This document is downloaded from DR-NTU (https://dr.ntu.edu.sg)
Nanyang Technological University, Singapore.

Communication Optimization of Iterative Sparse Matrix-Vector Multiply on GPUs and FPGAs

Rafique, Abid; Constantinides, George A.; Kapre, Nachiket

2015

Rafique, A., Constantinides, G. A., & Kapre, N. (2015). Communication Optimization of Iterative Sparse Matrix-Vector Multiply on GPUs and FPGAs. IEEE Transactions on Parallel and Distributed Systems, 26(1), 24-34.

https://hdl.handle.net/10356/81168

https://doi.org/10.1109/TPDS.2014.6

© 2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses, in any current or future media, including reprinting/republishing this material for advertising or promotional purposes, creating new collective works, for resale or redistribution to servers or lists, or reuse of any copyrighted component of this work in other works. The published version is available at: [http://dx.doi.org/10.1109/TPDS.2014.6].

Downloaded on 23 Jul 2024 05:04:39 SGT

Communication Optimization of Iterative Sparse Matrix-Vector Multiply on GPUs and FPGAs

Abid Rafique, George A. Constantinides, *Senior Member, IEEE* Nachiket Kapre, *Member, IEEE*

Abstract—Trading communication with redundant computation can increase the silicon efficiency of FPGAs and GPU in accelerating communication-bound sparse iterative solvers. While k iterations of the iterative solver can be unrolled to provide $O(k)$ reduction in communication cost, the extent of this unrolling depends on the underlying architecture, its memory model and the growth in redundant computation. This paper presents a systematic procedure to select this algorithmic parameter k, which provides communication-computation tradeoff on hardware accelerators like FPGA and GPU. We provide predictive models to understand this tradeoff and show how careful selection of k can lead to performance improvement that otherwise demands significant increase in memory bandwidth. On an Nvidia C2050 GPU, we demonstrate a $1.9\times-42.6\times$ speedup over standard iterative solver for a range of benchmarks and that this speedup is limited by the growth in redundant computation. In contrast, for FPGAs, we present an architecture-aware algorithm that limits off-chip communication but allows communication between the processing cores. This reduces redundant computation and allows large k and hence higher speedups. Our approach for FPGA provides a $0.3\times-4.4\times$ speedup over same generation GPU device where k is picked carefully for both architectures for a range of benchmarks.

Index Terms—Iterative Numerical Methods; Spare Matrix−Vector Multiply; Matrix Powers Kernel; Field Programmable Gate Arrays (FPGAs); Graphics Processing Units (GPUs)

1 Introduction

The cost of a high performance scientific computation operating on large datasets consists of two factors (1) *computation* cost of performing floating-point operations (2) *communication* cost (both latency and bandwidth) of moving data within the memory hierarchy in sequential case or between processors in parallel case. One of the communication-intensive scientific computations is an iterative solver used for solving large-scale sparse linear system of equations ($Ax = b$) and eigenvalue problems ($Ax = \lambda x$) [1]. The solution of these problems is computed from a Krylov subspace span(x, Ax, A^2x, ..., A^rx) [1], where a new vector is generated in each iteration. Iterative solvers are challenging to accelerate as they spend most of the time in communication-bound computations, like sparse matrix-vector multiply (SpMV) and vector-vector operations (dot products and vector additions). Additionally, the data dependency between these operations hinder overlapping communication with computation. For large-scale problems where the matrix A does no fit on-chip, no matter how much parallelism can be exploited to accelerate SpMV, the performance of the iterative solver is bounded by the available off-chip memory bandwidth, $e.g.$ with 2 flops per 4 bytes (single-precision) in SpMV, the maximum theoretical performance is 71 GFLOPs on an Nvidia C2050 GPU and 17 GFLOPs on a Virtex6 FPGA. This results in less than 7% and 4% efficiency of GPU and FPGA respectively (See Table 1 for peak single-precision GFLOPs and off-chip memory bandwidth).

The communication problem is connected to the *memory wall* problem [2]. Due to technology scaling, computation performance is increasing at a dramatic rate (flops/sec improves by 59% each year) whereas communication performance is improving but at a much lower rate (DRAM latency improves by 5.5% and bandwidth improves by 23% each year) [3]. It is a well-known idea to formulate algorithmic innovations that hide memory latency and optimize memory bandwidth [4] [5] [6]. For iterative solvers, Demmel *et al.* [7] trade communication with redundant computation by replacing k SpMVs with the matrix powers kernel. The key idea is to partition the matrix into blocks and performs 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 this way the communication cost is reduced by $O(k)$ at the expense of increase in redundant computation. They show that such an approach can minimize latency in a grid [7] and both latency and bandwidth on a multi-core CPUs [8] to give up to $4\times$ and $4.3\times$ speedup respectively over k SpMVs for banded matrices. While

- *Abid Rafique and George A. Constantinides are with the Department of Electrical and Electronic Engineering, Imperial College London, UK.*
- *Nachiket Kapre is with School of Computer Engineering, Nanyang Technological University, Singapore.*

TABLE 1
Architectural Features of FPGA and GPU (On-Chip memory of GPU refers to shared memory).

Device	Tech. (nm)	Peak GFLOPs (single-precision)	Memory (On-Chip)		Memory BW (On-Chip)		Memory BW (Off-Chip)	Freq. MHz
			Total. RAM	Registers	RAM	Registers		
Virtex6-SX475T	40	450 [18]	4 MB	74 KB	5.4 TB/s	36 TB/s	34 GB/s [18]	258
Nvidia C2050 Fermi	40	1030	672 KB	1.7 MB	1.3 TB/s [19]	8 TB/s [19]	144 GB/s	1150

the communication-avoiding approach is promising, there are two main challenges associated with this communication-avoiding approach on parallel architectures (1) how to keep the redundant computation as low as possible to minimize the computation cost and (2) how to select the optimal value of the algorithmic parameter k, which minimizes overall runtime by providing a tradeoff between computation and communication cost.

In this paper, we show how we can increase the silicon efficiency of FPGAs and GPUs in accelerating communication-bound sparse iterative solver. As a motivation, we show a tradeoff between computation and communication cost for FPGA and GPU to minimize overall runtime as shown in Figure 1. We observe that in standard iterative solver (k equal to 1), the communication cost is higher on FPGA as compared to GPU due to marked difference in off-chip memory bandwidth as shown in Table 1. However, we see a unique value of k, which trades communication with redundant computation to reduce overall cost and that this value needs to be selected carefully for each architecture. Additionally, we observe that unlike GPU, the computation cost does not grow in FPGAs allowing larger values of k, which leads to higher performance.

