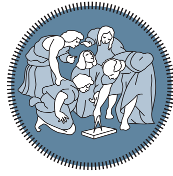


POLITECNICO DI MILANO
DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA
DOCTORAL PROGRAM IN INFORMATION TECHNOLOGY

# THE CASE FOR RECONFIGURABLE ARCHITECTURES IN HIGH-PERFORMANCE GRAPH AND SPARSE INFORMATION RETRIEVAL

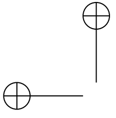
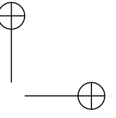
Doctoral Dissertation of:
**Alberto Parravicini**

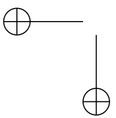
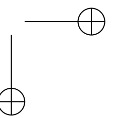
Supervisor:
**Prof. Marco D. Santambrogio**

Tutor:
**Prof. Raffaela Mirandola**

The Chair of the Doctoral Program:
**Prof. Luigi Piroddi**

2022 – XXXIV Cycle

# Ringraziamenti

Un mio grosso difetto, riflettendoci, è che non riesco a esprimere gratitudine bene quanto vorrei. Non il provare gratitudine, sia chiaro, ma esprimerla esteriormente in modo efficace. Non a caso, ho prima scritto tutto il resto e mi trovo ora a riempire, meditando parola dopo parola, l'ultima pagina che ancora attende materiale. A confronto, scrivere decine di pagine dal contenuto esoterico si è rivelato uno sforzo quasi meccanico. Definirlo facile sarebbe una bugia, e scrivere bugie in una pagina dedicata ai ringraziamenti è inopportuno. Tuttavia, mi sono impratichito nel raccontare gli aspetti tecnici e la visione della mia ricerca, e penso di riuscirci con più naturalezza di qualche anno fa. Dare forma e ordine al tutto, in queste pagine, è stato rassicurante.

Fossi un architetto che immagina case, e non un architetto che immagina calcolatori, partirei disegnando fondamenta. E le mie fondamenta, in questo percorso di crescita, sono state Marco Santambrogio.

Con le fondamenta, vengono poi i pilastri. Le persone senza il cui contributo questa tesi non ci sarebbe, un gran bel problema. Francesco, innanzitutto. Un solo aneddoto: quando mi hai proposto la tua tesi, ti dissi che eri tutto matto. Avevo ragione, ma fortunatamente lo sono anche io. Marco, compagno di *spin* di ricerca interminabili, ma di tanto in tanto fruttuosi. Luca, sarai anche la persona meno puntale del mondo, ma ho sempre adorato l'energia con cui ti butti in ogni nuovo progetto. Un gruppo un poco bislacco, ma abbiamo proprio fatto dei grandi lavori insieme.

I dettagli della vita da studente iniziano a sfumarsi, ma di alcune persone di certo non posso dimenticarmi. Senza Luca, cioè Storna, non avrei finito

il dottorato perchè molto probabilmente non lo avrei nemmeno iniziato. Mi ricordo le nostre conversazioni senza capo né ma con un loro filo conduttore tutto intricato. Mi hai davvero insegnato a immaginare, e a costruire meraviglie partendo dall'immaginazione. E poi Fulvio e Simone, cioè Ripa, splendidi amici durante tutta l'avventura degli studi di ingegneria. Le occasioni per vederci si sono un po' diradate, e per questo ho imparato ad apprezzarle ancora di più. Sono sicuro che in futuro riusciremo a vederci ancora e ancora, non importa dove saremo o cosa faremo.

Ci tengo a ringraziare di cuore tutti i miei amici, compagni e colleghi del NECSTLab. Con molti ho avuto di collaborare a stretto contatto, con altri leggermente meno, ma di tutti ho un ricordo indelebile. Di Davide, con cui ho condiviso gioie e dolori del dottorato, dell'immancabile sorriso di Guido, dei preziosi consigli di Rolando, ma anche di Emanuele, Eleonora, Francesco, Alberto Zeni e Alberto Scolari, Marco, Letizia, Lorenzo, Sara, Mirko, Andrea, Filippo, Carlotta, Fulvia e della nostra dottoranda *ad honorem* Silvia. Voglio anche ringraziare Qi ed Edoardo, per avermi tenuto al loro fianco durante le loro tesi. Spero di avervi insegnato almeno la metà di quello che voi avete insegnato a me.

Un elemento costante del mio percorso accademico è stata la collaborazione con Marco e Arnaud, e poi Rich e Mike. Qui sono riuscito solo di striscio ad accennare ai frutti del nostro lavoro, e ogni cosa ha bisogno dei propri spazi per essere apprezzata al meglio. Mi avete donato la libertà di fare ricerca sui temi più variegati, e di questo vi sono immensamente grato.

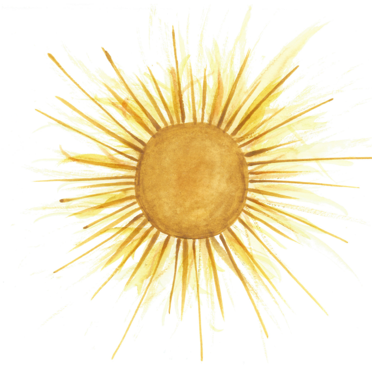
Devo assolutamente ringraziare Davide, cioè DBB, il mio spirito guida nel dottorato. Da te ho imparato a creare e sviluppare ricerca, a dare le giuste priorità, e tantissime altre cose fondamentali. Tre anni fa mi augurai di poter lavorare ancora con te, e per fortuna è stato così.

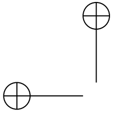
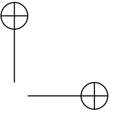
E Rene, inventore, anche un po' visionario, del meraviglioso progetto che ho poi ereditato, spero in modo degno. Certamente uno dei miei mentori, mi hai insegnato come i dettagli più piccoli sono anche i più importanti.

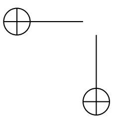
Nulla vale più di avere il supporto della propria famiglia, e poterlo ricambiare. I miei genitori Annalisa e Ambrogio, mio fratello Stefano, e i miei nonni e tutti i miei parenti. La convivenza forzata l'anno precendente, e la lontananza altrettanto forzata quando vivevo all'estero non hanno fatto altro che confermarmelo. Tralasciando le piccole divergenze quotidiane, sono davvero fortunato a poter contare su una famiglia unita.

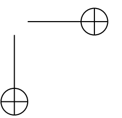
In particolare voglio ringraziare Gianandrea, lo zio migliore che ci sia. Mi hai trasmesso l'amore per la fotografia e insegnato nuovi modi per esprimermi. Sei davvero la persona su cui posso sempre contare, un'ancora inamovibile tanto nei momenti felici che in quelli più difficili.

Infine, Rebecca. Ho una pagina solo per te, per racchiudere poche ma importanti parole. Tre anni fa dissi che ci avrebbe atteso un futuro roseo. Una chiara sottostima. Il nostro futuro sarà radioso, e cosa più importante, saremo noi ad illuminarlo.





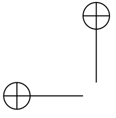
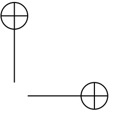
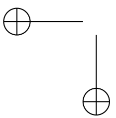
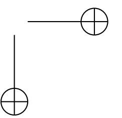


# Abstract

GRAPH analytics, information retrieval, and recommender systems are a pervasive component of our society, helping billions of users in finding web pages, movies to watch, and products to buy. Sparse linear algebra is now an essential building block of these workloads. By not storing redundant information, it enables real-time processing of an increasingly large amount of data. In particular, Sparse Matrix-Vector Multiplication (SpMV) is the cornerstone of many complex applications, from graph ranking to approximate search. SpMV does not perform well on general-purpose hardware architectures with traditional caching strategies, as it contains little data reuse and unpredictable memory access patterns.

Modern Field Programmable Gate Array (FPGA) accelerator cards have a few tricks up their sleeve. With abundant on-chip memory and HBM they enable novel data representations that maximize operational intensity and parallelism. Reduced-precision fixed-point arithmetic, a distinctive feature of FPGAs, opens the doors to trade-offs between performance and numerical accuracy in error-tolerant workloads such as recommender systems.

This thesis proposes a set of SpMV FPGA hardware designs targeted at the needs of graph analytics and recommender systems. These SpMV designs can quickly adapt to different workloads, such as graph ranking, sparse eigensolvers, and sparse embedding similarity search. In all cases, we provide state-of-the-art performance and energy efficiency and study the impact of reduced precision on accuracy. Overall, we establish that FPGAs are a fearsome contender in the race for high-performance sparse computations in graph analytics and recommender systems.

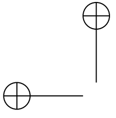
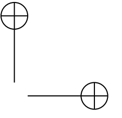





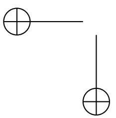
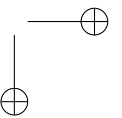
# Sommario

GRAPH analytics, information retrieval, e recommender systems sono ormai componenti imprescindibili della nostra società, e permettono a miliardi di utenti di trovare pagine web, film da guardare, e oggetti da comprare. L'algebra lineare sparsa è un elemento fondamentale di queste tecnologie. Eliminando i valori ridondanti, permette analisi in tempo reale di una mole sempre crescente di dati. In particolare, la Moltiplicazione di Matrice Sparsa per Vettore (SpMV) è alla base di complesse applicazioni come graph ranking e ricerca approssimata. SpMV non è però efficace su hardware general-purpose con politiche di caching tradizionali, avendo scarso riutilizzo di dati e accessi in memoria imprevedibili.

Le più recenti FPGA accelerator card hanno però diversi assi nella manica. Dispongono di abbondante memoria on-chip e di HBM, permettendo elevato parallelismo e l'uso di nuove strutture dati più efficienti. L'aritmetica fixed-point a precisione ridotta, un tratto distintivo delle FPGA, spalanca le porte a compromessi tra performance e accuratezza in applicazioni come recommender system che tollerano approssimazioni.

Questa tesi propone un set di design FPGA per SpMV, ottimizzati per le necessità di graph analytics e recommender systems. Questi design di SpMV possono rapidamente adattarsi a svariate applicazioni, come graph ranking e ricerca approssimata di embedding sparsi. In tutti questi casi dimostriamo performance e efficienza allo stato dell'arte, e misuriamo come la precisione numerica ridotta impatta l'accuratezza. Nel complesso, validiamo come le FPGA siano un temibile avversario nella gara per ottimizzare le computazioni sparse in graph analytics e recommender systems.

# Contents

**1 Introduction** **1**
1.1 Problem Statement . . . . . 3
1.1.1 The Memory Wall Keeps Rising . . . . . 4
1.1.2 Accelerating Sparse Linear Algebra by Leveraging Novel Memory Technologies . . . . . 4
1.1.3 Bringing FPGAs into the Equation . . . . . 5
1.2 Contributions . . . . . 6
1.2.1 A Reduced-Precision Streaming SpMV Hardware Design for Personalized PageRank on FPGA . . . . . 6
1.2.2 Solving Large Top-K Graph Eigenproblems with a Memory and Compute-optimized FPGA Design . . . 7
1.2.3 Scaling up HBM Efficiency of Top-K SpMV for Approximate Embedding Similarity on FPGAs . . . . . 7
1.3 Publications . . . . . 8
1.4 How to Read This Thesis . . . . . 9

**2 Background on Sparse Linear Algebra and Hardware Architectures** **11**
2.1 Introduction . . . . . 12
2.2 Introduction to Sparse Linear Algebra . . . . . 12
2.2.1 Building Blocks of Sparse Linear Algebra . . . . . 13
2.2.2 Data Structures for Sparse Linear Algebra . . . . . 15
2.2.3 Implementing SpMV with COO and CSR . . . . . 19
2.3 Notions of Numerical Mathematics . . . . . 21
2.3.1 Notions of Numerical Stability . . . . . 22

2.3.2 Machine Real Number Formats . . . . . . . . . . . . 24
2.3.3 Evaluating Accuracy in Information Retrieval . . . . 29
2.4 Hardware for Sparse Linear Algebra . . . . . . . . . . . . . 33
2.4.1 General-purpose CPUs (GPCPUs) . . . . . . . . . . 33
2.4.2 Graphics Processing Units (GPUs) . . . . . . . . . . 35
2.4.3 Field Programmable Gate Arrays (FPGAs) . . . . . . 37
2.4.4 FPGA Accelerator Cards . . . . . . . . . . . . . . . 42
2.4.5 Domain Specific Architectures (DSAs) and Custom Hardware Extensions . . . . . . . . . . . . . . . . . 45
2.5 Final Remarks . . . . . . . . . . . . . . . . . . . . . . . 47
2.5.1 The Importance of Memory Controllers . . . . . . . . 47
2.5.2 No architecture to rule them all . . . . . . . . . . . . 48

**3 A Reduced-Precision Streaming SpMV Hardware Design for Personalized PageRank on FPGA 51**
3.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . 52
3.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . 53
3.2.1 Numerical Optimizations . . . . . . . . . . . . . . . 53
3.2.2 CPU and GPU Implementations . . . . . . . . . . . . 53
3.2.3 FPGA Implementations . . . . . . . . . . . . . . . . 54
3.3 Problem Definition . . . . . . . . . . . . . . . . . . . . . 55
3.4 The Proposed FPGA Hardware Design . . . . . . . . . . . 58
3.4.1 Personalized PageRank Hardware Design . . . . . . . 58
3.4.2 Customizing SpMV for PPR . . . . . . . . . . . . . 59
3.4.3 PPR Buffers Design . . . . . . . . . . . . . . . . . . 60
3.4.4 Host Integration . . . . . . . . . . . . . . . . . . . . 64
3.5 Experimental Evaluation . . . . . . . . . . . . . . . . . . 65
3.5.1 Execution time . . . . . . . . . . . . . . . . . . . . . 67
3.5.2 Energy Efficiency . . . . . . . . . . . . . . . . . . . 69
3.5.3 Accuracy Analysis . . . . . . . . . . . . . . . . . . . 71
3.5.4 Fixed-point produces faster convergence . . . . . . . 73
3.6 Final Remarks . . . . . . . . . . . . . . . . . . . . . . . 74

**4 Solving Large Top-K Graph Eigenproblems with a Memory and Compute-optimized FPGA Design 77**
4.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . 78
4.1.1 Motivation . . . . . . . . . . . . . . . . . . . . . . . 78
4.1.2 Contributions . . . . . . . . . . . . . . . . . . . . . 79
4.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . 80
4.3 Solving the Top-K Sparse Eigenproblem . . . . . . . . . . . 82

4.3.1 The Lanczos Algorithm . . . . . . . . . . . . . . . . . 84
4.3.2 The Jacobi Eigenvalue Algorithm . . . . . . . . . . . 86
4.4 The Proposed FPGA Hardware Design . . . . . . . . . . . 88
4.4.1 Lanczos Hardware Design . . . . . . . . . . . . . . . . 88
4.4.2 SpMV Hardware Design . . . . . . . . . . . . . . . . . 89
4.4.3 Jacobi Systolic Array Design . . . . . . . . . . . . . . 93
4.5 Experimental Evaluation . . . . . . . . . . . . . . . . . . . 96
4.5.1 Execution Time . . . . . . . . . . . . . . . . . . . . . 97
4.5.2 Power Efficiency . . . . . . . . . . . . . . . . . . . . . 101
4.5.3 Accuracy Analysis of the Approximate Eigencomputation . . . . . . . . . . . . . . . . . . . . . . . . . 102
4.6 Final Remarks . . . . . . . . . . . . . . . . . . . . . . . . 102

**5 Scaling up HBM Efficiency of Top-K SpMV for Approximate Embedding Similarity on FPGAs 105**
5.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . 106
5.2 Related Work . . . . . . . . . . . . . . . . . . . . . . . . . 108
5.3 Theoretical Contributions . . . . . . . . . . . . . . . . . . . 109
5.3.1 Top-K SPMV Approximation . . . . . . . . . . . . . . 110
5.3.2 The Block-Streaming CSR Matrix Layout . . . . . . 113
5.4 The Proposed FPGA Hardware Design . . . . . . . . . . . 115
5.4.1 Leveraging URAM for Fast Random Access . . . . . 116
5.4.2 Top-K SpMV Algorithm Design . . . . . . . . . . . 117
5.4.3 Lower Precision, More Cores, Better Performance . . 118
5.4.4 Host Integration . . . . . . . . . . . . . . . . . . . . . 120
5.5 Experimental Evaluation . . . . . . . . . . . . . . . . . . . 120
5.5.1 Execution Time . . . . . . . . . . . . . . . . . . . . . 122
5.5.2 Power Efficiency . . . . . . . . . . . . . . . . . . . . . 124
5.5.3 Roofline Model Analysis . . . . . . . . . . . . . . . . 125
5.5.4 Approximation Accuracy Analysis . . . . . . . . . . 127
5.6 Final Remarks . . . . . . . . . . . . . . . . . . . . . . . . 129

**6 Conclusion and Future Work 131**
6.1 Limitations and Future Directions . . . . . . . . . . . . . . 133
6.1.1 A Reduced-Precision Streaming SpMV Hardware Design for Personalized PageRank on FPGA . . . . . . 133
6.1.2 Solving Large Top-K Graph Eigenproblems with a Memory and Compute-optimized FPGA Design . . . 134
6.1.3 Scaling up HBM Efficiency of Top-K SpMV for Approximate Embedding Similarity on FPGAs . . . . . 135

**List of Acronyms** **143**

**Bibliography** **147**

# CHAPTER 1

# Introduction

Graph analytics, information retrieval, and recommender systems process an always-increasing amount of data, often with strong real-time constraints, to suggest products, movies, news articles to billions of users. For better or worse, they are an integral part of our society.

Sparse linear algebra has become a staple of graph analytics and recommender systems. The original applications of sparse linear algebra were in the fields of image and signal processing. Now, sparse computations are seen as the only way to perform numerical computations on enormous heterogeneous datasets that would be impossible to represent in a dense format. Sparse representations are inevitable when processing graphs, as social networks and web graphs keep growing in popularity and scale. With Deep Learning (DL) and numerical embeddings, now a standard tool of recommender systems, sparsity is key to train larger models thanks to the regularization offered by having most weights equal to zero [83]. While sparse matrices representing weights and embeddings used in DL still have a high density ($0.01$, versus the $\leq 10^{-6}$ typical of graphs), we are observing a shift towards higher sparsity. An example is the introduction of hardware components specific to sparsity acceleration, such as Nvidia's Sparse Tensor Cores in recent Ampere Graphics Processing Units (GPUs) [37, 149].

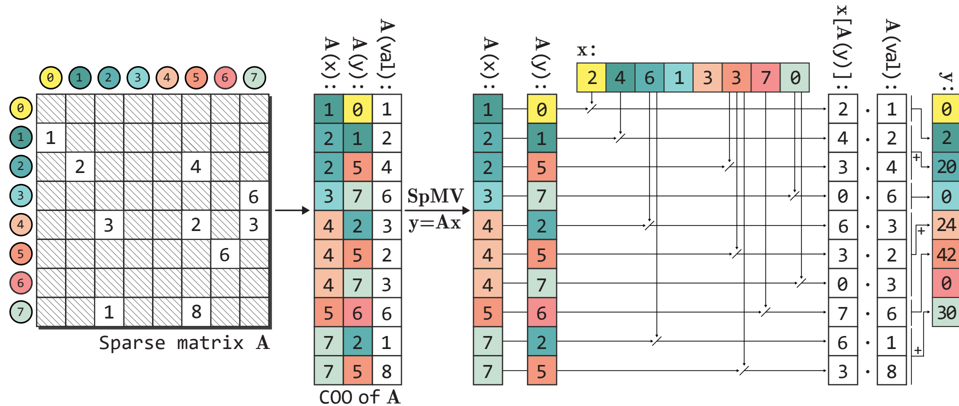



**Figure 1.1:** *Example of Sparse Matrix-Vector Multiplication (SpMV), a sparse linear algebra operation widely employed in graph analytics and recommender systems. In this example, the matrix A is mostly composed of* 0*. We store it with the COO representation (Section 2.2), and compute* $y = Ax$*, where both* x *and* y *are dense vectors.*

Indeed, sparse linear algebra has drawn the interest of the hardware research community for a long time [45, 50, 51, 139, 232]. This interest is now higher than ever. The recent popularity explosion of graph analytics and DL demands novel techniques to process sparse matrices with millions, billions, or possibly trillions of non-zero entries, representing web-scale graphs, social networks, or databases of embeddings. In the field of DL alone, the number of publications related to sparse computing has increased by 10× over the past ten years [83], with articles related to hardware optimizations now being 15–20 % of the total. However, research on hardware optimizations for recommender systems is still uncommon, and many problems are yet to be solved [63].

The challenges related to sparse linear algebra are not limited to scale but also performance and energy efficiency, as graph analytics and recommender system workloads often have to answer real-time queries from billions of users. Sparse computations are extremely memory intensive and present data-dependent and random memory accesses, making them unsuitable for general-purpose architectures with traditional caching policies.

On the other hand, Field-Programmable Gate Arrays (FPGAs) can fully leverage sparse data representations and reduced-precision arithmetic to improve operational intensity and performance in memory-starved computations. Exact numerical accuracy is not critical as long as the most relevant recommendations are correct. Compared to CPUs or GPUs, FPGAs can leverage optimized arbitrary-precision arithmetic and offer fine-

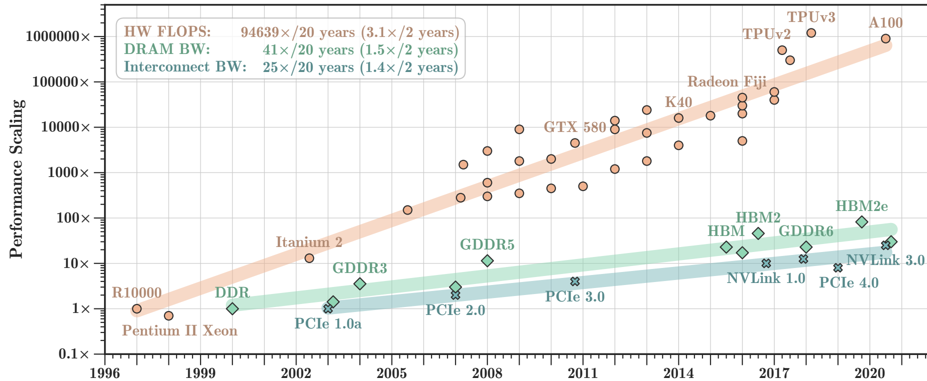



**Figure 1.2:** *Scaling of peak compute power (FLOPS), memory bandwidth (GB/s) and interconnection bandwidth (GB/s) from 1996 to today, adapted from Gholami et al. [64]. Peak compute power has improved by almost 100000× in the last 20 years thanks to GPUs and DSAs. On the other hand, memory bandwidth has grown by only 40×. As such, optimizing memory-bound sparse workloads is becoming increasingly difficult, as the size of data keeps rising.*

grained control over the desired target accuracy, providing better performance, lower resource utilization, and lower power consumption. Thanks to their flexibility, FPGAs have proven commercially successful in recommender system workloads since the seminal work of Project Catapult by Microsoft [33, 165].

Our goal is to improve the performance of Sparse Matrix-Vector Multiplication (SpMV) (Figure 1.1), the cornerstone operation of sparse linear algebra, in the context of graph analytics and recommender systems. We leverage the most recent advancements in FPGA accelerator cards to maximize the impact of High Bandwidth Memory (HBM), reduced-precision fixed-point arithmetic, and streaming computations. As a result, we propose a novel set of flexible FPGA hardware designs for SpMV, targeted at high-performance sparse computations in graph analytics and recommender systems.

## 1.1 Problem Statement

In this thesis, we explore the acceleration of sparse linear algebra in the context of graph analytics and recommender systems through the use of modern FPGA accelerator cards and approximate computing. Many research problems are still unsolved in this field. Here, we introduce the most important ones, which we try to address in this work.

### 1.1.1 The Memory Wall Keeps Rising

The size of predictive models and datasets used in recommender systems, information retrieval algorithms, and graph analytics keeps growing at a steady pace year after year [41, 63, 64, 91, 106]. Large recommender systems now require 1 to 10 trillions parameters, stored in up to $10\,\text{TB}$ of memory, and this value has increased more than $200\times$ in the past two years. Databases in recommender systems have been growing just as much. Social network graphs count billions of users, and recommender systems have millions of products and billions of purchases. In contrast, DRAM available in accelerator devices such as GPUs or FPGAs has improved by only $2\times$. Even worse, the scaling of memory bandwidth has been insignificant compared to the scaling in computing power (Figure 1.2). Peak compute power has increased by $3.1\times$ every two years or $90000\times$ over the last 20 years. On the other hand, memory bandwidth has grown by just $1.5\times$ every two years, i.e. a meager $40\times$ over the same 20 years. The insufficient size and bandwidth of memory create what is known as *memory wall*, a major bottleneck of today's recommender systems due to the low operational intensity of these workloads. Compression and sparsification techniques are helpful to increase the operational intensity, but results are still insufficient, given the size of input data (e.g. web-scale graphs) and predictive models [83]. While most researchers are worried about the ending of Moore's Law, the unstoppable growth of this memory wall is equally, if not more, troublesome. There is an increasing need for computational techniques that can maximize computation for every bit loaded from memory. In this regard, reduced-precision arithmetic and sparsification techniques are poised to become more and more necessary.

### 1.1.2 Accelerating Sparse Linear Algebra by Leveraging Novel Memory Technologies

Sparse linear algebra is a promising approach to improve the performance and scale of recommender systems and graph analytics workloads [13, 41, 63, 76]. However, effective parallelization and hardware acceleration of sparse primitive operations is still an open challenge. Compared to dense linear algebra, sparse operations have complex memory access patterns (random or data-dependent) and memory bandwidth is often their performance bound. In this area, new memory technologies such as HBM can be of great help, thanks to their abundant memory bandwidth. However, fully leveraging this bandwidth is not trivial as it requires many concurrent memory transactions on independent memory channels. When dealing

with sparse computations and their unpredictable memory accesses, it is often necessary to employ problem-specific solutions [93, 168, 185, 220], and generalization to a wide range of workloads cannot be taken for granted.

### 1.1.3 Bringing FPGAs into the Equation

Given the challenges that still exist in accelerating sparse linear algebra, especially in the context of recommender systems and graph analytics, FPGAs represent a very attractive architectural choice for a variety of reasons. FPGAs can leverage reduced-precision fixed-point arithmetic, which is highly effective in error-tolerant workloads typical of recommender systems. FPGAs are also more energy-efficient than competing hardware and have predictable execution latency, making them suitable for real-time data center workloads. FPGA accelerator cards are now equipped with HBM and provide the memory required by such demanding sparse workloads.

However, leveraging these FPGAs accelerator cards for sparse recommender system and graph analytics workloads is a field yet to be fully explored, even more so when reduced-precision fixed-point arithmetic is brought into the equation. Moreover, reaching high HBM bandwidth utilization is more complex than in competing architectures such as GPUs, often requiring ad-hoc partitioning techniques and memory access patterns. Finally, FPGAs require well-thought utilization of on-chip memory. Sparse matrices are usually too large to fit inside on-chip memory, and caches might not be effective due to the low data reuse in sparse workloads. Once again, problem-specific caching techniques are often a necessity.

Overall, the following unique circumstances motivates this thesis.

1. Recommender systems and graph analytics are more than ever an integral part of our society.

2. Sparsity is necessary to represent and process the large-scale datasets used in recommender systems and graph analytics

3. The output of these workloads is tolerant to numerical approximations, as they produce rankings instead of precise numerical values.

4. Recommender systems in data centers require low latency and energy consumption to sustain millions or even billions of daily queries.

5. Sparse linear algebra computations present complex memory access patterns, and are generally memory-bound instead of compute-bound.

6. Modern FPGA accelerator cards provide flexible reduced-precision arithmetic and allow deployment of low-power designs with real-time

latency. Their abundant off-chip memory, combined with the FPGA reconfigurable logic, enable optimized memory accesses and storage of large sparse matrices.

## 1.2 Contributions

We propose a novel set of hardware designs based on modern FPGA accelerator cards to improve the performance of sparse linear algebra in the context of foundational operations encountered in graph analytics and recommender systems. At the core of this work lies a **flexible SpMV hardware design**, created to leverage reduced precision fixed-point arithmetic and adapt to kinds of memories, from on-chip memories such as UltraRAM (URAM), to off-chip memories such as DDR4 and HBM2.

We prove how this design can be easily adapted to different sparse workloads, such as **Personalized PageRank (PPR)**, **sparse eigensolvers**, and **sparse embedding similarity search**. In each case, we study the impact of reduced precision on performance and accuracy and highlight the challenges in each application. In detail, we present the following contributions.

### 1.2.1 A Reduced-Precision Streaming SpMV Hardware Design for Personalized PageRank on FPGA

In Chapter 3, we first present our FPGA SpMV hardware design in the context of iterative graph ranking algorithms. PPR, in particular, is often used as a building block of recommender systems in e-commerce websites and social networks. In this context, low latency and high throughput are more valuable than exact numerical convergence, creating the ideal condition to experiment with reduced-precision fixed-point arithmetic. Through our novel streaming reduced-precision Coordinate (COO) SpMV design, we achieve the following results.

- We measure that our PPR design obtains speedups up to 6× over a state-of-the-art multi-threaded CPU implementation on 8 different graphs, with up to 42× higher energy efficiency (Section 3.5.1).

- We evaluate that reduced-precision fixed-point arithmetic does not negatively affect the accuracy. Just 26-bits are enough to provide almost-perfect results (Section 3.5.3).

- We analyze how fixed-point arithmetic translates to a **faster convergence**, often 2× faster than floating-point arithmetic (Section 3.5.4).

### 1.2.2 Solving Large Top-K Graph Eigenproblems with a Memory and Compute-optimized FPGA Design

In Chapter 4, we extend our SpMV hardware design to sparse eigensolvers, numerical methods that compute eigenvectors and eigenvalues on large sparse matrices. Large-scale sparse eigensolvers are a key component of graph analytics techniques based on spectral methods. In such applications, an exhaustive computation of all eigenvalues and eigenvectors is impractical and unnecessary. Even on enormous graphs, spectral methods can retrieve their most important properties with only the eigenvectors associated with the Top-K largest eigenvalues. Moreover, spectral methods adopted in information retrieval and recommender systems are tolerant to approximations introduced by fixed-point arithmetic. To the best of our knowledge, we are the first to propose an FPGA Top-K eigensolver for unstructured sparse matrices, with the following contributions.

1. We integrate the SpMV hardware design into a mixed-precision eigensolver based on the **Lanczos algorithm** and the **Jacobi eigenvalue algorithm** (Section 4.3).

2. We extend our FPGA SpMV hardware design to **leverage the flexibility of HBM**. Thanks to HBM, we partition the SpMV computation across multiple Compute Units (CUs), achieving more than 5× better performance than our original hardware design (Section 4.4).

3. We achieve a speedup of 6.22× compared to the highly optimized ARPACK library running on an 80-thread CPU, while keeping high accuracy and 49× better power efficiency (Section 4.5).

### 1.2.3 Scaling up HBM Efficiency of Top-K SpMV for Approximate Embedding Similarity on FPGAs

In Chapter 5, we analyze the problem of sparsity in recommender systems from a different perspective. Instead of optimizing computations on graphs, we consider the task of approximate similarity search on large sparse embedding tables, which can be efficiently implemented with Top-K SpMV. This sparse workload does not perform well on general-purpose NUMA systems with traditional caching strategies. Instead, we further extend our FPGA SpMV design to handle Top-K SpMV, optimizing the computation for tall matrices that benefit from on-chip caches. In this chapter, we present the following results.

1. We create a new approximation scheme for **Top-K SpMV**, parallelizing the computation over 32 independent SpMV CUs and HBM pseudo channels, exploiting the FPGA resources to its fullest.
2. We introduce a novel packet-wise CSR matrix compression called **BSCSR**. This format is optimized for streaming reduced-precision computations, and it increases operational intensity of SpMV by up to 3× over COO.
3. We measure that our FPGA design is 138× faster than a multi-threaded CPU implementation and 2.1× **faster than a GPU** with 20 % higher bandwidth, with 15× higher power-efficiency, proving that FPGAs are today the optimal solution for Top-K SpMV.

## 1.3 Publications

The main contributions of this doctoral dissertation refer to and extend the following articles.

- **Alberto Parravicini**, Francesco Sgherzi and Marco Domenico Santambrogio. *A reduced-precision streaming SpMV architecture for Personalized PageRank on FPGA.* In 2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC). Pages 378 – 383. January 2021 [158].
- Francesco Sgherzi, **Alberto Parravicini**, Marco Siracusa and Marco Domenico Santambrogio. *Solving Large Top-K Graph Eigenproblems with a Memory and Compute-optimized FPGA Design.* In 2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM). Pages 78 – 87. May 2021 [177].
- **Alberto Parravicini**, Luca Giuseppe Cellamare, Marco Siracusa and Marco Domenico Santambrogio. *Scaling up HBM Efficiency of Top-K SpMV for Approximate Embedding Similarity on FPGAs.* To appear in 58th Design Automation Conference (DAC). Pages 1 – 6. December 2021 [155].

Other publications, also produced during the Ph.D. research, follow outside the specific scope of this dissertation. Relevant excerpts are still mentioned whenever appropriate.

- **Alberto Parravicini**, Rhicheek Patra, Davide Basilio Bartolini and Marco Domenico Santambrogio. *Fast and accurate entity linking via*

*graph embedding.* In Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA). Pages 1 – 9. June 2019 [157].

- **Alberto Parravicini**, Arnaud Delamare, Marco Arnaboldi and Marco Domenico Santambrogio. *DAG-based Scheduling with Resource Sharing for Multi-task Applications in a Polyglot GPU Runtime.* In 2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS). Pages 111 – 120. May 2021 [156].

## 1.4 How to Read This Thesis

For those readers lacking a deep background knowledge of sparse linear algebra or reconfigurable computing, Chapter 2 provides a comprehensive overview of the building blocks on which the rest of the thesis is built. The subsequent chapters are organized to be self-sufficient, to guide the reader interested in only some of the contributions of this thesis. Chapters 3 to 5 represent the core of this thesis. Chapter 6 concludes the thesis with a reflection on challenges that are still open and further research directions opened by the present work. The core chapters can be read independently and provide an in-depth introduction with all the necessary notions, context, and relevant state-of-the-art required to appreciate them. However, it is best to read this thesis in order. Each chapter builds on the preceding ones, tracing a path started by experimenting with SpMV on a relatively narrow use-case (PPR) and progressively broadening the scope of the research to more complex applications and algorithms. Eventually, we show how a flexible reduced-precision SpMV hardware design can improve the performance and efficiency of a wide variety of real-world applications.

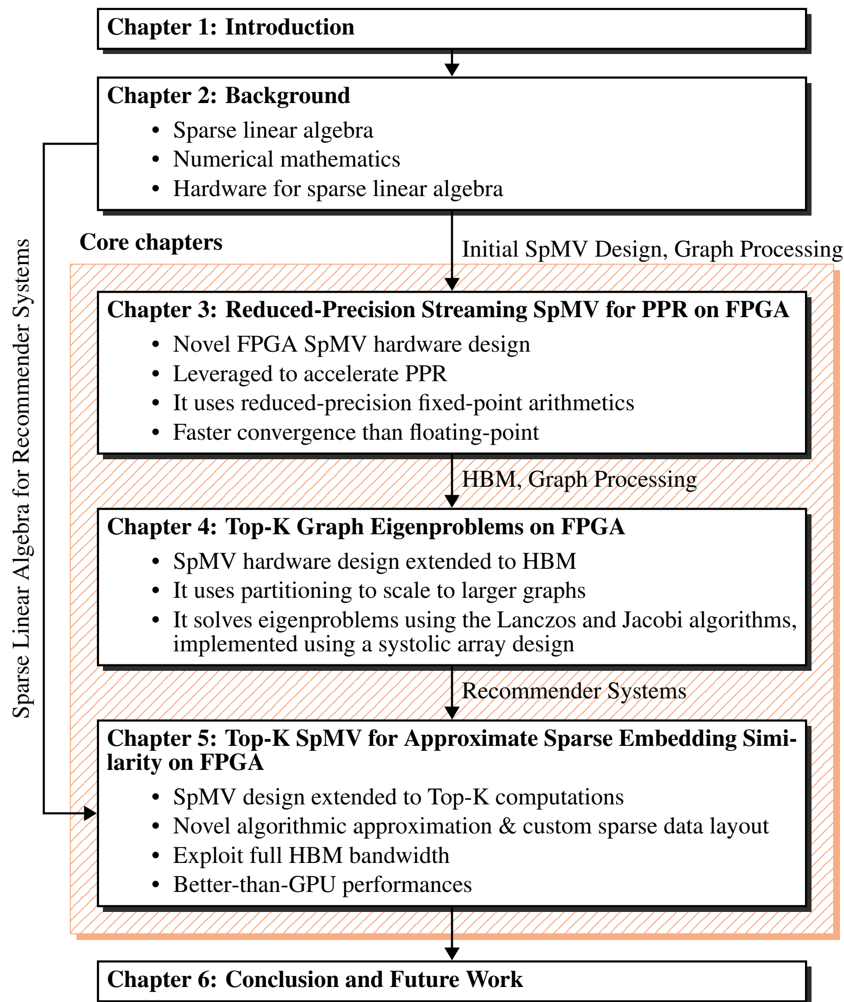



**Figure 1.3:** *Structure of this thesis, with the main contributions of each chapter.*

# CHAPTER 2

# Background on Sparse Linear Algebra and Hardware Architectures

*The topic and contributions of this thesis extend to multiple areas of computer science. This chapter introduces the main building blocks required in the following chapters, to make them accessible to anyone without strong expertise in sparse linear algebra and reconfigurable architectures.*

*First, we introduce the language and the challenges of sparse linear algebra. It will be clear sparse linear algebra employs complex memory access patterns, and simply providing more computational resources rarely results in an immediate performance improvement.*

*Second, we explain how fixed-point arithmetic and approximate computing are appealing solutions in information retrieval, where one can establish a successful trade-off between performance and accuracy.*

*Finally, we survey the landscape of hardware architectures in the context of sparse linear algebra, illustrating how each of them has different strengths, and few can currently leverage approximate computing. The case of FPGA accelerator cards will be presented, which can easily adapt to different accuracy requirements, providing top-of-the-line performance and power efficiency.*

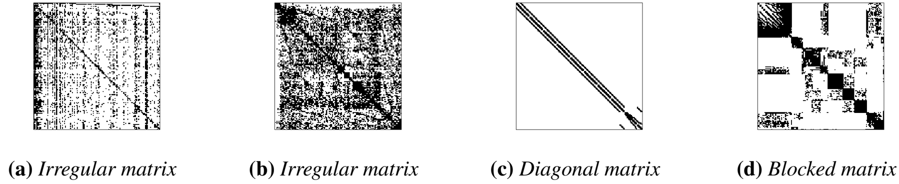
## 2.1 Introduction

Given the growing interest in recommender systems that can leverage sparse linear algebra and the limitations of current hardware architectures on such workloads, it is critical to provide fast and efficient architectural primitives for sparse computations. It is no surprise that Deep Learning (DL) is still dominated by dense linear algebra computations, as they are the simplest to parallelize on high-performance architectures such as Graphics Processing Units (GPUs). However, we observe how large-scale recommender systems, such as the ones used by Facebook [76] show an increasing percentage of sparse computations. Their bottlenecks are entirely different from dense computations – unpredictable memory accesses, embedding lookup, and manipulation, instead of heavyweight mathematical operations. Even if dense workloads are still dominant, sparse computations already account for 20,% of their AI inference cycles [75]. In the case of graph analytics, the need for high-performance sparse operations is even more evident, as graphs employed in real-world applications already have sizes that make dense computations prohibitive. In short, making sparse workloads faster and more power-efficient is high-priority research with a clear real-world impact for multiple fields, from computer architecture to machine learning.

This chapter provides the necessary background to **sparse linear algebra** and its primitive operations (Section 2.2), which will encounter in graph analytics and Information Retrieval (IR) algorithms, in the following chapters. We also provide an introduction to the concepts of **numerical stability** and **fixed-point arithmetic**, as both play a significant role in the approximate computing techniques that underpin this thesis (Section 2.3). Finally, we survey the landscape of **modern hardware architectures** in the context of sparse linear algebra and provide general considerations about the strengths and weaknesses of each (Section 2.4).

## 2.2 Introduction to Sparse Linear Algebra

The first step to understanding the content of this thesis is to define the basic language of sparse linear algebra (Section 2.2.1). While these primitive operations are seemingly equivalent to their dense linear algebra counterparts, we will see how sparse matrices can be represented with a variety of data structures, each with different implementations and algorithmic complexity (Section 2.2.2). Figure 2.1 shows four sparse matrices, giving an intuitive feeling of how diversified the landscape of sparse matrices is: each matrix will benefit from specific data structures and algorithms.



**(a)** *Irregular matrix* **(b)** *Irregular matrix* **(c)** *Diagonal matrix* **(d)** *Blocked matrix*

**Figure 2.1:** *Patterns encountered in sparse matrices from the SuiteSparse collection [43]. The first two matrices do not have any visible structure. The third has clearly a diagonal structure, while the fourth is composed of dense blocks.*

### 2.2.1 Building Blocks of Sparse Linear Algebra

Throughout this thesis, we adopt the following notation. A lowercase letter, such as $a$, denotes a scalar value. A bold lowercase letter, such as $\mathbf{x}$ or $\mathbf{y}$, denotes a vector. An uppercase bold letter, such as $\mathbf{A}$, denotes a matrix. Informally, we identify with typewriter font, e.g. `ptr`, the data structures (e.g. arrays) that implement these mathematical objects. Unless otherwise specified, assume that vectors are dense, i.e. most of their elements are different from zero, and all the values are explicitly stored. Conversely, assume that matrices are sparse, i.e. only a small fraction of their values are different from zero. We call such values *non-zeros*, or non-zero entries. We usually denote with uppercase letters the dimensionality of vectors and matrices. For example, a vector has $N$ elements, while a sparse matrix has $N$ rows, $M$ columns, and $nnz$ non-zeros. A matrix is sparse if $nnz \ll NM$, or, alternatively, if $nnz \sim \mathcal{O}(max(N, M))$. A matrix with $nnz < \mathcal{O}(max(N, M))$ is called hyper-sparse, but these matrices go beyond the scope of this work. Most vectors and matrices seen in this work contain real number, i.e. $\mathbf{x} \in \mathbb{R}^N$ and $\mathbf{A} \in \mathbb{R}^{N \times M}$. The following two definitions will also appear very often.

**Definition 2.1** (Degree)**.** The number of non-zeros in a row is called *degree* of the row, denoted as $\delta$. It is also common to mention the average degree, the average number of non-zeros per row, obtained as $\delta = nnz/N$.

**Definition 2.2** (Sparsity, and density)**.** The fraction of non-zeros in the matrix is called sparsity, and it is sometimes expressed as a percentage. Sparsity is computed as $s = nnz/(N \cdot M)$. Conversely, density is $d = 1 - s$.

While outside the scope of this work, one can also define sparse tensors by generalizing the notion of sparse vectors and matrices. Intuitively, an order-1 tensor is a vector, an order-2 tensor is a matrix, while an order-3 tensor is a 3-dimensional matrix, and so on. Sparse tensors are commonly

encountered in DL, and are at the center of very exciting research (Sections 2.4.2 and 2.4.5). We do not directly work on tensors, although many of the architectural considerations and optimizations presented in this work (such as operational intensity optimizations, reduced-precision arithmetic, and compressed matrix representations) also apply to tensors.

#### Primitive Operations of Sparse Linear Algebra

Like one can define basic algebraic operations on dense vectors and matrices, such as vector or matrix multiplication, it is possible to do the same for sparse linear algebra. In the context of dense linear algebra, these primitive operations are defined by the Basic Linear Algebra Subprograms (BLAS) specification [115, 141]. A similar set of operations is defined in the Sparse Basic Linear Algebra Subprograms (Sparse BLAS) specification [145]. Operations in BLAS and Sparse BLAS are divided into levels. Level 1 (L1) is used for operations on vectors. Level 2 (L2) is used for operations involving matrices and vectors, while Level 3 (L3) indicates operations that combine matrices with other matrices. Here we define the most important ones.

**L1 – Sparse dot-product:** $a \leftarrow \mathrm{x} \cdot \mathrm{y}$, or $a \leftarrow \langle \mathrm{x}, \mathrm{y} \rangle$. Inner product of two sparse vector. If both $\mathrm{x}$ and $\mathrm{y}$ have unitary L2 norm, the dot product is equivalent to the cosine similarity [183].

**L1 – Sparse vector update:** $\mathrm{z} \leftarrow a\mathrm{x} + \mathrm{y}$. Sum of two sparse vectors, with one scaled by a constant factor. Also known as sparse `axpy`.

**L1 – Sparse gather:** $\mathrm{z} \leftarrow \mathrm{x}|\mathrm{y}$, with $\mathrm{x} \in \mathbb{R}^N,\ \mathrm{y} \in \mathbb{N}^M,\ M \leq N, \forall y_i \in \mathrm{y} : y_i \leq N$. Assign to $\mathrm{z}$ the elements of $\mathrm{x}$ with index specified by the values of $\mathrm{y}$.

**L1 – Sparse scatter:** $\mathrm{x}|\mathrm{y} \leftarrow \mathrm{z}$. Inverse operation of sparse gather, assign the elements of $\mathrm{z}$ to the indices of $\mathrm{x}$ specified by $\mathrm{y}$.

**L2 – Sparse Matrix-Vector Multiplication (SpMV):** $\mathrm{z} \leftarrow a\mathrm{A}\mathrm{x} + b\mathrm{y}$. Product of sparse matrix by dense vector, with $\mathrm{A} \in \mathbb{R}^{N \times M}$ being a sparse matrix and $\mathrm{x} \in \mathbb{R}^M$, $\mathrm{y} \in \mathbb{R}^N$, $\mathrm{z} \in \mathbb{R}^N$ being dense vectors. The core operations which is analyzed and accelerated in this thesis. In most cases, we assume $a = 1,\ b = 0$.

**L3 – Sparse Matrix-Dense Matrix Multiplication (SpMM):** given $\mathrm{A} \in \mathbb{R}^{N \times K}$ sparse, and dense matrices $\mathrm{B} \in \mathbb{R}^{K \times M}$ and $\mathrm{C} \in \mathbb{R}^{N \times M}$, compute the dense matrix $\mathrm{D} \leftarrow a\mathrm{A}\mathrm{B} + \mathrm{C}$. From a practical standpoint, it can be seen as a SpMV applied to multiple vectors.

**L3 – Sparse Matrix-Matrix Multiplication (SpGEMM):** product of two sparse matrices, with $\mathrm{A} \in \mathbb{R}^{N \times K}$, $\mathrm{B} \in \mathbb{R}^{K \times M}$ and $\mathrm{C} \in \mathbb{R}^{N \times M}$ being all sparse matrices. SpGEMM computes the matrix $\mathrm{D} \leftarrow a\mathrm{AB} + \mathrm{C}$. Conventionally, $\mathrm{D}$ is also sparse, but in some implementations, it might be dense. Sometimes, the name SpMM is used in place of SpGEMM, e.g. [99]. Efficient algorithms for SpGEMM are extremely complex to create and, similarly to SpMV, still an active subject of research [60]. The most classic algorithm for SpGEMM is attributed to Gustavson [77], but many hardware-optimized algorithms have been recently proposed [99, 159, 218, 226].

The algorithmic space and time complexity of these operations are always strongly dependent on the data structures used to represent the sparse data and the characteristics of the underlying hardware.

#### Representing Graphs with Sparse Matrices

Graphs are also commonly stored using sparse matrices. Using sparse matrices is a perfectly reasonable choice. The friendship graph of a social network might have millions or billions of users (i.e. vertices), but each user has, on average, a few hundred friends. Similarly, a web graph representing links between web pages has billions of pages, but each page only links to a handful of other pages. When discussing graphs, we define it as $G = (V, E)$, with $V$ being the set of vertices, and $E \subseteq V \times V$ is the set of edges, i.e. links between pairs of vertices. $|V|$ and $|E|$ are the number of vertices and edges, respectively. In the context of sparse matrices, a graph can be represented as a square matrix $\mathrm{A} \in \{0, 1\}^{|V| \times |V|}$, where each row and column represent vertices, and a value of $1$ indicates the presence of an edge for the given pair of vertices. The number of edges $|E|$ is equal to the number of non-zeros $nnz$. Edges can have numerical values: in this case, we store the graph as $\mathrm{A} \in \mathbb{R}^{|V| \times |V|}$.

