Imperial College London
Department of Electrical and Electronic Engineering

# Communication Optimization in Iterative Numerical Algorithms: An Algorithm-Architecture Interaction

Abid Rafique

January 2014

Supervised by George A. Constantinides and Nachiket Kapre

Submitted in part fulfilment of the requirements for the degree of
Doctor of Philosophy in Electrical and Electronic Engineering of Imperial College London
and the Diploma of Imperial College London

## Declaration

I herewith certify that all material in this dissertation that is not my own work has been properly acknowledged.

Abid Rafique

I would like to dedicate this thesis to my beloved mother (late)

## Copyright Declaration

# Abstract

Trading communication with redundant computation can increase the silicon efficiency of common hardware accelerators like FPGA and GPU in accelerating sparse iterative numerical algorithms. While iterative numerical algorithms are extensively used in solving large-scale sparse linear system of equations and eigenvalue problems, they are challenging to accelerate as they spend most of their time in communication-bound operations, like sparse matrix-vector multiply (SpMV) and vector-vector operations. Communication is used in a general sense to mean moving the matrix and the vectors within the custom memory hierarchy of the FPGA and between processors in the GPU; the cost of which is much higher than performing the actual computation due to technological reasons. Additionally, the dependency between the operations hinders overlapping computation with communication. As a result, although GPU and FPGA are offering large peak floating-point performance, their sustained performance is nonetheless very low due to high communication costs leading to poor silicon efficiency.

In this thesis, we provide a systematic study to minimize the communication cost thereby increase the silicon efficiency. For small-to-medium datasets, we exploit large on-chip memory of the FPGA to load the matrix only once and then use explicit blocking to perform all iterations at the communication cost of a single iteration. For large sparse datasets, it is now a well-known idea to unroll $k$ iterations using a matrix powers kernel which replaces SpMV and two additional kernels, TSQR and BGS, which replace vector-vector operations. While this approach can provide a $\Theta(k)$ reduction in the communication cost, the extent of the unrolling depends on the growth in redundant computation, the underlying architecture and the memory model. In this work, we show how to select the unroll factor $k$ in an architecture-agnostic manner to provide communication-computation tradeoff on FPGA and GPU. To this end, we exploit inverse-memory hierarchy of the GPUs to map matrix power kernel and present a new algorithm for the FPGAs which matches with their strength to reduce redundant computation to allow large $k$ and hence higher speedups. We provide predictive models of the matrix powers kernel to understand the communication-computation tradeoff on GPU and FPGA. We highlight extremely low efficiency of the GPU in TSQR due to off-chip sharing of data across different building blocks and show how we can use on-chip memory of the FPGA to eliminate this off-chip access and hence achieve better efficiency. Finally, we demonstrate how to compose all the kernels by using a unified architecture and exploit on-chip memory of the FPGA to share data across these kernels.

Using the Lanczos Iteration as a case study to solve symmetric extremal eigenvalue problem, we show that the efficiency of FPGAs can be increased from 1.8% to 38% for small-to-medium scale dense matrices whereas up to 7.8% for large-scale structured banded matrices. We show that although GPU shows better efficiency for certain kernels like the matrix powers kernel, the overall efficiency is even lower due to increase in communication cost while sharing

data across different kernels through off-chip memory. As the Lanczos Iteration is at the heart of all modern iterative numerical algorithms, our results are applicable to a broad class of iterative numerical algorithms.

# Acknowledgement

I would like to begin thanking my supervisor Dr. George A. Constantinides. I am grateful to him for his complete support, motivation and encouragement throughout my PhD. He has continuously guided me, and at the same time gave me enough freedom to explore my own research direction. I must acknowledge his clarity of thought and intellect to polish my ideas during our meetings. I am particularly thankful to him for keeping me focused and help me writing my thesis in a timely manner. Lastly, I am also grateful to him for providing me enough travel opportunities to present my work in reputed conferences.

I would like to acknowledge my co-supervisor, Dr. Nachiket Kapre. He has not only helped me in research but also sharpen my presentation skills. I am thankful to him for the time he spent in multiple iterations of the papers.

I consider myself very fortunate to have two great supervisors. I would be indebted to them for their guidance and supervision. They helped me learn key skills necessary for independent research that will be helpful for the rest of my life.

I had the company of many brilliant colleagues at Imperial College with whom I had interesting and fruitful discussions. I would particularly thank to Dr. David Boland, Dr. Juan Jerez, Dr. Samual Bayliss, Dr. Shakil Ahmed, Dr. Ammar Hassan and Dr. Andrea Suardi. I would particularly thank Michael Anderson and Mark Hoemmen at PARLab, UC Berkeley who took time to revise some of my work and provided feedback that helped me in improving the overall quality.

My time at Imperial College would not have been the same without my friends. They kept me cheerful and encourage during the hard times working towards my PhD. I would like to thank particularly Qazi Rashid Hamid, Usman Adeel, Muhammad Usman and Hafiz-Ul-Asad.

On a personal note, I would like to express gratitude to my family and in particular my mother who is no more in this world to share these moments. She had been a continuous source of inspiration and great love.

Finally, I would like to acknowledge my wife, Nazia, and the greatest gift of God, *i.e.* my daughter Fatima. I am thankful to them for the love and patience they have shown throughout my PhD.

# Contents

# List of Figures

# List of Tables

## List of Acronyms

| | |
|---|---|
| **GPU** | Graphical Processing Unit |
| **FPGA** | Field Programmable Gate Array |
| **BLAS** | Basic Linear Algebra Sub-Routine |
| **MINRES** | Minimum Residual Method |
| **GMRES** | Generalized Minimum Residual Method |
| **CG** | Conjugate Gradient |
| **SVD** | Singular Value Decomposition |
| **SIMD** | Single Instruction Multiple Data |
| **SIMT** | Single Instruction Multiple Threads |
| **SM** | Symmetric Multiprocessor |
| **SDP** | Semi-definite Programming |
| **GFLOPs** | Giga Floating-Point Operations Per Second |
| **CSR** | Compressed Sparse Row |
| **CSC** | Compressed Sparse Column |
| **CDS** | Compressed Diagonal Storage |
| **SpMV** | Sparse Matrix-Vector Multiply |
| **SpMM** | Sparse Matrix-Multiple Vector Multiply |
| **PDE** | Partial Differential Equation |
| **ODE** | Ordinary Differential Equation |
| **ED** | Energy Detection |
| **MED** | Maximum Eigenvalue Detection |
| **DRAM** | Dynamic Random Access Memory |
| **NUMA** | Non-uniform Random Access Memory |
| **CA** | Communication-Avoiding |
| **MGS** | Modified Gram-Schmidt |
| **LAPACK** | Linear Algebra Package |
| **TSQR** | Tall-Skinny QR |

| | |
|---|---|
| **BGS** | Block Gram-Schmidt |
| **IPC** | Instructions Per Cycle |
| **MKL** | Math Kernel Library |
| **CUBLAS** | CUDA Basic Linear Algebra Subroutine |
| **ACML** | AMD Core Math Library |
| **DAG** | Directed Acyclic Graph |
| **ILP** | Instruction Level Parallelism |
| **LUT** | Look Up Table |
| **FF** | Flip Flop |
| **DSP** | Digital Signal Processor |
| **CLB** | Configurable Logic Block |
| **BW** | Bandwidth |
| **FP** | Floating-Point |
| **I/O** | Input Output |
| **BRAM** | Block Random Access Memory |
| **PE** | Processing Element |
| **FIFO** | First In First Out |
| **MCMC** | Monte Carlo Markov Chain |

# Nomenclature

| | |
|---|---|
| $\alpha$ | memory or network latency |
| $\alpha_g$ | latency of global memory (in cycles) |
| $\alpha_{sh}$ | latency of shared memory (in cycles) |
| $\alpha_{sync}$ | latency of synchronization between threads (in cycles) |
| $\beta$ | inverse memory or network bandwidth |
| $\beta_g$ | inverse global memory bandwidth (in cycles) |
| $\beta_{sh}$ | inverse shared memory bandwidth (in cycles) |
| $\gamma$ | time per floating-point operation |
| $\gamma_A$ | latency of floating-point adder (in cycles) |
| $\gamma_M$ | latency of floating-point multiplier (in cycles) |
| $\lambda_i$ | $i$th eigenvalue |
| $\lambda_{max}$ | maximum eigenvalue |
| $\lambda_{min}$ | minimum eigenvalue |
| $A \succeq 0$ | positive semi-definite, $i.e$ minimum eigenvalue is greater than equal to zero |
| $A$ | upper case $-$ matrix |
| $A^T$ | transpose of a matrix $A$ |
| $A_{ij}$ | the element of the matrix $A$ at row $i$ and column $j$ |
| $b$ | band size |
| $b_R$ | number of rows in each partition while partitioning the input matrix $A$ |
| $I_n$ | an identity matrix of size $n \times n$ |
| $k$ | algorithmic parameter to trade communication with computation in communication-avoiding iterative algorithms |
| $L$ | total latency of the matrix powers kernel (in cycles) |
| $L_q$ | latency of a single thread block in matrix powers kernel (in cycles) |

|  |  |
|---|---|
| $l_{compute}$ | compute latency per thread block (in cycles) |
| $l_{condition}$ | latency for evaulating conditional statements per thread block (in cycles) |
| $lA_{glb2reg}$ | latency per thread block in data transfer, global memory —> register file (in cycles) |
| $lx_{glb2sh}$ | data transfer latency per thread block, shared memory —> register file (in cycles) |
| $lx_{reg2glb}$ | data transfer latency per thread block, register file —> global memory (in cycles) |
| $lx_{reg2sh}$ | data transfer latency per thread block, register file —> shared memory (in cycles) |
| $m$ | number of rows |
| $n$ | number of columns |
| $N_q$ | number of partitions |
| $N_T$ | number of threads |
| $N_{tiles}$ | number of sub-matrices in tall-skinny QR factorization |
| $nnz$ | number of non-zero elements in a sparse matrix |
| $P$ | number of problems |
| $p$ | block size in Block Krylov methods |
| $P_e$ | total number of processing elements |
| $Q$ | a square $n \times n$ orthogonal matrix *i.e.* $Q^TQ = I_n$ |
| $q$ | partition number |
| $Q_r$ | a tall-skinny orthogonal matrix of size $m \times r$ *i.e.* $Q_r^TQ_r = I_r$ |
| $R$ | upper triangular matrix |
| $r$ | number of iterations |
| $t$ | cost per iteration (in seconds) |
| $T_r$ | a tri-diagonal matrix of size $r \times r$ |
| $t_r$ | cost for $r$ iteration (in seconds) |
| $t_{comm}$ | communication cost (in seconds) |
| $t_{comp}$ | computation cost (in seconds) |
| $v_i$ | householder reflector |
| $x$ | lower case $-$ vector |
| $x_j^i$ | $j$th element of vector $x^i$ |

| | |
|---|---|
| $G_{\text{measured}}$ | measured performance (in GFLOPs) |
| $G_{\text{modelled}}$ | modelled performance (in GFLOPs) |
| $t_{\text{measured}}$ | measured time (in cycles or seconds) |
| $t_{\text{modelled}}$ | modelled time (in cycles or seconds) |

# 1 Introduction

The main motivation of this thesis is to answer the question: how to optimize communication in iterative numerical algorithms in order to improve silicon efficiency of common hardware accelerators like Field Programmable Gate Arrays (FPGAs) and Graphics Processing Unit (GPU)? The cost of a numerical algorithm comprises two factors (1) the *computation* cost of performing floating-point operations and (2) the *communication* cost of moving data within the memory hierarchy in the sequential case and or between processors in the parallel case. The communication cost includes both latency and bandwidth. Usually the communication cost is much higher than the computation cost, and there is a wide gap between these costs, *e.g.* the DRAM latency and bandwidth is improving by 5.5% and 23% respectively whereas the cost per floating-point operation decreases by 59% per year [41]. Iterative numerical algorithms belong to the class of communication-intensive algorithms and are widely used to solve large-scale sparse linear systems of equations $Ax = b$ and eigenvalue problems $Ax = \lambda x$ [33]. These algorithms 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 dependencies between these operations hinder overlaping communication with computation. No matter how much parallelism can be exploited to accelerate SpMV, the performance of the iterative numerical algorithms 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 as shown in Table 1.1.

Table 1.1: Silicon Efficiency of FPGA and GPU for Iterative Numerical Algorithms (FPGA clock frequency is not reported in [87]).

| **Device** | **Tech. (nm)** | **Peak GFLOPs** (single-precision) | **Memory BW** (Off-Chip) | **Silicon Efficiency** (Upper bound) |
|---|---|---|---|---|
| **Virtex6** (SX475T) | 40 | 450 [87] | 34 GB/s [87] | 4% |
| **Nvidia C2050** (Fermi) | 40 | 1030 | 144 GB/s | 7% |

FPGAs have long been used as an alternative to microprocessors for computing tasks which do not involve floating-point computation [79] [13] [45]. With increasing silicon densities due to Moore's Law, FPGAs have seen much high peak floating-point performance. Similarly, GPUs have recently been used as another hardware accelerator in high performance scientific computing delivering a higher peak floating-point performance compared to FPGAs as shown in Table 1.1. Many important applications from science and engineering spend time solving linear systems of equations or eigenvalue problems, therefore, the question is how much of peak floating-point performance of GPU and FPGAs can be sustained while accelerating the iterative numerical algorithms?

Before discussing different approaches to improve efficiency of these hardware accelerators, we describe runtime of a single iteration of the numerical algorithm in terms of communication and computation costs by Equation (1.1)

$$\begin{aligned} t_{comm} &= \#msg \times \alpha + msize \times \beta. \\ t_{comp} &= flops \times \gamma. \\ t &= t_{comm} + t_{comp}. \end{aligned} \tag{1.1}$$

We consider communication happens in the form of messages, which can be words from slow memory to fast memory or between different processors. Overall runtime is the sum of three factors, memory latency ($\alpha$), inverse memory bandwidth ($\beta$) and time per flop ($\gamma$). For $r$ iterations of the iterative numerical algorithm, SpMV is launched $r$ times to build a representation of the Krylov subspace span($x$, $Ax$, $A^2x$, ...., $A^rx$) [34]. The total cost for $r$ iterations is then given by

$$t_r = r \times t_{comm} + r \times t_{comp}. \tag{1.2}$$

Due to technology scaling, computational 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) [41]. To minimize communication cost, knowing that the matrix $A$ stays constant, an obvious approach is to store the matrix in the on-chip memory [31] [61] and then reuse it for $r$ iterations. The communication cost is effectively reduced by a factor of $r$ as shown in Equation (1.3).

$$t_r^{'} = t_{comm} + r \times t_{comp}. \tag{1.3}$$

However, for large matrices which do not fit on-chip, the matrix needs to be moved $r$ times within the memory hierarchy making it both a latency-bound as well as bandwidth-bound problem [65]. In order to bridge the gap between computation and communication performance, the communication-avoiding iterative solver [48] is recently proposed, which is an algorithmic approach to trade communication with redundant computation. As the same matrix is used over and over to generate a new vector in each iteration, the SpMV kernel is replaced with a *matrix powers kernel* which unrolls $k$ iterations to generate $k$ vectors in a single sweep. 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 [48]. As a result, for a given accuracy, the number of iterations is reduced by $k$. This reduces the communication cost by $k$ but at the expense of $f(k)$

growth in redundant computation.

$$t_r^{''} = \frac{r}{k} \times t_{comm} + r \times t_{comp} + \frac{r}{k} \times f(k). \tag{1.4}$$

The maximum value of $k$ to which we can unroll depends on the underlying architecture, its memory model and computation to communication ratio for a given problem. Therefore, we need to pick this parameter $k$ carefully for different architectures as well as varying problem sizes. Besides replacing $k$ SpMVs with the matrix powers kernel, the vector-vector operations in a standard iterative solver are also replaced with new kernels like Block Gram-Schmidt Orthogonalization Orthogonalization (BGS) and QR factorization [48]. The addition of these kernels not only introduces extra computation but also poses a composition challenge involving more communication cost while sharing data across these kernels. There are three main challenges associated with this communication-avoiding approach on parallel architectures

- how to keep the redundant computation as low as possible in each kernel to minimize the computation cost?
- how to compose different kernels to minimize the communication cost in sharing data across these kernels?
- 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 thesis, we attempt to address these challenges in a systematic way to minimize communication cost and thereby increasing the silicon efficiency of FPGAs in accelerating iterative numerical algorithms. To this end, we present an algorithm for the matrix powers kernel that matches with the strengths of the FPGA to avoid redundant computation. We make use of the large on-chip memory of the FPGAs to share data across different kernels and highlight that such sharing is not possible using the present GPU architecture. In order to trade communication with computation, we provide a resource-constrained methodology to select the algorithmic parameter $k$. The end result is a recipe to generate custom hardware by picking the algorithmic parameter $k$ in an automatic fashion to minimize overall cost and thereby increase silicon efficiency. We compare our results with GPU and show that for a range of problem sizes, the FPGA outperforms the GPU.

## 1.1 Case Study: Lanczos Iteration for Solving Symmetric Extremal Eigenvalue Problem

The choice of iterative numerical algorithm depends on the characteristics of the matrix $A$, *e.g.* for solving $Ax = b$, the Conjugate Gradient [39] is used if $A$ is positive definite, often the Minimum Residual Method (MINRES) is used for indefinite but symmetric matrices. A recipe is given in [16] and [12] to choose the particular algorithm for solving linear system of equations and eigenvalue problem respectively. However, at the heart of each iterative numerical algorithm is a Lanczos Iteration [39] and the results obtained for the Lanczos Iteration are generally applicable to all iterative numerical algorithms. We use Lanczos Iteration to solve symmetric extremal eigenvalue problem, *i.e.* finding only the maximum and minimum eigenvalue of the symmetric matrix $A$. In various applications including semidefinite optimization programs [94] and eigen-based channel sensing [104], we need to solve multiple of these problems and therefore it is highly desirable to accelerate these iterative numerical algorithms.

## 1.2 Thesis Organization

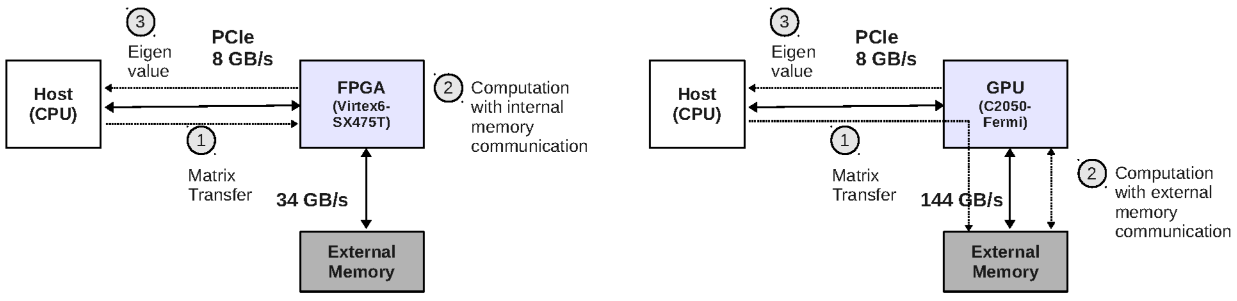
Chapter 2 introduces the reader to the four main problems in applied numerical linear algebra including solution to the linear systems of equation and eigenvalue problem. It discusses different structures involved in these problems. Chapter 2 also classifies the numerical methods as direct and iterative methods, which are used to solve dense and sparse problems respectively. Iterative numerical algorithms are discussed in general and the Lanczos Iteration in particular. The chapter also highlights the communication-bound kernels in the Lanczos Iteration. It then discusses different approaches used for accelerating communication-bound kernels and highlights the limitations in terms of increased communication cost particularly for large-scale problems. Finally, it talks about the communication-avoiding variant of the Lanczos Iteration, the Communication-Avoiding Lanczos, its building blocks and the challenges it poses for algorithm designers as well as hardware architects. We also introduce structurally sparse banded matrices which are of great interest both for their applications and their use as benchmark for architecture evaluation.

Chapter 3 discusses various computing platforms including multi-cores, many-cores (GPUs, Intel Xeon Phi Co-processor) and custom computing architectures like FPGAs. We compare their architectural features including peak floating-point performance, on-chip memory bandwidth, on-chip memory capacity and off-chip memory bandwidth. We finally highlight the low silicon efficiency of these architectures in accelerating the communication-bound iterative numerical algorithms.

Chapter 4 talks about hardware acceleration of Lanczos Iteration working on small-to-medium size data sets. We use explicit cache-blocking to minimize communication cost as shown in Equation (1.3). We design a single-precision hardware architecture for the Lanczos Iteration similar to [61] to minimize computation cost and thereby reduce overall runtime. We show how we can increase the silicon efficiency by exploiting pipeline parallelism and solving multiple independent symmetric extremal eigenvalue problems. We finally compare the results with GPU and show that FPGAs have better silicon efficiency due to explicit cache-blocking and pipeline parallelism. The problem setting for Chapter 4 is shown in Figure 1.1. In case of the FPGAs, we assume that the matrices are transferred from host memory to the internal memory of the FPGA, where they are blocked and are reused for all iterations. For GPUs, due to small on-chip memory, the matrices are transferred from host memory to the external memory, from where they are fetched in each Lanczos iteration. At the end, the eigenvalue is transferred back to the host.

Chapter 4 talks about hardware acceleration of Lanczos Iteration working on small-to-medium size data sets. We use explicit cache-blocking to minimize communication cost as shown in Equa-

tion (1.3). We design a single-precision hardware architecture for the Lanczos Iteration similar to [61] to minimize computation cost and thereby reduce overall runtime. We show how we can increase the silicon efficiency by exploiting pipeline parallelism and solving multiple independent symmetric extremal eigenvalue problems. We finally compare the results with GPU and show that FPGAs have better silicon efficiency due to explicit cache-blocking and pipeline parallelism. The problem setting for Chapter 4 is shown in Figure 1.1. In case of the FPGAs, we assume that the matrices are transferred from host memory to the internal memory of the FPGA, where they are blocked and are reused for all iterations. For GPUs, due to small on-chip memory, the matrices are transferred from host memory to the external memory, from where they are fetched in each Lanczos iteration. At the end, the eigenvalue is transferred back to the host.

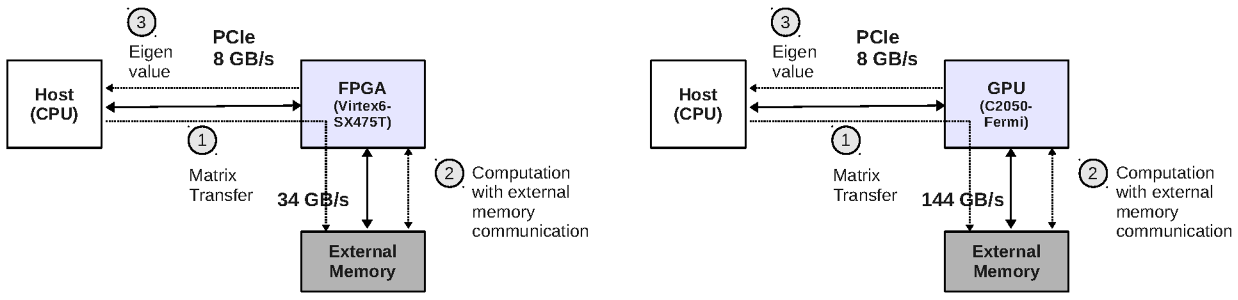


(a) FPGA Problem Setting (b) GPU Problem Setting

Figure 1.1: FPGA and GPU problem setting for solving small-to-medium scale problems. Step is not actually performed for both architectures
.

Chapter 5 is where we talk about accelerating the Lanczos Iteration for solving the large-scale symmetric extremal eigenvalue problem. We specifically discuss the matrix powers kernel, a fundamental block of Communication-Avoiding Lanczos, which performs $k$ SpMVs at the communication cost of single SpMV. We show how such an approach can minimize overall runtime on both GPUs and FPGAs by trading communication with redundant computation as shown in Equation (1.4). We present an algorithm for the matrix powers kernel that matches with the strengths of FPGAs to avoid redundant computation. We show how we can increase silicon efficiency of FPGAs and GPUs by carefully selecting the algorithmic parameter $k$. We also introduce our predictive model which shows how this algorithmic approach can increase efficiency which otherwise requires significant increase in memory bandwidth and latency. The problem setting for Chapter 5 and future chapters is shown in Figure 1.2. Both for FPGA and GPU, the matrices are first transferred to the external memory where they are partitioned in blocks. These blocks are then fetched to perform $k$ iterations of the Communication-Avoiding Lanczos. The final eigenvalue is then transferred back to the host.

Chapter 6 is where we introduce tall-skinny QR factorization (TSQR), another important kernel of Communication-Avoiding Lanczos. We show that the efficiency of the GPU is less than 1% for factorizing extremely tall and skinny matrices that not only arise in these iterative algorithms but many other real applications like stationary video background subtraction [8].



(a) FPGA Problem Setting (b) GPU Problem Setting

Figure 1.2: FPGA and GPU problem setting for solving large-scale problems. Step is not actually performed for both architectures

.

We observe that low efficiency is due to off-chip sharing of data across different kernels. We judiciously share data across these kernels using on-chip memory of the FPGAs and thereby eliminate off-chip communication which is unavoidable in case of GPU. We also introduce a custom architecture and show how we can increase silicon efficiency by exploiting pipeline parallelism to factorize multiple small matrices which arise in tall-skinny QR. We compare our results with state of the art QR routines on a GPU [8] and show the range of matrix sizes where FPGAs perform better than GPU.

Chapter 7 is where we compose all the kernels of Communication-Avoiding Lanczos and show how we can make use of architectural features of FPGAs to minimize communication in iterative solvers. We highlight GPU limitations in composition of kernels primarily due to off-chip sharing of data across these kernels. We show how the algorithmic parameter $k$ needs to be co-tuned for all the kernels. We finally conclude that a tight interaction between algorithm and architecture is required to enhance the silicon efficiency of FPGAs in accelerating iterative numerical algorithms.

Chapter 8 concludes our work and also provides potential areas for further research.

## 1.3 Statement of Originality

The four main original contributions of this thesis are each contained within each separate chapter. We provide details in the introduction chapter of these papers, however, we briefly summarize them here:

- A high-throughput architecture to accelerate multiple symmetric extremal eigenvalue problems. We show how we can use explicit cache-blocking and pipeline parallelism to increase silicon efficiency of FPGAs. We also provide comparison with GPU and show significant performance improvements. (Chapter 4, [77])

- A matrix powers kernel specifically targeting FPGAs in order to avoid redundant computation. A predictive model of GPU and FPGA to show how a tight algorithm-architecture interaction is required to enhance performance which otherwise requires significant increase in off-chip memory bandwidth and latency. (Chapter 5, [75])

- A custom architecture for enhancing performance of tall-skinny QR factorization. We show how we can make use of on-chip memory of FPGA to share data across different blocks which is only possible through off-chip shared memory in case of GPUs.(Chapter 6, [76])
- Application composition and communication optimization in iterative solvers using FPGAs.(Chapter 7, [78])

## 1.4 Publications

The following publications have been written during the course of this thesis:

- Abid Rafique, Nachiket Kapre, and George A. Constantinides, A high throughput FPGA-Based implementation of the Lanczos Method for the symmetric extremal eigenvalue problem. In Reconfigurable Computing: Architectures, Tools and Applications, pages 239−250. Springer, 2012.
- Abid Rafique, Nachiket Kapre, and George A. Constantinides, Enhancing performance of Tall-Skinny QR factorization using FPGAs. In Proceedings of the 22nd IEEE International Conference on Field Programmable Logic and Applications (FPL), pages 443−450, 2012.
- Abid Rafique, Nachiket Kapre, and George A. Constantinides, Application composition and communication optimization in iterative solvers using FPGAs. In Proceedings of the 21st IEEE International Conference on Field-Programmable Custom Computing Machines (FCCM), 2013.
- Abid Rafique, George A. Constantinides and Nachiket Kapre, Communication Optimization of Iterative Sparse Matrix-Vector Multiply on GPUs and FPGAs. accepted in IEEE Transactions on Parallel and Distributed Systems, 2014.

# 2 Applied Numerical Linear Algebra

In this chapter, we introduce basic problems of linear algebra. We introduce different matrix structures together with the numerical methods used to solve problems involving a particular structure. The scope of the thesis is also defined in this chapter. A major focus of this chapter is to identify communication as a basic problem in iterative numerical algorithms, which are used for solving sparse linear algebra problems. We discuss different approaches to minimize this communication on parallel and sequential architectures. Additionally, we also discuss related research questions that arise when these approaches are used for minimizing communication in iterative numerical algorithms on modern computing platforms like FPGA and GPU.

## 2.1 Standard Problems

There are four standard problems of numerical linear algebra, which often arise in engineering and scientific computations.

- *Linear systems of equations*: Given an $n \times n$ non-singular matrix $A$ and an $n \times 1$ vector $b$, solve $Ax = b$ to compute the $n \times 1$ unknown vector $x$. This problem arises in all fields of science and engineering: biology, chemistry, applied mathematics, finance, mechanical, civil and electrical engineering, *etc.* The most common source is the numerical solution of differential equations which arise in mathematical models of many engineering and physical systems. The numerical methods for solving such equations involve discretization of the system by finite element or finite difference methods. This discretization process leads to a linear system $Ax = b$ which is repeatedly solved to find an approximate solution to the differential equations. One such application is SPICE circuit simulation where $Ax = b$ is solved repeatedly to find solution of non-linear differential equations. Here $A$ is the matrix comprising conductances, $b$ is the vector containing the known currents and voltage quantities and $x$ contains the unknown branch currents and voltage quantities. Other application areas include mathematical optimization solvers [5], computational fluid dynamics [47] and finance [85].

- *Least squares problems*: Given an $m \times n$ matrix $A$ with $m > n$ and a $m \times 1$ vector $b$, compute the $n \times 1$ vector $x$ that minimizes $||Ax - b||^2$. When there are more observations than the number of variables, we solve $Ax = b$ to find the solution $x$ which minimizes sum of the squares of the errors made in the results of every single equation. Such problems arise in many areas like linear regression within machine learning [99] where we fit a polynomial or curve to experimental data, as well as in engineering applications such as signal processing [82].

- *Eigenvalue problems*: Given an $n \times n$ matrix $A$, find a $n \times 1$ vector $x$ and a scalar $\lambda$ such that $Ax = \lambda x$. A simple application is determining the non-negativity of the minimum eigenvalue $\lambda_{min}$ of the matrix $A$, a condition that needs to be checked to ensure semi-definiteness while solving semi-definite optimization programs [89]. Other applications include principal component analysis [53], eigen-based channel sensing [104], *etc.*

- *Singular value problems*: Given an $m \times n$ matrix $A$, find a $n \times 1$ vector $x$ and a scalar $\lambda$ such that $A^T Ax = \lambda x$. Examples include robust principal component analysis for stationary video background subtraction [8], where the vector $x$ corresponding to the largest $\lambda$ values of the matrix $A^T A$ represents the background information. This problem arises in many other areas like latent semantic indexing [36], total least squares minimization [49].

Although there are textbook methods which can be easily described to solve these problems, not all are tractable even for small problems. As an example, consider Cramer's rule [10] for solving $Ax = b$. Solving a 20×20 linear system using Cramer's rule on modern computing platforms can take millions of years using the usual definition of the determinant of a matrix [34]. Therefore, numerical methods are usually employed to solve these problems. Before discussing these numerical methods, we discuss a few matrix structures as they determine the selection of the numerical method.

## 2.2 Problem Structure

Based on the structure of the matrix $A$, we can broadly categorize linear algebra problems into three classes.

### 2.2.1 Dense

A problem is *dense* if a large number of elements in the matrix $A$ are non-zeros. Such problems arise while solving linear system of equations originating from semi-definite optimization problems [5], solution to electromagnetic scattering problems [74] and many other application areas. The dense matrices are stored as 2-D arrays.

### 2.2.2 Sparse

A problem is *sparse* if a large number of elements in the matrix $A$ are zeros as shown in Figure 2.1(a). A vast majority of engineering and scientific applications involve sparse matrices like solving partial differential equations, SPICE circuit simulation to name a few. Usually sparsity is exploited to minimize storage and computational cost. To store general sparse matrices, usually *compressed sparse row* (CSR) or *compressed sparse column* (CSC) format is used which does not require any knowledge of the sparsity pattern and also does not store any unnecessary elements. The CSR format puts all the matrix rows in contiguous memory locations. Three vectors are used: a vector (`val`) for storing floating-point values and the other two vectors (`col_ind`, `row_ptr`) storing the integer indices. The `val` vector contains all the non-zeros when the matrix is traversed in a row-wise fashion. The `col_in` stores the corresponding column number of the non-zero element. The `row_ptr` stores the location in the `val` vector that starts a row. We show this format using a toy example.

$$A = \begin{pmatrix} 10 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 3 & 0 & -1 & 0 \\ 0 & 0 & 0 & 4 \end{pmatrix}$$

The corresponding vectors will contain the following values for the sparse matrix $A$.

- `val`=[10, 1, 1, 3, -1, 4]
- `col_in`=[0, 2, 1, 0, 2, 3]
- `row_ptr`=[0, 2, 3, 5]

Instead of storing $n^2$ elements, the total storage requirement is $2nnz + n + 1$. Here, $nnz$ denotes the number of non-zeros in the matrix $A$. Although they are efficient in storage, they nonetheless have an overhead of indirect addressing for each scalar operation in a matrix-vector product, a kernel which is at the heart of iterative numerical algorithms used to solve sparse linear algebra problems (see Section 2.6).

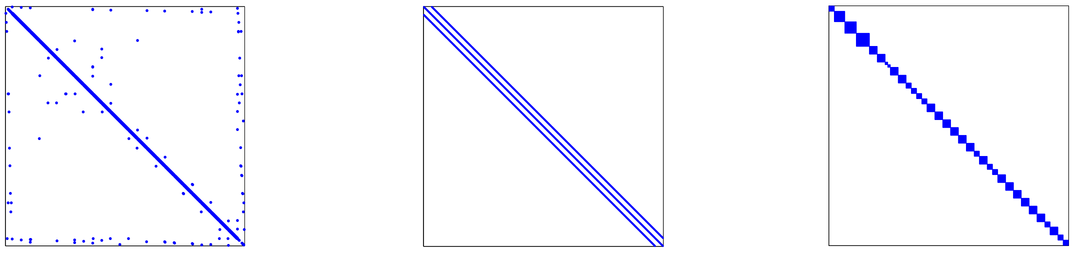

(a) `sparse` (b) `banded` (c) `block diagonal`

Figure 2.1: Matrix Structures.

### 2.2.3 Structured Sparse

Besides dense and sparse problems, there is another category where the problem contains matrices with a special structure, usually known as *structured sparse* matrices. A problem is considered structured sparse when the non-zeros in the matrix $A$ exhibit some structure. A common example is a tridiagonal matrix where all elements are zeros except the main diagonal, sub-diagonal and super-diagonal as shown below.

$$A = \begin{pmatrix} 10 & 1 & 0 & 0 \\ 3 & 1 & 2 & 0 \\ 0 & 4 & -1 & 2 \\ 0 & 0 & 1 & 4 \end{pmatrix}$$

Other possible structures include general banded matrices as shown in Figure 2.1(b) and block diagonal structure in Figure 2.1(c). The advantage with structured sparse matrix is that the storage scheme is efficient as compared to general sparse matrix formats like CSR and CSC. For banded matrices, we can store the sub-diagonals of the matrix in consecutive locations using *compressed diagonal storage* (CDS). Not only do we eliminate the vector required for storing row and column, but we can store the non-zeros in such a way as to make the matrix-vector product more efficient. We can store the example tridiagonal matrix $A$ in CDS format as shown below.

- `val(:,-1)`=[0, 3, 4, 1]
- `val(:, 0)`=[10, 1, -1, 4]
- `val(:,+1)`=[1, 2, 2, 0]

Here the second argument of the `val` matrix is the index of the sub-diagonal relative to the main diagonal. Although we need to store some zeros as well which are not present in the original matrix, but if the band of the matrix is small compared to the size of the matrix, then this overhead is minimal. See [20] for more details on the compressed diagonal storage for thin and wide bands. Instead of storing $n^2$ elements for a banded matrix of band size $b$, we need to store only $n \times b$ elements and there is no indexing overhead unlike CSR and CSC.

## 2.3 Numerical Methods

Depending on the structure of the matrix $A$, whether it is dense or sparse (including structured sparse), generally a broad class of *direct* or *iterative* numerical methods are used to solve these problems. However, the computational template of both of these classes comprises the following three steps:

1. The problem is first transformed into an easier to solve problem by converting the associated matrices into new matrices with special structure.
2. The transformed problem is then solved by exploiting the structure in the matrix.
3. The solution of the original problem is then recovered from the solution of the transformed problem.

We now briefly discuss direct and iterative methods along with their potential use.

### 2.3.1 Direct Methods

Direct methods are commonly used when the matrix $A$ is dense. In this case, we can get the solution in a fixed number of steps provided there is no roundoff error (an error generated due to finite data representation), and we need to perform all these steps before we can get the solution. For example, solving symmetric eigenvalue problem $Ax = \lambda x$ using the direct method of QR Algorithm [39] comprises following three steps.

- Transform the matrix $A$ into a tridiagonal matrix $T$ such that $A = Q^T AQ$.
- Use the QR Algorithm to compute the eigenvalues of the matrix $T$.

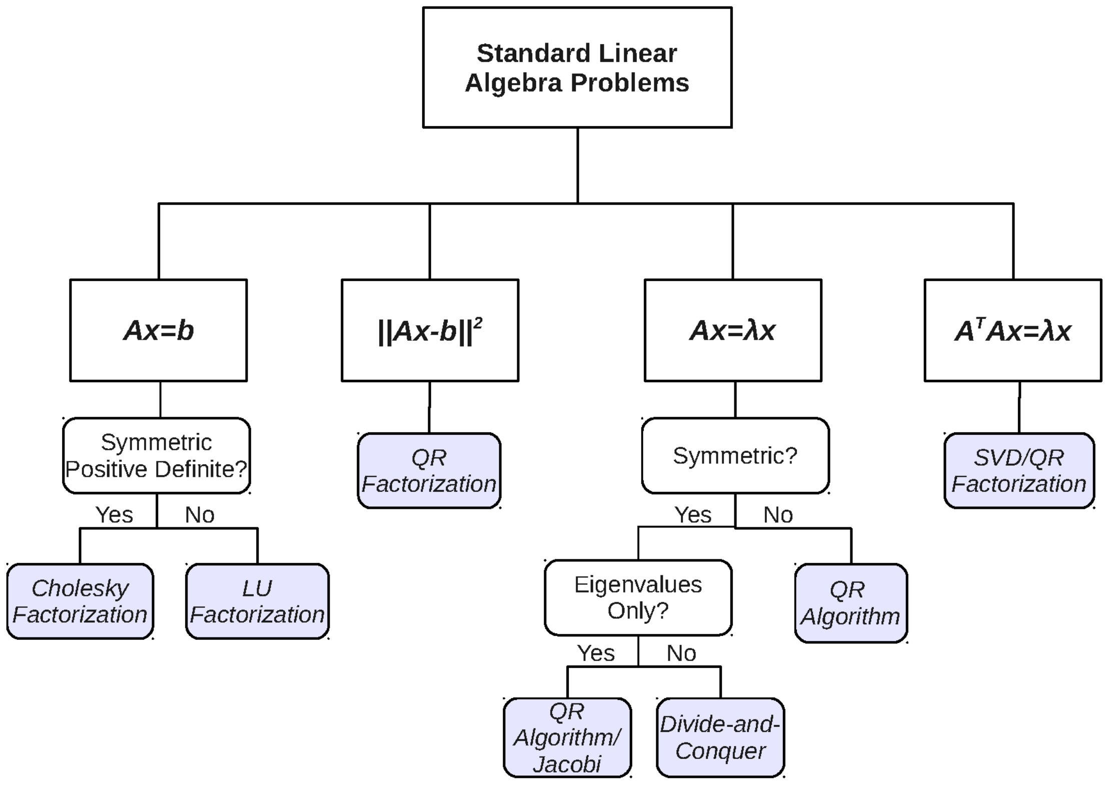


Figure 2.2: A taxonomy of direct methods for solving standard linear algebra problems. A positive definite matrix is the one where all the eigenvalues are greater than zero.

- Recover eigenvalues of the matrix $A$ from the eigenvalues of the matrix $T$.

Direct methods are stable as they are independent of the matrix entries. However, they have high computational complexity, *e.g.* tridiagonal reduction has a computational complexity of $O(n^3)$ and there is $O(n^2)$ complexity for finding the eigenvalues of the tridiagonal matrix $T$. Even if only a few eigenvalues are desired, we still have the same computational complexity as we need to perform all the steps. Additionally, with direct methods, we can not exploit sparsity in the matrix to reduce the computational complexity. We can classify direct methods based on the characteristics of the input matrix $A$ as shown by the decision tree in Figure 2.2. For example, in order to solve symmetric eigenvalue problem, one can choose either QR Algorithm or the Divide-and-Conquer method [34]. However, if additionally, one is interested in only eigenvalues and not the eigenvectors, then QR Algorithm is an appropriate choice due to its low complexity [34].

### 2.3.2 Iterative Methods

In contrast to the direct methods, iterative methods do not produce an exact solution after a finite number of steps. Instead, usually the error decreases by some fraction after each step (there are cases where the methods do not converge at all). One can terminate after the error decreases by a user-supplied threshold. The goal of these iterative methods is to decrease the error by a large amount after each iteration and do as little work per iteration as possible. The rate at which the iterative method converges depends on the spectrum (eigenvalue distribution) of the original matrix. Usually, the input matrix is transformed into a matrix with a favorable spectrum. This transformation matrix is called a *preconditioner* and without using this preconditioning the iterative method may even fail to converge.

Iterative methods are useful when only partial or a less accurate solution is desired, *e.g.* if one is interested in only in largest eigenvalue (in magnitude), the power method [39] can be used with a computational complexity of $O(rn^2)$ where $r$ is the number of iterations for a desired accuracy in the eigenvalue and $r<<n$. As a result, the computational complexity of iterative algorithms is much lower than that of $O(n^3)$ with the direct methods. Additionally, if the sparsity in the input matrix is exploited, this complexity is further decreased.

A vast majority of the iterative methods are based on a Krylov subspace $K_r(A,\, q_0) = [q_0,\, Aq_0,\, A^2q_0,\, .....\, ,\, A^{r-1}q_0]$ where $q_0$ is an initial $n \times 1$ random vector. The most common methods belonging to this class are the Conjugate Gradient (CG) [39], Generalized Minimum Residual Method (GMRES) [80] for solving $Ax = b$ and the Lanczos Iteration [39] for solving $Ax = \lambda x$. As these iterative methods only access the matrix $A$ through a matrix-vector multiplication, this provides opportunity to fully exploit sparsity or any special structure of $A$. Therefore, for large sparse problems, iterative methods are the only choice. Depending on the characteristics of the matrix, whether it is symmetric or non-symmetric, positive definite or indefinite (eigenvalues can be less than zero), we can select a particular iterative method from [16] for $Ax = b$ and [12] for solving $Ax = \lambda x$. We show the taxonomy of iterative methods for the four standard linear algebra problems in Figure 2.3.

## 2.4 Thesis Scope

Although the Krylov subspace-based iterative methods have nice properties of low complexity and better exploitation of sparsity in the input matrix, they are nonetheless *communication-intensive* algorithms due to the dominant matrix-vector multiplication. In this thesis, the main focus is on optimizing communication within these Krylov subspace-based iterative methods while accelerating solution to standard numerical linear algebra problems using hardware accelerators like GPU and FPGA. In this work, we target very large structured sparse banded matrices due to two main reasons. First, computations on such 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 [30]. Secondly, they naturally arise in numerous scientific computations like stencils in partial differential equation (PDE) solvers [84] and semi-definite optimization problems [1]. We take the Lanczos Iteration [39] as a case study to accelerate the solution to symmetric extremal eigenvalue problem, a sub-class of eigenvalue problem where one is interested in only either minimum or maximum eigenvalue of a symmetric matrix. Since the Lanczos Iteration is an integral part of all modern Krylov subspace-based methods including CG, MINRES, *etc.*, our results are directly applicable to solving general eigenvalue problem as well as other standard problems including solution to sparse linear systems of equations. Additionally, while accelerating the Lanczos Iteration for large-scale problems, we use the recently proposed communication-

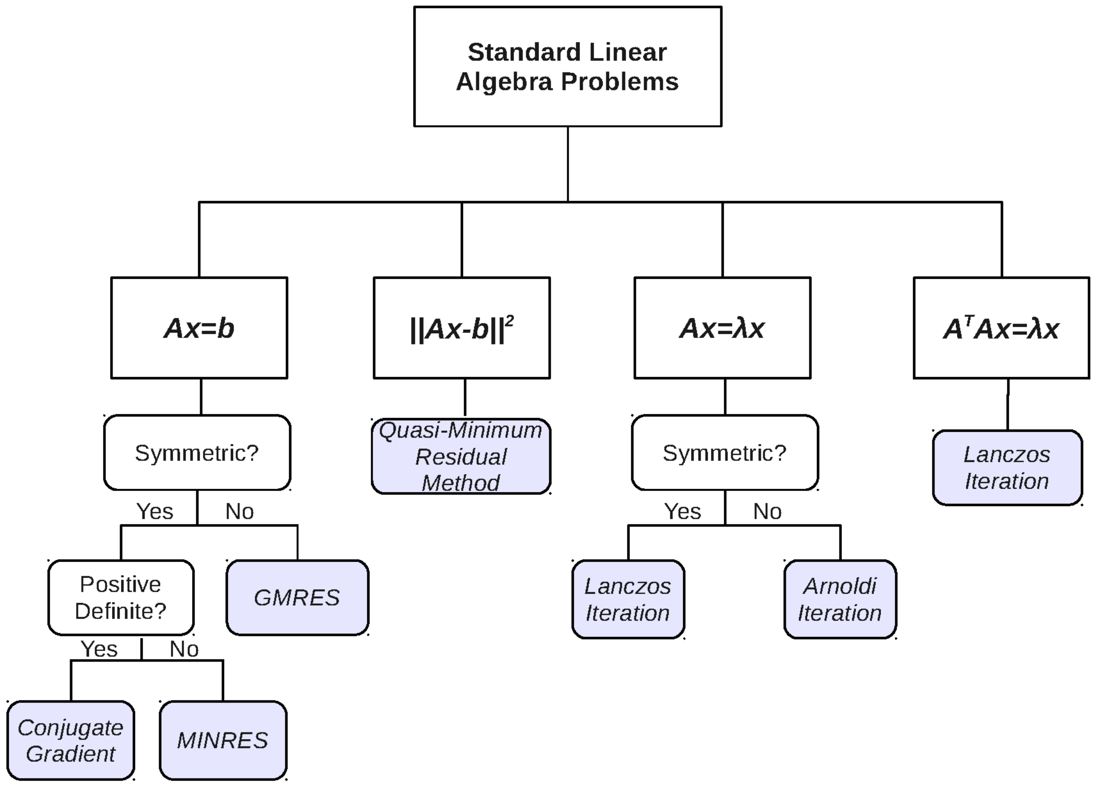


Figure 2.3: A taxonomy of iterative methods for solving standard linear algebra problems. A positive definite matrix is the one whose all eigenvalues are greater than zero.

avoiding approach [48]. Such a communication-avoiding approach includes QR factorization of a *dense* tall-skinny matrix in each iteration, a numerical method which is used to solve the least squares problem as well as singular value problem involving tall-skinny matrices (see Figure 2.2). In a nutshell, we aim to accelerate all four standard applied numerical linear algebra problems.

## 2.5 Symmetric Extremal Eigenvalue Problem

Let $A$ be an $n \times n$ real symmetric matrix, the eigenvalue problem is posed as finding the eigenvalues $\lambda$ and the corresponding eigenvectors $x$ such that

$$Ax = \lambda x \tag{2.1}$$

The eigenvalues $\lambda$ of a real symmetric matrix are also real and usually expressed in descending order $\lambda_1 > \lambda_2 > \lambda_3 > ,..... > \lambda_n$. All these eigenvalues are collectively called the *spectrum* of $A$. The symmetric extremal eigenvalue problem is a subset of this problem where we are only interested in finding $\lambda_1$ or $\lambda_n$ that satisfies the above equation. Such kind of eigenvalue problem arises in many applications. We consider a few important classes of applications where this problem is solved online.

### 2.5.1 Applications

#### Line Search in Interior-Point Method for Semi-definite Programming

Semi-definite Programming (SDP) [94] is a sub-class of convex optimization [22] where a linear objective function in matrix variable $X$ is minimized subject to some equality constraints and an additional constraint that $X \succeq 0$, *i.e.* positive semi-definite matrix (where all eigenvalues are greater than equal to 0). A most popular method for solving SDP is the primal-dual interior-point method [5] which is an iterative algorithm used to find the optimal values of the primal matrix variable $X \in \mathbb{R}^{n\times n}$ and dual matrix variable $Z \in \mathbb{R}^{n\times n}$ simultaneously. SDP has lot of applications in control theory [93] and polynomial optimization [63]. Of particular interest are the applications where the SDP is solved online [63] at each time instant and the sampling time is on the order of a few milliseconds.

In each iteration of the primal-dual interior-point method, starting with some initial values of $X$ and $Z$, we compute search directions $\Delta X$ and $\Delta Z$. In order to update $X = X + \alpha_{\mathrm{p}}\Delta X$ and $Z{=}Z + \alpha_{\mathrm{d}}\Delta Z$, we estimate $\alpha_{\mathrm{p}}$ and $\alpha_{\mathrm{d}}$ by solving the following *line search* problem for the newly computed search directions.

$$\alpha_{\mathrm{p}} = \max\{\alpha \in [0,1] : X + \alpha\Delta X \succeq 0\} \tag{2.2}$$

$$\alpha_{\mathrm{d}} = \max\{\alpha \in [0,1] : Z + \alpha\Delta Z \succeq 0\} \tag{2.3}$$

The condition $X + \alpha\Delta X \succeq 0$ and $Z + \alpha\Delta Z \succeq 0$ involves checking whether these matrices are positive semi-definite, *i.e.* whether the minimum eigenvalue of these matrices is greater than equal to zero.

#### Eigenvalue Based Sensing

Channel sensing, *i.e.* detecting the presence of any primary user, is one of the fundamental tasks in cognitive radio. Matched filters (MF) [81], [25] require prior knowledge of the channel whereas energy detection (ED) [92], [81] is optimal for detecting independent and identically distributed (i.i.d) signals but not optimal for detecting correlated signals. A covariance based approach is presented in [104] where a covariance matrix is formed from few samples of the received signals. The maximum eigenvalue of the covariance matrix is used as the test statistic. This maximum eigenvalue detection (MED) method can be used without any knowledge of the channel and the signal source. It is shown in [104] that MED is optimal for correlated signals whereas it approaches the ED for i.i.d signals.

For signal detection there are two hypotheses: $H_0$, signal does not exist and $H_1$, signal exists.

$$\begin{aligned} H_0 : x(n) &= \eta(n) \\ H_1 : x(n) &= s(n) + \eta(n) \end{aligned}$$

where $s(n)$ is the transmitted signal that passes through the wireless channel after going through path loss and fading, and $\eta(n)$ is white noise which is i.i.d with mean zero and variance $\sigma_\eta^2$. We can estimate the statistical covariance matrices of $x(n)$ and $s(n)$ as:

$$\begin{aligned} R_x &= E[x(n)x(n)^T] \\ R_s &= E[s(n)s(n)^T] \end{aligned}$$

Let $\lambda_{\max}$ and $\rho_{\max}$ be the maximum eigenvalue of $R_x$ and $R_s$ respectively then $\lambda_{\max} = \rho_{\max} + \sigma_\eta^2$. If the signal is present then $\lambda_{\max} > \sigma_\eta^2$. Thus channel sensing can be cast as a symmetric extremal eigenvalue problem where we have to find the maximum eigenvalue $\lambda_{\max}$ of the symmetric covariance matrix $R_x$ to test the two hypotheses $H_0$ and $H_1$.

### 2.5.2 Iterative Methods for Symmetric Extremal Eigenvalue Problem

Iterative methods are natural choice for the symmetric extremal eigenvalue problem as direct methods like the QR Algorithm perform all the steps even if only the maximum or minimum eigenvalue is desired. As a result, the computational complexity is still $O(n^3)$. On the other hand, iterative methods can approximate the extremal eigenvalue after a few steps and therefore offer lower computational complexity. We briefly survey iterative methods for solving the symmetric extremal eigenvalue problem.

#### Power Method

The power method is the most simple iterative method. The basic idea is that if a given vector is repeatedly multiplied by the symmetric matrix $A$, then it will ultimately lie in the direction of the eigenvector corresponding to the eigenvalue which has the largest absolute value. The power method is shown in Algorithm 1. As shown the only dominant operation is the matrix-vector multiplication. The problem with the power method is that it gives eigenvalue with the maximum absolute value, not the maximum numerical value. It can be used for a particular class of symmetric matrices which have either non-negative or negative eigenvalues but not both. However, in our application setting we are interested in finding extremal (minimum or maximum) eigenvalue of a general symmetric matrix and therefore this method can not be applied.