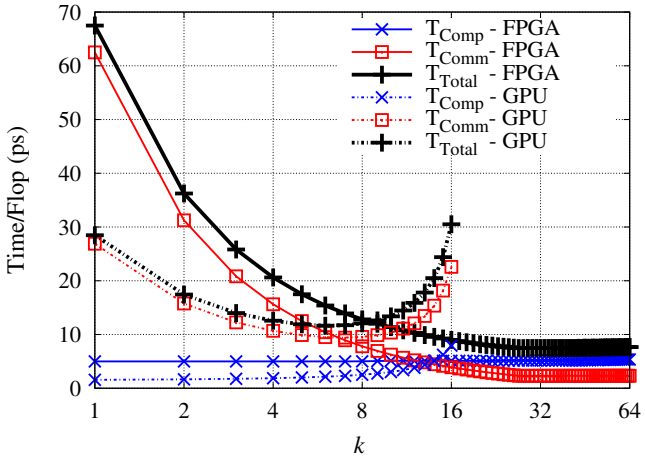


Fig. 1. Computation-communication tradeoff for a banded matrix with band size 27 and n = 1M on a Virtex6-SX475T FPGA and C2050 Fermi GPU.

The main contributions of this paper are:

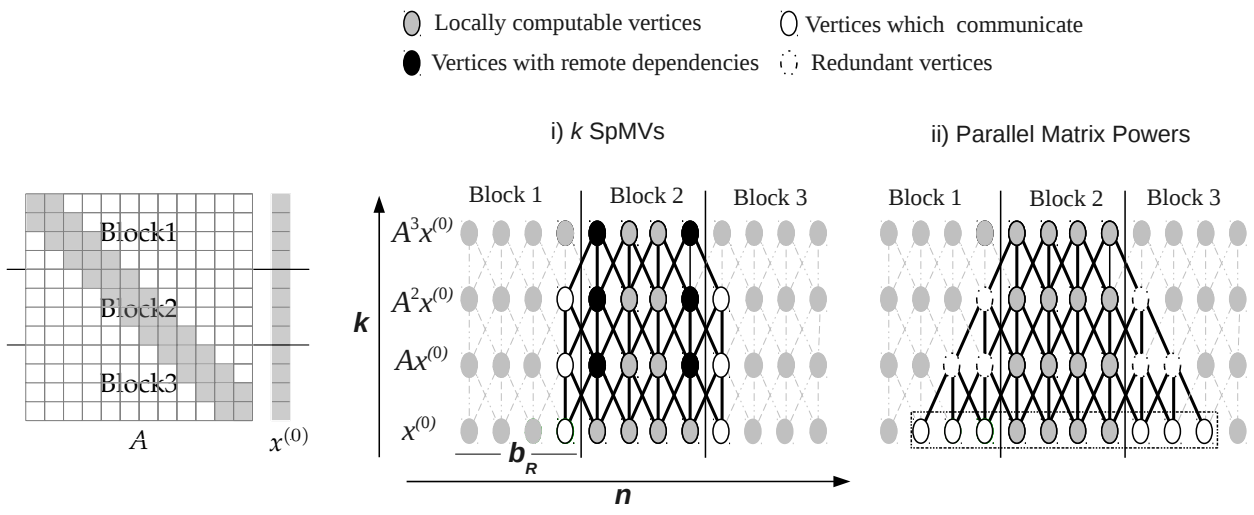
1) Communication optimization within the memory hierarchy of a single stream multiprocessor (SM) as well as between different SMs while mapping the matrix powers kernel to a GPU. As a result of these optimizations, we show $1.9\times-42.6\times$ speedup over k SpMVs from CUSP library [15] for a range of randomly generated banded matrices.
2) An architecture-aware matrix powers kernel that matches the strength of the FPGAs to avoid redundant computation and a resource-constrained methodology to pick k for a particular FPGA.
3) A unified predictive model of the matrix powers kernel for GPU and FPGA, which helps us understanding communication-computation tradeoffs in selecting the algorithmic parameter k. Using the steepest ascent approach, we also show which aspect of future GPU and FPGA architectures need to be improved to achieve higher performance.
4) For a range of problem sizes, a quantitative comparison of the matrix powers kernel on FPGA shows $0.3\times-3.2\times$ and $1.7\times-4.4\times$ speedup over GPU for largest and smallest band sizes respectively.

The paper is organized as follows. Section 2 provides the necessary background about the structurally sparse matrices used in this work. It also summarizes the matrix powers kernel based on [7] (See Section 1 of supplementary file for details). Section 3 shows how we map the parallel matrix powers kernel on GPUs along with discussion about the predictive model. Section 4 presents the proposed matrix powers kernel specifically targeting FPGAs. It also discusses custom architecture on FPGA and a resource-constrained methodology to select an optimal value of k. Section 5 introduces the evaluation methodology and performance of FPGA and GPU is compared in Section 6. Section 7 provides architectural insight and final conclusion is in Section 8.

2 Background

2.1 Matrix Structure

In this work, we target very large ($n \sim 10^6$) structured sparse banded matrices. The banded matrices are stored in *band storage* format where an $n\times n$ matrix of band size b is stored as an $n\times b$ matrix [13]. We choose this band structure for two main reasons. First, computations on such matrices have been used as



(a) Banded matrix with band size b =3 and $n = 12$.

(b) k SpMVs vs. Matrix Powers Kernel

Fig. 2. k SpMVs vs. parallel matrix powers kernel for a matrix with size $n = 12$, band size $b = 3$, number of levels $k = 3$ and number of blocks $N_q = 3$.

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 [14]. Secondly, they naturally arise in numerous scientific computations like stencils in partial differential equation (PDE) solvers [10], semi-definite optimization problems [11] and model predictive control [12]. Although stencil computation in PDE solvers may not explicitly store data in band storage form but in the other two applications, the banded matrix A is explicitly formed.

2.2 Matrix Powers Kernel

Given an $n \times n$ sparse matrix A, a dense vector $x^{(0)}$ of length n, the matrix powers kernel is computed as

$$x^{(i)} = Ax^{(i-1)} \qquad 1 \leq i \leq k \tag{1}$$

The computation in Equation (1) can be unrolled for k iterations as a graph as shown in Figure 2(b) for an example tri-diagonal matrix A (See details of the graph notation in Section 1 of the supplementary file). Each vertex of the graph represents vector entry $x_j^{(i)}$ and there are n vertices for $j = 0, 1, \ldots, n-1$ entries for each level $i = 0, 1, \ldots, k$. The vertices of new vector $x^{(i)} = Ax^{(i-1)}$ can be computed by multiplying the entries of the previous vector with the corresponding entries of the matrix as shown by the edges. To parallelize the matrix powers kernel, the graph can be partitioned into N_q blocks where each block q can be mapped on a single processor. In case, where we generate k vectors using repeated SpMV, we need to synchronize after each step to get the corresponding entries of the previous vector that are computed by the neighbouring processors as shown by white vertices in Figure 2(b)(i). The synchronization in k SpMVs is avoided by the parallel matrix powers kernel, which trades communication with redundant computation. The key idea is to determine the dependency chain of each partition q for computing the entries of the kth vector as shown by the bounded box in Figure 2(b)(ii). The vertices that are required from the neighbouring processors are fetched at once as shown by the white vertices in Figure 2(b)(ii). To avoid communication with neighbours, the redundant computation is then performed at each step (dotted white vertices). While this is well and good, two factors are crucial for optimal performance. First, it is desired to keep the *surface to volume ratio* ($\frac{\text{Redundant Flops}}{\text{Useful Flops}}$) as low as possible by efficient partitioning of the matrix. Secondly, the value of k needs to be picked carefully as redundant flops grow as $O(k^2b^2)$ [7].