### 2.2.2 Data Structures for Sparse Linear Algebra

There exist many different ways to store and represent sparse matrices. Choosing the most suitable data structure requires knowledge of the matrix content (e.g. if the non-zero elements have a predictable structure), the target application (e.g. if it is an iterative algorithm), and the architecture that will operate on the matrix. Each of these data structures has different advantages and disadvantages, and it is important to understand their algorithmic complexity and implementation details to adopt the most suitable one. For example, one might want (or have) to choose the data structure

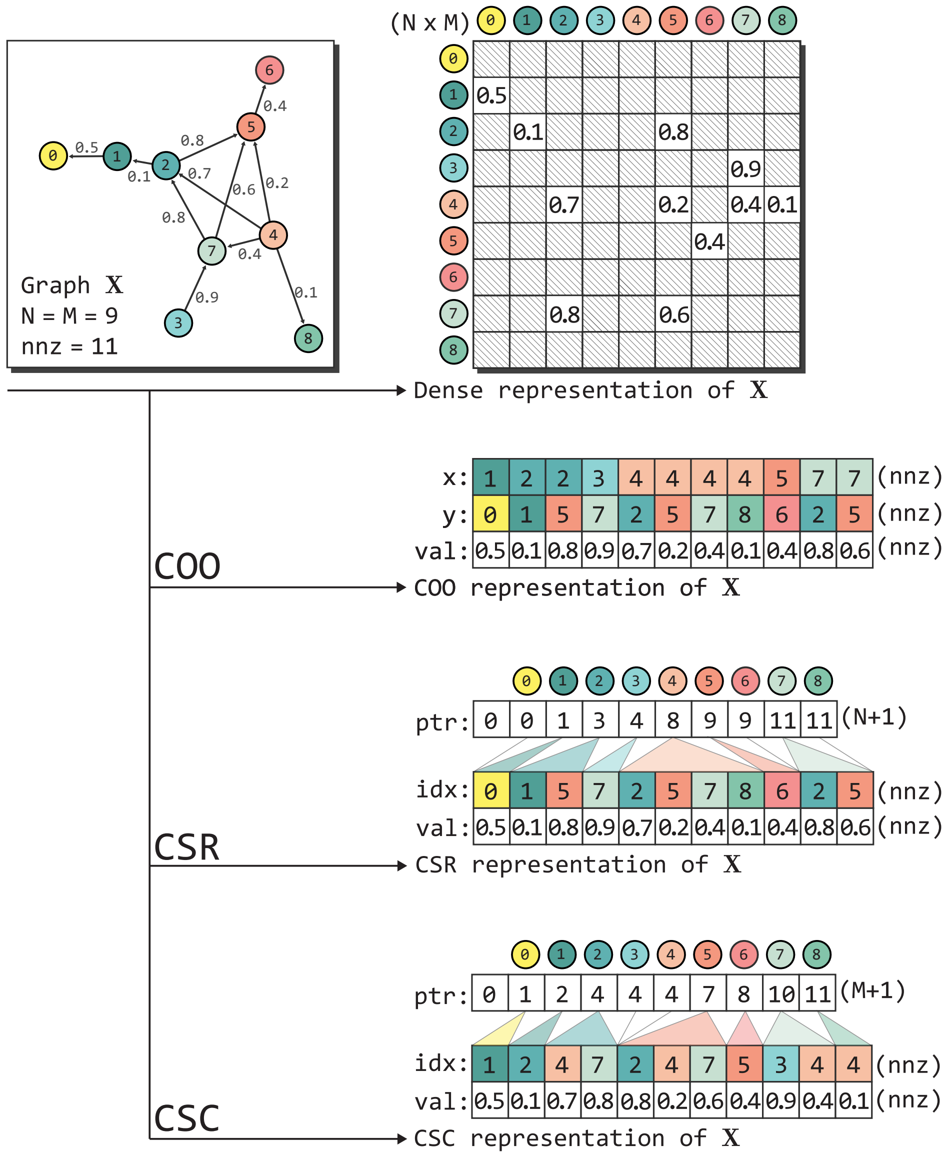



**Figure 2.2:** *The constellation of Hercules can be stored with different sparse representations. The dense representation requires 81 values, of which only 11 are different from zero. Sparse representations are more efficient, requiring only 31 or 33 values. This difference in efficiency becomes even more visible on large matrices such as the ones in Chapter 4, where less than 0.001 % of the values are $\neq 0$.*

with the best compression ratio, at the cost of a performance penalty, or pick a representation with a larger memory footprint but possibly the best peak performance. Below, we summarize the most common data structures

**Table 2.1:** *Storage size, traversal cost and non-zero lookup cost of conventional sparse matrix representations. Assume a sparse matrix* $\mathrm{A} \in \mathbb{R}^{N \times M}$*, with* $nnz$ *non-zeros, and* $B$ *dense blocks in the case of BSR.*

| Representation | Storage size and traversal cost | Non-zero lookup |
|---|---|---|
| **Dense** | $N \cdot M$ | 1 |
| **COO** | $3 \cdot nnz$ | $\mathcal{O}(nnz)$, or $\mathcal{O}(log(nnz) + nnz/N)$ if sorted |
| **CSR** | $2 \cdot nnz + N$ | $1 + \mathcal{O}(nnz/N)$ |
| **CSC** | $2 \cdot nnz + M$ | $1 + \mathcal{O}(nnz/M)$ |
| **Banded** | Problem dependent, $\sim \mathcal{O}(N)$ | $\mathcal{O}(1)$ |
| **BSR** | Problem dependent | $nnz$, $\mathcal{O}(B) + 1$ if storing start/end coordinates of each block |

for sparse matrices. In this section, assume that we deal with square sparse matrices with $N$ rows and columns. We summarize the main characteristics of each data structure in Table 2.1, and provide a visual example in Figure 2.2.

#### Coordinate Format (COO)

The most intuitive way to represent a sparse matrix is to store its non-zeros in a sequential list, along with their coordinates. The Coordinate (COO) format requires three vectors of size $nnz$. The first two, $\mathtt{x} \in \mathbb{N}^{nnz}$ and $\mathtt{y} \in \mathbb{N}^{nnz}$, store the coordinates of non-zero entries. The third vector, $\mathtt{val} \in \mathbb{R}^{nnz}$, stores the values of non-zero entries. If we are only storing the topology of a graph, the third array is not necessary. Alternatively, one can use a single array of triples ($\mathtt{x}_i$, $\mathtt{y}_i$, $\mathtt{val}_i$). Accessing random edges and iterating on non-zeros of a given row is extremely inefficient ($\mathcal{O}(nnz)$, possibly $\mathcal{O}(log(nnz) + N)$ if values are sorted with respect to `x` and `y`, as one can use binary search). However, COO is a highly efficient format for streaming computations that process all non-zeros in the matrix in a sequential fashion, as in SpMV. No nested loops are necessary, reducing the risk of branch mispredictions and pipeline stalls.

#### Compressed Sparse Row (CSR)

COO has two main drawbacks. First, it contains redundant information, as the `x` coordinates are repeated for every non-zero in the same row. Second, it does not allow for the traversal of non-zeros in random rows. In COO, if we want to sum the values of a row, we have to traverse all the $nnz$ entries in the matrix possibly.

The Compressed Sparse Row (CSR) format solves both problems. It uses a ptr $\in \mathbb{N}^{N+1}$ array that stores the cumulative degree of each row, starting from $0$. In other words, $\texttt{ptr}[0] = 0, \texttt{ptr}[i] = \texttt{ptr}[i-1]+\delta[i-1]$, counting rows starting from $0$, and with $[\cdot]$ being the conventional array access operation. The column coordinate of each non-zero is stored in the idx $\in \mathbb{N}^{nnz}$ array, while val $\in \mathbb{R}^{nnz}$ stores the values of non-zero entries. The non-zero entries of row $i$ (equivalently, the neighbours of vertex $i$ in a graph) are stored from $\texttt{idx}[\texttt{ptr}[i]]$ (included) to $\texttt{idx}[\texttt{ptr}[i+1]]$ (excluded). Iterating over all non-zeros requires only $2 \cdot nnz + N$ array accesses, versus the $3 \cdot nnz$ of COO. However, CSR introduces data-dependent array accesses (i.e. $\texttt{idx}[\texttt{ptr}[i]]$) which are more difficult to pipeline and optimize in hardware.

#### Compressed Sparse Column (CSC)

Compressed Sparse Column (CSC) is a data structure similar to CSR, with the difference that ptr contains the number of non-zeros in each column of the matrix, instead of the number of non-zeros in each row. CSC can be efficient when storing rectangular matrices with more rows than columns (so that ptr is smaller), or if it is necessary to iterate over the in-neighbors of a vertex (i.e. the elements of a column), as in PageRank (PR). Storing a matrix as CSC is identical to storing the transpose of the matrix as CSR. This property can be useful in numerical algorithms that require access to transposed matrices (e.g. sparse matrix factorization).

#### Banded sparse matrix

A banded matrix is a sparse matrix where the non-zeros are located in diagonal patterns, usually, but not necessarily, in correspondence with the main diagonal. These matrices can be stored with one or more arrays containing the diagonals' elements with non-zero entries. For example, a banded matrix with non-zeros along the main diagonal and the two diagonals immediately above and below the main diagonal (i.e. a tridiagonal matrix) is represented with three arrays, for a total of $3 \cdot N - 2$ values. Such matrices are common in engineering and numerical problems where matrices have a diagonal structure (e.g. Finite Element Method (FEM)) but are less relevant to our applications in graph analytics and recommender systems.

#### Block Compressed Row (BSR)

Some sparse matrices present a block-like structure, with small patches of dense values as in Figure 2.3. This phenomenon is common in some nu-

merical applications and sometimes when dealing with graphs with strong community structures. To represent these sparse matrices, we wrap them with additional data structures. For example, we can use an array of references to the dense matrices. Alternatively, we can use a CSR-like format where the `ptr` array stores the cumulative number of non-zeros in each dense matrix, and dense matrices are stored in a contiguous array. In this case, `ptr` identifies the start and end of each dense matrix. This representation is called Block Compressed Row (BSR) and assumes that dense matrices are disposed along the main diagonal. Alternative data structures, able to store dense blocks with arbitrary locations, do however exist.

$$\mathrm{A} = \begin{bmatrix} \mathrm{A}_1 & 0 & \cdots & 0 \\ 0 & \mathrm{A}_2 & 0 & \vdots \\ \vdots & 0 & \ddots & 0 \\ 0 & \cdots & 0 & \mathrm{A}_B \end{bmatrix}$$

**Figure 2.3:** *Block diagonal matrix. Each of the $B$ square sub-matrices $\mathrm{A_i}$ is dense.*

**Other formats for sparse matrices**

Besides the data structures presented above, there exist other formats used to store sparse matrices. Such formats are usually optimized for a specific target architecture or to enable properties that are not available in more traditional data structures, such as fast insertion of new non-zeros. On CPUs, Packed Compressed Sparse Row [208], STINGER [52] and GraphTinker [92] are data structures optimized for dynamic sparse matrices and graphs, while Dynamic Compressed Sparse Row enables similar updates on GPUs [103]. Chapter 5 will introduce Block-Streaming CSR (BS-CSR), a representation for streaming reduced-precision sparse linear algebra, optimized for Field-Programmable Gate Arrays (FPGAs). Recently, there have been attempts to unify all these representations in the context of sparse tensors through the notion of *fiber tree* to provide a more abstract matrix representation that can provide the best sparse representations for each tensor dimension [189].

### 2.2.3 Implementing SpMV with COO and CSR

To further clarify the differences between COO and CSR, and to show what a simple implementation of SpMV looks like, we present the pseudo code of SpMV implemented using COO (Algorithm 1) and CSR (Algorithm 2).

**Algorithm 1** Pseudo-code of COO SpMV. We assume $a = 1, b = 0$

**Require:** Input Matrix A $= (x, y, val) \in \mathbb{R}^{N \times M}$ in COO format
**Require:** $vec \in \mathbb{R}^M$, dense vector multiplied by the matrix
**Require:** $res \in \mathbb{R}^N$, dense vector used as output

```
1: function COO-SpMV(ptr, idx, val, vec, res, nnz)
2:     for i in [0, nnz] do
3:         res[x[i]] += val[i] · vec[y[i]]
4:     end for
5:     return res
6: end function
```

**Algorithm 2** Pseudo-code of CSR SpMV. We assume $a = 1, b = 0$

**Require:** Input Matrix A $= (ptr, idx, val) \in \mathbb{R}^{N \times M}$ in CSR format
**Require:** $vec \in \mathbb{R}^M$, dense vector multiplied by the matrix
**Require:** $res \in \mathbb{R}^N$, dense vector used as output

```
1: function CSR-SpMV(ptr, idx, val, vec, res, N)
2:     for i in [0, N] do
3:         for j in [ptr[i], ptr[i + 1]) do
4:             res[i] += val[j] · vec[idx[j]]
5:         end for
6:     end for
7:     return res
8: end function
```

We see how COO requires just an external loop but more memory accesses. On the other hand, CSR requires two loops but fewer accesses overall. To formalize these concepts, we define different types of memory accesses and the concept of operational intensity.

**Sequential access:** the simplest type of array access, typical of linear array scans. We find such accesses, for example, in the matrix accesses at line 3 of COO SpMV ($x[i]$, $y[i]$, $val[i]$). Sequential accesses are easy to predict and pipeline, even when the number of iterations is not known statically.

**Data-dependent access:** these are linear accesses whose number depends on some value obtained through another array access. One example is the inner loop of CSR SpMV at line 4 ($val[ptr[i]]$, $idx[ptr[i]]$). The `idx` and `val` vectors are traversed linearly, but the iteration boundaries depends on values of `ptr`, which are not known in advance.

Optimizing such accesses is difficult and usually involves prefetching of values in `ptr`. When dealing with low-degree sparse matrices, data-dependent accesses can be costly, as the pipelined execution of data-dependent loops will stall at the end of each loop.

**Random access:** the most problematic type of array access, performed at a location defined by another array access, making its location impossible to know in advance. Accesses to the dense `x` vector are completely random, in both COO and CSR SpMV. Caches can help if random accesses present a locality of some kind (for example, we tend to access values located in similar parts of the array), but otherwise, very little can be done to improve their performance.

**Definition 2.3** (Operational intensity)**.** Also known as *arithmetic intensity*, operational intensity is the ratio between computation and memory traffic [184]. The precise evaluation of this value is somewhat problem-dependent, but we can see it as the number of mathematical operations done by an algorithm (e.g. in FLOPs) over the amount of data read from memory (in bytes). Alternatively, we can see it as the number of bytes read from memory for each input data element being processed or for each output data element being computed.

SpMV has very low arithmetic intensity. In COO SpMV processing a non-zero entry requires five memory accesses, and only two arithmetic operations. The arithmetic intensity of CSR SpMV depends on the average degree. However, we still require more than four memory accesses (and only two arithmetic operations) to process each non-zero. As we cannot avoid reading all the input data, we need other optimizations: hiding the latency of memory transactions ($\approx$100 cycles for a single transaction, on FPGAs) using pipelined hardware designs, minimizing the number of memory transactions by reading larger data packets with each transaction, reducing the impact of random memory accesses by using low-latency caches.

## 2.3 Notions of Numerical Mathematics

Sparse linear algebra is often encountered in large-scale iterative numerical algorithms, such as Personalized PageRank (PPR) (Section 3.3) and the Lanczos algorithm (Section 4.3). Whether these algorithms converge to a solution or whether numerical errors and other sources of instability prevent the algorithms from producing meaningful results is determined by precise mathematical definitions and properties. Section 2.3.1 introduces the main

concepts of numerical stability and convergence. Then, one must consider how the numerical representation of real numbers on hardware can introduce other approximations, especially when working with matrices containing millions or billions of values, not only in iterative algorithms. As such, we present a brief overview of floating-point and fixed-point number representations, with a focus on their accuracy and the impact of rounding techniques (Section 2.3.2). Finally, we summarize the main metrics used to assess the quality of recommender systems and information retrieval in the context of approximate computations (Section 2.3.3).

### 2.3.1 Notions of Numerical Stability

We briefly present the notions of well-posed problems, condition number, stability, and convergence, which are necessary for the evaluation of iterative algorithms such as PPR and Lanczos. Further details about the definitions in this section are found in Quarteroni et al. [167].

Define the abstract problem of finding $x$ such that $P(x, d) = 0$, with $d \in D$ being the set of data from which the solution depends (for example, the input matrix and the input query), taken from the set of admissible data $D$ (for example, all possible input matrices and queries). $P$ is the functional relation between $d$ and $x$ (for example, the recurrence equation of PR, Section 3.3). In our context, $P(x, d) = 0$ is a *direct problem* as $P$ and $d$ are given, while $x$ is unknown.

**Definition 2.4** (Well-Posed Problems)**.** $P(x, d) = 0$ is *well-posed* if:

- A unique solution $x$ does exist.
- The solution $x$ changes with continuity if $d$ changes with continuity.

Well-posed problems are also called *stable* problems. A problem that is not well-posed is *ill-posed* or *unstable*. The second requirement of well-posed problems is formalized as

$$\begin{cases} P(x,d) = 0 \implies \exists \delta : \;\; d + \delta d \in D, \;\; P(x + \delta x, d + \delta d) = 0 \\ \exists K_0 = K_0(d) \;\; s.t. \;\; \forall \delta d : \;\; d + \delta d \in D, \;\; \|\delta x\|_x \leq K \|\delta d\|_d \end{cases} \tag{2.1}$$

In Equation (2.1), $\|\cdot\|_x, \|\cdot\|_d$ are arbitrary norms that are suitable for $x$ and $d$. The first equation states that, given a solution of the problem, a small change $\delta$ to $x$ and $d$ will still produce a valid solution. The second equation implies that finite changes to $d$ produce a change to $x$ that must

be bound by a finite term $K_0$. In other words, a finite change to $d$ cannot produce an infinitely large change in $x$.

An example of a well-posed problem is finding the eigenvalues of a symmetric matrix, as scaling the input matrix (i.e. $d$) by a constant scales the eigenvalues (i.e. $x$) by the same amount. On the other hand, finding the real roots of a polynomial is an ill-posed problem, as changes in the parameters of the polynomial might change the number of real solutions. For example, $x^2 + x + a = 0$ could have $0$, $1$ or $2$ real solutions.

The term $K_0$, which expresses the continuous dependency of $x$ from $d$, is not arbitrary. Instead, we can introduce the following definitions.

**Definition 2.5** (Relative condition number)**.** For the problem $P(x, d) = 0$, the *relative condition number* is

$$K(d) = sup\left\{ \frac{\|\delta x\|_x / \|x\|_x}{\|\delta d\|_d / \|d\|_d},\ \delta d \neq 0,\ d + \delta d \in D \right\} \tag{2.2}$$

Intuitively, $K(d)$ expresses the maximum proportional change on $x$ that a small change on $d$ can produce.

**Definition 2.6** (Absolute condition number)**.** If $\|x\|_x = 0$ or $\|d\|_d = 0$, we define the *absolute condition number* as

$$K(d) = sup\left\{ \frac{\|\delta x\|_x}{\|\delta d\|_d},\ \delta d \neq 0,\ d + \delta d \in D \right\} \tag{2.3}$$

Informally, a large $K(d)$ will produce an ill-posed problem in $d$. Being ill-conditioned is not a property of a given numerical algorithm: one can devise stable and unstable algorithms to solve a well-posed problem. The concept of a stable algorithm, or stable numerical method, is defined similarly to a stable problem. To do so, we need to define what a numerical method is and what properties it can have.

Given a well-posed problem $P(x, d) = 0$, an approximate numerical method for the resolution of $P(x, d) = 0$ creates a sequence of approximate problems $P_n(x_n, d_n) = 0,\ n \geq 1$, under the assumption that the sequence will converge to the exact solution, i.e. $x_n \to x$, $d_n \to d$, and $P_n \to P$ for $n \to \infty$.

**Definition 2.7** (Consistency of numerical methods)**.** We say that a numerical method $P_n(x_n, d_n) = 0$ is *consistent* if

$$\lim_{n \to \infty} P_n(x, d) - P(x, d) \to 0 \tag{2.4}$$

Similarly, $P_n(x_n, d_n) = 0$ is *strongly consistent* if $\forall n\ P_n(x_n, d) = 0$, i.e. $x$ is always a solution for each approximate relation $P_n$.

**Definition 2.8** (Stability of numerical methods)**.** A numerical method is *stable* (or *well-posed*) if and only if the following conditions hold.

1. $\forall n, P_n(x_n, d) = 0$.
2. The sequence $x_n$ is unique and reproducible, i.e. the sequence of approximate solutions produced by $P_n$ with input $d$ is always $x_n$.
3. The sequence $x_n$ changes with continuity if $d$ changes with continuity, with the same definition in Equation (2.1).

**Definition 2.9** (Convergence of numerical methods)**.** A numerical method is *convergent* if

$$\begin{aligned} &\forall \varepsilon > 0\ \exists n_0(\varepsilon),\ \delta = \delta(n_0, \epsilon) \text{ such that} \\ &\forall n > n_0(\varepsilon),\ \forall \delta d_n :\ \|\delta d\|_d \leq \delta \implies \|x(d) - x_n(d + \delta d_n)\|_x \leq \varepsilon \end{aligned} \tag{2.5}$$

In other words, for every arbitrarily small $\varepsilon$, it exists an iteration $n_0$ and a value $\delta$ such that for every iteration $n$ after $n_0$ and for each variation of $d_n$ smaller than $\delta$, it is guaranteed that the approximate solution $x_n$ has error $\leq \varepsilon$. To measure convergence error of $x_n$ to $x$, one can use *absolute error* and *relative error*, defined as

$$E_{abs} = \|x - x_n\|_x \qquad E_{rel} = \frac{\|x - x_n\|_x}{\|x\|_x}\ (with x \neq 0) \qquad (2.6, 2.7)$$

### 2.3.2 Machine Real Number Formats

Real numbers have many possible representations on hardware, each with its trade-offs in terms of precision, complexity, and flexibility. Floating-point arithmetic is arguably the most widely known and employed format in numerical mathematics, with a generic number $x$ encoded as

$$x = (-1)^s \cdot (0.m_1 m_2 \ldots m_M) \cdot \beta^{e_1 e_2 \ldots e_E} \tag{2.8}$$

with $m = m_1 m_2 \ldots m_M \in \mathbb{N}$ representing the *mantissa*, i.e. the fractional part of the number, expressed in base $\beta$, and $e = e_1 e_2 \ldots e_E \in \mathbb{N}$ being the *exponent* of the number, also in base $\beta$ [85, 167]. We define $M \in \mathbb{N}$ as the number of significant figures in $x$, and $\beta \in \mathbb{N}$ is the basis

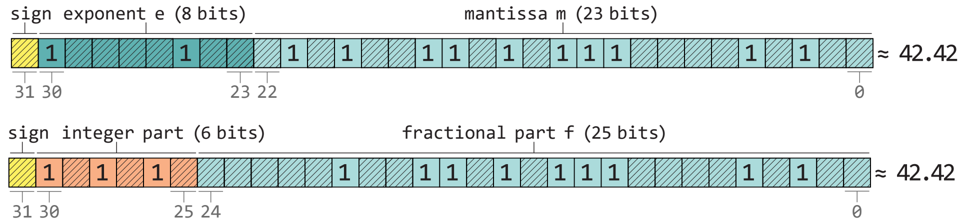



**Figure 2.4:** *Floating-point and fixed-point representations of 42.42, using the IEEE 754 single-precision 32-bits floating-point representation, and a* `S6.25` *fixed-point representation. Neither representation can encode 42.42 exactly. The floating-point representation stores 42.41999816894531, while the fixed-point representation, thanks to its additional two bits for the fractional part, stores the more accurate value of 42.41999998688697815.*

of the number (in binary notation, $\beta = 2$). We have that $s$ is the *sign* bit $\in \{0, 1\}$, and $N = M + E + 1$.

The value of $e$ defines an interval between two consecutive powers of $\beta$, i.e. $[\beta^e, \beta^{e+1})$. The value $m$ partitions the interval $[\beta^e, \beta^{e+1})$ into $\beta^M$ steps of size $(\beta^{e+1} - \beta^e)/\beta^M$. For example, if $M = 3$, $\beta = 2$ and $e = 0$, we divide $[0, 1)$ into $2^3 = 8$ steps of size $1/8 = 0.125$. On the other hand, if $e = 3$, we partition $[8, 16)$ in $4$ steps of size $8/8 = 1$. The bigger the exponent, the lower the resolution of the floating-point representation is.

The IEEE 754 standard provides a commonly accepted definition for the parameters encountered in floating-point representations. In single-precision floating-point arithmetic (F32), $E = 8$, $M = 23$, and $-125 \leq e \leq 128$; in double-precision floating-point arithmetic (F64), $E = 11$, $M = 52$, $-1021 \leq e \leq 1024$. In other words, IEEE 754 expresses single-precision floating-point numbers as

$$x = (-1)^s \cdot (1.m) \cdot 2^{e-127} \tag{2.9}$$

with $m \in [0, 2^{23}-1]$, $e \in [0, 2^8-1]$. For convenience, this representation normalizes the mantissa to represent the interval $[1, 2)$, and the exponent to be in $[-127, 128]$.

Conversely, one can simply fix the number of bits used for the integer and fractional part of a real number. The fixed-point representation of a number $x$, encoded with a total of $N = I + F + 1$ bits, of which $I$ are used for the integer part and $F$ are used for the fractional part, is given by

$$x = (-1)^s [x_{F+I-1} x_{F+I-2} \dots x_F \cdot x_{F-1} x_{F-2} \dots x_0] \tag{2.10}$$

which is equivalent to the following form

**Table 2.2:** *Summary statistics of IEEE 754 floating-point and fixed-point representations. Floating-point values are retrieved from Hopkins et al. [85].*

| | **F64** | **F32** | **S8.23** | **U1.31** | **U1.19** |
|---|---|---|---|---|---|
| **Accuracy** | $2^{-52}$ | $2^{-23}$ | $2^{-23}$ | $2^{-31}$ | $2^{-19}$ |
| **Minimum (exact)** | $2^{-1022}$ | $2^{-126}$ | $2^{-23}$ | $2^{-31}$ | $2^{-19}$ |
| **Minimum (approx.)** | $2.225 \cdot 10^{-308}$ | $1.175 \cdot 10^{-38}$ | $1.19 \cdot 10^{-7}$ | $4.65 \cdot 10^{-10}$ | $1.9 \cdot 10^{-6}$ |
| **Maximum (exact)** | $(2 - 2^{-52}) \cdot 2^{1023}$ | $(2 - 2^{-23}) \cdot 2^{127}$ | $2^8 - 2^{-23}$ | $2 - 2^-31$ | $2 - 2^-19$ |
| **Maximum (approx.)** | $1.798 \cdot 10^{308}$ | $3.403 \cdot 10^{38}$ | $255.99\dots$ | $1.9999\dots$ | $1.9999\dots$ |

$$x = (-1)^s \cdot \beta^{-F} \sum_{j=0}^{N-2} x_j \beta^j \tag{2.11}$$

In this case, the size of the step between any two numbers is *fixed*, and does not depend on the magnitude of the number. In this thesis, we denote as `SI.F` a signed fixed-point with $I$ bits of integer part and $F$ bits of fractional parts (e.g. `S8.23`). An unsigned fixed-point representation is denoted as `UI.F`. Figure 2.4 shows how we can represent the value 42.42 using IEEE 754 32-bits floating-point arithmetic, and with a `S6.25` fixed-point representation. Neither representation is 100 % accurate, although the fixed-point representation has a $100\times$ smaller error thanks to the additional bits used in the fractional part.

Table 2.2 summarizes the characteristics of IEEE 754 FP32, FP64, signed fixed-point with 8 bits for the integer part, unsigned 32-bits fixed-point with 1 bit of integer part, and 20-bits unsigned fixed-point with 1 bit of integer part. For each representation, we show its $\varepsilon$ (i.e. the smallest difference between two values that can be encoded), its minimum and maximum values, showing their exact and approximate expression.

In some situations, reduced-precision fixed-point arithmetic is provably more accurate than single-precision floating-point arithmetic: for example, computations with bound values such as PR, where fractional values are bound in $[0, 1]$. There, an unsigned fixed-point representation with $I = 1$ and $F = 25$ is more accurate than single-precision floating-point arithmetic, despite being 20 % smaller, as its $\varepsilon$ is $2^{-25}$ instead of $2^{-23}$.

From Equation (2.10), it follows that to a real number represented in fixed-point format, it corresponds an integer whose digits are interpreted accordingly to the fixed-point definition. Basic arithmetic operations on fixed-point arithmetic are trivially implemented by operating on the underlying integer representation. Given a real number $x = [s, i, f]$ in fixed-point format, we can define the following operations

**Addition:** $[s, i, f] + [s, i, f] = [s, i, f]$

**Subtraction:** $[s, i, f] - [s, i, f] = [s, i, f]$

**Multiplication:** $[s, i_a, f_a] - [s, i_b, f_b] = [s, i_a + i_b, f_a + f_b]$

Such operations are implemented using hardware integer arithmetic. Some additional considerations are required only in the case of multiplication, as the output of multiplication has length $s + i_a + i_b + f_a + f_b$, instead of $s + i_a + f_a$ or $s + i_b + f_b$ (with slight abuse of notation, using the number's components to denote their length). In this case, one must manipulate the result to obtain a representation with the same length of the original values, introducing a possible loss of precision.

There are multiple ways to truncate a number expressed in a fixed-point representation. Here, we summarize some of them [85, 214]. In the following examples, we convert to a `S1.3` representation an arbitrary fixed-point number with $I \geq 1$ and $F > 3$.

**Round-to-nearest:** it rounds to the closest representable value. In the case of ties, one can round-up, round-down, or round-to-even (i.e. round to the nearest value with an even least significant digit). With round-up or round-to-even, $1.25 \rightarrow 1.5$ and $-1.25 \rightarrow -1.0$.

**Round-to-nearest, ties-to-zero:** it rounds to the closest representable value. In the case of ties, rounding is towards zero. Redundant bits for positive values are deleted, while for negative values, one has to add the least significant bits until the nearest representable value, getting closer to zero. For example, $1.25 \rightarrow 1.0$ and $-1.25 \rightarrow -1.0$.

**Round-to-nearest, ties-to-minus-infinity:** same as round-down, it rounds to the closest representable value. In the case of ties, rounding is towards $-\infty$. Redundant bits for positive values are deleted, while for negative values one has to set the least significant bits, getting closer to $-\infty$. For example, $1.25 \rightarrow 1.0$ and $-1.25 \rightarrow -1.5$.

**Round-to-nearest, ties-to-plus-infinity:** same as round-up, it rounds to the closest representable value. In the case of ties, rounding is towards $\infty$. Redundant bits for positive values are set, while for negative values, the least significant bits are truncated, getting closer to $\infty$. For example, $1.25 \rightarrow 1.0$ and $-1.25 \rightarrow -1.5$.

**Round-to-nearest, ties-to-infinity:** opposite of ties-to-zero, it rounds to the closest representable value. In the case of ties, rounding is to-

wards $-\infty$ and $\infty$. Redundant bits for both positive and negative values are set, getting closer to $\infty$ and $-\infty$, respectively. For example, $1.25 \rightarrow 1.5$ and $-1.25 \rightarrow -1.5$.

**Truncation:** simply remove the additional bits, as if they did not exists. It always performs a rounding to $-\infty$, not only in the case of ties. For example, $1.25 \rightarrow 1.0$ and $-1.25 \rightarrow -1.5$.

**Truncation-to-zero:** remove the additional bits for positive vales, and set the least significant bits for negative values, rounding to zero. For example, $1.25 \rightarrow 1.0$ and $-1.25 \rightarrow -1.0$.

Finding the best rounding technique is not trivial, and it is strongly dependent on the application domain. In the applications presented in the following chapters, we found that simple truncation works best, as other methods required additional hardware resources without translating to faster execution or better numerical accuracy.

The reduced dynamic range of low-precision fixed-point arithmetic can conceptually lead to faster convergence: intuitively, there are fewer numbers than the output values can take. At the same time, low precision might result in no convergence at all or meaningless results. As a simple example, consider the task of retrieving from a collection of vectors the ones with the highest cosine similarity given an input vector (similarly to the problem in Chapter 5). Suppose the number of vectors is larger than the different similarity values that can be encoded with a given numerical representation. In that case, some of them will inevitably have the same similarity value.

For example, when using 10-bit fixed-point arithmetic to encode vectors' values and similarity values, it is possible to encode only $2^{10} = 1024$ different similarity values. Suppose we have millions of vectors to compare against our input vector. In that case, hundreds, if not thousands of vectors will likely have the same similarity and be indistinguishable from each other. In real applications, the effect might be less pronounced as similarities often follow a Zipfian distribution [163], with very few vectors having high similarity and the great majority of vectors having little-to-zero similarity. Even a very low bit-width can provide reasonably good results in such a situation. From an architectural perspective, one can instead minimize this problem by using mixed-precision hardware designs. Values are stored in off-chip memory using low bit-width to minimize memory utilization and allow for greater operational intensity, as more values can be loaded with a single memory transaction. Then, arithmetic operations on these values are done with higher precision arithmetic, and results are converted to the low-precision format only when stored in off-chip memory.

We employ this technique in Chapter 5, achieving great accuracy even when processing millions of embeddings stored with low-precision arithmetic.

From a hardware perspective, fixed-point arithmetic is vastly cheaper than floating-point arithmetic, being based on simpler integer operations: a 32-bit integer adder, almost directly equivalent to a fixed-point adder, demands 9× less energy and 30× less area than a 32-bit floating-point adder [85]. Quantization from reduced-precision floating-point to integer representations is a common strategy in DL, with quantization from FP4 (4-bit floating-point arithmetic) to INT4 achieving 5× faster training time with no visible accuracy degradation [35, 74, 174, 188]. Strong quantization, fixed-point arithmetic, and even binarization of weights are also common techniques used for inference, and sometimes training, acceleration of DL on FPGAs and custom hardware architectures [136, 193], achieving real-time classification even on embedded hardware. Reduced-precision arithmetic is commonly employed in GPUs also in the context of DL and numerical mathematics. Yan et al. [217] achieves 1.5× speedup on matrix multiplication kernels using half-precision floating-point arithmetic on Tensor Cores [12], while Haidar et al. [79] show a 4× speedup in the resolution of dense linear systems of equations using FP64-to-FP16 mixed-precision, also on Tensor Cores. Micikevicius et al. [133] obtain 2–6× DL training using mixed-precision training (FP16 and FP32), compared to a pure FP32 implementation, with no performance loss.

In the specific context of IR, where it is common to rank items from a collection and compute item similarities, a strong precision reduction might be unsuitable (as in the example below, with 10-bit fixed-point values). Still, there is often no need to employ 32-bit precision fixed-point arithmetic or even floating-point, as shown in the next chapters of this thesis: it will be shown how 20-bit fixed-point arithmetic is often enough, even when working with matrices with hundreds of millions of values.

### 2.3.3 Evaluating Accuracy in Information Retrieval

In this thesis, we focus on applications of sparse linear algebra – specifically, of SpMV – in the context of graph analytics, recommender systems, and information retrieval, using algorithmic approximations and reduced numerical precision to improve their performance. To evaluate the quality of our accelerated implementations, we cannot simply refer to the intrinsic inaccuracy of different number representations, as in Section 2.3.2. In other words, simply measuring how close the numerical values produced by our algorithms to a gold-standard reference (produced, for example, by

a double-precision floating-point implementation) might give a misleading picture of the real-world effectiveness of our implementations. This phenomenon occurs because algorithms encountered in these application domains do not require perfect numerical accuracy but can tolerate approximate results to a certain degree. For example, consider the task of recommending to a user $20$ movies that the user might enjoy watching. First, the accuracy of movies ranked high on the list will be more important than those of a movie ranked low. It is unlikely that the user might notice that the $20^{th}$ is more appealing than the $19^{th}$, while an error on the top recommendations will have more impact. Moreover, we do not even care about the individual score of each movie, but we only require its relative order with respect to the other movies to be correct. As such, it is common to evaluate different metrics: some simply take into account how many highly-ranked elements are correctly retrieved, others also judge if their relative order is correct, or introduce a stronger penalty for highly-ranked items that have been mispredicted [173]. Most metrics are not evaluated on the entire collection of items that can be ranked, but only for the Top-K, i.e. the K items with the highest ranking (e.g. Top-10, Top-20, etc.). Clearly, it is usually irrelevant how well movies (or products, or web pages) in the $1000^{th}$ or $10000^{th}$ positions are ranked. We define as $N$ the number of items in the collection (e.g. $N = 10^6$ or more), while $K$ is the number of elements relevant to the ranking (e.g. 10, 20, etc.). The following paragraphs summarize the metrics used later in our analyses.

**Number of errors**

The simplest way to evaluate a ranking, i.e. an ordered list of identifiers (such as the titles of the $20$ movies), is to compare it to a gold reference (e.g. the ranking obtained by an implementation with no approximations), and count how many elements are placed in the wrong place. Given the ranking $\mathrm{y} \in \mathbb{N}^K$ and the gold-reference $\widehat{\mathrm{y}} \in \mathbb{N}^N$, with $K \leq N$, the number of errors $\varepsilon$ is $\varepsilon = \sum_{i=1}^{i=K} \mathbb{1}\{\mathrm{y} \neq \widehat{\mathrm{y}}\}$, with $\mathbb{1}\{\mathrm{y} \neq \widehat{\mathrm{y}}\}$ being an indicator function such that $\mathbb{1}\{.\} = 1$ if $\{\mathrm{y} \neq \widehat{\mathrm{y}}\}$ and otherwise $\mathbb{1}\{.\} = 0$. This accuracy metric is very coarse-grained, as a single mistake can greatly affect the overall ranking. For example, if the correct Top-4 ranking is $\{2, 4, 8, 6\}$ and our implementation retrieves $\{4, 8, 6, 2\}$, this accuracy metric will report four errors, even though only a single value is displaced.

**Edit Distance**

A more robust accuracy metric, *edit distance* compares two rankings in terms of how many operations (insertion, deletion, substitution) are necessary to transform one ranking into the other [119] It can deal with value replacements and ordering shifts: in the previous example, the edit distance is just 1, as we can insert the value 2 at the beginning of the ranking obtained by our implementation, and ignore values located after the first 4 (as we consider the Top-4 items, in the example).

**Normalized Discounted Cumulative Gain**

Normalized Discounted Cumulative Gain (NDCG) [94] is a common metric used in IR to evaluate ranking *quality* of search engines and recommender systems. This metric dampens the relevance of a given item by a logarithmic factor dependent on its position. Items contribute differently to the gain of the ranking depending on the position that they are given: highly ranked items contribute more heavily to the cumulative gain compared to the lower-ranked ones. Given the golden ranking $\widehat{\mathrm{y}} \in \mathbb{N}^N$, and an approximate ranking $\mathrm{y} \in \mathbb{N}^K$, the relevance $rel_i$ of item $y_i \in \mathrm{y}$ is its position in $\widehat{\mathrm{y}}$, i.e. $rel_i = min\{j \mid \hat{y}_j = i\}$. The relevance values can, in principle, be obtained through other means (e.g. by asking them to a user), but simply using each item's positions in the golden ranking is also effective when dealing with large collections of items. Then, we define Discounted Cumulative Gain (DCG) as in Equation (2.12). The Ideal Discounted Cumulative Gain (IDCG), instead, is the DCG computed on $\widehat{\mathrm{y}}$ with respect to itself. DCG is normalized by dividing it by the IDCG, as in Equation (2.13).

$$\mathrm{DCG} = \sum_{i=1}^{K} \frac{rel_i}{log_2(i+1)} \qquad \mathrm{nDCG} = \frac{DCG}{IDCG} \qquad (2.12, 2.13)$$

**Mean Average Error**

Mean Absolute Error (MAE) evaluates the difference between the ranking scores (i.e. the numerical values produced by a ranking algorithm, from which rankings are produced) produced by a gold-reference implementation and the ones of an approximate implementation. Given the golden scores $\widehat{\mathrm{x}} \in \mathbb{R}^K$, and an approximate ranking $\mathrm{x} \in \mathbb{R}^K$, MAE is defined as in Equation (2.14). If instead of using the absolute difference of scores, we compute their squared difference, we obtain the Root Mean Square Error

(RMSE) norm, as in Equation (2.15). These two equations represent the distance of the ranking scores in terms of L1 norm and L2 norm.

$$\text{MSE} = \frac{1}{K}\sum_{i=1}^{K} |\hat{x}_i - x_i| \qquad \text{RMSE} = \sqrt{\frac{1}{K}\sum_{i=1}^{K} (\hat{x}_i - x_i)} \quad (2.14, 2.15)$$

#### Precision

Precision measures the correctness of rankings in the Top-K without keeping their relative order into account. In other words, it counts how many items in $\mathrm{y} \in \mathbb{N}^K$ also appear in $\widehat{\mathrm{y}}_\mathrm{K} \in \mathbb{N}^K$, the first K positions of the gold-reference ranking $\widehat{\mathrm{y}} \in \mathbb{N}^N$. Considering $\mathrm{y}$ and $\widehat{\mathrm{y}}_\mathrm{K}$ as sets of size K (as the order of their elements does not matter), precision $\pi$ is defined as

$$\pi = \frac{|\mathrm{y} \cap \widehat{\mathrm{y}}_\mathrm{K}|}{|\mathrm{y} \cup \widehat{\mathrm{y}}_\mathrm{K}|} \quad (2.16)$$

#### Kendall's Tau

Kendall's $\tau$ is a correlation coefficient commonly used in hypothesis testing to statistically validate the ordinal association of ordinal random variables, and can be seen as a measure of correlation between two rankings [180], as it penalizes out-of-order items in a ranking when compared to a reference ranking. Let $\mathrm{t} \in \mathbb{N}^N$ be a vector where each position represents an item in the collection, and $t_i \in [1, N]$ is its rank, produced by our approximate ranking algorithm. Let $\widehat{\mathrm{t}} \in \mathbb{N}^N$ be an array defined in the same way, but with rankings produced by the golden-reference algorithm. In other words, $\mathrm{t}$ and $\widehat{\mathrm{t}}$ are defined as the $rel$ vector in NDCG.

Consider all the pairs $(t_i, t_j) : t_i, t_j \in \widehat{\mathrm{t}}$ and $(\hat{t}_i, \hat{t}_j) : \hat{t}_i, \hat{t}_j \in \widehat{\mathrm{t}}$, with $i < j$. These two pairs are *concordant* if $t_i < t_j$ and $\hat{t}_i < \hat{t}_j$, or if $t_i > t_j$ and $\hat{t}_i > \hat{t}_j$. Otherwise, the two pairs are *discordant*. Any pair containing elements that appear in $\widehat{\mathrm{t}}$ but not in $\mathrm{t}$ (or in $\mathrm{t}$ but not in $\widehat{\mathrm{t}}$) is discordant. $C$ is the number of concordant pairs, $D$ is the number of discordant pairs.

Then, Kendall's $\tau$ is defined as $\tau = (C - D)/\binom{N}{2}$. To measure $\tau$ over an arbitrary number $K \leq N$ of items, simply select from $\mathrm{t}$ and $\widehat{\mathrm{t}}$ the K highest-ranked items in each vector.

## 2.4 Hardware for Sparse Linear Algebra

Sparse linear algebra has drawn the interest of the hardware research community for decades, with optimizations for storage and compilers dating back to the '70s [50]. This interest is now stronger than ever. The recent popularity explosion of graph analytics and DL demands novel techniques to handle sparse matrices with millions, billions, or possibly trillions of non-zero entries, representing web graphs, social networks, or databases of embeddings used in recommender systems [43]. In the field of DL alone, the number of publications related to sparse computing has increased by $10\times$ over the past 10 years [83].

This section presents an overview of how different hardware architecture, from traditional Central Processing Units (CPUs) to highly customized Domain-Specific Architectures (DSAs) and Application-Specific Integrated Circuitss (ASICs), have been used to improve the performance, efficiency, and scalability of sparse computations. We discuss the advantages and disadvantages of each, with a focus on FPGAs, and why they have been chosen as hardware of choice for this work. Finally, we present a brief outlook of what trends might be observed in the near future in terms of hardware acceleration of sparse computations.

### 2.4.1 General-purpose CPUs (GPCPUs)

General-purpose Central Processing Units (GPCPUs) have always been the starting point for optimized sparse linear algebra, with efforts to optimize CPU compilers dating back to the early '70s [50,65,78]. Nowadays, basic sparse linear algebra routines have been standardized in the Sparse BLAS specification, of which there exists many optimized CPU implementations [51]. Among libraries the implement all or part of the Sparse BLAS specification, we can mention SPBLAS [51, 145], Eigen [71], Armadillo [176], IntelMKL [88, 200], and OSKI [199]. On a similar note, the GraphBLAS specification defines basic mathematical routines for operations on graphs, and most of them rely on sparse linear algebra [28,42,101].

Using CPUs to implement sparse linear algebra routines is arguably the most intuitive choice: they provide quick random memory accesses, support for complex data structures, and the best performance in sequential operations. At the same time, the presence of complex architectural components such as branch predictors and multi-level caches can be a double-edged sword for the performance of sparse linear algebra computations. This phenomenon is evident when working with matrices stored as CSR that exhibit low average degree ($\delta < 10$) and large degree variance. The

large degree variance will cause the branch predictor to mispredict the number of non-zero values per row, while the low average degree means that the misprediction penalties can outweigh the benefits offered by the deep pipelines of modern CPUs.

Similar considerations occur with caches. In SpMV the sparse matrix is usually traversed linearly, but to each non-zero entry of the matrix, it corresponds an access to the dense vector multiplied by the matrix. The indices of these accesses are usually unpredictable and, especially in graphs, might not follow any visible pattern. As such, caching values of the dense vector might be infeasible, and each access to it will incur into overheads introduced by cache lookups and miss penalties. At the same time, sparse matrices might have a hidden structure that can be exploited. Social graphs, for example, are usually composed of densely connected communities. When traversing such graphs, subsequent memory accesses tend to happen within the same community.

The sub-field of matrix and graph reordering was born to provide algorithms that identify these structures, and enable the hardware to take advantage of them [110, 207]. For example, one can move rows (i.e. vertices) with identical degrees close to each other to reduce the number of branch mispredictions when using a CSR representation. The typical goal of state-of-the-art reordering algorithms such as Gorder is to minimize the number of cache misses, and guarantee that subsequent random memory accesses can be coalesced into a unique memory access through the Miss Status Holding Register (MSHR) [207]. Another recent technique [54] preserves the original community structure of the input graph, observing how strongly connected vertices are often processed within a time from one another. It should be kept in mind that these reordering algorithms introduce a non-negligible overhead, as they need to scan the entire matrix at least once (and often, multiple times). Such pre-processing overhead is irrelevant if the same computations are repeated multiple times on the same input matrix (e.g. with web-graphs that are updated once a day, Section 2.4.4), but are hardly useful to accelerate one-off computations (e.g. a single SpMV on a matrix that will not be used again). Graph reordering algorithms are usually encountered in the context of CPUs, as other architectures (e.g. GPUs, FPGAs) do not present architectural characteristics that are affected by the graph structure, such as deep cache hierarchies, out-of-order execution, and branch prediction. GPUs, however, still show a $1.5 - 1.9\times$ speedup in graph traversal thanks to reordering [61], if using Unified Virtual Memory (UVM) on graphs that do not completely fit into the GPU memory. In this case, the speedup does not come from the GPU hardware itself, but from the min-

imization of data transfer between CPU and GPU. Graph reordering does not affect the performance of the SpMV FPGA hardware designs presented in this work, as we do not resort to any speculative execution.

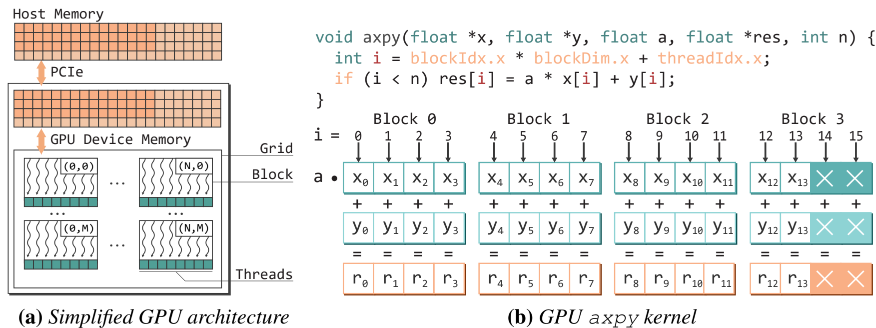
Recently, we observed a resurgence in the high-performance sparse linear algebra through GPCPUs, with the Fugaku supercomputer holding the highest score in the sparse High-Performance Conjugate Gradient (HPCG) benchmark. GPCPUs are still the inevitable choice if large in-memory analyses are required (TB of memory). Moreover, the usefulness of dense linear algebra accelerators for supercomputing has been put in question [48], and similar considerations might also be relevant for sparse workloads as their control flow is more complex and they are inherently more memory intensive. Despite the power of GPCPUs in large-scale sparse computations for supercomputing, hardware accelerators have proven effective times and times again in the context of hyperscaler data centers [33, 96, 97, 122, 123, 164]. The success of these accelerators is explained by repetitive Machine Learning (ML) and IR workloads that do not generally require partitioning the computation over dozens of nodes (if at all) to answer a given query.

### 2.4.2 Graphics Processing Units (GPUs)

GPUs are specialized computer architectures, originally intended for graphics rendering, that can process a large amount of data in parallel by running computational functions – kernels, in the GPU jargon – on each data item (for example, each pixel in an image, or each element in a list), or small groups of data items. GPUs have evolved to become more versatile, and are widely employed in a multitude of fields (computer graphics, artificial intelligence, engineering, and finance), thanks to their parallel computing power [1, 66, 108].