**Algorithm 1** Power Method

```
Require: Symmetric matrix $A \in \mathbb{R}^{n\times n}$, initial orthonormal vector $q_0 \in \mathbb{R}^{n\times 1}$ and number of iterations $r$
1: for $i = 1$ to $r$ - 1 do
2:     $q_i := Aq_{i-1}$
3:     $q_i := q_i/\|q_i\|_2$
4:     $\lambda_i := q_i^T A q_i$
5: end for
6: return $\lambda_{r-1}$ as the eigenvalue with the largest absolute value and $q_{r-1}$ as the corresponding eigenvector.
```

**Algorithm 2** Inverse Iteration

```
Require: Symmetric matrix $A \in \mathbb{R}^{n\times n}$, initial orthonormal vector $q_0 \in \mathbb{R}^{n\times 1}$ and number of iterations $r$ and shift $\sigma$
1: for $i = 1$ to $r$ - 1 do
2:     $q_i := (A - \sigma I)^{-1} q_{i-1}$
3:     $q_i := q_i/\|q_i\|_2$
4:     $\lambda_i := q_i^T A q_i$
5: end for
6: return $\lambda_{r-1}$ as the eigenvalue closest to $\sigma$ and $q_{r-1}$ as the corresponding eigenvector.
```

**Inverse Iteration**

In order to find not only the eigenvalue with the maximum absolute value but any desired eigenvalue closest to some initial guess $\sigma$, an inverse iteration method is usually used shown in Algorithm 2. Here, we have to solve a linear system in each iteration to get a new vector. The method is superior to the power method but is only useful once we have an approximate eigenvalue $\sigma$. It is particularly useful when we have an eigenvalue and we are interested in finding the corresponding esigenvector. In our case, the inverse iteration is not applicable because of the prior knowledge of the eigenvalue to be computed.

**Lanczos Iteration**

Starting with any given vector $q_0$, $r$ iterations of either the power method or inverse iteration produce a sequence of vectors $q_0$, $q_1$, $q_2$, $q_3$, ..... , $q_{r-1}$. These vectors span the Krylov subspace. In case of the power method this subspace is $K_r(A,\, q_0) = [q_0,\, Aq_0,\, A^2q_0,\, .....\, ,\, A^{r-1}q_0]$ and for inverse iteration it is $K_r((A - \sigma I)^{-1},\, q_0)$. Both the methods use only the last vector from this subspace to compute the eigenvector and the corresponding eigenvalue. Since $K_r$ has dimension $r$ (in general), we can actually use it to compute the $r$ best approximate eigenvalues and eigenvectors. The Lanczos Iteration is also a Krylov subspace method which utilizes the whole subspace $K_r(A,\, q_0)$ and approximates the eigenvalues of $A$. It is based on the same idea of similarity transformation on $A$ ($Q^TAQ = T$) which is a usual first step in direct methods to reduce a symmetric matrix $A$ to a tridiagonal matrix $T$. The matrix $T$ has the dimensions of $A$ and its eigenvalues are exactly the same as the eigenvalues of the original matrix A.

Instead of doing a complete tridiagonalization, the Lanczos Iteration performs partial tridiagonalization of the matrix A.

$$Q_r^T A Q_r = T_r$$

$$T_r = \begin{pmatrix} \alpha_1 & \beta_1 & 0 & \cdots & 0 \\ \beta_1 & \alpha_2 & 0 & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & \cdots & 0 & \alpha_{r-1} & \beta_{r-1} \\ 0 & \cdots & 0 & \beta_{r-1} & \alpha_r \end{pmatrix}$$

The eigenvalues of the tridiagonal matrix $T_r$ are good approximations to the $r$ eigenvalues of the input matrix $A$ and the approximations get better with increasing number of iterations. It has been shown in [34] that the extremal eigenvalues of $T_r$ converge to the extremal eigenvalues of $A$ much faster than the internal eigenvalues. The entries of the tridiagonal matrix $T_r$ are computed using Algorithm 3. We show the convergence behavior of the Lanczos Iteration in Figure 2.4. We first plot $\lambda_{\max}(A)$ = 1.407 and $\lambda_{\min}(A)$ = -2.084. We then plot $\lambda_{\min}(T_r)$ along with 2 other smallest eigenvalues and $\lambda_{\max}(T_r)$ along with 2 other largest eigenvalues. We show that $\lambda_{\min}(T_r)$ converges to $\lambda_{\min}(A)$ and $\lambda_{\max}(T_r)$ converges to $\lambda_{\max}(A)$ in a few iterations as compared to the internal eigenvalues.

We observe that the Lanczos Iteration is the only method for the extremal eigenvalue problem because since it converges to both ends of the spectrum, we can find algebraically minimum or maximum eigenvalue. We now introduce the kernels involved in the Lanczos Iteration and discuss issues related to their computational performance.

**Algorithm 3** Lanczos Method in Exact Arithmetic [39]

```
Require: Symmetric matrix $A \in \mathbb{R}^{n\times n}$, initial orthonormal vector $q_0 \in \mathbb{R}^{n\times 1}$ and number of
    iterations $r$, $\beta_0 = 0$.
1: for $i = 1$ to $r$ do
2:     $q_i := Aq_{i-1}$
3:     $\alpha_i := q_i^T q_{i-1}$
4:     $q_i := q_i$ - $\alpha_i q_{i-1}$ - $\beta_{i-1} q_{i-2}$
5:     $\beta_i = \|q_i\|_2$;
6:     $q_i := q_i/\beta_i$
7: end for
8: return Tridiagonal matrix $T_r$ containing $\beta_i$ and $\alpha_i$ $i$=1, 2,..., r
```

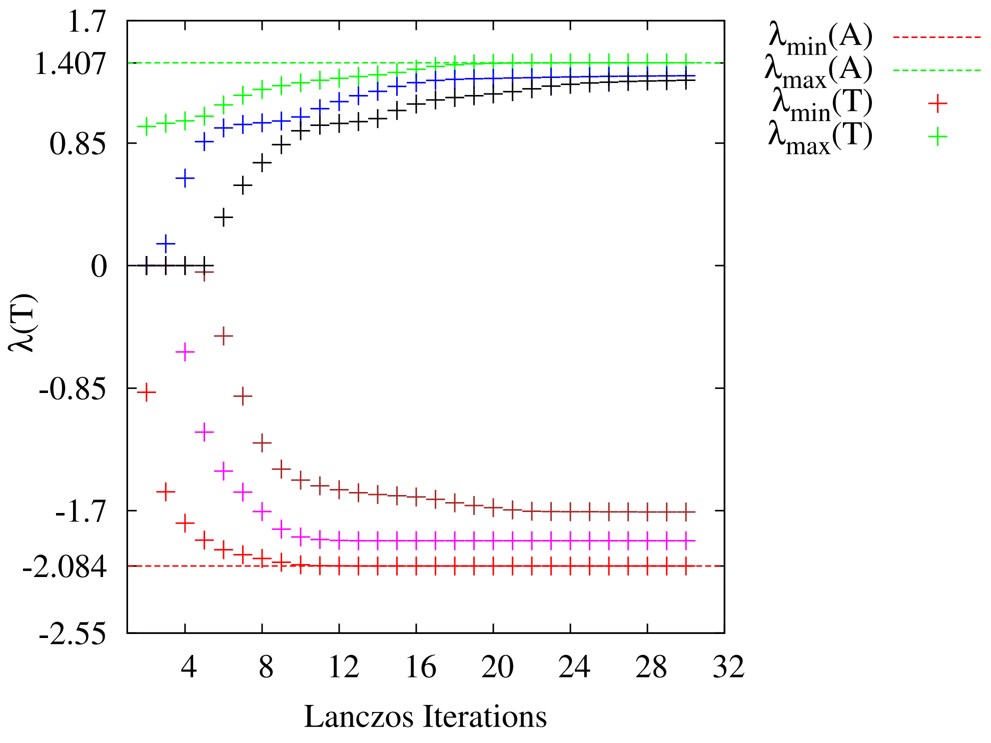


Figure 2.4: Lanczos Convergence

## 2.6 Computational Kernels of the Lanczos Iteration

If we analyze a Krylov method like the Lanczos Iteration shown in Algorithm 3, there are three kernels:

- Sparse matrix-vector multiplication (SpMV), $y = Ax$.

- Scalar-vector multiplication followed by vector-vector addition ($axpy$), $y = ax + y$.
- Dot products, $a = y^T x$.

After introducing communication formally in the next section, we show that the performance of these kernels is bounded by the communication cost. There is a data dependency between these operations. Due to this data dependency, the kernels' communication cost can not be overlapped by their computation cost. Therefore, if the kernels are communication-bound, so is the Krylov method which involves sequential execution of these kernels one after another. We now discuss the performance issues related to the communication cost (see [48] for details).

## 2.7 Communication and performance issues

In this section we describe what communication is, and why it is desirable to avoid communication for accelerating iterative numerical methods. We introduce the LogP [28] model which measures performance of an algorithm on computing platforms in terms of communication and computation cost. We provide evidence that communication is expensive relative to computation and this will likely be the case for many years to come. In Section 2.8, we discuss the impact of communication on the performance of kernels in the Lanczos Iteration and in Section 2.9, we introduce different approaches for avoiding this communication.

### 2.7.1 Communication is Data Movement

Communication refers to *data movement* between various levels of a memory hierarchy in sequential architectures and between processors in parallel architectures. In a parallel architecture, communication can take different forms:

- Message passing between processors in a distributed architecture.
- Cache coherency traffic in a shared memory architecture.
- Data transfer between a host (CPU) and a hardware accelerator (FPGA or GPU).

In a sequential architecture, data movement involves a slow memory (disk or a DRAM) and a fast memory (DRAM or a cache). This data movement may be under manual control (FPGA) or automatic control (CPU).

### 2.7.2 Modelling Performance

The total runtime of an algorithm involves computation and communication cost. A LogP model captures these costs separately. In this model, communication occurs in the form of *messages*. The message is used regardless of whether it is a distributed or shared-memory architecture or whether the data is moved between slow and fast memory in a sequential architecture. The time required to transfer a message comprising $M$ words can be modelled as:

$$t_{comm} = \alpha + \beta M. \tag{2.4}$$

where $\alpha$ is the latency (in seconds) and $\beta$ corresponds to the inverse bandwidth (seconds per word). If we represent $\gamma$ as the cost of performing a single floating-point operation, the computation cost of $M$ floating-point operations is then:

$$t_{comp} = \gamma M. \tag{2.5}$$

Most models do not take into account both latency and bandwidth terms, *e.g.* the introduction of *memory wall* by Wulf and McKee [103] only refers to the memory bandwidth. However, for all communication media as well as DRAM access, there is a different cost in sending a long and a short message. This is because of the latency term. We, therefore, use both these quantities as latency does matter even in dense linear algebra operations [15].

### 2.7.3 Communication is expensive

The communication problem has historical roots in the memory wall problem which is a hard upper bound on the performance of a sequential architecture due to memory bandwidth being slower than the instruction throughput. There is then the *power wall*, which constrains performance of the sequential architecture and shifts the focus towards parallel processing leading to an era of multi-cores and many-cores. All these techniques to improve sequential performance either lead to too much power dissipation, or have diminishing returns. More parallelism means more communication which then exacerbates the memory wall problem. Even radical changes in hardware technology is not going to break either of these walls any time soon, which means we should look for algorithmic approaches which minimize communication even if a few redundant computations are performed.

#### A. The Memory Wall Problem

The memory wall problem refers to exponentially increasing gap between the main memory bandwidth and the instruction throughput for performing floating-point operations. Wulf and McKee [41] coined this term. They argued that if a processor needs to perform $M$ floating-point operations and other non-memory instructions, if it then requires within a constant factor of that number of memory read/write operations, it will have *memory-bound* performance. It is true even if cache has infinite bandwidth and even true if the whole data can fit in the cache, as long as the data is not in the cache when the program starts. SpMV is particularly affected by the memory wall problem, because there are as many memory operations as there are floating-point operations.

#### B. Latency

It is well-known that DRAM and other storage media can be accessed with full bandwidth only if the data is contiguous or has a regular layout. The linear algebra algorithms operating on sparse data sets do not access data in a regular way, so they must pay latency cost to access their data. However, there is a big gap between improvements in latency cost and the cost of actually performing the floating-point operations. Graham *et al.* [41] show that between 1988 and 2004, the computational performance increases by 59% per year whereas the DRAM latency only decreases by 5.5%. In 1988, one floating-point operation took six times as long as fetching a single word from the main memory, but in 2004, one memory operation took as long as 100 floating-point operations. Even after the introduction of multi-cores, this trend continues with the DRAM latency being three order of magnitude higher than processor cycle time [18]. The latency of sending messages over a network follows similar trends like the DRAM latency. For example, sending a short message from one node to another over a network interface may take time as long as thousands of floating-point operations performed at the individual node [18].

In order to avoid the memory wall problem, hardware architects usually use deep memory hierarchies. However, for algorithms with lower temporal locality, this may exacerbate the situation due to the cache overhead. We will see in Section 2.8, that the Lanczos Iteration and most other Krylov methods lie in this category.

#### C. Bandwidth

Like latency, there is an exponential gap between computational performance and the memory bandwidth. While the computational performance has increased by 59% per year, the memory bandwidth only increases at annual rate of 26% [41]. Typical ways to improve bandwidth include:

- Interleaving data across different devices, with the overall bandwidth equal to the memory bandwidth multiplied by the number of attached devices. For example, in DRAM, this can be achieved using multiple *banks*.
- In shared-memory parallel architectures, partitioning the storage into regions each of which is assigned to a subset of processors. This increases effective bandwidth of a processor to its local subset of storage. For DRAM, this approach is known as Non-Uniform Random Memory Access (NUMA).

Although these bandwidth enhancement approaches increase the access bandwidth, they may increase latency as well [18]. For example, interleaving data across multiple banks increases the overhead of accessing a single word of data. Furthermore, increasing the bandwidth of a storage device or network interface increases the complexity of the underlying hardware, and therefore cost and energy consumption.

## 2.8 Communication in the Lanczos Iteration

In Section 2.6, we identified three kernels in the Lanczos Iteration: SpMV, *axpy* and dot products. We now analyze these kernels to see how the performance of these kernels is bounded by the communication cost.

### 2.8.1 Sparse matrix-vector multiply

#### A. Sequential sparse matrix-vector multiply

It is well-known that memory operations dominate the cost of performing sequential SpMV. The operation $y = Ax$, with an $n \times n$ sparse matrix $A$ and dense vectors $x$ and $y$, can be computed as

$$y_i = \sum_{0 \le j \le n-1,\ A_{ij} \ne 0} A_{ij} x_j \tag{2.6}$$

Different sparse matrix formats compute this sum in different ways. However, all the non-zero entries of $A$ need to be fetched at least once. If they are not available in cache, they need to be fetched at no greater than the memory bandwidth. For each word accessed from main memory, there are only two floating-point operations (multiply followed by add operation). Therefore, there is an *arithmetic intensity* (number of floating-point operations per each word fetched from the memory) of no more than two. As an example, the memory bandwidth of an NVidia C2050 GPU is 144 GB/s. For single-precision floating-point numbers we fetch four bytes for each word to perform two floating-point operations. As a result, the performance of C2050 GPU is bounded from above by this memory bandwidth and we can achieve a maximum performance of up to 72 GFLOPs which is approximately 7% of its peak single-precision performance (1030 GFLOPs). This is in sharp contrast to the performance of dense matrix-matrix multiplication which can achieve 80% of the peak single-precision performance. This is because the dense matrix-matrix multiplication has a higher arithmetic intensity ($O(n^2)$ memory operations and $O(n^3)$ floating-point operations).

Some optimizations can improve the performance of SpMV operation for certain types of matrices. For example, we can store a group of non-zeros of $A$ as a block−a technique called *register blocking*−and we can achieve up to a small constant factor of reuse of matrix entries [101]. Additionally, we can use another kind of optimization−*cache blocking*−which provides reuse in the source vector $x$ [69]. None of these optimizations can make the performance of the sequential SpMV any way closer to the performance of the dense matrix-matrix multiplication.

### B. Parallel sparse matrix-vector multiply

Compared to sequential SpMV, it is difficult to characterize the performance of SpMV on parallel architectures. The factors affecting the performance of SpMV are:

- structure of the matrix,
- how the vectors and the matrix is partitioned and divided among the processors,
- whether the partition assigned to a single processor fits in the cache.

There are many ways to partition and divide the matrix $A$ among the parallel processors. A common method is to

1. partition the vector $y$ into blocks $y_1, y_2, \ldots, y_{P_e}$ and assign these blocks to $P_e$ processors,
2. partition the matrix $A$ into blocks $A_{IJ}$ with $1 \le I, J \le P_e$,
3. assign $x_I$,$y_I$ and $A_{IJ}$ to the processor $I$.

This is a 1-D block row decomposition and the overall SpMV looks like:

$$\begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_{P_e} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1P_e} \\ A_{21} & A_{22} & \cdots & A_{2P_e} \\ \vdots & \vdots & \vdots & \vdots \\ A_{P_e1} & A_{P_e2} & \cdots & A_{P_eP_e} \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_{P_e} \end{pmatrix}$$

Now processor $I$ computes its vector partition $y_I$ by performing the following summation:

$$y_I = \sum_{1 \leq j \leq P_e} A_{IJ} x_J \tag{2.7}$$

Each $A_{IJ}$ is a sparse matrix partition which processor $I$ can access directly. If $I \neq J$, the processor $I$ needs to fetch the entry $x_J$ from the $J$th processor. In a distributed-memory architecture, this communication happens in the form of messages whereas in a shared-memory system we access memory region assigned to a different processor. In both cases, processor $I$ has to pay a latency as well as bandwidth cost in order to access $x_J$.

The above simple 1-D decomposition shows that in parallel SpMV, most of the time is spent in performing sequential SpMV on each processor. It is hard to estimate how much time is actually spent in sequential SpMV and how much time is consumed in communication between different processors. This is because there are different kinds of parallel processors and sparse matrices. Nevertheless, from experience, if both communication and computation loads are balanced evenly among the processors, the cost of communication is still enough to justify complicated reordering in order to avoid communication volume [102]. This indicates that inter-processor communication is an important part of parallel SpMV, and in order to achieve higher performance we should minimize it as much as possible.

### 2.8.2 *axpy* and dot products

Besides SpMV, *axpy* and dot products are also communication-bound as they require $O(n)$ memory operations and also perform $O(n)$ floating-point operations. For *axpy*, in the parallel case, there is no inter-processor communication but we may need synchronization before using the results. In the sequential case, the cost of reading the vectors from the main memory exceeds that of actually doing the computation. For parallel dot products, we need to perform sequential dot products at each processor followed by a scalar reduction operation. This scalar reduction is a latency-bound operation and its latency cost competes with the bandwidth cost of performing the dot products.

## 2.9 Avoiding Communication in the Lanczos Iteration

In the last section, we identified that all the kernels in the Lanczos Iteration and hence all Krylov methods are communication-bound. Improving communication performance through hardware modifications may not be entirely useful, increasing bandwidth might end up increasing the latency as well. Therefore, the only logical choice is to *avoid* communication wherever possible.

Usually, communication can be avoided in one of the two possible ways:

- *Blocking* the matrix in the fast memory and reuse it as much as possible [20] [61]. This is only applicable for small dense matrices which are loaded once from the main memory and are reused for all the iterations. This is even true for matrices having special structure like block-diagonal matrices shown in Figure 2.1(c). Here, disjoint blocks can be assigned to different processors and there is no need of inter-processor communication as no vector entries need to be shared.

- Designing *communication-avoiding* algorithms that aim at minimizing communication even if we have to perform redundant floating-point operations.

In this work, we have used both of these approaches to avoid communication on modern hardware accelerators like FPGA and GPU (see Chapter 3 for details about these computing platforms). While the first approach is well-known, we explore communication-avoiding approaches in the next section that tend to optimize communication in the Lanczos Iteration and other Krylov methods.

### 2.9.1 Block Lanczos

Block Lanczos and other Block Krylov methods are related to the Krylov subspace methods. The Krylov subspace method works with the subspace $K_r(A, q_0) = [q_0, Aq_0, A^2q_0, \ldots.. , A^{r-1}q_0]$, where $q_0$ is a single vector. On the other hand, Block Krylov methods operate on multiple vectors to generate

$$K_r(A,\ Q_0) = [Q_0,\ AQ_0,\ A^2Q_0,\ \ldots..,\ A^{r-1}Q_0] \tag{2.8}$$

where $Q_0 \in \mathbb{R}^{n\times p}$ is a dense matrix comprising linearly independent columns. Block methods originated as modifications of the Lanczos Iteration for finding eigenvalues of a sparse symmetric matrix [29] [40]. The Lanczos Iteration can not resolve multiple eigenvalues or closely located eigenvalues. Block Lanczos method resolves a cluster of eigenvalues as it operates on a different subspace. The parameter $p$ is selected based on the expected number of eigenvalues that we want to resolve.

Although Block Lanczos method was originally used for qualitative purposes, it nonetheless can help in performance improvement as well. Computing the subspace in Block Lanczos require a different kernel, sparse multiple-vector multiplication (SpMM), which multiplies the input matrix $A$ with multiple vectors in the dense matrix $Q_0$ as shown in Algorithm 4. This new kernel SpMM can be implemented in a communication-avoiding manner such that the matrix $A$ is read only once. The parameter $p$ provides better tradeoff between computation and communication cost [39]. In this case, for SpMM, we have $O(nnz)$ memory operations while we perform $O(nnz\times p)$ floating-point operations. Here, $nnz$ is the number of non-zeros in the input matrix $A$. This method has arithmetic intensity of $p$ which can be exploited by parallel architectures like multi-core, GPUs and FPGAs. The benefit is reduction in the communication cost as the number of iterations will be reduced to obtain eigenvalue of a desired accuracy as shown in Figure 2.5. It is generally assumed that Block Lanczos essentially reduces the communication cost by $p$ [12]. However, in practice, it is seen that reduction is not a factor of $p$ and that is because of slow convergence due to loss of orthogonality among the Lanczos vectors. This loss of orthogonality is due to finite precision [34]. The choice of block size $p$ is an important design parameter which not only determines the resolution factor for eigenvalues but also provides the tradeoff between computation and communication cost.

**Algorithm 4** Block Lanczos [39]

```
Require: Symmetric matrix $A \in \mathbb{R}^{n\times n}$, $Q_0 \in \mathbb{R}^{n\times p}$ and number of iterations $r1$,
    ($B_0 = 0$).
1: for i = 1 to $r1$ do
2:     $Q_i := AQ_{i-1}$        $C_i \in \mathbb{R}^{n\times p}$
3:     $M_i := Q_{i-1}^T Q_i$
4:     $Q_i := Q_i - Q_{i-1}M_i - Q_{i-2}B_{i-1}$
5:     $[Q_i, B_i] := \mathrm{qr}(Q_i)$
6:
7: end for
8: return Block tridiagonal matrix $T_{r1}$ containing $M_i$ and $B_i$ i=1, 2, ..., $r$ where $M_i \in \mathbb{R}^{p\times p}$ symmetric matrix
   and $B_i \in \mathbb{R}^{p\times p}$ upper triangular matrix.
```

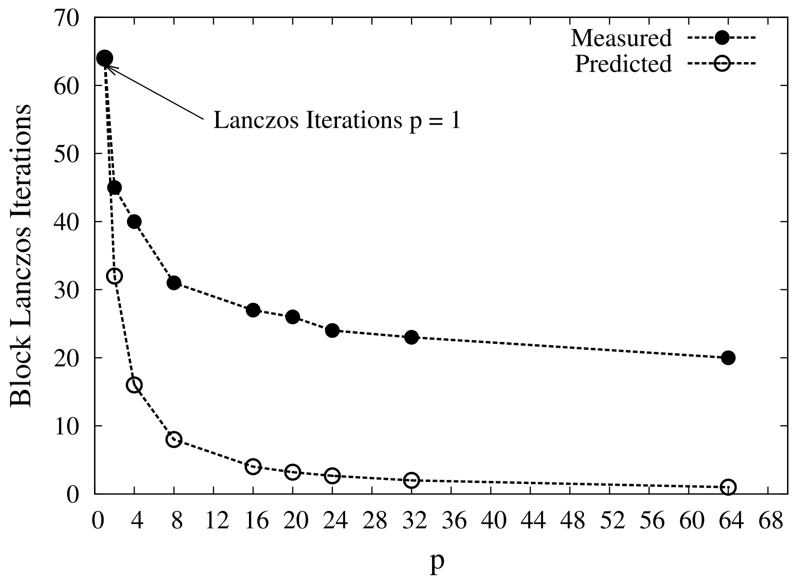


Figure 2.5: Iterations vs. $p$ for extremal eigenvalue computation with accuracy $10^{-3}$. The predicted value is calculated theoretically and not with any model.

### 2.9.2 Communication-Avoiding Lanczos

While Block Lanczos is mainly used to resolve cluster of eigenvalues, communication-avoiding Lanczos (CA-Lanczos) [48], on the other hand is designed to improve performance by moving $k$ steps into the Lanczos Iteration at once. CA-Lanczos is based on $s$-step Krylov methods [58] which break the dependency between SpMV and other operations including *axpy* and dot products. CA-Lanczos generates $k$ basis vectors at a time by repeated SpMV operation ($q_0$, $Aq_0$, $A^2q_0$, ..., $A^kq_0$) and as a result, overall communication cost can potentially be decreased by $k$. Although CA-Lanczos has not been studied previously from performance perspective, however such a communication-avoiding approach has been successfully used to speed up the solution to linear systems of equations. Mohiyuddin *et al.* [65] presented a communication-avoiding GMRES (CA-GMRES) to solve linear system of equations and demonstrated up to 4.1× speedup over standard GMRES method for solving banded matrices on an Intel 8-core clovertown machine. In this work, we intend to use CA-Lanczos as a potential communication-avoiding approach to minimize communication on hardware accelerators like FPGA and GPU. We first introduce some notation for CA-Lanczos followed by a brief description of the algorithm. Finally, we discuss the computational kernels of CA-Lanczos which help in avoiding communication to achieve higher performance.

## Notation

We borrow the following notations from [48].

- Vectors and group of vectors
  - $v_i$ denotes a single vector of length $n$
  - $V_i$ denotes a group of $k$ vectors in outer iteration $i$, *e.g.* $V_i$=[$v_{ik+1}$, $v_{ik+2}$,....., $v_{ik+k}$]
- Groups of basis vectors
  - Given the following matrix comprising basis vectors
    $\underline{V_i}$=[$v_{ik+1}$, $v_{ik+2}$,....., $v_{ik+k}$, $v_{ik+k+1}$]
  - We use a variety of notations
    $V_i$=[$v_{ik+1}$, $v_{ik+2}$,....., $v_{ik+k}$]
    $\underline{V_i}$=[$V_i$, $v_{ik+k+1}$]
    $V_i'$=[$v_{ik+2}$, $v_{ik+2}$,....., $v_{ik+k}$]
    $\underline{V_i'}$=[$V_i'$, $v_{ik+k+1}$]

## Outline of the Algorithm

Communication-avoiding Lanczos is presented in Algorithm 5. Although the algorithm seems complex, from a higher level perspective, it performs the following basic operations

- takes the input matrix $A$ of size $n \times n$ and the initial vector $v$ of length $n$,
- generates $k$ vectors by performing $k$ SpMVs,
- orthogonalize with previous set of $k+1$ vectors (if any),
- orthogonalize these $k$ vectors with each other using QR factorization,
- take the last vector from the set of $k$ vectors and repeat the whole process again

In order to explain CA-Lanczos algorithm, we need to introduce the matrix $\underline{B_i}$ = [$e_2$, $e_3$, $\ldots$, $e_{k+1}$] where $e_j$ is a unit vector of length $n$ with a one at an index $j$. Additionally, we re-write the $R$ matrices (see details in [48]).

**Algorithm 5** CA-Lanczos [48]

```
Require: Symmetric matrix $A \in \mathbb{R}^{n \times n}$, initial Lanczos vector $\mathrm{v}_1 \in \mathbb{R}^{n \times 1}$ and number of iterations $t=\frac{r}{k}$, where $r$ is the number of iterations in Lanczos Iteration and $k$ is the step size
 1: $\beta_0 = \|v_1\|_2$, $q_1 = \frac{v_1}{\beta_0}$
 2: for $i = 0$ to $t$-1 do
 3:     Compute $\underline{V}_i^{'}$ by repeatedly multiplying $A$ with $q_{ik+1}$          −− Matrix Powers−−
 4:     if $i == 0$ then
 5:         Compute the QR factorization $\underline{V}_0$=$\underline{Q}_0\underline{R}_0$     −−QR Factorization−−
 6:         $\underline{T}_0 = \underline{R}_0\ \underline{B}_0\ \underline{R}_0^{-1}$, $\underline{\Gamma_0}$=$T_0$
 7:     else
 8:         $\underline{R}_{i-1}^{'}$=$\underline{Q}_{i-1}^{T}\underline{V}_i^{'}$          −−Block Gram Schmidt Orthogonalization 1−−
 9:         $\underline{V}_i^{'}$= $\underline{V}_i^{'}$ − $\underline{Q}_{i-1}\underline{R}_{i-1}^{'}$          −−Block Gram Schmidt Orthogonalization 2−−
10:         Compute the QR factorization $\underline{V}_i^{'}$=$\underline{Q}_i^{'}\underline{R}_i{'}$     −−QR Factorization−−
11:         Compute $T_i$ from Eq. (2.9) and $\beta_{k(i+1)+1}$ from Eq. (2.10).
12:         $$\Gamma_i = \begin{pmatrix} \Gamma_{i-1} & \beta_{ik+1}e_{k(i-1)}e_1^T \\ \beta_{ik+1}e_1e_{k(i-1)}^T & T_i \\ 0_{1,k(i-1)} & \beta_{k(i+1)+1}e_k^T \end{pmatrix}$$
13:     end if
14: end for
15: return Approximate tridiagonal matrix $\Gamma_i$
```

$$\underline{R}_{i-1} = \begin{pmatrix} e_{k+1} & \underline{R}_{i-1}^{'} \end{pmatrix}$$

$$\underline{R}_i = \begin{pmatrix} e_1 & \underline{R}_i^{'} \end{pmatrix}$$

where the last row of $\underline{R}_{i-1}$ and first row of $\underline{R}_i$ are the same. The $k$+1 × $k$+1 matrix $\underline{R}_i$ can be further decomposed into

$$\underline{R}_i = \begin{pmatrix} R_i & z_i \\ 0 & \rho_i \end{pmatrix}$$

The $k$+1 × $k$ basis matrix $\underline{B}_i$ can be decomposed as

$$\underline{B}_i = \begin{pmatrix} B_i \\ e_k^T \end{pmatrix}$$

We now get the $T_i$ as

$$T_i = R_iB_iR_i^{-1} + \overline{\rho}_i^{-1}z_ie_k^T - \beta_{ik+1}e_1e_k^TR_{i-1}R_i^{-1} \tag{2.9}$$

where $\overline{\rho}_i = R_i(k,k)$ and similarly $R_{i-1}$ is the $k \times k$ sub-matrix of $\underline{R}_{i-1}$. The expression for $\beta_{k(i+1)+1} = \underline{T}_i(k+1,k)$ is given by

$$\beta_{k(i+1)+1} = \frac{\rho_i}{\overline{\rho}_i} \tag{2.10}$$

## 2.10 Computational Kernels of CA-Lanczos

If we analyze Algorithm 5, there are three computational kernels, the matrix powers kernel (line 3), QR factorization of a tall-skinny matrix (line 5 and 10) and Block Gram Schmidt Orthogonalization (line 8 and 9). All these kernels perform computation in a communication-avoiding manner. For example, to perform $k$ SpMVs in the matrix powers kernel, we can fetch the matrix once and re-use it $k$ times for generating $k$ vectors. We now discuss these kernels of CA-Lanczos and identify different research questions that we might have to answer in order to minimize communication within the iterative numerical algorithms in general and the Lanczos Iteration in particular.

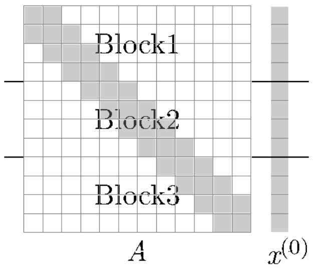
### 2.10.1 Matrix Powers Kernel

This kernel is a replacement for SpMV in standard Lanczos Iteration. It computes $k$ vectors in a single sweep and therefore it requires $\Theta(k)$ fewer messages in the parallel case and similarly $\Theta(k)$ fewer memory reads in the sequential architecture. In order to achieve this, it performs no more than a constant number of redundant floating-point operations. Given an $n{\times}n$ sparse matrix $A$, a general dense vector $x^{(0)}$ of length $n$, the matrix powers kernel is computed as

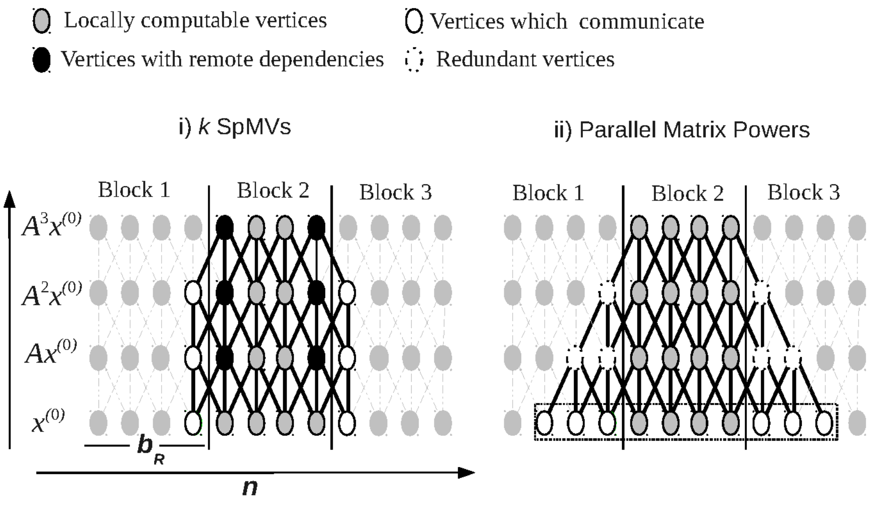
$$x^{(i+1)} \quad = \quad Ax^{(i)} \qquad 0 \le i \le k \tag{2.11}$$

In order to understand the communication patterns in this although simple looking kernel, we unroll the computation in Equation (2.11) as a graph. An example of such a graph is shown in Figure 2.6 for a tri-diagonal matrix $A$. Starting with a vector $x^i$ and matrix $A$, $k$ levels of the graph $G$ are generated by repeated matrix-vector multiplication as shown in Equation (2.11), *i.e* each new level corresponds to a new vector $x^{(i+1)}$. The dependency graph $G$ is defined as follows: associate a vertex with each element of the vector $x_j^{(i)}$ for $j = 0,\ 1, \ldots,\ n-1$ vector entries and $i = 0,\ 1, \ldots,\ k$ levels, and an edge from $x_m^{(i)}$ to $x_j^{(i+1)}$ if $A_{jm} \neq 0$. In other words, the edges correspond to the dependency of the $jth$ vector element $x_j^{(i+1)}$ on the elements of the previous vector $x^i$.
We assume that the graph $G$ can be partitioned into $N_q$ blocks. Each vertex $x_j^{(i)}$ has an *affinity* $q$ corresponding to the block where it is stored and all the vertices $x_j^{(0)}$ to $x_j^{(k)}$ have the same affinity depending only on $j$. If we consider any subset $S$ of vertices of $G$, then we denote $R(S)$ as the set of vertices that are reachable from any vertex in $S$ and $R(S,\ m)$ as the set of vertices on a path length at most $m$ from any vertex in $S$. We denote $R_q(S)$, $R^{(i)}(S)$ and $R_q^{(i)}(S)$ as the subsets of $R(S)$ with affinity $q$, with level $i$ and with affinity $q$ and level $i$ respectively. A similar notation is used for $R(S{,}m)$.
As the graph is partitioned in Figure 2.6(b)(i), each partition comprises three types of ver-

tices (1) vertices that can be computed locally (shown in gray) (2) vertices that have remote dependencies on other partitions (shown in black) and (3) vertices whose data need to be communicated (shown in white).
We illustrate the notation in Figure 2.6 with a simple example where we take a 12×12 tridiagonal matrix ($b$ =3) shown in Figure 2.6(a), divide it into three blocks and unroll the graph for three levels to generate vectors $x^{(1)}$, $x^{(2)}$ and $x^{(3)}$. If the set $V_2^{(3)}$ represents vertices in partition two at level three, then the dependency chain of all the vertices starting from level zero is also shown in Figure 2.6(b)(ii).



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



Locally computable vertices
Vertices which communicate
Vertices with remote dependencies
Redundant vertices
i) $k$ SpMVs
ii) Parallel Matrix Powers
Block 1
Block 2
Block 3
$A^3x^{(0)}$
$A^2x^{(0)}$
$Ax^{(0)}$
$x^{(0)}$
$k$
$b_R$
$n$

(b) $k$ SpMVs vs. Matrix Powers Kernel

Figure 2.6: $k$ 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.

Using the same notation, we first discuss naive matrix powers kernel using $k$ SpMVs and then describe the parallel and sequential algorithms for the matrix powers kernel.

### A. $k$ SpMVs

The matrix powers kernel can be computed naively using $k$ SpMVs. Consider graph $G$ shown in Figure 2.6(b)(i) with three partitions. Each partition contains two types of vertices, one with only local dependencies and the other which have remote dependencies. As the partitions reside on different processors in a parallel architecture, the remote dependencies need to be communicated at each level to generate a new vector. Hence, all the processors need to synchronize after each level and this makes this problem a communication-bound problem.

### B. Parallel Matrix Powers Kernel

A parallel matrix powers kernel trades extra computation for lower communication by generating $k$ vectors in a single sweep. The key idea is to eliminate the required synchronization in $k$ SpMVs and compute blocks of $k$ vectors on each processor without communicating with other processors. In order to compute the set $V_q^{(k)}$ for a block $q$ at level $k$, the dependency chain $R^{(0)}(V_q,\ k)$

is determined (see bounded box in Figure 2.6(b)(ii)) and the vertices in the set $R^{(0)}(V_q, k) - V_q^{(0)}$ which reside on other processors are communicated at level 0 (see white vertices in Figure 2.6(b)(ii)). We then use these vertices for redundant computation (see white vertices with dotted lines) to avoid communication at higher levels. This communication-avoiding approach is now well-known and the pseudo-code is shown in Algorithm 6 for a shared-memory parallel architecture.

**Algorithm 6** Parallel Matrix Powers Kernel (for proc. $p_q$) [65]

```
copy entries in $R^{(0)}(V_q, k) - V_q^{(0)}$ from shared memory
for $i = 1$ to $k$ do
    compute all $x_j^{(i)} \in R^{(i)}(V_q, k)$
end for
```

## C. Sequential Matrix Powers Kernel

For sequential architecture, the matrix $A$ is partitioned into blocks which are then loaded one after another into the fast memory. For architectures where the fast memory is managed explicitly, the parallel matrix powers kernel is used where each block is computed independently form the other in a communication-avoiding manner by performing redundant computation [65]. The blocks can be loaded in any order. On the other hand, for architectures with implicitly managed fast memory, there is no redundant compuation. However, since the the computation on the new block depends on the already computed vector components, one has to take care of the order in which blocks are loaded into the fast memory. An implicit sequential matrix powers kernel is shown in Algorithm 7.

**Algorithm 7** Implicit Sequential Matrix Powers Kernel [65]

```
$C = \emptyset$ ($C$ =set of computed entries)
for $q = 1$ to $N_q$ do
    for $i = 1$ to $k$ do
        compute all $x_j^{(i)} \in R^{(i)}(V_q, k) - C$
        $C \leftarrow R^{(i)}(V_q, k) \cup C$
    end for
end for
```

## D. Challenges

As the communication cost is highly dependent on the architecture, we need to choose or design architecture-aware algorithms for the matrix powers kernel. We will introduce different architectures in the next chapter. However, there are two main challenges associated with the matrix powers kernel on every architecture (1) how to minimize redudant computation to keep the computation cost as low as possible (2) how to select the value of $k$ which trades communication with computation. In Chapter 5, we address these challenges to minimize communication cost on GPU and FPGA while accelerating the matrix powers kernel.

### 2.10.2 QR Factorization

In each iteration of CA-Lanczos, we need to generate an orthogonal set of $k$ vectors for the Krylov subspace. QR factorization has been the de-facto technique to decompose an $m \times n$ matrix $A$ into an orthogonal component $Q$ of size $m \times m$ and an upper triangular matrix $R$ of size $m \times n$.

$$A = QR \tag{2.12}$$

When the matrix $A$ is tall and skinny with $m >> n$, then we perform economy-size QR factorization where $Q$ is an $m \times n$ matrix and $R$ is a square matrix of size $n$. Besides communication-avoiding iterative methods like CA-Lanczos, matrices with such aspect ratio arise in many different areas including block iterative methods, linear least squares problem and more importantly panel factorization of a general QR factorization. There are many different QR factorization methods including Cholesky QR, Gram Schmidt Orthogonalization, Modified Gram Schmidt Orthogonalization (MGS), Givens Rotations and Householder QR [39]. Often we require stable orthogonalization, *i.e.* the orthogonalization procedure should be independent of the input matrix entries. Out of all these methods, Householder QR is the most stable method along with MGS whose stability has been proved in the context of iterative numerical methods [42]. We first discuss the Householder QR approach, identify inherent problems for tall-skinny matrices and then review some approaches which have been recently proposed to achieve higher performance.

#### A. Householder QR

The Householder QR algorithm transforms the matrix $A$ of size $m \times n$ into an upper triangular matrix $R$ by applying successive transformations.

$$Q_n Q_{n-1} \dots Q_2 Q_1 A = R \tag{2.13}$$

The idea is to design $Q_i$ $(i = 1, 2, \dots,\ n)$ matrices in such a way that we zero out entries $i+1 : m$ of column $i$ of the matrix $A$ as shown below for a 4×3 matrix.

$$A = \begin{pmatrix} x & x & x \\ x & x & x \\ x & x & x \\ x & x & x \end{pmatrix} \rightarrow Q_1 A = \begin{pmatrix} x & x & x \\ 0 & x & x \\ 0 & x & x \\ 0 & x & x \end{pmatrix} \rightarrow Q_2 Q_1 A = \begin{pmatrix} x & x & x \\ 0 & x & x \\ 0 & 0 & x \\ 0 & 0 & x \end{pmatrix} \rightarrow Q_3 Q_2 Q_1 A = \begin{pmatrix} x & x & x \\ 0 & x & x \\ 0 & 0 & x \\ 0 & 0 & 0 \end{pmatrix}$$

The $Q_i$ matrix only affects rows $i : m$ and therefore does not change $i - 1$ rows and columns. We can write $Q_i$ matrix as

$$Q_i = \begin{pmatrix} I_{i-1} & O \\ O & H_i \end{pmatrix}$$

The $H_i$ matrix is called the *Householder reflector* matrix and is computed as $H = I - 2\frac{v_i v_i^T}{v_i^T v_i}$ where $v_i = -x_i \pm sign(x_i(1))||x_i||e_1$ for any column vector $x_i$. The algorithm is shown in Algorithm 8. The Householder QR is a communication-bound operation because of the dominant matrix-vector multiplication which is a Basic Linear Algebra Subroutine (BLAS) level 2 operation. Most existing QR routines for linear algebra libraries like LAPACK [7] use Householder QR

**Algorithm 8** Householder QR [39]

```
for $i = 1\ n$ do
    $x_i = A(i : m,\ i)$
    $v_i = x_i \pm sign(x_i(1))||x_i||_2 e_1$
    $\tau_i = \frac{-2}{v_i^T v_i}$
    $A(i : m,\ i : n) = A(i : m,\ i : n) + \tau_i v_i v_i^T A(i : m,\ i : n)$
end for
return The upper triangular part of $A$ containing the matrix $R \in \mathbb{R}^{n \times n}$ and matrix $V \in \mathbb{R}^{m \times n}$ where individual columns are indexed $v_i$ and a vector $t \in \mathbb{R}^{n \times 1}$ containing $\tau_i$ values.
```

for stability reasons but they employ its block variant to achieve higher performance. Blocked Householder QR comprises two stages, a *panel factorization* stage which is a BLAS2 operation and a *trailing matrix update* stage which involves BLAS3 matrix-matrix multiply as shown in Figure 2.7. For general matrices, the performance of the Blocked Householder QR is good on systems with a memory hierarchy due to dominant trailing matrix update with higher arithmetic intensity of the BLAS3 operations. However, for tall-skinny matrices, most or all of the time is spent on BLAS2 panel factorization. Hence, neither Householder QR nor its block variant is suitable for QR factorization of tall-skinny matrices such as those which arise in CA-Lanczos and other communication-avoiding iterative methods.

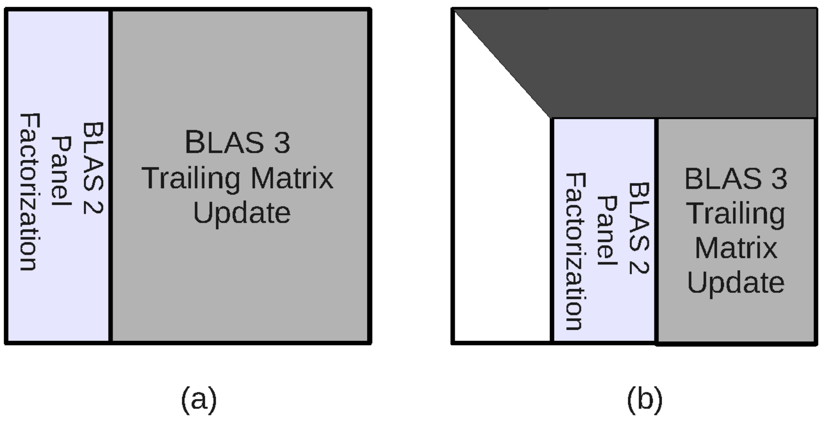

BLAS 2 Panel Factorization
BLAS 3 Trailing Matrix Update
(a)
BLAS 2 Panel Factorization
BLAS 3 Trailing Matrix Update
(b)

Figure 2.7: Computations within Blocked Householder QR.

## B. Tall-Skinny QR (TSQR)

The communication cost is the dominant factor in QR factorization of tall-skinny matrices. There are two reasons. First, Householder QR used in panel factorization is sequential in nature where each column is zeroed out one after another. Secondly, the algorithm is dominated by BLAS2 matrix-vector multiplication. As QR factorization of tall-skinny matrices arises in so many areas, a recent interest has been shown to devise new communication-avoiding algorithms for QR factorization of these matrices. Demmel *et al.* [33] propose the tall-skinny QR (TSQR) algorithm which divides the panel into smaller sub-matrices, factorizes these sub-matrices in parallel and then reduces the individual $R$ matrices using a tree structure. We illustrate the operations within parallel TSQR in Figure 2.8.

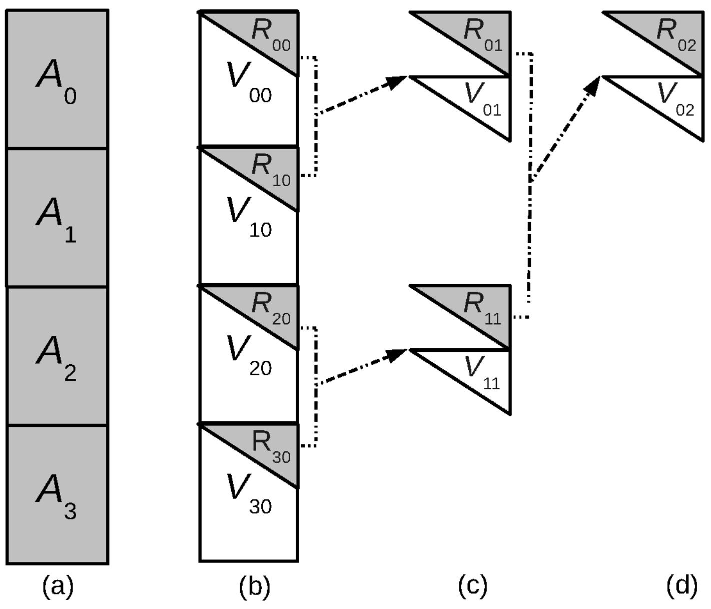


Figure 2.8: Tall-Skinny QR.

We take a toy example to explain the computations involved in parallel TSQR. Suppose we have a tall-skinny matrix $A$ which is divided into four sub-matrices $A_0$, $A_1$, $A_2$ and $A_3$. The first stage of TSQR is shown mathematically by the following equation.

$$A = \begin{bmatrix} A_0 \\ A_1 \\ A_2 \\ A_3 \end{bmatrix} = \begin{bmatrix} Q_{00}R_{00} \\ Q_{10}R_{10} \\ Q_{20}R_{20} \\ Q_{30}R_{30} \end{bmatrix} = \begin{bmatrix} Q_{00} & 0 & 0 & 0 \\ 0 & Q_{10} & 0 & 0 \\ 0 & 0 & Q_{20} & 0 \\ 0 & 0 & 0 & Q_{30} \end{bmatrix} \begin{bmatrix} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{bmatrix}$$

All sub-matrices are factorized in parallel (see Figure 2.8(b)) using Householder QR where $Q$ matrices are generated by $V$ matrices and $t$ vectors returned from Householder QR algorithm shown in Algorithm 8. The $R$ matrices are then stacked together in binary fashion to generate new sub-matrices which are then factorized to generate new $R$ sub-matrices (see Figure 2.8(c)).

$$A = \begin{bmatrix} Q_{00} & 0 & 0 & 0 \\ 0 & Q_{10} & 0 & 0 \\ 0 & 0 & Q_{20} & 0 \\ 0 & 0 & 0 & Q_{30} \end{bmatrix} \begin{bmatrix} Q_{01} & 0 \\ 0 & Q_{11} \end{bmatrix} \begin{bmatrix} Q_{02} \end{bmatrix} \begin{bmatrix} R_{02} \end{bmatrix}$$

The parallel TSQR algorithm exposes parallelism where the QR factorization of each sub-matrix can be mapped onto each processor within a parallel environment. The processors only exchange $R$ factors of the sub-matrices using the nearest neighbour communication as shown in Figure 2.8. It has been shown in [33], that such a parallel TSQR is optimal in terms of communication. For sequential architectures, the TSQR algorithm optimizes communication between fast and slow memories and it has been proved that such an algorithm is also communication-optimal.

#### C. Challenges

The TSQR algorithm addresses communication problem in Householder QR for both sequential and parallel architectures. However, there are some challenges associated with TSQR. Firstly, with each successive stage within TSQR, the workload decreases leading to under utilization of the compute resources. Secondly, the intermediate $R$ factors still need to be communicated which degrades performance due to memory or network latency. In Chapter 6, we address these challenges by exploiting the architectural features of the FPGA and demonstrate that TSQR on FPGA outperforms GPU due to tight algorithm-architecture interaction.

### 2.10.3 Block Gram Schmidt Orthogonalization

Gram-Schmidt Orthogonalization [39] 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 Algorithm 5, BGS is dominated by BLAS3 matrix-matrix multiplication and hence it is a compute bound operation. However, in Chapter 5, we identify that BGS involves matrix multiplication of short and fat matrix with a tall-skinny matrix and multiplication of these matrices have low arithmetic intensity compared to general square matrices. These aspect ratios lead to poor performance on architectures like GPU [56] where they are fetched from global memory. We demonstrate that we can achieve higher performance in BGS on FPGAs due to the on-chip storage of the input matrices.

## 2.11 Summary and Conclusion

We introduced the four main problems in applied numerical linear algebra. We presented various problem structures and the numerical methods used to solve each of these problems. We defined the scope of the thesis, *i.e.* to minimize communication in iterative numerical algorithms for solving structured sparse problems. We chose the Lanczos Iteration to solve the symmetric extremal eigenvalue as a case study. We identified the communication problem in these iterative methods and argued that this problem is going to stay for future computing architectures as well. We then presented a few algorithmic approaches, which aim at trading communication with computation to achieve higher performance.

We conclude that iterative numerical algorithms are communication-bound, and that they lead to poor silicon efficiency due to the ever increasing gap in communication and computation performance of modern computing platforms. Architecture-aware linear algebra algorithms including those for iterative numerical algorithms is a promising approach to minimize communication cost thereby increasing overall silicon efficiency. In future chapters of this thesis, we investigate how we can use the communication-avoiding approach to increase the silicon efficiency of FPGA and GPU in accelerating iterative numerical algorithms. We intend to achieve this in three different ways. Firstly, we design custom architecture using an FPGA that has an affinity for a particular algorithm. Secondly, we tune algorithmic parameters for both architectures including FPGA and GPU. Thirdly, we design architecture-aware algorithms.

# 3 Computing Platforms

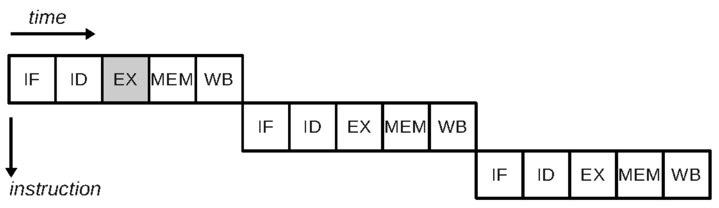
Historically, numerical computation is one of the benchmarks for evaluating the power of the computing platforms. The LINPACK benchmark suite [35] has been used over the years to measure floating-point power of the computing system. The core benchmark in this suite is LU factorization used for solving systems of linear equations $Ax = b$. Usually, highly optimized linear algebra libraries are written for numerical computation on these platforms, *e.g.* Linear Algebra Package (LAPACK) [7] is widely used on general purpose processors. In this chapter, we provide a brief introduction of different computing platforms that are evolved over the period, what is state of the art and how the future computing systems may evolve in order to achieve higher performance in numerical computation involving sparse iterative numerical algorithms. In doing so, our focus is not a distributed system comprising multiple nodes like a grid or a supercomputer, rather we limit ourselves with the computing power available at a single node. Additionally, we will go through different approaches to optimize communication while performing linear algebra operations on these systems.