3 Matrix Powers Kernel on a GPU

While mapping the matrix powers kernel on a GPU, we want to answer two important questions (1) How to optimally utilize the current GPU memory subsystem and pick the algorithmic parameter k to achieve the desired performance for a particular architecture? (2) How can current GPU architecture be changed to enhance the performance of iterative solvers? We first present the current GF100 GPU architecture and then discuss different optimization techniques that lead to high throughput. We then present an analytical model to predict and understand the performance of the matrix powers kernel on any GPU device (model parameters are obtained using micro-benchmarks). We use the same model to select k for current GPU architectures (See Section 6.1) and also make architectural projections to obtain a desired performance with future devices (See Section 7).

3.1 GPU Architecture

We select the Nvidia GF100 variant C2050 GPU, which is intended for high-performance numerical computing [14] [19]. A simplified architectural description of

the GPU is shown in Figure 3 highlighting memory hierarchy as well as capacity, bandwidth and latency of each memory.

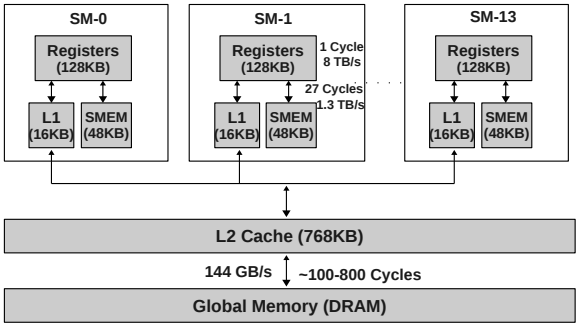


Fig. 3. GPU Architecture (Nvidia C2050 Fermi).

The GPU comprises 14 streaming multiprocessors (SMs) each operating at 1.15 GHz. Each SM has 32 floating-point cores capable of performing 1 single-precision flop/cycle reaching a peak throughput of 1.03 TFLOPs for single-precision and 515 GFLOPs for double-precision. Tasks are scheduled on GPU as *thread blocks*. Each thread block can run independently on an SM without any communication with other SMs during a single parallel task, $e.g.$ each SM can compute k SpMVs for a single block as shown in Figure 2(b)(ii) for $q = 2$.

3.2 Partitioning Strategy−One Partition Per Thread Block

Each of the N_q blocks within the matrix powers kernel is mapped to a thread block and all these thread blocks are computed independently in parallel and in any order. The size of each block is $b_R \times b$ where b_R is the number of rows in each block. However, the actual size is $(b_R + k(b-1)) \times b$ as we need some entries from neighbouring blocks for redundant computation that helps in avoiding communication. We assign each vertex to a single thread, which performs serial reduction to compute the dot product of the row with b components of vector $x^{(i)}$. If N_T denotes the number of threads, we can represent partition size b_R and the total number of partitions as

$$b_R = N_T - k(b-1) \quad (2)$$

$$N_q = \left\lceil \frac{n + b_R - 1}{b_R} \right\rceil \quad (3)$$

3.3 GPU Optimizations

To exploit memory hierarchy of the GPU for fast access of the matrix and vector partitions, we explore three possible mappings of the matrix powers kernel as shown in Table 2. We take a matrix of size $n = 10^6$ with band size $b = 9$, number of threads $N_T = 512$ and number of levels $k = 8$ and evaluate the performance of these mappings. We select the values of N_T and k to show performance scaling and later on show how these values impact performance and need to be picked carefully. We choose `spmv_dia_kernel` from CUSP library [15] as baseline to perform k SpMVs with a performance of 34.8 GFLOPs. We now briefly discuss the three possible GPU optimizations.

TABLE 2
Single-Precision Parallel Matrix Powers Kernel Parallel Mapping on C2050 GPU (n =1M, b =9, k =8). Efficiency = $\frac{\text{Sustained GFLOPs}}{\text{Peak GFLOPs}}$.

	Global Memory	Shared Memory	Reg.	Sustained GFLOPs	Efficiency
k SpMVs [15]	A, $x^{(i)}$			34.8	3.3%
Matrix Powers (Thread Blocking)	A	$x^{(i)}$		63	6.12%
Matrix Powers (Thread Blocking + Cache Blocking)		A, $x^{(i)}$		97.6	9.4%
Matrix Powers (Thread Blocking + Reg. Blocking)		$x^{(i)}$	A	123	11.9%

3.3.1 Thread Blocking (63 GFLOPs)

Each thread within the thread block is responsible for computing a single entry of the vector $x^{(i)}$ from the entries of matrix A and vector $x^{(i-1)}$. We, therefore, only store $N_T = b_R + k(b-1)$ components of the vector $x^{(i-1)}$ within the shared memory of each SM. To avoid communication with other SMs, the entries from neighbouring blocks are pre-fetched. As the entries of the matrix A are not modified and do not require inter-SM synchronization at each level, we can access A from global memory (See Listing 1 of supplementary file).

3.3.2 Thread Blocking + Explicit Cache Blocking (97.6 GFLOPs)

In this case, in addition to thread blocking, we also use *explicit cache blocking* to store the partition of matrix A into on-chip shared memory of each SM (See Listing 2 of the supplementary file). Using this approach, we not only avoid communication between different SMs but also with the global memory as well. As a result, we see a significant performance improvement over k SpMVs.

3.3.3 Thread Blocking + Register Blocking (123 GFLOPs)

GPUs have an inverse memory hierarchy [19], $i.e.$ registers have relatively large capacity as compared to L1 cache/shared memory and L2 cache. Also registers have low latency (∼1 cycle) compared to shared memory (∼27 cycles). To get high throughput, they have been recently used to block matrices arising in small linear algebra problems with high arithmetic intensities [16]. In the matrix powers kernel, each matrix partition can be blocked within the registers of the SM (See Listing 3 of the supplementary file). We store the $N_T \times b$ partition matrix in a row cyclic

distributed fashion within these registers, $i.e.$ each thread can store a single row of length b. The vector partition is not register-blocked as its entries need to be shared among different threads and is, therefore, kept in the shared memory. We show the performance of these optimizations in Figure 4.