#### The GPU Programming Model

Despite their success and due to their peculiar architecture, GPUs demand a low-level programming model and lack some of the programming tools that are widespread for CPUs. GPU programming is largely done in native languages such as CUDA [175], even though techniques to integrate GPUs within managed languages and environments [59, 105, 156] have been recently proposed. Their architecture is divided into different levels, roughly mapped to the logical levels that divide the computation. In GPUs manufactured by Nvidia, the computation is split across Stream Multiprocessors (SMs), each containing multiple Stream Processors (SPs) (or CUDA cores) that execute the same computational kernel on different data items. Other




**(a)** *Simplified GPU architecture* **(b)** *GPU* `axpy` *kernel*

**Figure 2.5:** *ⓐ Representation of a GPU architecture, with logical subdivision in grid, blocks and threads, as in the terminology of CUDA. ⓑ GPU* `axpy` *kernel written in CUDA, with subdivision in blocks and threads. Each thread processes an array element, and conditional expressions are required to avoid out-of-bounds array accesses.*

GPU manufacturers, such as AMD, and other GPU programming models, such as OpenCL, have similar concepts and slightly different terminology. A simplified representation of the GPU architecture is shown in Figure 2.5a. At a logical level, the computation is mapped to grids, thread blocks, and threads, with each thread processing one or more data items [151].

Figure 2.5b shows a simple `axpy` CUDA kernel, and the subdivision into threads and blocks that is necessary for its computation. The CUDA runtime spawns a number of threads at least equal to the number of elements in the input arrays, but that could be larger due to the chosen grid structure[1]. Each block contains an equal amount of threads: in our `axpy` kernel, if the user decides to have $128$ threads per block, and the array has $1000$ elements, $\lceil 1000/128 \rceil = 8$ blocks will be created, for a total of $128 \cdot 8 = 1024$ threads. Conditional statements are necessary to avoid out-of-bounds array accesses. Coordinating array sizes and grid structure is not trivial: block size is often determined by consideration about performance (and must lie between 32 and 1024), while the number of blocks is either a function of the input size or based on the available SMs [81].

### GPUs for Sparse Linear Algebra

GPUs, thanks to their massive parallelism and memory bandwidth, have become extremely appealing for sparse linear algebra during the last decade. One can find many optimized libraries and implementations of sparse linear algebra routines, including cuSPARSE [139], MAGMA [11, 128, 216], GraphBLAST (for graph routines based on sparse linear algebra) [219],

[1]The grid structure defines how many threads each block contains, and how many blocks are employed

GINKGO [10], and more [125, 218].

Modern GPUs deliver higher memory bandwidth than high-end FPGA accelerator cards or multi-core CPUs. For example, the recent Nvidia A100 has access to $2039\,\mathrm{GB/s}$ of bandwidth through its 80 GB of High Bandwidth Memory (HBM) memory [148]. Given that the previous two generations of high-end GPUs had up to $720\,\mathrm{GB/s}$ (Tesla P100) and $900\,\mathrm{GB/s}$ (Tesla V100) of memory bandwidth, it is reasonable to expect an even more impressive bandwidth growth in the next generations of GPUs. However, existing SpMV implementations are often unable to utilize the available bandwidth fully [143]. Moreover, the limited support for reduced-precision arithmetic constraints GPUs's operational intensity on sparse workloads. However, recent work explores mixed single and double-precision floating-point arithmetic [5], and half-precision floating-point arithmetic [2, 79].

Nvidia's Sparse Tensor Cores in recent Ampere GPUs [37, 149] for fine-grained structured sparsity, as evolution of the original Tensor Cores found in Volta GPUs [12, 130, 217]. However, Sparse Tensor Cores work well only on structured sparsity with relatively high density (e.g. 1 %, orders of magnitude more than what is encountered in real-world graphs – Tables 3.1 and 4.2), while handling low-density unstructured sparsity with the regular 3D grids found in GPUs is far harder, if not impossible. DL workloads, arguably the primary driver of GPUs's current popularity, are starting to adopt sparsification techniques. However, state-of-the-art sparse DL models present only 90–99 % sparsity, with structured sparsity patterns [83]. However, neural networks are getting sparser and sparser, and there's no reason to believe that structured sparsity inherently leads to better accuracy than unstructured sparsity. GPUs might have to introduce new hardware components or even revise their programming model to successfully support unstructured sparsity and remain competitive in DL applications.

### 2.4.3 Field Programmable Gate Arrays (FPGAs)

FPGAs are integrated circuits whose defining characteristic is being programmable at the hardware level after fabrication. Developers can configure the electrical circuits of an FPGA to implement arbitrary logic, operations, and interconnections. FPGAs are widely employed to prototype and test other integrated circuits, such as ASICs, and to create application-specific hardware accelerators that can outperform, in terms of power efficiency, throughput or latency, traditional hardware architectures such as CPUs and GPUs. FPGAs are not only programmable, but also *reconfigurable*, as the digital circuits that they implement can be updated at any

time, usually in a matter of seconds, adapting to new tasks, input data, or requirements.

An FPGA is composed of a bi-dimensional matrix of Configurable Logic Blocks (CLBs), of input/output blocks that link the FPGA to external resources, and of a programmable interconnection network that connect CLBs and other elements of the FPGA [44]. Modern FPGAs also contain other hardware components, such as Digital Signal Processors (DSPs) and several megabytes of on-chip memory, in the form of Block RAM (BRAM) and similar vendor-specific technologies. CLBs are the core elements of FPGAs. They can implement combinational and sequential circuits and provide simple and efficient storage. In turn, CLBs are implemented with a configurable combinational circuit composed of Look-Up Tables (LUTs). A LUT is a memory of $2^n$ bits capable of storing any combinational function with $n$ bits. Such functions are implemented with a truth table: the table stores all possible combinations of bits, and the correct output, given a certain input, is selected through a multiplexer. While the idea behind LUTs is very straightforward, combining multiple LUTs, and multiple CLBs, enables very complex hardware designs. For context, the CLB found in Xilinx UltraScale FPGAs (and also used in Alveo accelerator cards) has 8 LUTs with 6 inputs each, 16 Flip Flops (FFs) to implement sequential logic, multiplexers to combine the results of different LUTs, and arithmetic carry logic [210].

In this thesis, we focus on FPGAs manufactured by Xilinx and adopt the terminology encountered in Xilinx devices. We leverage the Xilinx Vitis HSL toolchain. Other vendors, such as Intel, provide similar technologies with small differences in nomenclature, features, and tools. However, it should be possible to adapt the hardware designs presented in this thesis to FPGAs by other manufacturers, with minimal performance differences.

#### Programming FPGAs with High-Level Synthesis

Programming an FPGA requires a configuration file, called *bitstream*, that specifies the different FPGA configuration points, including the LUTs' truth tables, the interconnection network, the on-chip memory, etc. The bitstream file is the product of a tortuous and vendor-specific synthesis process that begins with a description of the hardware design's functionality.

The traditional way of specifying such hardware description is with a Hardware Description Language (HDL) like VHDL or Verilog, through a functional description of the desired design. Such languages, however, are extremely complex to use and make the resulting design hard to extend to new hardware or to new applications. An alternative, and arguably more

user-friendly, FPGA programming flow, is based on High-Level Synthesis (HLS) tools. HLS tools allow developers to write hardware designs using programming languages such as C++ and OpenCL and translate such high-level representation to HDL. From there, the bitstream generation proceeds as if the design was originally written in HDL, transparently to the developer. HLS programmers use directives to steer the HDL generation process and the following synthesis towards the desired optimizations, such as unrolling small loops with statically-known trip-count or pipelining complex loops to hide the latency of memory transactions. Even more complex optimizations are possible: in this work, we propose data-flow designs, in which the FPGA is programmed with independent but interconnected hardware modules that communicate in a pipelined producer-consumer fashion. The output of one module is provided as input to the next module, and while the second module processes such value, the first module can already move to the next. Such hardware designs are highly effective in streaming computations such as SpMV, where the input is processed as a continuous stream of data without any reuse of values that have already been processed. Overall, the creation of a successful FPGA hardware design requires the following steps.

1. **High-level description** of the hardware design, using languages such as C++or OpenCL (optional step).
2. **HDL description**, or HDL generation if using HLS.
3. **Logic synthesis**, the process of translating HDL into a *netlist*, an implementation in terms of FFs and logic gates.
4. **Technology mapping, packing and placement**, a sequence of steps that progressively lowers the netlist to a more concrete hardware-dependent implementation, based on the target FPGA resources.
5. **Routing**, the task of interconnecting the FPGA resources that have been previously placed.
6. **Timing analysis**, to validate the speed of the circuit, and check that the clock signal can properly propagate across the circuit while respecting the timing constraints.
7. **Bitstream generation**, which converts the results of the previous steps into the final binary configuration file used to program the FPGA.

This entire process is not fast. The hardware design description is generally much more complex than an equivalent that traditional CPU or GPU

code even when using HLS. Besides that, the following steps, from logic synthesis to bitstream generation, can take hours or days, especially when working with large FPGAs such as the ones found in accelerator cards.

Modern HLS toolchains do much more than simplifying the hardware design process. Xilinx Vitis and Intel oneAPI, for example, provide integration with a host application using OpenCL or similar APIs. The host application can reconfigure the hardware with the chosen bitstream, transfer data to and from the FPGA, and start the computation, using software interfaces similar to the ones of GPUs.

#### FPGAs for Sparse Linear Algebra

The optimization of SpMV through FPGAs has been a hot research topic for years [45, 57, 232]. The work of Grigoras et al. [68] focuses on compressing the sparse matrix, moving the bottleneck from memory accesses to the decompression of the input data while lowering the storage demand. Then, Grigoras also proposes optimizations for FPGA SpMV based on instance-directed tuning, i.e. reconfiguring the hardware based on the input matrix characteristics [69].

Fowers et al. [57] already proposed a sparse matrix encoding to alleviate the inefficiencies of the FPGA Double Data Rate (DDR) memory subsystem. Rafique et al. [168] and Burovskiy et al. [30] both worked on minimizing communication in iterative SpMV applications, although they only targeted banded matrices (Section 2.2.2). Similarly, Grigoras et al. [70] optimized SpMV for FEM, working on block diagonal sparse matrices. While these techniques are very interesting, they do not apply to our case. The predictable structure of banded matrices is much simpler to handle than the unstructured matrices and graphs that we consider in our work, as it can be reconducted to traditional dense linear algebra.

Umuroglu et al. [194–196] leveraged local cache hierarchies and preprocessing schemes to maximize the amount of time in which values are kept in a fast local cache hierarchy, to minimize the negative impact of random accesses in sparse computations. More recently, Sadi et al. [171, 172] proposed an SpMV FPGA implementation that achieves significant speedup leveraging HBM and a data-compression scheme to reduce off-chip traffic. Jain et al. propose a modular DSA for SpMV, assembling a multi-block HLS and HDL desing, through a custom network-on-chip interconnection overlay. They achieve 92 % of the peak bandwidth of a single DDR4 module, although it remains to be seen whether their design can leverage multi-channel HBM-enabled accelerator cards. Moreover, they test their design on matrices two orders of magnitude smaller than ours,

and provide performance results on a hardware design that can hold matrices with only 64 thousand rows. It is not clear if their design can scale to larger input data and what the performance would be.

Xilinx has recently released a new implementation of SpMV, tailored for HBM and double-precision floating-point arithmetic [215]. It is claimed that this design can leverage 16 HBM pseudo channels on the Alveo U280, although we could not find information about what effective memory bandwidth this design achieves. Although their performance numbers are promising, we have not been able to replicate these results, probably due to an FPGA platform incompatibility on our side.

It is difficult to say whether all these hardware designs are still competitive in terms of performance, given the continuous evolution of FPGA hardware. Since the introduction of HBM, DDR-based designs can no longer provide comparable peak memory bandwidth, and thus performance, given how memory-intensive SpMV is. Similarly, previous research has focused on single or even double-precision floating-point arithmetic. This choice also limits the performance on SpMV, as floating-point arithmetic requires more hardware resources – limiting the number of parallel SpMV Compute Units (CUs) – and introduces additional data-dependencies that often prevents pipelined designs with Initiation Interval (II) of 1.

From an energy utilization perspective, accessing data in sparse matrices is more expensive than working on dense matrices with an identical memory footprint. That's because loading an entry from a dense matrix requires a single memory access, while loading a non-zero entry from a sparse matrix requires multiple memory accesses (e.g. three accesses for COO). On the other hand, operations on sparse matrices require fewer arithmetic operations as zero entries are not explicitly stored in the matrix, and they are simply ignored. There is certainly a trade-off worth exploring in the case of sparse matrices with high density (as in DL workloads, where 10 % density is common). However, in the context of graph analytics and recommender systems, sparse matrices often have extremely low density ($< 10^{-6}$), and they are impossible to represent as dense matrices. In this work, we benefit from the intrinsic low energy utilization of FPGAs, but the choice of using sparse matrices is not motivated by optimizing energy consumption. Instead, it is the only option to work on data with such scale.

Overall, while FPGAs have many individual highly-optimized hardware designs for SpMV and other sparse linear algebra operations, they still do not have any full-blown hardware design library with popularity comparable to Intel MKL or Nvidia cuSPARSE. To the best of our knowledge, we are the first to study the impact of reduced-precision fixed-point arith-

metic in the context of SpMV hardware designs, showing how lower precision benefits the performance of graph analytics and recommender system workloads, with no detriment to accuracy.

### 2.4.4 FPGA Accelerator Cards

Since the success of GPUs as external co-processors for computationally-intensive workloads, FPGA vendors have been trying to provide similar devices, developing what is commonly known as FPGA accelerator cards. An FPGA accelerator card is an external hardware module that, just like a GPU, can be connected to a traditional server to offload some computations from the server's CPU to the FPGA. They have access to their device memory, usually in the form of Dynamic Random-Access Memory (DRAM) or HBM, and have multiple interconnections, such as Peripheral Component Interconnect Express (PCIe) and Ethernet. FPGA accelerator cards are quickly becoming a valid alternative to GPUs for the acceleration of demanding workloads, with the ability to deliver comparable performance for a fraction of the Thermal Design Power (TDP).

#### PCI Express Interconnection

The most common way to deploy FPGA accelerator cards in data-centers is by connecting them to a host machine (i.e. a general-purpose server) through a PCIe interconnection. This configuration is used, for example, by cloud providers such as Amazon AWS [8] and Nimbix [144].

PCIe is a serial expansion bus standard, found commonly found in motherboards of both home computers and data-center servers. It is composed of different lanes, each of which is used as a full-duplex byte stream. The bandwidth of a PCIe interconnection is determined by the speed of each lane (identified by the PCIe generation) times the number of lanes available. The Xilinx Alveo U200 accelerator card has a 16-lanes PCIe3.0 interconnection, while the Xilinx Alveo U280 accelerator card has an 8-lanes PCIe4.0 interconnection. In both cases, the total bandwidth is of $16\,\mathrm{GB/s}$. As most servers support from 40 to $>$100 PCIe lanes, vendors usually connect up to 4 FPGAs to a single host machine. In this work, we focus on host machines equipped with a single FPGA accelerator cards. Considerations about the scalability of the proposed hardware designs to multiple FPGAs are provided in each chapter, as in each of the proposed algorithms we observe different challenges related to multi-FPGA scaling that must be addressed (Sections 3.6, 4.6 and 5.6).

The limited bandwidth of PCIe is a major bottleneck in computations

with low arithmetic complexity such as sparse linear algebra, graph analytics, and database workloads. In all these cases, it is common to find algorithms whose complexity scales linearly with the size of the data to be processed (e.g. SpMV, PR, database filter queries, etc.). The execution of such algorithms greatly benefits from the large memory bandwidth of hardware accelerators – FPGAs, but also GPUs. However, the end-to-end processing time, including the time required to transfer the input and the output from and to the hardware accelerator, might be significantly larger than the processing time achieved by executing the algorithm on a general-purpose CPU with significantly lower memory bandwidth.

To better contextualize this phenomenon, let's consider an SpMV on a sparse matrix that requires $10\,\mathrm{GB}$ of storage. This matrix roughly contains 800 million non-zero entries, assuming a COO storage, i.e. three arrays `x`, `y`, `val` with size equal to the number of non-zero entries of the matrix. With an 8-lanes PCIe 4.0 interconnection, it takes at least $625\,\mathrm{ms}$ to transfer the matrix to the accelerator card ($\frac{10\,\mathrm{GB}}{16\,\mathrm{GB/s}} = 625\,\mathrm{ms}$). The transfer time of the output depends on the number of rows of the matrix, which is usually much lower than the number of non-zero entries. If we assume an average degree of 10, consistent with the matrices used thorough this work (Tables 3.1, 4.2 and 5.3), it takes $\frac{10\,\mathrm{GB}/10}{16\,\mathrm{GB/s}} = 62.5\,\mathrm{ms}$ to transfer the output of the SpMV from the hardware accelerator to the host machine. This is also the transfer time of the dense input vector, for a total transfer time of $625\,\mathrm{ms} + 2 \cdot 62.5\,\mathrm{ms} = 750\,\mathrm{ms}$.

The most intuitive FPGA hardware design, in which each of the COO arrays is stored on a different DRAM bank that is accessed at peak bandwidth (similarly to the design proposed in Chapter 3) will process the matrix at around $50\,\mathrm{GB/s}$, resulting in $200\,\mathrm{ms}$ for the SpMV computation, i.e. $3.75\times$ less than the transfer time. In other words, the overall SpMV execution takes $950\,\mathrm{ms}$, of which only $21\%$ is actually spent computing the SpMV on the FPGA accelerator card. The limited bandwidth of PCIe is even more problematic as one develops more optimized hardware designs, for example, by partitioning the matrix across multiple HBM channels (Chapter 4).

Clearly, the bandwidth problem of PCIe is fully mitigated when employing workloads with large data reuse, like the ones in this work. Algorithms with large data reuse are, for example, iterative algorithms such as PR (and PPR, Chapter 3) and the Lanczos algorithm, which require the computation of multiple (usually between 5 and 30) sparse matrix multiplications. Alternatively, one can assume that a large number of queries (almost infinite, for all intent and purposes) are performed on the same data, and the input matrix is *permanently* resident on the device. In this context, *permanently*

would mean that the transfer time of the matrix is orders of magnitude lower than the time that such matrix stays on the device memory. This is the case of PPR, where we search for the most relevant vertices for a given input vertex: different users will provide a different input vertex, but the graph that is being analyzed does not change between queries. A similar situation occurs in Chapter 5, where we compute the similarity between an input embedding and a large collection of sparse embeddings (representing, for example, movies in the catalog of a streaming platform). In both cases, the sparse matrix will be updated sparingly (possibly once per day), making its transfer time a non-concern. Similar conclusions have been drawn in the context of databases [55, 169, 179], highlighting how hardware accelerators such as FPGAs and GPUs give the most benefits when processing data that are permanently resident on the accelerator device. The device is used as the main execution unit instead of being treated as a co-processor.

#### Memory Subsystem of FPGA Accelerator Cards

FPGA accelerator cards, such as the Xilinx Alveo series, integrate multiple types of memories. Beside traditional on-chip memories typical of FPGAs such as BRAM or UltraRAM (URAM) (in recent Xilinx FPGAs), FPGA accelerator cards offer a variety of additional memory interfaces. PLRAM is a low-latency on-chip memory that the host machine can also access, and is used as a small cache ($\approx 512\,\mathrm{kB}$) for host-device or inter-SLR data-transfer. Then, the most abundant type of memory comes in the form of DDR4 and HBM2. Alveo U200 cards have four DDR4 banks, for a total of $64\,\mathrm{GB}$ of DDR and $77\,\mathrm{GB/s}$ of off-chip memory bandwidth. Alveo U280 cards have only two DDR4 banks ($32\,\mathrm{GB}$, $38\,\mathrm{GB/s}$ of bandwidth), but provide also $8\,\mathrm{GB}$ of HBM2, with $460\,\mathrm{GB/s}$ of bandwidth. While the peak memory bandwidth is still lower than high-end GPUs, FPGAs provide fine-grained control over the hardware fabric, leading to greater operational intensity and possibly to higher performance, as shown in Chapter 5. On the other hand, extracting peak memory bandwidth from the Alveo U280's HBM subsystem is not trivial due to the complex nature of this subsystem.

The HBM subsystem is composed of two HBM stacks and of an HBM controller, created using the FPGA fabric [205, 213]. Each stack is divided into eight channels, and each channel is divided into two 64-bits pseudo channels, for a total of 32 pseudo channels. Each pseudo channel provides about $460/32 = 14.37\,\mathrm{GB/s}$ of bandwidth. Users have access to 32 Advanced eXtensible Interface (AXI) channels that can be mapped to pseudo channels. An AXI channel cannot be mapped to multiple pseudo channels, while multiple AXI channels can be mapped to the same pseudo channels.

In this case, AXI channels will share the bandwidth of the pseudo channel. The interconnection between AXI channels and HBM is implemented through a switch, although not all $32^2$ connections are directly implemented in hardware. Instead, the memory controller offers eight smaller switches, each connected to four AXI channels. In the following, unless otherwise specified, we do not strictly distinguish between AXI channels, pseudo channels, and HBM channels, and simply assume that each AXI channel used by our FPGA hardware designs is mapped to a different pseudo channel (and vice-versa), with no bandwidth sharing of any kind.

Overall, the necessary conditions to achieve the peak $460\,\mathrm{GB/s}$ of HBM2 bandwidth on the Alveo U280 are the following. One has to read 256 bits per clock cycle using one or more FPGA cores clocked at 450 MHz (alternatively, 512 bits per clock cycle at 225 MHz), using all of the 32 pseudo channels. The FPGA core must be pipelined with an II of 1, and perform burst transfers of maximum length. An example of such hardware design is presented in Chapter 5.

#### Super Logic Regions

Accelerator cards provide FPGAs so large that treating them as a unique piece of reconfigurable logic would not be feasible because of the extremely long synthesis process and of the timing constraints introduced by the physical distance of logic regions. Instead, these FPGAs are divided in Super Logic Regions (SLRs) [209]. The Alveo U200 and Alveo U280 accelerator cards both have 3 SLRs. Each SLR is a separate FPGA die slice, and can be treated as a separate FPGA from the point of view of reconfigurability. Individual FPGA cores cannot be placed across different SLRs. However, SLRs can communicate using PLRAM and off-chip memory, and are connected with a silicon interposer. Each SLR has direct access to different resources: for example, on the Alveo U280, SLR0 has direct access to HBM. Other SLRs can also access HBM, but additional routing logic is necessary. Figure 2.6 shows a schematic representation of the Xilinx Alveo U280 FPGA accelerator card, with its subdivision into three SLRs, and its complex HBM memory controller [44, 205, 211, 212].

### 2.4.5 Domain Specific Architectures (DSAs) and Custom Hardware Extensions

The ambitious goal of making sparse linear algebra faster and more efficient has pushed researchers to develop custom architectures specifically created to handle the idiosyncrasies of sparse workloads. Some researchers

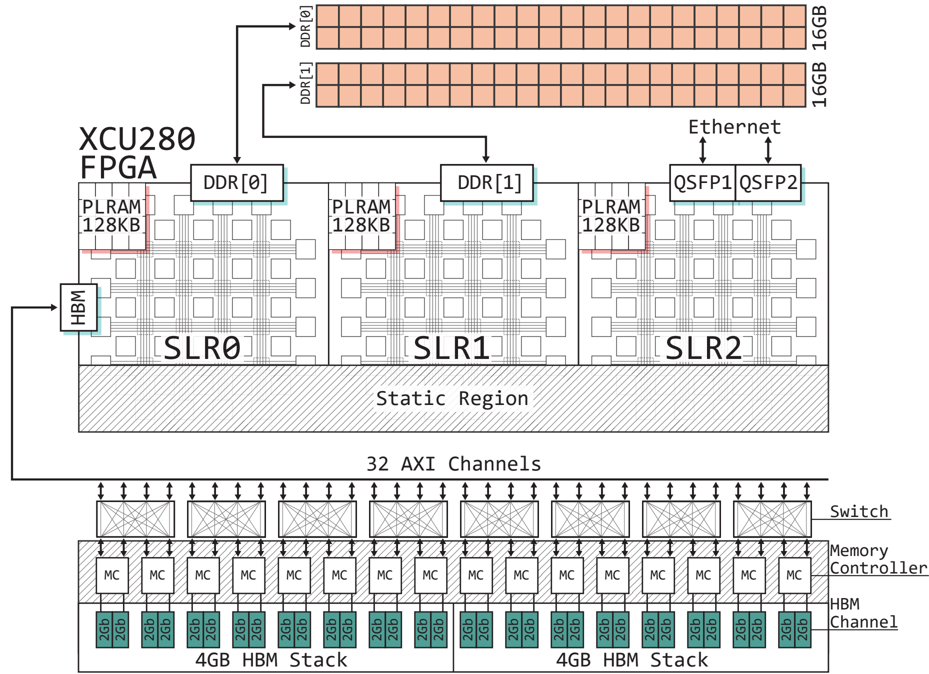



**Figure 2.6:** *Schematic representation of the Xilinx Alveo U280 FPGA accelerator card. The XCU280 FPGA is divided into three SLRs, with a common static region used to manage the PCIe interface and data transfers. We also show the complex structure of the HBM memory controller, with multiple switches and memory channels.*

created completely new DSAs, while other proposed extensions for current hardware such as CPUs.

Sadi et al. propose an SpMV architecture that leverages HBM and a data-compression scheme to reduce off-chip traffic, and can scale to matrices with billions of rows [172]. Xie et al. resort to Processing in Memory (PIM) for SpMV acceleration, showing how this novel computational paradigm can provide much greater performance than GPUs with significantly lower energy consumption. Zhang et al. extend the analysis to SpGEMM, proposing a new architecture focused on reducing DRAM accesses through condensed matrix representations, Huffman tree schedulers, and row prefetchers [226]. Instead of creating a completely novel architecture, Kanellopoulos et al. introduce a low-profile hardware-software extension for current CPUs [99]. They provide matrices encoded through a hierarchy of bitmaps and let custom hardware units interpret them efficiently. With their technique, they accelerate both SpMV and SpGEMM, improving the CPU performance by 40 %. Pavón et al. extend the x86–64 Instruc-

tion Set Architecture (ISA) with new vector instructions and a scratchpad memory that is optimized for sparse computations, achieving 4× better performance than their CPU baseline [159].

Besides hardware architectures for generic sparse computational kernels, we observe a growing interest in the hardware acceleration of sparse operations within recommender systems. Large players such as Facebook are proposing system-level techniques for fast retrieval and manipulation of large one-hot encoded embeddings [75, 76, 140]. Recommender systems at Facebook account for 79 % of AI inference cycles, with sparse operations accounting for almost 20 % of the total [76]. Kwon et al. proposed a new PIM architecture for operations on embeddings and tensors, showing how sparse features and embedding lookups are limited from memory capacity and bandwidth [109]. We also observe great interest in sparse DSAs for DL [41], with novel hardware-software codesign techniques [203, 220], and interest in sparsification for both training [166, 220] and inference [93].

Most of these custom architectures have been only simulated [166, 185, 203, 220]. A handful of them have prototypes implemented using FPGAs [13, 171, 172] or GPUs [109], although we still classify them as DSAs as they have not been created with the direct goal of exploiting features found in FPGAs and GPUs. Finally, a minority of DSAs already have hardware implementations, and might eventually be found in real deployments [13, 93, 171, 172]

## 2.5 Final Remarks

We are certainly observing a growing interest in sparse linear algebra. While its original applications were mostly in image and signal processing, sparse operations are now at the core of graph analytics, recommender systems, and deep learning. Sparse linear algebra is the only way to perform numerical computations on enormous datasets that would be impossible to represent in a dense format. From this chapter, we can draw a few conclusions that further motivate the rest of this work.

### 2.5.1 The Importance of Memory Controllers

Memory controllers are critical to achieving high performance in sparse computations. These workloads are **memory intensive** (they have low operational intensity, i.e. few mathematical operations are carried out for each value loaded from memory) and present **unpredictable memory accesses**. Memory controllers found in current FPGA accelerator cards allow users to choose the memory transaction's size that works better for their hardware

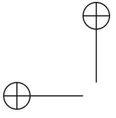
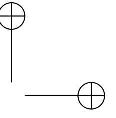
design. Exploiting this flexibility, FPGAs can reach greater operational intensity than competing architectures. They can access individual bits of each data packet loaded from memory and leverage compressed data representations that are out of reach for GPUs and CPUs. On the other hand, GPUs memory controllers are much easier to use from the user perspective, as they provide a unified memory address space instead of forcing users to deal with multiple pseudo channels. It is reasonable to assume that memory controllers currently found in FPGAs have been created with dense computations in mind and, if sparse workloads continue to grow in popularity, future generations of memory controllers will allow **greater flexibility than currently possible**. For example, the Alveo U280 HBM memory controller provides 32 AXI channels and 32 HBM pseudo channels. Reaching peak memory bandwidth requires that each AXI channel uses the full pseudo channel bandwidth, which is very difficult to achieve when dealing with random or data-dependent memory accesses. One can currently connect more than one AXI channel to a pseudo channel, but that implies leaving other pseudo channels unused, as the number of pseudo channels is the same as the AXI channels. Allowing more than 32 AXI channels is expensive from a resource standpoint. However, sparse hardware designs are usually inexpensive because of their low operational intensity (they have few mathematical operations, to begin with). They could easily bear such cost if it provided higher performance and flexibility. In this work, we show how to leverage the memory controllers of current FPGA accelerator cards at their best, but it is also important to remember the performance improvements that lie just a few interconnections away.

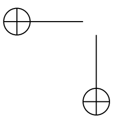
### 2.5.2 No architecture to rule them all

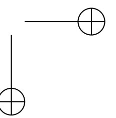
High-performance sparse linear algebra comes in many different hardware flavors, from traditional CPUs to exotic DSAs. Some architectures are better for the enormous scalability required in supercomputing (CPUs such as the ones in the Fugaku supercomputer), while other architectures provide the raw performance required by DL workloads (GPUs). FPGAs deliver a combination of performance and power efficiency. Most importantly, FPGAs are renowned for their predictable latency, making them the ideal choice for real-time applications in data-centers, especially in the context of recommender systems with billions of daily interactions and strict latency Service-Level Agreements (SLAs). FPGAs also provide access to optimized reduced-precision arithmetic, opening the doors to fine-grained trade-offs between performance and numerical accuracy in workloads such

as recommender systems, which are highly tolerant to approximate results.

We also observe a growing interest in DSAs, with novel technologies like in-memory processing and spatial computing. HBM, while still not widespread, is already employed in commercial GPUs and FPGAs accelerator cards, and also a critical key to success for the results presented in this work. Some of the largest players in the technology world have shown commitment in researching hardware acceleration for sparse computations [93, 166, 185, 203, 220, 222], and some of these accelerators, such as Sparse Tensor Cores, are already available in commercial hardware. However, we are still very far from taming the complexity of sparse computations, especially for **unstructured sparsity** in tensors found in DL. Even the largest neural network models still resort to structured sparsity [56], and fully exploiting unstructured representations is likely to require a **major shift from the current computation paradigms** towards new approaches such as PIM and spatial computing.





# CHAPTER 3

# A Reduced-Precision Streaming SpMV Hardware Design for Personalized PageRank on FPGA

*Sparse matrix-vector multiplication is often employed in many data-analytic workloads in which low latency and high throughput are more valuable than exact numerical convergence. FPGAs provide quick execution times while offering precise control over the accuracy of the results thanks to reduced-precision fixed-point arithmetic.*

*In this chapter, we introduce the novel streaming FPGA hardware design for Coordinate Format (COO) sparse matrix-vector multiplication used as corner stone for this thesis. We study its effectiveness when applied to the Personalized PageRank algorithm, a common building block of recommender systems in e-commerce websites and social networks. Our implementation achieves speedups up to 6x over a reference floating-point FPGA design and a state-of-the-art multi-threaded CPU implementation on eight different datasets while preserving the numerical fidelity of the results and reaching up to 42x higher energy efficiency compared to the CPU implementation. Moreover, we show that our fixed-point FPGA hardware design reaches convergence 2x faster than floating-point implementations.*

## 3.1 Introduction

Sparse Matrix-Vector Multiplication (SpMV) is a computational primitive widely employed in machine learning, engineering, and, most importantly, graph analytics [101, 224] as real-world graphs present an extremely high degree of sparsity. Personalized PageRank (PPR) [18], a variation of the famous PageRank (PR) algorithm ranks the most relevant vertices of the graph with respect to an input vertex, and is very often employed as part of complex graph analytics pipelines. In most cases, PPR must be computed with minimal latency, often on graphs with millions of edges, such as domain-specific knowledge bases, e-commerce websites, and social networks communities [118, 120, 157, 162], to find recommended posts in a social network while users interact with it, or recommended items for a given query on an e-commerce platform. Moreover, the precise numerical values produced by the algorithm are rarely useful, as long as the order of the top-ranked vertices is correct (consider the problem of recommending the Top-10 products for a user query). Numerical boundedness of PPR makes Field-Programmable Gate Arrays (FPGAs) suitable for computing PPR with throughput beyond the one of traditional architectures, leveraging fixed-point arithmetic to reduce execution time while preserving the correct ranking, and accelerate convergence.

In this chapter, we propose a novel FPGA architecture for a streaming edge-centric SpMV that uses Coordinate (COO) format matrices, and represents the cornerstone of this thesis. Here, we explore the application of our design PPR, given the ubiquity of this algorithm. In the following chapters, we extend our SpMV design to use High Bandwidth Memory (HBM) instead of DDR4, support partitioned matrices, and compute Top-K Sparse matrix-vector multiplication (Top-K SpMV), showing how our design can be adapted to a larger set of algorithms encountered in graph analytics and recommender systems. We leverage the COO format to enable a fully streaming computation, limiting the effect of exponential degree distributions [46] of real-world graphs, where degree distribution (the numbers of neighbors for each vertex) is highly skewed. Through reduced-precision fixed-point arithmetic, we maximize performance while reducing resource utilization and preserving the quality of the results.

In summary, this chapter presents the following contributions:

- We introduce an **optimized FPGA architecture of SpMV** that leverages a COO matrix and reduced-precision arithmetic, which we employ in a novel implementation of PPR (Section 3.4).

- We validate the practical applicability of our PPR implementation on 8 different graphs against a state-of-the-art multi-threaded CPU implementation and an equivalent 32-bits floating-point FPGA architecture, reaching speedups up to 6.8× and up to 42× higher energy efficiency.
- We characterize how reduced precision leads to **negligible accuracy loss** and 2× **faster convergence** on PPR, showing the effectiveness of reduced precision for graph ranking algorithms (Section 3.5).

## 3.2 Related Work

Optimizing PR and PPR is a longstanding problem in graph analytics. Existing implementations focus on the numerical properties of the algorithm or on finding new ways to exploit the full specific hardware architectures. In this section, we provide an overview of existing research on the optimization of PR and PPR for different hardware architectures, especially in the context of hardware-accelerated implementations based on sparse linear algebra and SpMV.

### 3.2.1 Numerical Optimizations

PPR is deeply connected to obtaining the main eigenvector of a matrix representing the input graph [26]. Numerous attempts have been made to accelerate this computation, by optimizing the Power Method iteration [98, 135], approximating a variation of PR that considers the rank of a subset of vertices [18, 20], or computing lower bounds for the PR scores [24]. However, the intricate control flow of these implementations is not always suitable for FPGA acceleration.

### 3.2.2 CPU and GPU Implementations

Leveraging sparse linear algebra for graph processing is the focus of the GraphBLAS project [101], which offers early implementations for both CPU and Graphics Processing Unit (GPU) [27, 219]. Highly tuned implementations of PPR exploit the graph data-layout to maximize cache usage [230], or employ multi-machine setups to process trillions of edges [86, 231]. Recently, variations of PPR have found application in the computation of embeddings, for downstream Machine Learning (ML) applications [221]. Green-Marl [84] and GraphIt [225] implements PPR using Domain-Specific Languages (DSLs) that abstract the intricacies of graph processing, and optimized to fully exploits the CPU hardware. PPR on

GPUs is less common: it is worth mentioning nvGRAPH [146] and GraphBLAST [219], that leverage sparse linear algebra to match and possibly outperform CPU implementations, while the implementation by Guo et al. [72] is optimized for dynamic graphs. Gunrock [204], instead, is a GPU graph analytics framework that implements PR with performance comparable to the ones by Nvidia. Finally, the work of Shi et al. optimizes the computation of Top-K PPR on GPUs for real-time applications [182].

### 3.2.3 FPGA Implementations

To the best of our knowledge, no existing work specifically addresses the computation of PPR on FPGA, either using reduced-precision arithmetic or algorithmic optimizations.

There have been multiple attempts at accelerating PR. Zhou et al. [229] propose an optimized data layout to guarantee sequential Dynamic Random-Access Memory (DRAM) accesses, and burst memory accesses to limit the negative impact of random memory accesses in graph computations. Their work shows promising results but was evaluated at simulator-level and does not compare against state-of-the-art CPU implementations. McGettrick at al. [132] optimize the use of SpMV within PR: however, using double-precision floating-point arithmetic results in unnecessary computations and leaves room for significant speedups.

However, optimizing SpMV computations on FPGAs has seen significant contributions that are worth mentioning as SpMV represents the main bottleneck of many PR implementations. The work of Grigoras et al. [68] focuses on compressing the sparse matrix, moving the bottleneck from memory accesses to the decompression of the input data while lowering the storage demand. Umuroglu et al. [195] leverage local cache hierarchies and pre-processing schemes to maximize the amount of time in which values are kept in a fast local cache. Using dataset partitioning and complex memory hierarchies enable SpMV computations on web-scale graphs, as seen in Shan et al. [178]: clearly, there is a performance trade-off introduced by supporting larger graphs, and a simpler design might be more beneficial for smaller datasets such as the ones in our intended use-case. Reduced-precision arithmetic has not been thoroughly studied in the context of graph ranking algorithms, but encouraging results were shown in numerical analysis and deep-learning [121, 201].

## 3.3 Problem Definition

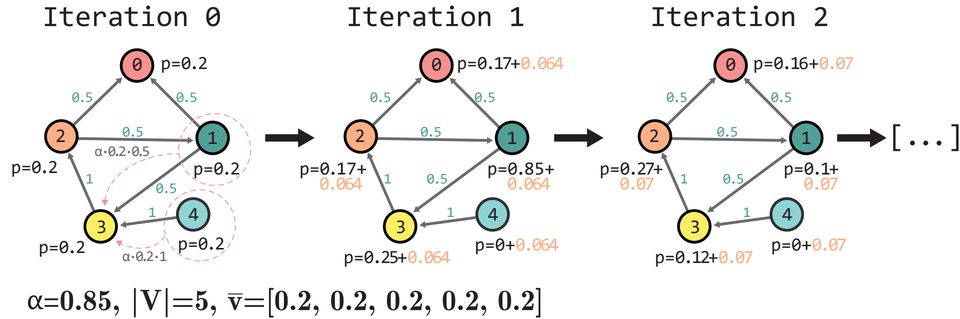
In this chapter, we apply a novel SpMV architecture to the computation of Personalized PageRank. This algorithm provides a *personalized* ranking of the graph vertices, such that vertices that are more *relevant* to an input vertex will have a higher score. PPR is a variation of the famous PR algorithm, which was originally developed to rank websites based on their importance by structuring the web as a graph where websites are vertices, and links between websites are directed edges [152]. On the other hand, the goal of the PPR algorithm is to provide a *personalized* ranking of the vertices of the graph, such that vertices that are more *relevant* to an input vertex will have a higher score. From a practical perspective, PPR can provide the users in a social network that are the most similar to a given user or retrieve the products in an e-commerce platform that might be of interest to a user who bought a certain item. We show the first few iterations of a PR and PPR computation in Figure 3.1. The PR score of each vertex is the sum of the PR scores of vertices that link to it, and the contribution of vertices with many outgoing links will be weighted less.

Given a graph $G$ with $|V|$ vertices and $|E|$ edges, we represent it using the adjacency matrix $\mathrm{A}$ and out-degree matrix $\mathrm{D}$ (a diagonal matrix with the number of out-going edges of each vertex). Define $\mathrm{X} = (\mathrm{D}^{-1}\mathrm{A})^T$ as the probability of transitioning from a vertex to one of its neighbors[1], a *personalization vertex* $v$, and a vector $\mathrm{p_t}$ of PageRank values, personalized w.r.t. $v$, computed at iteration $t$. $1 - \alpha$ is the probability of moving to any random vertex, and $\mathrm{d}$ is a *dangling* vector s.t. $\bar{d_i} = 1 \Leftrightarrow D_{ii} = 0$, $\bar{d_i} = 0 \Leftrightarrow D_{ii} \neq 0$. $\mathrm{d}$ is added to $\mathrm{D}$ to ensure that the computation is numerically stable [89], and $\mathrm{p_t}$ represents a valid probability distribution. The values in $\mathrm{p_t}$ are the probability of being on a given vertex after an infinite number of transitions starting from the personalization vertex $v$. The vector $\mathrm{v}$ is equal to $0$ except for the element at index $v$, which is $1$. The recurrence equation (as in [23, Section 3]) of PPR is

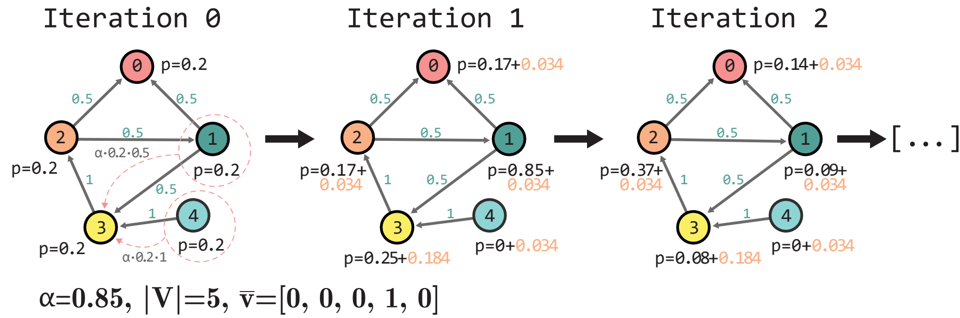
$$\mathrm{p_{t+1}} = \alpha \mathrm{X} \mathrm{p_t} + \frac{\alpha}{|V|}(\mathrm{d}\mathrm{p_t})1 + (1-\alpha)\mathrm{v} \quad (3.1)$$

The first term of the right-hand side is a matrix-vector multiplication, while the second and third terms (the *dangling factor* and the *personalization factor*) are obtained with dot-products. From an algorithmic perspective, PPR is similar to PR, except for the initialization of $\mathrm{v}$. In PR, all the values in $\mathrm{v}$ are identical and equal to $1/|V|$, while in PPR the values are de-

[1] With uniform probability, the probability of moving from vertex $x$ with out-degree $d$ to neighbor $y$ is $1/d$




**(a)** ***PageRank****: initially, all PageRank values are equal to $1/|V|$. In the first iteration, we show how values for vertex 3 are aggregated to update its PageRank value. The dangling term's contributions $(\alpha/|V|)(\mathrm{dp_t})1 + (1-\alpha)\mathrm{v}$ are highlighted in orange. In the basic formulation of PageRank, all dangling term's contributions are identical, as values $v_i \in \mathrm{v}$ are identical and equal to $1/|V|$.*




**(b)** ***Personalized PageRank****: compared to Figure 3.1a, values in* v *are not equal: only $v_3$, which corresponds to the personalization vertex, is 1. As such, dangling term's contributions will also be different for the personalization vertex, and the algorithm will output a result different from the one in Figure 3.1a.*

**Figure 3.1:** *First three iteration of PageRank ⓐ and Personalized PageRank ⓑ for a small example graph, with $|V| = 5$ and $\alpha = 0.85$.*

termined by the personalization vertices. This subtle difference ensures that PPR converges to a *personalized* result, with two practical consequences:

1. Optimizing the storage of the personalization term is less straightforward, as it might be necessary to store a vector of size $|V|$ instead of a single scalar value. In practice, as seen in Figure 3.1b, all the personalization values except for $v_3$ (the personalization vertex) are identical. An optimized implementation of PPR will not store the entire personalization vector, but only $\kappa + 1$ values (with $\kappa$ being the number of personalization vertices).

2. While PR provide a global score for the entire graph, PPR gives a result that is query-dependent. Computing and storing the PPR values for all the vertices in the graph is infeasible (up to $V^2$ time and

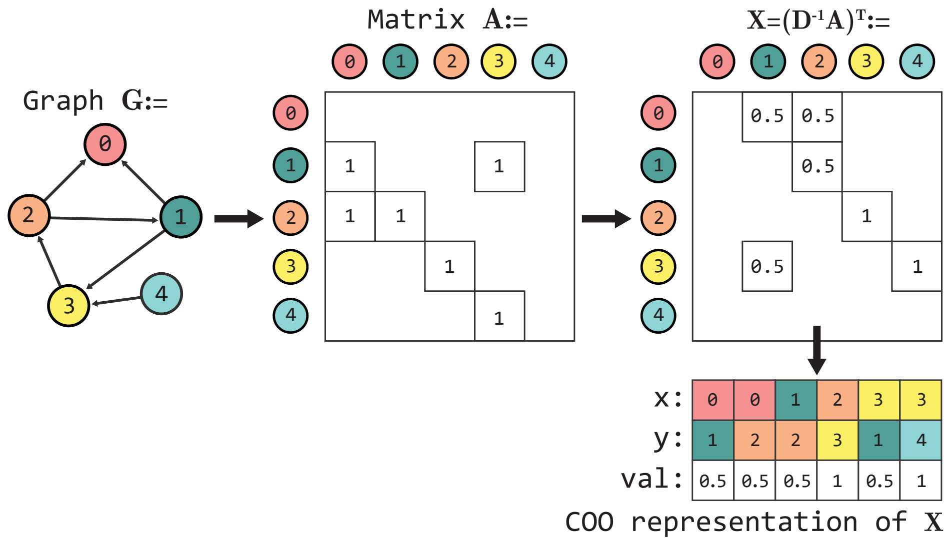



**Figure 3.2:** *COO representation of the transition matrix* $\mathrm{X}$ *for the sample graph in Figure 3.1, to perform a PageRank computation. In* $\mathrm{X}$*, each value* $val = (x, y)$ *can be seen as the probability of moving from* $y$ *to* $x$*. For example, from vertex* $y = 1$ *there is a* $0.5$ *probability of moving to vertex* $x = 0$ *and a* $0.5$ *probability of moving to vertex* $x = 3$.

storage). While PR values can be computed when the graph is updated (for example, overnight in recommender system applications), the PPR is often computed on-demand (i.e. when a user requests a recommendation), introducing strict real-time latency requirements.

The weighted adjacency matrix $\mathrm{X}$ is stored in a *sparse format* as it is extremely sparse: in a graph with $10^6$ vertices and average out-degree $10$, only $10 \cdot 10^6/10^{12}$ (i.e. $0.001\,\%$) of the entries of $\mathrm{X}$ are non-zero.

Compressed Sparse Column (CSC), a common storage format for sparse matrices [178], can be inefficient for real-world graphs with vertex degrees that follow an exponential distribution, as it limits pipelined architectures that demand precise knowledge of data boundaries. Instead, we employ the COO storage layout, which uses three equally sized arrays, containing, for each entry, its value and its two coordinates. Figure 3.2 represents a simple graph as COO: note the drastically lower memory footprint compared to the dense representation. COO simplifies array partitioning, enables burst reads from memory, and pipelined hardware designs, as entries are independent and the architecture is not bound to knowing the degree of each vertex. Instead, CSC-based designs often fail to handle graphs with expo-

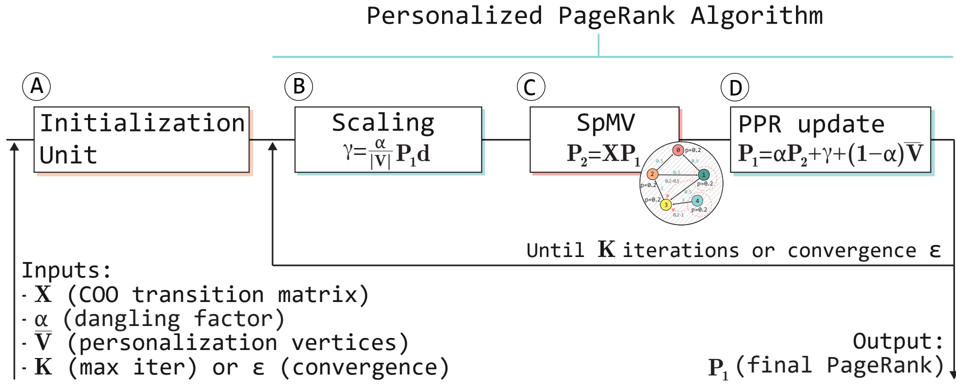



**Figure 3.3:** *Block diagram of the Personalized PageRank algorithm, as implemented in our hardware design. The SpMV unit aggregates the PageRank values of each vertex's in-neighbours, as in Figure 3.1b.*

nential distribution, especially if stream-like processing is demanded.

We compute $\kappa$ *personalization* vertices in parallel, to batch multiple user requests. We replace $\mathrm{p_t}$ with a matrix $\mathrm{P_t}$ of size $|V| \times \kappa$, and $\mathrm{v}$ with a matrix $\mathrm{V}$ where each column vector refers to one of the $\kappa$ *personalization* vertices. Updating $\mathrm{P_t}$ requires reading all the edges only once. This optimization boosts the efficiency of a memory-bound algorithm and enables higher throughput and scalability. The block diagram of PPR, as implemented in our hardware design, is shown in Figure 3.3: each element of Equation (3.1) is mapped to a different computational unit, as explained in Section 3.4.

## 3.4 The Proposed FPGA Hardware Design

We present the building blocks of our SpMV FPGA hardware design and how we integrated it in the PPR computation, our intended use-case.