## 3.1 Single-Core Microprocessor

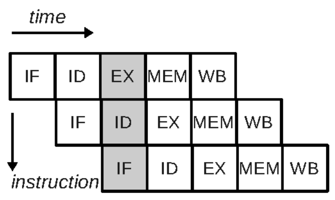
Since the invention of first silicon integrated circuit in 1959, computing technology has followed the Moore's Law, which states that the number of transistors in a given amount of silicon will double after every 18 months [67]. Most of the initial computers were based on the von Neumann architecture [68], which comprise a *processing unit* consisting of an arithmetic logic unit and registers, a *control unit* containing an instruction register and a program counter and a *memory* to store instructions and data. In the beginning, the focus was to design computer architecture which can process a set of sequential instructions as fast as possible. In this regard, new transistors were designed which can switch at higher clock frequencies and the abundant transistors, thanks to the Moore's Law, were arranged in a way to increase instruction throughput, *i.e.* number of instructions executed per clock cycle (IPC). We discuss a few techniques that are used to increase IPC.

### 3.1.1 Pipeline Parallelism

For an instruction to execute on a computer, it has to go through different stages like instruction fetch (IF), instruction decode (ID), execute (EX), memory read/write (MEM) and write back (WB). Figure 3.1(a) illustrates different stages of an instruction. If each stage takes one cycle, then five clock cycles are required to execute one instruction. In this case, one can achieve an IPC equal to $\frac{1}{5}$. To increase throughput, usually registers are introduced in between these stages to store intermediate results. Now the next instruction does not have to wait for the first instruction to complete, rather the new instruction can be fetched in the next clock cycle as shown in Figure 3.1(b). In this way, although the first instruction will still take five clock cycles (latency), we can achieve an IPC of 1 (throughput). The Intel x86 architecture family belongs to this class where we see aggressive pipelining to increase clock frequency. The pipelined architecture only supports in-order execution, and if there is any data dependency between successive instructions, the pipeline needs to be *stalled* leading to a decrease in IPC. See [46] for more details on pipeline hazards.



(a) Unpipelined Processor.



(b) Five-Stage Pipelined Processor.

Figure 3.1: Pipeline Parallelism.

### 3.1.2 Out of order execution

In order to overcome the data dependency in an in-order pipelined architecture, a natural way to improve IPC is to incorporate out-of-order execution of independent instructions. In 1990, IBM introduced first out-of-order microprocessor, the Power1, although out of order was limited to floating-point instructions only. Since then, all modern computers perform out-of-order execution including Intel Nehalem, Sandy Bridge and Ivy Bridge architectures. With more and more transistors, new architectures employ more than one pipeline and hence the IPC in these architectures can be greater than one.

LAPACK [7], an optimized linear algebra library, was also introduced during the same time period. The library contains routines for performing BLAS operations like vector-vector operations, matrix-vector multiply and matrix-matrix multiplications. Additionally, it contains higher level driver routines for performing factorizations like LU, Cholesky, QR, *etc.*

### 3.1.3 Data-level Parallelism

Besides instruction level parallelism and out of order execution, another potential opportunity to exploit parallelism is to perform operation on multiple data. As per Flynn's taxonomy [38], such an approach is called Single Instruction Multiple Data (SIMD) architecture. This approach has been used in 1970s on vector supercomputers. Intel used it for the first time in 1997 for a P5-based Pentium line of single-core microprocessors. Although there may be no difference in IPC compared to pipelined out-of-order execution, there is significant increase in performance in terms of floating-point operations per second. There are special instructions (SSE) to invoke SIMD hardware and are particularly useful to operate on vector data such as pixels of an image. Both Intel and AMD provide optimized linear algebra libraries like Intel Math Kernel Library (Intel MKL) [51] and AMD Core Math Library (ACML) [6], which take advantage of these SIMD instructions.

### 3.1.4 Thread-level Parallelism

Thread-level parallelism is a software approach which is traditionally used by operating systems to service multiple users at a time. The idea is to use threads to distribute the processor time among different users. This involves the costly operation of context switching which involves saving users register data in memory and then resuming when the user thread takes control of the processor. However, with the advancement in hardware technology and the thrust to get higher performance, computer architects built systems which have physical hardware support for multiple threads within a single-core. Intel Hyper-Threading [64] is one such example.

### 3.1.5 Communication

The communication in a single-core computer involves data movement between the main memory and the processor. The main memory access time (including both latency and inverse bandwidth) is much higher compared to actually performing the floating-point operations [41]. It is, therefore, desired to have high arithmetic intensity, *i.e.* more floating-point operations need to be performed than the words fetched from main memory. Computer architects introduce cache to exploit this arithmetic intensity. A cache is a small memory buffer located on-chip to minimize access latency and increase bandwidth if the same data has to be fetched again and again. The cache is particularly useful when there is temporal locality, *i.e.* the data from the neighbourhood is also fetched from the main memory into the cache since there is a high probability that it will be accessed in future. As computer architecture evolved, multi-level cache hierarchy has been introduced where besides an on-chip L1 cache, there is a relatively large off-chip L2 cache which serves as a bandwidth amplifier as shown in Figure 3.2(b).

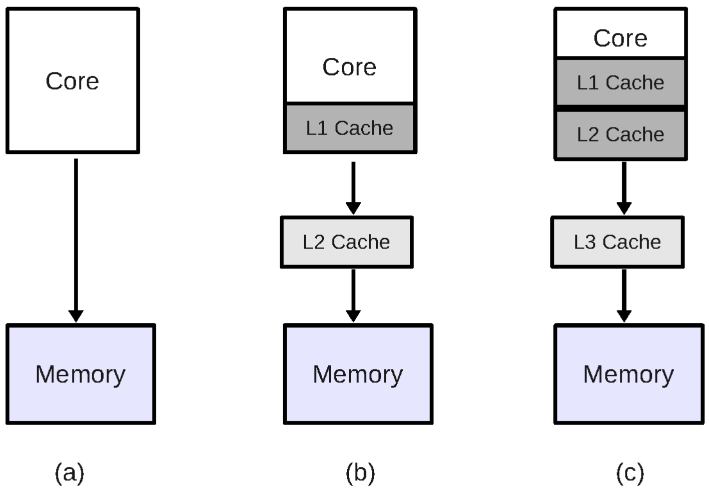


Figure 3.2: Memory Hierarchy in Single-Core Microprocessors (a) Un-Cached (b) L1 (on-chip) and L2 (off-chip) Cache (c) L1 and L2 (on-chip), L3 (off-chip) Cache

Some architectures have both L1 and L2 caches located on-chip with an off-chip L3 cache as shown in Figure 3.2(c).

As cache has lower latency and higher bandwidth compared to the main memory, the linear algebra libraries use explicit and implicit cache blocking to exploit data re-use in certain computations, *e.g.* the matrix-matrix multiplication. A naive algorithm for matrix-matrix multiplication is described in Algorithm 9 to compute $C = C + A \times B$ and this is illustrated in Figure 3.3(a). In this case, the total memory accesses can be calculated as

$$M = n^3(\text{for } B) + n^2(\text{for } A) + 2n^2(\text{for } C) \tag{3.1}$$

With $2n^3$ floating-point operations, for large $n$, the arithmetic intensity is 2. This is exactly the same arithmetic intensity in communication-bound matrix-vector multiplication. However, if we use blocking in the matrix-matrix multiplication, we can increase the arithmetic intensity. We divide the matrix into $N^2$ blocks, perform the multiplication on small blocks as shown in Figure 3.3(b). We describe the algorithm in Algorithm 10 and calculate the total memory accesses as

$$\begin{aligned} M = N \times n^2(\text{read each block of } B\ N^3 \text{ times}) + N \times n^2(\text{read each block of } A\ N^3 \text{ times}) \\ +2n^2(\text{read each block of } C \text{ once}) \end{aligned} \tag{3.2}$$

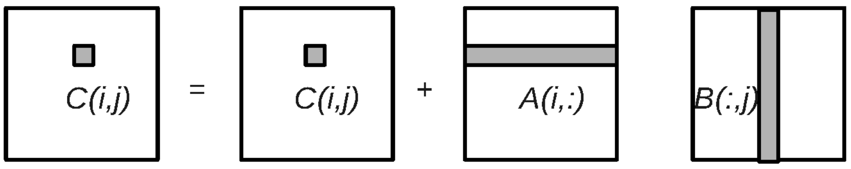
The arithmetic intensity in this blocked variant is $\frac{n}{N}$, which is equal to the block size. We can achieve higher performance if we increase the block size. In all linear algebra operations, blocking is used wherever possible to achieve performance. In fact, algorithms are usually modified to enable blocking like the Blocked Householder QR [83]. However, for operations like SpMV or even dense matrix-vector multiply, arithmetic intensity is low ($O(n^2)$ memory operations, $O(n^2)$ FLOPs) and therefore cache blocking is not a viable option for minimizing communication with the main memory.

**Algorithm 9** A naive matrix-matrix multiplication.

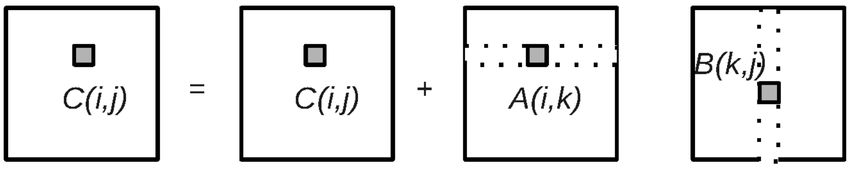
```
Require: Input square matrices $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times n}$.
 1: for $i = 1$ to $n$ do
 2:     Read row $i$ of $A$ into cache.
 3:     for $j = 1$ to $n$ do
 4:         Read $C(i,\ j)$ into cache.
 5:         Read col $j$ of $B$ into cache.
 6:         for $k = 1$ to $n$ do
 7:             $C(i,j) = C(i,\ j) + A(i,\ k) * B(k,\ j)$.
 8:         end for
 9:         Write $C(i,\ j)$ back to main memory.
10:     end for
11: end for
12: return Output matrix $C \in \mathbb{R}^{n\times n}$.
```

**Algorithm 10** A blocked matrix-matrix multiplication.

```
Require: Input square matrices $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times n}$, number of blocks $N$.
 1: for $i = 1$ to $N$ do
 2:     for $j = 1$ to $N$ do
 3:         Read block $C(i,\ j)$ into cache.
 4:         for $k = 1$ to $n$ do
 5:             Read block $A(i,\ k)$ into cache.
 6:             Read block $B(k,\ j)$ into cache.
 7:             $C(i,j) = C(i,\ j) + A(i,\ k) * B(k,\ j)$.
                matrix-multiply on blocks
 8:         end for
 9:         Write $C(i,\ j)$ block into main memory.
10:     end for
11: end for
12: return Output matrix $C \in \mathbb{R}^{n\times n}$.
```



(a) A naive matrix-matrix multiplication.



(b) A blocked matrix-matrix multiplication.

Figure 3.3: Linear algebra with cache blocking.

## 3.2 Multi-Core

The performance of single-core architectures is pushed to the limits by using instruction-level parallelism, data-level parallelism and thread-level parallelism together with drastic scaling of the clock frequency. In 2004, we hit the *power wall*, *i.e.* the clock frequency can not be scaled further due to huge power dissipation. This has changed the whole architecture design methodology leading to the era of *multi-cores*. The idea is to keep the frequency constant while increase the number of cores within a single die in order to achieve an overall high peak performance. Both Intel and AMD presented their dual-core, quad-core and up to deca-core (10 cores) architectures. Although, the raw computing power has increased, there is even more challenge with the communication between the cores as well as within the memory hierarchy for a single core. Figure 3.4 shows an Intel Nehalem 8-core architecture.

We see a number of design considerations to minimize communication cost. First of all, there is a three-level cache hierarchy to minimize communication with the off-chip main memory. All these caches are located on the die. In order to minimize access latency, an integrated memory controller (IMC) is located on the die and multiple banks are used to enhance bandwidth. The cores within the same processor communicate through an L3 shared cache/memory. Additionally, to minimize communication cost between cores of different processor on the same die, a dedicated point-to-point interconnect (QPI) is provided for non-uniform memory accesses (NUMA).

As multi-core technology changes the architectural landscape, the linear algebra libraries were re-designed to exploit this parallelism. Parallel Linear Algebra for Scalable Multi-core Architectures (PLASMA) [3] is one such library that contains routines for BLAS functions as well as high level routines like LU, QR and Cholesky factorization. PLASMA relies on tile algorithms [24] which provide fine-grain parallelism. The linear algebra algorithms can then be represented by Directed Acyclic Graph (DAG) [43] where nodes represent tasks performed on tiles and edges correspond to the dependencies between these tasks. A programming model is then used which uses asynchronous, out-of-order scheduling of operations and this allows scalability. A DAG having nodes with large number of incoming edges needs lot of inter-core communication,

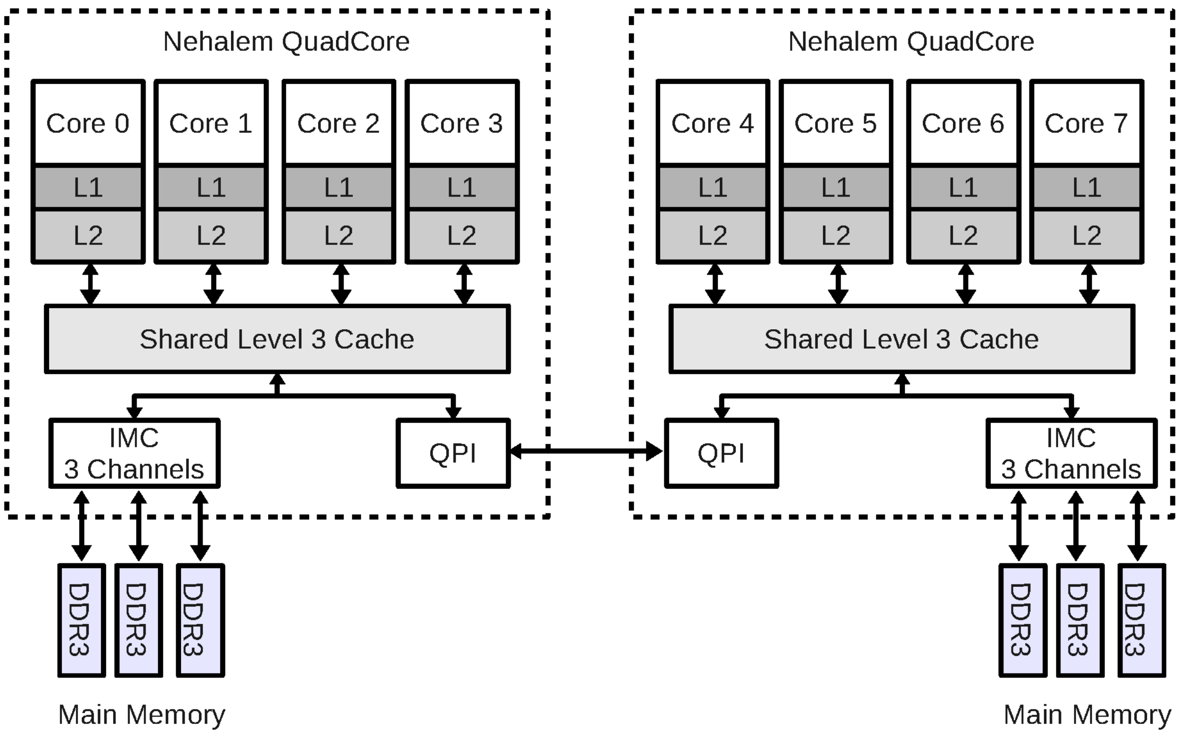


Figure 3.4: Intel Nehalem Eight-Core Architecture [66]. IMC and QPI stands for integrated memory controller and Quick Processor Interconnect respectively.

which degrades performance as communication cost is higher than the computation itself. An example of such a DAG is the graph for $k$ SpMVs in iterative numerical algorithms as shown in Figure 2.6(b)(i).

## 3.3 Many-Core

With the increase in silicon capacity, modern architectures tend to be many-core architectures like the Graphical Processing Unit (GPU) from Nvidia and AMD, the Cell Processor [54] from IBM and a more recent Xeon Phi [26] from Intel. We discuss GPU and Intel Phi architecture here as the Cell Processor is not widely used these days.

### 3.3.1 Graphical Processing Unit

The Graphical Processing Unit (GPU) has been traditionally used to accelerate graphics in computer games since 1999. The compute intensive transform and lighting calculations, triangle setup and polygon rendering are offloaded to a GPU which then processes them at higher speed than the general purpose processor. Recently, Nvidia came up with General Purpose GPU (GPGPU) with single-precision and double-precision floating-point support, which can be used for scientific computation, an area dominated by general purpose processor and supercomputers. The GPU is based on Single Instruction Multiple Thread (SIMT) paradigm, where all threads run the same code but operate on theirs own data segment. GPU is used as a hardware accelerater having a tight coupling with the host general purpose processor. While the general purpose processor supports a few threads and is suitable for control dominated tasks, GPUs on the other hand can spawn thousands of threads and they target data-parallel tasks. The architecture of a modern Nvidia C2050 GPU, widely used for scientific computing, is shown in Figure 3.5.

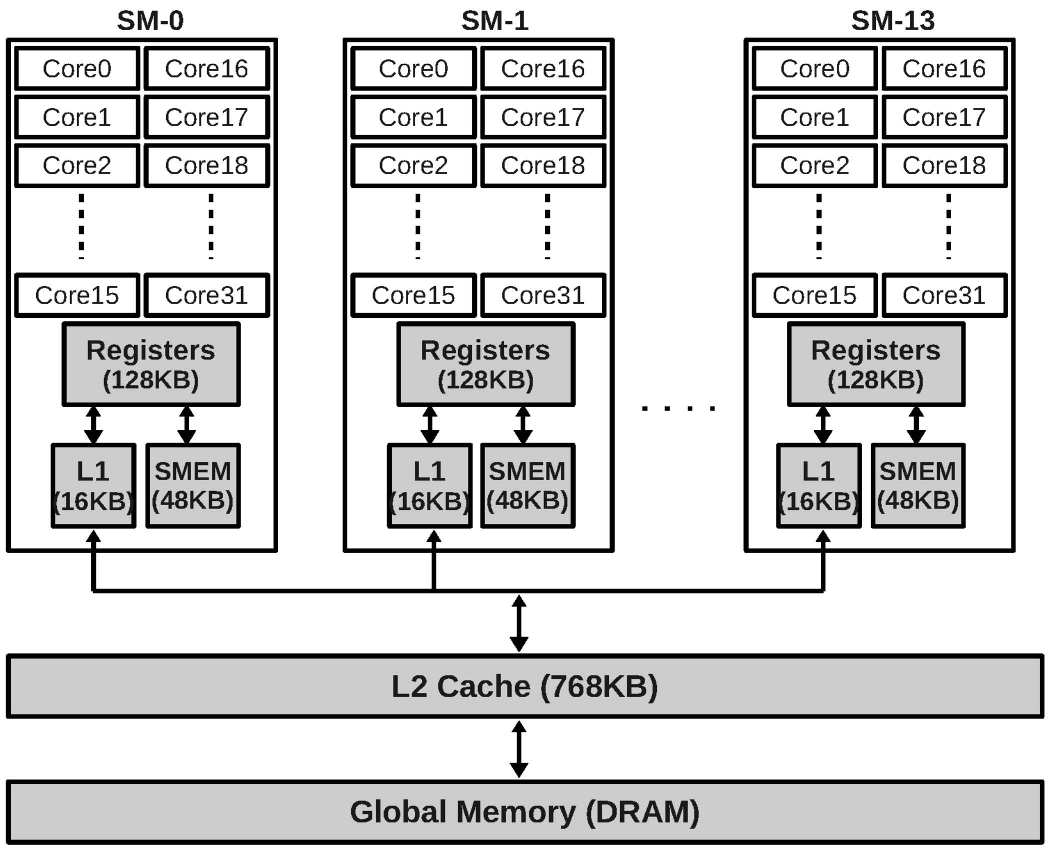


Figure 3.5: Nvidia C2050 Fermi Architecture [71]. SM stands for streaming multiprocessor.

The GPU comprises 14 streaming multiprocessors (SMs). Each SM has 32 floating-point cores running at 1.15 GHz capable of executing a single-precision FLOP per cycle. In total, the GPU is capable of 1.03 single-precision TFLOPs and 515 double-precision GFLOPs.

Tasks on a GPU (independent of any particular device) are scheduled in a group of threads called *thread block*. Each thread block can contain up to 1024 threads (depending on CUDA compute capability) which are scheduled as a group of 32 threads called *warp*. These threads share data with each other using an on-chip shared memory. Each thread block is an independent entity and different thread blocks can run in any order and on any SM. The threads within a thread block need to synchronize for sharing data with each other. If threads from different thread blocks want to share data with each other, they have to use the L2 cache or global device memory. There is a large register file in each SM which is distributed among the cores. GPU has an inverse memory hierarchy [96], *i.e.* the size of the register file is large compared to L1 and L2 caches. This memory can be used for blocking purposes. The pipeline inside the floating-point core can be used for instruction level parallelism (ILP). As all cores operate in a SIMT fashion, we can exploit data-level parallelism. Additionally, we have warp-level parallelism, where warps from multiple thread blocks can be executed concurrently. The warp-level parallelism is particularly useful to hide memory latency where a different warp is scheduled if the current warp has requested a memroy transaction.

GPUs are suitable for dense linear algebra operations having higher arithmetic intensity, *e.g.* the matrix-matrix multiplication. There are number of dense linear algbera libraries for GPU including the CuBLAS [72] and MAGMA [91]. It has been shown that the performance of dense linear algebra approaches the peak hardware capabilities of GPU [97]. For communication-bound dense operations like the matrix-vector multiplication which has low arithmetic intensity (2 FLOPs per word), the performance is bounded from above by the off-chip global memory bandwidth. For example, on Nvidia C2050 with 144 GB/s memory bandwidth, the maximum attainable performance is 72 single-precision GFLOPs. This is less than 7% of peak single-precision floating-point performance. For sparse matrix-vector multiplication (SpMV) using commonly used CSR format, where in addition to the data element we need to access the column index as well, the overall efficiency is even lower than 7%. A number of optimized sparse libraries are also available including CUSP [17] and CuSPARSE [73].

### 3.3.2 Intel Xeon Phi

The Intel Xeon Phi Coprocessor [26] has recently been introduced as a hardware accelerator in direct competition with GPUs. A device from this family, KNC Card SE10P, is shown in Figure 3.6.

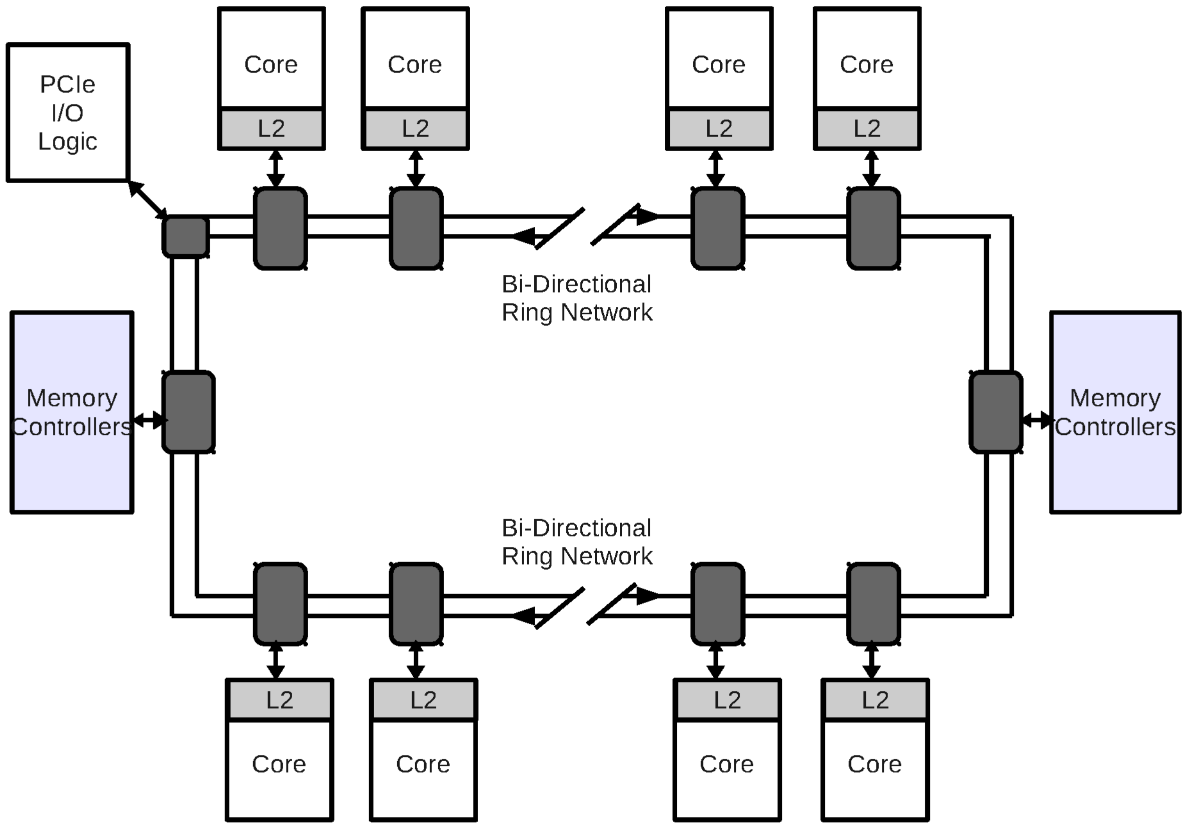


Figure 3.6: Intel Xeon Phi Coprocessor KNC Card SE10P.

It contains eight memory controllers to provide a memory bandwidth of 352 GB/s. There are 61 cores clocked at 1.05 GHz. All cores and memory controllers are connected using a ring network with a peak bandwidth of 220 GB/s. Each core has 32 KB L1 cache and a 512 KB L2 cache. There are four hardware contexts for a core at any one time. At each clock cycle, the instructions from a single thread runs on the core. No more than two instructions from the same context can run consecutively since the goal is to hide latency by launching threads from multiple contexts. The bulk of the performance comes from the vector processing unit within each core. There are 32×512-bit SIMD registers within each core, which can be used for either double-precision or single-precision operations. The vector processing unit performs basic functions like addition, multiplication, sine and sqrt allowing 16 single-precision operations. The unit also supports Fused Multiply-Add (FMA), which is counted as two operations for benchmarking purposes. Thus a total of 2.0496 single-precision TFLOPs and 1.0248 double-precision TFLOPs can be achived on SE10P card.

The main focus of this Intel Xeon Phi architecture is to accelerate communication-bound operations. It has recently been used for accelerating dense as well as sparse linear algebra operations. Due to its high memory bandwdith (352 GB/s) compared to contemporary GPUs like that of K20 (208 GB/s), the Intel Xeon Phi architecture can attain higher performance for the communication-bound operations, *e.g.* the matrix-vector multiplication.

## 3.4 Custom Computing–Field Programmable Gate Arrays

Custom computing is another architecture design approach where the hardware is specialized for a particular algorithm with a reconfiguration support to change all or some part of the hardware statically or dynamically. Although there are different devices which support custom computing, Field Programmable Gate Arrays are the most widely used technology. An FPGA is a two-dimensional array of simple hardware units, such as configurable logic blocks (CLBs) containing lookup tables (LUTs) to implement any boolean function, larger random-access memories (RAMs), some arithmetic units (DSP blocks) and a flexible routing structure as shown in Figure 3.7.

The content of each of the CLBs together with the information defining the topology of the interconnect between the computational units can be programmed to generate an arbitrary circuit. The potential to use FPGAs in high-performance computing arises from the fact that computer architecture can be specialized to accelerate a particular task. As an example, let us say that we have to implement the data-flow graph shown on the left side in Figure 3.8 on a CPU. All the variables are stored in memory and ALU needs to be shared in a temporal fashion to perform add and divide operations. The intermediate results are stored back in memory. On an FPGA, we can implement these operations as pipelined spatial circuits while implementing the dependencies between the operations physically using pipelined wires to get high performance. This allows the FPGA mapping to start a new evaluation in each cycle delivering higher throughput than the CPU. With FPGAs, we can exploit different types of parallelism including pipelining using deeply pipelined floating-point operators, spatial parallelism and data-level par-

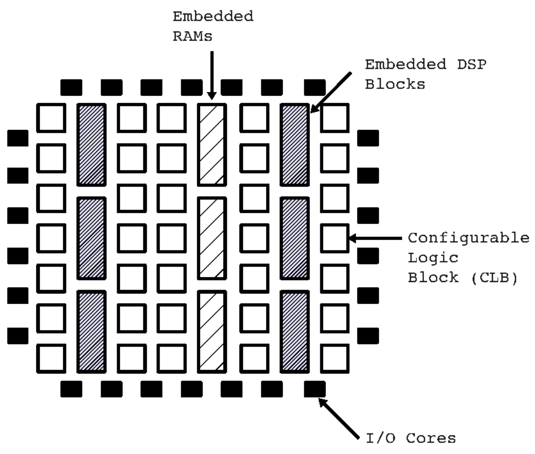


Figure 3.7: Partial FPGA schematic illustrating its architecture containing CLBs as small programmable ROMs, embedded RAMs in the region of 18k bit, DSP blocks as dedicated multiply/add circuitry. The routing fabric is not shown.

allelism. Additionally, we can choose arbitrary precision as compared to multi-cores and GPUs where only single-precision or double-precision floating-point numbers are allowed. However, in this work, we compare FPGAs with contemporary architectures for accelerating iterative numerical algorithms using either single-precision or double-precision number representation.

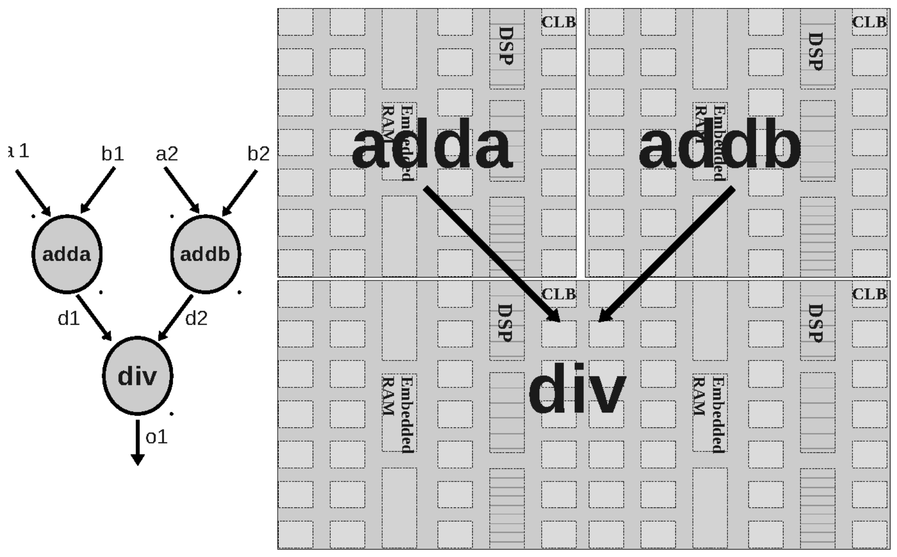


Figure 3.8: Spatial Computation using FPGAs.

## 3.5 Architectural Comparison

In this chapter, we have introduced various architectures for high performance computing. As we intend to increase the silicon efficiency while accelerating communication-bound sparse iterative numerical algorithms, we compare these architectures against some core parameters. These parameters are the peak floating-piont performance, on-chip memory capacity, on-chip memory bandwidth and off-chip memory bandwdith. We show these parameters in Table 3.1 for three different architectures including the Nvidia C2050 GPU, Virtex6-SX475T and an Intel Xeon Phi XNC SE10P.

Table 3.1: Architectural Features of FPGA and GPU (FPGA clock frequency is not reported in [87]).

| **Device** | **Tech.** | **Peak GFLOPs** | **Memory (On-Chip)** | | **Memory BW (On-Chip)** | | **Memory BW (Off-Chip)** |
|---|---|---|---|---|---|---|---|
| | **(nm)** | (single-precision) | Total. RAM | Registers | RAM | Registers | |
| **Virtex6 (SX475T)** | 40 | 450 [87] | 4 MB | 74 KB | 5.4 TB/s | 36 TB/s | 34 GB/s [87] |
| **Nvidia (C2050)** | 40 | 1030 | 672 KB Aggregate | 1.7 MB Aggregate | 1.3 TB/s [96] | 8 TB/s [96] | 144 GB/s |
| **Intel Xeon Phi KNC SE109** | 22 | 2049 | 31 MB (L2) Aggregate | 1.9 MB (L1) Aggregate | | 1.23 TB/s (L1) [37] | 352 GB/s |

From Table 3.1, we analyze that Intel Xeon Phi has relatively large peak floating-point performance, on-chip memory capacity and off-chip memory bandwidth compared to the other two devices. It is because of its high off-chip bandwidth, that this architecture will perform better for communication-bound operations like the sparse iterative numerical algorithms. However, until now, there is no benchmark studies on this architecture for such algorithms. However, an upper bound can be computed based on the dominant SpMV kernel where for each word that is fetched we need to perform two FLOPs. This results into 176 GFLOPs at an off-chip memory bandwidth of 352 GB/s. The silicon efficiency is 8.5%, slightly higher than the GPU device where it is 7% and on FPGA where it is close to 4%. We do not consider this architecture in our study because it has recently been introduced and we target more on improving the silicon efficiency of FPGA and GPU. As we have seen in case of Intel Xeon Phi architecture that by just changing the architectural parameters, although the floating-point performance of communication-bound operations may increase, there might be no effect on silicon efficiency. In future chapters, therefore, we discuss techniques how to improve silicon efficiency by a tight interaction between algorithm and architecture. However, as the architecture resembles GPU because of its SIMT compute organization, the results and techniques to improve silicon efficiency of GPU can well be extended to the Intel Xeon Phi architecture. Although FPGAs have the lowest off-chip memory bandwdith, they have the highest on-chip memory bandwidth. We intend to exploit this architecture feature to saturate the floating-point units thereby increasing the silicon efficiency. In order to minimize communication on FPGAs, we will see in subsequent chapters how we use different techniques including explicit cache blocking and more architecture-aware algorithms. We emphasize that the ultimate goal is to increase silicon efficiency, *i.e.* even if both architectures offer same sustained performance (GFLOPs), we are interested in picking the one that has better silicon efficiency (the ratio of sustained performance to peak performance).

# 4 Avoiding Communication in Iterative Numerical Algorithms for Small-to-Medium Size Problems

In this chapter we focus on accelerating iterative numerical algorithms for dense problems involving small-to-medium size data sets (matrix size $n\sim$ 10s to a few 100s). As mentioned in Chapter 2, we choose the symmetric extremal eigenvalue problem as a case study. We provide motivation to accelerate the solution with real examples where we need to solve multiple of such problems independently. We argue that previous approaches [4] [60] [23] have limitations in solving this problem for two reasons. First they deal with only very small matrices ($n\sim$20). Secondly they use direct methods which are good when one is interested in all eigenvalues and eigenvectors but are inefficient when only a few eigenvalues are required, especially if only the extremal (maximum or minimum) eigenvalue is desired. We show how the Lanczos Iteration [34] can be specialized for problems where only the extremal eigenvalue is desired. We highlight the key features of FPGAs which help us in avoiding communication in the Lanczos Iteration for such problem sizes and also provide us the opportunities to solve multiple problems simultaneously. We present an IEEE 754 single precision floating-point implementation of the Lanczos Iteration on an FPGA. The proposed approach is scalable and particularly suitable for small-to-medium sized problems. We show that by avoiding communication and exploiting parallelism of multiple problems, we can increase the efficiency of FPGA from a mere 1.8% to 38% and that this efficiency is limited by on-chip memory resources. We compare our results with other parallel architectures like multi-core and GPU which have 3.3% and 0.13% efficiency respectively. As the Lanczos Iteration is at the heart of all modern iterative numerical algorithms, our results are applicable to a broad class of algorithms which are not only used for solving eigenvalue problem but also solving linear system of equations. The main contributions in this chapter are therefore:

- A specialized iterative framework comprising the Lanczos Iteration for computing only the extremal eigenvalues of a dense symmetric matrix with a computational complexity of $O(rn^2)$ where $r$ is the number of iterations and $r \ll n$.
- A mechanism to avoid communication in the Lanczos Iteration and an architecture modified from [61] for accelerating multiple small-to-medium size symmetric extremal eigenvalue problems.
- An FPGA implementation of the proposed architecture capable of a sustained performance of 175 GFLOPs on a 260 MHz Virtex6-SX475T for a maximum matrix of size 335 $\times$ 335.

- A quantitative comparison with an Intel Xeon X5650 and an Nvidia C2050 GPU showing a speed up of 8.2-27.3× (13.4× geo. mean) and 26.2-116× (52.8× geo. mean) respectively when FPGA is solving a single eigenvalue problem whereas a speed up of 41-520× (103× geo.mean) and 131-2220× (408× geo.mean) when it is solving multiple eigenvalue problems.

## 4.1 Applications Involving Multiple Symmetric Extremal Eigenvalue Problems

In Chapter 2, we introduce applications where we need to solve the symmetric eigenvalue problem. In this section we show that often we need to solve multiple of such problems in these applications.

### 4.1.1 Multiple Minimum Eigenvalue Problems in Interior-Point Method for Semi-definite Programming

In each iteration of the primal-dual interior-point method to solve semi-definite programs [5], starting with some initial values of optimization variables $X$ and $Z$, we compute search directions $\Delta X$ and $\Delta Z$. In order to update $X = X + \alpha_p \Delta X$ and $Z = Z + \alpha_d \Delta Z$, we estimate $\alpha_p$ and $\alpha_d$ by solving the following *line search* problem for the newly computed search directions.

$$\alpha_p = \max\{\alpha \in [0,1] : X + \alpha \Delta X \succeq 0\} \tag{4.1}$$

$$\alpha_d = \max\{\alpha \in [0,1] : Z + \alpha \Delta Z \succeq 0\} \tag{4.2}$$

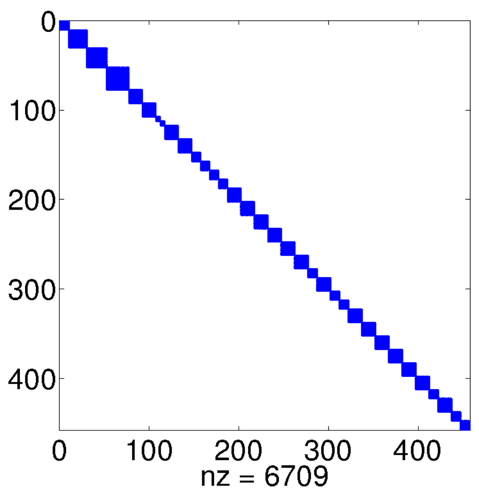
Referrring to Equation (4.1) and (4.2), in order to check $X + \alpha \Delta X \succeq 0$ and $Z + \alpha \Delta Z \succeq 0$ we need to compute the minimum eigenvalue of the matrices $X + \alpha \Delta X$ and $Z + \alpha \Delta Z$.

Traditionally, Equation (4.1) and (4.2) are computed using backtracking where one can select initial values of $\alpha_p$ and $\alpha_d$ close to 1 and keep decreasing the values until the minimum eigenvalue becomes greater than equal to zero. This process can be parallelized by using a set of values of $\alpha_p$ and $\alpha_d$ and solving multiple minimum eigenvalue problems. We can then finally pick the value from the set which corresponds to the minimum eigenvalue greater than equal to zero.

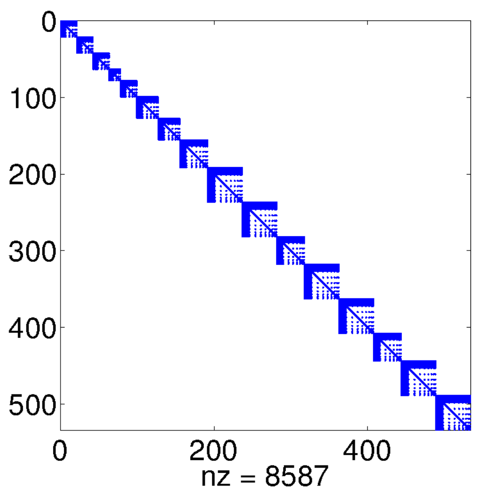
Another level of parallelism comes from the structure of the matrices $X$ and $Z$. For block-diagonal matrices that arise in polynomial optimization [98] and are shown in Figure 4.1, the minimum eigenvalue can be calculated by computing the minimum eigenvalue of each diagonal block independently and then finding out the minima among those eigenvalues.

### 4.1.2 Multiple Maximum Eigenvalue Problems in Eigenvalue Based Sensing

Channel sensing, *i.e.* detecting the presence of a primary user, is one of the fundamental tasks in cognitive radio. In IEEE 802.22, multiple channels used by the primary users are sensed simultaneously to look for inactivity. The idea is to provide these inactive channels to the secondary users for communication. A covariance based approach is presented in [104] where a covariance matrix is formed from a few samples of the received signals. The maximum eigenvalue



(a) block diagonal matrix with variable block size



(b) block diagonal matrix with variable block size and sparsity within the block

Figure 4.1: Sparsity Pattern of $X$, $Z$ [98].

of the covariance matrix is used as the test statistic. Thus the problem of multiple channel sensing can be cast as a multiple maximum eigenvalue problem, where we have to compute the maximum eigenvalue of the received signal covariance matrix from different channels.

## 4.2 Iterative Framework

In Chapter 2, we introduce the Lanczos Iteration, an iterative numerical algorithm, as the most appropriate method for solving symmetric extremal eigenvalue problem. We tailor a 2-stage iterative framework, which comprises the Lanczos Iteration for generating the tridiagonal matrix followed by the bisection method [39] to pick only the extremal eigenvalue.

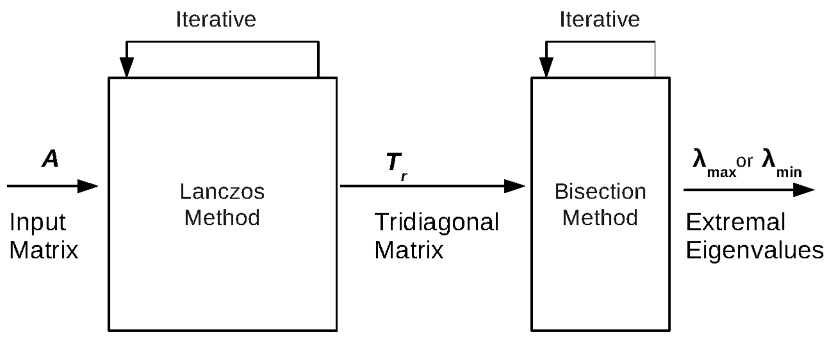


Figure 4.2: Symmetric Extremal Eigenvalue Computation.

### 4.2.1 Specializing the Lanczos Iteration

Given an $n \times n$ symmetric matrix $A$, the Lanczos Iteration applies orthogonal transformations to reduce it to a tridiagonal matrix $T_r$ in an iterative manner

$$Q_r^T A Q_r = T_r \tag{4.3}$$

where $r$ is the iteration count and $Q_r \in \mathbb{R}^{n \times r}$, $T_r \in \mathbb{R}^{r \times r}$. The eigenvalues of $T_r$ are approximations to the eigenvalues of $A$. The extremal eigenvalues start converging first after a few iterations as shown in Figure 4.3. The Lanczos Iteration is shown in Algorithm 11.

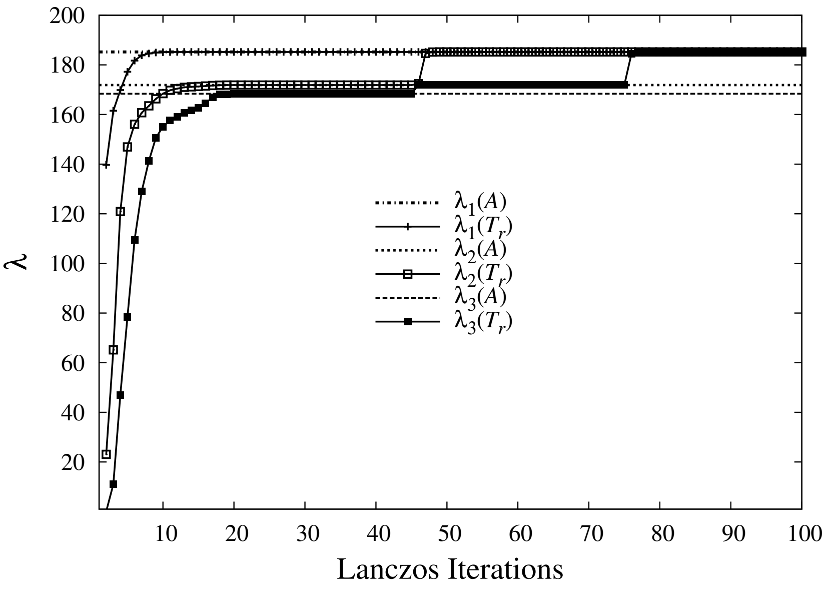


Figure 4.3: The Lanczos Iteration convergence to the maximum eigenvalue of $A$.

**Algorithm 11** Lanczos Iteration in Exact Arithmetic [39]

```
Require: Symmetric matrix $A \in \mathbb{R}^{n \times n}$, initial orthonormal Lanczos vector $q_0 \in \mathbb{R}^{n \times 1}$ and number of iterations $r$, $\beta_0 = 0$.
for $i = 1$ to $r$ do
    $\overline{q_i} := Aq_{i-1}$              $(lz1)$
    $c_i := \overline{q_i} - \beta_{i-1}q_{i-2}$   $(lz2)$
    $\alpha_i := \overline{q_i}^T q_{i-1}$         $(lz3)$
    $d_i := c_i - \alpha_i q_{i-1}$        $(lz4)$
    $b := d_i^T d_i$                $(lz5)$
    $\beta_i := \sqrt{b}$                $(lz6)$
    $f := 1/\beta_i$                $(lz7)$
    $q_i := f d_i$                 $(lz8)$
end for
return Tridiagonal matrix $T_r$ containing $\beta_i$ and $\alpha_i$ $i = 1, 2, \ldots, r$
```

**Algorithm 12** Bisection Method for $\lambda_{max}$ [39]

```
Require: $\alpha_i$ and $\beta_i$ for $i$ = 1,2, .... $r$
  a:= 0, b:= 0, $eps$ := 5.96x10$^{-8}$
  for $i = 1$ to $r$ do
      a := max(a , $\alpha_i - ( | \beta_i| + |\beta_{i-1}|)$)
      b := max(b , $\alpha_i + ( | \beta_i| + |\beta_{i-1}|)$)
  end for
  $i$ := 1, s:= 1
  while (|b−a| < $eps$(|a| + |b|)) do
      $\lambda$ := (a + b)/2
      p := $\alpha_i - \lambda - \beta_i^2$/s
      if (p > 0) then
          a := $\lambda$, $i$ := 1, s := 1
      else if ($i >= r$) then
          b := $\lambda$, $i$ := 1, s := 1
      else
          s := p, $i := i + 1$
      end if
  end while
  return $\lambda$
```

Loss of orthogonality among the vectors is an inherent problem with the Lanczos Iteration and often a costly step of re-orthogonalization is introduced [39], which has a computational complexity of $O(rn)$ and a storage requirement of $O(rn)$ as well. However, investigation of the loss of orthogonality reveals it does not affect extremal eigenvalues [34] and therefore we do not need to perform re-orthogonalization. This phenomenon is shown in Figure 4.3 where the loss of orthogonality affects only the 2nd and 3rd largest eigenvalues (they mis-converged to the maximum eigenvalue). We exploit this behaviour of the Lanczos Iteration to design an architecture highly specialized for extremal eigenvalues computation.

### 4.2.2 Bisection Method

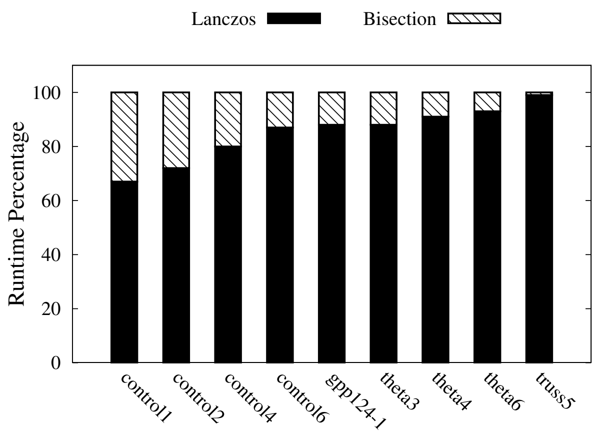
The bisection method is an efficient method for finding the eigenvalues of a symmetric tridiagonal matrix $T_r$ and it has a computational complexity of $O(r)$ for extremal eigenvalue computation [34]. An extremal eigenvalue of the matrix $T_r$ is computed by finding an extremal root of the polynomial.

$$p_r(\lambda) = \det(T_r - \lambda \mathrm{I}). \tag{4.4}$$

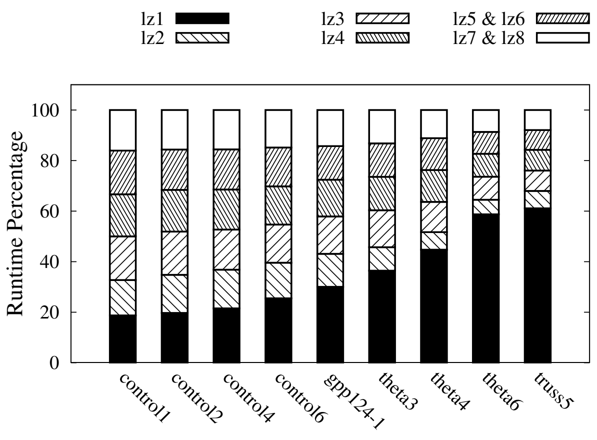
The extremal root can be computed recursively from the roots of the polynomials $p_i(\lambda)$ where $0 \leq i \leq r$ (see Algorithm 12). We find out the interval where the maximum or minimum eigenvalue is located and only perform bisection method in this interval instead of the whole spectrum.

## 4.3 Sequential Runtime Analysis

We use sequential runtime analysis to find out the computationally intensive part of the flow shown in Figure 4.2. We pick small to medium size matrices from SDPLIB [21], a collection of benchmarks for solving SDPs, and use the Intel MKL library for sequential implementation on an Intel Xeon X5650. We show the runtime distribution in Figure 4.4(a) and observe that it is the Lanczos Iteration which takes most of the time reaching 99% for higher order matrices. On the other hand, the time taken by the bisection method is independent of the problem size as it only varies with the number of desired eigenvalues and their distribution [39]. We therefore focus our attention on parallelizing the Lanczos Iteration.



(a) Extremal Eigenvalues Computation.



(b) The Lanczos Iteration.

Figure 4.4: Sequential Runtime Analysis.

The runtime distribution of the Lanczos Iteration is plotted in Figure 4.4(b) demonstrating that $lz1$ (matrix-vector multiplication) is the dominant operation with $O(rn^2)$ FLOPs and $O(rn^2)$ memory operations for $r$ Lanczos iterations. As memory access is costly compared to performing the floating-point operations, we need to minimize this cost in order to accelerate the Lanczos Iteration. We will now discuss how we accelerate Lanczos Iteration by minimizing memory access time using FPGAs.

## 4.4 Accelerating Lanczos Iteration using FPGAs

### 4.4.1 Cost Model

If we consider the Lanczos Iteration in Algorithm 11, we need to perform matrix-vector multiplication ($lz1$) in each iteration followed by some vector-vector operations. Assuming the vectors ($q_{i-1}$, $q_{i-2}$) and the matrix are stored in the off-chip memory and therefore need to be fetched in each iteration, the total cost for $r$ iterations can be shown by Equation (4.7).

$$t_{comm} = \#msg \times \alpha + msize \times \beta. \tag{4.5}$$

$$t_{comp} = flops \times \gamma. \tag{4.6}$$

$$t_r = r \times t_{comm} + r \times t_{comp}. \tag{4.7}$$

The cost model is based on LogP model [28] which comprises three terms *i.e.* latency $\alpha$, inverse bandwidth $\beta$ and the time per flop $\gamma$. The first two terms contribute towards the communication cost (Equation (4.5)) and the last term determines the computation cost (Equation (4.6)). We are interested in minimizing the communication cost in the Lanczos Iteration as it is a communication-bound computation (Section 4.2.1) but at the same time we intend to achieve high throughput to reduce the computation cost. Our desired cost model is shown in Equation 4.8.

$$t_r^{'} = \frac{r}{k} \times t_{comm} + r \times t_{comp}^{'} \qquad \text{where} \quad t_{comp}^{'} < t_{comp} \tag{4.8}$$

We assume $t_{comp}$ is the computation cost of a sequential implementation. While the algorithmic parameter $k$ in Equation (4.8) can be used as a control knob to reduce the communication cost, its maximum value depends on the underlying architecture and its memory model. For *small-to-medium size dense* problems, we only consider two choices of $k$, *i.e.* $k = 1$ where the matrix is fetched from off-chip memory in each iteration and $k = r$ where it is loaded in on-chip memory only once. This restriction is because of the dependencies between successive operations in the Lanczos Iteration. In subsequent chapters, we show how this dependency can be broken and the value of $k$ greater than 1 and less than $r$ is used to minimize communication cost for *large-scale sparse problems*. This becomes possible only at the expense of redundant computation which increases with $k$. In the particular case of $k = 1$ or $k = r$, the computation cost is independent of $k$ as we are not performing any redundant computation.