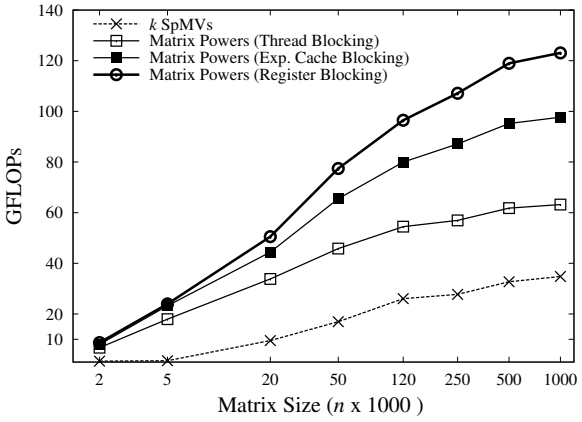


Fig. 4. GPU Optimizations ($b = 9$, $N_T = 512$, $k = 8$).

The matrix powers kernel with thread blocking and register blocking gives higher throughput as we not only avoid communication within different SMs and within memory hierarchy of a single SM but also utilize low latency and high bandwidth memories of GPU. We see more than 3.5× speedup over k SpMVs for large matrices and this speedup is even more pronounced for small matrices with higher value of k (See Table 6).

3.4 Modelling Performance

Mapping the matrix powers kernel on GPU and selecting optimal k to trade communication with computation is not obvious due to the complex GPU architecture. To understand and predict the performance of the matrix powers kernel on the GPU, we characterize our discussion using bandwidth and latency of entire GF100 memory hierarchy. To that end, we assume data is either stored in global or shared memory and use two simple models to predict GPU performance. Our model is based on the LogP model used for distributed architectures [17]. The global and shared memory models are shown as

$$\tau_{glb} = \#msg \times \alpha_{glb} + msize \times \beta_{glb} + flops \times \gamma$$
$$\tau_{sh} = \#msg \times \alpha_{sh} + msize \times \beta_{sh} + nsync \times \alpha_{sync} + flops \times \gamma$$

Like LogP model, our models comprise three parameters α, β and γ. Overall runtime is the sum of three factors, memory latency (α_{glb} or α_{sh}), inverse memory bandwidth (β_{glb} or β_{sh}) and time per flop (γ). Additionally, the shared memory model also captures time required for thread synchronizations ($nsync \times \alpha_{sync}$). We estimate the model parameters for GF100 architecture using micro-benchmarks [16] and summarize them in Table 3.

TABLE 3
Model Parameters for GPU Performance.

	Nvidia C2050	Nvidia C2075	Nvidia K20
Specifications			
Peak TFLOPs (single-precision)	1.03	1.03	3.95
Global memory clock (GHz)	3.0	3.0	5.2
Global memory bandwidth (GB/s)	144	144	250
Core clock rate ($Freq$) GHz	1.15	1.15	0.732
Number of SMs (P)	14	14	14
Number of cores per SM (N_c)	32	32	192
Parameter Estimation with Micro-benchmarks			
α_{glb} (cycles)	95	95	235
β_{glb} (s/GB)	$\frac{1}{108}$	$\frac{1}{96.5}$	$\frac{1}{129}$
α_{sh} (cycles)	27	26	23
β_{sh} (s/GB)	$\frac{1}{880}$	$\frac{1}{898}$	$\frac{1}{864}$
Sync. Latency (α_{sync}) cycles	154	114	53
FP Pipeline latency (γ) cycles	18	18	10

We build separate models for global and shared memory accesses and then combine them to find out the latency (L_q) of a single thread block (See Section 2.2 of the supplementary file for details about the model).

$$L_q = lA_{glb2reg} + lx_{glb2sh} + k(lx_{sh2reg} + l_{compute} + l_{condition} + lx_{reg2sh}) \quad (4)$$

Referring to Equation (4), $lA_{glb2reg}$ is the number of cycles required in loading a block of the matrix A from the global memory to the register file and similarly lx_{glb2sh} represents number of cycles utilized in loading partition of the vector x from the global memory to the shared memory. As shown, these blocks are fetched only once and then they are used k times to calculate partitions for the k vectors which are stored in the shared memory (See other terms in Equation (4)). We calculate total cycles by plugging in the parameters from Table 3 and then find out overall runtime L by taking into account the total number of thread blocks (N_q), number of SMs (P) and the number of thread blocks concurrently running per SM (we obtain this information using CUDA Visual Profiler [20]). We run CUDA code shown in Listing 3 of the supplementary file on the GPUs to measure the actual results for validation of the performance model. We compare the predicted performance with the actual measured results for a range of GPU devices in Figure 5.

To indicate the accuracy of our model, we also show the mean (ϵ_μ) and standard deviation (ϵ_σ) of error (absolute difference in measured and modelled performance) as a percentage of measured performance for each band size. Our model does not capture register spilling, $i.e.$ when the data does not fit in GPU registers, the data is stored to local memory, which is a part of the global memory. Each thread can have a maximum of 64 registers, which are enough to store a single row of the matrix, the length of which is equal to the band size. The band sizes that arise in all practical applications can fit in these registers and therefore, there is no register spilling in our implementation.

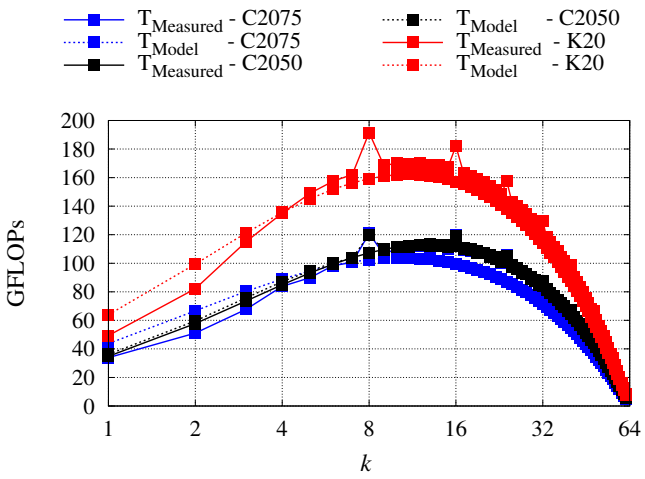


Fig. 5. Performance on different GPU architectures for $b = 9$, C2050 (ϵ_μ= 1.5%, ϵ_σ= 1.6%), C2075 (ϵ_μ = 13.8%, ϵ_σ = 5.3%) , and K20 (ϵ_μ = 7.5%, ϵ_σ = 4.2%)

3.5 Performance Optimization

Having an accurate model to predict performance on the GPU in terms of the problem parameters (n, b), the architectural parameters (γ,α_{glb},β_{glb},α_{sh},β_{sh},P) and the algorithmic parameters (k, b_R), we can solve the following optimization problem to select the algorithmic parameters.

$$\min_{k,b_R} \frac{L(n,b,\gamma,\alpha_{glb},\beta_{glb},\alpha_{sh},\beta_{sh},P,k,b_R)}{Freq}$$

subject to

$$k \leq 2\frac{b_R}{b-1} \quad (5)$$

Referring to (5), the constraint ensures only nearest neighbour communication in the matrix powers kernel as shown in Figure 2. We carry out sensitivity analysis of GPU performance with respect to the algorithmic parameters with constant architectural parameters in Section 6.1. We also highlight in Section 7 that by carefully picking the algorithmic parameters we can achieve higher performance over k SpMVs that otherwise requires significant architectural modifications in terms of global memory bandwidth and latency.