### 3.4.1 Personalized PageRank Hardware Design

Algorithm 3 contains the pseudo-code of the main PPR computation. The input graph is read from Double Data Rate 4 (DDR4) DRAM, with edges as packets of size $P_SIZE = 256$ to maximize the throughput of memory transactions, and process $B$ edges per clock cycle (8, if $P_SIZE = 256$ bits and each value is 32 bits). The initialization stage (lines 2 to 4 and Figure 3.3 Ⓐ) resets local buffers and initializes the PPR values in $\mathrm{P_1}$ according to the input personalization vertices in $\mathrm{V}$. Line 7 correspond to the computation of the scaling factor (Figure 3.3 Ⓑ and Algorithm 4), a value

identical for each vertex required to track dangling vertices and guarantee that $\mathrm{P}_1$ is a valid probability distribution. Line 8 of Algorithm 3 is the core of PPR, i.e the SpMV computation identified in Figure 3.3 Ⓒ and further detailed in Algorithm 5 and Figure 3.4. The $\kappa$ entries of the scaling vector are the sum of current PPR values of vertices with no outgoing edges. Values in the *dangling* bitmap are read in blocks with size $P_SIZE$ (each value is stored in a single bit), while $\mathrm{P}$ is cyclically partitioned to access $B$ contiguous values in a single clock cycle. Line 9 of Algorithm 3 is the vector scaling and addition (similar to an `axpy` BLAS operation [114]) found in Figure 3.3 Ⓓ: it uses the same partitioning applied to Algorithm 4. This operation combines the partial results found in lines 7 and 8 and adds the contribution of the personalization vertices, representing a probability of $alpha$ of restarting the PPR random walk from the starting personalization vertex. PPR values are stored as reduced-precision fixed-point values. Quantization truncates to zero the fractional bits with precision higher than representable in the given fixed-point precision. Other policies (e.g. rounding to the closest representable value) resulted in numerical instability.

In our PPR hardware design, all the steps in Algorithm 3 use the same fixed-point numerical precision (e.g. 25 bits). Given that all numerical values are in the interval $[0, 1]$, there is no straightforward gain in using multiple scales or numerical precision. We briefly explored different solutions (such as computing the scaling factor in floating-point arithmetic), but we did not observe any clear-cut benefit in terms of accuracy. Most likely, SpMV is the main responsible for the total numerical accuracy of the algorithm. On the other hand, our hardware design in Chapter 4 does use mixed-precision arithmetic, using fixed-point for bound numerical values and floating-point arithmetic for unbound values.

### 3.4.2 Customizing SpMV for PPR

In this section, we present our SpMV FPGA hardware design and how we optimized it for the computation of PPR.

The SpMV design has 4 main units, highlighted in Algorithm 3 and Figure 3.4. First, we read a graph packet from DDR4 (line 4 in Algorithm 5, Figure 3.4 Ⓐ), and store it in local buffers $x,\ y,\ val$ to read and update $B$ values at once. While we compute $\kappa$ PPR vectors in parallel, the edges of the graph are accessed only once. Lines 7–9 in Algorithm 5 (Figure 3.4 Ⓑ) compute *point-wise* PPR contributions for each edge in the current packet. Parallel accesses to $\mathrm{P}_\mathrm{t}$ retrieve PPR values for each *personalization* vertex: thanks to UltraRAM (URAM), we perform these accesses with low latency,

**Algorithm 3** Personalized PageRank

```
1:  function PPR(coo_graph, V, d, α, max_iter)
2:      Initialize local buffers to 0
3:      for k ← 0, κ do                                   ▷ Set PR=1 on pers. vertices
4:          P_1[k] = V[k]
5:      end for
6:      for i ← 0, max_iter do
7:          scaling_vec ← scaling(P_1, d)                 ▷ i.e. α/|V| P_1 d
8:          SpMV(coo_graph, P_1, P_2)                     ▷ Xp_i in eq. (3.1)
9:          P_1 = αP_2 + scaling_vec + (1 − α)V
10:     end for
11:     return P_1
12: end function
```

without strong constraints on the graph size. The $B$ *aggregator units* (lines 10–15 in Algorithm 5, Figure 3.4 Ⓒ) combine *point-wise* contributions to obtain the total contribution of a single vertex, as a packet can contain multiple edges referring to it (for example, updating $p_6$ in Figure 3.4 requires the contributions of 2 edges). *Aggregators* consider edges whose end is in the range $[x[0], x[0] + B]$, i.e. the maximum range that can be found in a packet. Local buffers and accumulators employ registers to unroll loops and perform the computation in $O(1)$ time.

The last unit adds PPR contributions of the current packet to the PPR arrays stored in URAM (lines 16–29 in Algorithm 5, Figure 3.4 Ⓓ). Contributions are stored in a buffer of size $2B$, with up to $B$ non-zero contiguous values. A Finite-state machine (FSM) with two buffers of size $B$ accumulates PPR entries and writes them to output at indices multiple of $B$, ensuring that updates can be performed in parallel as they are aligned to the partitioning factor of $\mathrm{P}_{\mathrm{t+1}}$. Each block of $res_1$ is written on URAM only once to avoid expensive $+=$ operations and Read-After-Write (RAW) conflicts in unrolled loops. The four main steps of the algorithm (Algorithm 5, line 2) are separate modules in a streaming *data-flow* region, enabling aggressive pipelining of loops and better resource allocation.

### 3.4.3 PPR Buffers Design

A critical aspect of designing a high-performance SpMV hardware implementation is the optimization of random accesses to the dense vector multiplied by the sparse matrix. If we want to process $B$ sparse matrix elements per clock cycle, we also have to perform $B$ random accesses to the dense

**Algorithm 4** Computing the scaling vector of PPR

```
1:  function SCALING(P, d)
2:      ▷ Blocks have B elements and size P_SIZE
3:      for i ← 0, |V|/B do
4:          dangling = d[i/(P_SIZE/B)]
5:          for k ← 0, κ do
6:              for b ← 0, B do
7:                  dp_buf[k, b] = P[k, i · B + b] + dangling[(i · B + j) % P_SIZE]
8:              end for
9:              dangling_vals[k, :] += sum(dp_buf[k, :])
10:         end for
11:     end for
12:     return α · dangling_vals
13: end function
```

vectors, to locations that are unknown in advance (i.e. before the matrix packet of size $B$ has been read). A naïve implementation that accesses the dense vector through a single DDR4 channel cannot guarantee satisfactory performance, as the hardware design is bound by the 32-bit random access bandwidth of the DDR4 channel, equal to slightly more than $2\,\mathrm{GB/s}$ [205]. The SpMV hardware design in our PPR adaptation performs pipelined burst reads of 256 bits packets from three DDR4 channels (out of the four available). As such, we need to guarantee that random accesses to the dense vector can sustain the full bandwidth of the three DDR4 channels (approximately, $54\,\mathrm{GB/s}$). To design an SpMV core with such bandwidth, we store dense vectors either in URAM or HBM.

#### Storing PPR Buffers in URAM

The SpMV hardware design presented in this chapter stores temporary PPR values in URAM, a type of memory available in recent Xilinx UltraScale+™ FPGAs. URAM can be seen as a middle-ground between slow but abundant DRAM and faster, but limited, Block RAM (BRAM). Using a Xilinx Alveo U200 Accelerator Card, we store up to 90MB of data on URAM, corresponding to around 20 million different PR values, assuming that the PR value of each vertex is stored in 32 bits. In practice, reduced fixed-point precision allows us to store even more vertices and scale to larger graphs. The maximum number of edges is bound by the available DDR4 and could reach about 5 billion on the $64\,\mathrm{GB}$ of DDR4 available in the Alveo U200 card. Our design can be easily scaled to compute multiple PPR vectors in

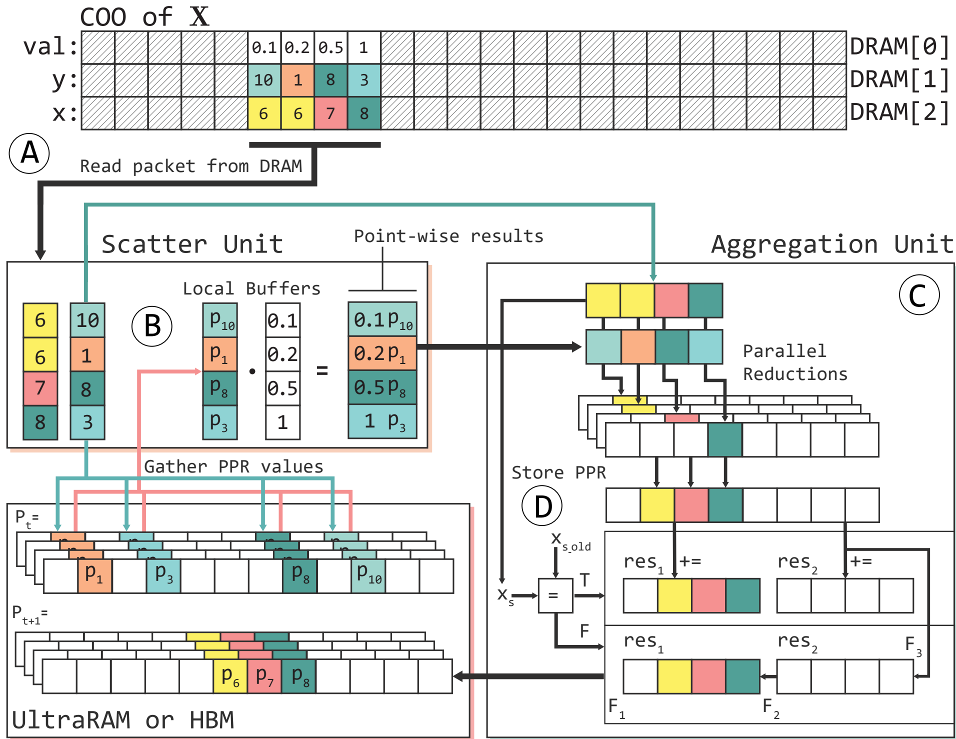



**Figure 3.4:** *Block-diagram of our PPR SpMV hardware design. The scatter and aggregation units show the computation for a single vertex, but they are replicated to support $\kappa$ vertices. Large arrows represent a streaming transfer between units*

parallel if the end-user can provide an upper bound over the number of vertices in its graphs. In our experiments, we achieve optimal performance if the number of vertices does not exceed 1 million (which is still larger than what is found in many real applications), and 8 to 16 personalization vertices are computed in parallel, using the same hardware resources required for a larger graph that does not consider multiple PPR vertices. On paper, our design can be adapted to FPGAs without URAM, using BRAM as a replacement. However, an Alveo U200 only has 10MB of BRAM, making this alternative design usable only on smaller graphs that require storing up to 2 million PR values.

The size of local memory buffers does not introduce strong practical limitations on the size of the graphs: our PPR implementation targets graphs encountered in social network communities and e-commerce platforms, whose size does not fill the available FPGA hardware resources [118] There also exist partitioning techniques [178, 195] that handle large web-scale

**Algorithm 5** COO SpMV for the computation of PPR

```
1:  function SpMV(coo_graph, P_t, P_{t+1})
2:      for i ← 0..|E|/B do
3:          ▷ 1. Process COO in packets of size B
4:          x ← coo_graph.x[i];   y ← coo_graph.y[i];   val ← coo_graph.val[i]
5:          for k ← 0..κ do                                  ▷ κ personalization vertices
6:              ▷ 2. Update edge-wise PPR values
7:              for j ← 0..B do
8:                  dp_buf[k, j] = val[j] · P_t[k, y[j]]
9:              end for
10:             ▷ 3. Aggregate partial PPR values
11:             for b1 ← 0..B do
12:                 for b2 ← 0..B do
13:                     agg_res[k, x[0]%B + b1] += dp_buf[k, b2] · ((x[0] + b1) == x[b2])
14:                 end for
15:             end for
16:             ▷ 4. Store PPR values on each vertex
17:             x_s ← ⌊x[0]/B⌋ · B
18:             if x_s == x_s_old then
19:                 for j ← 0..B do
20:                     res_1[k, j] += agg_res[k, j]
21:                     res_2[k, j] += agg_res[k, j + B]
22:                 end for
23:             else
24:                 for j ← 0..B do
25:                     res[k, j + x_s_old] = res_1[k, j]
26:                     res_1[k, j] = res_2[k, j] + agg_res[k, j]
27:                     res_2[k, j] = agg_res[k, j + B]
28:                 end for
29:             end if
30:             reset(agg_res);   x_s_old ← x_s
31:         end for
32:     end for
33: end function
```

graphs. Although we have not explored graph partitioning for PPR, we do partition the input graph in the hardware design of the Lanczos algorithm in Chapter 4, which can be seen as a generalized and more complex version

of PR. The SpMV hardware design in Chapter 4 partitions the graph across multiple HBM pseudo channels, achieving scalability to much larger graphs (Table 4.2). Partitioning the computation to handle web-scale graphs is interesting but not strictly required in our PPR use-case or to validate the performance of the SpMV implementation presented in this chapter. Being able to easily extend our SpMV core to support larger graphs, as presented in Chapter 4, is a further testament to the flexibility of our hardware design.

#### Storing PPR Buffers in HBM

New FPGA accelerator cards such as the Xilinx Alveo U280 provide access to HBM, a type of DRAM that achieves greater maximum bandwidth than DDR4 by exposing many parallel pseudo channels (32 on the Alveo U280) Section 2.4.4. While the work in this chapter focuses on the Alveo U200, which does not have HBM, we still evaluate the implications of storing the PPR dense vectors in HBM. Our SpMV hardware design can easily take advantage of FPGA accelerator cards with and without HBM: Chapters 4 and 5 present extensions of our SpMV core that fully exploit HBM to store the dense vectors, the input matrix, or both, achieving greater parallelism and memory access throughput.

Using HBM instead of URAM enables higher clock frequency (up to 450 MHz for the individual SpMV core), as large utilization of URAM introduces strong routing constraints that limit the maximum clock frequency attainable. However, performing concurrent random accesses to a dense vector stored in HBM requires data replication as shown in Section 4.4.2, possibly limiting the number of concurrent personalization vertices that can be processed in a single batch to a maximum of 7. We need four replicas for each of the four non-zero values processed in a clock cycle, and we also need to reserve four pseudo channels to store the output of PPR: as the Alveo U280 has 32 HBM pseudo channels, we can batch $(32 - 4)/4 = 7$ PPR requests. Further information about the Alveo U280 memory subsystem are given in Section 2.4.4 and Section 4.4.2. In summary, depending on the application constraints (maximum throughput or minimal latency), it is worth having the flexibility of switching between locating the dense vectors in URAM or HBM, as we do allow in our SpMV hardware design.

### 3.4.4 Host Integration

Our hardware design follows a host-accelerator model in which the *host* (a server) communicates with the *accelerator* (an FPGA) over Peripheral Component Interconnect Express (PCIe). The host loads the graph and pre-

processes it, i.e. it computes the $val$ vector (the numerical entries of the matrix X) and the *dangling bitmap* (in which each vertex is represented by a 1 if it has no ingoing edges, 0 otherwise). Pre-processing (e.g. loading the graph) is done once at the start and not for each computation of PPR, and does not impact the computation time of PPR: in real workloads, the input graph is updated much more infrequently than the rate of new PPR queries. Re-synthesizing the hardware design is required to change the fixed-point precision, $\kappa$, or the maximum number of vertices in URAM, but not for different graphs.

## 3.5 Experimental Evaluation

In this section, we analyze the performance of our PPR FPGA hardware design in terms of execution time and power efficiency, showing how it can outperform a state-of-the-art CPU implementation running on server-grade hardware with peak memory bandwidth comparable to our FPGA accelerator card. We then focus our attention on the impact of reduced-precision fixed-point arithmetic on accuracy and convergence speed, showing how fine-grained control over the PPR values bit-width can result in 2× faster convergence with no accuracy loss.

Our hardware design is implemented on a Xilinx Alveo U200 Accelerator Card with $64\,\mathrm{GB}$ of DRAM ($77\,\mathrm{GB/s}$ of total bandwidth) and equipped with a `xcu200-fsgd2104-2-e` FPGA offering 960 URAM blocks of 288Kb (with 72 bits port width) and 4320 BRAM blocks with 18Kb size each. This FPGA platform is mounted on a server with an Intel Core i7-4770 CPU @ 3.40GHz with 4 cores (8 threads) and $16\,\mathrm{GB}$ of DRAM. We compare our PPR implementation against the floating-point implementation in PGX 19.3.1[2], a powerful toolkit for in-memory graph analytics. Its state-of-the-art implementation of PPR [84] is fully multi-threaded. Experiments with PGX were conducted on a machine equipped with two Intel Xeon E5-2680 v2 @ 2.80GHz with 10 cores (20 threads) each, and $384\,\mathrm{GB}$ of DRAM. We analyze 5 versions of our design, with different arithmetic types: 26 bits unsigned fixed-point (`U1.25`), 24 bits (`U1.23`), 22 bits (`U1.21`), 20 bits (`U1.19`), and a 32-bit floating point version (`F32`). Lower bit-width negatively impacts the quality of results, while higher precision provides minimal gain (Section 3.5.3). The CPU baseline uses 32 bits floating-point arithmetic, and our CPU does not support arbitrary precision. Simulated fixed-precision arithmetic resulted in lower CPU performance and is not a meaningful comparison. Manually batch-

[2] docs.oracle.com/cd/E56133_01/latest/index.html

**Table 3.1:** *Summary of graph datasets used in the evaluation*

| **Graph Distribution** | \|V\| | \|E\| | **Sparsity** | **Size (GB)** |
|---|---|---|---|---|
| $G_{n,p}$ **(Erdős-Renyi)** | $10^5$ | 1002178 | $10^{-4}$ | 0.024 GB |
| | $2 \cdot 10^5$ | 1999249 | $4.9 \cdot 10^{-5}$ | 0.047 GB |
| **Watts–Strogatz small-world** | $10^5$ | 1000000 | $10^{-4}$ | 0.024 GB |
| | $2 \cdot 10^5$ | 2000000 | $5 \cdot 10^{-5}$ | 0.048 GB |
| **Holme and Kim power-law** | $10^5$ | 999845 | $0.99 \cdot 10^{-4}$ | 0.024 GB |
| | $2 \cdot 10^5$ | 1999825 | $4.9 \cdot 10^{-5}$ | 0.048 GB |
| **Amazon co-purchasing network** | 128000 | 443378 | $2.7 \cdot 10^{-5}$ | 0.010 GB |
| **Twitter social circles** | 81306 | 1572670 | $2.3 \cdot 10^{-4}$ | 0.037 GB |

**Table 3.2:** *Resource usage, power consumption for selected bit-widths of our PPR design.*

| **Bit-width** | **BRAM** | **DSP** | **FF** | **LUT** | **URAM** | **Clock (MHz)** | **Power Cons.** |
|---|---|---|---|---|---|---|---|
| 20 bits | 14% | 3% | 4% | 26% | 20% | 220 | 34 W |
| 26 bits | 14% | 3% | 4% | 38% | 20% | 200 | 35 W |
| 32 bits, **float** | 14% | 48% | 35% | 89% | 26% | 115 | 40 W |
| **Available** | 4320 | 6840 | 2364480 | 1182240 | 960 | | |

ing multiple requests in PGX through vector properties did not provide a speedup over the fast default implementation of PPR, which is already fully exploiting the CPU [225].

Our experimental setup contains 8 graphs: 6 are generated using different statistical distributions offered by the Python `networkx` library[3], while 2 are real-world graphs from the Stanford Large Network Dataset Collection [118]. Table 3.1 summarizes each graph used in the evaluation. Synthetic graphs are consistent in size, edge distribution, and sparsity to real-world graphs used in e-commerce and social network communities [118]; their COO representation has a size in line with recent work on sparse matrices on FPGAs [67]. Small-world graphs have large clusters of vertices (i.e. communities). Erdős-Renyi have uniformly distributed edges and no obvious communities, possibly increasing the number of vertices that contribute to the computation of PPR for a given vertex. Power-law

[3] `networkx.github.io/documentation/stable/`

graphs also have well-defined communities of vertices, similar to what is encountered in social networks. Most vertices have only a few edges, while a few vertices have many edges, following a power-law degree distribution. Synthetic graphs with identical sizes highlight how trends are similar across distributions (Section 3.5.1, Section 3.5.3), and we can extract insights on the convergence and precision of PPR as we change input graph and bit-width.

### 3.5.1 Execution time

We measure for each graph the time required to compute the PPR values for 100 random *personalization* vertices, to simulate a realistic batch workload performed by social networks and e-commerce platforms. Time spent transferring results from FPGA to CPU is included and is negligible compared to the total execution time. All tests are executed with an $\alpha$ of 0.85, for 10 iterations each (even a low amount of iterations is enough for convergence, see Section 3.5.3).

Figure 3.5 reports the speedups of different fixed-point sizes compared to the CPU baseline and an equivalent 32-bits floating-point FPGA hardware design. Reducing the bit-width of fixed-point values does not affect the number of clock cycles required to perform the computation but positively correlates with clock frequency and higher speedups. On graphs with around $10^6$ edges we obtain up to 6.47$\times$ speedup, thanks to the reduced bit-width and the ability to compute 8 PPR vectors at once. Results for synthetic graphs are averaged, as no difference was observed among distributions. We achieve similar results on real-world graphs, with up to 6.8$\times$ speedup on the highly sparse Amazon co-purchasing network. Processing 100 random requests on the FPGA takes from 280 ms for Amazon to 1000 ms for larger graphs, in line with the real-time requirement of our use-case. The floating-point FPGA hardware design is 6$\times$ slower than the fixed-point designs, with larger Digital Signal Processor (DSP) usage (48 % vs 3 %), and negligible accuracy gain compared to 26-bits fixed-point (Figure 3.6).

The clock frequency is between 200 and 220 MHz, but we can reach up to 350 MHz with a lower number of concurrent PPR vertices $\kappa$. The clock speed increases sublinearly with respect to $\kappa$ above 200 MHz, limiting the benefits of very low $\kappa$. On larger graphs, the speedups are less significant, as higher URAM utilization negatively impacts the clock frequency due to routing congestion. In our experiments, doubling the size of the PPR buffers lowers the clock speed by around 35-40 %. Resources utilization (summarized in Table 3.2 for $\kappa = 8$), is minimal for BRAM, DSPs and

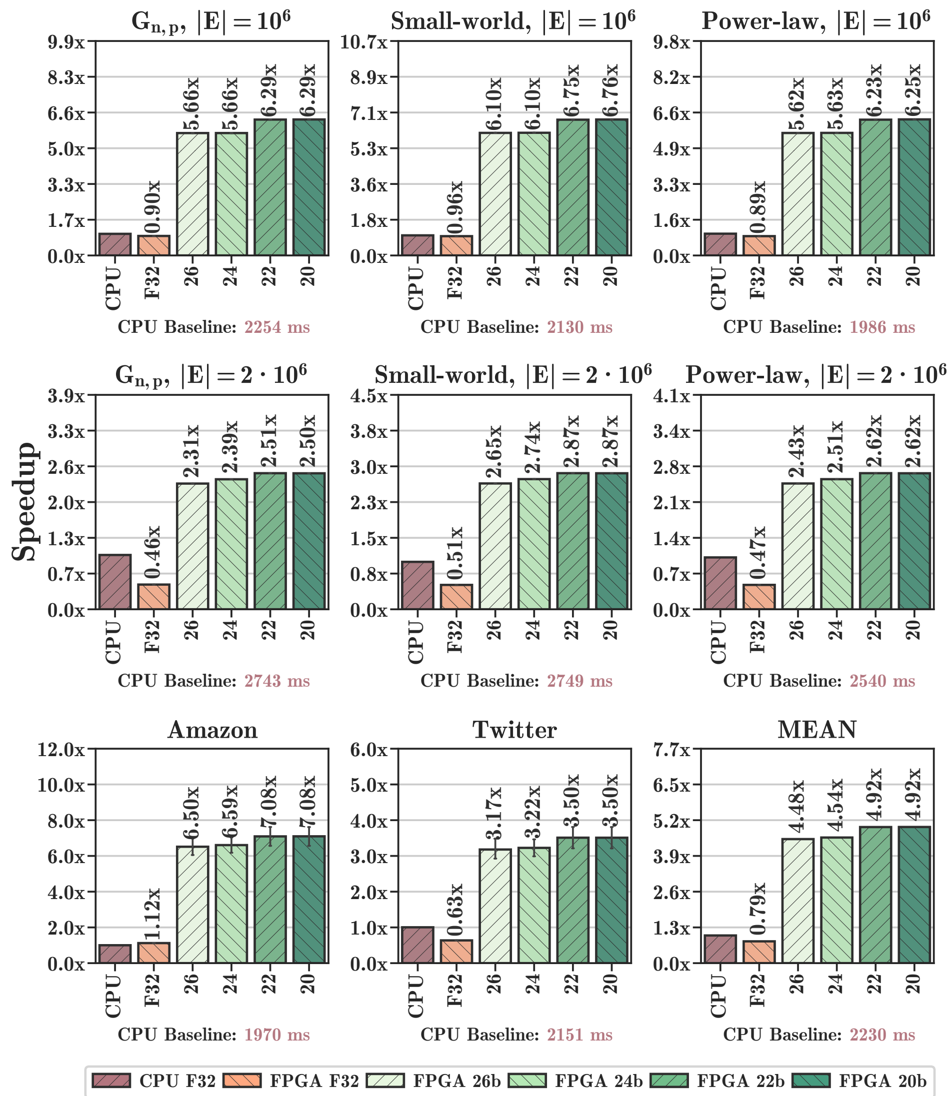



**Figure 3.5:** *Speedup of our Personalized PageRank FPGA reduced-precision hardware design (y-axis) with respect to the CPU baseline and the FPGA floating-point baseline, for decreasing bit-widths (x-axis).*

registers and is not impacted by fixed-point bit-width and PPR vector size. URAM usage grows linearly with PPR vector size (from 20 % to 40 % in our experiments). Look-Up Tables (LUTs) usage scales linearly with bit-width, from 26 % for 20 bits to 38 % for 26 bits.

### 3.5.2 Energy Efficiency

Our FPGA hardware design uses 35 Watts during execution, measured with an external power meter monitor, and increasing the PPR buffer or the fixed-point bit-width does not seem to affect the power consumption. The CPUs consume around 230 Watts, and our hardware design provides a Performance/Watt gain from $16.5\times$ to $42\times$ compared to it (geomean $28.2\times$). Even against a faster CPU or a GPU, our hardware design is likely to offer higher energy efficiency. Using fixed-point provides $5\times$ higher energy efficiency over the equivalent floating-point design, which, however, provides $2.5\times$–$5\times$ higher energy efficiency than the CPU baseline (geomean $4.3\times$).

These results allow us to formulate a few insights that also hold for the following chapters. All hardware designs in the thesis are inexpensive from a power consumption perspective. Not a single hardware design crosses the 50 Watts mark, far from the 230 Watts of our CPU or $> 250$ Watts of a GPU. The result is surprising as Alveo cards are rated for 225 Watts. However, we never came close to such power draw, even when using complex floating-point designs (Chapter 4) or reading from HBM at full bandwidth (Chapter 5). Then, we observe that fixed-point provides a slight reduction in power consumption, of around 5 Watts ($-15$ %). This reduction is smaller than we expected: possibly, the higher cost of floating-point arithmetic is compensated by the higher clock frequency of fixed-point designs.

Reducing precision does not noticeably affect power consumption. The memory bandwidth and the frequency of these accesses are not impacted by the fixed-point bit-width, as we always read one data packet of 256 bits from each memory channel in each clock cycle. Instead, we store more non-zero values in each packet, meaning that a reduction in fixed-point bit-width translates to a higher number of arithmetic operations done by the FPGA in each clock cycle. Individual arithmetic operations get cheaper due to the smaller number of bits in each value, but we do more operations per clock cycle, hiding the benefits of reduced-precision arithmetic from a power consumption perspective. This consideration might explain why a reduction of 33 % in terms of bit-width (32 bits to 20 bits) translates to a decrease in power consumption of only 1 Watt. Overall, a rough breakdown of power consumption is that the DDR4 (32 GB) takes about 12 Watts [134],

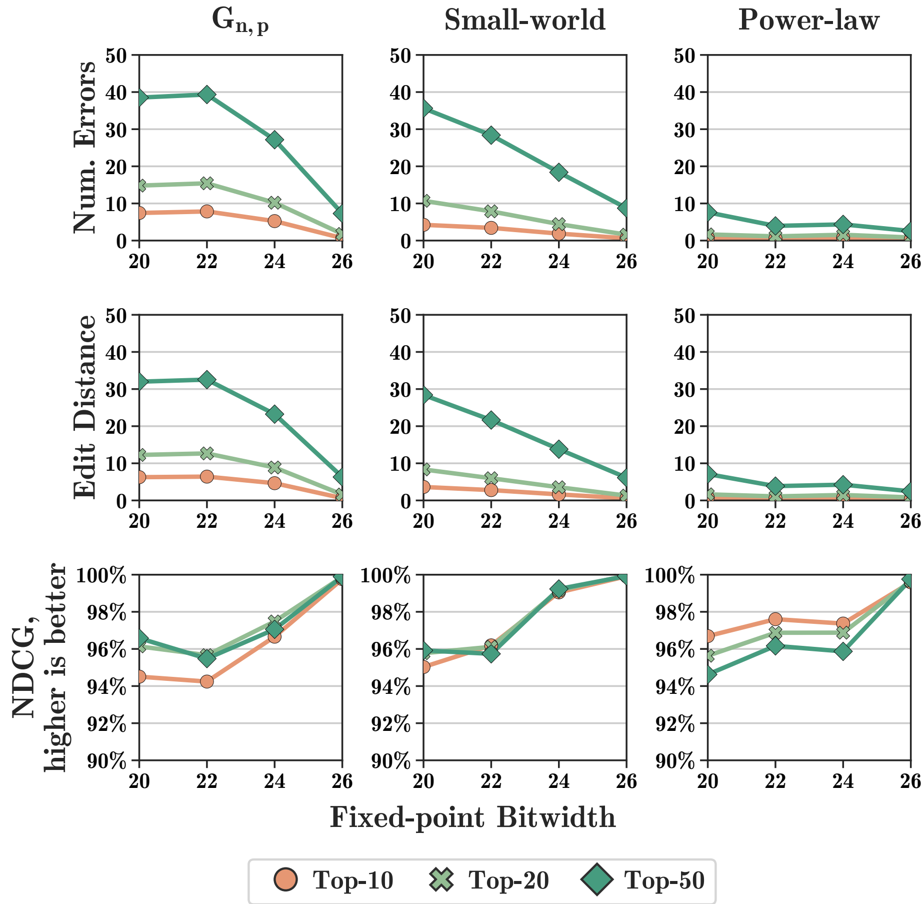



**Figure 3.6:** *Accuracy metrics measured on graphs with $2 \cdot 10^6$ edges, for increasing fixed-point bit-width. Number of errors and edit distance should be low, while NDCG must be close to 100 %. We evaluate these metrics for the top 10, 20 and 50 ranked vertices. In general, all metrics improve with higher numerical precision, although in some cases (e.g. Power-law graphs) even very low bit-widths guarantee high accuracy.*

and the remaining 20–30 Watts come from the FPGA itself.

### 3.5.3 Accuracy Analysis

We compared the accuracy of the rankings obtained with fixed-point precision (after 10 iterations) with the ones of the CPU implementation at convergence (with at least 100 iterations), using common Information Retrieval (IR) ranking metrics [173]. 100 iterations are enough to reach convergence even in web-scale graphs [113], although 10 iterations would often suffice (Figures 3.6 and 3.9).

#### Accuracy metrics

First, we look at the number of errors, i.e. the number of vertices with wrong ranking in the top 10, 20, and 50 compared to the CPU. This evaluation is consistent with the intended applications of PPR, i.e. finding the vertices more relevant to a given *personalization* vertex. This metric is very coarse-grained, as a single mistake can greatly affect the ranking: for example, if the correct Top-4 values are $\{2, 4, 8, 6\}$ and we retrieve $\{4, 8, 6, 2\}$, this metric reports 4 errors, although only a single value is displaced.

Edit Distance counts how many operations are needed to transform one sequence of Top-N vertices into another [119]; it handles ordering shifts: in the previous example, the edit distance is just 1, as we insert $2$ at the beginning and ignore values after the first N.

Normalized Discounted Cumulative Gain (NDCG) [94] is a common metric used in IR to evaluate ranking *quality* of search engines and recommender systems. This metric dampens the *relevance* of a given vertex by a logarithmic factor, dependent on its position. Highly ranked vertices contribute more heavily to the *cumulative gain* compared to the lower-ranked ones. Given a vector of PPR scores, let $rel_i$ be the *relevance* of the $i$-th vertex (in our case, $rel_i = |V| - i$), then we define Discounted Cumulative Gain (DCG) as in Equation (3.2). DCG is normalized by dividing it by the *Ideal Discounted Cumulative Gain*, i.e. the DCG of the reference CPU implementation, as in Equation (3.3).

$$\text{DCG} = \sum_{i=1}^{|V|} \frac{rel_i}{log_2(i+1)} \qquad \text{nDCG} = \frac{DCG}{IDCG} \qquad (3.2, 3.3)$$

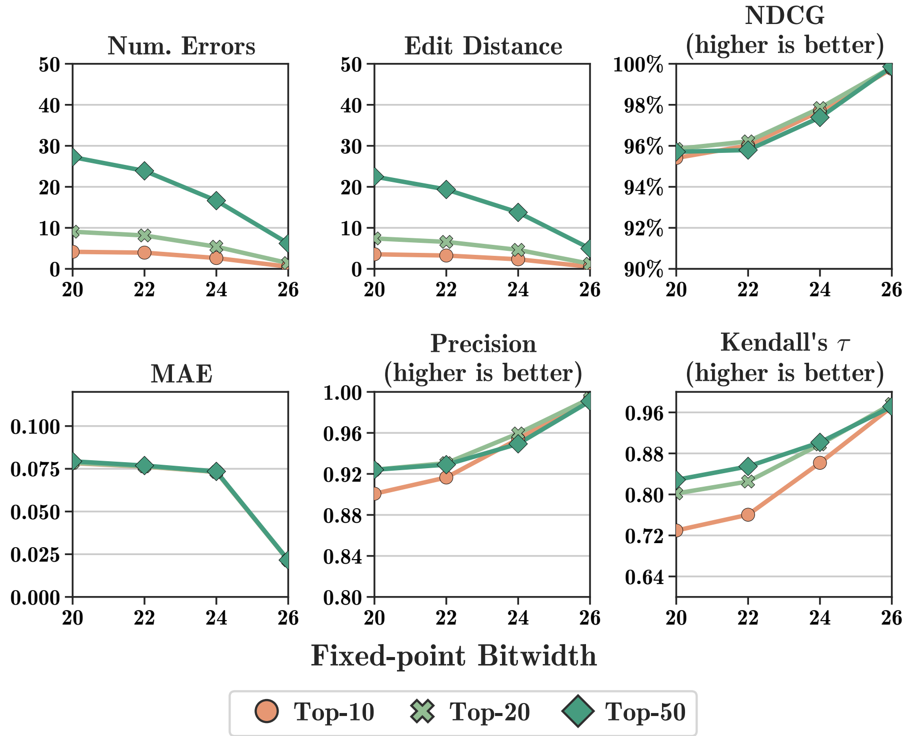



**Figure 3.7:** *We compute on all graphs the metrics in Figure 3.6 and include all the additional metrics presented in Section 3.5.3. We show results aggregated on all graphs, for increasing bit-widths and number of top ranked vertices. Aggregated accuracy metrics show trends in-line with Figure 3.6, and even low bit-width provides good predictions.*

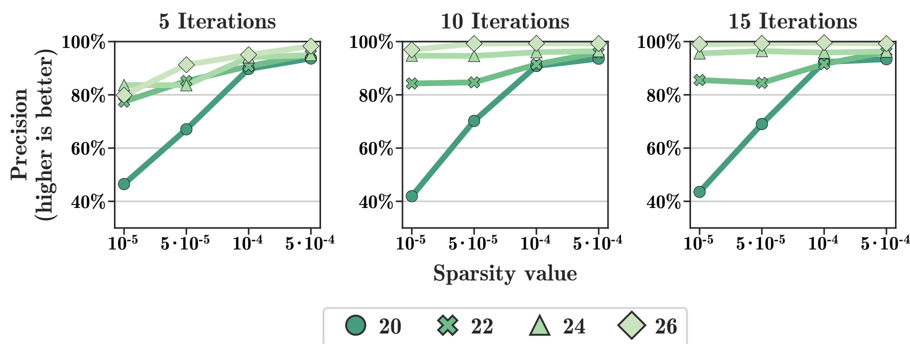



**Figure 3.8:** *Precision value, for decreasing sparsity, and increasing number of iterations, measured for different fixed-point bit-widths. Sparsity does not affect accuracy, except for very low bit-width, and 10 iterations are enough for convergence. Other metrics show similar trends as the Top-50 Precision.*

**Accuracy Discussion**

Figure 3.6 shows how metrics change by lowering the bit-width for each of the $2 \cdot 10^6$ edges graphs. Figure 3.7 shows additional accuracy metrics, aggregated on all graphs: *Mean Average Error* (MAE), *Precision* and *Kendall's* $\tau$. MAE evaluates how far FPGA PPR values are from the correct ones, while Precision measures the Top-N correctness without looking at the vertices order; just 20 bits are enough to retrieve 90 % of the best Top-50 items. Kendall's $\tau$ is a ranking metric that penalizes out-of-order predictions [180]. Results in Figure 3.7 are similar to Figure 3.6, with MAE and Precision mostly unaffected by a larger set of predictions.

Increasing bit-width is always beneficial, with diminishing returns. Using 26 bits provides near-to-perfect results, although even 22 or 24 bits provide satisfactory results, with more than half of the vertices being ranked correctly. 22 bits show a Top-10 edit distance of 3 and an NDCG value $> 95$ %. With 26 bits, the Top-20 edit distance is $< 3$, i.e. only 3 values in the first 20 are out-of-place. Results are impacted by graph distribution: Power-law (Holme and Kim) graphs, for which errors are lower, have dense communities, similarly to real social networks, while the behavior of the uniformly-distributed Erdős-Renyi is more unpredictable due to the lack of communities. Sparsity has a minor impact on accuracy (Figure 3.8): very low bit-width suffers from high sparsity, but in general results are consistent with Figure 3.6. We display the Top-50 precision, but other metrics show identical behaviors.

### 3.5.4 Fixed-point produces faster convergence

Fixed-point arithmetic produces faster convergence (Figure 3.9). We measure, after each iteration, the Euclidean norm of new and previous PPR values, to evaluate convergence. Less than 20 iterations are always enough for convergence, and even 10 iterations provide an error below $10^{-6}$ (a common convergence threshold for PPR [146]). Fixed-point arithmetic converges twice as fast compared to floating-point while preserving accuracy (Figure 3.6). In real computations, PPR stops when the error is below a threshold: a $2\times$ faster convergence immediately translates to an additional $2\times$ speedup over a floating-point implementation. To better showcase the performance improvement given by faster convergence, we compare the CPU floating-point baseline with our fixed-point FPGA hardware design, stopping the computation, not after 10 iterations as in Figure 3.5 but after a convergence error below $10^{-6}$ is met. Figure 3.10 shows how a 20-bit hardware design is an additional 80 % faster (8.95$\times$ versus the 4.92$\times$ in

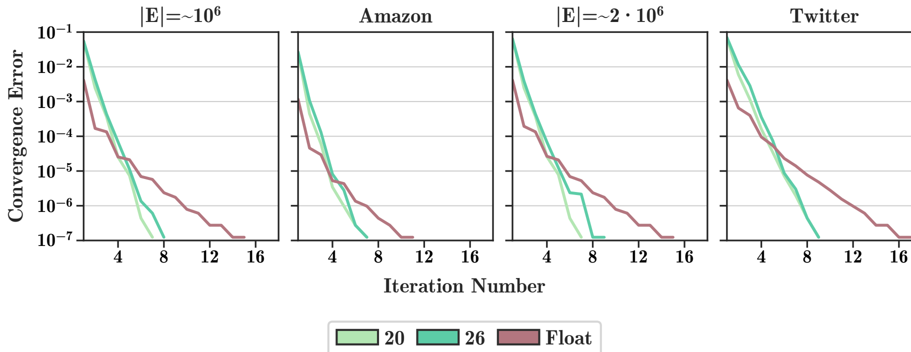



**Figure 3.9:** *Convergence error, measured as squared differences of PPR values computed iteration after iteration, for different bit-widths. Using fixed-point arithmetic results in faster convergence compared to 32-bits floating-point, often by a factor of 2x. Lines are truncated once the error is below* $10^{-7}$.

Figure 3.5) when using convergence as stopping criterion, compared to using a fixed number of iterations.

It is well-known that reduced-precision arithmetic gives faster convergence in the training of neural networks [133, 188]. However, it was not clear whether such findings could be applied to iterative graph algorithms such as PPR. What we find is that reduced-precision arithmetic does indeed make convergence faster. However, neural networks often employ extremely low-precision arithmetic (8 bits, 16 bits, often even less) without numerical instability or a significant loss of accuracy. In the case of PPR we observe that the gap between 26 bits and 20 bits is significantly less pronounced than moving from 32-bits floating-point arithmetic to fixed-point arithmetic. Moreover, accuracy starts rapidly decreasing when using less than 20 bits. As such, the low-precision arithmetic encountered in neural networks would not be applicable to the computation of PPR.

## 3.6 Final Remarks

This chapter presented a high-performance FPGA hardware design of a COO SpMV algorithm that leverages data-flow computation and reduced-precision fixed-point arithmetic. We applied our SpMV design to accelerate the PPR algorithm, outperforming a state-of-the-art CPU implementation by up to 6.8×, with up to 42× higher energy efficiency. With just 26-bits fixed-point values, we guarantee a speedup above 5.8× with negligible accuracy loss, with 2× faster convergence: average Top-10 Edit Distance

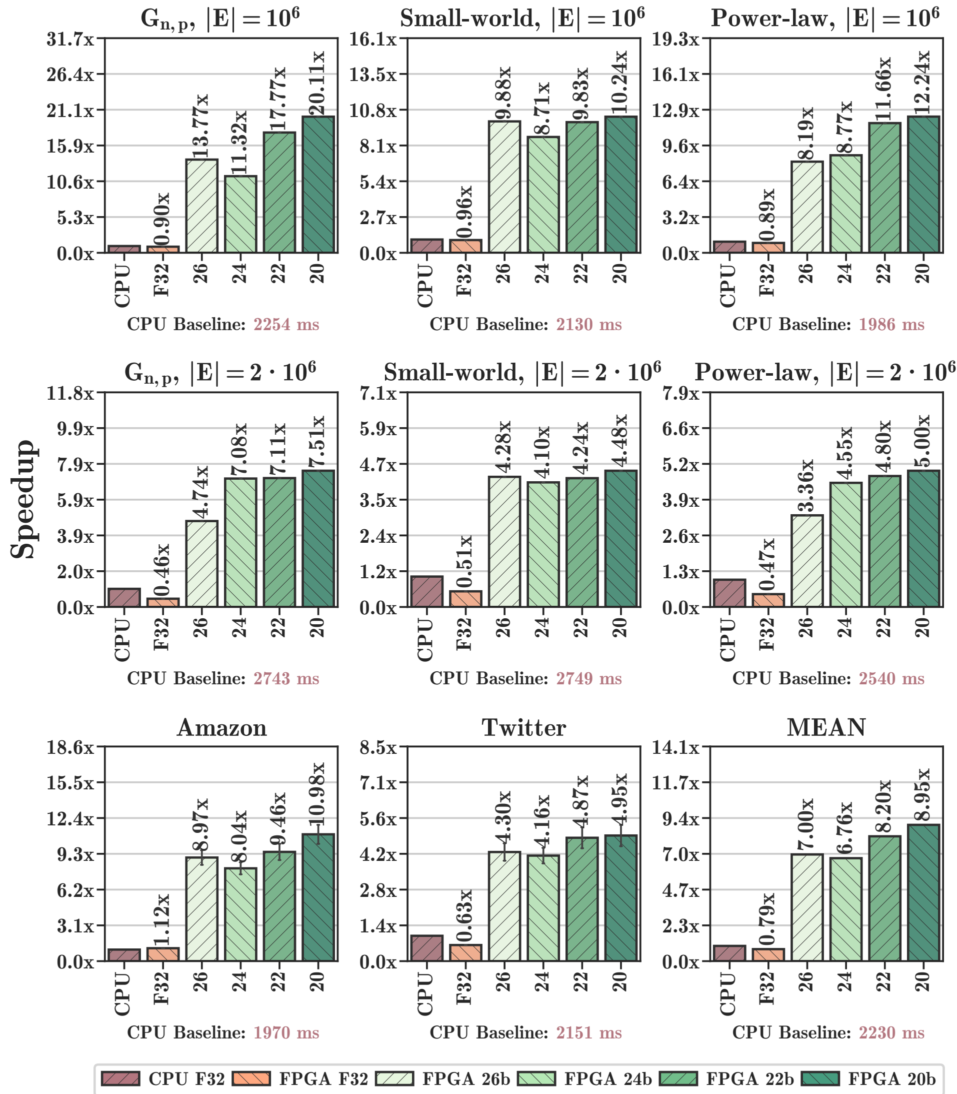



**Figure 3.10:** *Speedup of our Personalized PageRank hardware design (y-axis) with respect to the CPU baseline and the FPGA floating-point baseline, for decreasing bit-widths (x-axis), when accounting for the faster convergence obtained with reduced-precision fixed-point arithmetic. In most cases, the effective speedup is more than doubled compared to Figure 3.5.*

is below 1 and NDCG is above 99.9% compared to the CPU, showing how **graph ranking** algorithms can **benefit from approximate computing**.

Although the present work focuses on the design of a fixed point COO SpMV for a specific use-case and is not a general-purpose graph engine, we deem it valuable to integrate partitioning techniques [178,195] and support web-scale graphs, and study the optimal trade-off between partitioning overheads and FPGA resource utilization. To this end, Chapter 4 shows how to partition the input graph across multiple HBM pseudo channels, achieving greater throughput and enabling support for larger graphs.

Overall, this chapter has shown how current FPGA accelerator cards are promising but not yet ideal for high-performance sparse computations. To achieve state-of-the-art results, it is necessary to fully leverage their strengths (precise control over numerical precision and predictable throughput using streaming hardware designs) and to mask disadvantages with ad-hoc optimizations (limiting the impact of random accesses with batched requests in URAM or data replication in HBM). The goal of this thesis is to prove that it is possible to create **hardware designs** for sparse linear algebra that can be **easily reused and extendend** to different computations (such as PPR in this chapter and the Lanczos algorithm in the next chapter, Chapter 4), providing in each case a general framework that can achieve excellent performance with minimal adaptation overheads.

The results shown in this chapter go beyond FPGA accelerator cards. Recent GPUs provide exceptional memory bandwidth (up to $1.5\,\mathrm{TB/s}$ on the Nvidia Tesla A100 [150]), and more and more support for reduced-precision arithmetic, such as half-precision floating-point arithmetic. With the general trend of moving towards lower and lower precision arithmetic, thanks to breakthroughs in Deep Learning (DL) [62], it is not unlikely that future GPUs will also introduce hardware-accelerated lower-precision fixed-point arithmetic. If such hardware support becomes a reality, some of our results (e.g. faster convergence with low bit-width) will become immediately applicable to GPU implementations of PPR.

CHAPTER 4

# Solving Large Top-K Graph Eigenproblems with a Memory and Compute-optimized FPGA Design

*Large-scale eigenvalue computations on sparse matrices are a key component of graph analytics techniques based on spectral methods. In such applications, an exhaustive computation of all eigenvalues and eigenvectors is impractical and unnecessary, as spectral methods can retrieve the relevant properties of enormous graphs using just the eigenvectors associated with the Top-K largest eigenvalues. In this chapter, we propose a hardware-optimized algorithm to approximate a solution to the Top-K eigenproblem on sparse matrices representing large graph topologies. We build upon the SpMV hardware design presented in the previous chapter, and we prototype our algorithm through a custom FPGA hardware design that exploits HBM, Systolic Architectures, and mixed-precision arithmetic. We achieve a speedup of 6.22x compared to the highly optimized ARPACK library running on an 80-thread CPU, while keeping high accuracy and 49x better power efficiency.*

## 4.1 Introduction

Research in information retrieval and recommender systems has spiked novel interest in **spectral methods** [228], a class of Machine Learning algorithms able to detect communities in large social and e-commerce graphs, and compute the similarity of graph elements such as users or products [190]. At the core of many spectral methods lies the **Top-K eigenproblem for large-scale sparse matrices**, i.e. the computation of the eigenvectors associated with the largest eigenvalues (in modulo) of a matrix that stores only non-zero entries (Figure 4.1). For example, the famous Spectral Clustering algorithm boils down to computing the largest eigenvalues of a sparse matrix representing the graph topology [142]. Other applications of spectral methods include speech separation [17] and image segmentation, by thresholding eigenvectors [161, 181], or clustering the K-Nearest Neighbor graph of an image [192]. Even the PageRank values of the vertices in a graph [152] are nothing but the eigenvector corresponding to the highest eigenvalue of the modified adjacency matrix presented in Section 3.3. Despite the rise of theoretical interests for spectral methods, little research has focused on **improving the performance and scalability** of the Top-K sparse eigenproblem solvers, making them applicable to large-scale graphs.