### 4.4.2 FPGAs vs. GPU

Two most common hardware accelerators used in scientific computing are the GPUs and the FPGAs. The efficiency of both of these devices depends on the nature of the problem, *i.e.* whether it is a compute-bound problem or a communication-bound problem. We perform a comparative analysis of the two devices for the following two cases.

**Case 1: Fully Off-Chip ($k$ =1)**

In this case, the matrix is loaded from off-chip memory in each Lanczos iteration. We observe from Table 3.1 that, compared to FPGAs, the GPU has ∼5× higher off-chip memory bandwidth and ∼2× higher peak floating-point performance. One should expect GPU to perform better theoretically. Nonetheless, the performance is bounded by its off-chip memory bandwidth, *i.e.* with 2 flops per 4 bytes (single-precision) in dominant matrix-vector multiplication, the maximum theoretical performance of GPU is 72 GFLOPs with an efficiency of 7% (actual efficiency is even lower as maximum memory bandwidth is not utilized practically). Likewise, the maximum performance of the FPGAs is 17 GFLOPs with an efficiency of 4%. Although the efficiency of GPU and FPGA is very low, an arbitrarily large problem can be solved.

**Case 2: Fully On-Chip ($k = r$)**

In this case, the matrix and the vectors are loaded once and then they are re-used for $r$ Lanczos iterations. GPU has an inverse memory hierarchy [96], *i.e.* the registers have large capacity as compared to the on-chip shared memory. However, this is distributed among all streaming multiprocessors (SMs). If the total on-chip capacity is utilized to solve one large problem by all SMs, the data dependency between different operations in Algorithm 11 enforces the results to be stored back in the off-chip global memory. On the other hand, multiple independent problems can be solved by storing the matrices in the individual register file of each SM, however, the maximum size will be restricted, *e.g.* on C2050 GPU having a 128KB register file, the maximum size of the matrix that can be stored is 181 (in single-precision).

In contrast, compared to GPUs, FPGAs not only have 2× on-chip memory capacity (see Table 3.1) but also the memory is less spatially distributed and can be used to solve relatively large problems ($n$∼335). Additionally, FPGAs have ∼5× larger on-chip memory bandwidth which can be used to saturate the floating-point cores to achieve much higher silicon efficiency. We now discuss the parallelism potential within the Lanczos Iteration followed by an architecture which exploits this potential.

### 4.4.3 Parallelism Potential

We can identify the parallel potential in the Lanczos Iteration from its dataflow graph shown in Figure 4.5. We plot the number of floating-point operations per Lanczos iteration as well as the critical latency assuming ideal parallel hardware as a function of the matrix size in Figure 4.6. We find out that the work grows with $O(n^2)$ due to dominant matrix-vector multiplication ($lz1$) whereas the latency grows with $O(\log\, n)$, *i.e.* the latency of a single dot product circuit [61] (assuming there are $n$ such dot product circuits working in parallel to perform matrix-vector multiplication). Thus the Lanczos Iteration has a high degree of parallel potential.

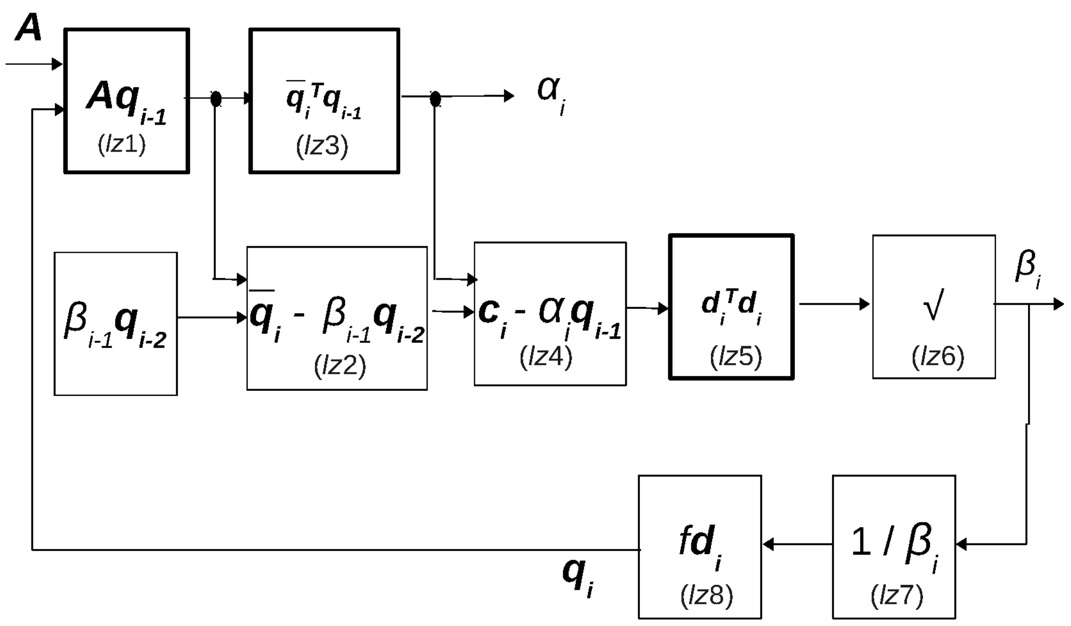


Figure 4.5: Data flow graph of the Lanczos Iteration.

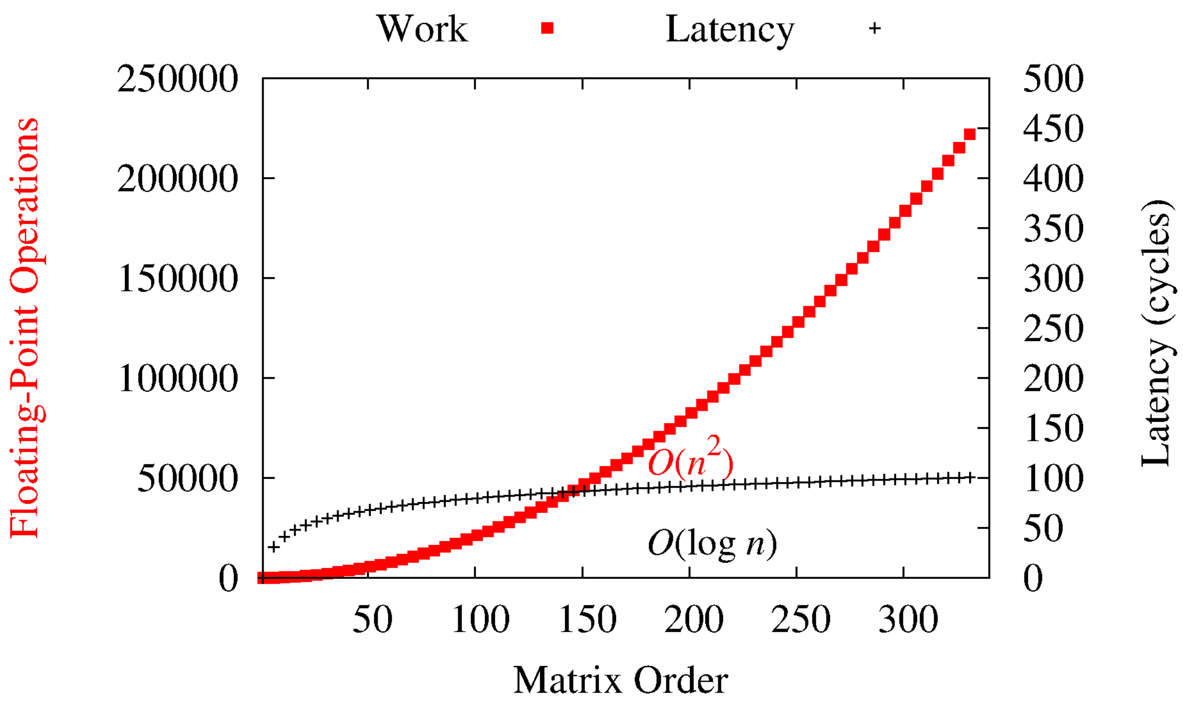


Figure 4.6: Work vs. Latency

We use high on-chip memory bandwidth to implement single dot product circuit as a reduction tree [61] and perform matrix-vector multiplication in a pipelined fashion as $n$ dot products where a new dot product is launched every clock cycle. From the dataflow in Figure 4.5, we also observe **thread-level parallelism** as some of the operations may be performed in parallel, *e.g.* ($lz2$) and ($lz3$). The operations ($lz2$), ($lz4$) and ($lz8$) can be unrolled completely with a latency of $\Theta(1)$ but since they contribute very little to the overall runtime of the system (see Section 4.3), we implement these sequentially.

We also use **pipeline parallelism** available in the architecture to solve multiple independent extremal eigenvalue problems arising in SDPs and eigenvalue based channel sensing while having the latency of solving a single extremal eigenvalue problem (see Section 4.4.5).

### 4.4.4 System Architecture

We present an architecture consisting of a parallelized Lanczos Iteration coupled to a sequential architecture for the bisection method (due to the latter's very low runtime contribution, see Section 4.3). We capture the high-level stream organization of the FPGA architecture in Fig 4.7.

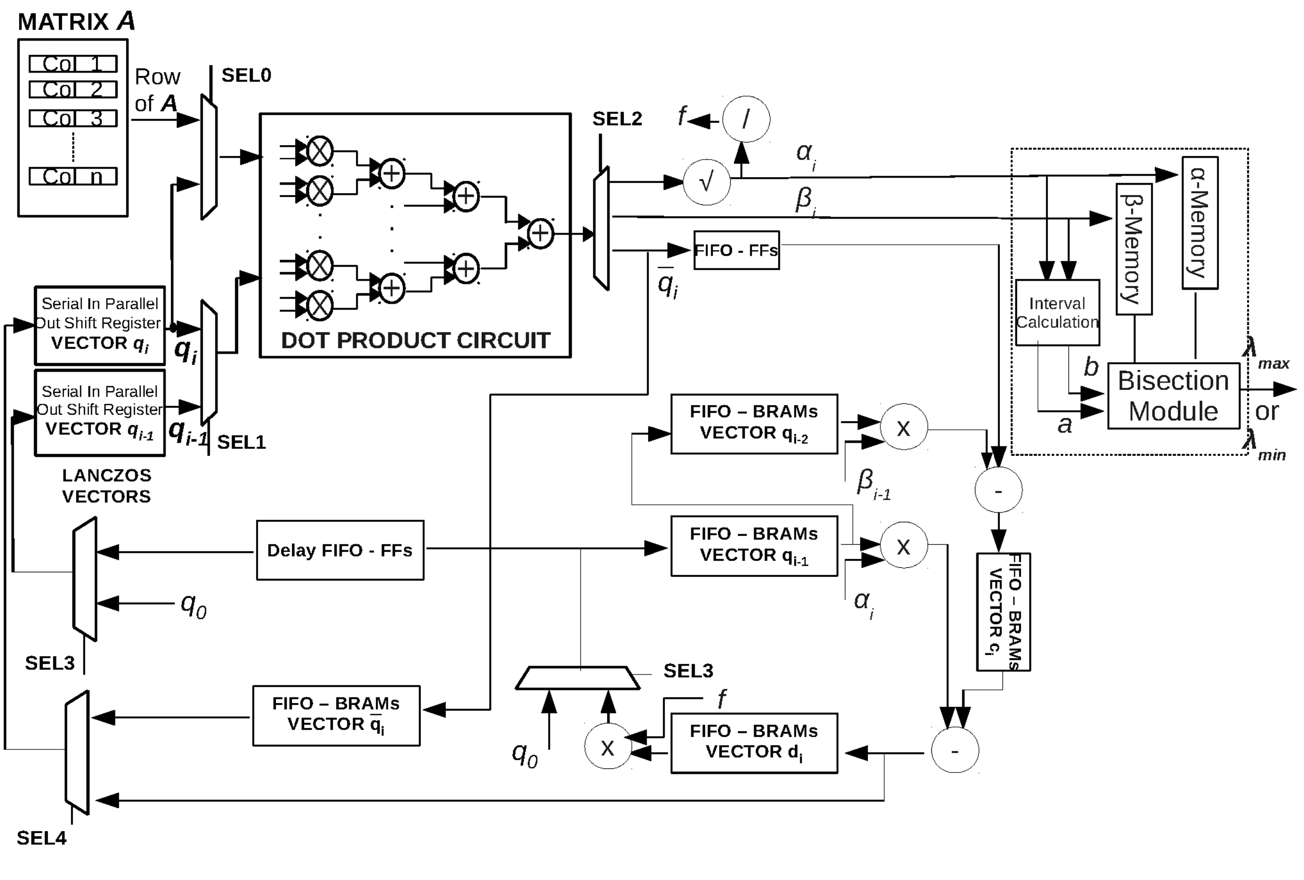


Figure 4.7: Partial schematic for the implementation of the Lanczos Iteration and the bisection method displaying main components including a dot product circuit module, FIFOs for storing Lanczos vectors ($q_{i-1}$, $q_{i-2}$) banked memory arrangement for matrix $A$, two memories for storing $\alpha_i$ and $\beta_i$ and a Bisection Module.

The matrix $A$ is loaded **only once** in the on-chip memory of the FPGA in a banked column fashion, and then the rows of $A$ are streamed along with the Lanczos vector $q_{i-1}$ to launch a new dot product in each clock cycle. We use deeply pipelined floating-point (FP) operators and aim to keep them busy all the time thus getting maximum throughput. The number of FP units is given by

$$\text{Total FP Units}(n) \quad = \quad 2n + 6. \tag{4.9}$$

Referring to Equation (7.2), $2n$ - 1 units are used for the dot product circuit whereas 7 FP units are used for other operations.

The Interval Calculation module computes the initial interval for the extremal eigenvalue using the Gershgorin circle theorem [39] and the Bisection Module computes the extremal eigenvalue in that interval in a sequential fashion as shown in Algorithm 12.

### 4.4.5 Solving Multiple Extremal Eigenvalue Problems

For solving a single extremal eigenvalue problem, the deeply pipelined nature of the dot product circuit in Figure 4.7 leads to high throughput but also considerable latency. As a result, the pipeline will be under-utilized if only a single problem is solved. Therefore, the mismatch between throughput and latency is exploited to solve multiple independent extremal eigenvalue problems. The initiation interval of this circuit is $n + 2$ clock cycles (for $lz1$, $lz3$ and $lz5$) after which a new problem can be streamed into this circuit. The pipeline depth (P) of the circuit is given by Equation (6.5) which indicates how many problems can be active in the pipeline at one time.

$$\text{Latency}(n) \quad = \quad 3n + c_1\lceil \log_2 n \rceil + c_2. \tag{4.10}$$

$$\text{Pipeline Depth P}(n) \quad = \quad \left\lceil \frac{3n + c_1\lceil \log_2 n \rceil + c_2}{n+2} \right\rceil. \tag{4.11}$$

Referring to Equation (4.10), the $3n$ comes from $n$ cycles for ($lz1$) and $2n$ cycles for ($lz4$) and ($lz8$). The $\log_2 n$ term comes from the adder reduction tree and $c_1 = 36$ and $c_2 = 137$ derive from the latencies of the single precision floating-point operators. We can see from Equation (4.11) that the number of independent eigenvalue problems approaches to a constant value for large matrices (P $\rightarrow$ 5 as $n \rightarrow \infty$).

### 4.4.6 FPGA I/O Considerations

We use double-buffering to store P matrices so that as one set of P problems are being solved, other set is being loaded from the off-chip memory. The total bytes required for P matrices, the initial vector and the eigenvalue output is given by Equation (4.12).

$$\text{I/O Bytes} = 4\text{P}(n^2 + n + 1) \tag{4.12}$$

The extremal eigenvalues start converging after a few iterations as shown in Figure 4.3. If we take $r$ as the number of Lanczos iterations, then the I/O bandwidth can be calculated by Equation (4.13).

$$\text{I/O Bandwdith} = \text{Freq} \times \frac{\text{I/O Bytes}}{(r \times \text{Latency})} \tag{4.13}$$

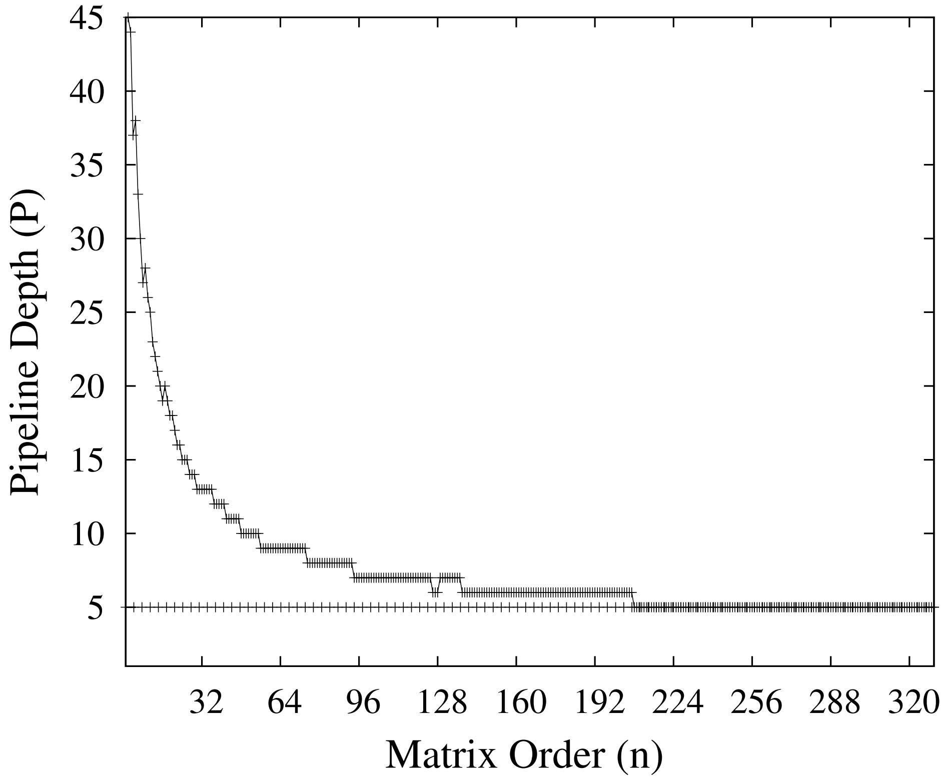


Figure 4.8: Asymptotic analysis of maximum number of problems.

For the maximum matrix size $n$ to be equal to 335, average number of Lanczos Iterations $r$ equal to 30 and a maximum operating frequency of 260 MHz (see Section 4.5), the required I/O bandwidth is ∼8 GB/s. This I/O bandwidth is approximately 25% of the maximum off-chip memory bandwidth available on Virtex6-SX475T FPGA as shown in Table 3.1.

## 4.5 Methodology

The experimental setup for performance evaluation is summarized in Table 4.1 and dense matrices are extracted from SDPLIB [21] benchmarks shown in Table 4.2. We implement the design for a single problem and estimate the results for multiple problems. We implement the proposed architecture in VHDL and synthesize the circuit for Xilinx Virtex6-SX475T FPGA, a device with the largest number of DSP48Es and a large on-chip capacity. The placed and routed design has an operating frequency of 260 MHz. We actually measure the latency of solving a single problem and then estimate the total number of problems P that can be solved in a pipeleined fashion. For a matrix of size 335×335, P is equal to 5 and we occupy nearly all the BRAMs available in the device. There is 50% utilization for the DSP48Es and 70% for the Slice LUTs and they show a linear increase as the number of floating-point units grows with $O(n)$. Optimized Basic Linear Algebra Subroutine (BLAS) libraries are used for the multi-core (Intel MKL) and GPU (CuBLAS) implementations to perform all operations in Algorithm 11. In the multi-core case, the number of threads is set equal to the number of physical cores whereas in case of a GPU, the grid configuration is picked by CuBLAS automatically. We do not use multi-

Table 4.1: Experimental Setup (FPGA clock frequency is not reported in [87] for peak single-precision GFLOPs).

| **Platform** | **Peak GFLOPs** Single Precision | **Compiler** | **Libraries** | **Timing** |
|---|---|---|---|---|
| Intel Xeon X5650 | 127.8 [27] | gcc (4.4.3(-O3)) | Intel MKL (10.2.4.032) | PAPI (4.1.1.0) |
| Nvidia GPU C2050 | 1050 [71] | nvcc | CUBLAS (3.2) | cudaEvent-Record() |
| Xilinx Virtex6-SX475T | 450 [87] | Xilinx ISE (10.1) | Xilinx Coregen | ModelSim |

Table 4.2: Benchmarks.

| **Benchmark** | $n$ |
|---|---|
| control1 | 15 |
| control2 | 30 |
| control4 | 60 |
| control6 | 90 |
| gpp124-1 | 124 |
| theta3 | 150 |
| theta4 | 200 |
| theta6 | 300 |
| truss5 | 335 |

threading in the multi-core for solving multiple eigenvalue problems. We do not consider the host to FPGA/GPU transfer time because it is negligible compared to the actual computation as shown in Figure 4.9.

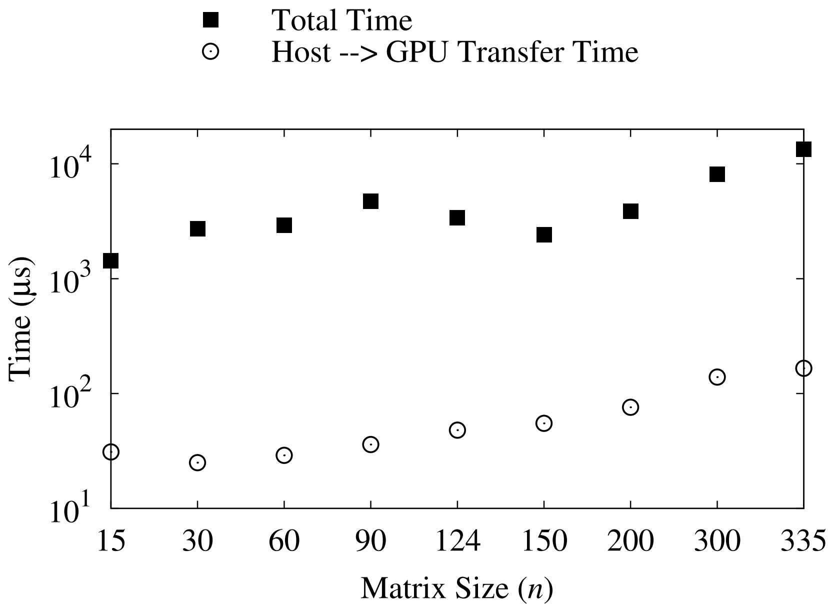


Figure 4.9: Total time vs. data transfer from host to GPU (Nvidia C2050 Fermi) with data averaged over 20 runs. The input matrix $A$ (from benchmarks in Table 4.2) is transferred only once and the Lanczos Iteration is run until we get an error less than $10^{-5}$ in the maximum eigenvalue.

We only consider single-precision implementation because for the applications of our interest, we require an accuracy of $10^{-3}$ in the maximum eigenvalue. In Figure 4.10, we show how the error in the maximum eigenvalue decreases as we increase iterations for both single-precision and double-precision implementations. We observe that to achieve the desired accuracy, the number of Lanczos iterations are approximately the same (with a difference of one iteration) for both single-precision and double-precision implementations. This behaviour results from the fact that the precision affects only the inner eigenvalues as shown in Figure 4.3, where the maximum eigenvalue once converged, remains unchanged. We therefore choose single-precision representation because it saves us in terms of FPGA resources that leads to not only savings ($\sim 2\times$) in terms of area but also storage as well.

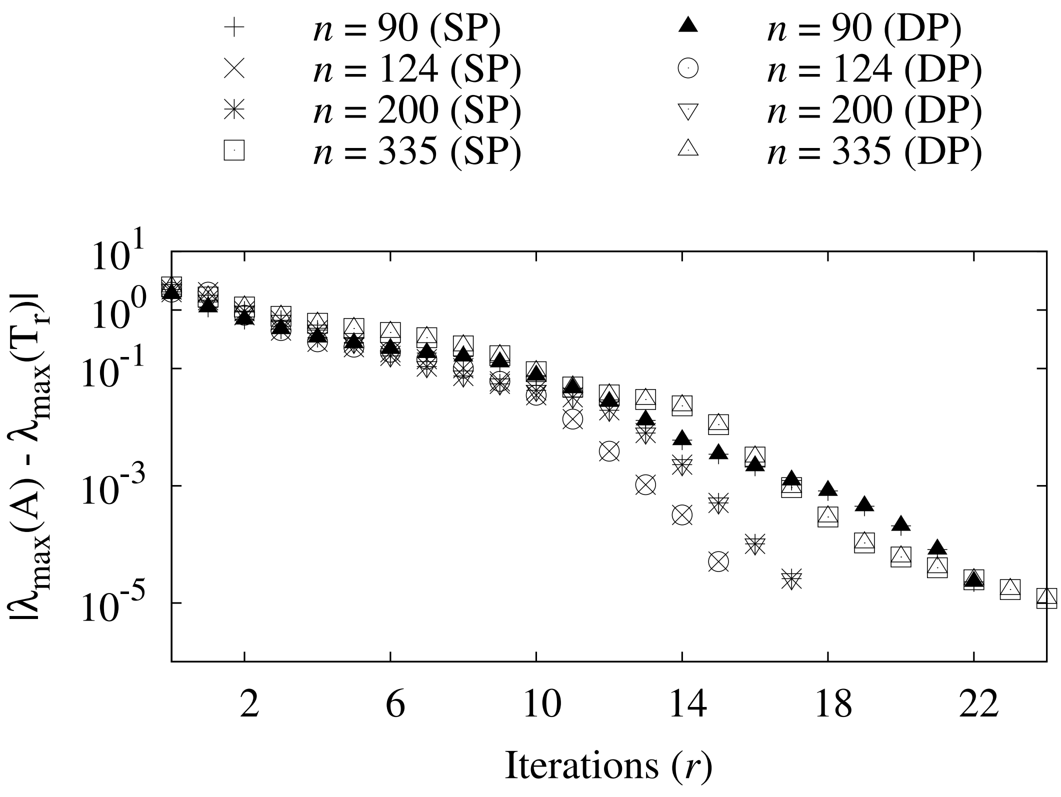


Figure 4.10: Single-precision (SP) vs. Double-precision (DP) on Nvidia C2050 Fermi. The input matrix $A$ is transferred only once and the Lanczos Iteration is run until we get an error less than $10^{-5}$ in the maximum eigenvalue.

## 4.6 Results

We now present the performance achieved by our FPGA design and compare it with previous FPGA-based implementations. We then compare our work with multi-core and GPU and then discuss the underlying factors that explain our results.

### 4.6.1 FPGA Performance Evaluation

As we load the matrix only once and that too in an overlapped fashion for a set of P problems, we do not consider the time for loading the matrix. We analyze the performance based on the time spent in floating-point operations. The peak and sustained single-precision floating-point performance of the FPGA is given by Equation (4.14) and Equation (4.15) respectively. For a matrix of size 335×335 and an operating frequency of 260 MHz, the peak performance of our design is 175 GFLOPs and sustained performance is 35 GFLOPs for a single problem. The sustained performance approaches the peak performance when P problems are solved simultaneously. There is a linear increase in performance with the problem size as the floating-point units grow with $O(n)$.

$$\begin{aligned}
\text{Peak Throughput} &= \text{Total FP Units} = 2n + 6 \quad \text{FLOPs/cycle.} && (4.14)\\
\text{Sustained Throughput} &= \frac{\mathrm{P}\ (2n^2 + 8n)}{\mathrm{P}(n+2) + \mathrm{P} - 1}\ \text{FLOPs/cycle.} && (4.15)\\
\text{Efficiency}(n) &= \frac{\mathrm{P}(2n^2 + 8n)}{\text{Total FP Units} \times (\mathrm{P}(n+2) + \mathrm{P} - 1)}. && (4.16)
\end{aligned}$$

where $2n^2 + 8n$ represent the number of floating-point operations per Lanczos iteration. It is observed that even for low order matrices a high efficiency (70%) is achieved for the architecture in Figure 4.7 and that the efficiency tends to 100% for large matrices. This is because the number of floating-point operators in the dot product circuit grows linearly with $n$ and by design the the dot product circuit remains busy. However, the overall efficiency of the FPGA is low as not all of the FPGA resources are utilized due to insufficient BRAMs (see Section 4.5).

### 4.6.2 Comparison with FPGA-based Eigensolvers

We briefly survey the existing FPGA-based eigensolvers. Ahmedsaid *et al.* [4], Liu *et al.* [60] and Bravo *et al.* [23] target eigenvalue applications involving the Principal Component Analysis (PCA) where the matrix size does not usually go over 20 × 20. The direct method of Jacobi [39] and its variants are used for these small eigenvalue problems. The main reason behind using the Jacobi method is its inherent parallelism which can be exploited by systolic architectures requiring $\frac{n}{2}$ diagonal processors and $\frac{n(n\ \text{-}\ 1)}{4}$ off-diagonal processors for a $n \times n$ matrix. Existing FPGA-based eigensolvers are not suitable for the extremal eigenvalue computation of matrices of our interest for two reasons. Firstly, they target very small matrices and are not resource efficient as the number of processors grows with $O(n^2)$. Additionally, since the Jacobi method inherently computes all eigenvalues and eigenvectors, the approach utilized in these architectures is wasteful for computing only the extremal eigenvalues. Table 4.3 summarizes the previous eigenvalue computations on FPGA with device, method, precision and the performance results.

Table 4.3: Comparison of FPGA-based Symmetric Eigenvalues Computation.

| **Ref.** | **Method** | $n$ | **Device** | **Eigen-values** | **Freq.** MHz | **GFLOPs** | **Precision** | **Resources** (asymptotic) |
|---|---|---|---|---|---|---|---|---|
| [4] 2003 | Direct | 8 | Virtex-E | All | 84.44 | Not Reported | fixed point (16-bit) | $O(n^2)$ |
| [60] 2006 | Direct | 8 | Virtex-II | All | 70 | Not Reported | fixed point (16-bit) | $O(n^2)$ |
| [23] 2006 | Direct | 16 | Virtex-II Pro | All | 110 | 0.243 | fixed point (18-bit) | $O(n^2)$ |
| This Work 2011 | Iterative | 335 | Virtex-6 | Extremal | 260 | 175 | floating point (32-bit) | $O(n)$ |

Unlike previous approaches, the proposed architecture can address medium-size ($n \sim$ a few 100s) problems and is resource efficient as the number of floating-point units grows with $O(n)$. Additionally due to repeated matrix-vector multiplication, we show much higher throughput compared to division and square root operations in direct methods.

### 4.6.3 Comparison with multi-core and GPU

Our multi-core and GPU implementations are based on the optimized BLAS libraries with two limitations. First, the data between different BLAS routines is shared using off-chip memory. Secondly, unlike FPGAs, there is no explicit cache blocking and the matrix is fetched in each iteration. The runtime percentage of each part of the Lanczos Iteration is plotted in Figure 4.11 for different architectures. We take the runtime on single-core as the baseline. In the case of an FPGA, matrix-vector multiplication ($lz1$) is computed as $n$ dot products where each new dot product is launched every clock cycle. Due to high on-chip memory bandwidth and the streaming nature of the architecture, the overall design has much lower latency and high throughput. We get a speed up of 7.6-25.9× (12.5× geo. mean) using the high on-chip memory bandwidth and high throughput architecture for dot product. We additionally get a 1.04-1.22× (1.14× geo. mean) using **thread-level parallelism** with a combined speed up of 8.8-27.3× (13.4× geo. mean) for solving a single eigenvalue problem as shown in Figure 4.12(a). Using **pipeline parallelism** we additionally get a speed up of 4.3-19.1× (7.19× geo. mean) and, therefore deliver an overall speed up of 41-520× (103× geo. mean) when solving P independent eigenvalue problems.

Although GPUs are highly efficient for dense linear algebra, we observe in Figure 4.11 that

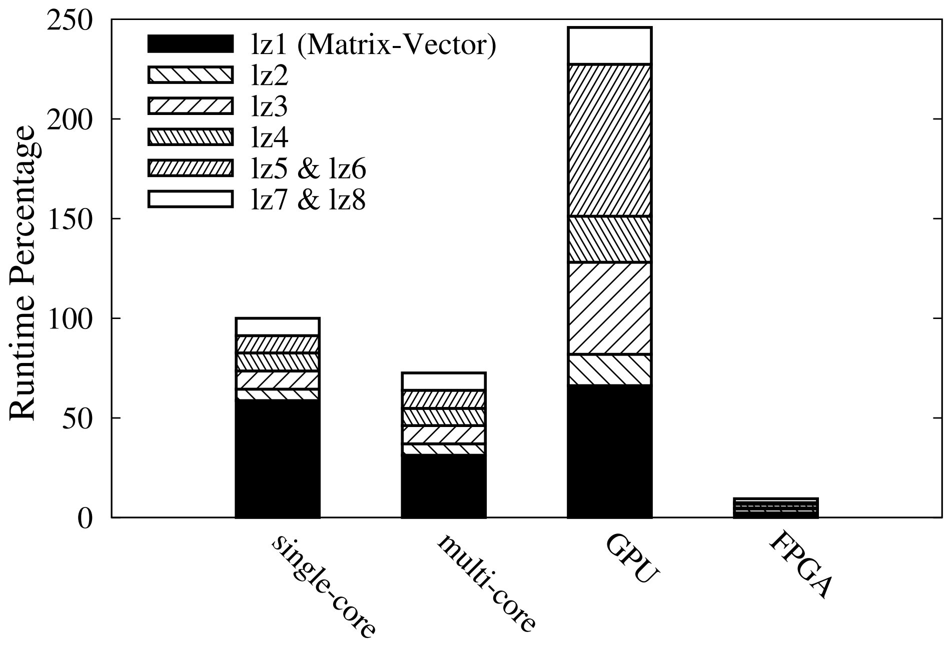
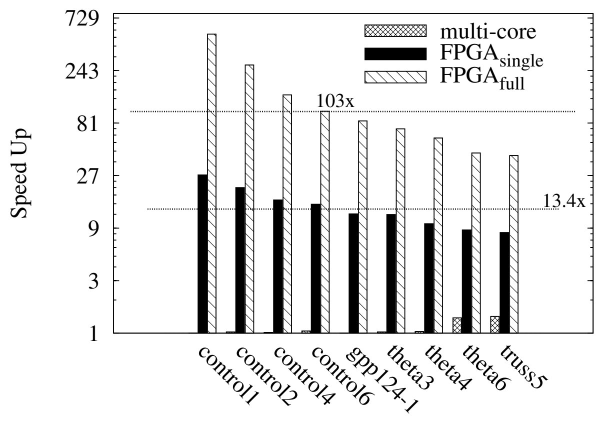
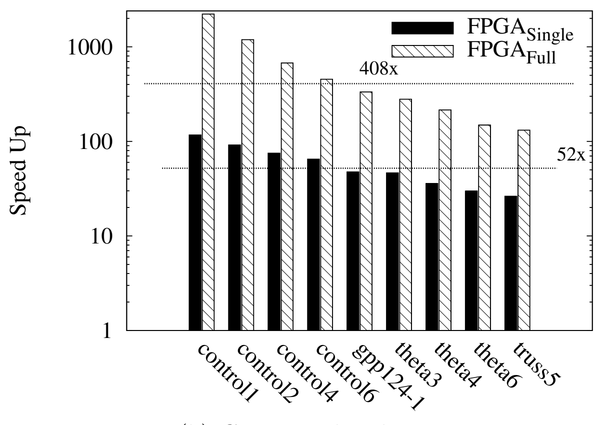


Figure 4.11: Runtime Breakdown (truss5).

the performance of GPU is even worse than the multi-core. This is due to medium-size data sets ($n \sim$ a few 100s) for which the GPU exhibits a performance less than 1 GFLOPs for BLAS 1 ($lz2$ to $lz8$) and BLAS 2 ($lz1$) operations [11]. Additionally, we are using CuBLAS routines for matrix-vector multiplication which does not cache the matrix in the shared memory or register file of the GPU and therefore the matrix is fetched repeatedly from off-chip memory in each iteration. When compared to our FPGA design, we get a speed up of 24.5-110.7× (49.4× geo. mean) using high bandwidth on-chip memory and high throughput architecture for dot



(a) single-core as a baseline.



(b) GPU as a baseline.

Figure 4.12: Performance Comparison ('single' is for 1 problem on FPGA, 'full' is for P problems on FPGA).

product. We additionally get 1.04-1.22× (1.14× geo. mean) using **thread-level parallelism** with a combined speed up of 26.2-116x× (52.8× geo. mean) for solving a single eigenvalue problem shown in Figure 4.12(b). Using **pipeline parallelism** we additionally get a speed up of 4.3-19.1× (7.19× geo. mean) and, therefore deliver an overall speed up of 131-2220× (408× geo. mean) when solving P independent eigenvalue problems.

### 4.6.4 Silicon Efficiency

The raw performance of our FPGA design is compared with that of the multi-core and GPU implementations in Figure 4.13. We observe that FPGA shows better performance primarily because we use explicit cache blocking and utilize high bandwidth on-chip memory. We show that the sustained performance approaches peak performance of the architecture as we solve multiple independent problems. We finally compare the silicon efficiency of all architectures as a proportion of peak performance (see Table 4.1) in Figure 4.14. For the maximum matrix size, the efficiency of the FPGA when solving single problem is approximately 7.79% whereas with P problems it increases to 38.9% of the peak performance. The efficiency of multi-core is around 3.34% and that of GPU is 0.13% and their efficiency increases with the problem size.

## 4.7 Summary

We use large on-chip memory of FPGA for explicit cache blocking to load matrix once and then re-use it for all Lanczos iterations to solve symmetric extremal eigenvalue problem. As the Lanczos Iteration is memory-bound, we use high-bandwidth on-chip memories to saturate deeply pipelined floating-point cores to demonstrate a sustained performance of 175 GFLOPs with an overall silicon efficiency of 38%. In contrast, we show that the multi-core and GPU are under-utilized for the medium-size data sets as the matrix is accessed from off-chip memory in each iteration. As a result they can achieve an efficiency of 3.34% and 0.13% respectively. We therefore highlight iterative numerical algorithms with high memory bandwidth requirements but medium-size data sets as highly appropriate for FPGA acceleration.

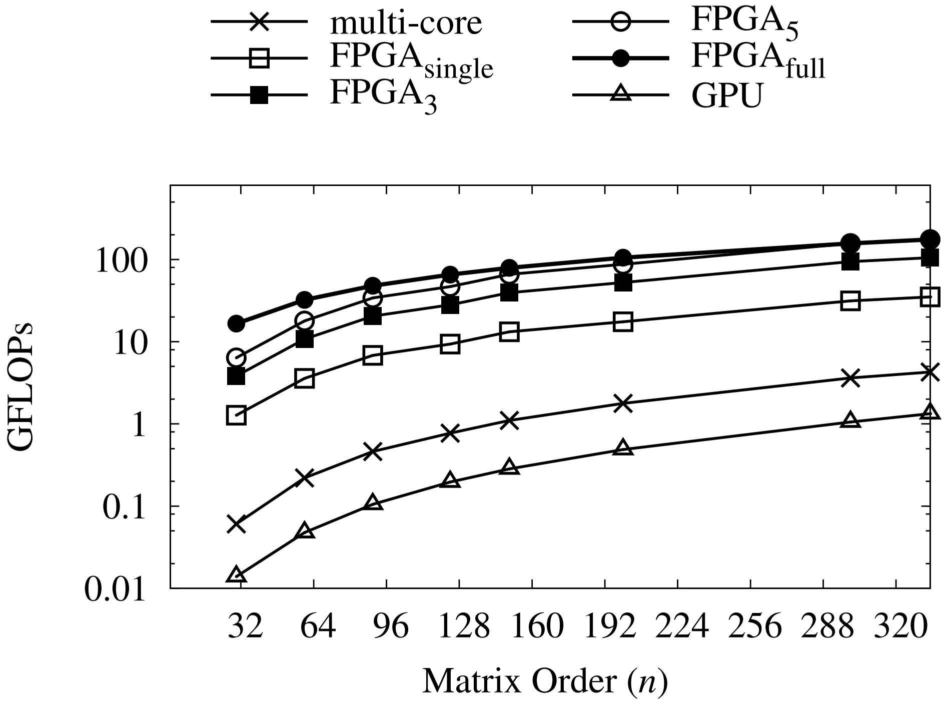


Figure 4.13: Raw Performance Comparison ('single' is for 1, 'full' is for P problems).

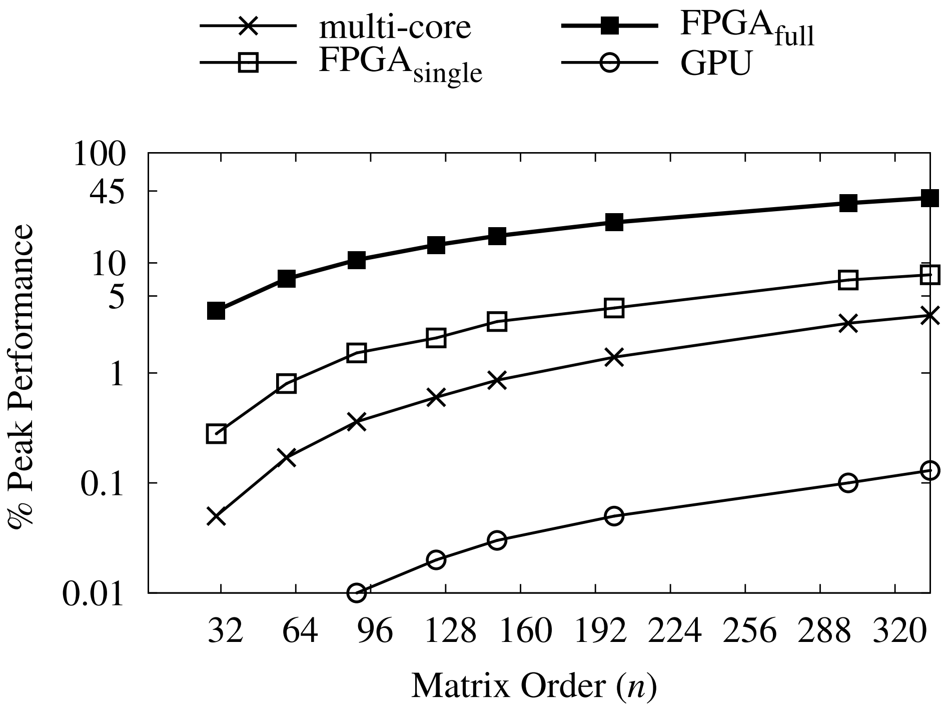


Figure 4.14: Efficiency Comparison ('single' is for 1, 'full' is for P problems).

# 5 Matrix Powers Kernel

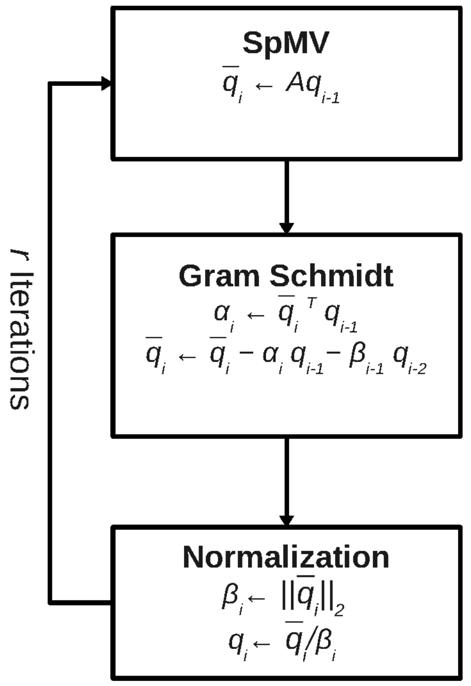
In the previous chapter, we target small-to-medium size dense problems where the matrices are fetched only once and can be re-used for $r$ iterations. However, if the matrices are large and can not be stored on-chip, they need to be fetched in each iteration from the off-chip memory. As a result, performance is bounded from above by the off-chip memory bandwidth of the modern computing platforms. While this is true for dense problems, for sparse problems, we can trade communication with redundant computation by algorithmic transformation discussed in Chapter 2. An important kernel in such a communication-avoiding approach is a matrix powers kernel which replaces SpMV in iterative numerical algorithms as shown in Figure 5.1. The key idea is to partition the matrix into blocks and performs 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. While $k$ iterations of an iterative numerical algorithm can be unrolled using this matrix powers kernel to provide $\Theta(k)$ reduction in communication cost, the extent of this unrolling depends on the underlying architecture, its memory model and the growth in redundant computation. In this chapter, we present a systematic procedure to select this algorithmic parameter $k$ which provides communication-computation tradeoff on hardware accelerators like FPGA and GPU as shown in Figure 5.2. 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 3.1 (see page 65). 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. We provide predictive models to understand this tradeoff and show how careful selection of $k$ can lead to performance improvement which otherwise demands significant increase in memory bandwidth.

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 [17] for a range of randomly generated banded matrices.

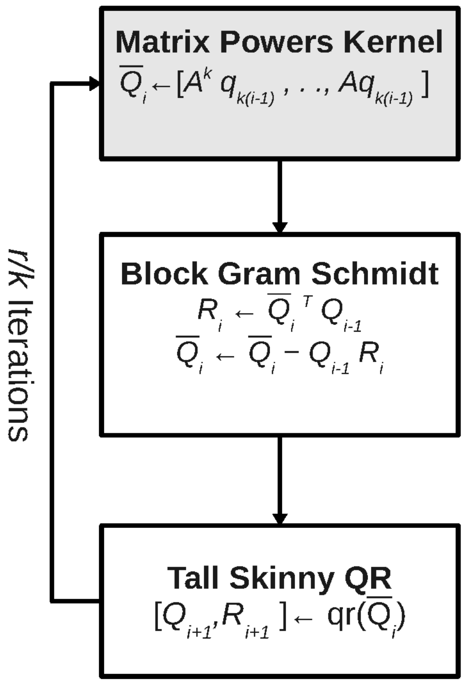
2. An architecture-aware matrix powers kernel that matches the strength of the FPGAs to avoid redundant computation and a resource-constrained methodology to pick $k$ for a particular FPGA.

3. A unified predictive model of the matrix powers kernel for GPU and FPGA, which helps



SpMV
$\overline{q}_i \leftarrow Aq_{i-1}$
Gram Schmidt
$\alpha_i \leftarrow \overline{q}_i^T q_{i-1}$
$\overline{q}_i \leftarrow \overline{q}_i - \alpha_i q_{i-1} - \beta_{i-1} q_{i-2}$
Normalization
$\beta_i \leftarrow \|\overline{q}_i\|_2$
$q_i \leftarrow \overline{q}_i/\beta_i$
r Iterations

(a) Lanczos Iteration



Matrix Powers Kernel
$\overline{Q}_i \leftarrow [A^k q_{k(i-1)}, \ldots, Aq_{k(i-1)}]$
Block Gram Schmidt
$R_i \leftarrow \overline{Q}_i^T Q_{i-1}$
$\overline{Q}_i \leftarrow \overline{Q}_i - Q_{i-1} R_i$
Tall Skinny QR
$[Q_{i+1}, R_{i+1}] \leftarrow \mathrm{qr}(\overline{Q}_i)$
r/k Iterations

(b) CA-Lanczos

Figure 5.1: Lanczos Iteration vs. Communication-Avoiding Lanczos (CALanczos) with emphasis on the Matrix Powers Kernel.

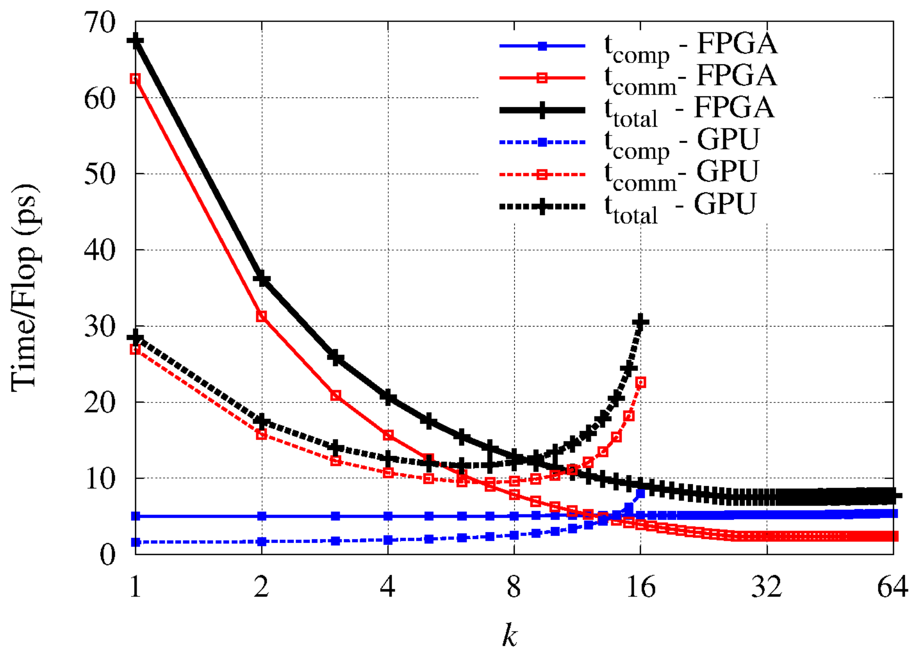


Figure 5.2: Computation-communication tradeoff for a banded matrix with band size 27 and $n$ = 1M on a Virtex6-SX475T FPGA and C2050 Fermi GPU.

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×−3.2× and 1.7×−4.4× speedup over GPU for largest and smallest band sizes respectively.

## 5.1 Algorithms for Matrix Powers Kernel

Although we provided necessary background about the matrix powers kernel in Chapter 2, we discuss important algorithms here, which we will refere in the rest of this chapter. We specifically target banded matrices, which naturally arise in numerous scientific computations like stencils in partial differential equation (PDE) solvers [84] and semi-definite optimization problems [1].

### $k$ SpMVs

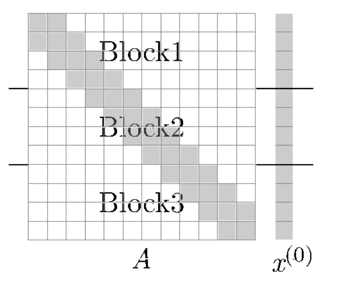
Given an $n{\times}n$ sparse matrix $A$ with band size $b$, a dense vector $x^{(0)}$ of length $n$, the matrix powers kernel is computed as

$$x^{(i)} \quad = \quad Ax^{(i-1)} \qquad 1 \leq i \leq k \tag{5.1}$$

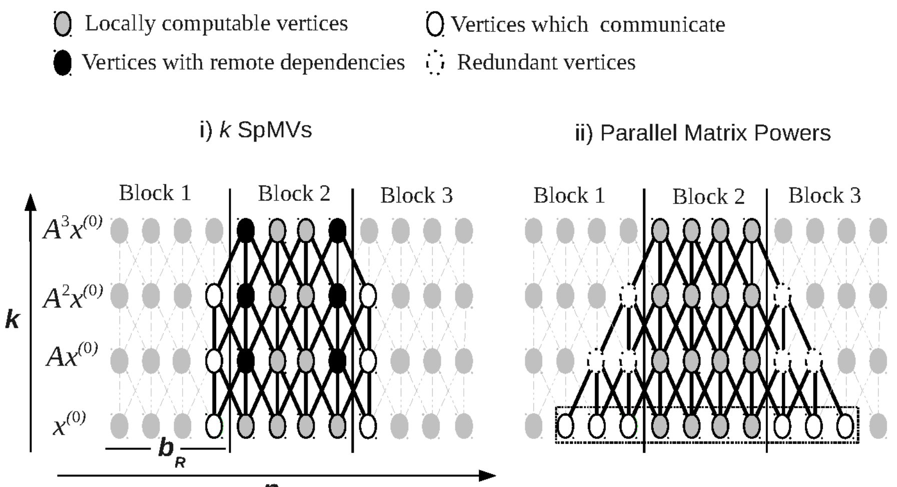
The computation in Equation (5.1) can be unrolled for $k$ iterations as a graph as shown in Figure 5.3(b) for an example tri-diagonal matrix $A$ (see details of the graph notation in Chapter 2). Each vertex of the graph represents vector entry $x_j^{(i)}$ and there are $n$ vertices for $j$ = 0, 1,. . . , $n - 1$ entries for each level $i$ = 0, 1,. . . , $k$. 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. In order to parallelize, the graph can be partitioned into $N_q$ blocks where each block $q$ can be mapped on a single processor. We can generate $k$ vectors using repeated SpMV, we need to synchronize after each step to get the corresponding entries of the previous vector which are computed by the neighbouring processors as shown by white vertices in Figure 5.3(b)(i). This frequent synchronization in $k$ SpMVs leads to a latency-bound problem in parallel architectures.

### Parallel Matrix Powers Kernel

The synchronization in $k$ SpMVs is avoided by the parallel matrix powers kernel which trades communication with redundant computation. The key idea is to compute the dependency chain for each partition $q$ in order to compute the entries of the $k$th vector as shown by the bounded box in Figure 5.3(b)(ii). The vertices which are required from the neighbouring processors are fetched at once as shown by the white vertices in Figure 5.3(b)(ii). The redundant computation is then performed at each step (dotted white vertices) in order to avoid communication with neighbours. While this approach helps in avoiding communication, 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 $O(k^2b^2)$ [33].