4 Matrix Powers Kernel on FPGA

The potential to use FPGAs in high-performance computing arises from the fact that computer architecture can be specialized to accelerate a particular task (See Section 3.1 of the supplementary file for details). Table 1 lists the important architectural features of FPGAs in terms of raw floating-point performance, on-chip memory capacity and on-chip as well as off-chip memory bandwidth. Referring to Table 1, although the off-chip memory bandwidth and peak floating-point performance is 5× and 2.3× lower than that of the same generation GPU device, it is the on-chip capacity and on-chip memory bandwidth coupled with rich communication-fabric, which make FPGAs suitable for accelerating iterative solvers. We now introduce the architecture-aware hybrid matrix powers kernel that exploits these architectural features to get high throughput. In this regard, we also present a resource-constrained methodology for selecting an optimal k for a target FPGA device.

4.1 Proposed Hybrid Matrix Powers Kernel

The proposed algorithm loads the blocks of the matrix from the slow memory into large on-chip memory using a sequential algorithm and then performs computations within the block in parallel without doing redundant computations. We show the proposed method in Algorithm 1.

Algorithm 1 Hybrid Matrix Powers Kernel

```
for q = 1 to N_q do
    load block q from slow memory into fast memory
    for i = 1 to k do
        compute all locally computable x_j^(i)
        wait for all the receives from neighbours to finish
        compute the remaining entries of x_j^(i) with dependencies
    end for
end for
```

The outer loop is a sequential algorithm that loads the blocks of matrix A such that the block fits into the on-chip memory of the FPGA. The inner loop is a parallel algorithm, which is very similar to the one shown in Figure 2(b)(i). The working of the algorithm is shown in Figure 6 with a toy example, where we have 2 outer blocks that are loaded sequentially from the slow memory into FPGA on-chip memory.

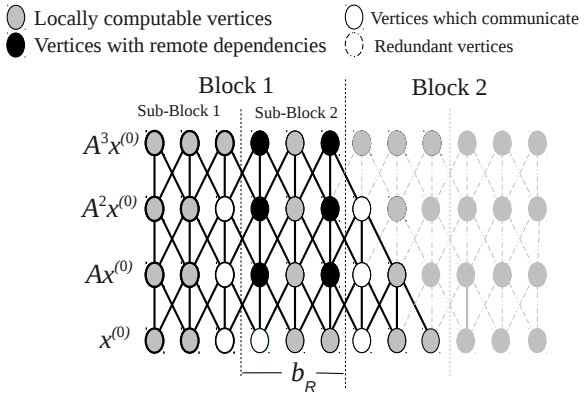


Fig. 6. Proposed hybrid matrix powers graph for n = 12, k = 3, b = 3 and number of blocks $N_q = 2$.

Each outer block is further partitioned into two sub-blocks that can be processed in parallel by an array of processing elements (PEs) working in a SIMD fashion as shown in Figure 7. All the vertices inside a sub-block are computed in a pipelined fashion using a reduction circuit within the PE (See Section 3.2 of the supplementary file for details). After each level in the graph, PEs need to communicate dependencies

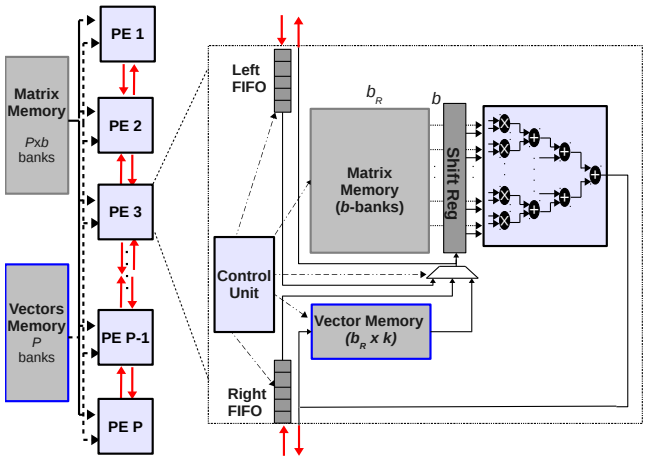


Fig. 7. FPGA data-path for the matrix powers kernel.

to their nearest neighbours. However, unlike GPU, where this communication is only possible using shared global memory and is therefore avoided using redundant computation, the PEs within the FPGA utilize low-latency FIFOs and hence avoid the redundant computation. This allows larger values of k and hence higher possible speedups. To provide motivation, we show the performance of this hybrid matrix powers kernel in Table 4 for the same matrix that we use for demonstrating GPU optimizations in Table 2. We observe that if A is blocked in on-chip memory of the FPGA along with the vector $x^{(i)}$, we can get $\sim$6$\times$ speedup over k SpMVs used in standard iterative solvers. This speedup factor is almost twice of what we achieved with parallel matrix powers kernel on GPU. Although the net performance of FPGA (85 GFLOPs) is less than that of GPU (123 GFLOPs) for this band size, however, we have better silicon efficiency with FPGA (18.8%) compared to GPU (11.9%). We show in Section 6 that for a range of matrix and band sizes, the FPGA even outperforms GPU because of the higher values of k.

TABLE 4
Hybrid Matrix Powers Kernel Mapping on Virtex6-SX475T FPGA (n =1M, b =9, k =10).

	Off-Chip Memory	On-Chip Memory	GFLOPs	Efficiency
k SpMVs	A, $x^{(i)}$		14.21	3.1%
k SpMVs	A	$x^{(i)}$	15.68	3.4%
Hybrid Matrix Powers		A,$x^{(i)}$	85	18.8%

4.2 Modelling Performance

To understand the performance of matrix powers kernel on FPGA, we use the same LogP model which comprises both computation as well as communication cost as shown in Section 3.4 for the GPU. The model is exact due to the highly predictive nature of FPGAs as a computing platform. We show the parameters of the model in Table 5.

As FPGA has relatively larger on-chip memory compared to GPU, we intend to store the k vectors

TABLE 5
Model Parameters for FPGA.