### 4.1.1 Motivation

Most existing high-performance implementations of eigenproblem algorithms operate on dense matrices and are completely unable to process matrices with millions of rows and columns (each encoding, for example, the user's friends in a social network graph) [137]. Even the highly optimized multi-core implementation of LAPACK requires more than 3 minutes to solve the full eigenproblem on a small graph with $\sim 10^4$ vertices and $\sim 50 \cdot 10^4$ edges on a Xeon 6248, as the eigenproblem complexity scales at least quadratically with the number of vertices in the graph. Many implementations that support sparse matrices, on the other hand, are either forced to compute all the eigenvalues or require an expensive matrix inversion before solving the eigenproblem [104].

The need for high-performance Top-K sparse eigenproblem algorithms goes hand in hand with custom hardware designs that can outperform traditional architectures in **raw performance** and **power efficiency**, given how applications on-top of Top-K eigenproblem are mostly encountered in data centers. In this chapter, **we tackle both problems** by presenting a new Top-K sparse eigensolver whose building blocks are specifically optimized for high-performance hardware designs.

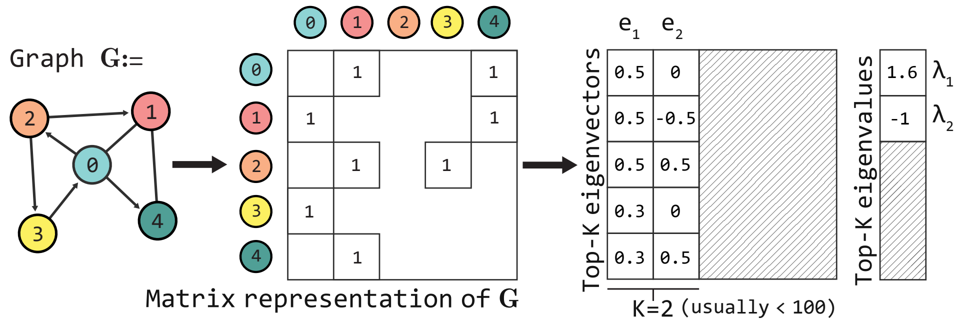



**Figure 4.1:** *Top-K eigencomputation of a graph $G$, represented as a sparse matrix. While $G$ can have millions of vertices, we often need just the Top-K eigenvectors ($K = 2$ in this example).*

### 4.1.2 Contributions

We introduce a novel algorithm to address the Top-K sparse eigenproblem and prototype it through a custom hardware design implemented on an Field-Programmable Gate Array (FPGA) accelerator card; to the best of our knowledge this is the first FPGA-based Top-K sparse eigensolver. Our algorithm is a 2-step procedure that combines the Lanczos algorithm (to reduce the problem size) [111] with the Jacobi algorithm (to compute the final eigencomponents) [170], as shown in Figure 4.2. The Lanczos algorithm, often encountered in Top-K sparse eigensolvers [198], has never been combined with the Jacobi algorithm. Part of the reason lies in their different computational bottlenecks: the Lanczos algorithm demands large memory bandwidth, while the Jacobi algorithm is strongly compute-bound. Our approach exploits the strengths of FPGA accelerator cards and overcomes the limitations of traditional architectures in this class of algorithms.

First, the Lanczos algorithm presents a Sparse Matrix-Vector Multiplication (SpMV) as its main bottleneck, an extremely memory-intensive computation with indirect and fully random memory accesses (Figure 4.2 Ⓑ). Optimizing SpMV requires high peak memory bandwidth and fine-grained control over memory accesses, without being able to rely on the traditional caching policies encountered in general-purpose architectures due to the randomness of these memory accesses. Our hardware design features an iterative dataflow SpMV with multiple Compute Units (CUs). This design leverages every High Bandwidth Memory (HBM) channel through a custom memory subsystem to efficiently handles indirect memory accesses. We build upon the SpMV hardware design presented in Chapter 3, adding support for HBM and for graph partitioning, enabling multiple CUs to work

in parallel on different portions of the input matrix.

Then, we introduce a Systolic Array (SA) design for the Jacobi eigenvalue algorithm, a computationally intensive operation that operates on reduced-size inputs ($K \times K$) (Figure 4.2 Ⓓ). The Jacobi algorithm maps naturally to a SA that ensures $\mathcal{O}(log(K))$ convergence, while traditional architectures do not ensure the same degree of performance. CPUs cannot guarantee that all the data are kept in the L1 cache and are unlikely to have enough floating-point arithmetic units to parallelize the computation. This results in $\Omega(K^2)$ computational complexity and execution times more than 50 times higher than a FPGA (Section 4.5). Instead, GPUs cannot fill all their Stream Processors, as the input size is much smaller than what is required to utilize the GPU parallelism fully [38].

Moreover, our FPGA-based hardware design employs optimized mixed-precision arithmetic, partially replacing traditional floating-point computations with faster fixed-precision arithmetic. While high numerical accuracy is usually demanded in eigenproblem algorithms, we employ fixed-precision arithmetic in parts of the design that are not critical to the overall accuracy and resort to floating-point arithmetic when required to guarantee precise results.

In summary, we present the following contributions:

- A novel algorithm for approximate resolution of **large-scale Top-K sparse eigenproblems** (Section 4.3), optimized for custom hardware designs.

- A modular mixed-precision FPGA design for our algorithm that **efficiently exploits** the available programmable logic and the bandwidth of **DDR and HBM** (Section 4.4).

- A performance evaluation of our Top-K eigendecomposition algorithm against the multi-core ARPACK CPU library, showing a speedup of 6.22× and a power efficiency gain of 49×, with a reconstruction error due to mixed-precision arithmetic as good as $10^{-3}$ (Section 4.5).

## 4.2 Related Work

To the best of our knowledge, no prior work optimizes Top-K eigenproblem for unstructured sparse matrices with custom FPGA hardware designs.

The most well-known large-scale Top-K sparse eigenproblem solver on CPU is the ARPACK library [117], a multi-core Fortran library that is also available in SciPy and MATLAB through thin software wrappers.

ARPACK implements the Implicitly Restarted Arnoldi Method (IRAM), a variation of the Lanczos algorithm that supports non-Hermitian matrices. Other sparse eigensolvers provide techniques that have been optimized for specific domains or matrix types, although none is as widely employed as ARPACK [6, 82, 116, 131].

On GPUs, the cuSOLVER [147] library by Nvidia provides a simple eigensolver based on the shift-inverse method that retrieves only the largest eigenvalue and its eigenvector (i.e. $K = 1$), which is significantly more limited than the general Top-K eigenproblem. The nvGRAPH library [146], also developed by Nvidia, provides an implementation of spectral clustering at whose core lies the Lanczos algorithm. However, the implementation of the inner Lanczos algorithm is not publicly available. To the best of our knowledge, there is no publicly available GPU implementation of the Lanczos algorithm that can solve large-scale sparse eigenproblems required by spectral methods. The MAGMA library [128] solves the Top-K sparse eigenproblem through the alternative LOBPCG algorithm [104], which requires multiple iterations (each containing at least one SpMV) to compute even a single eigenvector, to the contrary of the Lanczos algorithm. Other GPU-based Top-K eigensolvers are domain-specific, do not support large-scale inputs, or do not leverage features of modern GPUs such as HBM memory or mixed-precision arithmetic [49, 53]. Eigensolvers for dense matrices are more common on GPUs, as they easily exploit the enormous memory bandwidth of these architectures: Myllykoski et al. [137] focus on accelerating the case of dense high-dimensional matrices (around $10^5$ rows) while Cosnuau [38] operates on multiple small input matrices. Clearly, none of the techniques that operate on dense matrices can easily scale to matrices with millions of rows as simply storing them requires terabytes of memory. Graphics Processing Unit (GPU) implementations of the Jacobi algorithm are rare and only target small dense matrices [34, 202].

Specialized hardware designs for eigensolvers are limited to resolving the full eigenproblem on small dense matrices through the QR-Householder Decomposition and Jacobi eigenvalue algorithm. Most formulations of the Jacobi algorithm [29, 73] leverage Systolic Array, a major building block of high performance domain-specific architectures from their inception [107] to more recent results [14, 97, 112]. However, hardware designs of the Jacobi algorithm based on SA cannot scale to large matrices, as the resource utilization scales linearly with the size of the matrix. Implementations of the *QR-Houseolder* algorithm face similar problems [15, 197] as they also leverage systolic architectures, although research research about resource-efficient designs do exist [47].

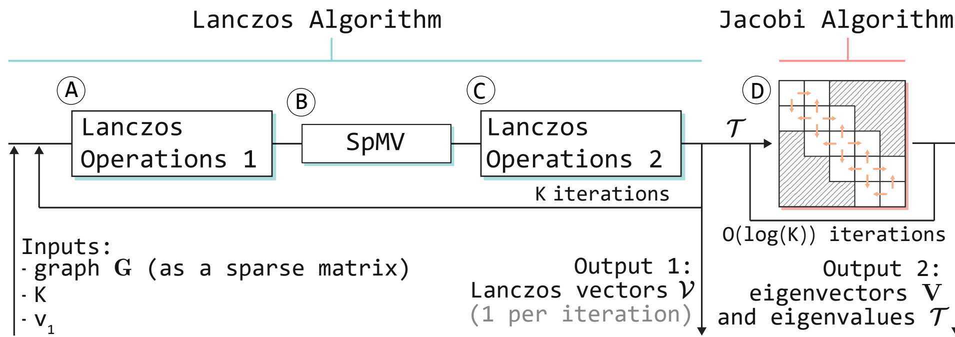



**Figure 4.2:** *Steps of our Top-K sparse eigenproblem solver, which combines the Lanczos algorithm with a Systolic Array formulation for the Jacobi eigenvalue algorithm.*

## 4.3 Solving the Top-K Sparse Eigenproblem

The core step of algorithms like Spectral Clustering is a Top-K sparse eigenproblem, i.e. finding eigenvalues and eigenvectors of sparse matrices representing, for instance, graphs with millions of vertices and edges.

Given a sparse square matrix $\mathrm{M} \in \mathbb{R}^{n\times n}$ and an integer $K \ll n$ the goal of the Top-K sparse eigenproblem is to find the $K$ eigenvalues with the highest magnitude, and their associated eigenvectors. While a full eigendecomposition factorizes $\mathrm{M}$ as $\mathrm{M} = \mathrm{Q}\mathrm{Q}^T$, with $\mathrm{Q} \in \mathbb{R}^{n\times n}$ being the eigenvector matrix and $\in \mathbb{R}^{n\times n}$ being the diagonal eigenvalue matrix, the Top-K sparse eigenproblem produces the approximate decomposition $\mathrm{M} \approx \mathrm{M}_K = \mathrm{Q}_{K\,K}\mathrm{Q}_K^T$, with $\mathrm{Q}_K \in \mathbb{R}^{n\times K}$ and $_K \in \mathbb{R}^{K\times K}$. $_K$ is a diagonal matrix containing the Top-K largest eigenvalues in modulo, while $\mathrm{Q}_K$ contains its corresponding eigenvectors. Indeed, computing all the $n$ eigenvalues of the matrix is intractable for large matrices and redundant for many applications that require only a handful of eigencomponents. For example, Spectral Clustering and many of its variations rely only on the Top-K eigenvectors, with K rarely above $\sim 10$.

In this chapter, we propose a novel algorithm to solve the Top-K sparse

$$\begin{bmatrix} \alpha_1 & \beta_1 & 0 & 0 & 0 \\ \beta_1 & \alpha_2 & \beta_2 & 0 & 0 \\ 0 & \beta_2 & \alpha_3 & \beta_3 & 0 \\ 0 & 0 & \beta_3 & \alpha_4 & \beta_4 \\ 0 & 0 & 0 & \beta_4 & \alpha_5 \end{bmatrix}$$

**Figure 4.3:** *Example of* $(5 \times 5)$ *tridiagonal matrix, obtained as output of the Lanczos algorithm for* $K = 5$.

**Algorithm 6** Pseudo-code of the Lanczos algorithm for the Top-K eigenvectors

```
1:  function LANCZOS(M, K, v_1)
2:      β_1 ← 0                                          ▷ Initialization
3:      v_0 ← 0_N
4:      for i in 1, K do                                 ▷ Main Lanczos loop
5:          ▷ Normalize Lanczos vector obtained in the previous iteration
6:          if i > 1 then
7:              β_i ← ||w'_{i−1}||_2
8:              v_i ← w'_{i−1}/β_i                       ▷ Compute new Lanczos vector
9:          end if
10:         ▷ Begin computation of new Lanczos vector
11:         w_i ← M v_i                                  ▷ Sparse matrix-vector multiplication
12:         α_i ← w_i v_i
13:         w'_i ← w_i − α_i v_i − β_i v_{i−1}
14:         Orthogonalize w'_i with respect to V
15:     end for
16:     ▷ Tridiagonal matrix T and Lanczos vectors V
17:     return {T = [α_1, ..., α_K], [β_1, ..., β_{K−1}]}
18:     return V = [v_1, ..., v_K]
19: end function
```

eigenproblem, combining the Lanczos algorithm and the Jacobi eigenvalue algorithm. Our technique is particularly suited for highly optimized and modular hardware designs. The first phase leverages the Lanczos algorithm, taking as input the original matrix $\mathrm{M}$, the number of desired eigencomponents $K$ and an *L2-normalized* random vector $v_1 \in \mathbb{R}^n$. We initialize $v_1$ to have all values equal to $1/n^2$, but any other L2-normalized initialization is acceptable. The Lanczos algorithm outputs a $K \times K$ symmetric tridiagonal matrix $\mathcal{T}$ (Figure 4.3) and a set of orthogonal Lanczos vectors $\mathcal{V} \in \mathbb{R}^{K \times n}$. As the second step, we apply the Jacobi eigenvalue algorithm to $\mathcal{T}$. This algorithm transforms $\mathcal{T}$ into a diagonal matrix containing its eigenvalues, and returns a matrix $\mathrm{V}$ with the eigenvectors of $\mathcal{T}$. Each eigenvalue $\lambda$ of $\mathcal{T}$ is also an eigenvalue of the original matrix $\mathrm{M}$. Moreover, if $x$ is the eigenvector of $\mathcal{T}$ associated to $\lambda$, then $\mathcal{V}x$ is the eigenvector of $\mathrm{M}$ associated to $\lambda$. $\mathrm{M}_K$ can be obtained as $\mathrm{M}_K = (\mathcal{V}\mathrm{V})\mathcal{T}(\mathcal{V}\mathrm{V})^T$, although many applications in spectral analysis only require the Top-K eigenvalues and eigenvectors of $\mathrm{M}$ instead of retrieving $\mathrm{M}_K$.

$$\begin{bmatrix} c_i & s_i \\ -s_i & c_i \end{bmatrix} \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} \begin{bmatrix} c_i & -s_i \\ s_i & c_i \end{bmatrix} = \begin{bmatrix} \alpha' & 0 \\ 0 & \delta' \end{bmatrix}$$

**(a)** *Operations for the Diagonal Processor $p_{ii}$ (Figure 4.5A).*

$$\begin{bmatrix} c_i & s_i \\ -s_i & c_i \end{bmatrix} \begin{bmatrix} \alpha & \beta \\ \gamma & \delta \end{bmatrix} \begin{bmatrix} c_j & -s_j \\ s_j & c_j \end{bmatrix} = \begin{bmatrix} \alpha' & \beta' \\ \gamma' & \delta' \end{bmatrix}$$

**(b)** *Operations for the Offdiagonal Processor $p_{ij}$ (Figure 4.5C).*

$$\begin{bmatrix} w & x \\ y & z \end{bmatrix} \begin{bmatrix} c_j & -s_j \\ s_j & c_j \end{bmatrix} = \begin{bmatrix} w' & x' \\ y' & z' \end{bmatrix}$$

**(c)** *Operations for the Eigenvector Processor $p_{ij}$ (Figure 4.5D).*

**Figure 4.4:** *Operations performed by different processors in the Jacobi eigenvalue Systolic Array architecture. Values $c_i$ and $s_i$ indicate $cos(\theta_i)$ and $sin(\theta_i)$, with $\theta_i = \frac{1}{2}\arctan\frac{2\beta}{\alpha-\delta}$.*

### 4.3.1 The Lanczos Algorithm

The Lanczos algorithm retrieves the Top-K eigencomponents of a matrix and is often employed as a building block of large-scale eigenproblem algorithms [32, 65, 117]. The algorithm takes as input a real matrix $\mathrm{M}$, the value $K$ and an *L2-normalized* vector $v_1 \in \mathbb{R}^n$. The $K \times K$ output tridiagonal matrix $\mathcal{T}$ is significantly smaller than the input ($K \ll n$) and also simpler in structure, as elements outside of the band enclosing the main diagonal and the ones immediately above and below are zero. Reducing the dimensionality of the problem instance is crucial for rendering the problem tractable and preserving only the relevant components of the input matrix. The pseudo-code of the algorithm is provided in Algorithm 6. For each of the $K$ iterations, it computes a Lanczos vector $v_i$ by normalizing $w'_{i-1}$, obtained at the previous iteration (lines 6–7 and Figure 4.2A). From $v_i$, we obtain $w'_i$ by projecting the matrix **M** into $v_i$ (lines 11–13 and Figure 4.2B), followed by an orthogonalization (line 14 and Figure 4.2C). Mathematically, the algorithm builds an orthonormal basis $\mathcal{V}$ for an order-$K$ Krylov subspace by repeated embedding of the vector $v$ into the matrix **M** (Algorithm 6, line 11), followed by normalization (line 7–8) and orthogonalization (line 14) on the following iteration. The algorithm produces one of the $K$ vectors in $\mathcal{V}$ for each iteration. From $\mathcal{V}$, we derive the eigenvalues and eigenvectors of $\mathrm{M}$. As we compute only $K$ eigenvectors and $K \ll n.$, the Lanczos is algorithm very efficient.

The Lanczos algorithm is particularly efficient on sparse matrices, as its most expensive operation is an iterative SpMV, bounding its computational complexity to $\mathcal{O}(K \cdot E)$, with $E$ being the number of non zero elements

**Algorithm 7** Jacobi eigenvalue algorithm with Systolic Arrays

```
 1: function JACOBI(𝒯)
 2:     V ← 𝟙_K                                          ▷ Identity matrix of size K × K
 3:     repeat
 4:         for i in 1, K/2 do                                          ▷ Diagonal CU
 5:             p_ii ← 𝒯[2i : 2i + 1, 2i : 2i + 1]
 6:             θ_i ← 1/2 arctan 2β/(α−δ)
 7:             Rotate p_ii                                ▷ Full equation in Figure 4.4a
 8:             Propagate c_i and s_i
 9:         end for
10:         for j in 1, K/2 − 2 do                                   ▷ Offdiagonal CU
11:             i ← j + 1
12:             Receive c_i, c_j, s_i, s_j from p_ii, p_jj
13:             p_ij ← 𝒯[2i : 2i + 1, 2j : 2j + 1]
14:             Rotate p_ij                                ▷ Full equation in Figure 4.4b
15:         end for
16:         for i in 1, K/2 do                                      ▷ Eigenvector CU
17:             for j in 1, K/2 do
18:                 v_ij ← V[2i : 2i + 1, 2j : 2j + 1]
19:                 Receive c_j, s_j from p_jj
20:                 Rotate v_ij                            ▷ Full equation in Figure 4.4c
21:             end for
22:         end for
23:         Permute rows and columns of 𝒯 and V                          ▷ Figure 4.5
24:     until 𝒯 becomes diagonal
25:     Output 𝒯                                       ▷ Eigenvalues of the input 𝒯
26:     Output V                                      ▷ Eigenvectors of the input 𝒯
27: end function
```

of $\mathrm{M}$. In our hardware design, we optimize the memory-intensive SpMV computation through multiple independent CUs, so that we can take advantage of all the available 32 HBM channels of a Xilinx Alveo U280 FPGA accelerator card (Section 4.4.2).

This algorithm is prone to numerical instability as the Lanczos vectors $\mathcal{V}$ can quickly lose pairwise orthogonality if $K$ is very large. To prevent instability, we normalize the input matrix in *Frobenius norm* as eigencomponents are invariant to constant scaling: values of the matrix are in the range $(-1, 1)$, which implies that eigenvalues and eigenvectors are also in the range $(-1, 1)$. This property enables the use of fixed-point arith-

metic to improve performance and reduce resource usage (Section 4.5.3). We further improve numerical stability by adopting a version of the algorithm that reorders operations [153] and reorthogonalizes Lanczos vectors through Gram-Schmidt orthogonalization in each iteration [154]. The order of operation in Algorithm 6 has been proven to be the most stable [153] and comes at no implementation cost. Reorthogonalization (Algorithm 6, line 10) requires $K^2/2$ more operations of cost $\mathcal{O}(n)$, increasing complexity to $\mathcal{O}(K(E + nK^2/2))$. We also introduce the option of performing Gram-Schmidt reorthogonalization every 2 iterations, for a lower overhead of $\mathcal{O}(n(K/2)^2/2)$, with negligible accuracy loss (Section 4.5.3). In practice, execution time is usually dominated by SpMV making reorthogonalization a viable option.

### 4.3.2 The Jacobi Eigenvalue Algorithm

The Jacobi eigenvalue algorithm computes the eigenvalues and eigenvectors of a dense symmetric real matrix. It is an iterative procedure that performs rotations on square submatrices. While being highly computationally intensive due to the large number of trigonometric operations required for rotations ($\Omega(K^2)$ per iteration in a naïve implementation), this algorithm is particularly well suited to solve eigenproblems on small tridiagonal matrices by exploiting their structure. As many matrix values are zero and cannot introduce data dependencies in rotations, it is possible to parallelize the entire computation at the hardware level.

The Jacobi eigenvalue algorithm has sought many formulations to improve its parallelism and resource utilization. The best-known formulation of the algorithm was proposed by Brent and Luk [25] and has been the standard for implementing the algorithm on FPGA to this day [29, 73]. Our design improves this formulation with a more resource-efficient procedure for interchanging rows and columns, and its structure is shown in Figure 4.5.

We employ a SA design that maps the input matrix as $2 \times 2$ submatrices to $K^2/4$ adjacent *processors* (or CU) (Figure 4.5, Ⓐ). The systolic architecture propagates the rotation angles Ⓑ and the values stored in each processor Ⓔ.

Starting from $\mathcal{T}$, the algorithm sets to zero $K/2$ off-diagonal entries per iteration using rotations. Diagonal processors annihilate $\beta$ and $\gamma$ components (Algorithm 7, line 7) with a rotation of angle $\theta$. This angle is propagated (line 8) to the off-diagonal processor (line 9), and to the eigenvector processor that applies the same rotation to the identity matrix (line 14).

To ensure convergence, diagonal CUs are fed non-zero elements at each

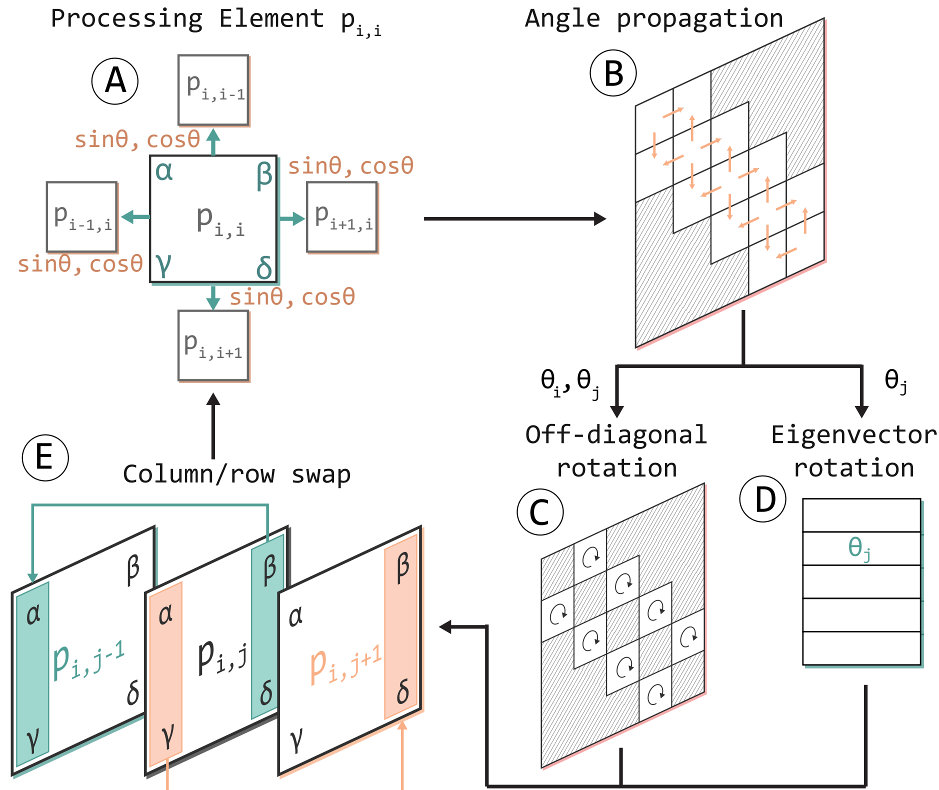



**Figure 4.5:** *Steps of the Jacobi eigenvalues computation using Systolic Arrays. Each Processing Element (PE)* $p_{ij}$ *holds 4 values* $\alpha$, $\beta$, $\gamma$, $\delta$, *and* $\theta = \frac{1}{2} \arctan \frac{2\beta}{\alpha - \delta}$.

iteration in the $\beta$ and $\gamma$ position. New non-zero elements are provided to the diagonal CUs by swapping rows and columns since eigencomponents are invariant to linear combinations of the input matrix. We improve the swap procedures of Brent and Luk [25] by swapping vectors *in reverse*, obtaining the same results with fewer resources (Section 4.4.3).

The SA formulation allows performing each iteration of the algorithm in constant time, enabling complexity equal to the number of iterations, $\mathcal{O}(log(K))$, instead of having cost above $\Omega(K^2 \cdot \log(K))$ due to the matrix multiplications [25]. The systolic formulation of the Jacobi eigenvalue algorithm cannot scale beyond very small matrices ($K \sim 32$) due to the large number of resources required for trigonometric operations. While resource utilization has prevented widespread adoption of the Jacobi eigenvalue algorithm for general eigenproblem resolution, we do not incur in this limitation: we apply this algorithm on small $K \times K$ inputs by design, as we first reduce the problem size through the Lanczos algorithm.

## 4.4 The Proposed FPGA Hardware Design

This section presents our custom FPGA-based hardware design for the Top-K sparse eigenproblem algorithm previously introduced. The logical division between the Lanczos and Jacobi algorithms is also present in the hardware implementation. Our hardware design is composed of two macro-areas that are mapped to separate reconfigurable Super Logic Regions (SLRs) of the FPGA, to provide more efficient resource utilization and higher flexibility in terms of clock frequency, memory interfaces, and reconfigurability. Figure 4.6 shows a high level view of our FPGA design. We prototyped our hardware design on a Xilinx Alveo U280 accelerator card with 8 GB of HBM2 memory and 32 GB of DDR4 memory. The accelerator card is equipped with a `xcu280-fsvh2892-2L-e` FPGA with 3 SLR and whose available resources are reported in Table 4.1. The Lanczos algorithm, being a memory-intensive computation, is mapped to SLR0, which provides direct access to all the HBM2 memory interfaces on the accelerator card. SLR1 and SLR2 host different replicas of the IP core implementing the Jacobi algorithm, optimized for different numbers of eigenvectors $K$. SLR2 can either be left empty or reconfigured to provide more Jacobi cores specialized for multiple numbers of eigenvectors $K$. For example, while a Jacobi core with $K = 32$ can compute any number of eigenvectors $k' < 32$, having additional Jacobi cores for $k' = 8, k' = 16, \dots$ can boost performance when computing a number of eigenvectors below $32$.

### 4.4.1 Lanczos Hardware Design

The left part of Figure 4.6 highlights the Lanczos algorithm hardware design components. Partitions of the sparse input matrix are read from HBM Ⓐ and distributed to 5 parallel SpMV CUs Ⓑ (Algorithm 6, line 11). Partial results from every partition are merged Ⓒ into a single vector to be used by the remaining linear operations Ⓓ (lines 7, 8, 12, 13, 14). Operations are then repeated $K$ times to produce the $3 \cdot K - 2$ values in the tridiagonal matrix $\mathcal{T}$ and the $K$ Lanczos vectors in $\mathcal{V}$, stored in DDR memory. Algorithm 8 extends the pseudo-code in Algorithm 6 with lower-level details about our implementation. As the computation is divided on 5 CUs, we also use a partitioned input matrix $\mathrm{M}_1, \dots, \mathrm{M}_5$ and partitioned temporary buffers $v_{tmp}$, $v_{nxt}$ and $v_i$. Most operations (e.g. SpMV) operate through independent CUs, with a central control unit handling the control flow of the algorithm and results' aggregation (e.g. the loop in line 3 and adding partial $\alpha_i$ in line 11). For simplicity, we show orthogonalization as if it were done in each iteration. This step is optional, and satisfactory results

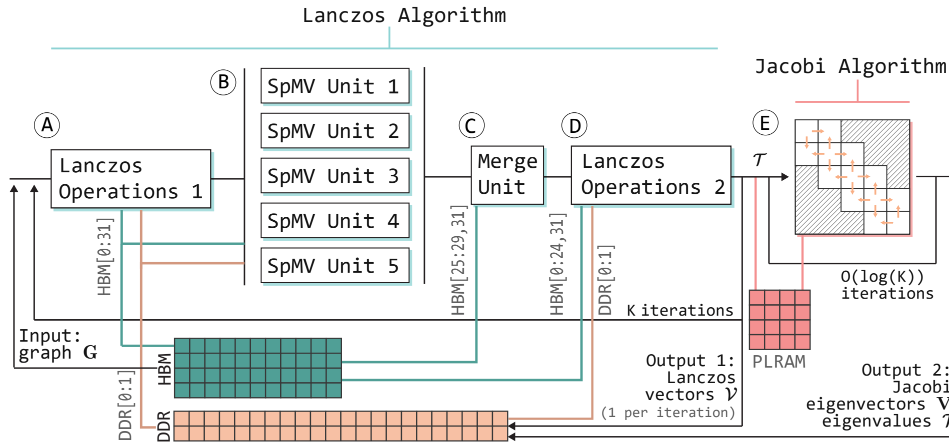



**Figure 4.6:** *High-level architecture of our Top-K Sparse Eigencomputation FPGA design. We highlight interconnections between FPGA computational units and the FPGA board memory. The design is divided into two main parts, mapped to the Lanczos and Jacobi algorithms. Each part is executed for $K$ and $\mathcal{O}(log(K))$ iterations, respectively. The two parts of the design are located into different SLRs and communicate a small amount of data: we use PLRAM as a fast scratchpad memory for data exchange.*

can be achieved by reorthogonalizing every two iterations as well, reducing the computational cost of this step (Section 4.5.3).

### 4.4.2 SpMV Hardware Design

The biggest bottleneck in the Lanczos algorithm is an iterative SpMV computation (Algorithm 8, line 10). While other computations in the Lanczos algorithm are relatively straightforward to optimize and parallelize, SpMV is well-known for being a complex, memory-intensive computation that presents indirect and random memory accesses [143]. Although significant research has been made into developing high-performance SpMV implementations on FPGA [68, 90, 155, 172, 195], the Lanczos algorithm introduces circumstances that prevents us from using an out-of-the-box FPGA SpMV implementation. The SpMV design must perform multiple iterations without communication from device to host, as data transfer and synchronizations would hinder performance. Then, the SpMV must be easily partitioned and replicated to provide flexibility over the hardware resources. Finally, we require access to multiple HBM channels to maximize the overall memory bandwidth achieved in the computation.

Our final SpMV design extends and improves the one proposed in Chapter 3 in the context of graph ranking algorithms [158], which are also varia-

**Algorithm 8** Detailed lower-level pseudo-code of the Lanczos algorithm

**Require:** Input Matrix $\mathrm{M}$ in COO format, partitioned in $\mathrm{M}_1 \dots \mathrm{M}_5$
**Require:** $K$, number of output eigenvectors
**Require:** $L2$-normalized input vectors $v_1 := \{v_1^1, v_1^2, v_1^3, v_1^4, v_1^5\}$
**Require:** Temporary Vectors $v_{tmp} := \{v_{tmp}^1, v_{tmp}^2, v_{tmp}^3, v_{tmp}^4, v_{tmp}^5\}$
**Require:** Temporary Next Vectors $v_{nxt} := \{v_{nxt}^1, v_{nxt}^2, v_{nxt}^3, v_{nxt}^4, v_{nxt}^5\}$

1: **function** LANCZOS($\mathrm{M}, K, v_1, v_{tmp}, v_{nxt}$)
2: $\quad \alpha_1 \leftarrow 0; \quad \beta_1 \leftarrow 0$ ▷ Initialization
3: $\quad$ **for** $i$ in $1, K$ **do**
4: $\quad\quad$ ▷ Normalize and compute new Lanczos vector
5: $\quad\quad$ **if** $i \neq 1$ **then**
6: $\quad\quad\quad \beta_i \leftarrow \sqrt{\sum_{j=1}^{5} (v_{nxt}^j)^2}$ ▷ $\beta_i \leftarrow \|v_{nxt}\|_2$
7: $\quad\quad\quad v_i^{[1\dots5]} \leftarrow \forall_{j\in[1\dots5]}(v_{nxt}^j/\beta_i)$ ▷ $v_i \leftarrow v_{nxt}/\beta_i$
8: $\quad\quad$ **end if**
9: $\quad\quad$ ▷ Projection on the Krylov subspace for the next Lanczos vector
10: $\quad\quad v_t^{[1\dots5]} \leftarrow SpMV(\mathrm{M}_1 \dots \mathrm{M}_5, v_i^{[1\dots5]})$
11: $\quad\quad \alpha_i \leftarrow \sum_{j=1}^{5} v_i^j \cdot v_t^j$
12: $\quad\quad v_{nxt}^{[1\dots5]} \leftarrow v_{tmp}^{[1\dots5]} - \alpha_i v_i^{[1\dots5]} - \beta_i v_{i-1}^{[1\dots5]}$
13: $\quad\quad$ ▷ Orthogonalization
14: $\quad\quad$ **for** $j \in [1, i]$ **do**
15: $\quad\quad\quad$ **if** $j\%2 \neq 0$ **then**
16: $\quad\quad\quad\quad o \leftarrow \sum_{k=1}^{5} v_{j,k} \cdot v_{t,k}$
17: $\quad\quad\quad\quad v_t^{[1\dots5]} \leftarrow v_t^{[1\dots5]} - o \cdot v_j^{[1\dots5]}$
18: $\quad\quad\quad$ **else**
19: $\quad\quad\quad\quad o \leftarrow \sum_{k=1}^{5} v_{j,k} \cdot v_{n,k}$
20: $\quad\quad\quad\quad v_n^{[1\dots5]} \leftarrow v_n^{[1\dots5]} - o \cdot v_j^{[1\dots5]}$
21: $\quad\quad\quad$ **end if**
22: $\quad\quad\quad$ **if** $i == j$ **then** ▷ Copy to temporary vector
23: $\quad\quad\quad\quad v_t^{[1\dots5]} \leftarrow v_n^{[1\dots5]}$
24: $\quad\quad\quad$ **end if**
25: $\quad\quad$ **end for**
26: $\quad$ **end for**
27: $\quad$ **return** $\{\mathcal{T} = [\alpha_1, \dots, \alpha_K], [\beta_1, \dots, \beta_{K-1}]\}; \quad \mathcal{V} = [v_1, \dots, v_K]$
28: **end function**

tions of the power iteration method used by the Lanczos algorithm. Below we introduce how we leveraged HBM memory in our SpMV design.

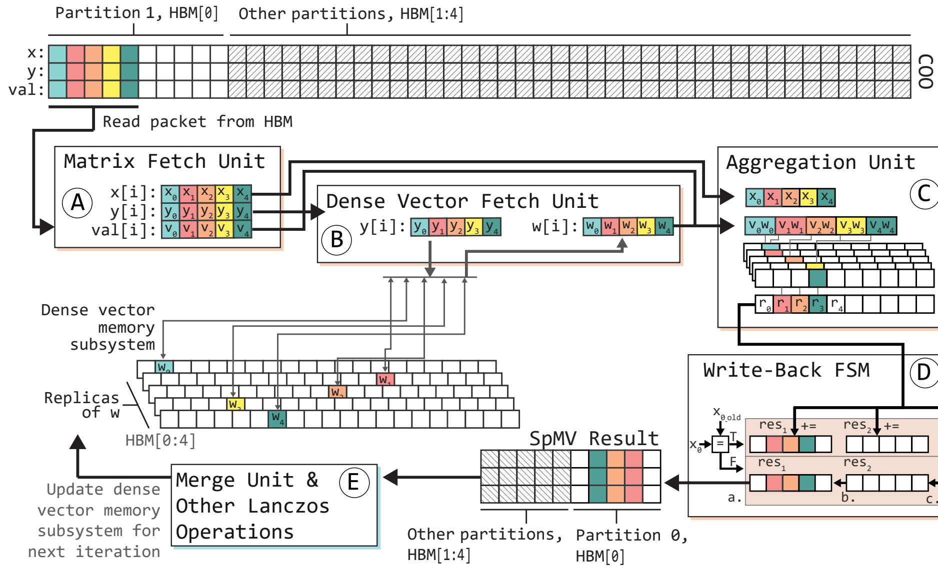



**Figure 4.7:** *Block diagram of one iterative SpMV CU. Each CU processes a portion of the input matrix through a 4-stage dataflow design, and results are replicated on the dense vector memory subsystem after each iteration. Thanks to multiple CUs and the dense vector memory subsystem we can scale to large matrices while preserving efficient HBM bandwidth utilization.*

### SpMV Dataflow Architecture

As SpMV is an extremely memory-intensive computation, a good SpMV implementation should make efficient use of the memory bandwidth made available by the underlying hardware. Figure 4.7 shows the structure of one of our SpMV CUs. We employ a streaming dataflow SpMV design that reads the sparse input matrix stored using the Coordinate (COO) format. In the COO format, non-zero entries of the matrix are stored using 3 32-bits values: the row and column index in the matrix and the value itself. Compared to other sparse matrix data-layouts, such as Compressed Sparse Row (CSR), the COO format does not present indirect data accesses that can severely reduce the opportunities for a pipelined design. The *Matrix Fetch Unit* in each CU is connected to a single HBM channel and reads, for each clock cycle, a packet of 512 bits containing 5 non-zero matrix entries Ⓐ. Memory transactions happen in continuous bursts of maximum AXI4 length (256 beats): each CU reads the matrix at the maximum bandwidth offered by the HBM channel (14.37 GB/s, for a total of 71.87 GB/s using 5 CU). For each of the 5 non-zero values in each COO packet, the *Dense Vector Fetch Unit* performs a random access to the SpMV dense vector Ⓑ.

This step is critical to the overall SpMV performance: compared to the SpMV design in Section 3.4.2 and in Parravicini et al. [158], we leverage HBM instead of UltraRAM (URAM), achieving better scalability and performance. We detail our *Dense Vector Memory Subsystem* below and in Figure 4.8. The *Aggregation Unit* sums results within a single data-packet that refers to the same matrix column Ⓒ. A *Write-Back Finite-State Machine* stores results of each CU to HBM Ⓓ. Each write-transaction is a 512-bits data packet containing up to 15 values, each referring to a single matrix row. As another improvement over the design in Section 3.4.2, we reduce the number of write transactions by 3 times the average number of non-zeros per row. As such, we can store results through the same HBM channels of the dense vector with no detriment to performance.

To the contrary of the original SpMV design in Section 3.4.2, we support multiple SpMV CUs that operate on partitions on the COO input matrix, created by assigning an equal number of rows to each CU. We employ up to 5 SpMV CUs (Figure 4.6). While in principle it is possible to place more CUs, our current design is limited by the hardened HBM controller in the Alveo U280 that prevents using more than 32 AXI master channels, which we fully employ [213]. Each SpMV CU computes a portion of the output vector: partial results are aggregated by the *Merge Unit* (Figure 4.6 Ⓒ) and replicated across HBM channels to use them in the following iteration.

#### SpMV Dense Vector Memory Subsystem

Each SpMV CU processes 5 non-zero matrix entries per clock cycle, and for each non-zero entry it must perform a random access on a dense vector of size $n$ (in our case, the Lanczos vector $v_i$ at iteration $i$). As each AXI master channel can handle only one read transaction per cycle, we need to replicate the dense vector 5 times, similarly to [102]. The hardened AXI switch in the Alveo U280 renders it highly inefficient to attach multiple AXI master channels to the same HBM bank: only 32 AXI master channels are available, and small memory transactions (32 bits) have the same performance as larger transactions, preventing sustained bandwidth sharing [36, 126, 205]. We solve the issue by leveraging the abundant HBM memory on the Alveo U280, and replicating the dense vector 5 times for each CU, as in Figure 4.8. A more flexible AXI switch would enable multiple 32-bits read transactions on the same HBM channel in a single clock cycle, reducing the demand for data replication. Our HBM-based memory subsystem marks a significant improvement from the design in Section 3.4.2, as we avoid URAM to store the intermediate dense vector and results. Instead of being limited by the FPGA's 90 MB of URAM, we store

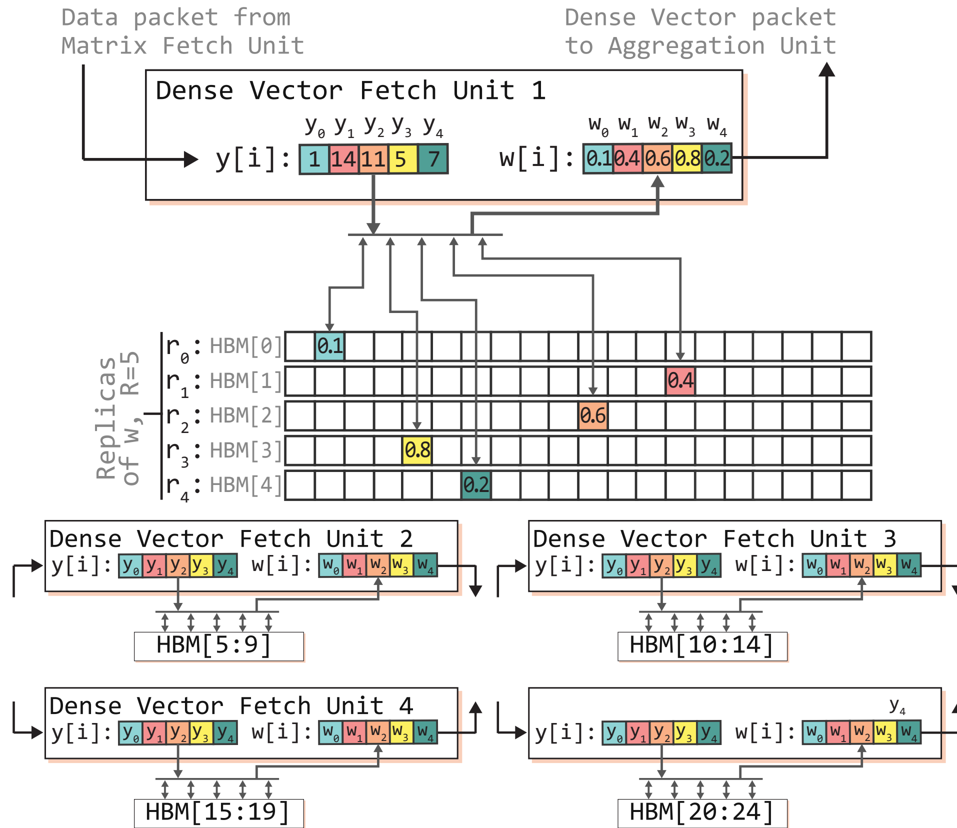



**Figure 4.8:** *Dense vector memory subsystem of our SpMV FPGA design. Index $y_i$ accesses replica $r_i$, guaranteeing a pipelined design with 5 random vector accesses per clock cycle. Each Fetch Unit accesses 5 different replicas of the dense vector, and provides the results to its corresponding Aggregation Unit.*

the dense vector using individual HBM banks with 250 MB of capacity, allowing computations on matrices with up to 62.4 million rows. Moreover, high URAM consumption significantly limits the maximum attainable frequency, while we do not incur into this limitation (Table 4.1).

### 4.4.3 Jacobi Systolic Array Design

The Jacobi eigenvalue algorithm is very computationally intensive. Although it processes a small input of size $K \times K$, unoptimized implementations still require a significant amount of time as the algorithm contains a large number of dense matrix multiplications. Moreover, its convergence rate is implementation-dependent and as high as $\mathcal{O}(K^2)$. By adopting a SA-based design, we overcome both issues. By parallelizing the com-

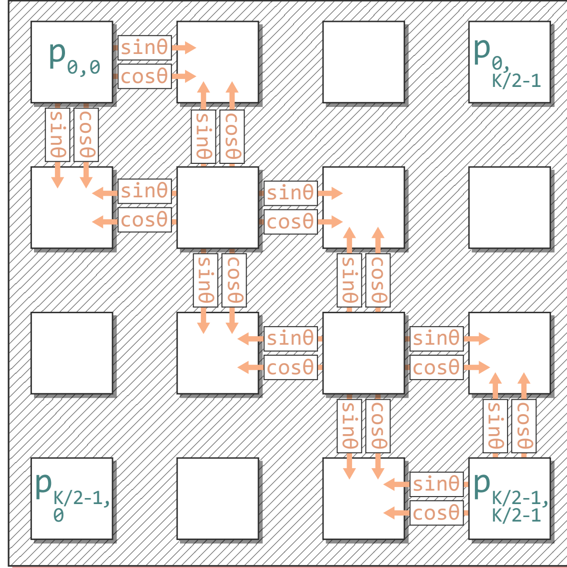



**Figure 4.9:** *Detail of the $sin(\theta)$ and $cos(\theta)$ propagation between Compute Units in the Jacobi Systolic Array architecture, for $K = 4$.*

putation through a SA formulation and performing rotations concurrently, we decrease the number of iterations for convergence to $\mathcal{O}(log(K))$. Rotations, equivalent to multiplications on $2 \times 2$ submatrices, are unrolled and performed in constant time. Instead, a traditional CPU implementation has a cost of $\mathcal{O}(\log(K) \cdot K^{2.3728596})$, if employing the matrix multiplication algorithm by Alman and Williams [7], which is the currently-known matrix multiplication algorithm with the lowest complexity, equal to $\mathcal{O}(K^{2.3728596})$.

Our design for the Jacobi algorithm is optimized to compute up to $K$ eigenvalues. While it can compute a lower amount of eigenvalues without a reconfiguration, we place in the same FPGA bitstream multiple *Jacobi cores* optimized for specific $K$ (4, 8, 16, etc.). We can configure both SLR1 and SLR2 with Jacobi cores to fully utilize the FPGA resources and opening the doors for independent optimization on specific values of $K$ by reconfiguring individual SLRs. The *Lanczos Core* on SLR transfers only the $3K - 2$ values of $\mathcal{T}$ to the Jacobi cores on SLR1 and SLR2. We prevent inefficiencies in inter-SLR communication by moving data through PLRAM, while also avoiding the long read-write latency of DDR and HBM.

In practice, the systolic formulation cannot scale beyond very small matrices ($K \approx 32$) due to the large number of resources required for trigonometric operations in each CU. While resource utilization has prevented

widespread adoption of the Jacobi algorithm for general eigenproblem resolution, it is not a limitation for our use case, as we apply the Jacobi eigenvalue algorithm on small $K \times K$ inputs by design.

On CPU, approaches such as QR factorization are more common [31], because efficient systolic array formulations of the Jacobi algorithm require full control over cache eviction policies. Moreover, even modern CPUs lack enough floating-point arithmetic units to perform the operations required for an iteration at once: even for a small $K$ such as $K = 8$, the Jacobi algorithm computes 16 trigonometric operations and about $800$ floating-point multiplications per iteration.

Instead, we leverage the abundant hardware resources of our FPGA platform to perform all these operations concurrently, making it the optimal choice for our Jacobi SA design.

#### Diagonal And Offdiagonal CU

Diagonal CU (Algorithm 7, line 4) annihilate elements immediately outside the diagonal via a matrix rotation. Although the rotation angle is arbitrary, the fastest convergence is achieved by setting $\theta = \frac{1}{2}\arctan\frac{2\beta}{\alpha-\delta}$, which eliminates the $\beta$ and $\gamma$ components (Figure 4.4a).

Computation of $\theta$ is performed via a Coordinate Rotation Digital Computer (CORDIC) core; since the argument of $\arctan$ is unbounded, less resource-intensive methods like Taylor Series Approximation would lead to inaccurate results as the magnitude of the input increases. On the other hand, as $\theta$ itself is $\in [-\frac{\pi}{4}, \frac{\pi}{4}]$, we can leverage Taylor series expansion to compute the components of the rotation matrix efficiently. Even an order-3 approximation provides excellent accuracy ($\sim 10^{-6}$ at $\pm\frac{\pi}{4}$), using significantly fewer Digital Signal Processors (DSPs) and Block RAMs (BRAMs) than the CORDIC core. Rotation on the diagonal (Figure 4.5 Ⓐ) are performed by $K/2$ parallel cores, propagating rotation values Ⓑ in constant time to the Offdiagonal CU (Algorithm 7, line 9).

