost (See Section **??**). Secondly, the matrices involved in BGS and QR are either short and fat or long and thin and both of these aspect ratios are not suitable on GPUs [**?**]. As a result CA-Lanczos on GPU is up to $\sim 3\times$ slower than the standard Lanczos method. On the other hand, as the vectors are stored on-chip in FPGAs, our architecture expliots high on-chip bandwidth and communication-rich fabric of FPGAs to keep the computation cost of BGS and QR kernels as low as possible as shown in all of our results. For small band size $b = 3$ and small to medium problem sizes where vectors can be stored fully on-chip, we get orders of magnitude speedup and for the largest problem size FPGA is $1.6\times$ as fast as the standard Lanczos method from CUSP. However, for large matrix and band sizes, the problem becomes communication-bound and FPGA is up to



(a) band size b = 3

(b) band size b = 27

Figure 9. Performance Comparison of CA-Lanczos on GPU (left bar) and FPGA (right bar). The optimal value of k is shown at the top of the bars. The speedup factor of FPGAs over GPU is also shown in red for standard Lanczos method on GPU and in blue for CA-Lanczos on GPU.

$\sim$3$\times$ slower.

VIII. Conclusion

Communication-avoiding algorithms are an alternative to multi-level cache hierarchy where we trade communication with redundant computation. In current GPU architectures, composition of different kernels involves sharing data using off-chip global memory and this increases the computation cost to the point where we do not see any benefit of using communication-avoiding approach. In FPGAs, by explicitly sharing data across kernels using on-chip memory and desiging an architecture which exploits the on-chip memory and communication bandwidth, we keep the computation cost as low as possible. We show how to pick algorithmic parameter to optimally trade communication with redundant computation. Using CA-Lanczos as a case study, we show upto 4.2$\times$ performance improvement over vanilla algorithm on FPGAs, up to orders of magnitude speedup over GPU for small problems and a single-digit performance improvement for medium to large-scale problems. For large problems where we can not store data on-chip, we see a $\sim$0.3$\times$ speedup as the problem becomes communication-bound.

Acknowledgment

We would like to thank Michael Anderson, PAR Lab, University of California, Berkeley for providing us the GPU source code for QR factorization [**?**]. Also, we would like to thank Magnus Gustafsson from Uppsala University, Sweden for providing us the MATLAB code for numerical experiments with CA-Lanczos.

References

[1] G.H. Golub and C.F. Van Loan *Matrix Computations* (3rd Edition). The Johns Hopkins University Press, Baltimore, Maryland, 1996.

[2] M. Snir and S. Graham, *Getting up to speed: The Future of Supercomputing*, National Research Council, 2004.

[3] M. Hoemmen, *Communication-Avoiding Krylov subspace methods*, PhD Thesis. University of California Berkeley, 2010.

[4] E. Klerk, *Exploiting special structure in semidefinite programming: A survey of theory and applications*, European Journal of Operational Research. 201(1): 1-10, Elsevier, 2010.

[5] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf and K. Yelick *Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures*, In Proceedings of ACM/IEEE conference on Supercomputing, 2008.

[6] A. Rafique, N. Kapre and G. Constantinides *Avoiding communication in GPU and FPGA-based sparse iterative solvers: An algorithm-architecture interaction*, to be submitted in IEEE Transactions on Parallel and Distributed Systems.

[7] W.A. Wulf and S.A McKee *Hitting the memory wall: Implications of the obvious*, ACM SIGARCH Computer Architecture News, vol. 23(1), pp. 20-24, 1995.

[8] S.A. Toledo *Quantitative performance modeling of scientific computations and creating locality in numerical algorithms*, PhD. Thesis, Massachusetts Institute of Technology, 1995.

[9] M. Strout, L. Carter and J. Ferrante *Rescheduling for locality in sparse matrix computations*, ICCS, 137-146, 2001.

[10] D. Boland and G. Constantinides *Optimising memory bandwidth use for matrix-vector multiplication in iterative methods*, In Proceedings of 6th International Symposium on Applied Reconfigurable Computing, pp. 169-181, 2010.

[11] M. Gustaffson, J. Demmel and S. Holmgren *Numerical evaluation of the Communication-Avoiding Lanczos algorithm*, Technical Report, University of California Berkely, 2012.

[12] A. Greenbaum, M. Rozložník and Z. Strakoš *Numerical behaviour of the modified Gram-Schmidt GMRES implementation*, BIT Numerical Mathematics, 37(3), pp. 706-719, 1997.

[13] N. Kapre, A. Dehon *SPICE^2 : Spatial Processors Interconnected for Concurrent Execution for Accelerating the SPICE Circuit Simulator Using an FPGA*, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol.31(1), pp. 9-22, 2012.

[14] J. Villasenor, B. Schoner, K.N. Chia, C. Zapata, H.J. Kim, C. Jones, S. Lansing, and B. Mangione-Smith, *Configurable computing solutions for automatic target recognition*, in Proceedings of IEEE Symposium on FPGAs for Custom Computing Machines, pp.70-79, 1996.

[15] N. Bell and M. Garland *Cusp: Generic parallel algorithms for sparse matrix and graph computations.*, Available at: code.google.com/p/cusp-library, 2010.

[16] M. Anderson, G. Ballard, J. Demmel and K. Keutzer *Communication-avoiding QR decomposition for GPUs*, In Proceedings of IPDPS, pp. 48-58, 2011.

[17] S. Kestur, J.D. Davis and O. Williams *Blas comparison on fpga, cpu and gpu*, In Proceedings of the 2010 IEEE Annual Symposium on VLSI, pp. 288-293, 2010.

[18] Sundararajan, P. *High Performance Computing using FPGAs*, www.xilinx.com/support/documentation/white_papers/wp375_HPC_Using_FPGAs.pdf, 2010.

[19] V. Volkov, *Better performance at lower occupancy*, in Proceedings of the GPU Technology Conference, GTC, 2010.