Parameters	Virtex6-SX475T
Global memory latency (α_{glb}) cycles	6
Global memory inverse bandwidth (β_{glb}) s/GB	$\frac{1}{34}$ [18]
FP Add latency (γ_A) cycles	11 [22]
FP Mult latency (γ_M) cycles	8 [22]
FP Operating Frequency ($Freq$) MHz	258
No. of FP Adders	$P(b-1)$
No. of FP Multipliers	Pb

on-chip to be utilized by subsequent modules in communication-avoiding iterative solver. There are three stages in the matrix powers kernel on FPGA: loading the block, computing the sub-blocks in parallel and an optional stage for storing the k vectors back to the off-chip memory if they do not fit on-chip. The latency (L_q) of a single block is the summation of the latency of these three stages (See Section 3.3 of the supplementary file for details about the model).

$$L_q = (lA_{glb2local} + lx_{glb2local}) + k \times l_{compute} + k \times lx_{local2glb} \quad (6)$$

The overall latency L is then calculated by multiplying L_q with total number of blocks N_q.

4.3 Resource-Constrained Methodology

Like GPU, the performance of the matrix powers kernel depends on the problem parameters (n,b), the architectural parameters (P,γ_A,γ_M,α_{glb},β_{glb}) and the algorithmic parameters (k,b_R). We find the maximum number P of PEs that can be synthesized within the FPGA device for the given band size b (the number of floating-point units only depends on this parameter shown in Table 5). We calculate the memory bandwidth required for these P PEs ($2b$ words per PE), partition the available on-chip memory in $b_R \times b$ blocks and assign these blocks to P PEs. We solve the following constrained optimization problem to pick k on a particular FPGA for a given problem.

$$\min_{k,b_R} \frac{L(n,b,\gamma_A,\gamma_M,\alpha_{glb},\beta_{glb},P,k,b_R)}{Freq}$$

subject to

$$\begin{aligned} R(P) &\leq FPGA_{Logic} \\ M(P,b_R,k) &\leq FPGA_{BRAMs} \\ k &\leq 2\frac{b_R}{b-1} \end{aligned} \quad (7)$$

Referring to Equation (7), our objective is to minimize the runtime based on constraints on FPGA resources. $M(P,k,b_R)$ is the number of BRAMs (FPGA on-chip memories) required and $R(P)$ is a vector containing the number of resources in terms of LUTs, FFs and DSP48Es that are used in floating-point adders and multipliers [22]. The last constraint ensures only nearest neighbour communication in the matrix powers kernel as shown in Figure 6.

5 Evaluation Methodology

We use the same generation (40nm) of GPU (NVidia C2050) and FPGA (Virtex6-SX475T) devices as mentioned in Table 1. We use CUDA 5.0 for compiling CUDA kernels and also use `cusp-v0.3.1` [15], which is a sparse library optimized for GPUs. We prefer CUSP over vendor-specific cuSPARSE [21] since cuSPARSE lacks SpMV routine for banded matrices. We use Xilinx IP Core [22] for BRAMs, adders and multipliers. We synthesize as well as place and route the circuit using Xilinx ISE 13.4. To evaluate the performance of matrix powers kernel on GPU and FPGA, we use a range of randomly generated matrices with varying band sizes.

6 Results

As FPGAs and GPU are radically different computing platforms, we first analyze how the communication-avoiding approach of the matrix powers kernel can enhance their individual performance over k SpMVs in standard iterative solvers. We then compare the matrix powers kernel with optimal k for both GPU and FPGA and show which architecture is better in different problem and band sizes.

6.1 Sensitivity to Algorithmic Parameters

We use the formulations in (5) and (7) to select the algorithmic parameters for minimizing the runtime on GPU and FPGAs respectively. There are two algorithmic parameters, the partition size b_R and the unroll factor k. While both parameters affect the surface to volume ratio but the impact of the partition size b_R is marginal as compared to the unroll factor k (See Section 2.4 of the supplementary file). To show the performance variation with k, we take a problem size (n = 1M) and show both the communication and computation costs for band size equal to 9 in Figure 8 and for band size equal to 27 in Figure 1. We observe from Figure 8 and Figure 1 that in GPU, the optimal value of k decreases as we increase the band size whereas in case of FPGA, it shows opposite trend. After a certain value of k, both computation and communication costs dominate on GPU. The computation cost increases due to $O(k^2b^2)$ growth in redundant operations and as a result the larger the band size, the smaller the value of k whereas the communication cost increases with increasing value of k due to shared memory accesses (See Figure 4 of the supplementary file).

In case of FPGA, the optimal value of k increases with increasing band size. The communication cost decreases with increasing value of k for all band sizes until it flattens as the vectors can no more be stored on-chip and they have to be stored back. For large band sizes, the computation to communication ratio is large and these vectors can be stored in an overlapped fashion. This allows large values of k as shown in Figure 1 and 8. As a result of careful selection of the unroll factor k on both GPU and FPGA, we see a significant increase in the silicon efficiency of these architectures as shown in Figure 9. We also observe that FPGAs have much better silicon efficiency as compared to GPU because of the relatively large values of k as shown in Table 6.

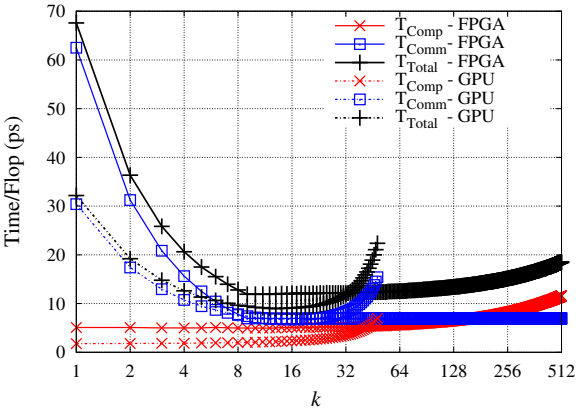


Fig. 8. Algorithmic Sensitivity ($b = 9$, $n = 1$M).

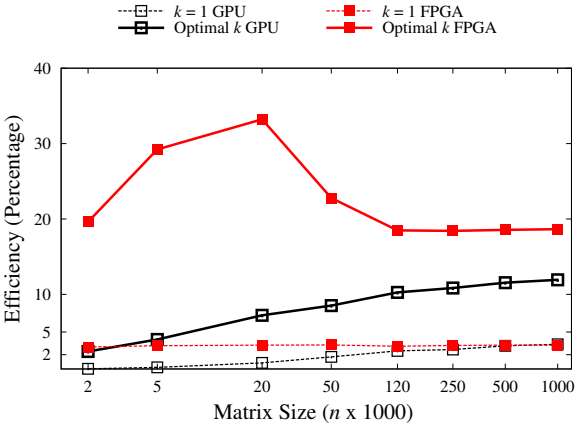


Fig. 9. Efficiency of FPGA and GPU as a percentage of peak single-precision performance ($b = 9$).