This propagation step is more clearly visualized in Figure 4.9, for a computation with $K = 4$. As each CUs holds only 4 elements, matrix multiplications are fully unrolled and performed in constant time. Eigenvectors (Algorithm 7, line 14) Ⓓ are computed in parallel to the rotation of the Offdiagonal CU Ⓒ as they only require rotation values.

#### Row/Column Interchange

Each CU has 8 connections to propagate input and output values of $\alpha, \beta, \gamma, \delta$ values to adjacent processors, in addition to communicating the rotation

**Table 4.1:** *Resource usage and clock frequency in our FPGA hardware design, divided by algorithm. We also report the total amount of resources available to the FPGA.*

| Algorithm | SLR | LUT | FF | BRAM | URAM | DSP | Clock (MHz) |
|---|---|---|---|---|---|---|---|
| **Lanczos** | SLR0 | 42% | 13% | 15% | 0% | 16% | 225 |
| **Jacobi** | SLR1 | 40% | 42% | 0% | 0% | 68% | 225 |
| **Jacobi** | SLR2 | 15% | 17% | 0% | 0% | 34% | 225 |
| **Available** | | 1097419 | 2180971 | 1812 | 960 | 9020 | |

value $\theta$. As shown in Figure 4.5E, each processor $p_{i,j}$ with $i$ and $j \neq (1, K/2)$ propagates its $\alpha$ and $\gamma$ values to the $\beta$ and $\delta$ slots of $p_{i,j+1}$ and its $\beta$ and $\delta$ values to the $\alpha$ and $\gamma$ slots of $p_{i,j-1}$.

Processors in the first column ($p_{i,1}$) propagate $\beta$ and $\delta$ to to the $\alpha$ and $\gamma$ slots of $p_{i,2}$. Processors $p_{i,K/2}$ propagate $\beta$ and $\delta$ to their own $\alpha$ and $\gamma$ slots. Operations for the column interchange are symmetrical. As $\alpha$ and $\gamma$ of $p_{i,1}$ are never propagated, more swaps are performed towards lower indices than higher indices. These additional swaps require $K$ temporary vectors to store rows that the swaps would overwrite. To avoid wasting resources for these temporary vectors, we execute operations in *reverse*, from $K/2$ to $1$. As row/column swaps do not introduce data dependencies, we perform them in a single clock cycle with Flip Flops (FFs).

## 4.5 Experimental Evaluation

To prove that our custom FPGA design is suitable for solving large-scale Top-K sparse eigenproblems, we compare it against the popular ARPACK library, measuring how it compares in terms of execution time, power efficiency, and accuracy. The ARPACK library [117] is a multi-threaded Top-K sparse eigensolver that employs IRAM, and is considered the standard reference both in terms of performance and numerical accuracy. We run ARPACK run on two Intel Xeon Gold 6248 (80 threads in total) and $384\,\mathrm{GB}$ of DRAM using single-precision floating-point arithmetic. Our eigensolver is prototyped on a Xilinx Alveo U280 accelerator card equipped with $8\,\mathrm{GB}$ of HBM2 memory, $32\,\mathrm{GB}$ of Double Data Rate 4 (DDR4) memory, and a `xcu280-fsvh2892-2L-e` FPGA whose resources are reported in Table 4.1, along with the resource utilization of our hardware designs. Results are averaged over 20 runs.

Tests are carried out using a collection of large sparse matrices repre-

**Table 4.2:** *Matrices/graphs in the evaluation, sorted by number of edges/non-zero entries (in millions). For each matrix, we report the memory footprint when stored as COO.*

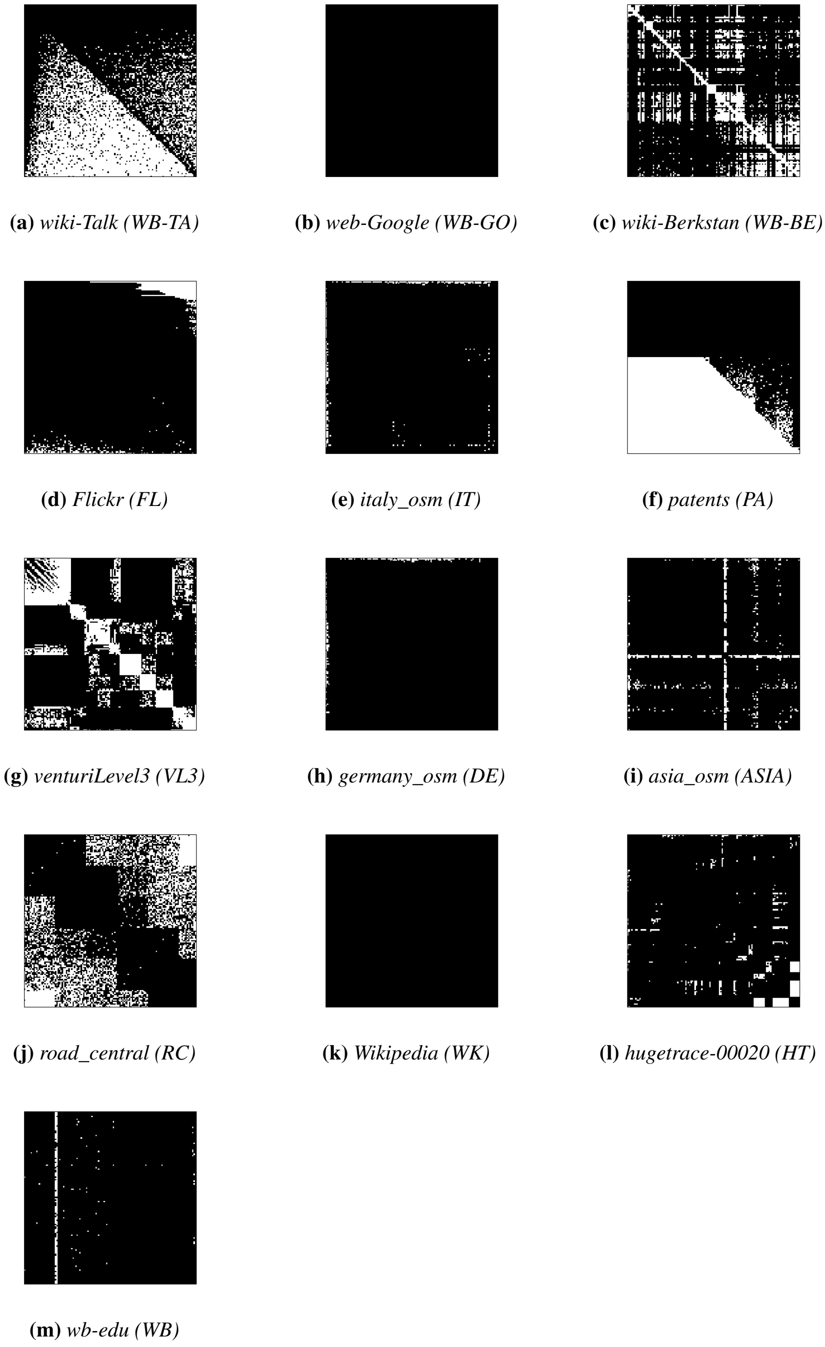
| ID | Name | Rows (M) | Non-zeros (M) | Sparsity (%) | Size (GB) |
|---|---|---|---|---|---|
| **WB-TA** | wiki-Talk | 2.39 | 5.02 | $8.79 \times 10^{-4}$ | 0.06 GB |
| **WB-GO** | web-Google | 0.91 | 5.11 | $6.17 \times 10^{-4}$ | 0.07 GB |
| **WB-BE** | web-Berkstan | 0.69 | 7.60 | $1.60 \times 10^{-3}$ | 0.10 GB |
| **FL** | Flickr | 0.82 | 9.84 | $1.46 \times 10^{-3}$ | 0.13 GB |
| **IT** | italy_osm | 6.69 | 14.02 | $3.13 \times 10^{-5}$ | 0.18 GB |
| **PA** | patents | 3.77 | 14.97 | $1.05 \times 10^{-4}$ | 0.19 GB |
| **VL3** | venturiLevel3 | 4.02 | 16.10 | $9.96 \times 10^{-5}$ | 0.21 GB |
| **DE** | germany_osm | 11.54 | 24.73 | $1.86 \times 10^{-5}$ | 0.32 GB |
| **ASIA** | asia_osm | 11.95 | 25.42 | $1.78 \times 10^{-5}$ | 0.33 GB |
| **RC** | road_central | 14.08 | 33.87 | $1.71 \times 10^{-5}$ | 0.43 GB |
| **WK** | Wikipedia | 3.56 | 45.00 | $3.55 \times 10^{-4}$ | 0.60 GB |
| **HT** | hugetrace-00020 | 16.00 | 47.80 | $1.87 \times 10^{-5}$ | 0.61 GB |
| **WB** | wb-edu | 9.84 | 57.15 | $5.90 \times 10^{-5}$ | 0.73 GB |

senting graph topologies, each containing millions of rows and non-zero entries (Table 4.2). All test matrices come from the SuiteSparse collection [43], and a graphical representation for each matrix, as reported by the respective authors, is reported in Figure 4.10. While our evaluation is focused on sparse matrices representing graphs, our Top-K sparse eigenproblem FPGA design is applicable to other domains such as image analysis [161, 181, 192].

Resource utilization and clock frequency of our design are reported in Table 4.1. The Lanczos algorithm and Jacobi algorithm have similar utilization, with around 20 % LUT utilization each (50 % of the available LUTs in each SLR). Although the SA architecture of the Jacobi algorithm processes small $K \times K$ inputs, it requires the computation of many trigonometric operations and multiplications (16 and $> 800$ for $K = 8$) in each iteration. Resource utilization of the Jacobi algorithm scales quadratically with the number of eigenvalues $K$, while the Lanczos algorithm is not affected.

### 4.5.1 Execution Time

We measure the execution time speedup of the FPGA-based hardware design implementing our Top-K sparse eigenproblem solver against the CPU baseline and report results in Figure 4.11. To better visualize results, we distinguish between small-scale (Figure 4.11a) and large-scale matrices



**(a)** *wiki-Talk (WB-TA)* **(b)** *web-Google (WB-GO)* **(c)** *wiki-Berkstan (WB-BE)*

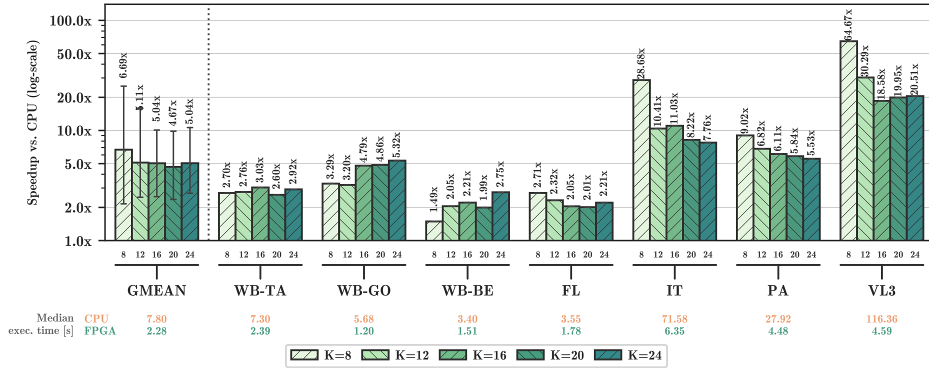
**(d)** *Flickr (FL)* **(e)** *italy_osm (IT)* **(f)** *patents (PA)*

**(g)** *venturiLevel3 (VL3)* **(h)** *germany_osm (DE)* **(i)** *asia_osm (ASIA)*

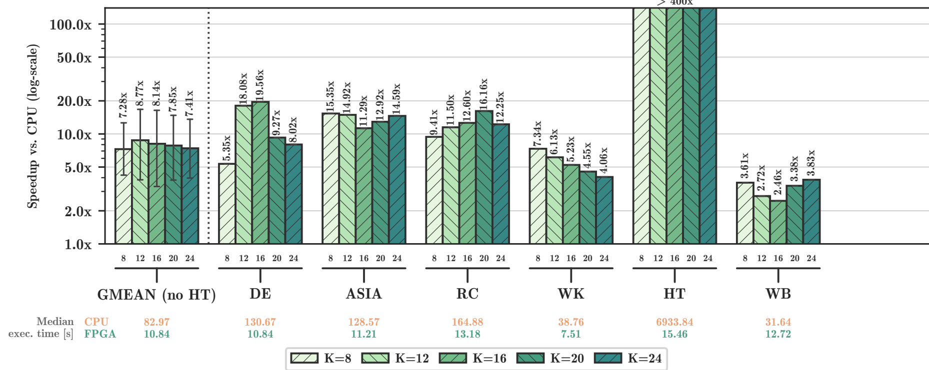
**(j)** *road_central (RC)* **(k)** *Wikipedia (WK)* **(l)** *hugetrace-00020 (HT)*

**(m)** *wb-edu (WB)*

**Figure 4.10:** *Graphical representation of the sparsity patterns of matrix used in the evaluation, as reported by the respective authors on the SuiteSparse collection [43].*



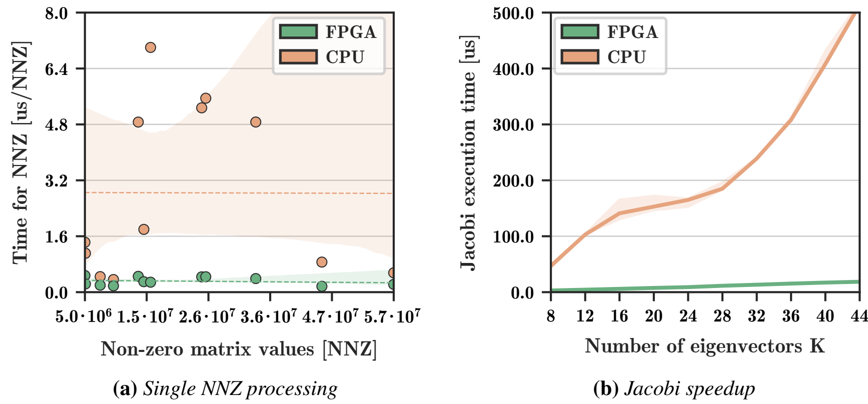

**(a)** *Speedup of our Top-K sparse eigensolver vs. the CPU baseline, on small-scale matrices.*




**(b)** *Speedup of our Top-K sparse eigensolver vs. the CPU baseline, on large-scale matrices.*

**Figure 4.11:** *Speedup (higher is better) of our Top-K sparse eigensolver vs. the ARPACK multi-core CPU library, for increasing number of eigenvalues $K$. Matrices are ordered by size, divided into small matrices ⓐ and large matrices ⓑ. In ⓑ, geomean excludes the outlier matrix HT, where the speedup of our FPGA design exceeds 400x.*

(Figure 4.11b). Despite the difference in size, the two categories present similar average speedups, showing how the performance of our hardware design is not negatively affected by the matrix size. We are always faster than the baseline, with a geometric mean speedup of 6.22×, up to 64× for specific matrices. Figure 4.11 visualizes the average speedup divided by $K$, the number of eigencomponents computed by the algorithm. The speedup is mostly unaffected by $K$, showing how our design can efficiently compute many eigenvalues at once. To understand how different the behavior of the CPU implementation and of the FPGA hardware design are, we inspect in Figure 4.12a the time required by each hardware to process a single matrix value. In the plot, each circle represents a matrix, with the number of non-




**(a)** *Single NNZ processing* **(b)** *Jacobi speedup*

**Figure 4.12:** ⓐ *Relation between the number of matrix non-zero values and time to process a single value. Each circle is a matrix, and we show a linear regression with the 95 % confidence interval bands.* ⓑ *Speedup vs. CPU of our Systolic Array architecture for the Jacobi algorithm, for increasing number of eigenvalues $K$.*

zero entries on the x-axis and the execution time of the CPU and FPGA executions (as reported in Figure 4.11) divided by the number of non-zeros on the y-axis. We compute a linear regression on these values, and report the 95 % confidence interval. The time required by our FPGA design to process a single matrix value is unaffected by the overall matrix size, while the CPU behavior is drastically more unpredictable.

We estimate that the Lanczos dominates the overall execution time due to the SpMV computations, taking more than 99 % of the execution time. However, optimizing the Jacobi algorithm with a SA design is still worth the effort compared to running this step on CPU. Figure 4.12b shows the speedup of our Jacobi SA design compared to an optimized C++ CPU implementation, for increasing number of eigencomponents $K$: the execution time on CPU grows quadratically due to repeated matrix multiplications, becoming a non-negligible part of the execution time for large $K$. On the other hand, the FPGA implementation's scaling is coherent with the $\mathcal{O}(log(k))$ complexity of our SA architecture in the Jacobi algorithm.

Our hardware design was synthesized at 225 MHz on the Alveo U280 accelerator card. A clock frequency beyond 225 MHz does not significantly improve performance as SpMV represents the main computational bottleneck in the computation, and its performance is bound by HBM bandwidth [126]. Each SpMV CU processes data at the maximum bandwidth offered by the HBM channel from which it reads the matrix ($14.37\,\mathrm{GB/s}$, for a total of $71.87\,\mathrm{GB/s}$ using 5 CU).

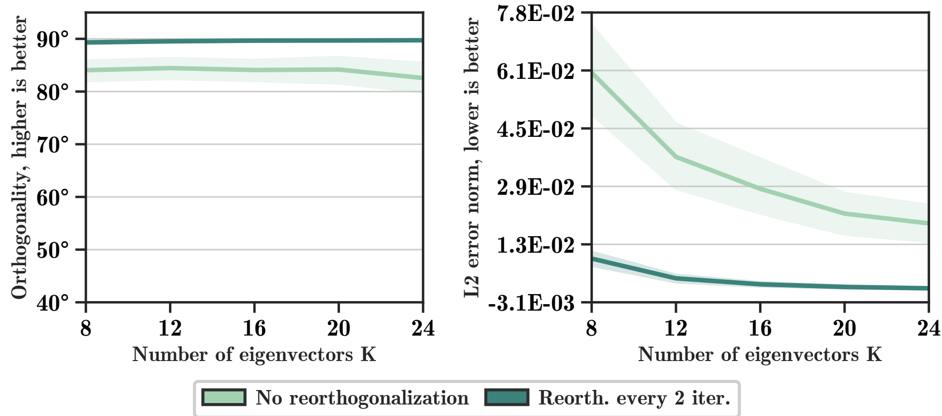



**Figure 4.13:** *Accuracy of our Top-K sparse eigensolver, in terms of orthogonality and reconstruction error, for increasing $K$. Applying reorthogonalization guarantees almost-perfect results, although the implementation without reorthogonalization can still be useful in approximate ML applications.*

We have not been able to compare the performance of our hardware design against a reference GPU implementation, as no publicly available GPU implementation of the Lanczos algorithm exists, to the best of our knowledge. Still, we can make some consideration about the performance of such implementation. From our later analysis in Section 5.5 and Figure 5.9 ⓑ, we observe how the reference SpMV GPU implementation in cuSPARSE operates at around 55 % of the peak memory bandwidth of the GPU, i.e. $250\,\mathrm{GB/s}$ out of $450\,\mathrm{GB/s}$. Given that our SpMV hardware design is limited to $72\,\mathrm{GB/s}$ (Section 4.4.2) by the FPGA HBM memory controller, we expect the Lanczos algorithm on GPU to be no more than $3.5\times$ faster than our design. In practice, the GPU speedup is likely to be lower. Using reduced-precision arithmetic results in higher operational intensity (Figure 5.9), and the Jacobi algorithm must be executed on the CPU instead of GPU (Section 4.2) as the input size is too small to utilize the GPU parallelism fully [38]. Even assuming a GPU implementation $3.5\times$ faster than our design, we would still have a better Performance/Watt ratio, as our design requires around 35 Watts versus the 250 Watts of a GPU.

### 4.5.2 Power Efficiency

We measured via an external power meter that our FPGA design consumes about 38W during execution, plus 40 Watts for the host server. The CPU implementation consumes around 300 Watts during execution. Our FPGA

design provides 49× higher Performance/Watt ratio (24× if accounting for the FPGA host machine): we provide higher performance without sacrificing power efficiency, making our design suitable for repeated computations typical of data center applications.

### 4.5.3 Accuracy Analysis of the Approximate Eigencomputation

The Lanczos algorithm is known to suffer from numerical instability [153]. To limit this phenomenon, we reorganize the algorithm's operations as in [153] and apply reorthogonalization as in [154]. To assess the stability of our design, we measure the eigenvectors' pairwise orthogonality and the eigenvector error norm. Eigenvectors must form an orthonormal basis and be pairwise orthogonal, i.e. their dot product is 0, or equivalently their angle is $\pi/2$. For each pair of eigenvectors, we measure the average angle that occurs. Then, if $\lambda$ is an eigenvalue of $\mathrm{M}$ and $v$ is its associated eigenvector, it must hold $\mathrm{M}v = \lambda v$. By measuring the L2 norm of $\mathrm{M}v - \lambda v$ for all $v$ we evaluate how precise the eigenvector computation is. Results of orthogonality angle and L2 error norm, with and without reorthogonalization, for increasing $K$, aggregated on all matrices, are reported in Figure 4.13. Accuracy is excellent if reorthogonalization is applied every two iterations, but even without this procedure results are satisfactory. Despite using fixed-precision arithmetic in the Lanczos algorithm, the average reconstruction error is below $10^{-3}$, and the average orthogonality is $> 89.9$ degrees when applying reorthogonalization every two iterations. Orthogonality is not affected by $K$, while the average reconstruction error improves as $K$ increases. Spectral methods in machine learning applications use eigenvectors to capture the most important features of their input and do not usually require the same degree of precision as other engineering applications. Reorthogonalization adds an overhead up to $\mathcal{O}(nK^2/2)$ to the algorithm compared to Figure 4.11. On large matrices, this overhead is negligible compared to SpMV, and is a viable option in applications where maximum accuracy is necessary. Still, our hardware design can provide excellent accuracy while being significantly faster than a highly optimized CPU implementation.

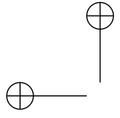
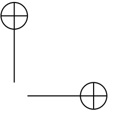
## 4.6 Final Remarks

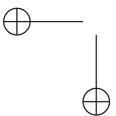
The computation of the Top-K eigenvalues and eigenvectors on large graphs represented as sparse matrices is critical in spectral methods, a class of powerful Machine Learning algorithms that can extract useful features from graphs. We solved the Top-K sparse eigenproblem with a **new algorithm**

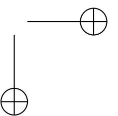
that is **optimized for reconfigurable hardware designs**. In the first part of the computation, **we exploited** the enormous bandwidth of **HBM** through the Lanczos algorithm, showing how the SpMV core presented in Chapter 3 can be extended to support matrices partitioned on multiple HBM pseudo channels. The second part introduced a **systolic array architecture** that efficiently parallelizes the compute-intensive Jacobi eigenvalue algorithm. Compared to the popular **ARPACK CPU library**, our hardware design achieves a geometric mean speedup of 6.22$\times$ on 13 graph topologies with millions of vertices, **raising the bar for high-performance Top-K sparse eigensolvers** at a large scale.

As future research directions to build upon this work, we will extend our hardware design to support non-Hermitian matrices through the Implicitly Restarted Arnoldi Method. Scalability to multiple FPGAs is interesting but not easy to achieve due to the iterative nature of the Lanczos algorithm. One would have to partition SpMV over multiple devices, with each device operating on a subset of rows. Aggregating results of each iteration requires communication through Peripheral Component Interconnect Express (PCIe), and is a major bottleneck in the computation. In this chapter, we also realized how the limitations of current HBM memory controllers prevent us from scaling to an arbitrarily large number of CUs. In some computations, such as the Top-K Sparse matrix-vector multiplication (Top-K SpMV) presented in the next chapter, we can overcome this limit through custom algorithmic approximations, achieving great performance with minimal quality degradation of results.

Most importantly, we believe that this work is a perfect case study for the optimization of applications that benefit from input-aware FPGA reconfiguration. Indeed, when analyzing the complexity of the Lanczos algorithm, we observe how matrices and graphs with relatively low sparsity ($|E| \gg |V|$) benefit from more SpMV CUs, while highly sparse matrices ($|E| \approx |V|$) benefit from a larger partitioning factor in linear operations (e.g. vector normalization), with possibly a single SpMV CU. Although the hardware design presented in this chapter is a solid compromise between these two extremes, it is certainly interesting to evaluate the performance trade-offs on additional matrices and extend this analysis to other algorithms with similar characteristics.





CHAPTER 5

# Scaling up HBM Efficiency of Top-K SpMV for Approximate Embedding Similarity on FPGAs

*Top-K SpMV is a key component of similarity-search on sparse embeddings. This sparse workload does not perform well on general-purpose NUMA systems that employ traditional caching strategies. Instead, modern FPGA accelerator cards have a few tricks up their sleeve. In the chapter, we extend our SpMV FPGA hardware design to support Top-K SpMV. Our new Top-K SpMV hardware design leverages reduced precision and a novel packet-wise CSR matrix compression called BSCSR, enabling custom data layouts and delivering bandwidth efficiency often unreachable even in architectures with higher peak bandwidth. With HBM-based boards, we are 138x faster than a multi-threaded CPU implementation and 2.1x faster than a GPU with 20 % higher bandwidth, with 15x higher power-efficiency, proving that FPGAs are today the optimal solution for Top-K SpMV.*

## 5.1 Introduction

Information Retrieval (IR) and recommender systems process an always-increasing amount of data, often with strong real-time constraints, to suggest products, movies, news articles to billions of users. Top-K Sparse matrix-vector multiplication (Top-K SpMV) is a key building block of **high-performance similarity-search applications** found in recommender systems that store items or documents as sparse *embeddings*, short numerical representations usually obtained through a neural network [16, 19]. Sparse embeddings guarantee low memory footprint and reduced execution time, often without decreasing accuracy, as they capture important information while filtering out noise [22]. Top-K SpMV computes the highest K values of the product between a sparse matrix (a matrix where only non-zero entries are stored) and a dense vector. In our case, it matches an input embedding against a collection of sparse embeddings and finds the K most similar ones (Figure 5.1).

The computation of Top-K SpMV is extremely **memory intensive** and presents sequential, indirect, and fully random memory accesses, making it unsuitable for general-purpose architectures with traditional caching policies. Improving its performance through a custom hardware design demands a **theoretically sound approach** to optimize the **data-access strategy**. Field-Programmable Gate Arrays (FPGAs) can leverage application-specific data representations thanks to reduced-precision arithmetic and a customizable memory subsystem. In agreement with the Roofline methodology [184], this approach is key to improving operational intensity and performance in memory-starved computations. Exact numerical accuracy for large values of K is not critical, as long as the most similar embeddings are retrieved: compared to CPUs or Graphics Processing Units (GPUs), FPGAs can leverage optimized arbitrary-precision arithmetic and offer fine-grained control over the desired target accuracy, providing better performance, lower resource utilization, and lower power consumption.

To utilize off-chip bandwidth fully, we devise an **approximation scheme based on matrix partitioning** that enables a flexible multi-core design. Each core uses a single High Bandwidth Memory (HBM) channel and UltraRAM (URAM) for caching high-reuse or random access data (Section 5.3.1). The introduction of HBM to FPGA accelerator cards provides new opportunities to achieve performance beyond traditional Double Data Rate (DDR) memory: on an Alveo U280 accelerator card, our design accesses 32 HBM2 pseudo channels with a total bandwidth of $460\,\mathrm{GB/s}$, compared to the meager $38\,\mathrm{GB/s}$ offered by a dual-channel DDR4.

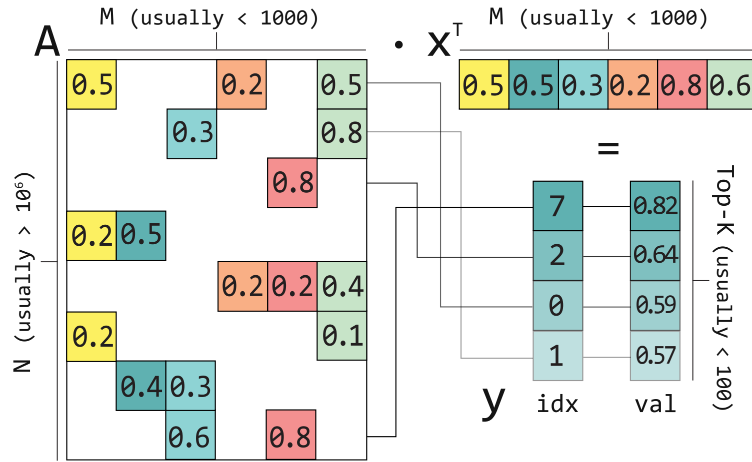



**Figure 5.1:** *Top-K SpMV between a sparse matrix* A *(in our case, a collection of sparse embeddings) and a dense vector* x *(a dense embedding), with notation as in Section 5.3.*

Then, we maximize bandwidth efficiency through a **novel streaming sparse matrix format** called Block-Streaming CSR (BS-CSR). This format leverages coalesced HBM accesses and reduced numerical precision, and is oblivious to the matrix non-zero entries distribution (Section 5.3.2). We extend one more time our streaming SpMV FPGA design to include these improvements and approximate the computation of Top-K SpMV[1]. We present the following contributions:

- A new **approximation scheme** for Top-K SpMV that allows splitting the computation over an arbitrary number of independent partitions, with a negligible accuracy cost even when processing matrices with millions of rows (Section 5.3.1).
- A novel packet-wise **sparse matrix data-layout called BS-CSR**. Thanks to arbitrary precision data-types, BS-CSR increases the operational intensity of the computation by a factor of 3× (Section 5.3.2).
- A multi-core Top-K SpMV FPGA design that leverages our new algorithmic approximations and sparse matrix data-layout to effectively exploit the available HBM bandwidth (Section 5.4).
- A performance evaluation of our FPGA design, showing 138× faster execution versus a state-of-the-art CPU implementation. We are 2.1× faster than a GPU with 20 % higher bandwidth, with 15× higher power-efficiency and no significant loss of Top-K precision (Section 5.5).

[1] Code is available at `github.com/necst/approximate-spmv-topk`

## 5.2 Related Work

To the best of our knowledge, no prior work optimizes the computation of Top-K SpMV on FPGA or GPU, although existing research covers approximation techniques on FPGA for dense matrix multiplications and deep learning [3,206]. On CPUs, `sparse_dot_topn` is a C++ multi-threaded implementation that employs the Compressed Sparse Row (CSR) format and is specialized for sparse embeddings and documents similarity [19]. That said, the CPU performance in this computation is inherently held back by their limited memory bandwidth and the inability to consistently perform quick random accesses, as there are no guarantees that requested values have not been evicted from cache. Approximate similarity-search on embedding has also seen many alternative algorithms based on approximate K-Nearest Neighbors (KNN) on dense embeddings, often with GPU and sometimes FPGA implementations [40, 87, 95]. These techniques usually employ pre-built graph indices [58] or Locality-Sensitive Hashing (LSH) to quickly retrieve similar items [127,227]. Sparse Top-K SpMV can either be a sub-step of these algorithms (to further filter a set of similar embeddings), or as an alternative pre-selection search technique, where the Top-K embeddings retrieved by Top-K SpMV are processed with other downstream techniques (for example, by re-computing the similarity on their original dense representation).

Although no existing work specifically optimizes Top-K SpMV on GPU, there exist multiple highly-optimized implementations of Sparse Matrix-Vector Multiplication (SpMV) [124, 125, 139, 186]. While modern GPUs provide higher memory bandwidth than even high-end FPGA accelerator cards (from $549\,\mathrm{GB/s}$ of the Nvidia P100 to the $1555\,\mathrm{GB/s}$ of the Nvidia A100), existing SpMV implementations are often unable to utilize the available bandwidth fully [143]. Optimizations for reduced precision are currently limited to Half-Precision (HP) floating-point. [2,79]. The lack of support for reduced-precision fixed-point arithmetic limits their maximum operational intensity and precise control over numerical accuracy (Section 5.5.3), although recent work explores mixed single and double-precision floating-point arithmetic [5]. Reduced-precision arithmetic has not yet been thoroughly studied in the context of SpMV, but encouraging results have been obtained in numerical analysis [9, 187] and deep-learning [80, 121, 138, 201].

Existing research on SpMV acceleration with FPGAs focuses mostly on sparse matrix compression [68], moving the bottleneck from memory accesses to the input data decompression. Fowers et al. [57] already proposed

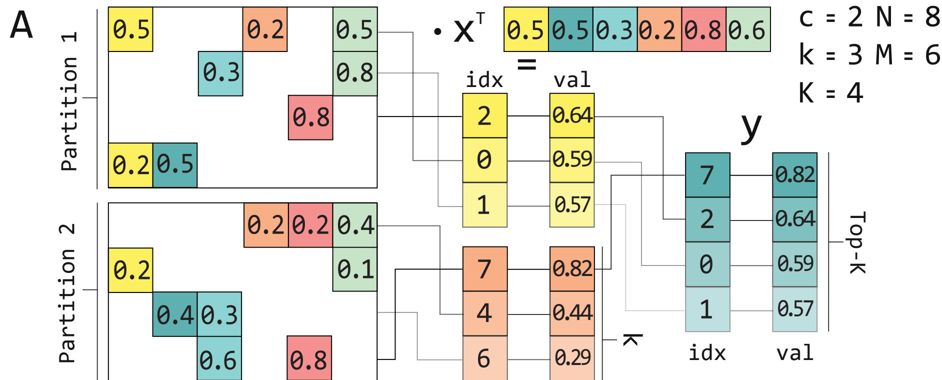



**Figure 5.2:** *Simplified approximation scheme for Top-K SpMV. No errors occur if all partitions have less than* k *Top-K values.*

a sparse matrix encoding to alleviate the inefficiencies of the FPGA DDR memory subsystem. Umuroglu et al. [195], instead, maximized the time in which values are kept in a fast local cache hierarchy. More recently, Sadi et al. [172] propose an SpMV FPGA implementation that achieves significant speedup leveraging HBM and a data-compression scheme to reduce off-chip traffic. The efficient use of HBM on FPGA also has received the academic community's attention, with Kara et al. [100] showing how HBM can provide an order of magnitude speedup over CPU on a variety of data-intensive workloads, and Wang et al. [205] offering a complete characterization of the HBM subsystem in modern FPGA accelerator cards. However, directly employing off-the-shelf SpMV FPGA designs, similar to the ones presented in the previous chapters (Sections 3.4.2 and 4.4.2 [158, 177]), is undesirable: it would waste memory to store for the entire output vector (which can contain millions of values, Section 5.3) and demand unnecessary computation to sort the output afterward.

## 5.3 Theoretical Contributions

Two key theoretical contributions enable our FPGA hardware design. First, we introduce a partitioning scheme that splits and approximates Top-K SpMV on independent cores, giving access to the entire HBM bandwidth (Section 5.3.1). Then, we present BS-CSR, a sparse matrix format optimized for streaming coalesced memory transactions and reduced-precision arithmetic (Section 5.3.2).

Given a sparse matrix $A \in \mathbb{R}^{N \times M}$, and a dense vector $x \in \mathbb{R}^{M \times 1}$, the result of the SpMV $y = Ax$ is a vector $y \in \mathbb{R}^{N \times 1}$. Top-K SpMV does

not track the entire $y$, but only its largest K values (and the corresponding indices). Figure 5.1 shows a small example of Top-K SpMV, with a sparse matrix $A$ being compared against a dense vector $x$ to retrieve the $K = 4$ most-similar rows in $A$ (each representing a sparse embeddings).

Formally, given $A \in \mathbb{R}^{N \times M}$ and $x \in \mathbb{R}^{M \times 1}$, Top-K SpMV outputs a vector $y \in (\mathbb{N} \times \mathbb{R})^{K \times 1}$, composed of tuples $\bar{y} : (\bar{y}^i, \bar{y}^v) \in \mathbb{N} \times \mathbb{R}$ (the index and value, respectively), such that

$$\forall \bar{y}_j \in y, y_j \in y : \bar{y}^v_j \geq y_j \tag{5.1}$$

If $A$ and $x$ are $L2$-normalized embeddings, Top-K SpMV retrieves the rows of $A$ with highest cosine similarity to $x$. In our task, $N$ is the size of the embedding collection (usually millions of entries), while $M$, the dense embedding size, has only a few hundred values [223]. $x$ is stored in URAM to perform random accesses in a single clock cycle, which is infeasible with an unconstrained $M$. Similarly, we track just the Top-K entries of $y$ on a Look-Up Table (LUT) scratchpad instead of performing data-dependent write-backs on HBM and sharing bandwidth with read-channels, which would undermine the maximum memory throughput. $x$ is dense as many sparsification algorithms operate on dense matrices (e.g. batches of $A$) and cannot efficiently sparsify a single vector [4, 129, 191].

### 5.3.1 Top-K SPMV Approximation

In Top-K similarity-search applications, it is often acceptable to trade maximum accuracy on large values of $K$ for lower execution time and power consumption. Instead of computing the total Top-K values on the matrix **A**, we partition it in $c$ sub-matrices with $N/c$ rows each. Each sub-matrix is processed by an independent FPGA core, which computes the top $k < K$ results for the partition (with $k \cdot c \geq K$). We obtain $k \cdot c$ results, which represents an approximation of the original Top-K. Figure 5.2 shows the approximation scheme in action. First, we split $A$ in $c = 2$ sub-matrices. In the example, each partition only has four rows/embeddings, but in a real application, each sub-matrix would still have millions of rows. Then, we compare $x$ against each sub-matrix to retrieve $k = 3$ partial results. Finally, we quickly merge the partial results to obtain the approximate Top-K, with $K = 4$ as in Figure 5.1. In this example, the result is exact, with no approximation being visible in the final result.

As the number of sub-matrices $c$ increases, so does the approximation accuracy. We always retrieve the top $k$ values, and as a consequence, the approximation does not affect the best-ranked rows.

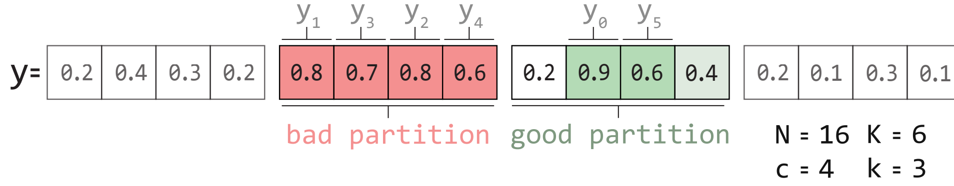



**Figure 5.3:** *The vector* $\mathrm{y}$ *is divided into* $\mathrm{c} = 4$ *partitions. In each, we retrieve the* $\mathrm{k} = 3$ *highest values. As the second partition contains* $\mathrm{k}' = 4 > \mathrm{k}$ *Top-K values, it is flagged as a bad partition, and the approximation scheme commits an error. If for a given disposition of Top-K values we do not encounter any bad partition, we successfully retrieve all Top-K values.*

**Theorem 1.** The expected value of correctly retrieved Top-K values (i.e. the *precision*) is expressed in closed form as

$$\mathbb{E}[P] \approx \frac{1}{K} \sum_{K_i=1}^{K} 1 - \frac{\mathrm{c} \cdot \sum_{\mathrm{k}_i=\mathrm{k}+1}^{min(K_i, \lfloor N/\mathrm{c} \rfloor)} \binom{\lfloor N/\mathrm{c} \rfloor}{\mathrm{k}_i}}{\binom{N}{K_i}} \tag{5.2}$$

*Proof.* We can model the problem of finding the Top-K elements as having a vector $\mathrm{y} \in \mathbb{R}^{N \times 1}$, and finding the indices $\bar{y}_i$ for $i \in [0, K)$ of the highest K values. The vector $\mathrm{y}$ is the output of the SpMV $\mathrm{Ax}$, but the computation of $\mathrm{y}$ is not relevant for the proof: we do not perform any approximation to produce the elements $y_i$ of $\mathrm{y}$.

We assume that Top-K values are uniformly distributed over $\mathrm{y}$. The relative order of those Top-K does not matter, as we only care if a certain $y_i$ belongs to the Top-K, i.e. if $(i, y_i) \in \mathrm{y}$, following Equation (5.1).

The Top-K elements can arrange in $\binom{N}{K}$ different disposition. Following our approximation scheme, we divide $\mathrm{y}$ in $\mathrm{c}$ partitions, each of size $\lfloor N/\mathrm{c} \rfloor$. In each partition, we select the $\mathrm{k} < K$ highest ranked values. We refer to a generic partition as $\hat{c}$, while $\mathrm{c}$ is the number of partitions. We have an error in partition $\hat{c}$ if partition $\hat{c}$ contains $\mathrm{k} < \mathrm{k}' \leq K$ Top-K elements.

**Definition 5.1.** A *bad partition* is a partition $\hat{c}$ that contains $\mathrm{k} < \mathrm{k}' \leq K$ Top-K elements. A *bad disposition* for $\hat{c}$ is a disposition of the Top-K elements such that $\hat{c}$ is a bad partition. A *good partition* is a partition that is not bad. A *good disposition* is a disposition of the Top-K elements such that no partition is bad (i.e. we correctly retrieve all Top-K values). An example of good and bad partitions is given in Figure 5.3.

A bad dispositions has at least $\mathrm{k} + 1$ Top-K values in it, and a partition can have at most $min(K, \lfloor N/\mathrm{c} \rfloor)$ Top-K values in it. The number of dis-

**Table 5.1:** *Estimated precision of Top-K indices for increasing number of partitions. We average the results of 1000 tests.*

| **Number of matrix rows** | **Number of partitions** | **Top-K Value,** with $k = 8$ | | | | | |
|---|---|---|---|---|---|---|---|
| | | **8** | **16** | **32** | **50** | **75** | **100** |
| $N = 10^6$ | $c = 16$ | 1 | 1 | 0.999 | 0.998 | 0.983 | 0.942 |
| | $c = 28$ | 1 | 1 | 1 | 0.999 | 0.999 | 0.996 |
| | $c = 32$ | 1 | 1 | 1 | 0.999 | 0.999 | 0.997 |
| $N = 10^7$ | $c = 16$ | 1 | 1 | 1 | 0.999 | 0.986 | 0.947 |
| | $c = 28$ | 1 | 1 | 1 | 0.999 | 0.999 | 0.995 |
| | $c = 32$ | 1 | 1 | 1 | 0.999 | 0.998 | 0.998 |

positions with $k < k' \leq K$ Top-K values for a given partition is $\binom{\lfloor N/c \rfloor}{k'}$. As such, we count the sum of bad dispositions for all $k'$ from $k + 1$ to $min(K, \lfloor N/c \rfloor)$, obtaining the number of bad dispositions for a partition $\hat{c}$.

$$\gamma(K) = \sum_{k_i = k+1}^{min(K, \lfloor N/c \rfloor)} \binom{\lfloor N/c \rfloor}{k_i} \tag{5.3}$$

As we have c partitions, we obtain a total of $c \cdot \delta(K)$ bad dispositions. The fraction of bad dispositions over the total number of possible Top-K dispositions is $c \cdot \gamma(K)/\binom{N}{K}$, so the fraction of good dispositions is

$$\delta(K) = 1 - \frac{c \cdot \gamma(K)}{\binom{N}{K}} \tag{5.4}$$

This fraction can be interpreted as the probability that any given disposition is a good disposition. The value $\delta(K)$ gives the probability of a good disposition for a precise value of K, i.e. the probability of not making mistakes when looking at exactly K values. Instead of considering a single value of K, we also have to evaluate $\delta(K_i)\ \forall k_i \in [1, K]$, i.e. obtain the probability of not making 1 error, 2 errors, ..., K errors.

Finally, to estimate $\mathbb{E}[P]$, the expected value of precision $P$, we average the probability of errors using the sample average as a non-biased estimator, and obtain Equation (5.2) as $\mathbb{E}[P] \approx \frac{1}{K} \sum_{K_i=1}^{K} \delta K_i$.

■

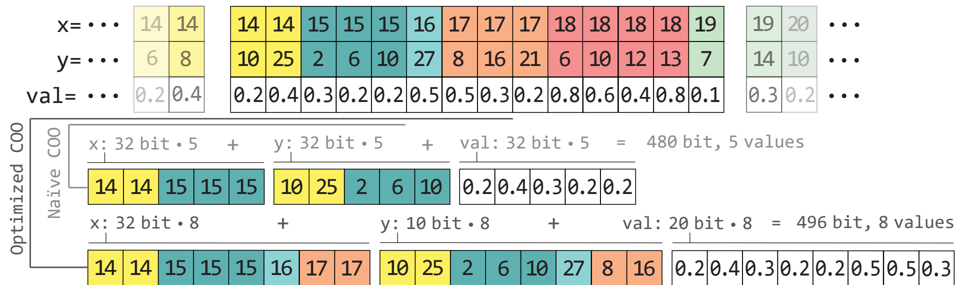



**Figure 5.4:** *Naïve Coordinate (COO) allows only 5 non-zero entries in a 512-bits packet. Using fewer bits for* `y` *and* `val` *(assuming* $y < 1024$ *and* `val` *stored as 20-bits fixed-point) permits only 3 more non-zeros per packet. However,* `x` *still contains significant redundancy.*

Precise evaluation of Equation (5.2) for realistic sizes (i.e. $N$, the number of rows, at least in the order of millions) introduces problems of numerical accuracy due to the extremely large binomial coefficient involved in the computation. Instead, we estimate $\mathbb{E}[P]$ through a Monte Carlo simulation for different $K$ and number of partitions, with $K$ from 8 to 100, common thresholds in IR [39]. Having at least 16 partitions guarantees a minimal loss of precision, even for large matrices. Using 32 partitions, as in our final design, guarantees almost no accuracy loss even for $\mathrm{K} = 100$.

### 5.3.2 The Block-Streaming CSR Matrix Layout

The matrix $\mathrm{A}$ is stored in a *sparse format* to save only non-zero values and lower its memory footprint. Different storage techniques are possible, depending on the non-zero elements distribution and the desired type of access patterns.

The common CSR format is unsuitable for fully-pipelined streaming FPGA designs as it contains data dependencies to access matrix values and hiding the memory controller latency is not trivial, given the lack of a built-in hardware prefetcher. Instead, the COO layout uses three equally sized arrays to store, for each matrix entry, its two coordinates and the value of the entry itself. COO allows streaming iterations with burst memory transactions on non-zero matrix entries to saturate bandwidth, as the architecture does not have to perform data-dependent memory accesses determined by the number of non-zero values of each row, as in the CSR format. COO allows coalesced memory transactions of multiple non-zero entries but requires redundant storage of coordinate values, limiting the overall operational intensity [218]. The COO format also simplifies fine-grained array

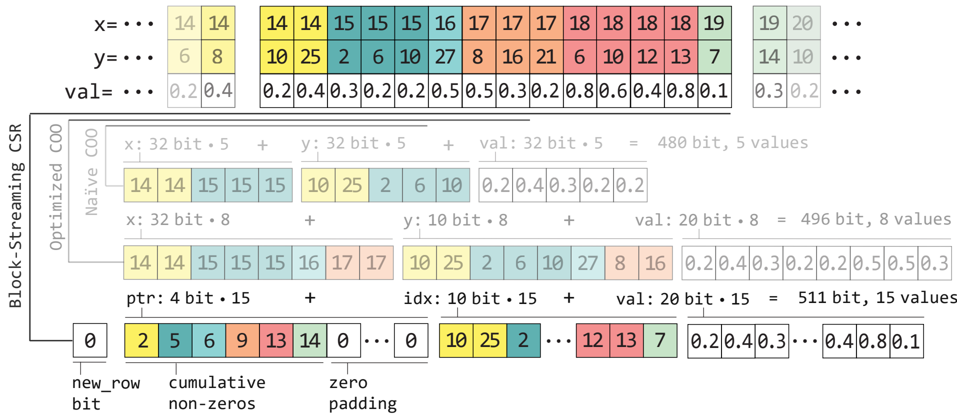



**Figure 5.5:** *BS-CSR drastically reduces the space required for* `ptr`*, storing a cumulative sum similarly to the CSR format, but counting values only in the current packet. For a packet of size* $B$*,* `ptr` *takes only* $\lceil log_2 B \rceil \cdot B$ *bits. In this example, BS-CSR enables storing 15 non-zeros in a 512-bit packet, compared to the 5 to 8 of COO in Figure 5.4.*

partitioning, as computation on each non-zero value can be done independently from the others.

When working with partitioned or streaming COO data, it is reasonable to store non-zero entries as `(x, y, val)` triples, instead of using separate arrays. Moreover, the HBM memory controller of many architectures, including one of GPUs and of the Alveo U280 [205], reaches higher throughput with coalesced memory transactions of large data packets (e.g. 512 bits). As such, each memory transaction will load multiple non-zero entries of the COO matrix, similar to what was done in Section 4.4.2. In Figure 5.4, we load 512-bits packets each containing 5 COO matrix entries, as values in `x`, `y` and `val` require 32 bits each ($\lfloor 512/(32 \cdot 3) \rfloor = 5$. However, when working with tall matrices ($N >> M$, as in our use-case) and reduced-precision arithmetic, one can encode values of `y` and `val` with fewer bits. Storing entries of `y` using 8 bits each (assuming $M < 1024$) and entries of `val` in 20 bits, we can store 8 non-zero entries in a single 512-bits data packet. Although we can optimize the encoding of `y` and `val`, the storage of `x` is still wasteful, because of the 32-bits encoding and of the redundancy intrinsic of the COO format.