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



Locally computable vertices
Vertices which communicate
Vertices with remote dependencies
Redundant vertices
i) k SpMVs
ii) Parallel Matrix Powers
Block 1
Block 2
Block 3
$A^3x^{(0)}$
$A^2x^{(0)}$
$Ax^{(0)}$
$x^{(0)}$
k
$b_R$
n

(b) $k$ SpMVs vs. Matrix Powers Kernel

Figure 5.3: $k$ 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.

## 5.2 Related Work

The communication problem in scientific computations is connected to the *memory wall* problem [41]. It is a well-known idea to formulate algorithmic innovations which hide memory latency and optimize memory bandwidth [90] [86]. For iterative solvers, Demmel *et al.* [33] trade communication with redundant computation by replacing $k$ SpMVs with the matrix powers kernel. They show that such an approach can minimize latency in a grid [33] and both latency and bandwidth on a multi-core CPUs [65] to give up to 4× and 4.3× speedup respectively over $k$ SpMVs for banded matrices. Although we see significant performance improvement on multi-cores for banded matrices in [65], the maximum efficiency is less than 5% of the peak floating-point performance as parallel matrix powers kernel algorithm limits the maximum value of $k$ and hence the maximum throughput due to $O(k^2b^2)$ growth in redundant flops.

In this work, we address the communication problem in accelerators like FPGA and GPU. Although the GPU has a high global memory bandwidth (see Table 3.1), for communication-bound problems like iterative solvers, the efficiency of GPU is still very low (7% of peak single-precision performance). Using the parallel matrix powers kernel, we avoid communication between different streaming multi-processors (SMs) and by exploiting inverse memory hierarchy [96] we avoid communication within the memory hierarchy of a single SM. In this way, we achieve higher efficiency as compared to $k$ SpMVs. However, we see similar restrictions on $k$ beyond which the computation cost dominates overall runtime due to redundant flops.

In FPGA-based iterative solvers, if the matrix is stored in an off-chip memory then the efficiency is even worse (4% of peak single-precision performance) purely because of difference in off-chip memory bandwidth of GPU and FPGA. Much of the research in FPGA community is focused on maximizing the use of on-chip memory (BRAMs) to load the largest possible matrix at once in order to avoid off-chip memory access for $k$ SpMVs. DeLorimier *et al.* [31] presented a general architecture for SpMV on FPGAs for large arbitrary sparse matrices. A GraphStep framework [32] is also proposed to map SpMV graph on FPGAs by exploiting high on-chip memory bandwidth to perform computation in a data parallel fashion. As on-chip BRAMs are limited on a particular FPGA, large matrices require multiple FPGAs which lead to communication problem as after each iteration, results need to be exchanged from processing elements located in different FPGAs. Boland *et al.* [20] proposed an Integer Linear Programming (ILP) framework to optimally utilize the on-chip BRAMs to buffer the largest possible symmetric banded matrix on a single FPGA and re-use it for all iterations. However, their approach is restricted to small matrix sizes ($n = 8$k with band size $b = 20$ on Virtex6-SX475T). In all cases, if the matrix is large enough and does not fit on-chip, the performance is bounded by the off-chip memory bandwidth of the FPGA. In this work, we present a hybrid matrix powers kernel specifically for FPGAs for such matrices. Knowing that FPGAs are rich in on-chip communication between floating-point cores, we minimize redundant flops at the cost of synchronization between floating-point cores at each step. We show that a relatively large value of $k$ is possible using this tight coupling between algorithm and architecture and this can lead to significant performance improvement over GPU.

We perform a quantitative comparison of these parallel architectures and show which architecture is better for different problem and band sizes. We also highlight the performance limiting factors and give insight into architectural improvements for next generation of these parallel architectures.

## 5.3 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 get 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 5.6.1) and also make architectural projections to get a desired performance with future devices (see Section 5.7).

### 5.3.1 GPU Architecture

We select the Nvidia GF100 variant C2050 GPU which is intended for high-performance numerical computing [30] [96]. A simplified architectural description of the GPU is shown in Figure 5.4 highlighting memory hierarchy as well as capacity, bandwidth and latency of each memory.

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 SM without any communication with other SMs during a single parallel task, *e.g.* each SM can

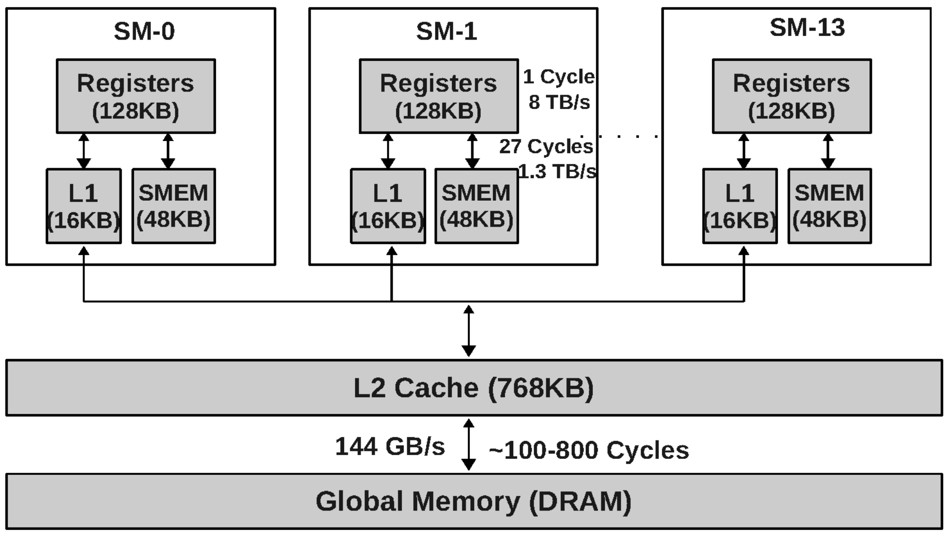


Figure 5.4: GPU Architecture (Nvidia C2050 Fermi).

compute $k$ SpMVs for a single block as shown in Figure 5.3(b)(ii) for $q = 2$.

### 5.3.2 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 $(b_R + k(b-1))\times b$ where $b_R\times b$ is the number of rows in each block and $k(b-1))\times b$ entries are fetched from neighbouring blocks. The entries fetched from neighbours are then used for redundant computation which 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

$$b_R = N_T - k(b-1) \tag{5.2}$$

$$N_q = \left\lceil \frac{n + b_R - 1}{b_R} \right\rceil \tag{5.3}$$

### 5.3.3 GPU Optimizations

In order to exploit memory hierarchy of the GPU for fast access of the matrix and vector partitions, we explore three possible mappings of matrix powers kernel as shown in Table 5.1. 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 [17] as baseline to perform $k$ SpMVs with a performance of 34.8 GFLOPs. We now briefly discuss the three possible GPU optimizations.

Table 5.1: Single-Precision Parallel Matrix Powers Kernel Parallel Mapping on C2050 GPU ($n$ =1M, $b$ =9, $k$ =8).

| | **Global Memory** | **Shared Memory** | **Reg.** | **GFLOPs** | **Efficiency** |
|---|---|---|---|---|---|
| $k$ SpMVs [17] | $A$, $x^{(i)}$ | | | 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% |

### 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. The entries from neighbouring blocks are pre-fetched in order to avoid communication with other SMs. 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.

### 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 5.1). 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.

### Thread Blocking + Register Blocking (123 GFLOPs)

GPUs have an inverse memory hierarchy [96], *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 [9]. In the matrix powers kernel, each matrix partition can be blocked within the registers of SM (see Listing 5.2). 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 5.5.

The matrix powers kernel with thread blocking and register blocking gives higher throughput

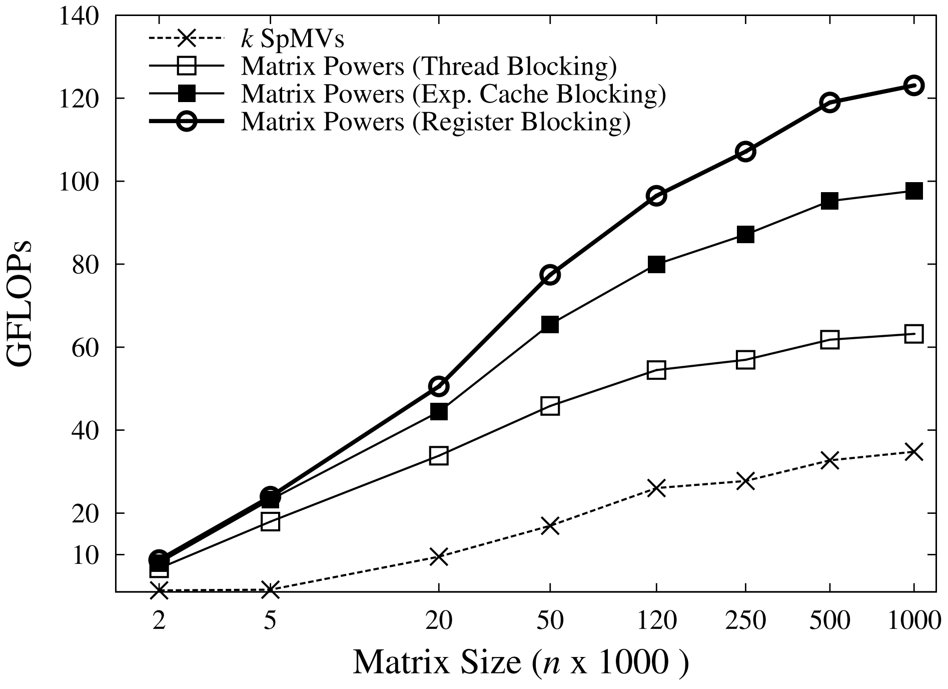


Figure 5.5: GPU Optimizations ($b = 9$, $N_T = 512$, $k = 8$).

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 5.7).

### 5.3.4 Kernels

We show CUDA code for variants of the matrix powers kernel in Listing 5.1− 5.2. For the parallel matrix powers kernel graph $G$ shown in Figure 5.3, all partitions $q = 1, 2, \ldots, N_q$ run the same kernel. The computation comprises two loops, an outer loop moving through $k$ levels in the graph and an inner loop performing dot product operation at each vertex using serial reduction. The only difference in all these kernels is the memory layout for storing matrix and vector partitions as highlighted in Listings 5.1− 5.2.

### 5.3.5 Modelling Performance

Mapping matrix powers kernel on GPU and selecting optimal $k$ to trade communication with computation is not obvious due to the complex GPU architecture. We characterize our discussion using bandwidth and latency of entire GF100 memory hierarchy in order to understand and predict the performance of matrix powers kernel on the GPU. 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 [28].

Listing 5.1: Matrix Powers Kernel with cache-blocking.

```
{
 int tid = blockIdx.x*bR + threadIdx.x;
 __shared__ float A_block_sh[b*NT];
 float x_block_reg[b];
  __shared__ float x_sh[NT];
 ..........................
 load_A_sh(A,A_block_sh,tid,N,k);
 load_x_sh(x,x_sh,tid,N,k);
 __syncthreads();
 int row = tid - k*(b-1)/2;
 for(int i =0;i<k;i++)
 {
  load_x_reg(x_sh,x_block_reg);
  sum=0;
  #pragma unroll
  for(int j =0;j<b;j++)
  {
   sum+=A_block_sh[threadIdx.x+b*j]*x_block_reg[j];
  }
 ..........................
 y[index+row] = sum;
 x_sh[threadIdx.x] = sum;
 __syncthreads();
 index += N;
 }
}
```

Listing 5.2: Matrix Powers Kernel with register-blocking.

```
{
 int tid = blockIdx.x*bR + threadIdx.x;
 float A_block_reg[b*NT];
 float x_block_reg[b];
 __shared__ float x_sh[NT];
 ..........................
 load_A_reg(A,A_block_reg,tid,N,k);
 load_x_sh(x,x_sh,tid,N,k);
 __syncthreads();
 int row = tid - k*(b-1)/2;
 for(int i =0;i<k;i++)
 {
  load_x_reg(x_sh,x_block_reg);
  sum=0;
  #pragma unroll
  for(int j =0;j<b;j++)
  {
   sum+=A_block_reg[j]*x_block_reg[j];

  }
 ..........................
 y[index+row] = sum;
 x_sh[threadIdx.x] = sum;
 __syncthreads();
 index += N;
 }
}
```

The global and shared memory latency models (in cycles) are shown as

$$l_{glb} = \#msg \times \alpha_{glb} + msize \times \beta_{glb} + flops \times \gamma. \quad (5.4)$$

$$l_{sh} = \#msg \times \alpha_{sh} + msize \times \beta_{sh} + nsync \times \alpha_{sync} + flops \times \gamma. \quad (5.5)$$

Like the LogP model, our models comprise three parameters $\alpha$, $\beta$ and $\gamma$. Overall runtime is the sum of three factors, memory latency ($\alpha_{glb}$ or $\alpha_{sh}$), inverse memory bandwidth ($\beta_{glb}$ or $\beta_{sh}$) and time per flop ($\gamma$). 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 [9] and summarize them in Table 5.2.

We estimate the performance of the matrix powers kernel shown in Listing 5.2. Each thread within the thread block executes three phases.

1. *Load*: Load partition $N_T \times b$ of matrix $A$ with each thread loading a single row of length $b$ in its registers. Load partition $N_T \times 1$ of vector $x^{(0)}$ in shared-memory with each thread loading only a single element. We show the total cycles required in loading the partition from the global memory in the first row of Table 5.3. In order to give intuition behind the analytical expression, $\alpha_{glb}$ is the latency of accessing the global memory, $N_T \times b \times 4$ is the total number of bytes to be loaded, $\beta_{glb}/P_e$ is the inverse global memory bandwidth allocated to a single SM, $Freq$ is the operating frequency (1.15 GHz) of the SM. In this way, we calculate the total number of cycles required to fetch the partition from the global memory into the register file.

2. *Compute*: Load $b$ components of $x^{(i-1)}$ parition from shared-memory in registers. Perform

dot product operation on vectors of length $b$ involving a row of matrix $A$ and entries of vector $x^{(i-1)}$ loaded in registers.

3. *Store*: Store the result as an element of $x^{(i)}$ back in shared memory by overwriting the location of $x^{(i-1)}$. Store the same result as an element of new vector $x^{(i)}$ in global memory. Repeat *Compute* and *Store* phases for $k$ steps.

Table 5.2: Model Parameters for GPU Performance.

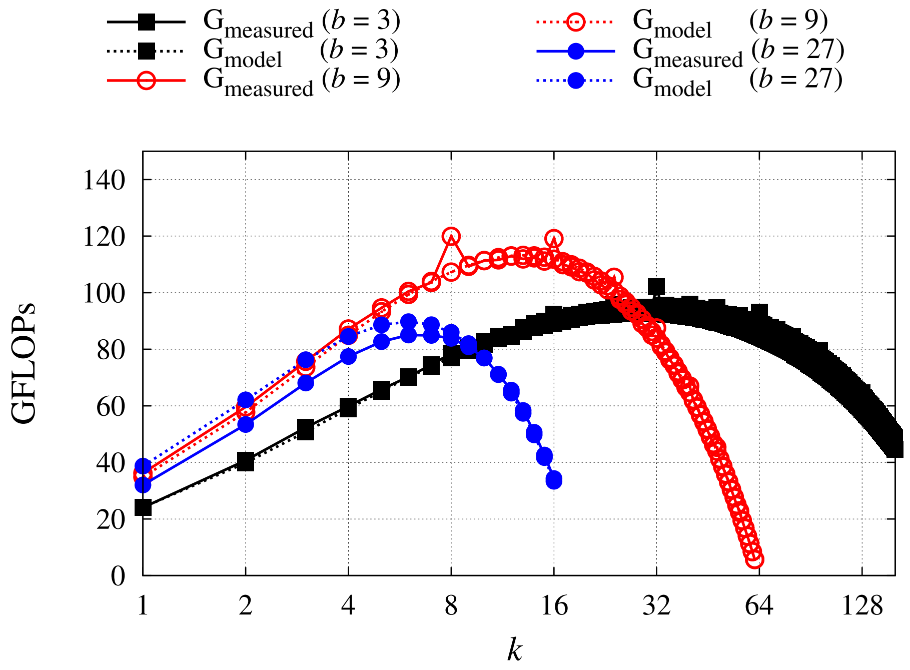
| | **C2050** | **C2075** | **K20** |
|---|---|---|---|
| **Specifications** | | | |
| Peak TFLOPs (single-precision) | 1.03 | 1.03 | 3.95 |
| Global memory clock (GHz) | 3.0 | 3.0 | 5.2 |
| Global memory bandwidth (GB/s) | 144 | 144 | 250 |
| Core clock rate ($Freq$) GHz | 1.15 | 1.15 | 0.732 |
| Number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

We show the communication and computation estimates of the matrix powers kernel for a single thread block mapped on a single SM in Table 5.3. We use inverse bandwidth per SM ($\beta_{glb}/P_e$, $\beta_{sh}/P_e$) for our estimates. We count floating-point multiply-add a single $\gamma$ as GF100 has a dual-issue pipeline. We build separate models for global and shared memory accesses

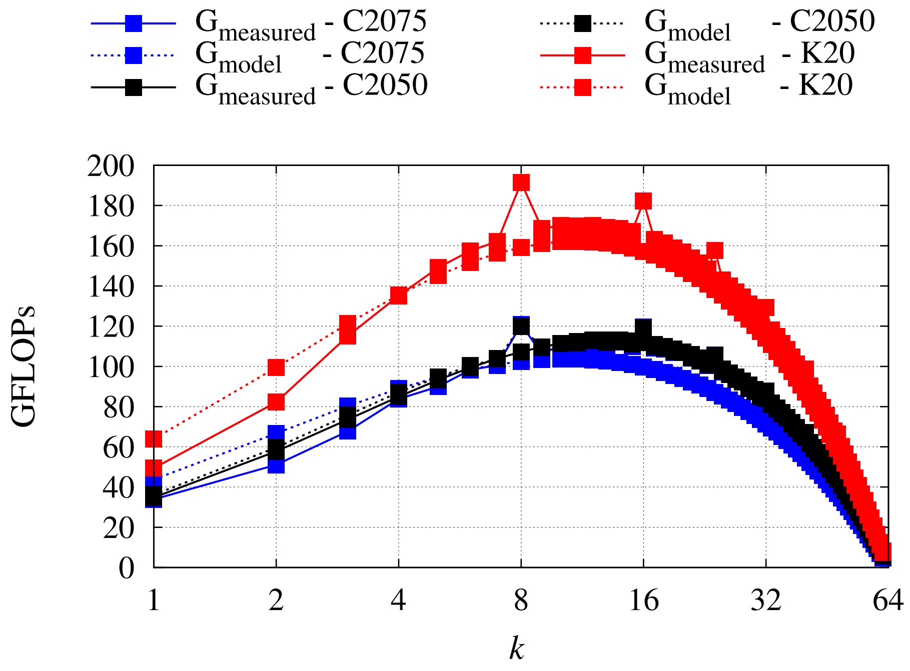
Table 5.3: GPU Analytical Model for Single-Precision Matrix Powers Performance.

| **Phase** | **Latency** |
|---|---|
| Load | $lA_{glb2reg} = \alpha_{glb} + \frac{b \times N_T \times Freq \times 4}{\beta_{glb}/P_e}$ <br> $lx_{glb2sh} = \alpha_{glb} + \frac{N_T \times Freq \times 4}{\beta_{glb}/P_e} + \frac{N_T \times Freq \times 4}{\beta_{sh}/P_e} + \alpha_{sync}$ |
| Compute | $lx_{sh2reg} = \alpha_{sh} + \frac{b \times N_T \times Freq \times 4}{\beta_{sh}/P_e}$ <br> $l_{compute} = b \times \gamma$ <br> $l_{condition} = 9\gamma$ |
| Store | $lx_{reg2sh} = \frac{N_T \times Freq \times 4}{\beta_{sh}/P_e} + \alpha_{sync}$ <br> $lx_{reg2glb} = \frac{N_T \times Freq \times 4}{\beta_{glb}/P_e}$ |
| Matrix Powers | $L_q = lA_{glb2reg} + lx_{glb2sh} + k(lx_{sh2reg} + l_{compute} + l_{condition} + lx_{reg2sh})$ |

and then combine them to find out the latency $L_q$ of a single thread block. We calculate total cycles by plugging in the parameters from Table 5.2 and then find out overall runtime $L$ by taking into account total number of thread blocks ($N_q$), number of SMs ($P_e$) and the number of thread blocks concurrently running per SM (we obtain this information using CUDA Visual Profiler [70]). We compare the predicted performance with the actual measured results for a range of band sizes on C2050 GPU as shown in Figure 5.6(a).



(a) Performance on C2050 for varying band sizes, $b$ = 3 ($\epsilon_\mu$= 5.4%, $\epsilon_\sigma$= 3.7%), $b$ = 9 ($\epsilon_\mu$= 1.5%, $\epsilon_\sigma$= 1.6%), $b$ = 27 ($\epsilon_\mu$= 5.6%, $\epsilon_\sigma$= 6.1%)



(b) Performance on different GPU architectures for $b$ = 9, C2050 ($\epsilon_\mu$= 1.5%, $\epsilon_\sigma$= 1.6%), C2075 ($\epsilon_\mu$ = 13.8%, $\epsilon_\sigma$ = 5.3%) , and K20 ($\epsilon_\mu$ = 7.5%, $\epsilon_\sigma$ = 4.2%)

Figure 5.6: Matrix powers kernel performance (GFLOPs vs. $k$), modelled vs. measured for $n$ = 1M.

In order to indicate the accuracy of our model, we also show the mean ($\epsilon_\mu$) and standard deviation ($\epsilon_\sigma$) 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. To ensure the general applicability of the performance model, we apply it on other GPUs as well like the GF100 variant C2075 GPU and the newer Kepler based K20 GPU as shown in Figure 5.6(b).

### 5.3.6 Performance Optimization

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

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

subject to

$$k \leq 2\frac{b_R}{b-1} \tag{5.6}$$

Referring to (5.6), the constraint ensures only nearest neighbour communication in the matrix powers kernel as shown in Figure 5.3. We carry out sensitivity analysis of GPU performance with respect to the algorithmic parameters with constant architectural parameters in Section 5.6.1. We also highlight in Section 5.7 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.

## 5.4 Mapping Matrix Powers Kernel to 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. Table 3.1 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 3.1, 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 proposed hybrid matrix powers kernel which exploits these 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.

### 5.4.1 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 13. The outer loop is a sequential algorithm which loads the blocks of matrix $A$ such that the block

**Algorithm 13** 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
```

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 5.3(b)(i). The working of the algorithm is shown in Figure 5.7 with a toy example, where we have 2 outer blocks which are loaded sequentially from the slow memory into FPGA on-chip memory. Each outer block is further partitioned into two

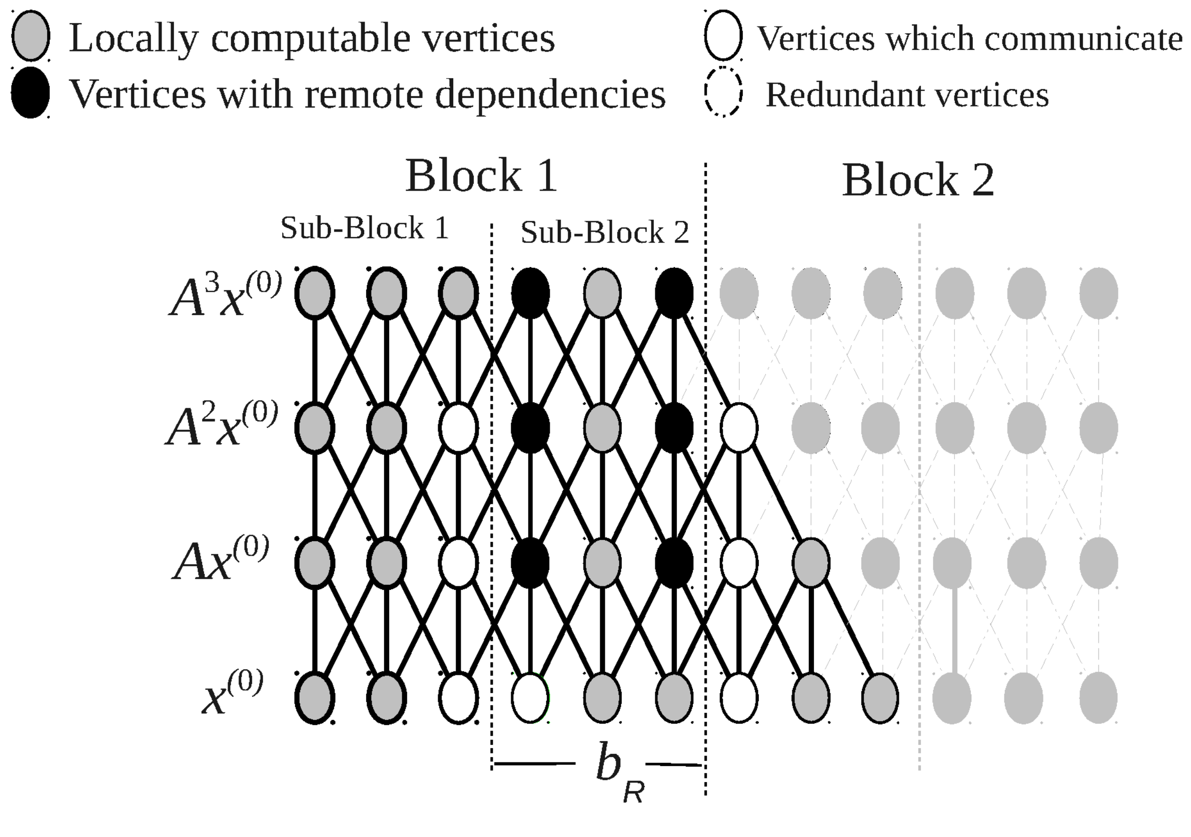


Figure 5.7: Proposed hybrid matrix powers graph for $n = 12$, $k = 3$, $b = 3$ and number of blocks $N_q = 2$.

sub-blocks which can be processed in parallel by an array of processing elements (PEs) working in a SIMD fashion as shown in Figure 5.8.

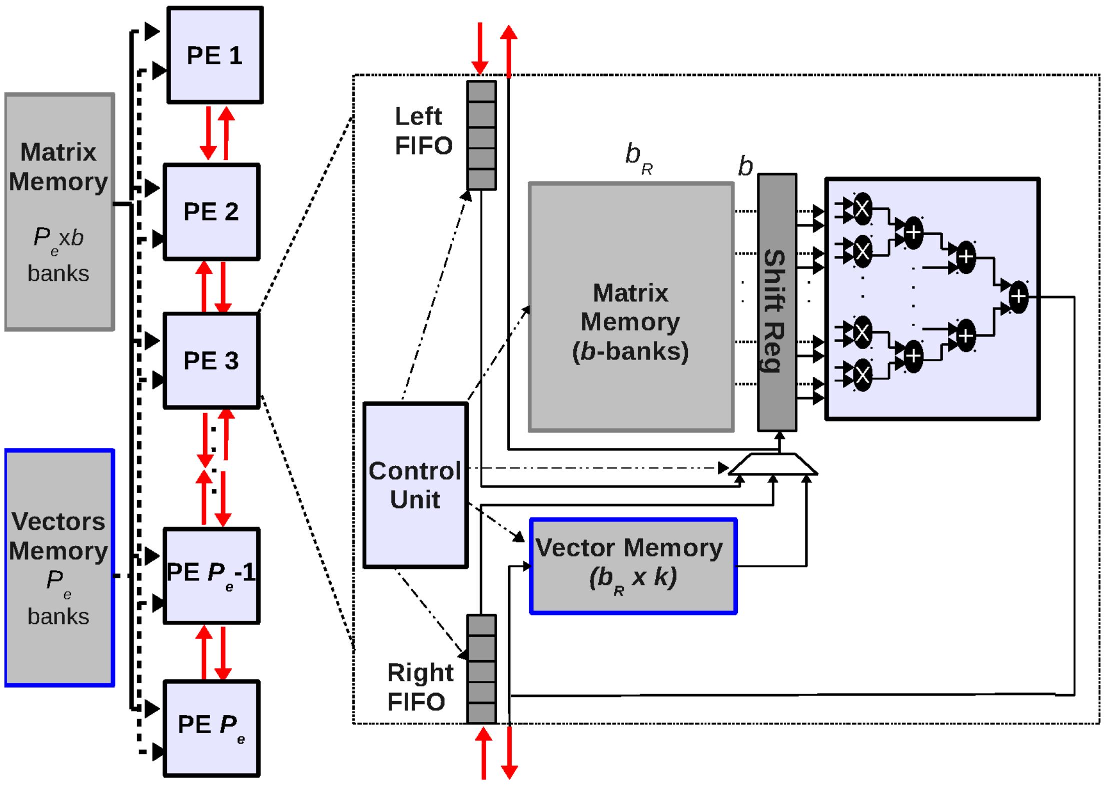


Figure 5.8: FPGA Data-path for Matrix Powers Kernel.

All the vertices inside a sub-block are computed in a pipelined fashion using a reduction circuit within the PE. After each level in the graph, PEs need to communicate dependencies 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 5.4 for the same matrix which we use for demonstrating GPU optimizations in Table 5.1. We observe that if the partitions of $A$ are stored 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 5.6 that for a range of matrix and band sizes, the FPGA even outperforms GPU because of the higher values of $k$.

Table 5.4: 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% |

### 5.4.2 Custom Hardware Design

We organize the processing elements (PEs) as a linear array which work in a SIMD fashion to perform $k$ matrix-vector multiplications for sub-blocks in parallel. The distributed on-chip memory of the FPGA provides the necessary bandwidth to saturate these PEs. Unlike the GPU, where the SMs need to perform redundant computation in order to avoid communication through high-latency global shared memory, the PEs within FPGA avoid redundant computation by exchanging dependencies using low-latency FIFOs.

#### Data-path

Each vertex in the graph involves a dot product operation on vectors of length $b$ (see Figure 5.7). In order to get high throughput, we use a fully-parallel multiplier array followed by an adder reduction-tree as shown in Figure 5.8. Although the design has high throughput, it has considerable latency as well due to deeply pipelined floating-point adders and multipliers [2]. We exploit the pipeline depth to feed a new dot product operation in each clock cycle in order to compute the next vertex. In this way, we keep the data-path busy all the time. The latency of the data-path for computing all the vertices for a single sub-block at any given level is given by

$$l_{compute} \quad = \quad b + b_R - 1 + \gamma_A \lceil \log_2 b \rceil + \gamma_M \tag{5.7}$$

Referring to Equation (5.7), $b$ cycles are required to load the shift register before the computation begins, $b_R - 1$ cycles are used to compute all the remaining vertices within the sub-block except the first one which takes $\gamma_A \lceil \log_2 b \rceil + \gamma_M$ cycles. Here, $\gamma_A$ and $\gamma_M$ are the latencies of single-precision floating-point adder and multiplier respectively and $\lceil \log_2 b \rceil$ is the depth of the adder reduction-tree.

#### Memory Subsystem

The input matrix $A$ is partitioned into $N_q$ blocks such that each block fits in the on-chip memory. Each block is further partitioned into sub-blocks. Each sub-block is stored as a $b$-bank *Matrix Memory* of depth $b_R$ and provides the necessary bandwidth to saturate the PE. *Vector Memory* is used to store $x^{(i)}$ to $x^{(i+k)}$ for that particular block. There are two left and right FIFOs which are used to receive remote data for computing vertices with remote dependencies.

The shift register is used to hold $b$ components of vectors $x^{(i)}$ at any given instant to compute dot product on each vertex. The memory required by each PE in terms of BRAMs (FPGA on-chip memories each 18kbit) is given by

$$\textit{Matrix Memory} = b\left\lceil \frac{32b_R}{18 \times 1024} \right\rceil \tag{5.8}$$

$$\textit{Vector Memory} = \left\lceil \frac{32b_R k}{18 \times 1024} \right\rceil \tag{5.9}$$

$$\textit{FIFOs} = 2\left\lceil \frac{32(b-1)}{2 \times 18 \times 1024} \right\rceil \tag{5.10}$$

**Control Unit**

Each PE has its own light-weight control unit which performs address generation for the *Matrix Memory* and the *Vector Memory*. It controls the read and write from the left and right FIFO as well as it manages the loading of shift register in order to compute local vertices or vertices with remote dependencies.

### 5.4.3 Modelling Performance

In order to understand the performance of matrix powers kernel on FPGA, we propose an analytical model for FPGA which comprises both computation as well as communication cost similar to GPU shown in Section 5.3.5. We show the parameters of the model in Table 5.5.

Table 5.5: Model Parameters for FPGA.

| **Parameters** | **Virtex6-SX475T** |
|---|---|
| Global memory latency ($\alpha_{glb}$) cycles | 6 |
| Global memory inverse bandwidth ($\beta_{glb}$) s/GB | $\frac{1}{34}$ [87] |
| FP Add latency ($\gamma_A$) cycles | 11 [2] |
| FP Mult latency ($\gamma_M$) cycles | 8 [2] |
| FP Operating Frequency ($Freq$) MHz | 258 |
| No. of FP Adders | $P_e(b-1)$ |
| No. of FP Multipliers | $P_e b$ |

The model is exact due to the highly predictive nature of FPGAs as a computing platform. We validate our model (Equation 5.7) using measurements from Modelsim in Figure 5.9. There is six cycles difference between the modelled and the measured cycles and these cycles are actually taken by the finite state machine within the control unit of our architecture.

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

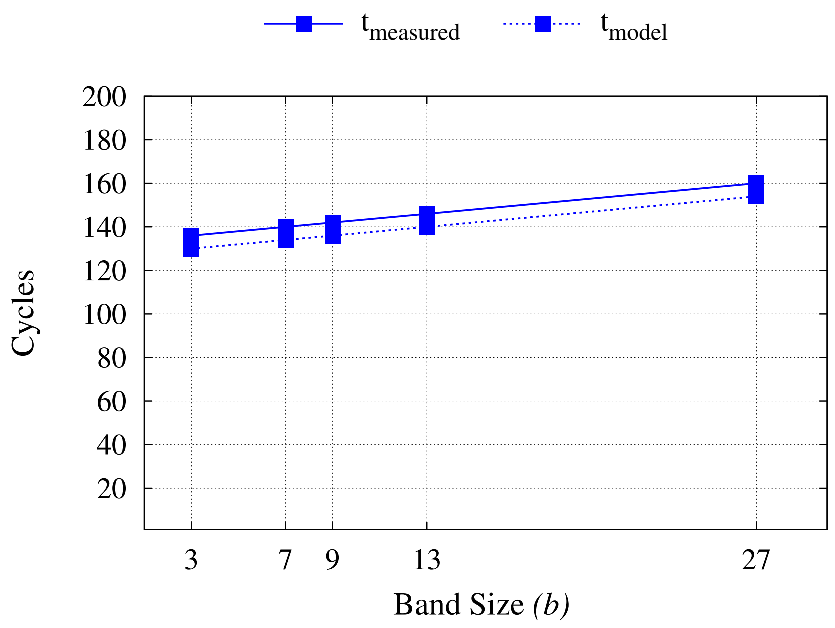


Figure 5.9: FPGA performance, modelled vs. measured (Modelsim), here $b_R$ is equal to 128.

1. *Load*: Load partition $b_R \times b$ of matrix $A$ and $b_R$ components of the vector $x^{(i)}$ for each PE.
2. *Compute*: Compute $b_R$ components of the vector $x^{(i+1)}$ in a pipelined fashion.
3. *Store*: Store the result back in the off-chip memory. Repeat *Compute* and *Store* phases for $k$ steps.

We show the detailed models for each of these three stages followed by the latency of a single block in Table 5.6.

Table 5.6: FPGA Analytical Model for Single-Precision Matrix Powers Performance.

| **Phase** | **Latency** |
|---|---|
| Load | $lA_{glb2local} = \alpha_{glb} + \frac{b \times P_e \times b_R \times Freq \times 4}{\beta_{glb}}$ <br> $lx_{glb2local} = \alpha_{glb} + \frac{P_e \times b_R \times Freq \times 4}{\beta_{glb}}$ |
| Compute | $l_{compute} = b + b_R - 1 + \gamma_A \lceil \log_2 b \rceil + \gamma_M)$ |
| Store | $lx_{local2glb} = \frac{P_e \times b_R \times Freq \times 4}{\beta_{glb}}$ |
| Matrix Powers | $L_q = lA_{glb2local} + lx_{glb2local} + k(l_{compute}$ <br> $+ lx_{local2glb})$ |

The latency $L_q$ of a single block is the summation of the latency of these three stages. The overall latency $L$ is then calculated by multiplying $L_q$ with total number of blocks $N_q$.

### 5.4.4 Resource-Constrained Methodology

Like GPU, the performance of the matrix powers kernel depends on the problem parameters $(n,b)$, the architectural parameters $(P_e,\gamma_A,\gamma_M,\alpha_{glb},\beta_{glb})$ 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 5.5). We calculate the memory bandwidth required for these $P_e$ PEs ($2b$ words per PE), partition the available on-chip memory in $b_R \times b$ blocks and assign these blocks to $P_e$ PEs. We solve the following constrained optimization problem to pick $k$ on a particular FPGA for a given problem.

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

subject to

$$\begin{aligned} R(P_e) &\leq FPGA_{Logic} \\ M(P_e, b_R, k) &\leq FPGA_{BRAMs} \\ k &\leq 2\frac{b_R}{b-1} \end{aligned} \tag{5.11}$$

Referring to Equation (5.11), our objective is to minimize the runtime based on constraints on FPGA resources. $M(P_e, k, b_R)$ is the number of BRAMs (FPGA on-chip memories) required and $R(P_e)$ is a vector containing the number of resources in terms of LUTs, FFs and DSP48Es that are used in floating-point adders and multipliers [2]. The last constraint ensures only nearest neighbour communication in the matrix powers kernel as shown in Figure 5.7.

## 5.5 Evaluation Methodology

We use the same generation (40nm) of GPU (NVidia C2050) and FPGA (Virtex6-SX475T) devices as mentioned in Table 3.1. We use CUDA 5.0 for compiling CUDA kernels and also use `cusp-v0.3.1` [17] which is a sparse library optimized for GPUs. The `spmv_dia_kernel` routine from this library is used for computing $k$ SpMVs and is used as a baseline for GPU. For FPGA, we implement a set of 3 PEs alongwith their memory subsystem and their FIFO interconnections. We use Xilinx IP Core [2] for BRAMs, adders and multipliers. We synthesize as well as place and route the circuit using Xilinx ISE 13.4. We actually measure the total number of DSP48Es, FFs and LUTs required for a single PE and then estimate the total number of PEs that can be synthesized on a given FPGA device. We actually measure the latency of computing a single block using a single PE and then estimate the total number of cycles required for the whole matrix powers kernel when all PEs are working in parallel. We use a range of randomly generated matrices with varying band sizes in order to evaluate the performance of matrix powers kernel on GPU and FPGA.

## 5.6 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.

### 5.6.1 Sensitivity to Algorithmic Parameters

We use the formulations in (5.6) and (5.11) 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 surface to volume ratio, the impact of the partition size $b_R$ is marginal compared to the unroll factor $k$ as shown in Figure 5.10.

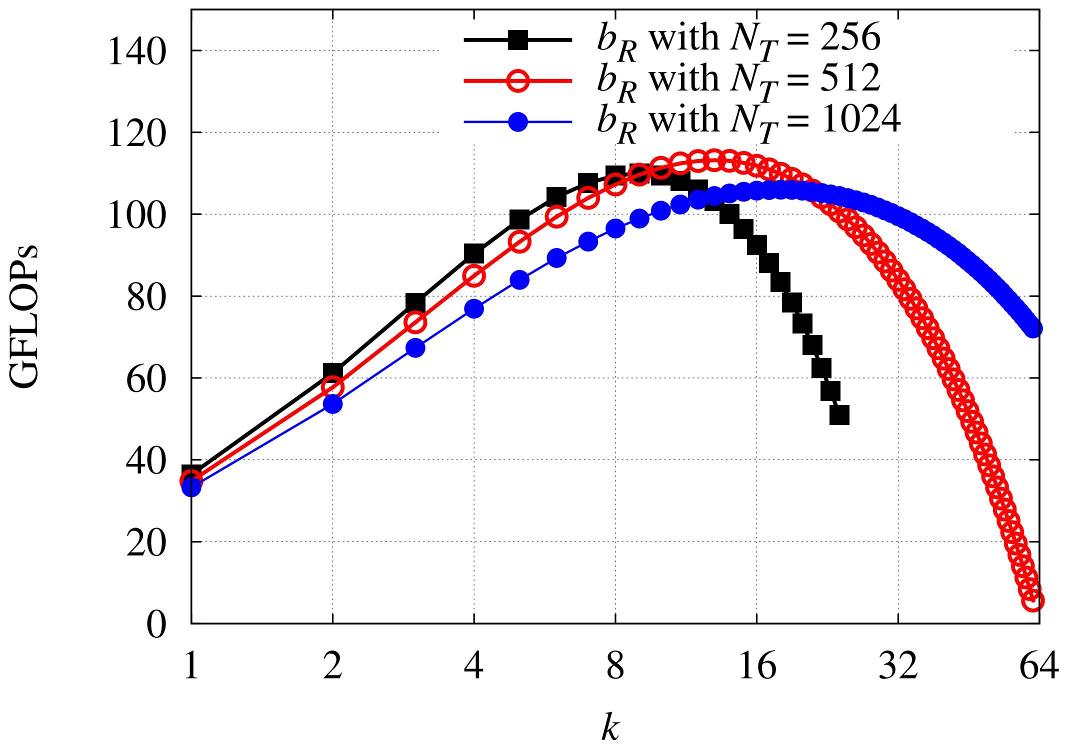
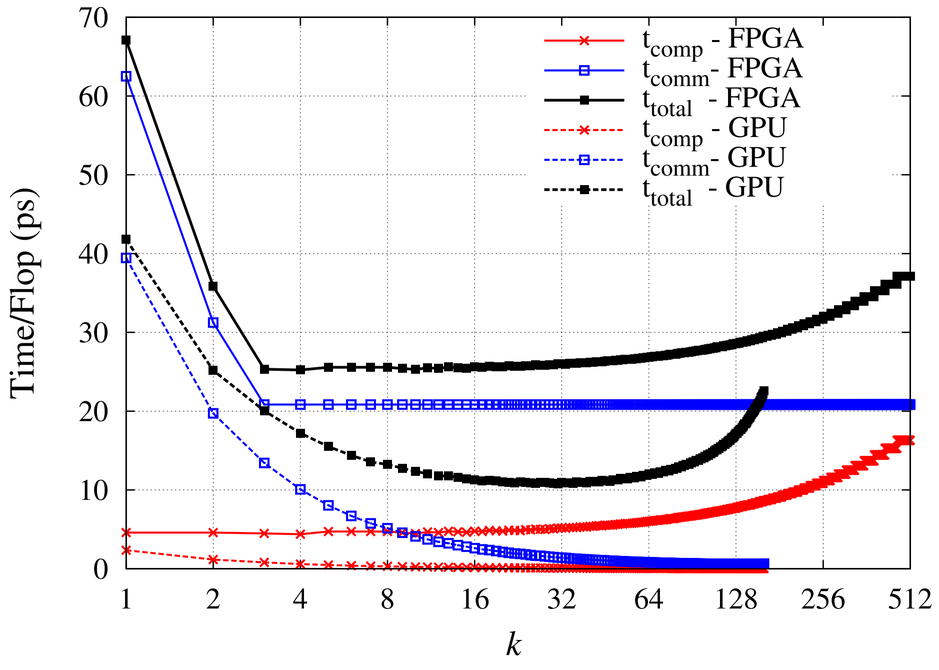


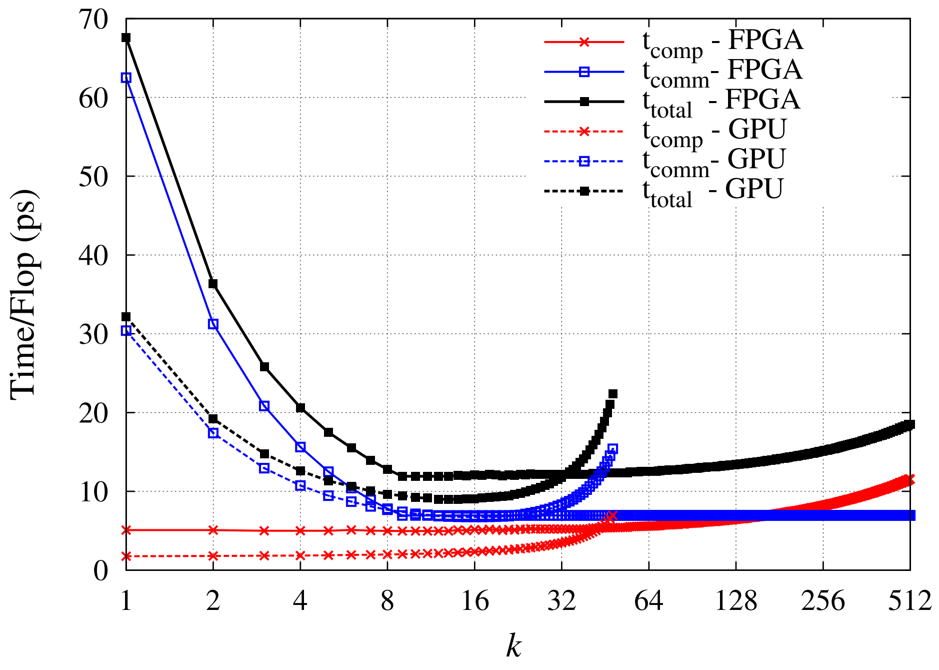
Figure 5.10: Algorithmic Sensitivity– GPU performance as a function of algorithmic parameter $k$ for different values of $b_R$ for a matrix with band size $b = 9$ and $n = 1$M.

In order to see how the performance of GPU and FPGA varies with $k$, we take a problem size with $n$ equal to $1 \times 10^6$ and show both the communication and computation costs for band size equal to 3 and 9 in Figure 5.11 and for band size equal to 27 in Figure 5.2. We observe that in case of GPU, the optimal value of $k$ decreases as we increase the band size whereas in case of FPGA, it shows the opposite trend. After a certain value of $k$, both computation and communication costs dominate on GPU. Using the performance model, we further explore what limits the maximum value of $k$ and hence the maximum performance. We separately measure both computation as well as communication costs. The communication cost involves both global as well as shared memory accesses. We use $\frac{time}{flop}$ as the metric to measure these cost factors and show them in Figure 5.12(a) and Figure 5.12(b) for band size equal to 9 and 27 respectively.

We observe that as we increase the value of $k$, the global communication cost decreases whereas the computation as well as the shared memory communication cost starts increasing until there is a point which minimizes the overall cost. The computation cost increases because of $O(k^2b^2)$ growth in redundant operations and shared memory communication increases as for each thread



(a) Time/Flop vs. $k$, $b$ = 3)

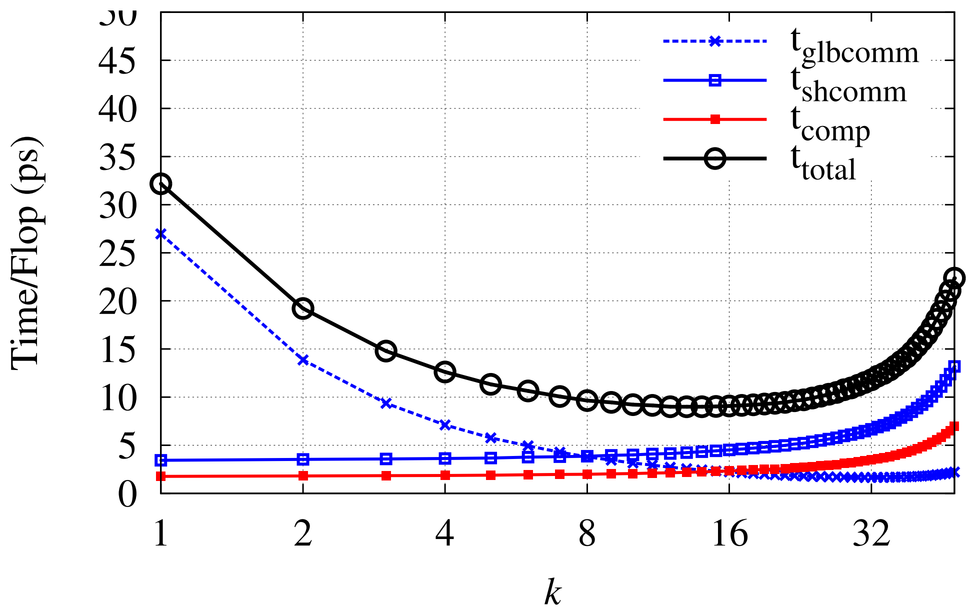


(b) Time/Flop vs. $k$, $b$ = 9 ($n$ =1M)

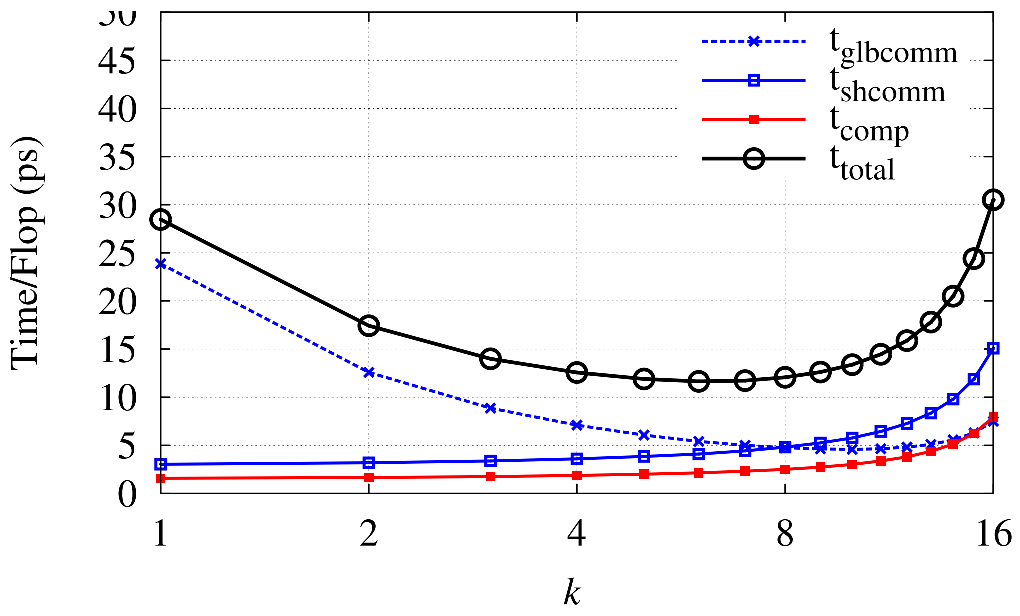
Figure 5.11: Algorithmic Sensitivity for FPGA and GPU ($n$ = 1M).

we need to load $b$ components of the vector from shared memory and we have to access them for $k$ iterations. Both of these costs dominate for large band sizes and hence we see smaller values of $k$ as shown in Figure 5.12(b).

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 computa-



(a) Time/Flop vs. $k$, $b = 9$ ($n = 1$M)



(b) Time/Flop vs. $k$, $b = 27$ ($n = 1$M)

Figure 5.12: Analyzing performance limiting factors of matrix powers kernel on C2050 GPU.

tion 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 5.2 and Figure 5.11. 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 5.13. 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 5.7.

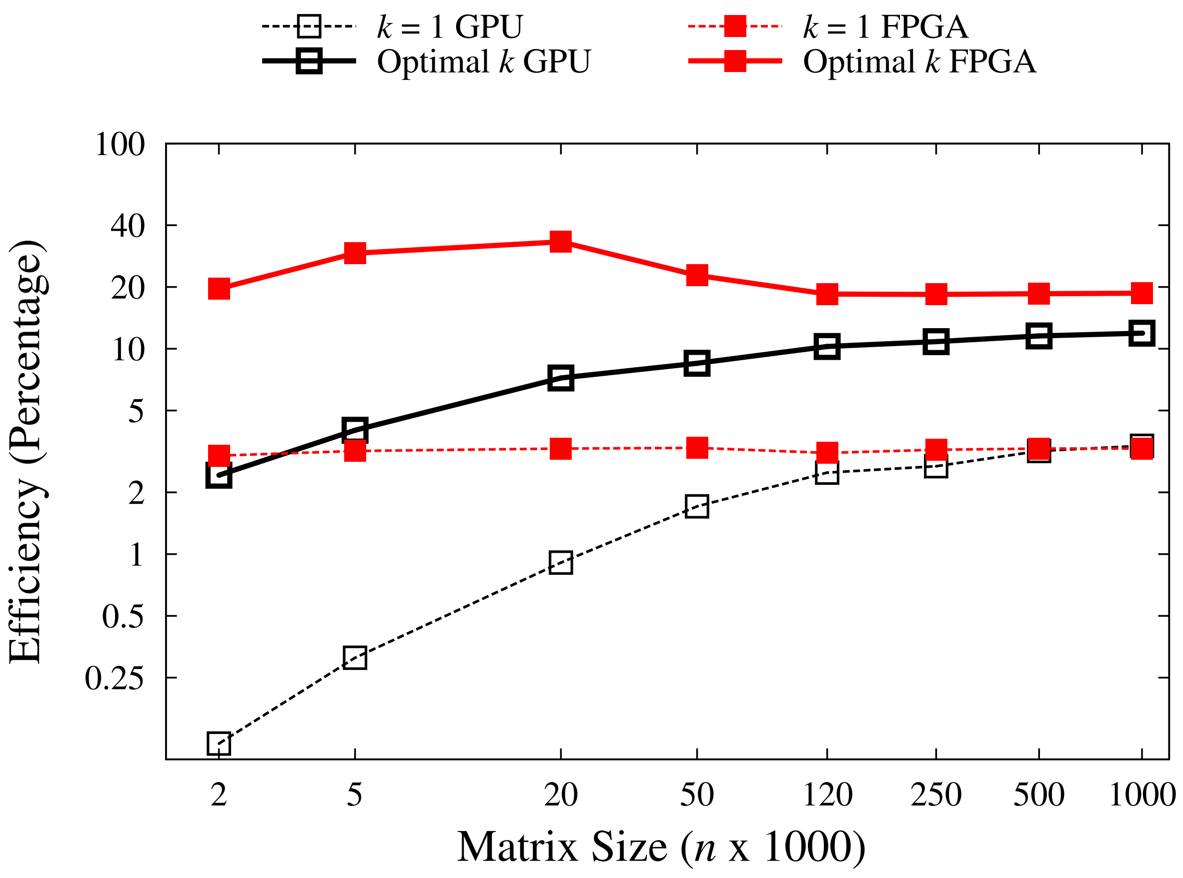


Figure 5.13: Efficiency of FPGA and GPU as a percentage of peak single-precision floating-point performance ($b = 9$).

Table 5.7: 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× |

### 5.6.2 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 5.14.

#### Small Problem Sizes ($n \leq$ 20K)

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, therefore, the problem remains compute-bound leading to high throughput. On the other hand, with GPU, the value of $k$ decreases with increasing band sizes due to increase in redundant operations. As a result, 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.

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

For large problem sizes and small band sizes, GPU performs slightly better than FPGA since the vectors 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 $\sim 3\times$ speedup.

**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$\times$ speedup over GPU.

## 5.7 Architectural Insight

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

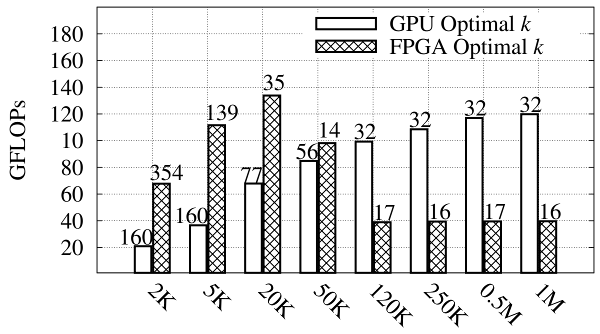
- How to optimally change the architectural parameters to get a desired performance?
- How we can achieve the same performance for a fixed architecture by only changing the algorithmic parameters?

### 5.7.1 Sensitivity to GPU Architectural Parameters

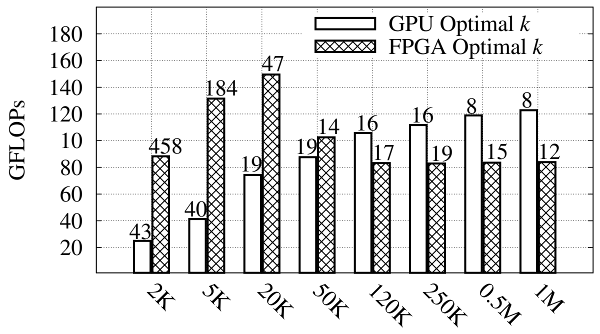
We solve the optimization problem in (5.6) using a steepest ascent method. The steepest ascent curve shows different points of performance enhancement and the corresponding architectural parameters as shown in Figure 5.15. We show that our optimization vector ($\alpha_{glb}$, $\beta_{glb}$,$\alpha_{sh}$,$\beta_{sh}$) for a hypothetical GPU should be scaled as ($\sim\frac{1}{10}$,$\sim$10, $\frac{1}{13}$,1.23) in order to get a 3.5$\times$ speedup over the same problem running on C2050 GPU. However, using our predictive model and measured results, we have already shown in Figure 5.6(a) that same performance can be obtained by careful selection of $k$ without changing the architecture.

### 5.7.2 Sensitivity to FPGA Architectural Parameters

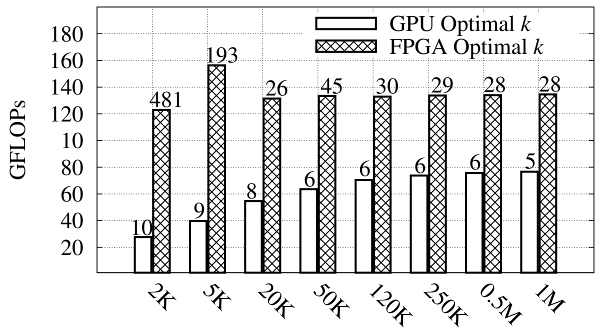
We show the sensitivity of the FPGA performance with respect to off-chip memory bandwidth in Figure 5.16 for two cases. Firstly, when we have fixed algorithm with $k$ =1 and second with optimal value of $k$. We observe that by carefully picking $k$ we can get a 5.6$\times$ performance for a Virtex6-SX475T FPGA. In order to achieve similar performance, $k$ =1 curve shows that off-chip memory bandwidth needs to be scaled by 8.6$\times$. 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.



(a) Band Size $b = 3$



(b) Band Size $b = 9$



(c) Band Size $b = 27$

Figure 5.14: Matrix Powers Performance Comparison vs. Matrix Size. The optimal values of $k$ are mentioned on the top of the bar for both FPGA and GPU.

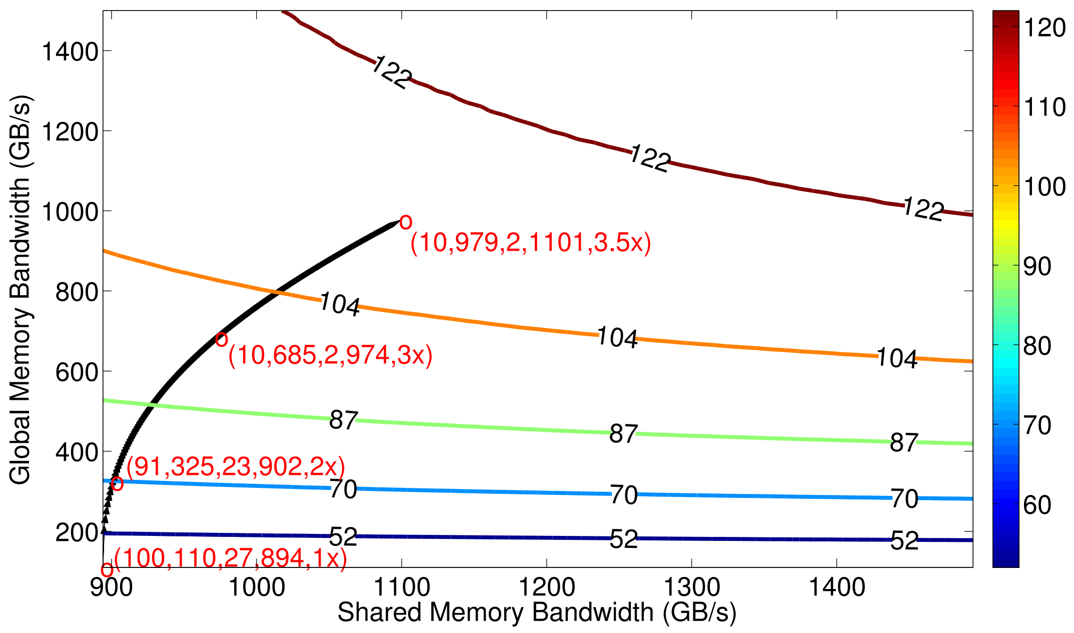


Figure 5.15: Architectural Sensitivity– GPU performance contours in GFLOPs as a function of global memory bandwidth ($\beta_{glb}$) and shared memory bandwidth ($\beta_{sh}$) for band size $b = 9$ and $n = 1$M. Specific points (in red) on the steepest ascent curve (in black) are shown representing ($\alpha_{glb}$,$\beta_{glb}$,$\alpha_{sh}$,$\beta_{sh}$,$\frac{L_{base}}{L_{pred}}$) where $L_{base}$ is the performance obtained on C2050 GPU. $\alpha_{glb}$ and $\alpha_{sh}$ are in cycles whereas $\beta_{glb}$ and $\beta_{sh}$ are in GB/s.

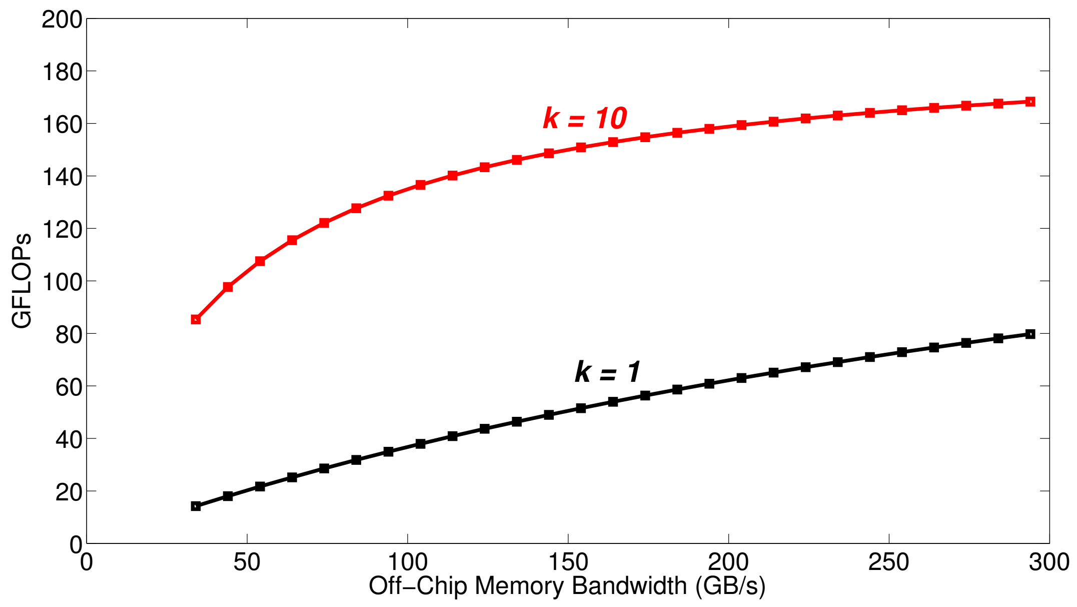


Figure 5.16: Architectural Sensitivity– FPGA performance in GFLOPs as a function of off-chip memory bandwidth ($\beta_{glb}$) for band size $b = 9$ and $n = 1$M. The starting point of the curves is a Virtex6-SX475T architecture with an off-chip bandwidth of 34 GB/s.

## 5.8 Summary

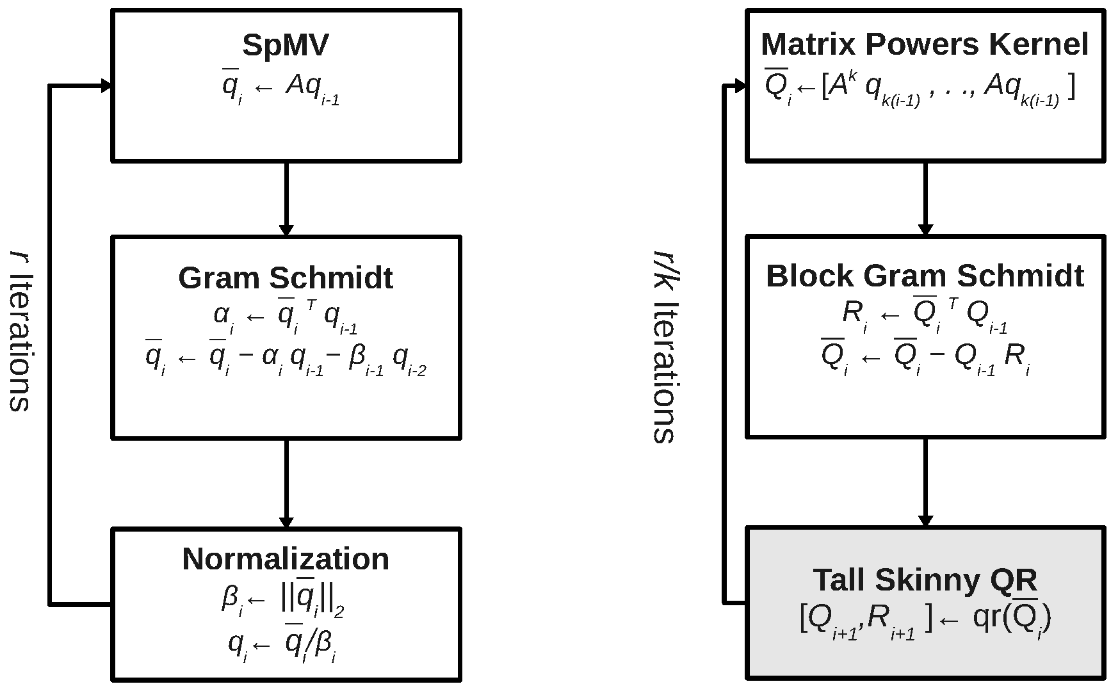
Trading communication with computation increases the silicon efficiency of hardware accelerators like FPGAs and GPU for accelerating communication-bound sparse iterative solver. Although unrolling $k$ iterations using the matrix powers kernel provides significant performance improvement compared to standard $k$ SpMVs on a GPU, the performance is constrained due to quadratic growth in redundant computations and increase in shared memory communication. 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.

# 6 Tall-Skinny QR Factorization

In the previous chapter, we introduce the matrix powers kernel where we fetch the matrix once and generate $k$ vectors in one shot. This provides a $k$ times reduction in the communication cost which in turn increases the silicon efficiency in accelerating the iterative numerical algorithms. However, this reduction in communication is only possible at the expense of redundant computation. One such computation is the QR factorization of a tall-skinny matrix used for orthogonalization of these $k$ vectors, a step necessary for the convergence of the communication-avoiding variant of the iterative numerical algorithm. QR factorization is a fundamental problem in linear algebra where an $m \times n$ matrix $B$ is factorized into an $m \times m$ orthogonal matrix $Q$ and an $m \times n$ upper triangular matrix $R$ [39]. Of particular interest is the QR factorization of tall-skinny matrices where $m \gg n$ and the aspect ratio can be 5 to 1, 100 to 1 or in some cases even 100,000 to 1. Matrices with such extreme aspect ratios not only arise in communication-avoiding iterative numerical algorithms as shown in Figure 6.1 but they also exist naturally in many practical applications of QR factorization. These include least squares data fitting [19], stationary video background subtraction [8] and block iterative methods [39] used for solving linear systems and eigenvalue problems. These applications demand high-performance tall-skinny QR factorization. Subtracting the stationary video background from a 10-second surveillance video, for example, requires over a teraflop of computation [8].

Traditionally, in high performance linear algebra libraries like LAPACK and Intel MKL, QR factorization is performed using Block Householder QR [83]. Block Householder QR comprises two kernels, a communication-bound *panel factorization* with dominant matrix-vector multiplications followed by a compute-bound *trailing matrix update* with dominant matrix-matrix multiplications. While most of the time is spent in trailing matrix update for square matrices, it is the sequential panel factorization which dominates in case of extremely tall-skinny matrices. Recently, Demmel *et al.* [33] proposed Tall-Skinny QR (TSQR), a communication-avoiding algorithm for tall-skinny matrices which parallelizes the panel factorization by decomposing it into tiles, performing local QR factorization on tiles using Householder QR and then merging the results (see Section 6.1.1). The key idea is to do less communication at the cost of more computations because of the ever increasing gap between advancements in compute (GFLOPs) and communication (bandwidth and latency) capabilities of modern architectures.

In this chapter, we present a custom architecture for TSQR which bridges the performance gap that still exists between the parallel potential available in TSQR and that which has been exploited by mapping on variety of parallel architectures like multi-cores [44], GPUs [8] and FPGAs [88]. In order to motivate, we show the performance scaling trends for QR factorization as a function of matrix width in Figure 6.2. We observe that as the matrix gets skinny, the proposed architecture clearly outperforms the existing implementations and the performance improvement is even more pronounced for extremely tall-skinny matrices as shown in Figure 6.3.


SpMV
$\overline{q}_i \leftarrow Aq_{i-1}$
Gram Schmidt
$\alpha_i \leftarrow \overline{q}_i^{\,T} q_{i-1}$
$\overline{q}_i \leftarrow \overline{q}_i - \alpha_i q_{i-1} - \beta_{i-1} q_{i-2}$
Normalization
$\beta_i \leftarrow \|\overline{q}_i\|_2$
$q_i \leftarrow \overline{q}_i/\beta_i$
r Iterations
Matrix Powers Kernel
$\overline{Q}_i \leftarrow [A^k q_{k(i-1)}, \ldots, Aq_{k(i-1)}]$
Block Gram Schmidt
$R_i \leftarrow \overline{Q}_i^{\,T} Q_{i-1}$
$\overline{Q}_i \leftarrow \overline{Q}_i - Q_{i-1} R_i$
Tall Skinny QR
$[Q_{i+1}, R_{i+1}] \leftarrow \mathrm{qr}(\overline{Q}_i)$
r/k Iterations

(a) Lanczos Iteration (b) CA-Lanczos

Figure 6.1: Lanczos Iteration vs. Communication-Avoiding Lanczos (CALanczos) with emphasis on QR factorization of a tall-skinny matrix ($A \in \mathbb{R}^{n1 \times n1}$, $\overline{Q_i} \in \mathbb{R}^{n1 \times k}$, $R_i \in \mathbb{R}^{k \times k}$, $q_i \in \mathbb{R}^{n1 \times 1}$, $n1 \gg k$)
.

Although TSQR is a dense problem suited to GPUs, the performance is limited due to memory-bound Basic Linear Algebra Subroutines (BLAS) Level 2 operations in local QR factorizations (see Section 6.1.2) and the communication latency of the global memory used in the merge stage. We show how we can exploit fine-grain parallelism in BLAS Level 2 operations by using high-bandwidth on-chip memories and deeply-pipelined floating-point cores (see Section 6.3.3). We additionally perform the merge stage entirely on-chip to avoid communication latency (see Section 6.3.4).

The key contribution of this chapter are thus:

- A high throughput deeply-pipelined architecture for Householder QR to perform local QR factorizations.
- Mapping of TSQR on the same architecture using pipeline parallelism and avoiding communication latency by merging the intermediate results entirely on-chip.
- Quantitative comparison with double-precision QR routines from optimized linear algebra libraries showing a 21.7× - 50.2× (31.5× geo.mean) over Intel MKL, 7.28× - 36.7× (16× geo.mean) over MAGMA [91] and 52× - 258× (117× geo.mean) over CULA [50].
- Quantitative comparison with a custom architecture for FPGA proposed in [88] and a highly optimized GPU-based implementation in [8] showing a speedup of 0.57× - 7.7× (2.2 × geo.mean) and 3.38× - 12.70× (6.47× geo.mean) respectively.

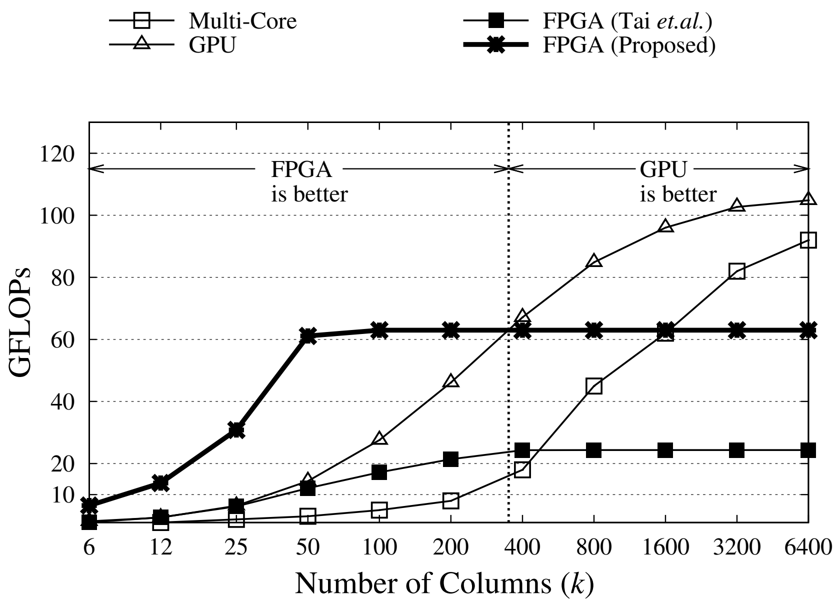


Figure 6.2: Performance Scaling Trends for double-precision QR Factorization (No. of rows $n$ = 6400).

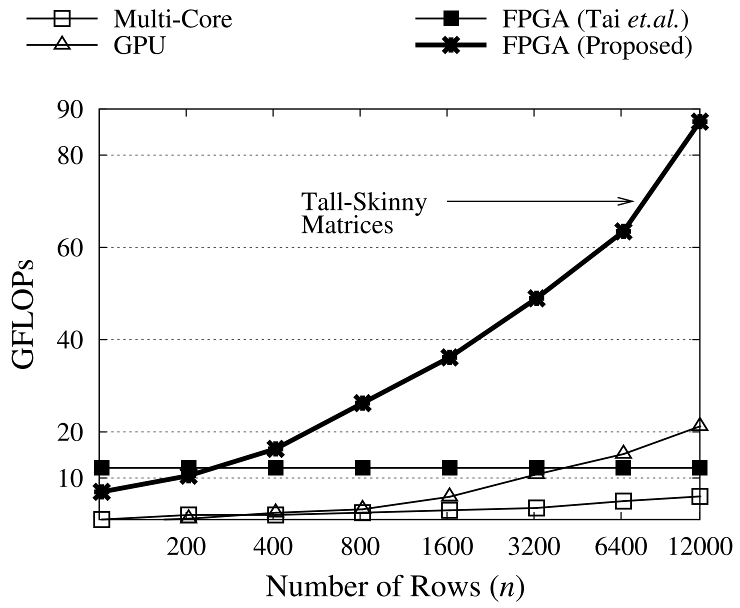


Figure 6.3: Performance Scaling Trends for double-precision QR Factorization (No. of columns $k$ = 51).

## 6.1 Background

### 6.1.1 Tall-Skinny QR

The Tall-Skinny QR (TSQR) factorizes the matrix in a divide-and-conquer fashion with optimal communication between processing elements [33]. It comprises a *local QR stage* followed by a *merge stage* as shown in Algorithm 14. In the local QR stage, the input matrix $\overline{Q_i} \in \mathbb{R}^{n \times k}$ is divided into small tiles each of size $b_R \times k$, where $b_R$ is the number of rows in the tile ($b_R = 2k$ for binary tree) and there are $N_{tiles} = \lceil \frac{n}{b_R} \rceil$ tiles in total. These tiles may then be factorized in parallel using various techniques such as Householder QR [39]. In the merge stage, the $R \in \mathbb{R}^{k \times k}$ factors of local QR factorizations are stacked and factorized in a tree fashion. We get a final $R$ factor and a series of small $Vs$ which, if needed, can be used to explicitly compute the orthogonal matrix $Q$.

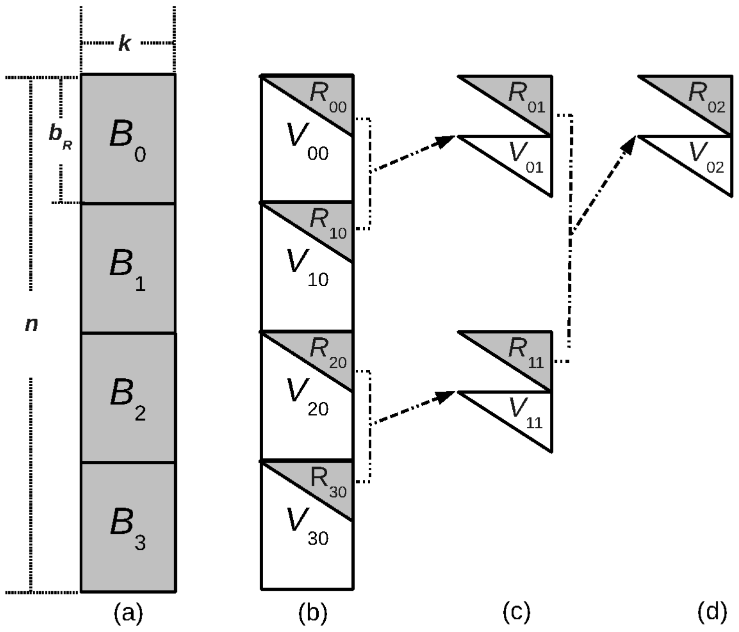


Figure 6.4: Tall-Skinny QR Facotorization [33], $B = \overline{Q_i}$, (a) local QR stage. (b, c & d) merge stage.

Let us see the case shown in Figure 6.4 where $N_{tiles} = 4$. In the first stage of local QR factorization, $B_0$, $B_1$, $B_2$ and $B_3$ are decomposed in parallel as shown in Figure 6.4(a).

$$\overline{Q_i} = B = \begin{bmatrix} B_0 \\ B_1 \\ B_2 \\ B_3 \end{bmatrix} = \begin{bmatrix} Q_{00}R_{00} \\ Q_{10}R_{10} \\ Q_{20}R_{20} \\ Q_{30}R_{30} \end{bmatrix} = \begin{bmatrix} Q_{00} & 0 & 0 & 0 \\ 0 & Q_{10} & 0 & 0 \\ 0 & 0 & Q_{20} & 0 \\ 0 & 0 & 0 & Q_{30} \end{bmatrix} \begin{bmatrix} R_{00} \\ R_{10} \\ R_{20} \\ R_{30} \end{bmatrix}$$

In the merge stage, the $R$ factors are stacked and local QR factorization is performed to generate the final $R$ factor in $\log_2 N_{tiles}$ steps as shown in Figure 6.4 (b), (c) and (d). Finally, the matrix $B$ is factorized as

$$B = \begin{bmatrix} Q_{00} & 0 & 0 & 0 \\ 0 & Q_{10} & 0 & 0 \\ 0 & 0 & Q_{20} & 0 \\ 0 & 0 & 0 & Q_{30} \end{bmatrix} \begin{bmatrix} Q_{01} & 0 \\ 0 & Q_{11} \end{bmatrix} \begin{bmatrix} Q_{02} \end{bmatrix} \begin{bmatrix} R_{02} \end{bmatrix}$$

The QR factorizations in local QR and merge stages can be performed by various methods including Modified Gram-Schmidt (MGS), Givens rotations, Cholesky QR and Householder

**Algorithm 14** Tall-Skinny QR