6.2 Performance Comparison

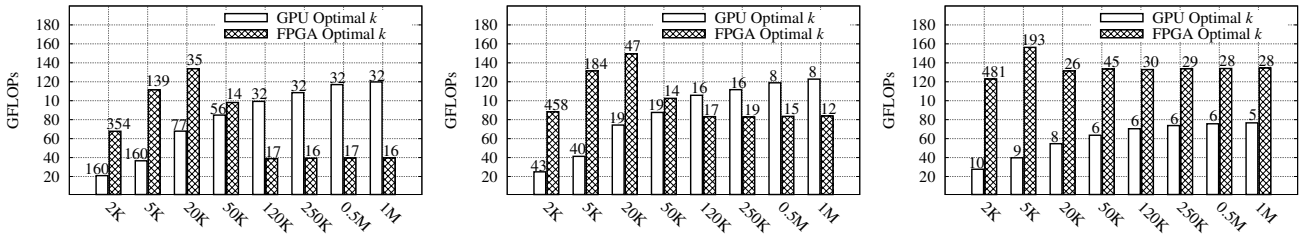
Although FPGAs have better silicon efficiency than GPU as a result of careful selection of k, to compare these architectures in terms of raw performance for a range of problem and band sizes, we see three interesting scenarios in Figure 10.

6.2.1 *Small Problem Sizes ($n \leq 20$K)*

In this case, across all band sizes, due to small matrix size, k vectors can also be stored in the on-chip memory of the FPGA with k having large values. Since there is no off-chip communication involved except loading the matrix once, we see up to 4.4× speedup over GPU in this region due to the large on-chip capacity and zero redundant operations in FPGA.

6.2.2 *Large Problem Sizes ($n \geq 20$K), Small Band Sizes ($b \leq 9$)*

For large problem sizes and small band sizes, GPU performs slightly better than FPGA since the vectors



(a) Band Size b = 3 (b) Band Size b = 9 (c) Band Size b = 27

Fig. 10. Matrix Powers Performance Comparison vs. Matrix Size. The optimal value of k is shown on the top of the bar for both FPGA and GPU.

TABLE 6
Matrix powers kernel performance comparison (Range is for n =2k − 1M).

Band Size	GPU			FPGA			FPGA vs. GPU SpeedUp
	k Range	Efficiency(%)		k Range	Efficiency(%)		
		k =1	Optimal k		k =1	Optimal k	
3	160 − 32	0.04 − 2.6	2 − 11.6	354 − 16	2.8 − 3.3	15.1 − 8.8	3.2× − 0.3×
7	58 − 16	0.1 − 3.2	2.3 − 10.5	436 − 28	3.0 − 3.3	19.8 − 16.1	3.7× − 0.7×
9	43 − 8	0.11 − 3.3	2.4 − 11.9	458 − 12	3.0 − 3.3	19.6 − 18.6	3.5× − 0.7×
13	30 − 16	0.2 − 3.5	2.7 − 9.1	466 − 21	3.0 − 3.3	23.1 − 22.7	3.6× − 1.1×
27	10 − 5	0.3 − 3.7	2.6 − 7.4	481 − 28	3.1 − 3.3	27.3 − 29.9	4.4× − 1.7×

spill into off-chip memory in case of FPGA and due to its relatively low off-chip memory bandwidth, the problem becomes communication-bound. On the other hand, GPU has higher off-chip bandwidth and as a result we see up to ∼3× speedup.

6.2.3 *Large Problem Sizes($n \geq$ 20K), Large Band Sizes ($b >$ 9)*

In this region, as the band size increases, number of redundant operations grow rapidly which constrain GPU performance and as a result we see very small values of k. On the other hand, as the computation and communication (storing the vectors) ratio is high , the problem remains compute-bound as vectors can be stored in an overlapped fashion. As a result, FPGAs perform better and we get up to 1.7× speedup over GPU.

7 Architectural Insight

Since we have an accurate predictive model for GPU and FPGA, we can answer questions

- if we do not change the algorithmic parameters, how we might have to change the architectural parameters?
- how to optimally change the architectural parameters to obtain a desired performance?

7.1 Sensitivity to GPU Architectural Parameters

We solve the optimization problem in (5) using a steepest ascent method. The steepest ascent curve shows different points of performance enhancement and the corresponding architectural parameters as shown in Figure 11. For example, to achieve a 3.5× speed up over a problem running on C2050 GPU, we show that our optimization vector (α_{glb}, β_{glb},α_{sh},β_{sh}) for a hypothetical GPU should be scaled as ($\sim\frac{1}{10}$,∼10, $\frac{1}{13}$,1.23). However, using our predictive model and measured results, we have already shown in Figure 5 that same performance can be obtained by careful selection of k without changing the architecture.

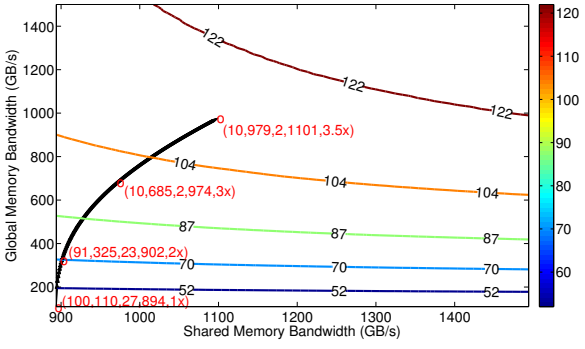


Fig. 11. Architectural Sensitivity− GPU performance contours in GFLOPs as a function of global memory bandwidth (β_{glb}) and shared memory bandwidth (β_{sh}) for band size b = 9 and n = 1M. Specific points (in red) on the steepest ascent curve (in black) are shown representing (α_{glb},β_{glb},α_{sh},β_{sh},$\frac{L_{base}}{L_{pred}}$) where L_{base} is the performance obtained on C2050 GPU. α_{glb} and α_{sh} are in cycles whereas β_{glb} and β_{sh} are in GB/s.

7.2 Sensitivity to FPGA Architectural Parameters

We show the sensitivity of the FPGA performance with respect to off-chip memory bandwidth in Figure 12 for two cases. Firstly, when we have fixed algorithm with k =1 and second with optimal value of k. We observe that by carefully picking k we can get a 5.6× performance for a Virtex6-SX475T FPGA. To achieve similar performance, k =1 curve shows that off-chip memory bandwidth needs to be scaled by 8.6×. In case we can not tolerate such significant modifications in architecture, a tight algorithm-architecture interaction is necessary to accelerate such kind of communication-bound problems.

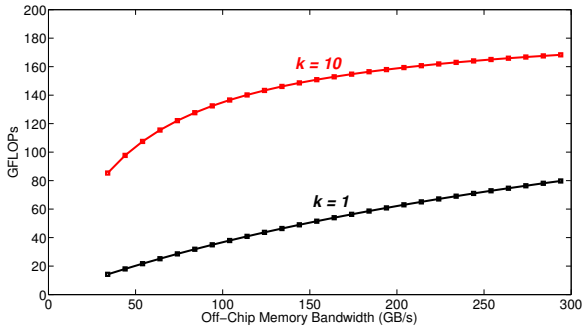


Fig. 12. Architectural Sensitivity− FPGA performance in GFLOPs as a function of off-chip memory bandwidth (β_{glb}) for band size $b = 9$ and $n = 1$M. The starting point of the curves is a Virtex6-SX475T architecture with an off-chip bandwidth of 34 GB/s.