To solve these problems, we propose BS-CSR, a new sparse matrix storage format that combines the low memory footprint of CSR with the streaming properties of COO, and is enabled by reduced precision datatypes. As the architecture of the Alveo U280 HBM memory controllers incentivizes memory transactions of data packets (or blocks) between 256

and 512 bits [205], we build each packet in BS-CSR as an independent CSR partition (Figure 5.5): values of `idx` and `val` are identical to standard CSR, using less than 32 bits for each value if possible (in our case, 10 bits are enough for `idx` entries). The `ptr` vector tracks the cumulative non-zero entries with respect to the packet itself: each entry of `ptr` is just $\lceil log_2 B \rceil$ bits for a packet with $B$ non-zero entries (4 bits for a packet $B = 15$), instead of 32 bits like in a naïve CSR or COO (as the number of rows is not bound). The $new_row$ bit tracks if the first row of a packet continues the last row of the previous packet. Missing rows are handled with placeholder 0 values, although rows are never fully empty in our application domain. BS-CSR does not provide an explicit reference to row indices: this is not a limitation in streaming algorithms such as SpMV and its variations, as we process the entire matrix sequentially and track the current rows by counting non-zero entries in each `ptr` packet, plus the $new_row$ bit. We can fit 2 to 3 times as many non-zero entries in each 512-bits packet and improve the operational intensity by the same amount.

BS-CSR is extremely effective on tall matrices ($N >> M$) as we can greatly reduce the number of bits required for `ptr` and `idx`. However, it is still beneficial when working with square matrices representing graphs, as in Chapter 3 and Chapter 4. Even if storing `idx` and `val` with 32 bits for each entry, a 512-bits packet can contain 7 values, 40 % more than the 5 of COO. By assuming $N = M < 2^{24}$ (as in the largest matrix used in the evaluation of Chapter 4, Table 4.2) and `val` stored with 20 bits per entry, BS-CSR can contain 10 values, compared to the 7 of an equivalent COO.

Although significant work has been made to optimize SpMV on matrices with irregular row density distribution [172], we avoid the problem entirely through a fully-streaming data format that is oblivious to the number of non-zero matrix entries per row. In spite of its simplicity, BS-CSR can be a life-saver in streaming memory-bound computations (Section 5.5.3).

## 5.4 The Proposed FPGA Hardware Design

Our Top-K SpMV FPGA design offers multiple low-profile cores that operate independently. Each core processes a partition of the input matrix, as explained in Section 5.3.1. Each core uses a single HBM channel to read the input matrix and store its output at the very end of the computation to guarantee maximum flexibility in terms of the number of cores placed on the FPGA. As cores read 512 bits per clock cycle (at 225 MHz) from their respective HBM memory port and memory transactions happen in continuous maximum length AXI4 bursts (256 beats), our FPGA design

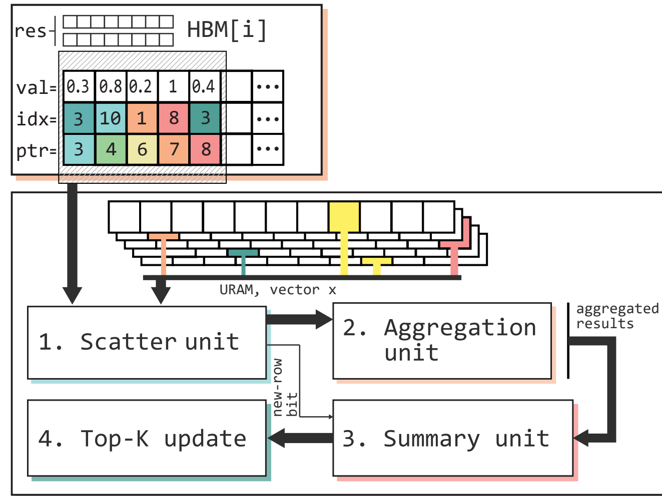



**Figure 5.6:** *Simplified block diagram of a single core of our Top-K SpMV hardware design, for BS-CSR with 5 non-zero* `y` *and* `val` *entries per packet.*

can theoretically operate at the maximum bandwidth offered by HBM, by coalescing transactions with BS-CSR packets containing multiple non-zero entries, and without expensive distributed memory controllers. Each core reads A in packets of size $B$. $B$ ranges from 7 to 15, depending on the desired numerical precision and the embedding size: at worst we use 32 bits for `idx` and `val`, but realistic size bounds (e.g. `idx` $< 1024$) allow much greater coalescing and operational intensity (Figure 5.9).

### 5.4.1 Leveraging URAM for Fast Random Access

The input vector x is stored in URAM. Our core performs $B$ random accesses per cycle on x. As each URAM bank only has 2 read ports, we replicate x $\lceil B/2 \rceil$ times to allow all random accesses. This replication does not constitute a limitation of our design: x represents a dense embedding whose size does not go beyond a few hundred values in real applications. With our implementation, x can have a size up to 80000 (assuming a worst-case scenario with 32 bits values, 32 cores, and 8 replicas of x per core), given a URAM size of around 90MB. Compared to the SpMV hardware design in Chapter 4, we store the dense vector x in URAM instead of using HBM. By partitioning x on independent URAM cells, we access $B$ arbitrary values per clock cycle, without being limited by the number of HBM pseudo channel as in Section 4.4.2. As such, we can replicate our Top-K SpMV core 32 times, compared to the 5 of Chapter 4. The original SpMV

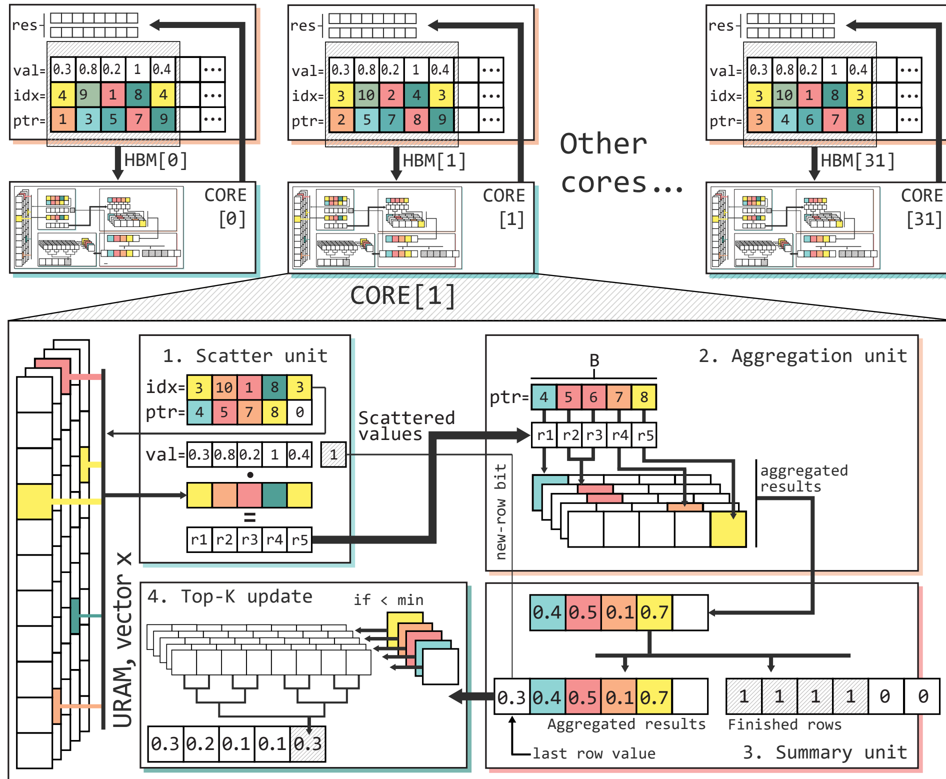



**Figure 5.7:** *Block diagram of our multi-core Top-K SpMV, for BS-CSR with 5 non-zero* y *and* `val` *entries per packet.*

design in Chapter 3 also stored dense vectors in URAM. In this chapter, however, we work with tall matrices ($N \geq M$) instead of square matrices ($N = M$): storing small vectors of size $M$ in URAM does not introduce constraints that affect our use-case, and does not cause any slowdown in the synthesized design's clock frequency due to excessive congestion.

### 5.4.2 Top-K SpMV Algorithm Design

The main Top-K SpMV algorithm (Algorithm 9) is implemented as a 4-stage pipelined data-flow computation, decoupling memory transfer and computation. A simplified block-diagram of the architecture is given in Figure 5.6. Each stage processes the input matrix as packets of $B$ elements. The first stage retrieves $B$ entries from A per cycle from HBM and URAM and computes point-wise products ①. Values corresponding to the same matrix row are aggregated in the second stage ②. The third stage performs book-keeping for rows that span more than one packet and finds which rows

have been fully processed in the current packet ③. The final stage updates the current Top-K values for each complete row if the row output value is in the Top-K ④. Our data-flow design hides the Top-K update cost, an otherwise expensive part of the computation. Figure 5.7 provides a more detailed representation of our Top-K SpMV hardware design. Compared to the previous SpMV designs in Chapter 3 and Chapter 3, we observe how each core is connected to an independent HBM pseudo channel, and that BS-CSR results in a simpler summary unit ③ as each BS-CSR provides a bit that explicitly tracks whether the first row in the packet continues the last row in the preceding packet. On the other hand, Top-K SpMV requires a significantly more complex write-back/update unit ④, as we require multiple parallel tree reductions to track the highest K entries of the result.

Each core retrieves the top $\mathrm{k} = 8$ values of each matrix partition. Higher $\mathrm{k}$ results in lower clock frequency due to RAW data-dependencies in the `argmin` computation, while lower $\mathrm{k}$ decreases accuracy. To guarantee that our design can sustain a pipeline with Initiation Interval (II) of 1 (i.e. that we read and process $B$ new values in each clock cycle) we store partial results in $B$ different buffers of size $K$ (one buffer for each non-zero entry in the packet), and update them independently. This optimization is required as each packet can contain more than one finished row, and we need to update the Top-K output vector without creating data dependencies that cannot be handled within a clock cycle. Partial results are aggregated by the FPGA at the end of the execution. In practice, the similarity value of a row $i$ is stored in the output buffer $B_0$ if it is the first row to finish in a given packet; it will be stored in $B_1$ if another row finished before $i$ in the same packet; it will be stored in $B_2$ if two rows finished before $i$ in the same packet, and so on.

Output buffers $B_i$ with $i > 0$ are used only if at least one row is completely contained in a packet ($B_0$ can be used for rows that started in a previous packet). It is unlikely that many rows are entirely contained in a single packet $B$: we avoid consumption of unnecessary logic by tracking results for at most $\mathrm{r}$ rows per packet. In our experiments, using $B/4 < \mathrm{r} < B/2$ provided resource savings up to 50 % with no accuracy loss.

### 5.4.3 Lower Precision, More Cores, Better Performance

Our hardware design processes $\mathrm{c} \cdot B$ non-zeros per clock cycle, with $\mathrm{c}$ being the number of Top-K SpMV cores. Maximizing this performance equation is not trivial, as increasing $B$ increments FPGA resource consumption and possibly prevents the placement of $\mathrm{c} = 32$ cores, a requirement to achieve

**Algorithm 9** Approximate BS-CSR Top-K SpMV

1: **function** TOP-K-SPMV($bscsr_matrix$, vec)
2: $x_{s_old} \leftarrow 0; last_packet_output \leftarrow 0$
3: **for** $i \leftarrow 0..\text{NNZ}/B$ **do** ▷ For each matrix packet
4: ▷ **1. Process BS-CSR in packets of size $B$**
5: **for** $j \leftarrow 0..B$ **do** ▷ Unroll loops of size $B$
6: $bscsr_packet \leftarrow bscsr_matrix[i]$
7: $x_{loc}[j] \leftarrow bscsr_packet.x[j]; \quad val_{loc}[j] \leftarrow bscsr_packet.val[j]$
8: $res_{tmp}[j] \leftarrow val_{loc}[j] \cdot \text{vec}[j][bscsr_packet.y[j]]$
9: **end for**
10: ▷ **2. Aggregate partial $res_{loc}$ values**
11: **for** $b1 \leftarrow 0..B$ **do** ▷ $x_{loc}[-1] = 0$ by convention
12: $row_{curr} \mathrel{+}= (x_{loc}[b1] \neq x_{loc}[b1-1])$
13: **for** $b2 \leftarrow x_{loc}[b1-1]..x_{loc}[b1]$ **do**
14: $res_{agg}[b1] \mathrel{+}= res_{tmp}[b2]$
15: **end for**
16: **end for**
17: $row_{curr} \mathrel{+}= bscsr_packet.new_row - 1$
18: ▷ **3. Check if first value was split among packets**
19: **for** $j \leftarrow 1..B$ **do** ▷ Find finished rows
20: $rows_{fin}[j] \leftarrow x_{loc}[j-1] \neq 0$
21: **end for**
22: **if** $bscsr_packet.new_row$ **then**
23: $res_{agg}[0] \leftarrow last_packet_output; \quad rows_{fin}[0] \leftarrow true$
24: **else** ▷ This packet continues a previous row
25: $res_{agg}[1] \mathrel{+}= last_packet_output$
26: $res_{agg}[0] \leftarrow 0; \quad rows_{fin}[0] \leftarrow false$
27: **end if**
28: ▷ **4. Update Top-K values**
29: **for** $j \leftarrow 0..B$ **do** ▷ Process only finished rows
30: **if** $res_{agg}[j] \geq worst_{curr}[j]$ && $rows_{fin}[j]$ **then**
31: $res_{loc}[j] \leftarrow res_{agg}[j]; \quad res_{idx}[j] \leftarrow row_{curr} + j - 1$ ì
32: **end if**
33: $argmin(res_{loc},\ worst_{curr}[j],\ worst_{idx}[j])$
34: **end for**
35: $reset(res_{agg})$
36: **end for**
37: **end function**

full HBM bandwidth utilization. Indeed, $\mathrm{c}$ is a non-linear function of $\mathrm{k}$, $\mathrm{r}$ and $B$, with $B$ depending from $M$ and from the number of bits $V$ used for entries of `val`, as in $B \cdot (\lceil log_2 B \rceil + \lceil log_2 M \rceil + V) + 1 = 512$. $M$ can be safely assumed in the order of thousands. Choosing $V$, $\mathrm{k}$ and $\mathrm{r}$ requires a performance-accuracy trade-off: based on theoretical bounds in Table 5.1 we observe that $\mathrm{k} = 8$ is sufficient. A larger value of $k$ drastically increases resource utilization without translating to observably better accuracy. In the experiments in Section 5.5.4, we test different values of numerical accuracy $V$, from 20 bits to 32. Even $V = 20$ is sufficient to provide great accuracy. Combined with $\mathrm{r}$ between 4 and 8 we guarantee a 32-cores design that can exploit all HBM channels without significant accuracy loss. Table 5.2 summarizes characteristics and resource usage of the designs in our evaluation.

### 5.4.4 Host Integration

Our proposed Top-K SpMV FPGA hardware design follows a standard host-accelerator model in which the host (a traditional server) communicates with the accelerator (a Xilinx Alveo U280 accelerator card) over a PCIe 4.0 interconnection. The host loads the matrix and computes its BS-CSR compressed representation. While this compression requires a full scan of the matrix, it is also trivial to parallelize across multiple CPU cores. Moreover, we can assume that the input matrix, representing a corpus of embeddings, stays relatively fixed in time and is updated much more infrequently compared to the number of input queries.

## 5.5 Experimental Evaluation

We employ a Xilinx Alveo U280 Accelerator Card equipped with $8\,\mathrm{GB}$ of HBM2 memory ($460\,\mathrm{GB/s}$ of total bandwidth over 32 channels) and a `xcu280-fsvh2892-2L-e` FPGA whose available resources are reported in Table 5.2. Our CPU baseline is `sparse_dot_topn`, a multi-threaded C++ implementation of Top-K SpMV [19], running on two Intel Xeon Gold 6248 and $384\,\mathrm{GB}$ of DRAM. We compare our design against GPUs, using a Tesla P100 ($549\,\mathrm{GB/s}$ of HBM bandwidth): as we are not aware of any GPU implementation of Top-K SpMV, we combined a fast GPU implementation of SpMV (from cuSPARSE [139]) with the GPU radix sort of Thrust [21], using both single and HP floating-point arithmetic. To provide a worst-case comparison, we also assume a zero-cost GPU sorting, as if cuSPARSE already retrieved Top-K values at no cost.

**Table 5.2:** *Resource usage, clock frequency and power consumption of our architecture. Best values in bold. We report the clock frequencies of the designs in Parravicini et al. [155] and the best frequencies obtained after further optimizations.*

| **Bit-width** | **Cores** | **LUT** | **FF** | **BRAM** | **URAM** | **DSP** | **Clock (MHz)** | **Power Cons.** |
|---|---|---|---|---|---|---|---|---|
| **20 bits** | 32 | 38% | 35% | 20% | 33% | **7%** | **253-336** | **34 W** |
| **25 bits** | 32 | 38% | 36% | 20% | 30% | 11% | 240-322 | 35 W |
| **32 bits** | 32 | **35%** | **33%** | 20% | 27% | 17% | 249-306 | 35 W |
| **32 bits, float** | 32 | 44% | 37% | 20% | **26%** | 19% | 204-237 | 45 W |
| **Available** | | 1097419 | 2180971 | 1812 | 960 | 9020 | | |

We analyze 4 FPGA design: 32 bits unsigned fixed-point (`U1.31`), 25 bits (`U1.24`), 20 bits (`U1.19`), and a 32-bit floating-point version (`F32`) (Table 5.2). The number of HBM pseudo channels limits the maximum number of cores to 32, although we could easily place more cores given our design's low resource footprint. The CPU baseline uses floating-point arithmetic: our CPU does not support arbitrary reduced-precision, and simulated reduced-precision fixed-point resulted in lower performance. The experimental setup comprises 19 synthetic and real sparse embedding matrices (Table 5.3), with different sizes and distributions. Sizes are reported using BS-CSR as in Figure 5.5: if stored as a naïve COO, they would take 3 times as much space. We simulate different non-zero distributions (uniform and left-skewed $\Gamma$), with 20 or 40 average non-zero entries per row (a sparsity factor of 2–8 %). The use of synthetic matrices provides full control over the desired data distribution, as the chosen sparsification procedure influences the non-zero distribution in real embedding matrices. To the best of our knowledge, no public dataset of sparse embedding with sizes comparable to ours (millions of rows) is available. Instead, we sparsify the GloVe [160] embedding corpus with the technique in [129]. Our performance is mostly unaffected by the underlying data distribution (Section 5.5.1): as our FPGA design processes the matrix in a streaming fashion, density does not affect performance. Embedding values are normalized in $L2$ norm to guarantee that similarities are scaled correctly with respect to each other and that we can use a single bit for the fixed-point integer part. We perform each test 30 times, with different random vertices $\mathrm{x}$.

**Table 5.3:** *Matrices in the evaluation, with $M = 512$ and $1024$, and memory occupation using BS-CSR as in Figure 5.5. Distribution controls the number of non-zeros per row.*

| **Distribution** | **Rows** | **Non-zeros (min-max)** | **Size (min-max, GB)** |
|---|---|---|---|
| Uniform | $0.5 \cdot 10^7$ | $10^8 - 2 \cdot 10^8$ | $0.4\,\text{GB} - 0.8\,\text{GB}$ |
| | $1.0 \cdot 10^7$ | $2 \cdot 10^8 - 4 \cdot 10^8$ | $0.8\,\text{GB} - 1.7\,\text{GB}$ |
| | $1.5 \cdot 10^7$ | $3 \cdot 10^8 - 6 \cdot 10^8$ | $1.2\,\text{GB} - 2.5\,\text{GB}$ |
| $\Gamma(k = 3, \theta = 4/3)$ | $0.5 \cdot 10^7$ | $9.7 \cdot 10^7 - 1.97 \cdot 10^8$ | $0.4\,\text{GB} - 0.8\,\text{GB}$ |
| | $1.0 \cdot 10^7$ | $1.9 \cdot 10^8 - 3.95 \cdot 10^8$ | $0.8\,\text{GB} - 1.7\,\text{GB}$ |
| | $1.5 \cdot 10^7$ | $2.9 \cdot 10^8 - 5.92 \cdot 10^8$ | $1.2\,\text{GB} - 2.5\,\text{GB}$ |
| Sparsified GloVe | $0.2 \cdot 10^7$ | $2.4 \cdot 10^7 - 4.6 \cdot 10^7$ | $0.1\,\text{GB} - 0.3\,\text{GB}$ |

**Table 5.4:** *Execution time of our Top-K SpMV FPGA hardware designs with different clock frequencies. Although the HBM pseudo channels of the Alveo U280 supposedly reach peak performance when reading 512 bits per cycle at 225 MHz, we still observe a performance improvement even with higher frequencies. Between parenthesis, we show the relative frequency improvement and the execution time speedup.*

| **Bit-width** | **Frequency (Mhz)** | | | **Execution time (ms)** | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | | $\mathrm{N} = 0.5 \cdot 10^7$ | | | $\mathrm{N} = 1.0 \cdot 10^7$ | | | $\mathrm{N} = 1.5 \cdot 10^7$ | | | **GloVe** | | |
| **20 bits** | 253 | 336 | (1.32×) | 2.76 | 2.16 | (1.27×) | 4.80 | 3.66 | (1.31×) | 7.04 | 5.26 | (1.32×) | 0.86 | 0.76 | (1.15×) |
| **25 bits** | 240 | 322 | (1.34×) | 3.24 | 2.53 | (1.27×) | 5.78 | 4.42 | (1.30×) | 8.39 | 6.38 | (1.31×) | 1.08 | 0.89 | (1.21×) |
| **32 bits** | 249 | 306 | (1.22×) | 3.72 | 3.03 | (1.22×) | 5.71 | 5.41 | (1.05×) | 9.70 | 7.86 | (1.22×) | 1.13 | 0.95 | (1.18×) |
| **32 bits, float** | 204 | 237 | (1.17×) | 6.48 | 5.47 | (1.17×) | 11.8 | 9.98 | (1.17×) | 17.37 | 14.94 | (1.16×) | 1.88 | 1.53 | (1.22×) |

### 5.5.1 Execution Time

We measure the speedup against CPU and GPU implementations, for $K = 100$, and distinguishing between matrix sizes and non-zero distributions. Figure 5.8 shows the speedup of each GPU implementation and FPGA hardware design, using the CPU execution time (reported below each plot) as the baseline. In the case of GPUs, we show both the speedup of the implementation with sorting and the speedup of the idealized implementation with zero-cost sorting. While both GPU and FPGA implementations are significantly faster than the CPU baseline, our FPGA design achieves a $2.1\times$ speedup over an idealized GPU Top-K SpMV implementation with zero-cost sorting, despite the $20\,\%$ lower peak memory bandwidth. When accounting for sorting costs on the GPU, this speedup can be as large as $7\times$. We expect to provide competitive performance even against high-end GPUs with significantly higher memory bandwidth (e.g. Tesla A100, $1.5\,\text{TB/s}$),

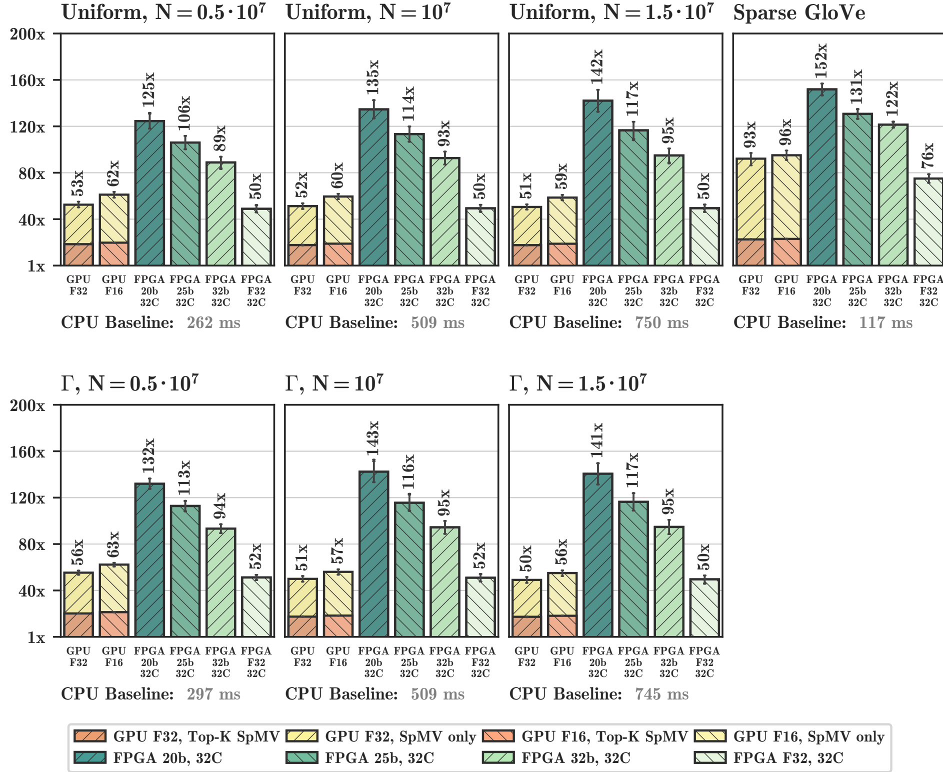



**Figure 5.8:** *Execution time speedup on Top-K SpMV of our FPGA designs vs. GPU and CPU. Results are divided based on the distribution of the non-zero entries in the matrix, and based on the number of rows in the matrix. We can process over 77 billion non-zeros per second, 2.1× more than the GPU SpMV.*

and always provide greater power-efficiency (Section 5.5.2). Fixed-point arithmetic guarantees higher speedups than floating-point, thanks to the pipelines' lower initiation interval. The largest speedups, both for GPUs implementations and FPGA designs, are found on the relatively small GloVe dataset. On the other hand, FPGA designs show the largest performance gap over GPUs when dealing with the biggest matrices. This phenomenon is explained by OpenCL overheads that occur in the FPGA computation scheduling and are better amortized over longer computations. Reduced precision enables packing more non-zeros per transfer (higher $B$), resulting in an increased operational intensity and better performance. Our 32-cores design can find the Top-K values of a matrix with $10^7$ rows and 200 million non-zero entries in less than $3\,\mathrm{ms}$ and is suitable for real-time applications.

Compared to the results reported in Parravicini et al. [155], we man-

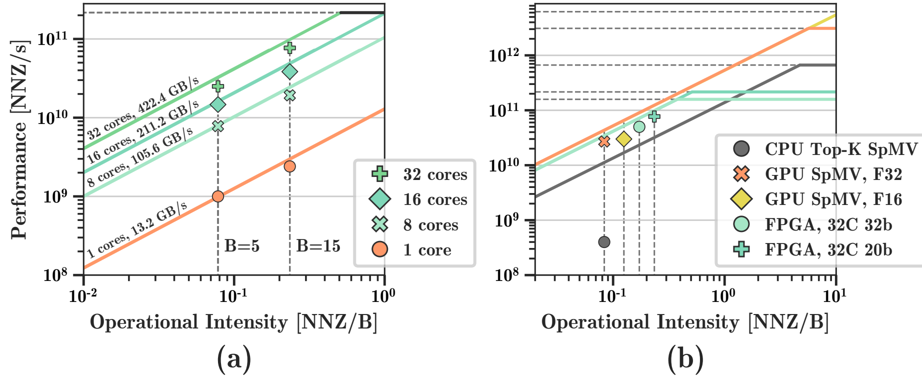



**Figure 5.9:** *Roofline model of our Top-K SpMV architecture. ⓐ Operational intensity increase with BS-CSR. ⓑ Comparison of FPGA vs. CPU and GPU. Thanks to BS-CSR, we provide the highest operational intensity and the best performance.*

aged to synthesize our hardware design with higher clock frequencies. The original clock frequencies, and the best ones that have been achieved, are reported in Table 5.2. We expected the performance of our hardware design to plateau at 225 MHz, since the HBM pseudo channels on the Alveo U280 reach peak throughput when reading 256 bits per clock cycle at a frequency of 450 MHz, or equivalently, 512 bits per clock cycle at a frequency of 225 MHz [205, 213]. Since our hardware design is fully bound by memory throughput (Section 5.5.3), we expected that higher frequencies would not improve performance, and possibly introduce stalls in the pipeline due to discrepancies between the memory controller frequency and the cores frequency. Table 5.4 shows the clock frequencies and execution time of each FPGA hardware design, aggregated for matrices with the same number of rows. We observe that increasing the clock frequency above 225 Mhz translates to an almost-linear speedup. Further analyses with tools provided by Xilinx revealed that the original design suffered from pipeline stalls, which might have been mitigated by the higher clock frequency. As a rule of thumb, we can preliminary conclude that targeting clock frequencies above the one of the FPGA memory controller can be beneficial, even in FPGA hardware designs that are heavily memory-bound, such as ours.

### 5.5.2 Power Efficiency

Our FPGA hardware design consumes about 35 Watts during execution, plus 40 Watts for the host server, measured with an external power meter monitor. Changing bit-width did not affect the power consumption in

a measurable way. The CPU implementation consumes around 300 Watts during execution, while the GPU requires 250 Watts (plus 40 Watts for the host). Our fixed-point FPGA design provides 550× higher Performance/Watt ratio than the CPU, and 15× compared to the idealized GPU implementation (8.1× when accounting for an equal host machine): we provide higher performance without any sacrifice of power efficiency.

### 5.5.3 Roofline Model Analysis

Besides looking at raw execution time values, we want to understand how well our FPGA hardware design leverages the resources available to it, and how it compares against other architectures. We would expect to be completely memory bound and to be using most, if not all, of the $460\,\mathrm{GB/s}$ of HBM2 bandwidth offered by the Alveo U280. To answer these questions, we build a Roofline Model, following the methodology in [184] (Figure 5.9). On the x-axis, we measure the operational intensity of the computation. In our case, we evaluate how many non-zero elements of the sparse matrix we can process for each byte read from off-chip memory. For FPGA designs, this value is immediately computed from the number of non-zero entries stored in a data packet (e.g. 5 for COO or 15 for BS-CSR, Figure 5.5). For CPU and GPU implementations, we obtain an upper-bound estimate of the operational intensity by counting how many memory accesses are required to read the matrix fully, and optimistically assume that the dense input vector is only read once and then kept in cache. On the y-axis, we display the performance, expressed as non-zeros entries processed per second. Diagonal lines express the maximum off-chip memory bandwidth of each architecture and FPGA hardware design (based on how many HBM pseudo channels it uses). Horizontal lines mark the peak performance. For CPU and GPU implementations, we use the single and half-precision floating-point peak performance as given by the hardware manufacturer. For FPGA designs, the peak performance is estimated by fitting as many Top-K SpMV cores as possible – 64 in our case – without being limited by the 32 HBM pseudo-channels of the Alveo U280, and staying within a conservative 80 % resource utilization bound (Table 5.2).

Figure 5.9 ⓐ shows that the performance (non-zeros per second) of our FPGA design scales linearly to the total bandwidth of the HBM channels: this result enables predicting performance when deploying our design on an FPGA board with fewer (or more) HBM channels. Most importantly, BS-CSR increases the operational intensity up to 3× compared to a naïve COO ($B = 15$ vs. $B = 5$), which immediately translates to an equivalent

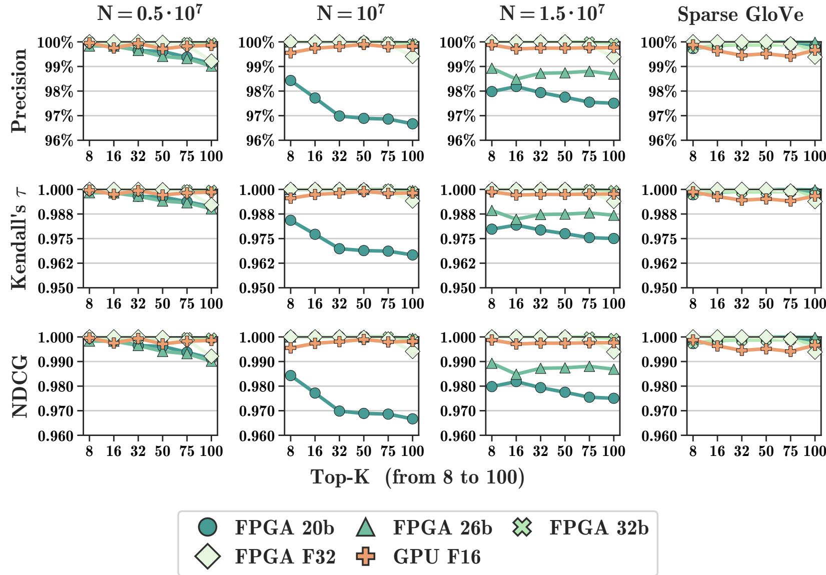



**Figure 5.10:** *Top-K SpMV accuracy (higher is better) for different architectures and types of reduced-precision arithmetic, aggregated on matrices with the same size.*

performance improvement. Figure 5.9 ⓑ reports the performance against CPU and GPU. For the GPU, we provide a worst-case analysis by considering only the cost of SpMV and assuming zero-cost sorting. When compared against CPU and GPU, our FPGA design is the best in terms of operational intensity and bandwidth usage, both in absolute and percentage terms. Even with 20 % less bandwidth than the GPU and considering a Top-K SpMV GPU implementation with zero-cost sorting, the increased operational intensity of BS-CSR still provide the highest performance.

We see that our FPGA design always operates with a memory bandwidth close to the maximum achievable for a given number of HBM pseudo channels, and its performance scales linearly with respect to the number of cores and HBM pseudo channels. To further improve performance, we would need to increase the operational intensity, e.g. by processing more non-zero entries per clock cycle through additional data compression.

We can also compare the performance of our BS-CSR Top-K SpMV hardware designs to what we could achieve using alternative designs based on COO or CSR as sparse matrix storage layout. The COO design has a matrix fetch unit and a scatter unit identical to the ones in Chapter 4, and store the dense vector in URAM as in the design presented in this chapter. The naïve COO encoding in Figure 5.4 stores 5 non-zero entries in each

512-bits data packet, giving the performance shown in Figure 5.9 ⓐ for $B = 5$. The optimized COO encoding in Figure 5.4 would not fare much better, being equivalent to $B = 8$. It is 60 % better than the naïve COO encoding, but still 2× worse than BS-CSR.

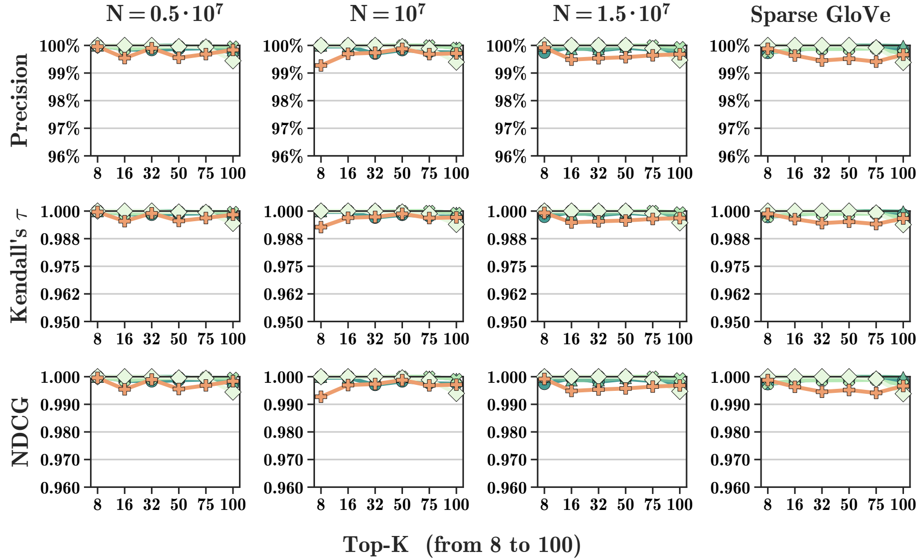
In our experiments, a CSR created by adapting our existing hardware designs is about 10× slower than naïve COO. We cannot obtain a fully streaming design due to indirect memory accesses introduced by the `ptr` vector in the CSR representation, because it is not possible to know at design time how many non-zero entries there are in each matrix row.

### 5.5.4 Approximation Accuracy Analysis

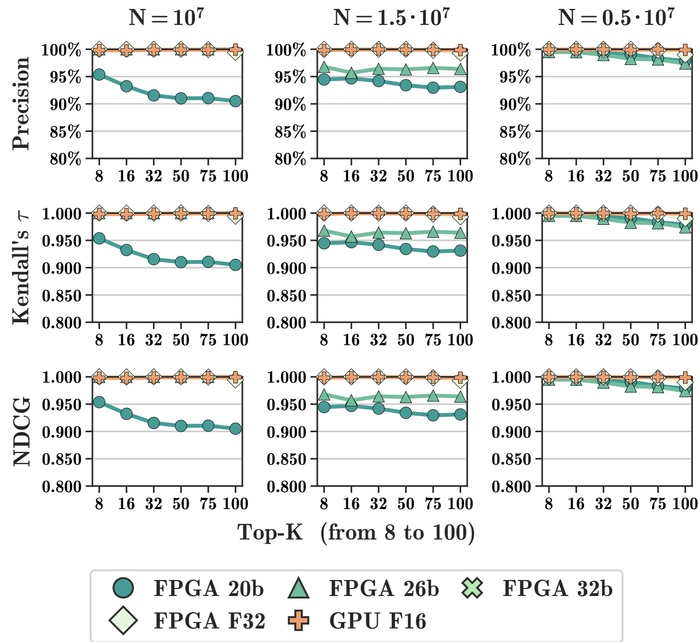
We compare the accuracy of our approximated Top-K SpMV with the exact CPU results and with the approximated GPU results with HP floating-point. We look at values of K from 8 to 100 and evaluate accuracy using common Recommender System evaluation metrics such as Precision, Kendall's $\tau$, and Normalized Discounted Cumulative Gain (NDCG) [180]. Precision does not penalize out-of-order results; the other two metrics do. Further explanations of these metrics are given in Section 2.3.3 and Section 3.5.3. We show accuracy results aggregated over all matrices with a given number of rows in Figure 5.10. Figure 5.11a presents results for matrices with uniformly-distributed non-zero entries and for the GloVe dataset, while Figure 5.11b shows results for matrices with $\Gamma$ non-zero distribution.

Our accuracy results are in line with the theoretical estimations in Table 5.1, showing only a minor accuracy dip for large K. Even 20-bit fixed-point designs provide excellent accuracy across the board, with Precision above 97 % when looking at average results in Figure 5.10. Moreover, 32-bits fixed-point designs provide accuracy above the HP floating-point GPU implementation, despite our algorithmic approximation.

Accuracy results for matrices with uniformly distributed non-zero entries, and for GloVe, show that our hardware design achieves close-to-perfect results even for $K = 100$, and is almost always more accurate than an exact computation with HP floating-point arithmetic. When non-zeros follow a $\Gamma$ distribution we observe a somewhat lower accuracy, especially in designs with 20-bit fixed-point arithmetic. This lower precision is explained by having many small rows ending in the same data packet. Due to our approximations in Section 5.4.3, we might lose track of the similarity value of some of these rows, resulting in lower accuracy. Still, even in this case, we can guarantee accuracy results well above 90 % in each metric.




**(a)** *Error on matrices with uniform distribution and GloVe.*



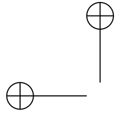
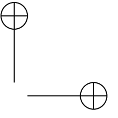
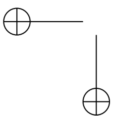
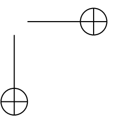

**(b)** *Error on matrices with* Γ *distribution.*

**Figure 5.11:** *Top-K SpMV accuracy (higher is better) for different architectures and types of reduced-precision arithmetic, divided by matrices with uniform non-zeros distribution and the GloVe dataset ⓐ and matrices with* Γ *non-zeros distribution ⓑ.*

## 5.6 Final Remarks

Top-K SpMV is a cornerstone of IR and recommender systems based on sparse embeddings similarity and must be computed, guaranteeing real-time latencies and power efficiency. As these constraints are hardly met on general-purpose architectures, we presented a novel approximate FPGA multi-core design for Top-K SpMV that leverages HBM, fixed-point arithmetic, and our new BS-CSR matrix layout to achieve **high-performance and modularity over the FPGA resources**. Our 32-cores 20-bit FPGA design achieves 138× the performance of a state-of-the-art CPU baseline and 2.1× the GPU performance with 20 % higher bandwidth. We deliver **real-time results** on sparse matrices with **hundreds of millions of values** while keeping 15× higher power efficiency than the GPU.

Future work will focus on adaptive compressed matrix representations by reconfiguring the FPGA in terms of numerical precision to guarantee desired targets of accuracy or performance. We will also apply our design to smaller FPGA accelerator cards: with similar memory bandwidth, the computation can be cheaper and even more power-efficient, with no performance loss. Finally, we will study how to apply BS-CSR to other computations and possibly other architectures.

CHAPTER *6*

# Conclusion and Future Work

*In this final chapter, we summarize the contributions presented in this thesis, illustrate the limitations of our current work, and highlight opportunities created by our research. Throughout this thesis, we addressed the challenges and the benefits of providing hardware acceleration to graph analytics and recommender systems using sparse linear algebra and reconfigurable architectures. Our contributions have shown how novel memory technologies, combined with reduced-precision arithmetic and custom data representations, enable low-latency computations with minimal power consumption. Still, further research opportunities do exists, such as extending our hardware designs to a broader set of problems, and scaling to even larger computations.*

Recommender systems are now the majority of AI workloads found in datacenters [63]. Moreover, sparse computations are seen as the key to scale to larger datasets and to incorporate heterogeneous information such as graph topologies and categorical values. A graph with one million vertices, a common occurrence in graph analytics, can be stored in a few megabytes using sparse matrices. A dense representation of the same graph requires about four terabytes, making it impossible to handle in most scenarios. Despite this real-world need, hardware optimizations for recommender systems are still a minority, compared to all research targeted at computer architectures for machine learning [63].

This thesis tried to address open research challenges in hardware acceleration for sparse linear algebra applied to graph analytics and recommender systems. We have seen how the *memory wall*, the growing memory gap between peak compute power and memory size and bandwidth, is a major cause of concern in sparse computations, as they are extremely memory-intensive. Breaking this memory wall will require significant technological advancements. In the meantime, compression techniques and reduced-precision arithmetic can successfully mitigate its impact by providing an increased operational intensity and maximizing the computation on every bit loaded from memory. Field-Programmable Gate Array (FPGA) accelerator cards, thanks to the availability of High Bandwidth Memory (HBM) and abundant on-chip memory, are a valid solution for high-performance sparse computations in graph analytics and recommender systems. No off-the-shelf hardware architecture can provide the same performance per watt in these workloads while guaranteeing real-time execution latency and fine-grained control over accuracy versus execution time thanks to reduced-precision fixed-point arithmetic.

The core of this work was a **flexible Sparse Matrix-Vector Multiplication (SpMV) hardware design**, created to leverage reduced-precision fixed-point arithmetic and multiple kinds of memories, such as UltraRAM (URAM) and HBM2. We have proven how this design can adapt to **different sparse workloads**, always reaching **state-of-the-art performance**: graph ranking algorithms, sparse eigensolvers, and sparse embedding similarity search. In detail, we presented the following contributions.

- In Chapter 3 we first presented our FPGA SpMV hardware design and used it to accelerate the computation of **Personalized PageRank (PPR)**, obtaining speedups up to 6× over a CPU and 42× higher energy efficiency. We have also shown how fixed-point arithmetic translates to a 2× **faster convergence**, with no significant accuracy loss.

- In Chapter 4 we extended our SpMV hardware design to develop the first FPGA Top-K eigensolvers for unstructured sparse matrices. We integrated it into a complex mixed-precision pipeline and shown how to **leverage HBM** in sparse computations to partition the SpMV hardware design across multiple Compute Units (CUs). Our eigensolver achieved a 6.22× speedup versus a highly optimized CPU implementation while keeping high accuracy and 49× better power efficiency.
- In Chapter 5 we broadened the scope of our analysis by optimizing approximate similarity search on large sparse embedding tables. We improved our hardware design to support Top-K SpMV, improved its operational intensity with a **custom** sparse matrix **representation**, and developed a novel **approximation scheme** for Top-K SpMV. Through these contributions, our hardware design was 2.1× faster than a GPU with 20 % higher bandwidth, with 15× higher power-efficiency, proving the strength of FPGAs in approximate sparse computations.

## 6.1 Limitations and Future Directions

For each contribution presented in this work, there is undoubtedly still room left for growth and improvements. Moreover, each of these contributions opens further research directions that are worth mentioning.

### 6.1.1 A Reduced-Precision Streaming SpMV Hardware Design for Personalized PageRank on FPGA

The PPR hardware design in Chapter 3 represents our first attempt at accelerating graph analytics with sparse linear algebra and reduced-precision fixed-point arithmetic. Results in Chapter 3 have been achieved with a hardware design that uses DDR4 off-chip memory to store the graph topology, instead of HBM. As such, we could achieve significant speedups against a CPU implementation that also employs DDR4 and offers similar maximum memory bandwidth. However, we expect HBM-based architectures to provide even better performance, given how memory-intensive SpMV and PPR are. Moreover, our hardware design required on-chip memory such as URAM to provide low-latency access to PPR values, limiting the maximum size of input graphs. To solve these issues, we optimized our SpMV core to use HBM in Chapter 4, greatly improving its performance and adding support for larger matrices.

Another remarkable finding is that reduced-precision fixed-point arithmetic increases convergence speed, with no significant accuracy loss. This

result opens the door to further research in this field, and we are already observing a general trend of moving towards lower and lower precision arithmetic. It is not unlikely that other architectures, such as CPUs and Graphics Processing Units (GPUs) will provide better support for lower-precision fixed-point arithmetic. Studying the impact of mixed-precision arithmetic in numerical algorithms and recommender systems, possibly in the context of different hardware architectures, is certainly a research avenue worth pursuing.

### 6.1.2 Solving Large Top-K Graph Eigenproblems with a Memory and Compute-optimized FPGA Design

The Top-K sparse eigensolver presented in Chapter 4 is a greatly improved variant of the hardware design in Chapter 3. By exploiting HBM, we lifted most limitations of the original SpMV design, such as the inability to process partitioned matrices, or the limited scale of input graphs.

Our hardware design assumes that the input matrix can be stored in HBM memory, limiting its applicability to web-scale graphs with trillions of edges. That said, the amount of HBM in accelerator devices such as GPUs is growing steadily, with recent Nvidia Tesla A100 offering $80\,\mathrm{GB}$ of HBM (10× more than the Xilinx Alveo U280 FPGA accelerator card). As such, we believe that the constraint of HBM size will become less relevant in the future. Partitioning the computation to multiple FPGAs might be a way to support extremely large matrices. Multi-device support is, however, not easy to achieve due to the iterative nature of the Lanczos algorithm. One would have to partition SpMV over multiple devices, with each device operating on a subset of rows and results of each aggregation being transferred through the limited bandwidth of PCIe. Our current work assumes matrices containing real values. Supporting non-Hermitian matrices, which are encountered in engineering applications but are less common in recommender systems and graph analytics, requires implementing the full Implicitly Restarted Arnoldi Method. While we have observed that our approximate eigensolver can compute accurate results, we have not analyzed its performance in an end-to-end pipeline for spectral analysis (e.g. inside spectral clustering). We expect our design to provide high-quality results still, and quantitative end-to-end analyses would confirm the hypothesis.

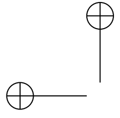
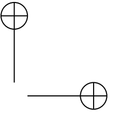
Finally, this work is a perfect case study for input-aware FPGA reconfiguration. Indeed, we observe how graphs with relatively low sparsity ($|E| \gg |V|$) benefit from more SpMV CUs, while highly sparse matrices ($|E| \approx |V|$) benefit from a larger partitioning factor in linear operations

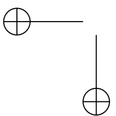
(e.g. vector normalization), with possibly a single SpMV CU. While our current hardware design is a solid compromise between these two extremes, it is interesting to evaluate the performance trade-offs on additional matrices and extend this analysis to other algorithms with similar characteristics, such as sparse Conjugate Gradient.

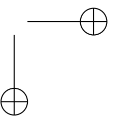
### 6.1.3 Scaling up HBM Efficiency of Top-K SpMV for Approximate Embedding Similarity on FPGAs

The hardware design presented in Chapter 5 is quite different from the ones in the previous chapters, although its starting point, the reduced-precision SpMV design, was originally the same. As the computation of Top-K SpMV is not iterative, the hardware design could easily be partitioned across multiple FPGAs. Such partitioning allows scaling to extremely large embedding tables without the communication overheads that limit the effectiveness of a multi-device Lanczos implementation. Indeed, the multi-device overheads for Top-K SpMV are limited to assembling the partial Top-K results, a step whose cost is insignificant compared to the rest of the computation, and we expect an almost-linear performance scaling with the adoption of multiple FPGAs. Moreover, increasing the number of partitions is beneficial to the accuracy of our approximation scheme, further cementing the advantages of such an implementation.

It is also interesting to study how our Block-Streaming CSR (BS-CSR) representation benefits other computations, such as PPR or the Lanczos algorithm. BS-CSR is certainly more beneficial to tall matrices than to the square matrices used to store graphs. Even in the case of square matrices, BS-CSR provides up to 50 % better operational intensity over a naïve COO representation. Finally, we focused our current study on multiplication between a sparse matrix and a dense vector. However, recommender systems often employ one-hot encoded feature vectors to encode categorical features (e.g. users' categories). Such vectors can effectively be implemented as arrays of bits. We expect our implementation to provide extremely high performance when working with arrays of bits and possibly become compute-bound as a single BS-CSR data-packet could contain more than 30 non-zeros, 2× our current reduced-precision fixed-point design.