```
Require: A matrix $\overline{Q_i} \in \mathbb{R}^{n \times k}$ decomposed into $N_{tiles}$ tiles.
  – local QR stage –
  for $i = 0$ to $N_{tiles}$-1 do
    $B_i = \overline{Q_i}(i \times b_R : (i+1) \times b_R)$
    $[V_{i,0}, t_{i,0}\ R_{i,0}]$ := Householder QR($B_i$)
  end for

  – merge stage binary tree –
  for $j = 1$ to $\log_2(N_{tiles})$ do
    $p := 0$
    $q := 0$
    while $p < \frac{N_{tiles}}{2^{j-1}}$ do
      $[V_{q,j}, t_{q,j}\ R_{q,j}]$ := Householder QR($[R_{p,j-1}; R_{p+1,j-1}]$)
      $p := p + 2$
      $q := q + 1$
    end while
  end for
  return The upper triangular matrix $R \in \mathbb{R}^{k \times k}$ and intermediate $V$ matrices and t vectors
  which can be combined to find the final $Q$ matrix.
```

QR [39]. We pick Householder QR due to its high numerical stability [33].

### 6.1.2 Householder QR

In Householder QR, the matrix $B \in \mathbb{R}^{n \times k}$ is factorized as $R = Q_k \cdots Q_2 Q_1 B$ where $Q = Q_1^{-1} \cdots Q_{k-1}^{-1} Q_k^{-1}$ and $Q_i = \begin{pmatrix} I_{i-1} & 0 \\ 0 & H_i \end{pmatrix}$. The matrix $H_i$ is the Householder transformation matrix and is computed as $H_i = I_i - \tau_i v_i v_i^T$ where $v_i$ is called the *Householder reflector* for column $i$ having length $n-i-1$. The computational complexity of Householder QR is $O(nk^2)$ and is primarily dominated by BLAS Level 1 and 2 operations as shown in Algorithm 15.

## 6.2 Related Work

We survey recent work on tall-skinny QR factorization on parallel architectures like multi-cores [44], GPUs [8] and FPGAs [88]. While the performance of multi-cores is good for square matrices (90 GFLOPs with 58.8% efficiency), it decreases significantly for tall-skinny matrices (2 GFLOPs with 1.2% efficiency). This is because of the dominant matrix-vector multiplications in Householder QR (inner loop of Algorithm 15) which are less efficient on multi-cores due to low memory bandwidth. We observe 7× performance improvement with GPU for tall-skinny matrices because they fine-tuned the matrix-vector multiplication by keeping the matrix inside the register file. However, with a limited number of registers per multiprocessor inside GPU, there are fewer threads to saturate the floating-point units. Additionally, the intermediate results are merged using global memory leading to high communication latency. We therefore see an extremely low efficiency for tall-skinny matrices, *e.g.* 2.8% efficiency on Nvidia C2050 for

**Algorithm 15** Householder QR [39]

```
  – Notations –
  – $C(i{:}b_R, j)$ represents a column vector starting from row $i$ to row $b_R$ –
  – $\mathrm{x}_i$ represents $i^{th}$ column vector –
  – $\mathrm{x}_i(l)$ represents element $l$ in vector $\mathrm{x}_i$ –
  – $ddot$(x, y, $b_R$) represents $\mathrm{x}^T\mathrm{y}$ for vectors having length $b_R$ –
  – $axpy$($\alpha$, x, y, $b_R$) represents y← $\alpha$x + y with x, y of length $b_R$ –
Require: A matrix $C \in \mathbb{R}^{b_R \times k}$
  for $i = 1$ to $k - 1$ do
    – Generate Householder reflector –
    $\mathrm{x}_i := C(i : b_R, i)$                                  ($hqr1$)
    $d_1 :=$ ddot($\mathrm{x}_i$, $\mathrm{x}_i$, $b_R - i - 1$)      ($hqr2$)
    $d_2 := \sqrt{d_1} = \|\mathrm{x}_i\|_2$                          ($hqr3$)
    $\mathrm{v}_i := \mathrm{x}_i$                                    ($hqr4$)
    $\mathrm{v}_i(1) := \mathrm{x}_i(1) + \mathrm{sign}(\mathrm{x}_i(1))d_2$   ($hqr5$)
    $d_3 :=$ ddot($\mathrm{v}_i$, $\mathrm{v}_i$, $b_R - i - 1$)      ($hqr6$)
    $\tau_i := \frac{-2}{d_3}$                                        ($hqr7$)