8 Conclusion

Trading communication with computation increases the silicon efficiency of hardware accelerators like FPGAs and GPU for accelerating communication-bound sparse iterative solver. Although unrolling k iterations using the matrix powers kernel provides significant performance improvement compared to standard k SpMVs on a GPU, the performance is constrained due to quadratic growth in redundant computations. Our proposed hybrid matrix powers kernel for FPGA exploits the architectural features of this radically different platform to minimize redundant computations. This allows us large value of k and hence superior silicon efficiency compared to GPU. For a range of randomly generated banded matrices, we demonstrate $0.3\times-3.2\times$ and $1.7\times-4.4\times$ speedup over GPU for small and large band sizes respectively. Our architectural insight shows a tight algorithm-architecture interaction can provide similar performance, which otherwise requires significant enhancements in memory bandwidth.

9 Future Work

Besides Nvidia GPUs, we intend to validate our predictive model for the matrix powers kernel on other GPUs as well. We intend to extend the work to general sparse matrices with hyper-graph partitioning as a pre-processing step. We intend to use the matrix powers kernel instead of SpMV for applications [11] where we have to solve $Ax = b$ repeatedly in each iteration. As the sparsity pattern does not change over the iterations, we believe that the cost of this pre-processing step will be quite low as compared to the actual computation.

Acknowledgment

Dr. George A. Constantinides would like to acknowledge the support of EPSRC (EP/I020357/1 & EP/G031576/1). We would like to thank Michael Anderson, PAR Lab, University of California, Berkeley for providing us the GF100 micro-benchmarks. Also, we would like to thank Mark Hoemmen and Marghoob Mohiyuddin, University of California Berkeley for giving useful suggestions that help in improving the draft.

References

[1] G. H. Golub and C. F. Van Loan, *Matrix Computations*, 3rd ed. John Hopkins University Press, 1996.
[2] 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.
[3] M. Snir and S. Graham, *Getting up to speed: The Future of Supercomputing*, National Academies Press, 2004.
[4] S.A. Toledo, *Quantitative performance modeling of scientific computations and creating locality in numerical algorithms*, PhD. Thesis, Massachusetts Institute of Technology, 1995.
[5] M. Strout, L. Carter and J. Ferrante, *Sparse Tiling for Stationary Iterative Methods*, International Journal of High Performance Computing Applications, vol. 18(1), pp. 95-114, 2004.
[6] S. K. Kim and A. T. Chronopoulos, *A class of Lanczos-like algorithms implemented on parallel computers*, Journal of Parallel Computing, vol. 17(6), Elsevier, 1991.
[7] J. Demmel, M. Hoemmen, M. Mohiyuddin and K. Yelick, *Avoiding Communication in Sparse Matrix Computations*, In Proceedings of IPDPS. April, 2008.
[8] M. Mohiyuddin, M. Hoemmen, J. Demmel and K. Yelick, *Minimizing communication in sparse matrix solvers*, In Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 2009.
[9] V. Volkov, *Programming inverse memory hierarchy: case of stencils on GPUs*, In GPU Workshop for Scientific Computing, International Conference on Parallel Computational Fluid Dynamics (ParCFD), 2010.
[10] G. Sewell, *The numerical solution of ordinary and partial differential equations*. Wiley-Interscience, 2005.
[11] E. Klerk, *Exploiting special structure in semidefinite programming: A survey of theory and applications*, European Journal of Operational Research, Elsevier, 2010.
[12] C.V Rao, S.J. Wright and J.B. Rawlings, *Application of interior-point methods to model predictive control*, Journal of optimization theory and applications, vol. 99(3): pp. 723-757, Springer, 1998.
[13] E. Anderson, Z. Bai, C. Bischof, L.S. Blackford, J. Demmel, Jack J. Dongarra, J. Du Croz, S. Hammarling, A. Greenbaum, A. McKenney and D. Sorensen, *LAPACK Users' Guide (Third Edition)*, vol. 9: Society for Industrial and Applied Mathematics, Philadelphia, USA, 1999.
[14] 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 multi-core architectures*, in IEEE International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1-12, 2009.
[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.J. Anderson, D. Sheffield and K. Keutzer, *A Predictive Model for Solving Small Linear Algebra Problems in GPU Registers*, In Proceedings of 26th IEEE International Symposium on Parallel and Distributed Systems, pp. 2-13, 2012.
[17] D. Culler, R. Karp, D. Patterson, A. Sahay, E.K. Schauser, E. Santos, R. Subramonian and T.V. Eicken, *LogP: Towards a realistic model of parallel computation*, ACM, vol.28(7), 1993.
[18] P. Sundararajan, *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.
[20] *Compute Visual Profiler*, http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/VisualProfiler/Compute_Visual_Profiler_User_Guide.pdf, 2010.
[21] *Nvidia cuSPARSE Library*, http://docs.nvidia.com/cuda/pdf/CUSPARSE_Library.pdf, 2013.
[22] *Xilinx DS816 Floating-Point Operator v6.0*, http://www.xilinx.com/support/documentation/ip_documentation/floating_point/v6_0/ds816_floating_point.pdf, 2012.

Abid Rafique received the MSc. degree from Technical University Munich, Germany in 2010. Since 2010, he is a PhD student in the Circuits and Systems research group at Imperial College London. His research interests include high performance computing, parallel architectures and communication optimization in iterative numerical algorithms.

George A. Constantinides (S'96-M'01-SM'08) received the Ph.D. degree from Imperial College London in 2001. Since 2002, he has been with the faculty at Imperial College London, where he is currently Professor of Digital Computation and Head of the Circuits and Systems research group. He will be program (general) chair of the ACM International Symposium on Field-Programmable Gate Arrays in 2014 (2015). He serves on several programme committees and has published over 150 research papers in peer refereed journals and international conferences. Dr Constantinides is a Senior Member of the IEEE and a Fellow of the British Computer Society.

Nachiket Kapre is an Assistant Professor at Nanyang Technological University, Singapore since October 2012. He has received a PhD in Computer Science (2010) and a MS in Computer Science (2006) and an MS in Electrical Engineering (2005) all from California Institute of Technology, Pasadena, USA. He was awarded the prestigious Imperial College Junior Research Fellowship in 2010. He has won the best paper award at FPT 2011 and a HiPEAC paper award for his FCCM 2013 paper. His FCCM 2006 paper was featured in the FCCM20 list as one of the influential papers in past 20 years at FCCM conferences. He is broadly interested in exploiting the limits of modern VLSI architectures though reconfigurability, parallelism and domain-specific frameworks.