    – Update trailing columns of $C$ –
    for $j = i$ to $k$ do
      $\mathrm{y}_j := C(i : b_R, j)$                                 ($hqr8$)
      $d_4 :=$ ddot($\mathrm{y}_j$, $\mathrm{v}_i$, $b_R - i - 1$)    ($hqr9$)
      $d_5 := \tau_i d_4$                                             ($hqr10$)
      $\mathrm{y}'_j :=$ axpy($d_5$, $\mathrm{v}_i$, $\mathrm{y}_j$, $b_R - i - 1$)   ($hqr11$)
      $C(j : b_R, j) : = \mathrm{y}'_j$                               ($hqr12$)
    end for
  end for
  return The upper triangular part of $C$ containing the matrix $R \in \mathbb{R}^{k \times k}$ and matrix $V \in$
  $\mathbb{R}^{b_R \times k}$ where individual columns are indexed $\mathrm{v}_i$ and a vector $\mathrm{t} \in \mathbb{R}^{k \times 1}$ containing $\tau_i$ values.
```

6400×51 matrix. We discuss this in more detail in Section 6.5.2. We observe low performance with previous FPGA-based QR factorization [88] as the architecture is optimized for large square matrices. Table 6.1 summarizes the performance of different implementations with the year, method, GFLOPs and the efficiency for square and tall-skinny matrices. We use 6400×6400 as the square matrix in order to compare against the results reported in [44] and 6400×51 as the tall-skinny matrix since 51 is the maximum number of columns in our design as discussed in Section 6.4.

Table 6.1: Comparison of QR Factorization (double-precision).
(Square: 6400×6400, Tall-Skinny (TS): 6400×51, FPGA clock frequency is 315 MHz for our design whereas clock frequency used for peak GFLOPs is not reported in [87]).

| **Ref.** | **Year** | **Method** | **Device** | **Matrix Structure** | **GFLOPs** | **Efficiency** (% Peak) |
|---|---|---|---|---|---|---|
| [8] | 2010 | CAQR | Nvidia C2050 | Square<br>**TS** | 104.7<br>**14.8** | 20.3%<br>**2.8%** |
| [44] | 2010 | Tile CAQR | Intel E7340 | Square<br>**TS** | 90<br>**2.0** | 58.8%<br>**1.2%** |
| [88] | 2011 | Tile CAQR | Virtex-6 LX760 | Square<br>**TS** | 24<br>**12** | 11%<br>**7%** |
| Our Work | 2012 | TSQR | Virtex-6 SX475T | **TS** | **62** | **36%** |

## 6.3 Proposed Architecture

### 6.3.1 Parallelism

#### Coarse-Grain Parallelism in TSQR

In TSQR, tiles in the local QR stage and within each merge stage can be factorized in parallel as shown in Figure 6.4.

#### Fine-Grain Parallelism in Householder QR

We show the data-flow graph (DFG) of Algorithm 15 in Figure 6.5. From the DFG, we observe that the main computationally extensive parts are the BLAS Level 1 *ddot* operations, *i.e.* dot products $\mathrm{x}_i^T\mathrm{x}_i$ (*hqr*2), $\mathrm{v}_i^T\mathrm{v}_i$ (*hqr*6) and $\mathrm{y}_j^T\mathrm{v}_i$ (*hqr*9) where $i \leq j \leq k$. *ddot* operation has a sequential latency of $O(b_R)$ but it can be implemented as a tree-reduction circuit with a $O(\log b_R)$ latency. Additionally, the inner loop of Algorithm 15 can be fully unrolled to compute $\mathrm{y}_j^T\mathrm{v}_i$ (*hqr*9) in parallel for different values of $j$.

There is another BLAS Level 1 operation *axpy*, *i.e.* multiplication of vector by a scalar followed by a vector addition operation (*hqr*11). *axpy* has a sequential latency of $O(b_R)$ but since it is a data-parallel operation it can be fully unrolled to complete with $O(1)$ latency.

### 6.3.2 Work vs. Critical Latency

We now explore the gap between the parallelism that is available and the parallelism that can be exploited with limited resources. The latency of the critical path of fully-parallel TSQR is given by Equation (6.1) and (6.2).

$$l_{TSQR} = (\lceil \log_2 N_{tiles} \rceil + 1) l_{hqr} = (\lceil \log_2 \frac{n}{2k} \rceil + 1) l_{hqr} \quad (6.1)$$
$$l_{hqr} = (k-1)(c_1 \lceil \log_2 k \rceil + c_2). \quad (6.2)$$

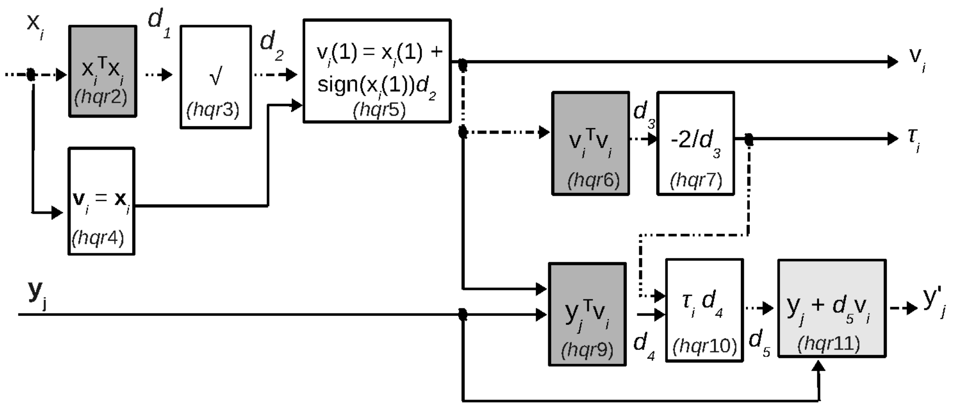


Figure 6.5: Householder QR DFG showing the dark grey blocks for *ddot* and the light grey block for *axpy*. The critical path of the Householder QR is shown as the blocks connected using dotted arrows.

Referring to Equation (6.1) and Equation 6.2, $l_{TSQR}$ contains a single $l_{hqr}$ term for local QR factorization plus $\lceil \log_2 N_{tiles} \rceil \times l_{hqr}$ for all the merge stages (assuming all QR factorizations in local QR stage and within each merge stage are performed in parallel). $l_{hqr}$ represents the latency of Householder QR from Algorithm 15. $c_1\lceil \log_2 k \rceil + c_2$ is the latency of the critical path shown in Figure 6.5, $\lceil \log_2 k \rceil$ is the latency of *ddot* and $c_1$, $c_2$ are constants representing latencies of the double-precision floating-point operators ($c_1 = 24$ and $c_2 = 237$ based on latency values from Xilinx Coregen Library). $k - 1$ corresponds to the iterations of the outer loop of Algorithm 15.

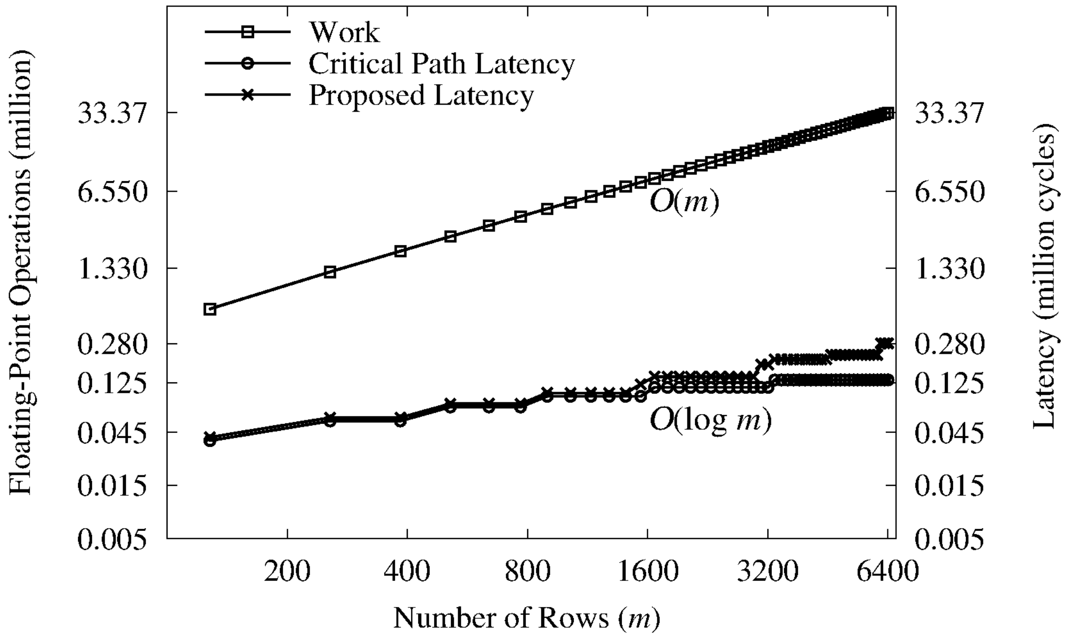


Figure 6.6: Work vs. Critical Latency ($k$ = 51).

We show the work ($2nk^2 - \frac{2}{3}k^3$ FLOPs) vs. critical latency of fully-parallel TSQR in Figure 6.6. From the figure we observe that there is a considerable gap between the latency of sequential and fully-parallel implementation of TSQR *e.g.* a 278× speed up can be achieved for a 6400×51 matrix and this gap increases with the increase in the height of the matrix. We also plot the proposed latency and show that it approaches critical latency of the fully-parallel implementation. We now discuss the parallel architecture which achieves the proposed latency.

### 6.3.3 Parallel Architecture for Householder QR

Before introducing our proposed design, we briefly survey a few architectures already presented for Householder QR. A systolic architecture is introduced in [59] for QR factorization in real-time signal processing applications having very small matrices ($n \sim$ a few 10s). Recently, Tai *et.al.* [88] presented an architecture comprising linear array of processing elements (PEs) for the Householder QR targeting large square matrices. Each PE is responsible for computing an iteration of the outer loop in Algorithm 15. Starting from the first column, the first PE of the linear array computes the Householder reflector and then performs the trailing update on the second column. Meanwhile, the first PE is performing the trailing update on the rest of the columns, the second column is then transferred to the second PE for repeating the same calculations. Hence, this architecture only exploits fine-grain parallelism within a single tile without paying attention to the coarse-grain parallelism as discussed in Section 6.3.1. Our architecture is novel in the sense that it exploits both coarse-grain as well as fine-grain parallelism. We now discuss how we exploit the fine-grain parallelism in Householder QR and leave the coarse-grain parallelism until Section 6.3.4.

The computational complexity of Householder QR is $O(b_R k^2)$ with $O(b_R k)$ memory operations for a $b_R \times k$ input matrix. We design our architecture to exploit this arithmetic intensity. As identified in Section 6.3.1, there are two main computational blocks in Householder QR, *i.e. ddot* and *axpy*. We take advantage of high on-chip memory bandwidth of the FPGA and use a deeply-pipelined tree-reduction circuit for *ddot* which is capable of computing a new dot product at every clock cycle. We share the circuit for computing dot products ($hqr2$, $hqr6$ and $hqr9$) in Algorithm 15. We store the matrix $B_i$ in a banked row fashion and feed the vectors $\mathrm{x}_i$ or $\mathrm{y}_j$ to the dot product circuit during different phases of Algorithm 15. We use a parallel-in parallel-out shift register to store $\mathrm{v}_i$. We completely unroll the *axpy* operation and use an array of multipliers and adders to perform this operation in parallel. The parallel architecture is shown in Figure 6.7 where the number of floating-point units grows linearly with $n$ as given by Equation (6.3). Table 6.2 lists the actual number of each floating-point unit used in our proposed architecture.

$$\text{Total FP Units} = 8k + 3. \tag{6.3}$$

We exploit all the fine-grain parallel potential available within Householder QR except that we do not unroll the inner loop of Algorithm 15 due to high memory bandwidth requirements and instead, use pipelining to feed a new dot product operation in each clock cycle to perform $\mathrm{y}_j^T \mathrm{v}_i$ ($hqr9$) for different values of $j$. The latency of QR factorization of $B_i \in \mathbb{R}^{2k \times k}$ using the

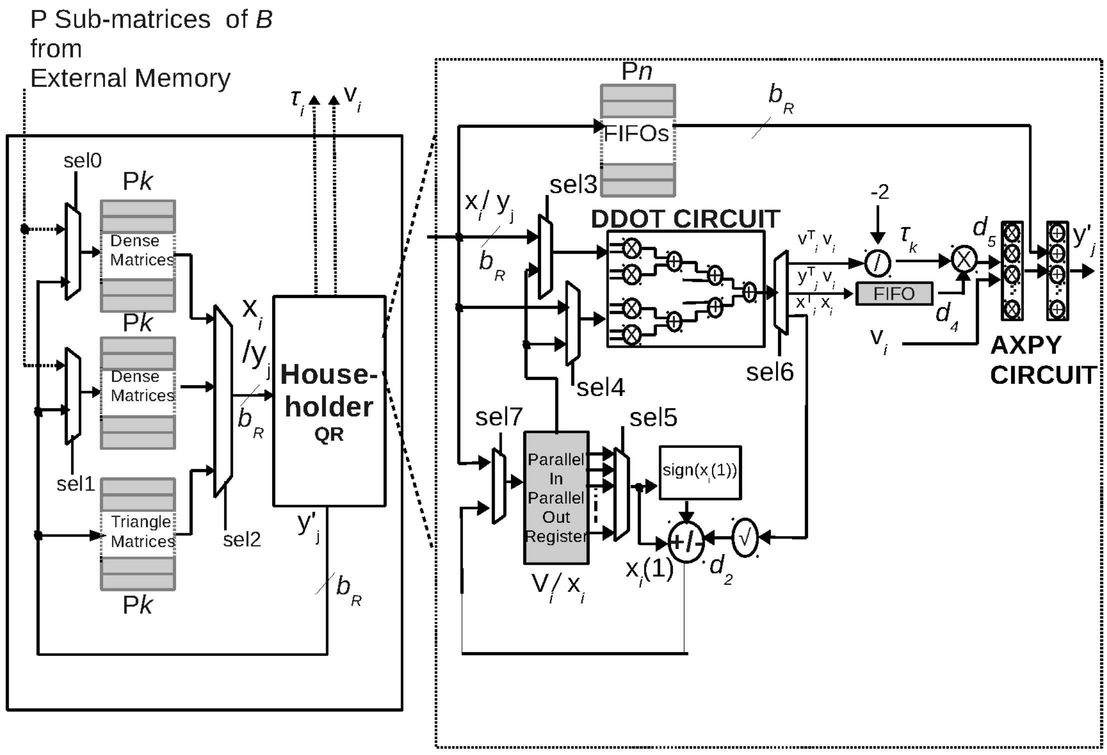


Figure 6.7: Parallel Architecture for TSQR, *Dense Matrices* memory stores $B_i$s whereas *Triangle Matrices* memory stores intermediate $R$ factors.

Table 6.2: Householder QR floating-point units.

| **Operation** | **Floating Point Units** |
|---|---|
| Dot Product | $2b_R - 1$ |
| Vector by Vector Addition | $b_R$ |
| Scalar by Vector Multiplier | $b_R$ |
| Adder/Subtractor | 1 |
| Multiplier | 1 |
| Square root | 1 |
| Divider | 1 |
| Total Floating Point Units | $4b_R$+3 = $8k$+3 |

proposed architecture is given by Equation (6.4).

$$l'_{hqr} = \frac{k(k+1)}{2} + (k-1)(c_1\lceil \log_2 k\rceil + c_2). \tag{6.4}$$

Comparing Equation (6.2) and (6.4), we can see that the term $\frac{k(k+1)}{2}$ is introduced due to pipelined implementation of inner loop of Algorithm 15.

### 6.3.4 Pipeline Parallelism for Mapping TSQR

For QR factorization of a single tile, the deeply-pipelined nature of the dot product circuit in Figure 6.7 leads to high throughput but also considerable latency. As a result, the pipeline will be underutilized if only single tile is factorized, therefore, we exploit this mismatch between throughput and latency to factorize multiple independent tiles within TSQR. The initiation interval of this circuit is $2(k-1) + \frac{k(k+1)}{2}$ clock cycles (for $hqr2$, $hqr6$ and inner loop of Algorithm 15 containing $hqr9$) after which a new QR factorization can be streamed into this circuit. The pipeline depth (P) of the circuit is given by Equation (6.5) which indicates how many QR factorizations can be active in the pipeline at one time as shown in Figure 6.8.

$$\mathrm{P}(k) \quad = \quad \left\lceil \frac{\frac{k(k+1)}{2} + (k-1)(c_1\lceil \log_2 k \rceil + c_2)}{2(k-1) + \frac{k(k+1)}{2}} \right\rceil . \tag{6.5}$$

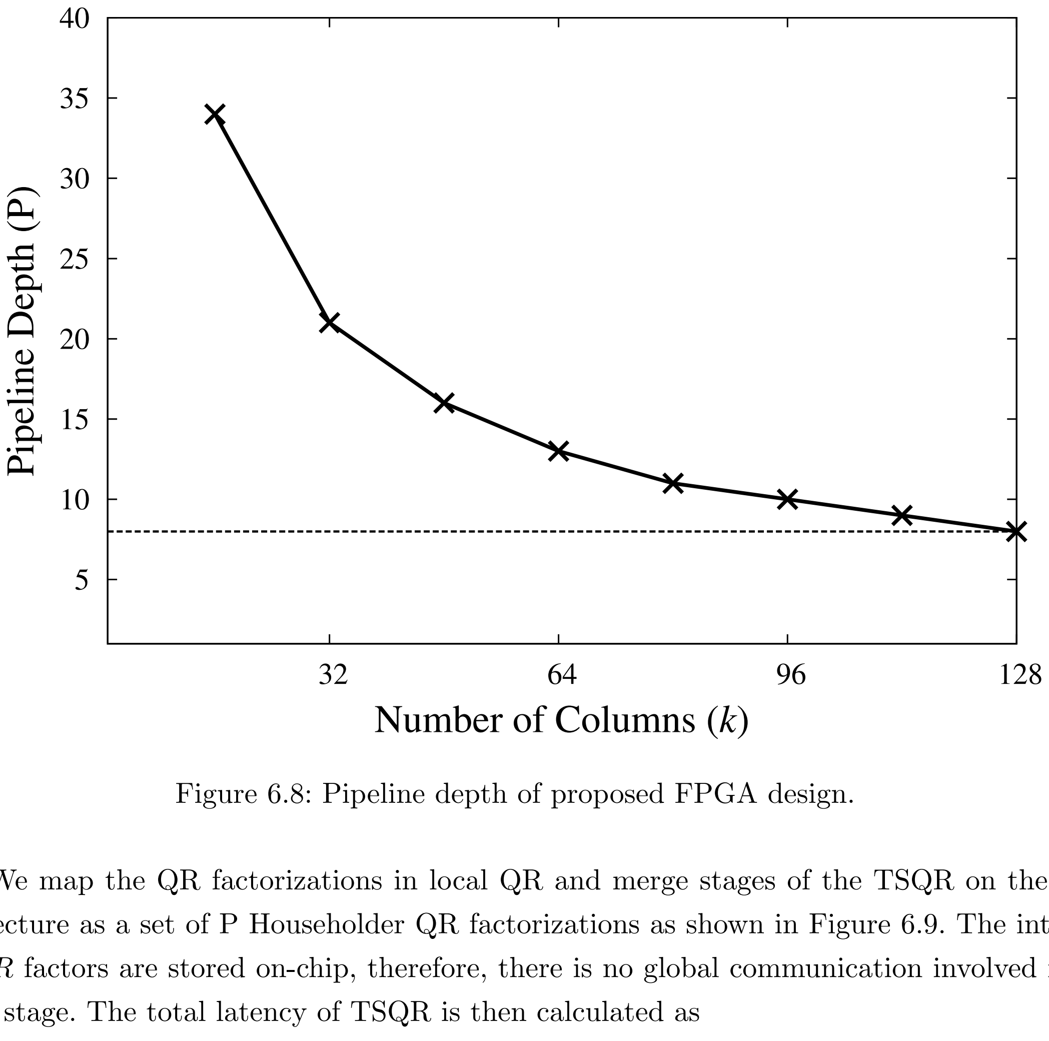


Figure 6.8: Pipeline depth of proposed FPGA design.

We map the QR factorizations in local QR and merge stages of the TSQR on the same architecture as a set of P Householder QR factorizations as shown in Figure 6.9. The intermediate $R$ factors are stored on-chip, therefore, there is no global communication involved in the merge stage. The total latency of TSQR is then calculated as

$$l_{TSQR} = l_{hqr} \sum_{i=0}^{\lceil \log_2 N_{tiles} \rceil} \left\lceil \frac{N_{tiles}}{2^i \mathrm{P}} \right\rceil . \tag{6.6}$$

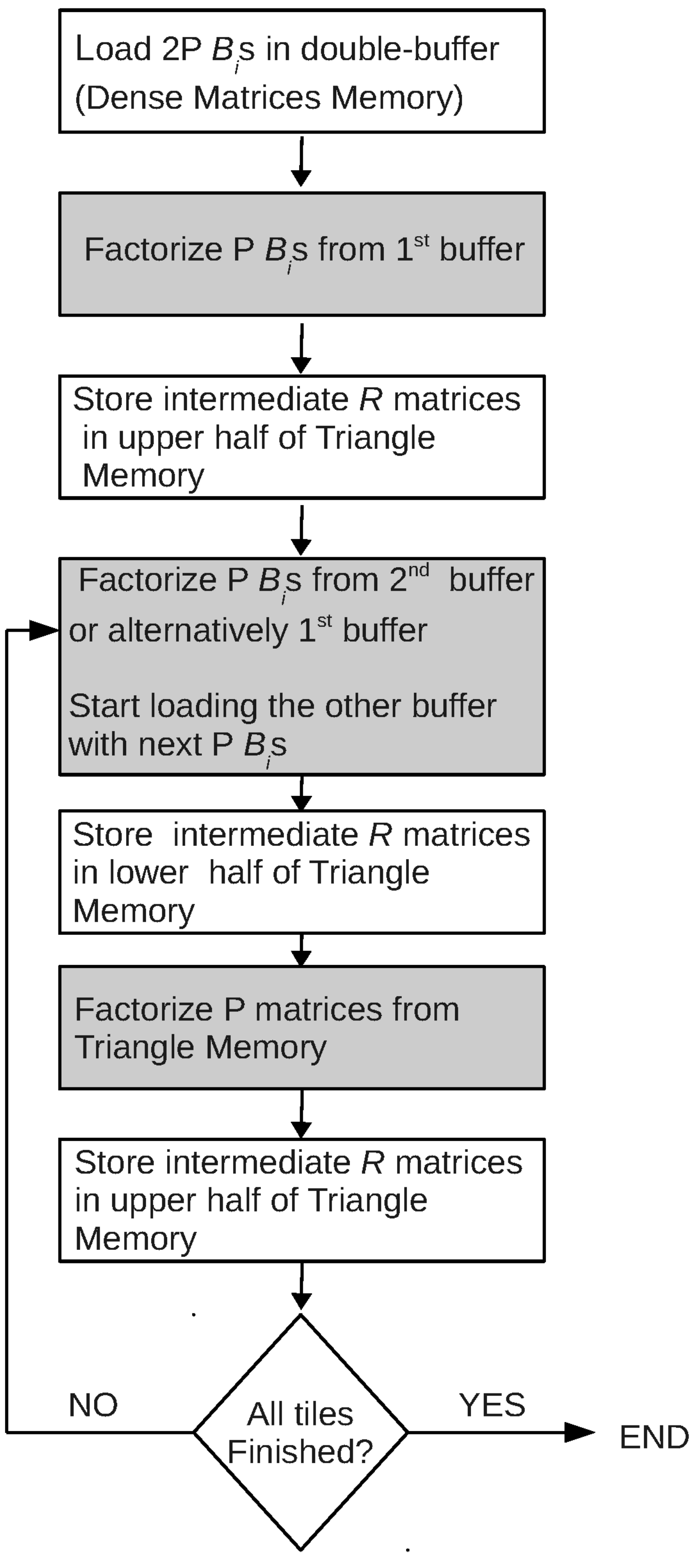

Load 2P $B_i$s in double-buffer (Dense Matrices Memory)
Factorize P $B_i$s from 1st buffer
Store intermediate $R$ matrices in upper half of Triangle Memory
Factorize P $B_i$s from 2nd buffer or alternatively 1st buffer
Start loading the other buffer with next P $B_i$s
Store intermediate $R$ matrices in lower half of Triangle Memory
Factorize P matrices from Triangle Memory
Store intermediate $R$ matrices in upper half of Triangle Memory
All tiles Finished?
NO
YES
END

Figure 6.9: Mapping of TSQR, $B_i$s $\in \mathbb{R}^{2k \times k}$ whereas $R \in \mathbb{R}^{k \times k}$.

Referring to Equation (6.6), each term corresponds to latency of a single TSQR stage shown in Figure 6.4.

### 6.3.5 I/O Considerations

We assume that the matrix is stored in an off-chip memory just like in the GPU case. We factorize P dense sub-matrices from the double-buffer and P sub-matrices from Triangle Matrices memory before we require a new set of P sub-matrices from the off-chip memory. Therefore, our I/O time to fetch P sub-matrices is double the computation time for QR factorization of P such sub-matrices.

$$\text{I/O bandwidth} \quad = \quad \frac{64\mathrm{P}(3\mathrm{b_R}k - k^2 + k)}{2\mathrm{P}(2k - 2 + \frac{k(k+1)}{2})} \text{ bits/cycle.} \qquad (6.7)$$

Referring to Equation (6.7), we require $64\mathrm{P}(3b_Rk - k^2 + k)$ bits to be exchanged between the FPGA and the off-chip memory where the $b_Rk$ term comes from the size of input sub-matrix, $2b_Rk - k^2 + k$ from the $v_i$ vectors and $\tau_i$ values for explicit formation of Q matrix (see Section 6.1.2). The latency of factorizing P sub-matrices after matching the pipeline depth is $\mathrm{P}(2k - 2 + \frac{k(k+1)}{2})$ cycles and it is twice of this time that is available to load new set of P sub-matrices. Given the maximum value of $b_R$ ($2k$ for binary tree) to be 102 resulting into maximum value of $k = 51$ (see Section 6.4), we find the I/O requirement to be 11.53 GB/sec (∼30% of maximum memory bandwidth available in Virtex-6 SX475T [87]).

## 6.4 Evaluation Methodology

The experimental setup for performance evaluation is summarized in Table 6.3. We use GFLOPs as the metric for comparing performance on different architectures. We use highly optimized linear algebra libraries for GPU (CULA), multi-cores (Intel MKL) and hybrid systems (MAGMA). For FPGA, we only implemented the Householder QR block which can factorize a single sub-matrix. We actually measure the latency of factorizing a single sub-matrix and estimate the total number of sub-matrices that can be factorized in a pipelined fashion. The double-buffering as well as mapping of TSQR on the same architecture is only estimated. We implement the proposed architecture in VHDL using double-precision floating-point cores and synthesize the circuit for Xilinx Virtex-6 SX475T FPGA, a device with the largest number of DSP48Es and a large on-chip capacity. The placed and routed design has an operating frequency of 315 MHz. We find out the maximum value of $\mathrm{b}_R$ to be 102 before 90% of the Slice LUTs, 96% of DSP48Es and 77% of BRAMs are utilized. Although, $m$ can take on any value only limited by off-chip memory, $b_R$ limits the maximum value of $n$ because $b_R$ should be greater than or equal to $n$ (in our design $b_R = 2k$ for binary tree). The value of $k$ is appropriate for tall-skinny QR applications where it is on the order of a few 10s. The proposed design needs to be re-synthesized only if the input matrix size changes in the column dimension ($k$).

## 6.5 Results

We now present the performance achieved by our FPGA design, compare it with optimized QR routines from linear algebra libraries and the state of the art in FPGAs, multi-cores and GPU and then discuss the underlying factors that explain our results.

Table 6.3: Experimental Setup (FPGA clock frequency used for peak GFLOPs is not reported in [87]).

| **Platform** | **Peak GFLOPs** Double-Precision | **Compiler** | **Libraries** | **Timing** |
|---|---|---|---|---|
| Intel Xeon X5650 **(32 nm)** | 63.9 [27] | gcc (4.4.3(-O3)) | Intel MKL (10.2.4.032) | PAPI (4.1.1.0) |
| Nvidia GPU C2050 **(40 nm)** | 515 | nvcc | CULA-R11 MAGMA-rc5 | cudaEvent-Record() |
| Virtex-6 SX475T **(40 nm)** | 171 [87] | Xilinx ISE (10.1) | Xilinx Coregen | ModelSim |

### 6.5.1 FPGA Performance Evaluation

The peak and sustained double-precision floating-point performance of the FPGA is given by Equation (6.8) and Equation (6.9) respectively.

$$\text{Peak Throughput} = 8k+3 \quad \text{FLOPs/cycle.} \tag{6.8}$$

$$\text{Sustained Throughput} = \frac{2nk^2 - \frac{2}{3}k^3}{t_{TSQR}}\text{FLOPs/cycle} \tag{6.9}$$

where $8k+3$ is the total number of floating-point units in the proposed design and $2nk^2 - \frac{2}{3}k^3$ represents FLOPs in TSQR. For an operating frequency of 315 MHz, the peak performance of our design for maximum value of $k = 51$ is 129 GFLOPs and it is observed that the efficiency of the proposed architecture is greater than 80% for tall-skinny matrices as shown in Figure 6.10.

### 6.5.2 Comparison with GPU

We compare our work against a custom implementation of QR factorization using communication-avoiding QR (CAQR) algorithm on Nvidia C2050 Fermi [8]. In this implementation, the matrix is divided into panels which are factorized using TSQR (*hh_factor_dense* and *hh_factor_triangle*) and then the trailing matrix update (*hh_update_dense* and *hh_update_triangle*) is performed. We use Compute Visual Profiler [70] to profile GPU code for a range of extremely tall-skinny to square matrices as shown in Figure 6.11. We observe that TSQR dominates in case of extremely tall-skinny matrices (∼70% of runtime contribution).

Firstly, we compare the arithmetic intensity in TSQR in GPU with our design. In GPU, the Householder QR factorizations are performed by keeping the tiles inside the register file due to its high access bandwidth (∼ 8 TB/s). The panel width is tuned to a value of 8 and a tile size of 64×8 is chosen such that it can fit into the register file of each stream multiprocessor (SM) for best performance. This restricts the arithmetic intensity in Householder QR as there are $O(b_R k^2)$ FLOPs for $O(b_R k)$ memory operations. The width of the panel ($k$) can be increased at the expense of height ($b_R$) but it then increases the number of stages in the merge stage of

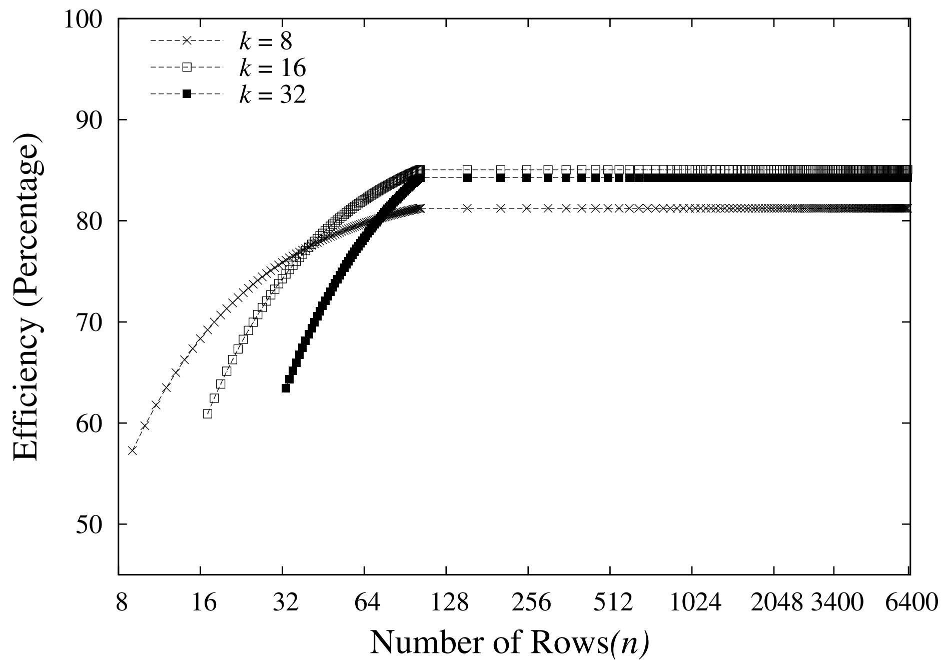


Figure 6.10: Efficiency of proposed FPGA design as percentage of its peak performance.

TSQR. In our design, the panel width can be at most 51, a 49× increase in arithmetic intensity compared to GPU. The relatively large panel width not only reduces the number of merge stages in TSQR but also the number of panels to be factorized, *e.g.* for a matrix of size 6400×51, there will be seven panels which need to be factorized using TSQR in GPU whereas in our design only single TSQR factorization is required.

Secondly, we compare the type of memory and its bandwidth used in local QR and merge stages of TSQR for both cases. In the GPU, during local QR stage, tiles are loaded into the register file of each SM from global memory. The tiles are then factorized using the register file as well as shared memory and the local $R$ factors are then stored back in global memory. In the merge stage, the $R$ factors are loaded back into register file from distributed locations of global memory and are then factorized. In our design, however, we perform both local QR and merge stages using on-chip memory to minimize memory latency. Table 6.4 lists the type of memories used in each stage for GPU as well as FPGA.

Table 6.4: Type of Memory and its peak bandwidth for GPU and proposed FPGA design in different stages of TSQR.

| **Kernel** | **GPU Memory** | | | **FPGA Memory** | |
|---|---|---|---|---|---|
| | Register (∼8 TB/s) | Shared (∼1.3 TB/s) | Global (144 GB/s) | Register (∼36 TB/s) | BRAMs (∼5.4 TB/s) |
| `local QR` | √ | √ | | √ | √ |
| `merge stage` | √ | √ | √ | √ | √ |

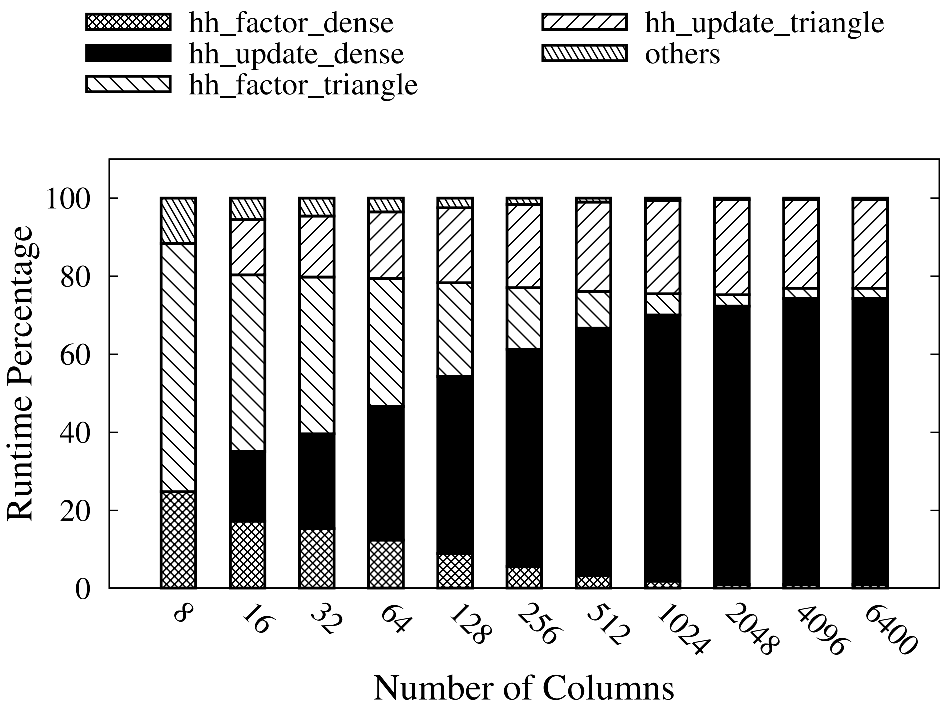


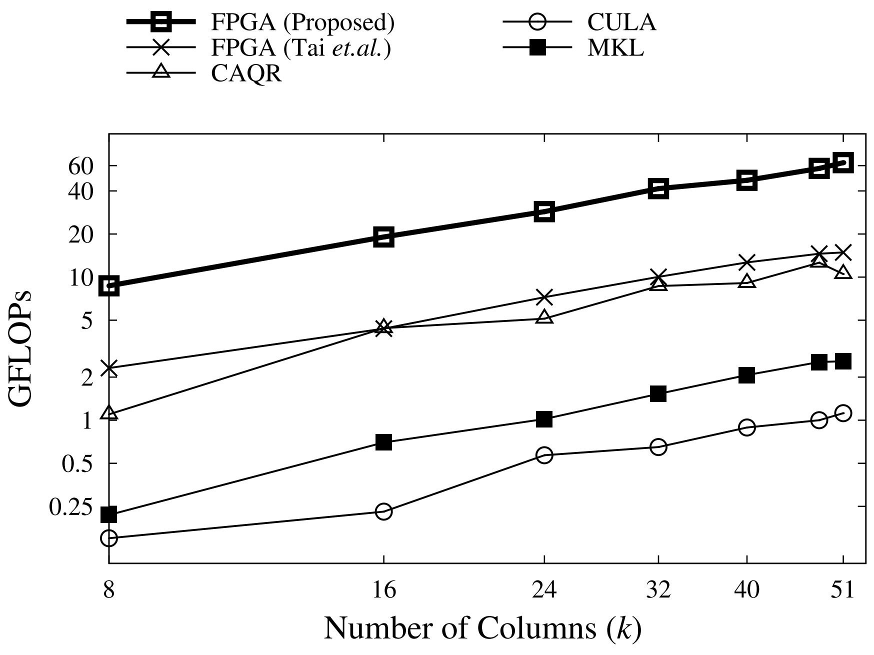
Figure 6.11: GPU Performance Analysis (Number of Rows = 6400).

Lastly, we compare the utilization of floating-point units in GPU and our custom architecture. We identify the limiting factors in the kernels used for TSQR in GPU as shown in Table 6.5. Both the kernels perform QR factorization by a thread block having 64 threads and there are 8 thread blocks used per SM. Each thread has 63 registers and therefore maximum number of threads is limited and as a result occupancy is low. It is due to this low occupancy ratio particularly in the *hh_factor_triangle* kernel that we get very low performance for tall-skinny matrices. On the other hand, we get almost 80% of peak performance (129 GFLOPs) for extremely tall-skinny matrices (see Section 6.5.1). As a result, we get a speed up of 3.38× - 12.70× (6.47× geo.mean) shown in Figure 6.12(a) and 6.12(b).

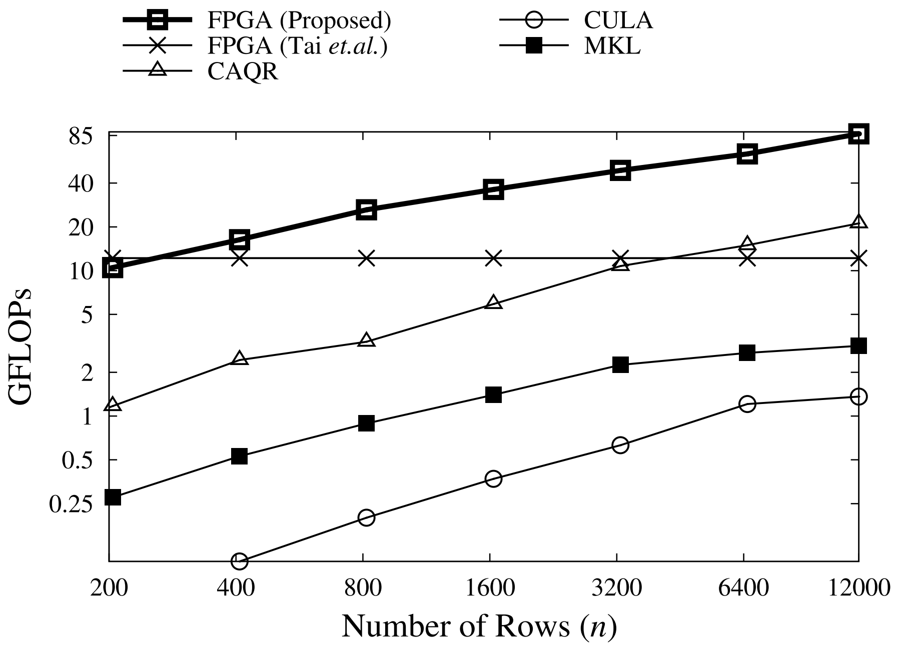
Table 6.5: Limiting Factors for Tall-Skinny Matrices (6400×51) on GPU. A *warp* comprises 32 threads and there are 48 active warps per cycle for an occupancy of 1.

| **Kernel** | **Active Warps / Active Cycles** | **Occupancy Ratio** | **Limiting Factor** |
|---|---|---|---|
| `hh_factor_dense` | 9.05 | 0.18 | Registers |
| `hh_factor_triangle` | 2.98 | 0.06 | Registers |

We also compare QR routines from CULA which uses Block Householder QR with dominant sequential panel factorization for tall skinny matrices. We show a speed up of 52× - 258× (117× geo.mean) due to parallel TSQR algorithm and its efficient FPGA implementation.



(a) GFLOPs vs. $k$ ($n$ = 6400)



(b) GFLOPs vs. $n$ ($k$ = 51)

Figure 6.12: Performance Comparison with Multi-Cores (Intel MKL), GPUs (CULA, CAQR), and best FPGA work.

### 6.5.3 Comparison with Multi-Cores and Related FPGA Work

We now compare our design with QR routines from MKL and MAGMA which perform panel factorization in a sequential fashion on multi-cores. We observe a speed up of 21.7× - 50.2× (31.5× geo.mean) and 7.28× - 36.7× (16× geo.mean) over MKL and MAGMA respectively. We finally compare our design with the most recent work on QR factorization on FPGAs by Tai *et.al.* [88]. In [88], the architecture is tuned for large square tiles and therefore we see increase in performance with increase in the width of the matrix as shown in Figure 6.12(a). However, since no parallelism is exploited along the row dimension, therefore, we see constant performance as we increase the height of the matrix shown in Figure 6.12(b). We, therefore, observe a speed up of 0.57× - 7.7× (2.2 × geo.mean) across a range of matrix sizes.

## 6.6 Summary

We show that customizing an FPGA architecture to the shape of the matrix for TSQR can give us up to 7.7× and 12.7× speed up over state of the start in FPGAs and GPUs respectively. We highlight the low efficiency of GPUs and multi-cores as percentage of their peak theoretical performance for QR factorization of tall-skinny matrices. We identify that on GPUs it is due to low arithmetic intensity caused by limited registers, low occupancy during the merge stage, and global communication during the merge stage where as in case of multi-cores it is primarily due to low memory bandwidth. We show how we can exploit the high on-chip bandwidth of the FPGA to design a high-throughput architecture for QR factorization and large on-chip capacity to perform the merge stage without any global communication. We conclude that even though GPU has 3× higher peak double-precision floating-point performance, by exploiting the architectural features of FPGAs we can outperform the GPU for communication-avoiding linear algebra algorithms like TSQR.

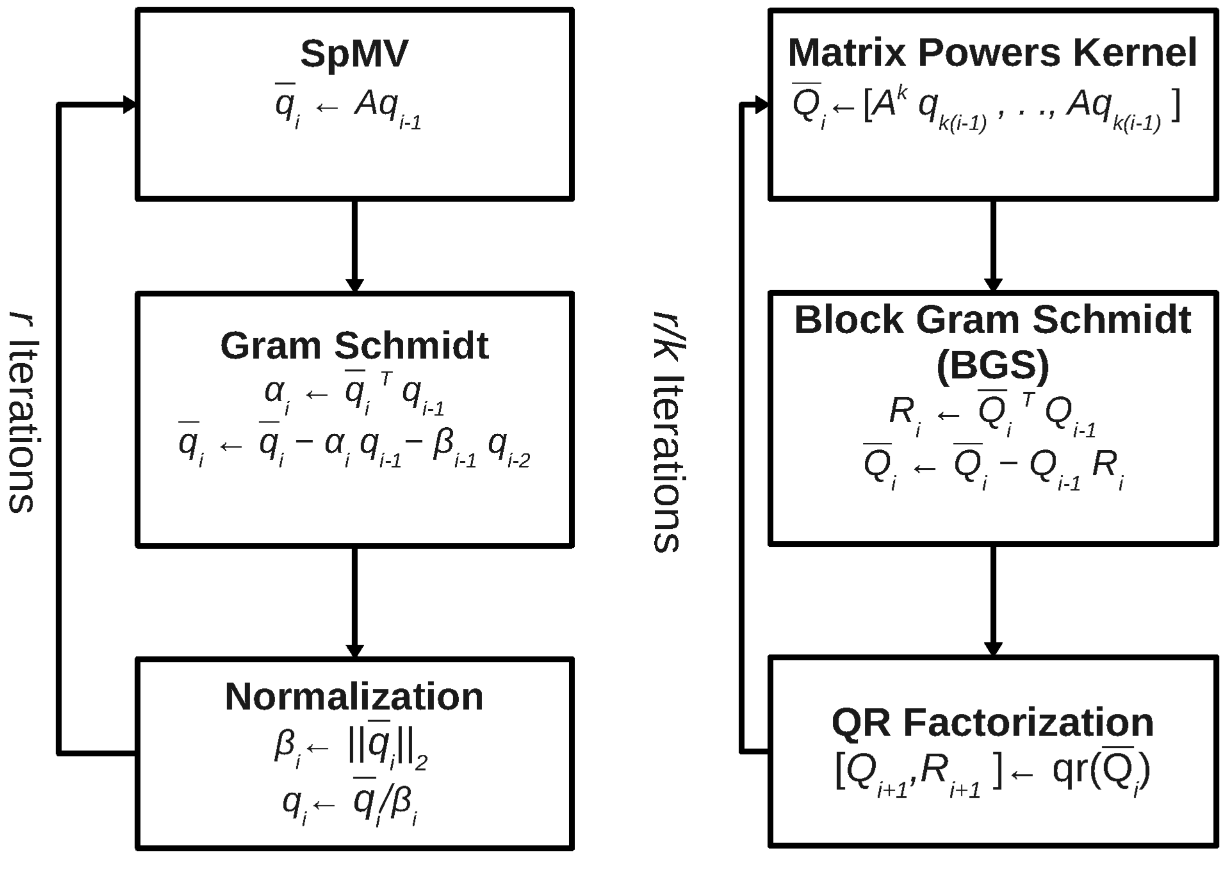
# 7 Communication Optimization in Iterative Numerical Algorithms for Large-Scale Problems

In Chapter 1, we introduce the communication problem in iterative numerical algorithms for solving sparse linear system of equations $Ax = b$ and eigenvalue problems $Ax = \lambda x$. This communication problem arises due to the repeated access of the matrix $A$ from the off-chip memory for sparse matrix-vector multiplication (SpMV) to generate a new vector in each iteration. We further highlight the low silicon efficiency of the common hardware accelerators (FPGAs and GPUs) in accelerating such algorithms due to a big gap in their communication (off-chip memory bandwidth and latency) and computational (flops/sec) performance. In Chapter 4, for small-to-medium scale problems, we show how we can exploit large on-chip memory of the FPGAs to store the matrix and then re-use it for all iterations. In this way, we reduce the communication cost by a factor of $r$ equal to the total number of iterations. This helps us to increase the silicon efficiency from 4% to 38% in case of a Virtex6-SX475T FPGA. In Chapter 5 and Chapter 6, we focus on individual kernels within the communication-avoiding iterative numerical algorithm, an algorithmic approach which trades communication with redundant computation to minimize overall runtime for the problems where the matrix $A$ can not be stored in the on-chip memory. The basic idea behind the communication-avoiding approach is partitioning the matrix into blocks, fetching the blocks once and generating $k$ vectors in one shot. This reduces the communication cost by a factor of $k$ but at the expense of an increase in computation as shown in Equation (1.4) for $r$ iterations, repeated below for convenience.

$$t_r'' \;=\; \frac{r}{k} \times t_{comm} + r \times t_{comp} + \frac{r}{k} \times f(k).$$

While the re-use of the matrix reduces the iterations by a factor of $k$ for a desired accuracy in solving $Ax = b$ and $Ax = \lambda x$, overall performance is determined by two factors (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 $k$ that minimizes overall runtime by providing a tradeoff between computation and communication cost.

In this chapter, we present a systematic approach to compose multiple linear algebra kernels and show how to carefully select the algorithmic parameter $k$ within the communication-avoiding iterative numerical algorithms. There are multiple factors that influence the selection of the parameter $k$. Firstly, a higher value of $k$ incurs an additional computation cost of $O(nk^2)$ due to the introduction of new kernels like BGS and QR factorization as shown in Figure 7.1. Secondly, we need an extra on-chip storage cost of $O(nk)$ as $k$ vectors need to be stored. In


SpMV
$\overline{q}_i \leftarrow Aq_{i-1}$
Gram Schmidt
$\alpha_i \leftarrow \overline{q}_i^T q_{i-1}$
$\overline{q}_i \leftarrow \overline{q}_i - \alpha_i q_{i-1} - \beta_{i-1} q_{i-2}$
Normalization
$\beta_i \leftarrow \|\overline{q}_i\|_2$
$q_i \leftarrow \overline{q}_i/\beta_i$
r Iterations
r/k Iterations
Matrix Powers Kernel
$\overline{Q}_i \leftarrow [A^k q_{k(i-1)}, \ldots, Aq_{k(i-1)}]$
Block Gram Schmidt (BGS)
$R_i \leftarrow \overline{Q}_i^T Q_{i-1}$
$\overline{Q}_i \leftarrow \overline{Q}_i - Q_{i-1} R_i$
QR Factorization
$[Q_{i+1}, R_{i+1}] \leftarrow \mathrm{qr}(\overline{Q}_i)$

(a) Lanczos Iteration (b) CA-Lanczos

Figure 7.1: Lanczos Iteration vs. Communication-Avoiding Lanczos (CA-Lanczos) ($A \in \mathbb{R}^{n \times n}$, $\overline{Q_i} \in \mathbb{R}^{n \times (k+1)}$, $Q_i \in \mathbb{R}^{n \times k}$ $R_i \in \mathbb{R}^{k \times k}$, $q_i \in \mathbb{R}^{n \times 1}$, $n \gg k$).

the case they can not be stored on-chip, the communication cost further increases as BGS and QR operates on these vectors. Thirdly, there are algorithmic constraints on the maximum value of $k$ as the iterative algorithm breaks down due to loss of orthogonality [62]. We therefore, present a methodology which takes into account resource constraints, algorithmic restrictions and computation-communication tradeoff to select the value of $k$ for a particular FPGA device. Using the Lanczos Iteration operating on structured sparse banded matrices as a case study, we demonstrate that minimizing communication on FPGAs by this tight algorithm-architecture interaction can increase its silicon efficiency 4 fold from 2% to 8%. We observe that while GPUs may offer higher throughput for some of the individual kernels, overall application performance is limited by the inability to support on-chip sharing of data across different kernels. We therefore show superior performance of the FPGA over GPU despite the letter's ∼5× larger off-chip memory bandwidth and ∼2× greater peak single-precision floating-point performance. We give you a early preview of the results in Figure 7.2 where we compare the performance of the Lanczos Iteration and its variant Communication-Avoiding Lanczos (CA-Lanczos) [48] on GPU and FPGAs.

We see three distinct regions.

1. For small problems, CA-Lanczos is better than the standard Lanczos method on both

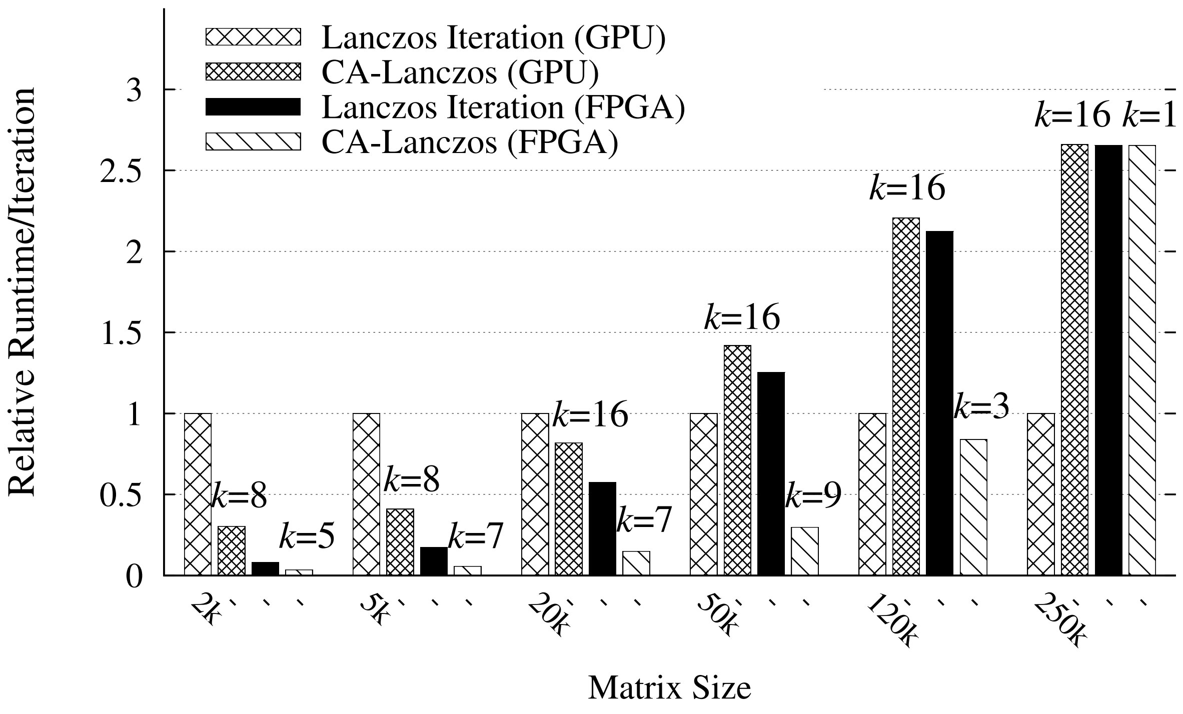


Figure 7.2: Single-Precision performance comparison of Communication-Avoiding Lanczos (CA-Lanczos) on GPU (Nvidia C2050 Fermi) and FPGA (Virtex6-SX475T) in terms of time/iteration. The input matrices are banded with a band size $b = 27$. $k$ is an algorithmic parameter whose optimal value is shown at the top of each bar for CA-Lanczos.

GPU and FPGAs.

2. For medium to large scale problems, composition of kernels on a GPU increases overall runtime due to off-chip sharing of data (see Section 7.3.2). While performance of standard Lanczos Iteration 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.

3. For extremely large problems, compared to the Lanczos Iteration, 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 GPUs 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 the on-chip memory of the FPGAs (see Section 7.4.1). We achieve a silicon efficiency of up to 8% on FPGAs compared to a 2.1% on GPU in accelerating iterative numerical algorithms (see Section 7.6.2). The main contributions of this chapter 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 algorithm.

- A resource-constrained methodology for selecting algorithmic parameter for a particular FPGA.
- A quantitative comparison between FPGA and GPU highlighting their architectural limitations.

## 7.1 Basic Linear Algebra Kernels in CA-Lanczos

CA-Lanczos advances by $k$ steps into the Lanczos Iteration by generating $k$ vectors in a single sweep as shown in Algorithm 16. CA-Lanczos comprises three kernels, a matrix powers kernel (Line 2) replacing SpMV to generate $k$ vectors, a Block Gram-Schmidt Orthogonalization (BGS) [48] 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. We now briefly discuss each kernel with its basic linear algebra blocks.

**Algorithm 16** Communication-Avoiding Lanczos [48]

```
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 $\frac{r}{k}$ do
 2:     $Qi \leftarrow [A^k q_{ki}, A^{k-1}q_{ki}, \ldots, Aq_{ki}]$   −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 $R_{i+1}$−
 7:         $q_l \leftarrow \frac{q_l}{r_{ll}}$         −$q_l$ is the $l^{th}$ 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 $R_{i+1}$ and $\overline{R_i}$
```

### 7.1.1 Matrix Powers Kernel

As discussed in Chapter 5, the matrix powers kernel replaces SpMV and re-uses the matrix to generate $k$ vectors in one shot. There are both sequential and parallel variants of the matrix powers kernel where the former minimizes communication within the memory hierarchy whereas the latter reduces communication between proccessors. In Chapter 5, we also propose a hybrid algorithm which matches the strengths of FPGAs for structured banded matrices. We minimize communication with the off-chip memory but allow communication locally between different floating-point units. This reduces redundant computation and allows larger values of $k$ and hence higher speedups. In order to compute an element of the new vector, we need to perform a dot production operation $x^T y$ where $x, y \in \mathbb{R}^b$ and $b$ is the band size of the banded matrix $A$.

### 7.1.2 Block Gram-Schmidt Orthogonalization

Gram-Schmidt Orthogonalization [39] 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 16. The computations involved are $x^T y$ and $ax + y$ on Line 3 and 4 respectively with $x, y \in \mathbb{R}^n$ and $a$ is a scalar.

### 7.1.3 QR Factorization

The $k$ vectors compose a tall-skinny matrix on Line 4 of Algorithm 16 and QR factorization of this matrix is required using some numerically stable method. We use Modified Gram-Schmidt Orthogonalization (MGS) [39] due to its parallel potential and its stability which is proved for iterative numerical algorithms [42]. Like BGS, the computations involved are $x^T y$ (Line 6 and 9) and $ax + y$ (Line 7 and 10) with $x, y \in \mathbb{R}^n$. We summarize the basic linear algebra blocks for all three kernels in Table 7.1.

Table 7.1: 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 $y \leftarrow ax + y$ ($x$, $y \in \mathbb{R}^n$) |
| QR Factorization | $z \leftarrow x^T y$ and $y \leftarrow ax + y$ ($x$, $y \in \mathbb{R}^n$) |

## 7.2 Related Work

### 7.2.1 Communication Optimization

The communication problem in scientific comptuations has historical roots in the *memory wall* [41]. Additionally, with the introduction of many-core and multi-core architectures offering high peak floating-point performance, the inter-core communication determines how much of this peak can be sustained on these architectures. This severely limits the silicon efficiency of modern parallel architectures. As a result, we see a family of new algorithms which communicate less (both within the memory hierarchy as well as between different cores) at the expense of increase in computation. Communication-avoiding iterative numerical algorithms [48], communication-avoiding dense linear algebra [15], communication-avoiding QR factorization [33] and communication-avoiding parallel Strassen [14] are a few examples. Such algorithms not only approach theoretical lower bounds of communication but also show good performance in practice. For QR factorization, Anderson *et al.* [8] show that communication-avoiding approach is promising on GPUs and demonstrate up to 13× speedup compared to standard QR factorization. For iterative numerical algorithms, Mohiyuddin *et al.* [65] show that compared to Generalized Mininum Residual Method (GMRES), its variant the Communication-Avoiding GMRES can provide a 4.3× speedup for banded matrices and up to 2.3× for general sparse matrices on an 8-core Intel Clovertown architecture.

In this work, we investigate how such a communication-avoiding approach can enhance the performance of hardware accelerators like GPUs and FPGAs while accelerating iterative numerical algorithms. We highlight the factors which limit the performance on current GPU architectures and show how we can exploit FPGAs to minimize communication and thereby provide better silicon efficiency. On FPGAs, we minimize communication in two ways. Firstly, we propose a matrix powers kernel which avoids redundant computation at the expense of an increase in local communication within floating-point cores of the FPGA. This allows large $k$ and hence less communication with the off-chip memory. Secondly, we keep the resulting $k$ vectors entirely on-chip so that other blocks (BGS and QR factorization) avoid off-chip memory access and instead exploit high on-chip memory bandwidth to minimize computation time. To this end, we provide a resource-constrained framework which selects the maximum value of $k$ such that the resulting vectors can fit in the on-chip memory of the FPGAs and which minimizes overall runtime by providing a communication-computation tradeoff.

### 7.2.2 Composition

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

- Fully spatial architecture for each kernel [55].
- A dynamically reconfigurable architecture using full FPGA area for each kernel [95].
- 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 7.1 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 kernels and reconfiguration overhead of the second approach. 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 the 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.

## 7.3 Minimizing Communication For GPU

GPUs support two types of communication 1) data movement between global memory and shared memory or register file of a single streaming-multiprocessor (SM) 2) data sharing between different SMs through global memory. Although modern GPU architectures offer teraflops of raw computing power, their communication performance is at least an order of magnitude lower than their computational performance (see Table 3.1). As a result, there is very low silicon efficiency in accelerating communication-intensive iterative numerical algorithms. We briefly discuss the GPU implementations of the Lanczos Iteration and CA-Lanczos and then compare their performance.

### 7.3.1 Lanczos Iteration on GPU

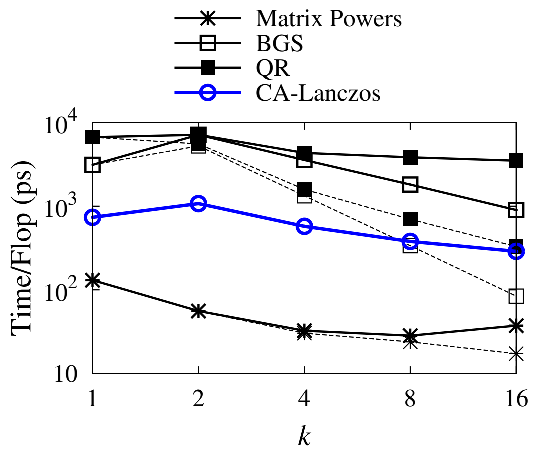
The Lanczos Iteration involves the SpMV kernel followed by vector-vector operations as shown in Figure 7.1(a). We use CUSP [17], an open-source library optimized for sparse linear algbera operations on GPU. The `spmv_dia_kernel` is used for SpMV, a routine which is optimized for banded matrices stored in compressed diagonal storage (CDS) format. For vector-vector operations, we use level 1 BLAS routines.

### 7.3.2 CA-Lanczos on GPU

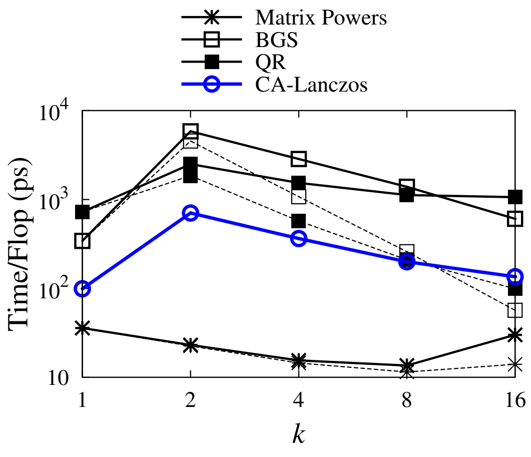
CA-Lanczos comprises three kernels as shown in Figure 7.1(b). The matrix powers kernel involves sparse linear algebra operations whereas the other two kernels (BGS and QR) involve dense operations. We use a highly optimized parallel matrix powers kernel implementation as discussed in Chapter 3. BGS involves multiplication of a short wide matrix ($\overline{Q_i}$) with a tall skinny matrix ($Q_{i-1}$). The generalized matrix-matrix multiplication routine `cublassgemm` from CuBLAS library suffers from kernel overhead and short vector effects [56] for matrices of this aspect ratio as the block sizes are optimized for square matrices. Therefore, we tune the block size of an open source `magmablas_sgemm` routine from the MAGMA [91] library and get a 4× speedup over `cublassgemm`. The QR factorization involves a tall-skinny matrix ($\overline{Q_i}$) and factorizing such matrices involves more communication than any other aspect ratio due to the sequential nature of the computation. We, therefore, use a communication-avoiding QR routine for GPUs [8] which provides up to 13× speedup over standard QR routines from GPU libraries like CULA [50].

### 7.3.3 Performance Analysis

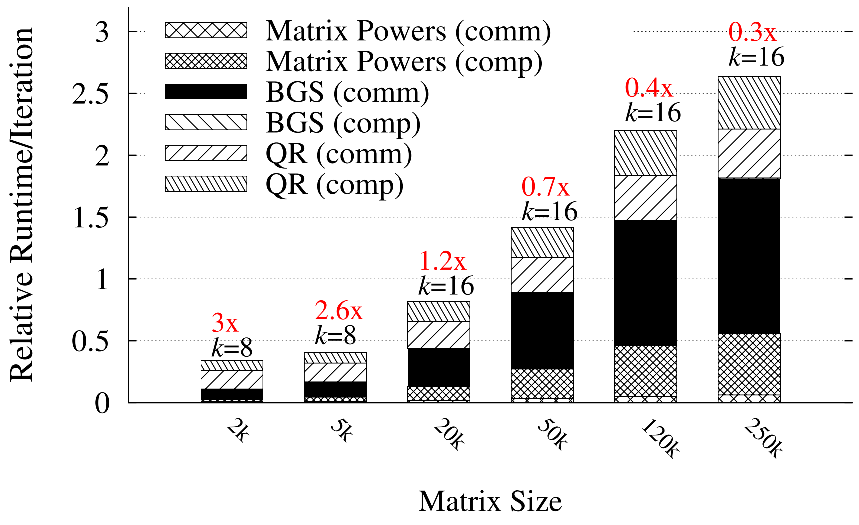
We use $\frac{time}{flop}$ as the metric to see whether we get any performance improvement using the communication-avoiding approach on GPUs. In Figure 7.3(a) and 7.3(b), we vary $k$ and show the performance considering useful operations (*i.e.* not counting the redundant computation) as well as actual 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 with a computational complexity of $O(nk^2)$. 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 (dashed lines) 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. 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 cost becomes the dominating factor due to low arithmetic intensity in these kernels as shown in Figure 7.3(c). Hence, composition of kernels on GPU is inefficient due to this off-chip sharing of data across the kernels. We, therefore, see 3×−0.3× speedup over the Lanczos Iteration from small to large problems as shown in Figure 7.3(c).



(a) Performance vs. $k$ ($n$ = 5k)



(b) Performance vs. $k$ ($n$ = 50k)



(c) Performance vs. $n$

Figure 7.3: CA-Lanczos performance analysis on Nvidia C2050. In Figure 7.3(a) and 7.3(b), we show how the algorithmic parameter $k$ is co-tuned for all the kernels. The speedup over the Lanczos Iteration ($k = 1$) is shown at the top of the bar in Figure 7.3(c) for a range of matrices with band size $b = 27$. The dashed lines in Figure 7.3(a) and 7.3(b) represent the cost for doing useful operations only.

## 7.4 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 3.1. So how we can use this on-chip capacity and memory bandwidth?

### 7.4.1 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 7.4.

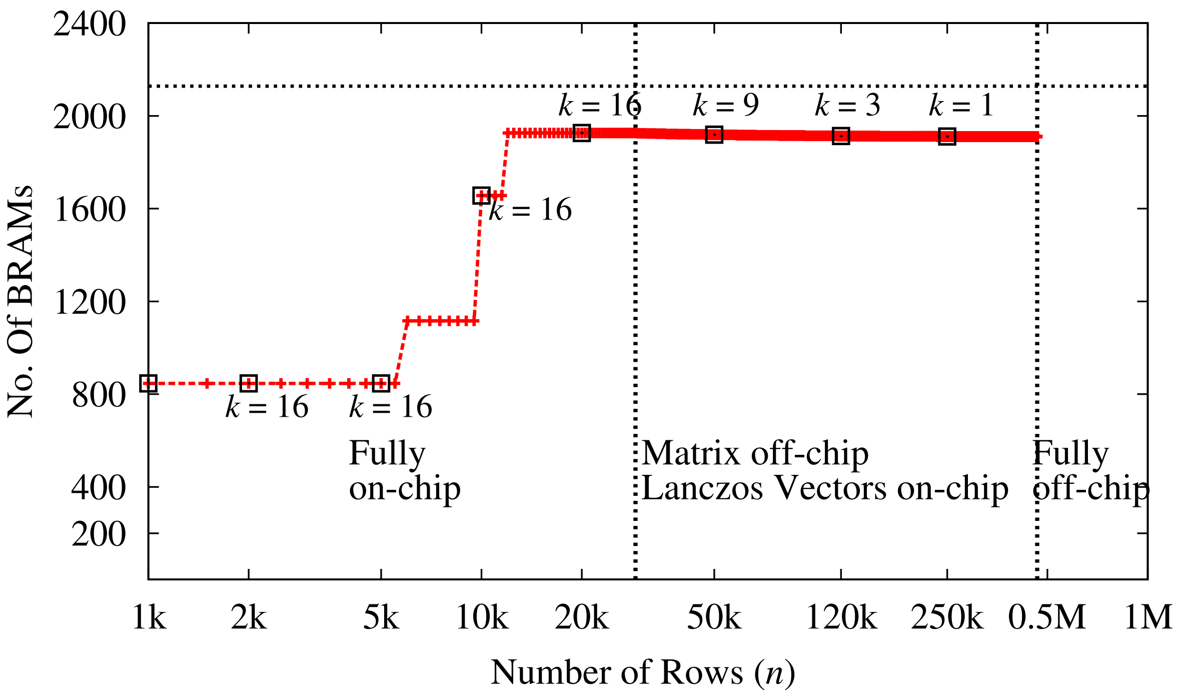


Figure 7.4: Three distinct scenarios for CA-Lanczos on Virtex6-SX475T (2128 18Kb BRAMs) for problems with band size $b = 27$. The maximum possible value of $k$ is shown for some matrix sizes.

As the parameter $k$ influences the decision where to store the data, we therefore select it carefully to optimize performance (see Section 7.4.5). From Figure 7.4, we observe that there is a wide range of matrices where we can keep 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 the 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 ($\sim 2\times$ on Virtex7), the range of matrices where Lanczos vectors can be stored on-chip will be pushed further.

### 7.4.2 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 application time itself. We design a unified architecture for basic linear algebra blocks identified in Table 7.1 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.

#### Basic Linear Algebra Subroutine (BLAS) Circuit

The GPU organizes everything in a simple data-parallel fashion ideally suitable for data-parallel $ax + y$ but inefficient for reduction operations like $x^T y$. We show the performance of these operations on GPU and FPGA in Figure 7.5 (assuming $x$ and $y$ are stored in global memory of GPU as in Section 7.3.2 and in on-chip memory of FPGAs as shown in Section 7.4.1). Using superior communication and on-chip memory bandwidth of the FPGAs, we design a high throughput architecture for these compute patterns in Figure 7.6 which we refer to as a Basic Linear Algebra Subroutine (BLAS) circuit.

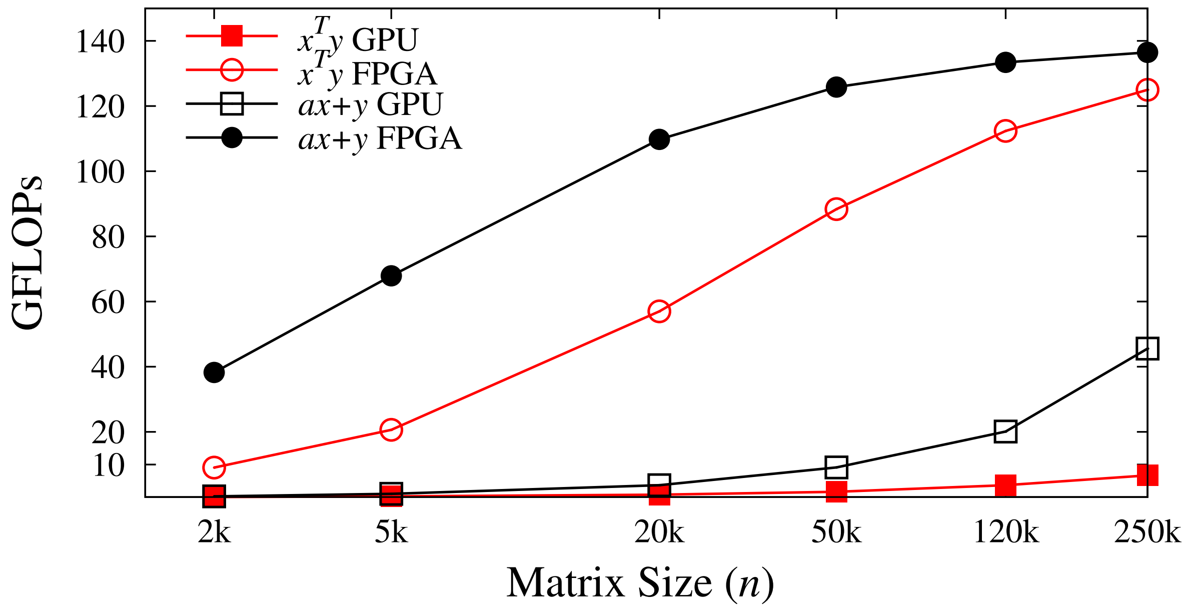


Figure 7.5: $x^T y$ and $ax + y$ on GPU and FPGA.

For $x$ and $y$ vectors of length $N$, both $ax+y$ and $x^T y$ have a sequential latency of $O(N)$ cycles. Our proposed architecture has $O(1)$ cycles latency for $ax + y$ and performs $x^T y$ in $O(\log N)$

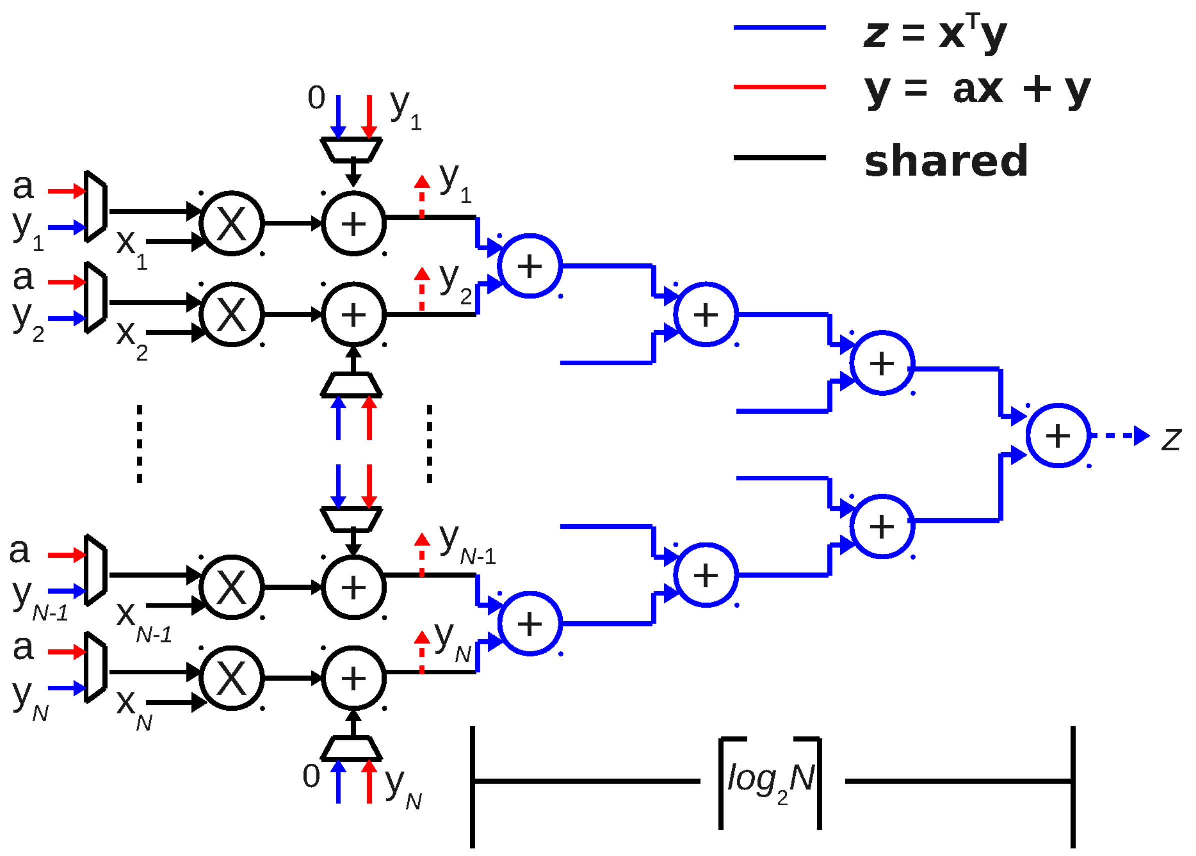


Figure 7.6: A BLAS Circuit for $z \leftarrow x^T y$ and y←a$x + y$ where $x$, $y \in \mathbb{R}^N$. The dotted arrow links show outputs.

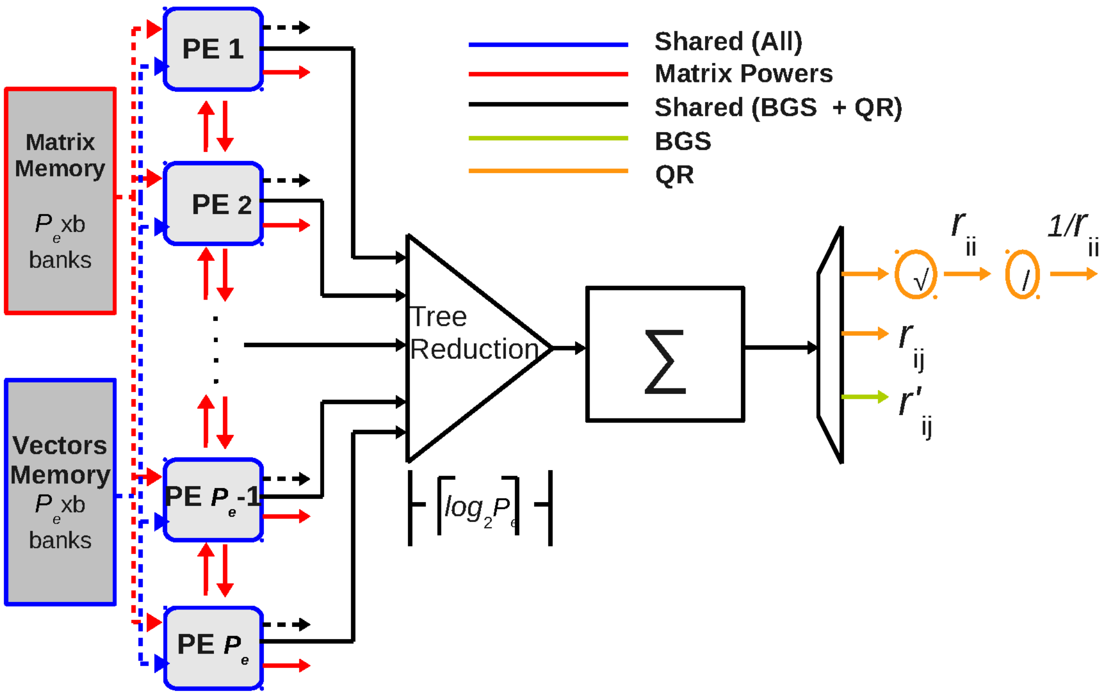
cycles. This is a highly efficient architecture with an initiation interval of one clock cycle, *i.e.* new set of inputs can be applied at every clock cycle.

### DataPath

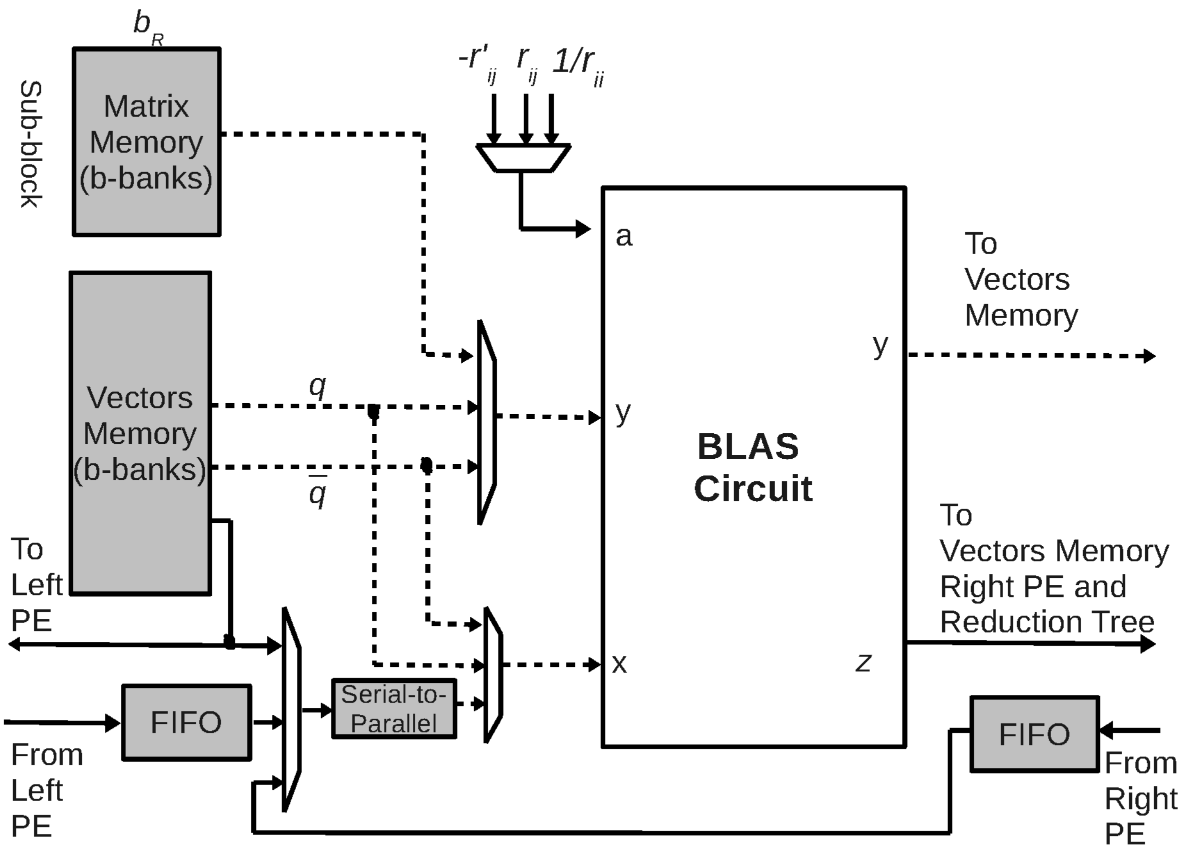
We arrange the BLAS circuits into a large tree reduction architecture as shown in Figure 7.7(a). We aim to minimize the time required in doing the reduction operation which is dominant in all the kernels. Each sub-tree operating on vectors of length $b$ is denoted as a processing element (PE) with its internal architecture shown in Figure 7.7(b). In order to provide the nearest neighbour communication required in mapping the matrix powers kernel compute graph shown in Figure 5.7, we also include left and right FIFOs each of length $\frac{b-1}{2}$ in each PE. The data from $P_e$ PEs is reduced by an adder reduction tree and finally accumulated to support an 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 7.2.

### 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 $N_b = \lceil \frac{n}{P_e b_R} \rceil$ where $P_e$ is



(a) A unified architecture for CA-Lanczos.



(b) Processing Element Design.

Figure 7.7: A time-multiplexed architecture for CA-Lanczos. The dotted lines show vector links of length $b$ whereas solid line represents scalars.

Table 7.2: Floating-Point Units for CA-Lanczos.

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

the number of PEs and $b_R$ is the number of rows in each sub-block. *Matrix Memory* as well as the *Vectors Memory* (for Lanczos vectors) is distributed across $P$ PEs with each PE accessing them as a bank of width $b$ as shown in Figure 7.7(b). The memory used in single-precision implementation of CA-Lanczos in terms of BRAMs (18kbit each) is given by

$$\textit{Matrix Memory} = P_e b \left\lceil \frac{32 b_R}{18 \times 1024} \right\rceil \tag{7.1}$$

$$\textit{Vectors Memory} = 2 P_e b \left\lceil \frac{32 \left\lceil \frac{n}{P_e b} \right\rceil (k+1)}{18 \times 1024} \right\rceil \tag{7.2}$$

$$\textit{FIFOs} = 2 P_e \left\lceil \frac{32(b-1)}{2 \times 18 \times 1024} \right\rceil \tag{7.3}$$

### 7.4.3 Compute Schedule

The architecture shown in Figure 7.7 implements CA-Lanczos in a time-multiplexed fashion. We show the compute schedule for CA-Lanczos in Algorithm 17 and highlight the blocks used in each kernel in Figure 7.7(a).

### 7.4.4 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 7.2. Using the compute schedule in Algorithm 17, we show the latencies (in cycles) of all the kernels in Table 7.3.

### 7.4.5 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 $P_e$ of PEs that can be synthesized within the FPGA for a given band size $b$.
- Find the memory bandwidth required to saturate these $P_e$ PEs. Partition the available on-chip memory such a way that Lanczos vectors can fit on-chip.

**Algorithm 17** Compute Schedule

**Matrix Powers Kernel**

**Step 1** : Load a block of the 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 3. 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 $N_b = \lceil \frac{n}{P_e b_R} \rceil$ 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 7.7(b). Compute dot product and accumulate the results from $P$ PEs.

**Step 4** : Go to Step 3 and load next $b$ components $\zeta = \lceil \frac{n}{P_e b} \rceil$ times until we reach at the end of the vectors. Compute all dot products at Line 2 of Algorithm 16. Save the accumulator output in $\overline{R}_i$.

**Step 5** : Compute Line 3 of Algorithm 7.1(b) by configuring each PE to compute $y \leftarrow ax+y$. Here $a$ 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.

Table 7.3: CA-Lanczos FPGA Analytical Performance Model.

| Kernel | Latency | Reference |
|---|---|---|
| Matrix Powers | $l_{BC} = l_M + l_A + l_A \lceil \log_2 b \rceil$<br>$l_{MP} = N_b k(b + l_{BC} + b_R - 1)$ | Figure 7.6 |
| BGS | $l_{BGS,1} = (k^2+1)(l_{BC} + \zeta + \ l_{red})$<br>$l_{BGS,2} = (k^2+1)(l_M + l_A + \zeta)$<br>$\zeta = \lceil \frac{n}{Pb} \rceil$, $l_{red} = \lceil \log_2 P_e \rceil + l_{acc}$ | Line 3 of Algorithm 16<br>Line 4 of Algorithm 16 |
| QR Factorization | $l_{QR,1} = (k+1)(l_{BC} + l_{red} + l_M$<br>$+ l_A + 2\zeta + l_S + l_D)$<br>$l_{QR,2} = k\frac{(k+1)}{2}(l_{BC} + l_{red} + l_M$<br>$+ l_A + 2\zeta)$ | Outer loop of QR<br><br>Inner loop of QR |
| CA-Lanczos | $L = l_{MP} + l_{BGS,1} + l_{BGS,2}$<br>$+ l_{QR,1} + l_{QR,2}$ | |

- Pick $k$ and $b_R$ based on the following constrained optimization problem

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

subject to

$$\begin{aligned} M(P_e,k,b_R) &\leq \text{FPGA}_{BRAMs} \\ R(P_e) &\leq \text{FPGA}_{Logic} \\ k &\leq 16 \\ k &\leq 2\frac{b_R}{b-1} \end{aligned} \tag{7.4}$$

Referring to Equation (7.4), $\frac{time(cycles)}{flop}$ is our objective we want to minimize. $L(k, P_e, b_R)$ corresponds to the total compute latency per CA-Lanczos iteration shown in Table 7.3 and $k(2nb + 7n)$ is the total number of flops in $k$ iterations of the Lanczos method. $M(P_e, k, b_R)$ is the number of BRAMs required and $R(P_e)$ is a vector containing the number of resources in terms of LUTs, FFs and DSP48Es from Table 7.2. $k \leq 16$ is an algorithmic constraint due to loss of orthogonality in CA-Lanczos [62]. The last constraint is to allow the partition size large enough to ensure only nearest neighbour communication at the boundaries of blocks in the matrix powers kernel shown in Figure 5.7.

## 7.5 Experimental Setup

The experimental setup comprises a Virtex6-SX475T FPGA and an Nvidia C2050 Fermi device with architectural features shown in Table 3.1. We use $\frac{time}{iteration}$ as our metric relative to the Lanczos Iteration 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. For FPGA, we implemented a set of 3 PEs with their memory subsystems and FIFO connections. 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 actually calculate the total number of DSP48Es, LUTs and FFs which are required for a single PE and then estimate the total number of PEs that can be synthesized on a given FPGA device. We show the resource utilization for different band sizes in Table 7.4. 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 7.4: FPGA Resource Utilization (Estimated).

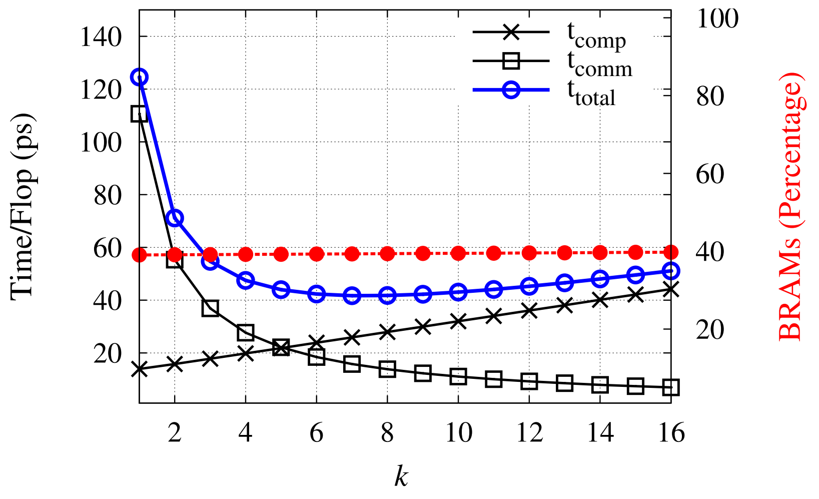
| **Band Size** | **DSP48Es (%)** | **LUTs (%)** | **FFs (%)** | **PEs** |
|---|---|---|---|---|
| 3 | 99.6 | 70.3 | 41.3 | 95 |
| 9 | 97.5 | 68.8 | 40.4 | 31 |
| 27 | 94.3 | 66.6 | 39.0 | 10 |

## 7.6 Evaluation

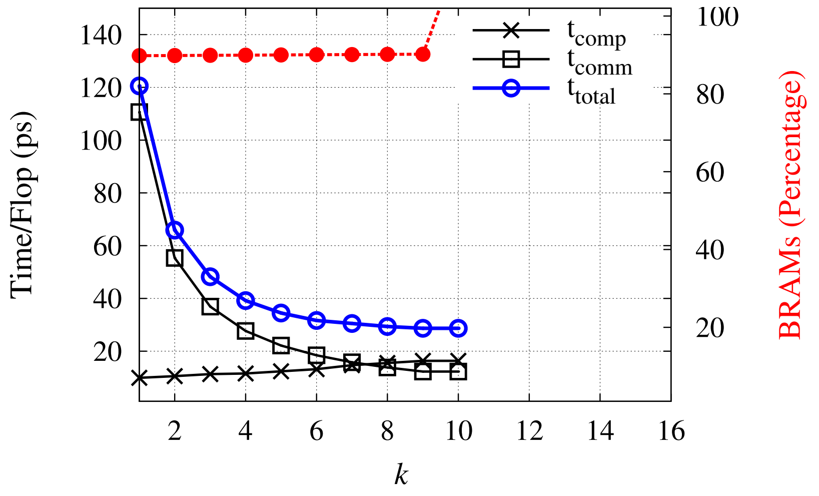
We first evaluate the impact of $k$ on FPGA performance and then compare the results of CA-Lanczos on FPGA and GPU.

### 7.6.1 Impact of $k$ on FPGA Performance

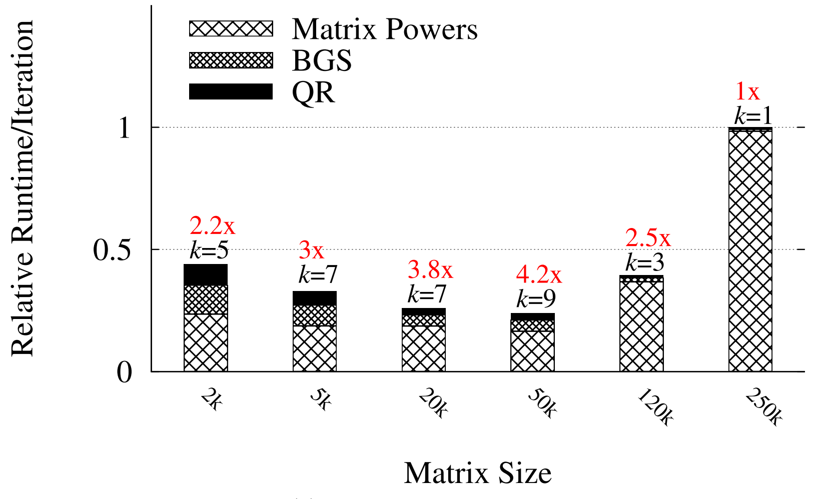
We show $\frac{time}{flop}$ for computation ($t_{comp}$) and communication ($t_{comm}$) separately in Figure 7.8. We select the value of $k$ which minimizes total time, *i.e.* $t_{total}=t_{comp}+t_{comm}$. Ideally, we should see a reduction in this cost by $k$ as the matrix is fetched only once to generate $k$ vectors. However from Figure 7.8(a), we see the total cost decreases until $k = 7$ and then it increases due to



(a) Performance vs. $k$, ($n$ =5k)



(b) Performance vs. $k$, ($n$ =50k)



(c) Performance vs. $n$

Figure 7.8: CA-Lanczos performance analysis on FPGA. The value of $k$ is selected using the resource-constrained framework in Figure 7.8(a) and Figure 7.8(b). The speedup over the Lanczos method ($k = 1$) on FPGA is shown in Figure 7.8(c) for a range of matrices with the band size $b = 27$.

computation cost ($O(nk^2)$ work in BGS and QR). This is the optimal value of $k$ which provides a ∼3× performance improvement over the standard Lanczos Iteration ($k = 1$). In Figure 7.8(b), the value of $k$ is restricted due to BRAMs as beyond $k$ =9 the vectors can no longer 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×−4.2× speedup over the FPGA-based Lanczos Iteration shown in Figure 7.8(c).

### 7.6.2 Performance Comparison with GPU

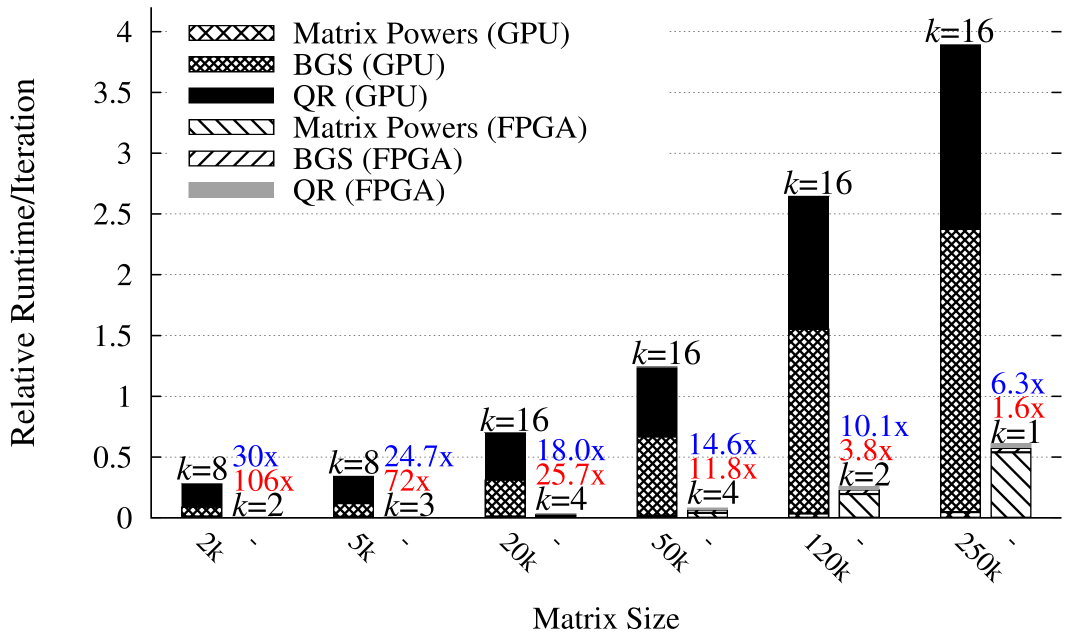
We compare the performance of our proposed design with a GPU in Figure 7.9 for $b = 3$ and $b = 27$, showing the runtime breakdown of each kernel. From these results, we find that the matrix powers kernel on the GPU is efficient compared to the FPGA because of the GPUs ∼5× larger off-chip bandwidth to fetch the matrix $A$. However, the communication-avoiding approach is not as useful on the GPU as it is on the FPGA for two reasons. First the composition of kernels requires 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 streaming multiprocessors (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 7.3.2). 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 well suited on GPUs [56]. As a result, CA-Lanczos on the GPU is up to ∼3× slower than the standard Lanczos Iteration.

On the other hand, as the vectors are stored on-chip in FPGAs, our architecture expliots high on-chip bandwidth 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 the FPGA is 1.6× as fast as the standard Lanczos Iteration from CUSP. However, for large matrix and band sizes, the problem becomes communication-bound as the vectors can no longer be stored in the on-chip memory of the FPGA and therefore we see that FPGA is ∼3× slower than the GPU.

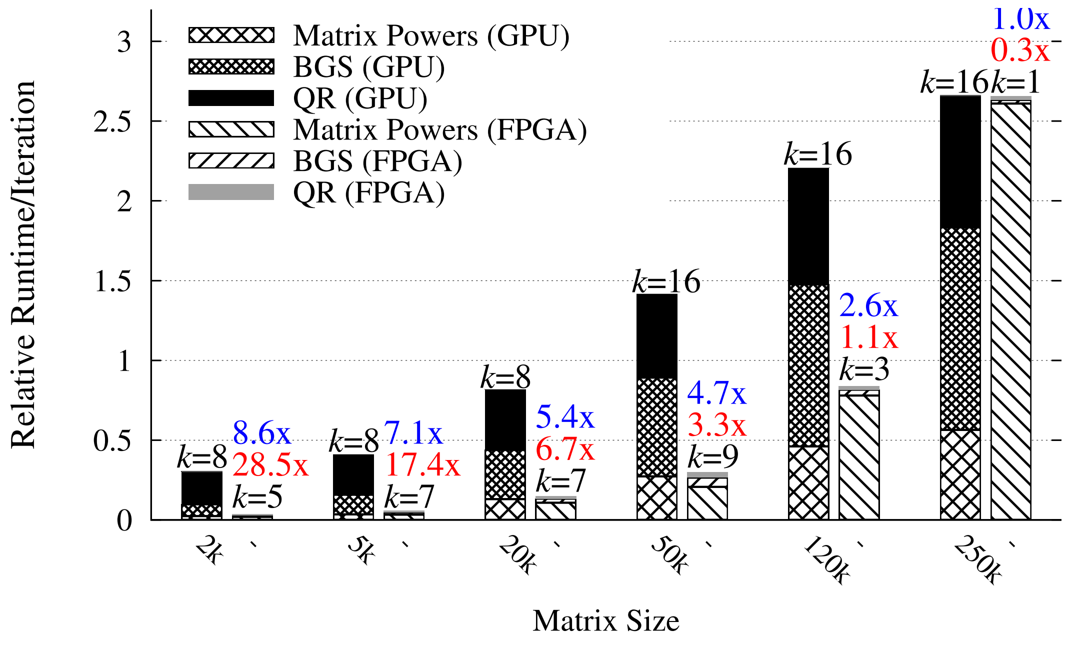
We finally compare the silicon efficiency of FPGA and GPU in Figure 7.10. For the GPU, we see CA-Lanczos is even worse than the standard Lanczos Iteration except for small problems where the communication cost of BGS and QR is relatively low as shown in Figure 7.9. For the FPGA, for small band sizes, there is very little improvement in efficiency due to small value of $k$. However, for larger band size in Figure 7.10(b), we see up to 8% efficiency compared to 2% with the standard Lanczos Iteration. This is only possible due to relatively large values of $k$. With future FPGA devices having relatively large on-chip memory capacity, we can achieve much higher efficiency due to large values of $k$ as the vectors can be stored on-chip.

## 7.7 Summary

Communication-avoiding algorithms are an alternative to multi-level cache hierarchy where we trade communication with redundant computation. In current GPU architectures, composition

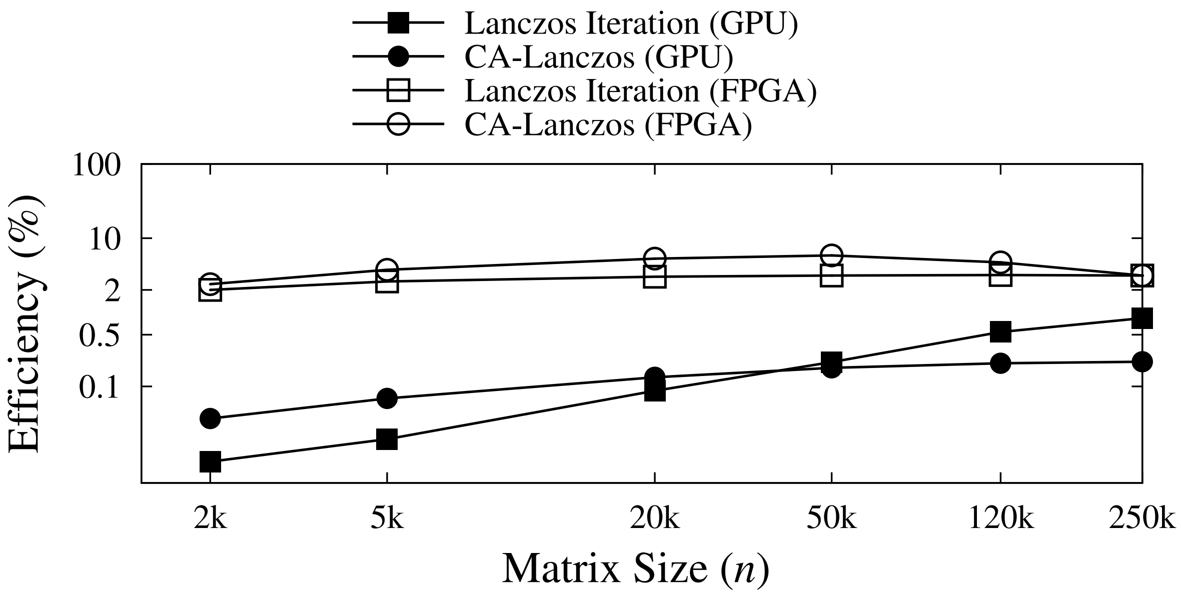


(a) band size $b = 3$

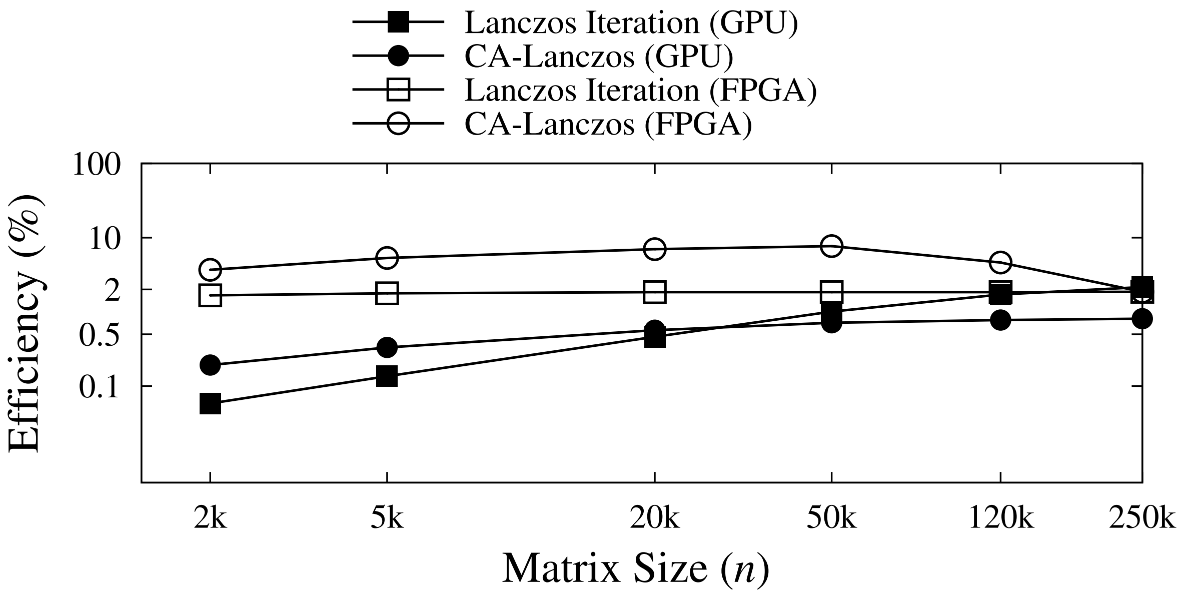


(b) band size $b = 27$

Figure 7.9: 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.



(a) band size $b$ =3



(b) band size $b$ =27

Figure 7.10: Silicon efficiency of FPGA and GPU for iterative numerical algorithms.

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 the 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 select the algorithmic parameter to optimally trade communication with redundant computation. Using CA-Lanczos as a case study, we show up to 4.2× performance improvement over vanilla algorithm on the 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 cannot store data on-chip, we see a ∼3× slow down as the problem becomes communication-bound.

# 8 Conclusion

This thesis has presented techniques to improve silicon efficiency of FPGAs and GPUs in accelerating communication-bound sparse iterative solvers. In this chapter we summarize the major contributions in this thesis and present some research directions as future work.

Different approaches have been used to minimize communication in iterative numerical algorithms for small-to-medium size dense data sets and large structured banded data sets, which arise in different application settings such as solving semi-definite optimization problems. It is a common myth that GPUs with their higher peak floating-point capabilities are highly suitable for accelerating dense linear algebra. While this is true for compute-bound problems like matrix-matrix multiplication, we show that for communication-bound matrix-vector multiplication, a dominant operation in iterative numerical algorithms, the performance is bounded from above by the off-chip memory bandwidth of GPUs. This is also true for the FPGAs. However, we use explicit cache blocking to load the matrices in the relatively large on-chip memory of the FPGAs and re-use it for all iterations thereby minimize communication with the off-chip memory. This explicit cache blocking allows us to utilize higher on-chip memory bandwidth to saturate the floating-point cores of the FPGA, which leads to higher silicon efficiency. We therefore conclude that iterative numerical algorithms operating on small-to-medium size data sets requiring high memory bandwidth are highly appropriate for acceleration using FPGAs.

For large dense problems that do not fit on-chip, the performance of standard iterative numerical algorithms is still bounded by the off-chip memory bandwidth. However, for sparse problems, algorithmic transformations have been recently proposed, which trade communication with redundant computation. The matrix powers kernel is an important kernel in this approach, which performs $k$ SpMVs at the communication cost of a single SpMV. We proposed an architecture-aware algorithm for the FPGAs, which avoids redundant computation to allow large values of $k$ and hence higher speed ups. Using a predictive model, we show that by careful selection of the algorithmic parameter, we can increase the silicon efficiency of FPGAs and GPUs which otherwise require significant architectural modifications.

There is extremely low silicon efficiency of multi-cores and GPUs in tall-skinny QR factorization (TSQR), a kernel that arises in communication-avoiding iterative algorithms and many other areas of science and engineering. We identify that on GPUs this is due to low arithmetic intensity caused by limited registers, low occupancy during the merge stage, and global communication during the merge stage whereas in case of multi-cores it is primarily due to low memory bandwidth. We show how we can exploit the high on-chip bandwidth of the FPGA to design a high-throughput architecture for QR factorization and large on-chip capacity to perform the merge stage without any global communication. We conclude that even though GPU has 3$\times$ higher peak double-precision floating-point performance, by exploiting the architectural features of FPGAs we can outperform the GPU for communication-avoiding linear algebra algorithms like TSQR.

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 the communication-avoiding approach. In FPGAs, by explicitly sharing data across kernels using on-chip memory and designing an architecture which exploits the on-chip memory and communication bandwidth, we keep the computation cost as low as possible. Using the CA-Lanczos as a case study, we demonstrate that FPGAs have better support for composing different kernels due to relatively large on-chip memory. This leads to superior performance over GPU even though FPGAs have lower peak floating-point performance.

From architectural perspective, in order to increase silicon efficiency in communication-bound operations like the iterative numerical algorithms, we believe that future FPGA and GPU architectures might require some modifications. First, instead of having a shared memory architecture in GPU where different SMs can only communicate through global off-chip memory, an on-chip interconnect, *e.g.* a ring network in Intel Xeon Phi Co-processor, can lead to better silicon efficiency. Secondly, the shared memory of each SM needs to be increased like a large on-chip L2 cache in modern multi-cores. Thirdly, there should be architectural support for on-chip sharing of data across different kernels. In case of FPGAs, besides increasing on-chip memories, the off-chip memory bandwidth needs to be increased much like GPUs and Intel Xeon Phi Co-processor.

Lastly, even with advancements in hardware technology, architecture-aware linear algebra algorithms need to be designed in order to exploit the full potential of a particular architecture. We have demonstrated this using our hybrid matrix powers kernel for the FPGAs. We therefore conclude that an inter-disciplinary research is required involving computing and numerical linear algebra community to increase silicon efficiency of modern computing platforms.

## 8.1 Future Work

### 8.1.1 Large Sparse Problems

The work in this thesis can be extended 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 [5] where we have to solve $\mathrm{A}x = b$ 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 compared to the actual computation.

### 8.1.2 Low Power Applied Linear Algebra

In this thesis, the focus is on high performance computing. However, since we minimized communication with the off-chip memory, a major source of power consumption, it needs to be investigated how such a technique can reduce overall power consumption. This will help in reducing energy footprint of data centers, *e.g.* a low power tall-skinny QR factorization can reduce power consumption in Singular Value Decomposition (SVD), which is an important kernel used in Latent Semantic Indexing (LSI) [36], a ubiquitous operation for information retrieval in large data centers.

### 8.1.3 Parameterizable Tall-Skinny QR Factorization

The hardware core presented in this thesis for the tall-skinny QR factorization takes an input matrix with variable number of rows but fixed number of columns. Although this meets the requirement in applications such as iterative numerical algorithms, there are other applications where we require QR factorization of a tall-skinny matrix with variable number of columns. An example for such a QR factorization arises in Monte Carlo Markov Chain (MCMC) based data association for real-time target tracking [57].

### 8.1.4 Fixed-Point QR Factorization

In this thesis, we have only used single-precision or double-precision floating-point number presentation. These are the only two number representations available with multi-cores and GPUs, but FPGAs can be customized for any arbitrary precision. There is some recent work on fixed-point Lanczos Iteration [52] where all the variables are bounded using a novel preconditioner and sustained TFLOPs performance is demonstrated. One can extend this work for a fixed-point QR factorization to improve performance for a given orthogonalization error $||Q^TQ - I||_2$.

# Bibliography

[1] SMCP benchmarks. `http://abel.ee.ucla.edu/smcp/benchmarks/index.html`, 2010.

[2] Xilinx DS816 Floating-Point Operator v6.0. `http://www.xilinx.com/support/documentation/ip_documentation/floating_point/v6_0/ds816_floating_point.pdf`, 2012.

[3] E. Agullo, J. Demmel, J. Dongarra, B. Hadri, J. Kurzak, J. Langou, H. Ltaief, P. Luszczek, and S. Tomov. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. In *Journal of Physics: Conference Series*, volume 180. IOP Publishing, 2009.

[4] A. Ahmedsaid, A. Amira, and A. Bouridane. Improved SVD systolic array and implementation on FPGA. In *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT)*, pages 35–42. IEEE, 2003.

[5] F. Alizadeh, J. A. Haeberly, and M. L. Overton. Primal-dual interior-point methods for semidefinite programming: convergence rates, stability and numerical results. *SIAM Journal on Optimization*, 8(3):746–768, 1998.

[6] AMD. Core Math Library (ACML). http://developer.amd.com/wordpress/media/2013/05/acml.pdf, 2012.

[7] E. Anderson, Z. Bai, C. Bischof, L. S. Blackford, J. Demmel, J. Dongarra, J. Du. Croz, A. Greenbaum, S. Hammarling, A. Mckenney, and D. Sorensen. *LAPACK Users' guide.* Society for Industrial and Applied Mathematics, third edition, 1999.

[8] M. Anderson, G. Ballard, J. Demmel, and K. Keutzer. Communication-avoiding QR decomposition for GPUs. In *Proceedings of the 25th IEEE Interantional Parallel and Distributed Processing Symposium (IPDPS)*, pages 48–58, 2011.

[9] M. J. Anderson, D. Sheffield, and K. Keutzer. A predictive model for solving small linear algebra problems in GPU registers. In *Proceedings of the 26th IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 2–13, 2012.

[10] H. Anton. *Elementary linear algebra.* Wiley, 2010.

[11] H. Anzt, T. Hahn, V. Heuveline, and B. Rocker. *GPU Accelerated Scientific Computing: Evaluation of the NVIDIA Fermi Architecture; Elementary Kernels and Linear Solvers.* KIT, 2010.

[12] Z. Bai, J. Demmel, J. Dongarra, A. Ruhe, and H.v.d Vorst. *Templates for the solution of algebraic eigenvalue problems*. Society for Industrial and Applied Mathematics, 2000.

[13] Z. K. Baker and V. K. Prasanna. Efficient hardware data mining with the Apriori algorithm on FPGAs. In *Proceedings of the 13th Annaual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 3–12, 2005.

[14] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proceedinbgs of the 24th ACM symposium on Parallelism in Algorithms and Architectures*, pages 193–204. ACM, 2012.

[15] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz. Minimizing communication in numerical linear algebra. *SIAM Journal on Matrix Analysis and Applications*, 32(3):866–901, 2011.

[16] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. V. Vorst. *Templates for the solution of linear systems: building blocks for iterative methods*. Society for Industrial and Applied Mathematics, 1987.

[17] N. Bell and M. Garland. CUSP: Generic parallel algorithms for sparse matrix and graph computations. http://code. google. com/p/cusp-library, 2009.

[18] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller, et al. Exascale computing study: Technology challenges in achieving exascale systems. Technical report, Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), 2008.

[19] Å. Björck. *Numerical methods for least squares problems*. Society for Industrial and Applied Mathematics, 1996.

[20] D. Boland and G. A. Constantinides. Optimizing memory bandwidth use and performance for matrix-vector multiplication in iterative methods. *ACM Transactions on Reconfigurbale Technology and Systems (TRETS)*, 4(3):22:1–22:14, August 2011.

[21] B. Borchers. SDPLIB 1.2, a library of semidefinite programming test problems. *Optimization Methods and Software*, 11(1):683–690, 1999.

[22] S. Boyd and L. Vandenberghe. *Convex Optimization*. Cambridge University Press, New York, NY, USA, 2004.

[23] I. Bravo, P. Jiménez, M. Mazo, J. L. Lázaro, and A. Gardel. Implementation in FPGAs of Jacobi method to solve the eigenvalue and eigenvector problem. In *Proceedings of IEEE International Conference on Field Programmable Logic and Applications*, pages 1–4, 2006.

[24] A. Buttari, J. Langou, Kurzak J, and J. Dongarra. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Computing*, 35(1):38–53, 2009.

[25] H. S. Chen, W. Gao, and D. G. Daut. Signature based spectrum sensing algorithms for IEEE 802.22 WRAN. In *Proceedings of the IEEE International Conference on Communications*, pages 6487–6492, 2007.

[26] G. Chrysos. Intel® Xeon Phi coprocessor (codename Knights corner). In *Proceedings of the 24th Hot Chips Symposium, HC*, 2012.

[27] Intel Corporation. Intel microprocessor export compliance metrics. `http://download.intel.com/support/processors/xeon/sb/xeon_5600.pdf`, 2010.

[28] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. Vonf Eicken. *LogP: Towards a realistic model of parallel computation*, volume 28. ACM, 1993.

[29] J. Cullum and W. E. Donath. A block Lanczos algorithm for computing the q algebraically largest eigenvalues and a corresponding eigenspace of large, sparse, real symmetric matrices. In *Proceedings of the IEEE Conference on Decision and Control including the 13th Symposium on Adaptive Processes*, volume 13, pages 505–509. IEEE, 1974.

[30] 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 the ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[31] M. Delorimier and A. DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *Proceedings of the 13th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 75–85, 2005.

[32] M. Delorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. Uribe, T. F. Knight, and A. DeHon. GraphStep: A system architecture for sparse-graph algorithms. In *Proceedings of the 14th Annaual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 143–151, 2006.

[33] J. Demmel, M. Hoemmen, M. Mohiyuddin, and K. Yelick. Avoiding communication in sparse matrix computations. In *Proceedings of the 22nd IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, pages 1–12, 2008.

[34] J. W. Demmel. *Applied Numerical Linear Algebra.* Society for Industrial and Applied Mathematics, 1997.

[35] J. J. Dongarra. *LINPACK Users' Guide.* Society for Industrial and Applied Mathematics, 1979.

[36] S. Dumais, G. Furnas, T. Landauer, S. Deerwester, et al. Latent semantic indexing. In *Proceedings of the Second Text Retrieval Conference*, pages 271–289, 1995.

[37] J. Fang, A. L. Varbanescu, H. Sips, L. Zhang, Y. Che, and C. Xu. Benchmarking Intel Xeon Phi to Guide Kernel Design. Technical Report PDS-2013-005, Delft University of Technology, 2013.

[38] M. J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, 1966.

[39] G. H. Golub and C. F. Van Loan. *Matrix Computations.* The Johns Hopkins University Press, third edition, 1996.

[40] G. H. Golub and R. Underwood. The block Lanczos method for computing eigenvalues. *Mathematical software*, 3:361–377, 1977.

[41] S. L. Graham, M. Snir, and C. A. Patterson. *Getting up to speed: The future of supercomputing.* National Academies Press, 2004.

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

[43] J. L. Gross and J. Yellen. *Handbook of graph theory.* CRC press, 2003.

[44] B. Hadri, H. Ltaief, E. Agullo, and J. Dongarra. Enhancing Parallelism of Tile QR Factorization for Multicore Architectures. *Matrix*, 2(4):8, 2010.

[45] B. Harris, A. C. Jacob, J. M. Lancaster, J. Buhler, and R. D. Chamberlain. A banded Smith-Waterman FPGA accelerator for Mercury BLASTP. In *Proceedings of the IEEE International Conference on Field Programmable Logic and Applications*, pages 765–769, 2007.

[46] J. L. Hennessy and D. A. Patterson. *Computer architecture: a quantitative approach.* Elsevier, fourth edition, 2012.

[47] C. Hirsch. *Numerical Computation of Internal and External flows: The Fundamentals of Computational Fluid Dynamics.* Butterworth-Heinemann, second edition, 2007.

[48] M. Hoemmen. *Communication-avoiding Krylov subspace methods.* PhD thesis, University of California at Berkeley, Berkeley, USA, 2010.

[49] S. V. Huffel and J. Vandewalle. *The total least squares problem: computational aspects and analysis*, volume 9. Society for Industrial and Applied Mathematics, 1991.

[50] J. R. Humphrey, D. K. Price, K. E. Spagnoli, A. L. Paolini, and E. J. Kelmelis. CULA: hybrid GPU accelerated linear algebra routines. In *Proceedings of SPIE*, pages 770502–770507. International Society for Optics and Photonics, 2010.

[51] Intel. Intel math kernel library. http://software.intel.com/sites/products/documentation/ // hpc/userguides/mkl_userguide_lnx.pdf, 2007.

[52] J. L. Jerez, , G. A. Constantinides, and E. C. Kerrigan. Fixed Point Lanczos: Sustaining TFLOP-equivalent Performance in FPGAs for Scientific Computing. In *Proceedings of 20th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 53–60. IEEE, 2012.

[53] I. Jolliffe. *Principal Component Analysis.* Wiley Online Library, 2005.

[54] J. Kahle. The Cell Processor Architecture. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 3–3. IEEE Computer Society, 2005.

[55] N. Kapre and 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*, 31(1):9–22, 2012.

[56] S. Kestur, J. D. Davis, and O. Williams. BLAS comparison on FPGA, CPU and GPU. In *Proceedings of IEEE Computer Society Annual Symposium on VLSI (IVLSI)*, pages 288–293, 2010.

[57] Z. Khan, T. Balch, and F. Dellaert. MCMC data association and sparse factorization updating for real time multitarget tracking with merged and multiple measurements. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 28(12):1960–1972, 2006.

[58] S. K. Kim and A. T. Chronopoulos. A class of Lanczos-like algorithms implemented on parallel computers. *Parallel Computing*, 17(6):763–778, 1991.

[59] K. R. Liu, S. F. Hsieh, and K. Yao. Systolic block Householder transformation for RLS algorithm with two-level pipelined implementation. *IEEE Transactions on Signal Processing*, 40(4):946–958, 1992.

[60] Y. Liu, C. S. Bouganis, P. Y. K. Cheung, P. H. W. Leong, and S. J. Motley. Hardware efficient architectures for eigenvalue computation. In *Proceedings of Design, Automation and Test in Europe*, volume 1, pages 1–6. IEEE, 2006.

[61] A. R. Lopes and G. A. Constantinides. A high throughput FPGA-based floating point conjugate gradient implementation. In *Proceedings of Reconfigurable Computing: Architectures, Tools and Applications*, pages 75–86. Springer, 2008.

[62] G. Magnus, J. Demmel, and S. Holmgren. Numerical evaluation of the communication-avoiding Lanczos algorithm. Technical Report 2012-001, Department of Information Technology, Uppsala University, Uppsala, Sweden, 2012.

[63] A. Majumdar, A. A. Ahmadi, and R. Tedrake. Control design along trajectories with sums of squares programming. In *Proceedings of the IEEE International Conference on Robotics and Automation (ICRA)*, 2013.

[64] D. T. Marr. Hyper-threading technology architecture and microarchitecture: a hyperhtext history. *Intel Technology Journal*, 2002.

[65] M. Mohiyuddin, M. Hoemmeno, J. Demmel, and K. Yalick. Minimizing communication in sparse matrix solvers. In *Proceedings of the 21st ACM Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, 2009.

[66] D. Molka, D. Hackenberg, R. Schone, and M. S. Muller. Memory performance and cache coherency effects on an Intel Nehalem multiprocessor system. In *Proceedings of 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 261–270. IEEE, 2009.

[67] G. E. Moore et al. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, 1998.

[68] J. V. Neumann. First Draft of a Report on the EDVAC. *Annals of the History of Computing, IEEE*, 15(4):27–75, 1993.

[69] R. Nishtala, R. W. Vuduc, J. Demmel, and K. Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, 2007.

[70] Nvidia. Compute Visual Profiler. `http://developer.download.nvidia.com/compute/cuda/3_2_prod/toolkit/docs/VisualProfiler/Compute_Visual_Profiler_User_Guide.pdf`, 2010.

[71] Nvidia. TESLA C2050 / C2070 GPU computing processor, supercomputing at 1/10th the cost. `http://www.nvidia.co.uk/docs/IO/43395/NV_DS_Tesla_C2050_C2070_jul10_lores.pdf`, 2010.

[72] Nvidia. CuBLAS library. `https://developer.nvidia.com/cublas`, 2011.

[73] Nvidia. CUSPARSE library. `https://developer.nvidia.com/cusparse`, 2011.

[74] A. F. Peterson, S. L. Ray, and R. Mittra. *Computational methods for electromagnetics*, volume 24. IEEE Press, New York, 1998.

[75] A. Rafique, G. A. Constantinides, and N. Kapre. Communication optimization of iterative sparse matrix-vector multipy on GPUs and FPGAs. *Submitted in Transactions on Parallel and Distributed Systems*, 2013.

[76] A. Rafique, N. Kapre, and G. A. Constantinides. Enhancing performance of Tall-Skinny QR factorization using FPGAs. In *Proceedings of the 22nd IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 443–450, 2012.

[77] A. Rafique, N. Kapre, and G. A. Constantinides. A high throughput FPGA-Based implementation of the Lanczos Method for the symmetric extremal eigenvalue problem. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 239–250. Springer, 2012.

[78] A. Rafique, N. Kapre, and G. A. Constantinides. Application composition and communication optimization in iterative solvers using FPGAs. In *Proceedings of the 21st IEEE International Conference on Field-Programmable Custom Computing Machines (FCCM)*, 2013.

[79] B. D. Ruijsscher, G. N. Gaydadjiev, J. Lichtenauer, and E. Hendriks. FPGA accelerator for real-time skin segmentation. In *Proceedings of the IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia*, pages 93–97, 2006.

[80] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM Journal on Scientific Computing and Statistical Computing*, 7(3):856–869, 1986.

[81] A. Sahai and D. Cabric. Spectrum sensing: fundamental limits and practical challenges. In *Proceedings of the IEEE International Symposium on New Frontiers in Dynamic Spectrum Access Networks (DySPAN)*, November 2005.

[82] L. L. Scharf. *Statistical Signal Processing*, volume 98. Addison-Wesley, 1991.

[83] R. Schreiber and C. V. Loan. A storage-efficient WY representation for products of Householder transformations. *SIAM Journal on Scientific and Statistical Computing*, 10(1):53–57, 1989.

[84] G. Sewell. *The numerical solution of ordinary and partial differential equations.* Wiley-InterScience, 2005.

[85] R. U. Seydel. *Tools for computational finance.* Springer, fifth edition, 2012.

[86] M. M. Strout, L. Carter, and J. Ferrante. Rescheduling for Locality in Sparse Matrix Computations. In *Proceedings of the International Conference on Computational Science, Lecture Notes in Computer Science*, pages 28–30. Springer-Verlag, 2001.

[87] P. Sundararajan. High Performance Computing Using FPGAs. *WP (Xilinx): WP375 (v1. 0)*, 10:105, September 2010.

[88] Y.G. Tai, K. Psarris, and C. T. D. Lo. Synthesizing Tiled Matrix Decomposition on FPGAs. In *Proceedings of IEEE International Conference on Field Programmable Logic and Applications (FPL)*, pages 464–469, 2011.

[89] K. C. Toh. A note on the calculation of step-lengths in interior-point methods for semidefinite programming. *Computational Optimization and Applications*, 21(3):301–310, 2002.

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

[91] S. Tomov, J. Dongarra, V. Volkov, and J. Demmel. MAGMA library. http://icl.cs.utk.edu/magma/software/, 2009.

[92] H. Urkowitz. Energy detection of unknown deterministic signals. *Proceedings of the IEEE*, 55(4):523–531, 1967.

[93] L. Vandenberghe and V. Balakrishnan. Algorithms and software for LMI problems in control. *IEEE Control Systems Magazine*, 17(5):89–95, 1997.

[94] L. Vandenberghe and S. Boyd. Semidefinite Programming. *SIAM Review*, 38(1):49–95, 1996.

[95] J. Villasenor, B. Schoner, K. N. Chia, C. Zapata, H. J. Kim, C. Jones, S. Lansing, and B. M. Smith. Configurable computing solutions for automatic target recognition. In *Proceedings of the IEEE International Symposium on Field-Programmable Custom Computing Machines*, pages 70–79, 1996.

[96] V. Volkov. Better performance at lower occupancy. In *Proceedings of the GPU Technology Conference, GTC*, volume 10, 2010.

[97] V. Volkov and J. Demmel. Benchmarking GPUs to tune dense linear algebra. In *Proceedings of the ACM/IEEE Conference on Supercomputing*, pages 31:1–31:11. IEEE Press, 2008.

[98] H. Waki, S. Kim, M. Kojima, M. Muramatsu, and H. Sugimoto. Algorithm 883: SparsePOP—a sparse semidefinite programming relaxation of polynomial optimization problems. *ACM Transactions on Mathematical Software (TOMS)*, 35(2):1–13, 2008.

[99] S. Weisberg. *Applied Linear Regression*, volume 528. Wiley, 2005.

[100] J. H. Wilkinson. *The algebraic eigenvalue problem.* Oxford University Press, USA, 1988.

[101] S. Williams, L. Oliker, R. Vuduc, J. Shalf, K. Yelick, and J. Demmel. Optimization of sparse matrix–vector multiplication on emerging multicore platforms. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, SC' 07, pages 1–12, New York, NY, USA, 2007. ACM.

[102] M. M. Wolf, E. G. Boman, and B. Hendrickson. Optimizing parallel sparse matrix-vector multiplication by corner partitioning. *PARA08, Trondheim, Norway*, 2008.

[103] A. W. Wulf and A. S. Mckee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH computer architecture news*, 23(1):20–24, 1995.

[104] Y. Zeng, C. L. Koh, and Y. C. Liang. Maximum eigenvalue detection: theory and application. In *Proceedings of the IEEE International Conference on Communications*, pages 4160–4164, 2008.

# Glossary

**basis vectors** basis vectors are orthogonal vectors that can span a whole subspace, *e.g.* [1,0,0], [0,1,0] and [0,0,1] are the basis vectors for spanning a three dimensional space. 42

**extremal** either maximum or minimum value (not the magnitude). 21

**Gram-Schmidt Orthogonalization** a technique to orthogonalize two vectors $x$ and $y$ such that $x^T y = 0$, two operations are required: $a = x^T y$ and then $y = y - ax$. 20

**QR factorization** decompose an $m \times n$ matrix $A$ into an $m \times m$ orthogonal matrix $Q$ and a $m \times n$ upper triangular matrix $R$. 20

**Silicon Efficiency** a ratio of sustained performance to the peak performance of a particular architecture. 18

**SIMD** Single Instruction Multiple Data is a compute architecture where all processors are executing a same instruction on their own portion of the data. 54

**spectrum** all eigenvalues of a matrix. 31

**tall-skinny matrix** an $m \times n$ matrix where $m >> n$. 31