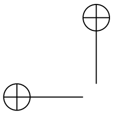
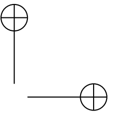


# List of Figures

1.1 Example of Sparse Matrix-Vector Multiplication (SpMV), for a sparse matrix $A$ represented as COO. . . . . . . . . . . 2
1.2 Scaling of peak compute power (FLOPS), memory bandwidth (GB/s) and interconnection bandwidth (GB/s) from 1996 to today . . . . . . . . . . . . . . . . . . . . . . . . . . 3
1.3 Structure of this thesis, with the main contributions of each chapter. . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

2.1 Examples of sparse matrices the SuiteSparse collection . . . 13
2.2 Sparse matrix representations for the constellation of Hercules. 16
2.3 Block diagonal matrix. Each of the $B$ square sub-matrices $\mathrm{A_i}$ is dense. . . . . . . . . . . . . . . . . . . . . . . . . . 19
2.4 Floating-point and fixed-point representations of 42.42. . . . 25
2.5 ⓐ Representation of a GPU architecture, with logical subdivision in grid, blocks and threads, as in the terminology of CUDA. ⓑ GPU `axpy` kernel written in CUDA, with subdivision in blocks and threads. . . . . . . . . . . . . . . . . . 36
2.6 Schematic representation of the Xilinx Alveo U280 FPGA accelerator card. . . . . . . . . . . . . . . . . . . . . . . . 46

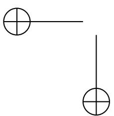
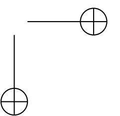
3.1 First three iteration of PageRank and Personalized PageRank for a small example graph, with $|V| = 5$ and $\alpha = 0.85$. . 56
3.2 COO representation of the transition matrix of a graph . . . 57
3.3 Block diagram of the Personalized PageRank algorithm, as implemented in our hardware design . . . . . . . . . . . . . 58



3.4 Block-diagram of our PPR SpMV hardware design. The *scatter* and *aggregation* units show the computation for a single vertex, but they are replicated to support $\kappa$ vertices. Large arrows represent a streaming transfer between units . 62
3.5 Speedup of our Personalized PageRank FPGA reduced-precision hardware design (y-axis) with respect to the CPU baseline and the FPGA floating-point baseline, for decreasing bit-widths (x-axis). . . . . . . . . . . . . . . . . . . . . . . . 68
3.6 Accuracy metrics measured on graphs with $2 \cdot 10^6$ edges, for increasing fixed-point bit-width. . . . . . . . . . . . . . . . 70
3.7 We compute on all graphs the metrics in Figure 3.6 and include all the additional metrics presented in Section 3.5.3. We show results aggregated on all graphs, for increasing bit-widths and number of top ranked vertices. . . . . . . . . . . 72
3.8 Precision value, for decreasing sparsity, and increasing number of iterations, measured for different fixed-point bit-widths. 72
3.9 Convergence error of PPR, for increasing number of iterations. 74
3.10 Speedup of our Personalized PageRank hardware design when accounting for the faster convergence. . . . . . . . . . . . 75

4.1 Top-K eigencomputation of a graph $G$, represented as a sparse matrix. . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
4.2 Steps of our Top-K sparse eigenproblem solver, which combines the Lanczos algorithm with a Systolic Array formulation for the Jacobi eigenvalue algorithm. . . . . . . . . . . . 82
4.3 Example of ($5 \times 5$) tridiagonal matrix, obtained as output of the Lanczos algorithm for $K = 5$. . . . . . . . . . . . . . 82
4.4 Operations performed by different processors in the Jacobi eigenvalue Systolic Array architecture. . . . . . . . . . . . . 84
4.5 Steps of the Jacobi eigenvalues computation using Systolic Arrays. . . . . . . . . . . . . . . . . . . . . . . . . . . . 87
4.6 High-level architecture of our Top-K Sparse Eigencomputation FPGA design. . . . . . . . . . . . . . . . . . . . . . 89
4.7 Block diagram of one iterative SpMV CU. . . . . . . . . . . 91
4.8 Dense vector memory subsystem of our SpMV FPGA design. 93
4.9 Detail of the $sin(\theta)$ and $cos(\theta)$ propagation between Compute Units in the Jacobi Systolic Array architecture, for $K = 4$. 94
4.10 Graphical representation of the sparsity patterns of matrix used in the evaluation. . . . . . . . . . . . . . . . . . . . . 98

4.11 Speedup (higher is better) of our Top-K sparse eigensolver vs. the ARPACK multi-core CPU library. . . . . . . . . . . 99

4.12 ⓐ Relation between the number of matrix non-zero values and time to process a single value. ⓑ Speedup vs. CPU of our Systolic Array architecture for the Jacobi algorithm, for increasing number of eigenvalues $K$. . . . . . . . . . . . . . 100

4.13 Accuracy of our Top-K sparse eigensolver, in terms of orthogonality and reconstruction error, for increasing $K$. . . . 101

5.1 Top-K Sparse matrix-vector multiplication (Top-K SpMV) between a sparse matrix $\mathrm{A}$ (in our case, a collection of sparse embeddings) and a dense vector $\mathrm{x}$ (a dense embedding), with notation as in Section 5.3. . . . . . . . . . . . . . . . . . . 107

5.2 Simplified approximation scheme for Top-K SpMV. No errors occur if all partitions have less than $\mathrm{k}$ Top-K values. . . 109

5.3 Graphical representation of our approximation scheme for Top-K SpMV. . . . . . . . . . . . . . . . . . . . . . . . 111

5.4 Naïve Coordinate (COO) representations of data packets. Even with reduced-precision arithmetic, each packet contains significant redundancy. . . . . . . . . . . . . . . . . . 113

5.5 Graphical representation of BSCSR data packets, showing drastically higher information density compared to COO. . . 114

5.6 Simplified block diagram of a single core of our Top-K SpMV hardware design, for BS-CSR with 5 non-zero `y` and `val` entries per packet. . . . . . . . . . . . . . . . . . . . . . . 116

5.7 Block diagram of our multi-core Top-K SpMV, for BS-CSR with 5 non-zero `y` and `val` entries per packet. . . . . . . . . 117

5.8 Execution time speedup on Top-K SpMV of our FPGA designs vs. GPU and CPU. . . . . . . . . . . . . . . . . . . . 123

5.9 Roofline model of our Top-K SpMV architecture. ⓐ Operational intensity increase with BS-CSR. ⓑ Comparison of FPGA vs. CPU and GPU. Thanks to BS-CSR, we provide the highest operational intensity and the best performance. . 124

5.10 Top-K SpMV accuracy *(higher is better)* for different architectures and types of reduced-precision arithmetic, aggregated on matrices with the same size. . . . . . . . . . . . 126

5.11 Top-K SpMV accuracy *(higher is better)* for different architectures and types of reduced-precision arithmetic, divided by matrices with uniform non-zeros distribution and the GloVe dataset ⓐ and matrices with Γ non-zeros distribution ⓑ. . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 128

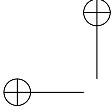
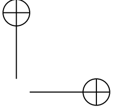
# List of Tables

2.1 Storage size, traversal cost and non-zero lookup cost of conventional sparse matrix representations. Assume a sparse matrix $A \in \mathbb{R}^{N \times M}$, with $nnz$ non-zeros, and $B$ dense blocks in the case of BSR. . . . . . . . . . . . . . . . . . . . . . . 17
2.2 Summary statistics of IEEE 754 floating-point and fixed-point representations. Floating-point values are retrieved from Hopkins et al. [85]. . . . . . . . . . . . . . . . . . . . 26

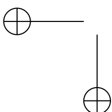
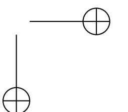
3.1 Summary of graph datasets used in the evaluation . . . . . . 66
3.2 Resource usage, power consumption for selected bit-widths of our PPR design. . . . . . . . . . . . . . . . . . . . . . . 66

4.1 Resource usage and clock frequency in our FPGA hardware design, divided by algorithm. We also report the total amount of resources available to the FPGA. . . . . . . . . . . . . . 96
4.2 Matrices/graphs in the evaluation, sorted by number of edges/non-zero entries (in millions). For each matrix, we report the memory footprint when stored as COO. . . . . . . . . . . 97

5.1 Estimated precision of Top-K indices for increasing number of partitions. We average the results of 1000 tests. . . . . . . 112
5.2 Resource usage, clock frequency and power consumption of our architecture. Best values in bold. We report the clock frequencies of the designs in Parravicini et al. [155] and the best frequencies obtained after further optimizations. . . . . 121

5.3 Matrices in the evaluation, with $M = 512$ and $1024$, and memory occupation using BS-CSR as in Figure 5.5. *Distribution* controls the number of non-zeros per row. . . . . . . 122
5.4 Execution time of our Top-K SpMV FPGA hardware designs with different clock frequencies. . . . . . . . . . . . . 122

# List of Acronyms

**A**

ASIC Application-Specific Integrated Circuits. 33, 37

AXI Advanced eXtensible Interface. 44, 45, 48

**B**

BLAS Basic Linear Algebra Subprograms. 14

BRAM Block RAM. 38, 44, 61, 62, 67, 95

BS-CSR Block-Streaming CSR. 19, 107, 109, 114–122, 124–127, 129, 135, 139, 142

BSR Block Compressed Row. 19

**C**

CLB Configurable Logic Block. 38

COO Coordinate. 6, 17–21, 41, 43, 52, 57, 63, 66, 74, 76, 91, 92, 113–115, 121, 125–127, 139

CORDIC Coordinate Rotation Digital Computer. 95

CPU Central Processing Unit. 33–35, 37, 43, 48

CSC Compressed Sparse Column. 18, 57

CSR Compressed Sparse Row. 18–21, 33, 34, 91, 108, 113–115, 126, 127

CU Compute Unit. 7, 8, 41, 79, 85–88, 91, 92, 94, 95, 100, 103, 133–135, 138

**D**

| | |
|---|---|
| DCG | Discounted Cumulative Gain. 31, 71 |
| DDR | Double Data Rate. 40, 44, 106, 109 |
| DDR4 | Double Data Rate 4. 58, 59, 61, 64, 69, 96 |
| DL | Deep Learning. 1, 2, 12, 14, 29, 33, 37, 41, 47–49, 76 |
| DRAM | Dynamic Random-Access Memory. 42, 43, 46, 54, 58, 61, 64, 65 |
| DSA | Domain-Specific Architecture. 33, 40, 46–49 |
| DSL | Domain-Specific Language. 53 |
| DSP | Digital Signal Processor. 38, 67, 95 |

**F**

| | |
|---|---|
| FEM | Finite Element Method. 18, 40 |
| FF | Flip Flop. 38, 39, 96 |
| FPGA | Field-Programmable Gate Array. 2–8, 19, 21, 29, 33–35, 37–45, 47–49, 52–54, 58, 59, 61, 62, 64–69, 71, 73, 74, 76, 79, 80, 85, 86, 88, 89, 92–97, 99–103, 106–110, 113, 115, 118, 120–126, 129, 132–135, 138, 139, 142 |
| FSM | Finite-state machine. 60 |

**G**

| | |
|---|---|
| GPCPU | General-purpose Central Processing Unit. 33, 35 |
| GPU | Graphics Processing Unit. 1, 2, 4, 5, 12, 19, 29, 34–37, 39, 40, 42–44, 46–49, 53, 54, 76, 81, 101, 106–108, 114, 120, 122–127, 129, 134, 139 |

**H**

| | |
|---|---|
| HBM | High Bandwidth Memory. 3–8, 37, 40–46, 48, 49, 52, 61, 64, 69, 76, 79, 88–94, 100, 101, 106, 107, 109, 110, 114–118, 120–122, 124–126, 129, 132–134 |
| HDL | Hardware Description Language. 38–40 |
| HLS | High-Level Synthesis. 39, 40 |

| | |
|---|---|
| HP | Half-Precision. 108, 120, 127 |
| HPCG | High-Performance Conjugate Gradient. 35 |

**I**

| | |
|---|---|
| IDCG | Ideal Discounted Cumulative Gain. 31 |
| II | Initiation Interval. 41, 45, 118 |
| IR | Information Retrieval. 12, 29, 31, 35, 71, 106, 113, 129 |
| IRAM | Implicitly Restarted Arnoldi Method. 81, 96 |
| ISA | Instruction Set Architecture. 46 |

**K**

| | |
|---|---|
| KNN | K-Nearest Neighbors. 108 |

**L**

| | |
|---|---|
| LSH | Locality-Sensitive Hashing. 108 |
| LUT | Look-Up Table. 38, 69, 110 |

**M**

| | |
|---|---|
| MAE | Mean Absolute Error. 31 |
| ML | Machine Learning. 35, 53 |
| MSHR | Miss Status Holding Register. 34 |

**N**

| | |
|---|---|
| NDCG | Normalized Discounted Cumulative Gain. 31, 32, 71, 76, 127 |

**P**

| | |
|---|---|
| PCIe | Peripheral Component Interconnect Express. 42, 43, 64, 103, 120, 134 |
| PE | Processing Element. 87 |
| PIM | Processing in Memory. 46, 47, 49 |
| PPR | Personalized PageRank. 6, 9, 21, 22, 43, 44, 52–67, 69, 71, 73, 74, 76, 132, 133, 135 |
| PR | PageRank. 18, 22, 26, 43, 52–57, 61, 62, 64 |

**R**

| | |
|---|---|
| RAW | Read-After-Write. 60 |

| | |
|---|---|
| RMSE | Root Mean Square Error. 31 |

**S**

| | |
|---|---|
| SA | Systolic Array. 80, 81, 86, 87, 93–95, 97, 100 |
| SLA | Service-Level Agreement. 48 |
| SLR | Super Logic Region. 45, 88, 94, 97 |
| SM | Stream Multiprocessor. 35, 36 |
| SP | Stream Processor. 35 |
| Sparse BLAS | Sparse Basic Linear Algebra Subprograms. 14, 33 |
| SpGEMM | Sparse Matrix-Matrix Multiplication. 15, 46 |
| SpMM | Sparse Matrix-Dense Matrix Multiplication. 14, 15 |
| SpMV | Sparse Matrix-Vector Multiplication. 3, 6–9, 14, 15, 17, 19–21, 29, 34, 35, 37, 39–43, 46, 52–55, 58–62, 64, 74, 76, 79, 81, 84–86, 88–93, 100–103, 108, 109, 111, 115, 116, 118, 120, 123, 126, 132–135, 138 |

**T**

| | |
|---|---|
| TDP | Thermal Design Power. 42 |
| Top-K SpMV | Top-K Sparse matrix-vector multiplication. 52, 103, 106–110, 115–118, 120, 122–126, 128, 129, 139, 140, 142 |

**U**

| | |
|---|---|
| URAM | UltraRAM. 6, 44, 59–62, 64, 65, 67, 69, 76, 92, 93, 106, 110, 116, 117, 126, 132, 133 |
| UVM | Unified Virtual Memory. 34 |

# Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283, 2016.

[2] Ahmad Abdelfattah, Hartwig Anzt, Erik G Boman, Erin Carson, Terry Cojean, Jack Dongarra, Mark Gates, Thomas Grützmacher, Nicholas J Higham, Sherry Li, et al. A survey of numerical methods utilizing mixed precision arithmetic. *arXiv preprint arXiv:2007.06674*, 2020.

[3] E. Adams, S. Venkatachalam, and S. Ko. Energy-Efficient Approximate MAC Unit. In *2019 IEEE ISCAS*, pages 1–4, 2019.

[4] Michal Aharon, Michael Elad, and Alfred Bruckstein. K-svd: An algorithm for designing overcomplete dictionaries for sparse representation. *IEEE Transactions on signal processing*, 54(11):4311–4322, 2006.

[5] Khalid Ahmad, Hari Sundar, and Mary Hall. Data-driven Mixed Precision Sparse Matrix Vector Multiplication for GPUs. *ACM Transactions on Architecture and Code Optimization (TACO)*, 16(4):1–24, 2019.

[6] Hasan Metin Aktulga, Chao Yang, Esmond G Ng, Pieter Maris, and James P Vary. Topology-aware mappings for large-scale eigenvalue problems. In *European Conference on Parallel Processing*, pages 830–842. Springer, 2012.

[7] Josh Alman and Virginia Vassilevska Williams. A refined laser method and faster matrix multiplication. In *Proceedings of the 2021 ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 522–539. SIAM, 2021.

[8] Amazon. Amazon EC2 F1 Instances. `aws.amazon.com/ec2/instance-types/f1/`. Retrieved 2021-07-25.

[9] HT Anson, Gary CT Chow, Qiwei Jin, David B Thomas, and Wayne Luk. Optimising performance of quadrature methods with reduced precision. In *International Symposium on Applied Reconfigurable Computing*, pages 251–263. Springer, 2012.

[10] Hartwig Anzt, Terry Cojean, Goran Flegar, Fritz Göbel, Thomas Grützmacher, Pratik Nayak, Tobias Ribizel, Yuhsiang Mike Tsai, and Enrique S Quintana-Ortí. Ginkgo: A modern linear operator algebra framework for high performance computing. *arXiv preprint arXiv:2006.16852*, 2020.

[11] Hartwig Anzt, William Sawyer, Stanimire Tomov, Piotr Luszczek, Ichitaro Yamazaki, and Jack Dongarra. Optimizing Krylov Subspace Solvers on Graphics Processing Units. In *Fourth International Workshop on Accelerators and Hybrid Exascale Systems (AsHES), IPDPS 2014*, Phoenix, AZ, 05-2014 2014. IEEE, IEEE.

[12] Jeremy Appleyard and Scott Yokim. Programming Tensor Cores in CUDA 9. `developer.nvidia.com/blog/programming-tensor-cores-cuda-9`, 2017. Retrieved 2021-07-29.

[13] Bahar Asgari, Ramyad Hadidi, Jiashen Cao, Sung-Kyu Lim, Hyesoon Kim, et al. Fafnir: Accelerating sparse gathering by using efficient near-memory intelligent reduction. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 908–920. IEEE, 2021.

[14] Bahar Asgari, Ramyad Hadidi, and Hyesoon Kim. Proposing a Fast and Scalable Systolic Array for Matrix Multiplication. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 204–204. IEEE, 2020.

[15] Semih Aslan, Sufeng Niu, and Jafar Saniie. Fpga implementation of fast QR decomposition based on givens rotation. *Midwest Symposium on Circuits and Systems*, pages 470–473, 08 2012.

[16] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. Ann-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms, 2018.

[17] Francis R Bach and Michael I Jordan. Learning spectral clustering, with application to speech separation. *Journal of Machine Learning Research*, 7(Oct):1963–2001, 2006.

[18] Bahman Bahmani, Abdur Chowdhury, and Ashish Goel. Fast incremental and personalized PageRank. *Proceedings of the VLDB Endowment*, 4(3):173–184, 2010.

[19] ING Bank. sparse_dot_topn. "`github.com/ing-bank/sparse_dot_topn`", 2017.

[20] Ziv Bar-Yossef and Li-Tal Mashiach. Local approximation of PageRank and reverse PageRank. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 279–288. ACM, 2008.

[21] Nathan Bell and Jared Hoberock. Thrust: A productivity-oriented library for CUDA. In *GPU computing gems Jade edition*, pages 359–371. Elsevier, 2012.

[22] Gábor Berend. Sparse Coding of Neural Word Embeddings for Multilingual Sequence Labeling. *Transactions of the Association for Computational Linguistics*, 5:247–261, 2017.

[23] Paolo Boldi, Massimo Santini, and Sebastiano Vigna. Pagerank: functional dependencies. *ACM Transactions on Information Systems (TOIS)*, 27(4):1–23, 2009.

[24] Christian Borgs, Michael Brautbar, Jennifer Chayes, and Shang-Hua Teng. A sublinear time algorithm for PageRank computations. In *International Workshop on Algorithms and Models for the Web-Graph*, pages 41–53. Springer, 2012.

[25] Richard P Brent, Franklin T Luk, et al. The solution of singular-value and eigenvalue problems on systolic arrays. In *Mathematical Programming and Numerical Analysis Workshop*, pages 38–64. Centre for Mathematics and its Applications, Mathematical Sciences Institute, 1984.

[26] Kurt Bryan and Tanya Leise. The $25,000,000,000 eigenvector: The linear algebra behind Google. *SIAM review*, 48(3):569–581, 2006.

[27] Aydın Buluç and John R Gilbert. The Combinatorial BLAS: Design, implementation, and applications. *The International Journal of High Performance Computing Applications*, 25(4):496–509, 2011.

[28] Aydin Buluç, Tim Mattson, Scott McMillan, José Moreira, and Carl Yang. Design of the GraphBLAS API for C. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 643–652. IEEE, 2017.

[29] Alwyn Burger, Patrick Urban, Jayson Boubin, and Gregor Schiele. An Architecture for Solving the Eigenvalue Problem on Embedded FPGAs. In André Brinkmann, Wolfgang Karl, Stefan Lankes, Sven Tomforde, Thilo Pionteck, and Carsten Trinitis, editors, *Architecture of Computing Systems – ARCS 2020*, pages 32–43, Cham, 2020. Springer International Publishing.

[30] Pavel Burovskiy, Stephen Girdlestone, Craig Davies, Spencer Sherwin, and Wayne Luk. Dataflow acceleration of Krylov subspace sparse banded problems. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–6. IEEE, 2014.

[31] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. Parallel tiled QR factorization for multicore architectures. *Concurrency and Computation: Practice and Experience*, 20(13):1573–1590, 2008.

[32] Daniela Calvetti, Lothar Reichel, and Danny Chris Sorensen. An implicitly restarted Lanczos method for large symmetric eigenvalue problems. *Electronic Transactions on Numerical Analysis*, 2(1):21, 1994.

[33] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A cloud-scale acceleration architecture. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–13. IEEE, 2016.

[34] Lung-Sheng Chien. Jacobi-based Eigenvalue Solver on GPU. "on-demand.gputechconf.com/gtc/2017/presentation/s7121-lung-sheng-chien-jacobi-based-eigenvalue-solver.pdf", 2017. Retrieved on 2021-08-12.

[35] Jungwook Choi, Swagath Venkataramani, Vijayalakshmi Srinivasan, Kailash Gopalakrishnan, Zhuo Wang, and Pierce Chuang. Accurate and Efficient 2-bit Quantized Neural Networks. In *MLSys*, 2019.

[36] Young-kyu Choi, Yuze Chi, Jie Wang, Licheng Guo, and Jason Cong. When HLS Meets FPGA HBM: Benchmarking and Bandwidth Optimization. *arXiv preprint arXiv:2010.06075*, 2020.

[37] Jack Choquette, Wishwesh Gandhi, Olivier Giroux, Nick Stam, and Ronny Krashinsky. Nvidia A100 tensor core GPU: Performance and innovation. *IEEE Micro*, 41(2):29–35, 2021.

[38] Alain Cosnuau. Computation on GPU of Eigenvalues and Eigenvectors of a Large Number of Small Hermitian Matrices. *Procedia Computer Science*, 29:800–810, 12 2014.

[39] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. Are we really making much progress? A worrying analysis of recent neural recommendation approaches. In *Proceedings of the 13th ACM Conference on Recommender Systems*, pages 101–109, 2019.

[40] Dimitrios Danopoulos, Christoforos Kachris, and Dimitrios Soudris. Fpga Acceleration of Approximate KNN Indexing on High-Dimensional Vectors, July 2019.

[41] Shail Dave, Riyadh Baghdadi, Tony Nowatzki, Sasikanth Avancha, Aviral Shrivastava, and Baoxin Li. Hardware acceleration of sparse and irregular tensor computations of ML models: A survey and insights. *Proceedings of the IEEE*, 2021.

[42] Timothy A Davis. Algorithm 1000: SuiteSparse: GraphBLAS: Graph algorithms in the language of sparse linear algebra. *ACM Transactions on Mathematical Software (TOMS)*, 45(4):1–25, 2019.

[43] Timothy A Davis and Yifan Hu. The University of Florida sparse matrix collection. *ACM Transactions on Mathematical Software (TOMS)*, 38(1):1–25, 2011.

[44] Emanuele Del Sozzo. *On how to effectively target FPGAs from domain specific tools*. PhD thesis, Politecnico di Milano, 2019.

[45] Michael DeLorimier and André DeHon. Floating-point sparse matrix-vector multiply for FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 75–85, 2005.

[46] Weibing Deng, Wei Li, Xu Cai, and Qiuping A Wang. The exponential degree distribution in complex networks: Non-equilibrium network theory, numerical simulation and empirical data. *Physica A: Statistical Mechanics and its Applications*, 390(8):1481–1485, 2011.

[47] P. Desai, S. Aslan, and J. Saniie. Fpga implementation of Gram-Schmidt QR decomposition using high level synthesis. In *2017 IEEE International Conference on Electro Information Technology (EIT)*, pages 482–487, 2017.

[48] Jens Domke, Emil Vatai, Aleksandr Drozd, Peng ChenT, Yosuke Oyama, Lingqi Zhang, Shweta Salaria, Daichi Mukunoki, Artur Podobas, Mohamed WahibT, et al. Matrix engines for high performance computing: A paragon of performance or grasping at straws? In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1056–1065. IEEE, 2021.

[49] Jérôme Dubois, Christophe Calvin, and Serge Petiton. Accelerating the explicitly restarted Arnoldi method with GPUs using an autotuned matrix vector product. *SIAM Journal on Scientific Computing*, 33(5):3010–3019, 2011.

[50] Iain S Duff. A survey of sparse matrix research. *Proceedings of the IEEE*, 65(4):500–535, 1977.

[51] Iain S Duff, Michael A Heroux, and Roldan Pozo. An overview of the sparse basic linear algebra subprograms: The new standard from the BLAS technical forum. *ACM Transactions on Mathematical Software (TOMS)*, 28(2):239–267, 2002.

[52] David Ediger, Rob McColl, Jason Riedy, and David A Bader. Stinger: High performance data structure for streaming graphs. In *2012 IEEE Conference on High Performance Extreme Computing*, pages 1–5. IEEE, 2012.

[53] Nikolay M Evstigneev. Implementation of implicitly restarted Arnoldi method on multiGPU architecture with application to fluid dynamics problems. In *International Conference on Parallel Computational Technologies*, pages 301–316. Springer, 2017.

[54] Priyank Faldu, Jeff Diamond, and Boris Grot. A closer look at lightweight graph reordering. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 1–13. IEEE, 2019.

[55] Jian Fang, Yvo TB Mulder, Jan Hidders, Jinho Lee, and H Peter Hofstee. In-memory database acceleration on FPGAs: a survey. *The VLDB Journal*, 29(1):33–59, 2020.

[56] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021.

[57] Jeremy Fowers, Kalin Ovtcharov, Karin Strauss, Eric S Chung, and Greg Stitt. A high memory bandwidth fpga accelerator for sparse matrix-vector multiplication. In *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 36–43. IEEE, 2014.

[58] Cong Fu, Chao Xiang, Changxu Wang, and Deng Cai. Fast Approximate Nearest Neighbor Search with the Navigating Spreading-out Graph. *Proc. VLDB Endow.*, 12(5):461474, January 2019.

[59] Juan Fumero, Michel Steuwer, Lukas Stadler, and Christophe Dubach. Just-In-Time GPU Compilation for Interpreted Languages with Partial Evaluation. In *Proceedings of the 13th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '17, pages 60–73, New York, NY, USA, 2017. ACM.

[60] Jianhua Gao, Weixing Ji, Zhaonian Tan, and Yueyan Zhao. A systematic survey of general sparse matrix-matrix multiplication. *arXiv preprint arXiv:2002.11273*, 2020.

[61] Prasun Gera, Hyojong Kim, Piyush Sao, Hyesoon Kim, and David Bader. Traversing large graphs on GPUs with unified memory. *Proceedings of the VLDB Endowment*, 13(7):1119–1133, 2020.

[62] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630*, 2021.

[63] Amir Gholami, Michael Mahoney, Kurt Keutzer, Suresh Krishna, Ravi Krishna, Zhewei Yao, Zhen Dong, Sehoon Kim, Tianren Gao, Bohan Zhai Zhai, and Nrusimha Ani. Emerging AI Applications: Moving Beyond ResNet50 on ImageNet. "amirgholami.org/assets/talks/2021_02_Intel_ISAS.pdf", 2 2021. Retrieved 2021-08-24.

[64] Amir Gholami, Zhewei Yao, Sehoon Kim, Michael W. Mahoney, and Kurz Keutzer. Ai and Memory Wall. https://medium.com/riselab/ai-and-memory-wall-2cb4265cb0b8, 2021. Retrieved 2021-08-27.

[65] Gene H Golub and Richard Underwood. The block Lanczos method for computing eigenvalues. In *Mathematical software*, pages 361–377. Elsevier, 1977.

[66] Scott Grauer-Gray, William Killian, Robert Searles, and John Cavazos. Accelerating financial applications on the GPU. In *Proceedings of the 6th Workshop on General Purpose Processor Using Graphics Processing Units*, pages 127–136. ACM, 2013.

[67] Paul Grigoras. *Instance directed tuning for sparse matrix kernels on reconfigurable accelerators*. PhD thesis, Imperial College London, 2018.

[68] Paul Grigoras, Pavel Burovskiy, Eddie Hung, and Wayne Luk. Accelerating SpMV on FPGAs by compressing nonzero values. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 64–67. IEEE, 2015.

[69] Paul Grigoras, Pavel Burovskiy, and Wayne Luk. Cask: Open-source custom architectures for sparse kernels. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 179–184, 2016.

[70] Paul Grigoraş, Pavel Burovskiy, Wayne Luk, and Spencer Sherwin. Optimising sparse matrix vector multiplication for large scale fem problems on fpga. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–9. IEEE, 2016.

[71] Gaël Guennebaud, Benoit Jacob, et al. Eigen. *URl: http://eigen. tuxfamily. org*, 3, 2010.

[72] Wentian Guo, Yuchen Li, Mo Sha, and Kian-Lee Tan. Parallel personalized PageRank on dynamic graphs. *Proceedings of the VLDB Endowment*, 11(1):93–106, 2017.

[73] K. D. Gupta, M. Wajid, R. Muzammil, and S. J. Arif. Hardware Architecture for Eigenvalues Computation using the Modified Jacobi Algorithm on FPGA. In *2019 5th International Conference on Signal Processing, Computing and Control (ISPCC)*, pages 243–246, 2019.

[74] Suyog Gupta, Ankur Agrawal, Kailash Gopalakrishnan, and Pritish Narayanan. Deep learning with limited numerical precision. In *International conference on machine learning*, pages 1737–1746. PMLR, 2015.

[75] Udit Gupta, Samuel Hsia, Vikram Saraph, Xiaodong Wang, Brandon Reagen, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995. IEEE, 2020.

[76] Udit Gupta, Carole-Jean Wu, Xiaodong Wang, Maxim Naumov, Brandon Reagen, David Brooks, Bradford Cottel, Kim Hazelwood, Mark Hempstead, Bill Jia, et al. The architectural implications of Facebook's DNN-based personalized recommendation. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 488–501. IEEE, 2020.

[77] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978.

[78] Fred G Gustavson, Werner Liniger, and R Willoughby. Symbolic generation of an optimal Crout algorithm for sparse systems of linear equations. *Journal of the ACM (JACM)*, 17(1):87–109, 1970.

[79] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 603–613. IEEE, 2018.

[80] Song Han, Xingyu Liu, Huizi Mao, Jing Pu, Ardavan Pedram, Mark A Horowitz, and William J Dally. Eie: Efficient inference engine on compressed deep neural network. *ACM SIGARCH Computer Architecture News*, 44(3):243–254, 2016.

[81] Mark Harris. Write flexible kernels with grid-stride loops. `developer.nvidia.com/blog/cuda-pro-tip-write-flexible-kernels-grid-stride-loops`, 2013-04-13. Retrieved on 2021-05-23.

[82] V Hernandez, JE Roman, A Tomas, and V Vidal. A survey of software for sparse eigenvalue problems. *Universitat Politecnica De Valencia, SLEPs technical report STR-6*, 2009.

[83] Torsten Hoefler, Dan Alistarh, Tal Ben-Nun, Nikoli Dryden, and Alexandra Peste. Sparsity in Deep Learning: Pruning and growth for efficient inference and training in neural networks. *arXiv preprint arXiv:2102.00554*, 2021.

[84] Sungpack Hong, Hassan Chafi, Edic Sedlar, and Kunle Olukotun. Green-Marl: a DSL for easy and efficient graph analysis. *ACM SIGARCH Computer Architecture News*, 40(1):349–362, 2012.

[85] Michael Hopkins, Mantas Mikaitis, Dave R Lester, and Steve Furber. Stochastic rounding and reduced-precision fixed-point arithmetic for solving neural ordinary differential equations. *Philosophical Transactions of the Royal Society A*, 378(2166):20190052, 2020.

[86] Guanhao Hou, Xingguang Chen, Sibo Wang, and Zhewei Wei. Massively Parallel Algorithms for Personalized PageRank. *PROCEEDINGS OF THE VLDB ENDOWMENT*, 14(9):1668–1680, 2021.

[87] Jui-Ting Huang, Ashish Sharma, Shuying Sun, Li Xia, David Zhang, Philip Pronin, Janani Padmanabhan, Giuseppe Ottaviano, and Linjun Yang. Embedding-based retrieval in Facebook search. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 2553–2561, 2020.

[88] Intel. Intel Math Kernel Library. `software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html#gs.8w0k0g`.

[89] Ilse CF Ipsen and Teresa M Selee. Pagerank computation, with special attention to dangling nodes. *SIAM Journal on Matrix Analysis and Applications*, 29(4):1281–1296, 2008.

[90] Abhishek Kumar Jain, Hossein Omidian, Henri Fraisse, Mansimran Benipal, Lisa Liu, and Dinesh Gaitonde. A Domain-Specific Architecture for Accelerating Sparse Matrix Vector Multiplication on FPGAs. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2020.

[91] Paras Jain, Ajay Jain, Aniruddha Nrusimha, Amir Gholami, Pieter Abbeel, Kurt Keutzer, Ion Stoica, and Joseph E Gonzalez. Checkmate: Breaking the memory wall with optimal tensor rematerialization. *arXiv preprint arXiv:1910.02653*, 2019.

[92] Wole Jaiyeoba and Kevin Skadron. Graphtinker: A high performance data structure for dynamic graph processing. In *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1030–1041. IEEE, 2019.

[93] Jun-Woo Jang, Sehwan Lee, Dongyoung Kim, Hyunsun Park, Ali Shafiee Ardestani, Yeongjae Choi, Channoh Kim, Yoojin Kim, Hyeongseok Yu, Hamzah Abdel-Aziz, et al. Sparsity-Aware and Re-configurable NPU Architecture for Samsung Flagship Mobile SoC. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 15–28. IEEE, 2021.

[94] Kalervo Järvelin and Jaana Kekäläinen. Cumulated gain-based evaluation of IR techniques. *ACM Transactions on Information Systems (TOIS)*, 20(4):422–446, 2002.

[95] Jeff Johnson, Matthijs Douze, and Hervé Jégou. Billion-scale similarity search with GPUs. *IEEE Transactions on Big Data*, 2019.

[96] Norman Jouppi, Cliff Young, Nishant Patil, and David Patterson. Motivation for and evaluation of the first tensor processing unit. *IEEE Micro*, 38(3):10–19, 2018.

[97] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *Proceedings of the 44th annual international symposium on computer architecture*, pages 1–12, 2017.

[98] Sepandar D Kamvar, Taher H Haveliwala, Christopher D Manning, and Gene H Golub. Extrapolation methods for accelerating PageRank computations. In *Proceedings of the 12th international conference on World Wide Web*, pages 261–270. ACM, 2003.

[99] Konstantinos Kanellopoulos, Nandita Vijaykumar, Christina Giannoula, Roknoddin Azizi, Skanda Koppula, Nika Mansouri Ghiasi, Taha Shahroodi, Juan Gomez Luna, and Onur Mutlu. Smash: Co-designing software compression and hardware-accelerated indexing for efficient sparse matrix operations. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 600–614, 2019.

[100] Kaan Kara, Christoph Hagleitner, Dionysios Diamantopoulos, Dimitris Syrivelis, and Gustavo Alonso. High Bandwidth Memory on FPGAs: A Data Analytics Perspective. *arXiv preprint arXiv:2004.01635*, 2020.

[101] Jeremy Kepner, Peter Aaltonen, David Bader, Aydin Buluç, Franz Franchetti, John Gilbert, Dylan Hutchison, Manoj Kumar, Andrew Lumsdaine, Henning Meyerhenke, et al. Mathematical foundations of the GraphBLAS. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–9. IEEE, 2016.

[102] Srinidhi Kestur, John D Davis, and Eric S Chung. Towards a universal FPGA matrix-vector multiplication architecture. In *2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, pages 9–16. IEEE, 2012.

[103] James King, Thomas Gilray, Robert M Kirby, and Matthew Might. Dynamic sparse-matrix allocation on GPUs. In *International Conference on High Performance Computing*, pages 61–80. Springer, 2016.

[104] Andrew V Knyazev. Toward the optimal preconditioned eigensolver: Locally optimal block preconditioned conjugate gradient method. *SIAM journal on scientific computing*, 23(2):517–541, 2001.

[105] Christos Kotselidis, James Clarkson, Andrey Rodchenko, Andy Nisbet, John Mawer, and Mikel Luján. Heterogeneous Managed Runtime Systems: A Computer Vision Case Study. *SIGPLAN Not.*, pages 74–82, April 2017.

[106] Suresh Krishna and Ravi Krishna. Accelerating Recommender Systems via Hardware scale-in. *arXiv preprint arXiv:2009.05230*, 2020.

[107] Hsiang-Tsung Kung. Why systolic architectures? *IEEE computer*, 15(1):37–46, 1982.

[108] Fang-An Kuo, Matthew R Smith, Chih-Wei Hsieh, Chau-Yi Chou, and Jong-Shinn Wu. Gpu acceleration for general conservation equations and its application to several engineering problems. *Computers & Fluids*, 45(1):147–154, 2011.

[109] Youngeun Kwon, Yunjae Lee, and Minsoo Rhu. Tensordimm: A practical near-memory processing architecture for embeddings and tensor operations in deep learning. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 740–753, 2019.

[110] Kartik Lakhotia, Shreyas Singapura, Rajgopal Kannan, and Viktor Prasanna. Recall: Reordered cache aware locality based graph processing. In *2017 IEEE 24th International Conference on High Performance Computing (HiPC)*, pages 273–282. IEEE, 2017.

[111] Cornelius Lanczos. An iteration method for the solution of the eigenvalue problem of linear differential and integral operators. *Journal of Research of the National Bureau of Standards*, 1950.

[112] Martin Langhammer, Sergey Gribok, and Gregg Baeckler. High Density 8-Bit Multiplier Systolic Arrays For FPGA. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 84–92. IEEE, 2020.

[113] Amy N Langville and Carl D Meyer. Deeper inside PageRank. *Internet Mathematics*, 1(3):335–380, 2004.

[114] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic Linear Algebra Subprograms for Fortran Usage. *ACM Trans. Math. Softw.*, 5(3):308–323, September 1979.

[115] Chuck L Lawson, Richard J. Hanson, David R Kincaid, and Fred T. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software (TOMS)*, 5(3):308–323, 1979.

[116] Dongjin Lee, Takeo Hoshi, Tomohiro Sogabe, Yuto Miyatake, and Shao-Liang Zhang. Solution of the k-th eigenvalue problem in large-scale electronic structure calculations. *Journal of Computational Physics*, 371:618–632, 2018.

[117] Richard B Lehoucq, Danny C Sorensen, and Chao Yang. *ARPACK users' guide: solution of large-scale eigenvalue problems with implicitly restarted Arnoldi methods*. SIAM, 1998.

[118] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014.

[119] Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.

[120] Jun Li, Xiaoling Zheng, Yafeng Wu, and Deren Chen. A computational trust model in c2c e-commerce environment. In *2010 IEEE 7th International Conference on E-Business Engineering*, pages 244–249. IEEE, 2010.

[121] Shuang Liang, Shouyi Yin, Leibo Liu, Wayne Luk, and Shaojun Wei. Fp-BNN: Binarized neural network on FPGA. *Neurocomputing*, 275:1072–1086, 2018.

[122] Heng Liao, Jiajin Tu, Jing Xia, Hu Liu, Xiping Zhou, Honghui Yuan, and Yuxing Hu. Ascend: a Scalable and Unified Architecture for Ubiquitous Deep Neural Network Computing: Industry Track Paper. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 789–801. IEEE, 2021.

[123] Heng Liao, Jiajin Tu, Jing Xia, and Xiping Zhou. Davinci: A scalable architecture for neural network computing. In *2019 IEEE Hot Chips 31 Symposium (HCS)*, pages 1–44. IEEE Computer Society, 2019.

[124] Weifeng Liu and Brian Vinter. Csr5: An efficient storage format for cross-platform sparse matrix-vector multiplication. In *Proceedings of the 29th ACM on International Conference on Supercomputing*, pages 339–350, 2015.

[125] Yongchao Liu and Bertil Schmidt. Lightspmv: Faster CSR-based sparse matrix-vector multiplication on CUDA-enabled GPUs. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 82–89. IEEE, 2015.

[126] Alec Lu, Zhenman Fang, Weihua Liu, and Lesley Shannon. Demystifying the memory system of modern datacenter FPGAs for software programmers through microbenchmarking. In *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2021.

[127] Kejing Lu, Hongya Wang, Wei Wang, and Mineichi Kudo. Vhp: approximate nearest neighbor search via virtual hypersphere partitioning. *Proceedings of the VLDB Endowment*, 13(9):1443–1455, 2020.

[128] MAGMA. magma_slobpcg. `icl.cs.utk.edu/projectsfiles/magma/doxygen/group__magmasparse__ssyev.html`, 2020.

[129] Julien Mairal, Francis Bach, Jean Ponce, and Guillermo Sapiro. Online dictionary learning for sparse coding. In *Proceedings of the 26th annual international conference on machine learning*, pages 689–696, 2009.

[130] Stefano Markidis, Steven Wei Der Chien, Erwin Laure, Ivy Bo Peng, and Jeffrey S Vetter. Nvidia tensor core programmability, performance & precision. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 522–531. IEEE, 2018.

[131] Kristyn J Maschhoff and Danny C Sorensen. P_arpack: An efficient portable large scale eigenvalue package for distributed memory parallel architectures. In *International workshop on applied parallel computing*, pages 478–486. Springer, 1996.

[132] Seamas McGettrick, Dermot Geraghty, and Ciaran McElroy. An FPGA architecture for the PageRank eigenvector problem. In *2008 International Conference on Field Programmable Logic and Applications*, pages 523–526. IEEE, 2008.

[133] Paulius Micikevicius, Sharan Narang, Jonah Alben, Gregory Diamos, Erich Elsen, David Garcia, Boris Ginsburg, Michael Houston, Oleksii Kuchaiev, Ganesh Venkatesh, et al. Mixed precision training. *arXiv preprint arXiv:1710.03740*, 2017.

[134] Micron Technology, Inc. How Much Power Does Memory Use? `https://www.crucial.com/support/articles-faq-memory/how-much-power-does-memory-use`, 2019.

[135] Ioannis Mitliagkas, Michael Borokhovich, Alexandros G Dimakis, and Constantine Caramanis. Frogwild!: fast PageRank approximations on graph engines. *Proceedings of the VLDB Endowment*, 8(8):874–885, 2015.

[136] Sparsh Mittal. A survey of FPGA-based accelerators for convolutional neural networks. *Neural computing and applications*, 32(4):1109–1139, 2020.

[137] Mirko Myllykoski and Carl Christian Kjelgaard Mikkelsen. Task-based, GPU-accelerated and robust library for solving dense nonsymmetric eigenvalue problems. *Concurrency and Computation: Practice and Experience*, 33(11):e5915, 2021.

[138] Zoltán Nagy and Péter Szolgay. Configurable multilayer CNN-UM emulator on FPGA. *IEEE Transactions on Circuits and Systems I: Fundamental Theory and Applications*, 50(6):774–778, 2003.

[139] M Naumov, LS Chien, P Vandermersch, and U Kapasi. Cusparse library. In *GPU Technology Conference*, 2010.

[140] Maxim Naumov, John Kim, Dheevatsa Mudigere, Srinivas Sridharan, Xiaodong Wang, Whitney Zhao, Serhat Yilmaz, Changkyu Kim, Hector Yuen, Mustafa Ozdal, et al. Deep learning training in Facebook data centers: Design of scale-up and scale-out systems. *arXiv preprint arXiv:2003.09518*, 2020.

[141] Netlib. BLAS (Basic Linear Algebra Subprograms). `www.netlib.org/blas/`.

[142] Andrew Y Ng, Michael I Jordan, and Yair Weiss. On spectral clustering: Analysis and an algorithm. In *Advances in neural information processing systems*, pages 849–856, 2002.

[143] T. Nguyen, S. Williams, M. Siracusa, C. MacLean, and N. Wright D. Doerfler. The Performance and Energy Efficiency Potential of FPGAs in Scientific Computing. In *IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, 2020.

[144] Nimbix. Nimbix Xilinx Alveo Accelerator Cards. `www.nimbix.net/alveo`. Retrieved 2021-07-25.

[145] NIST. Sparse Basic Linear Algebra Subprograms (BLAS) Library. `math.nist.gov/spblas`.

[146] Nvidia. nvgraph. `docs.nvidia.com/cuda/nvgraph/index.html`, 2019.

[147] Nvidia. cusolver. `docs.nvidia.com/cuda/cusolver/index.html`, 2020.

[148] Nvidia. Nvidia A100 Tensor Core GPU. `www.nvidia.com/en-us/data-center/a100/`, 2020. Retrieved on 2021-08-21.

[149] Nvidia. Nvidia A100 Tensor Core GPU Architecture. `images.nvidia.com/aem-dam/en-zz/Solutions/data-center/nvidia-ampere-architecture-whitepaper.pdf`, 2020. Retrieved 2021-07-29.

[150] Nvidia. Nvidia Tesla A100 Data-sheet. `www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/a100/pdf/nvidia-a100-datasheet.pdf`, 2020. Retrieved on 2020-11-12.

[151] Nvidia. Cuda programming guide, thread hierarchy. `docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#thread-hierarchy`, 2020-10-27. Retrieved on 2021-05-23.

[152] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[153] Christopher C Paige. Computational variants of the Lanczos method for the eigenproblem. *IMA Journal of Applied Mathematics*, 10(3):373–381, 1972.

[154] Beresford N. Parlett. *The Symmetric Eigenvalue Problem*. Society for Industrial and Applied Mathematics, Philadelphia, 1998.

[155] Alberto Parravicini, Luca Giuseppe Cellamare, Marco Siracusa, and Marco D Santambrogio. Scaling up HBM Efficiency of Top-K SpMV for Approximate Embedding Similarity on FPGAs. In *To appear in Proceedings of the 58th Design Automation Conference (DAC)*, 2021.

[156] Alberto Parravicini, Arnaud Delamare, Marco Arnaboldi, and Marco D Santambrogio. DAG-based Scheduling with Resource Sharing for Multi-task Applications in a Polyglot GPU Runtime. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 111–120. IEEE, 2021.

[157] Alberto Parravicini, Rhicheek Patra, Davide B Bartolini, and Marco D Santambrogio. Fast and accurate entity linking via graph embedding. In *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*, pages 1–9, 2019.

[158] Alberto Parravicini, Francesco Sgherzi, and Marco D Santambrogio. A reduced-precision streaming SpMV architecture for Personalized PageRank on FPGA. In *2021 26th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 378–383. IEEE, 2021.

[159] Julian Pavon, Ivan Vargas Valdivieso, Adrián Barredo, Joan Marimon, Miquel Moreto, Francesc Moll, Osman Unsal, Mateo Valero, and Adrian Cristal. Via: A Smart Scratchpad for Vector Units with Application to Sparse Matrix Computations. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 921–934. IEEE, 2021.

[160] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global Vectors for Word Representation. In *Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, 2014.

[161] Pietro Perona and William Freeman. A factorization approach to grouping. In *European Conference on Computer Vision*, pages 655–670. Springer, 1998.

[162] Maria Pershina, Yifan He, and Ralph Grishman. Personalized Page Rank for named entity disambiguation. In *Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pages 238–243, 2015.

[163] David MW Powers. Applications and explanations of Zipfs law. In *New methods in language processing and computational natural language learning*, 1998.

[164] Andrew Putnam. Fpgas in the datacenter: Combining the worlds of hardware and software development. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 5–5, 2017.

[165] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*, pages 13–24. IEEE, 2014.

[166] Eric Qin, Ananda Samajdar, Hyoukjun Kwon, Vineet Nadella, Sudarshan Srinivasan, Dipankar Das, Bharat Kaul, and Tushar Krishna. Sigma: A sparse and irregular GEMM accelerator with flexible interconnects for DNN training. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 58–70. IEEE, 2020.

[167] Alfio Quarteroni, Riccardo Sacco, and Fausto Saleri. *Numerical mathematics*, volume 37. Springer Science & Business Media, 2010.

[168] Abid Rafique, George A Constantinides, and Nachiket Kapre. Communication optimization of iterative sparse matrix-vector multiply on GPUs and FPGAs. *IEEE Transactions on Parallel and Distributed Systems*, 26(1):24–34, 2014.

[169] Ran Rui, Hao Li, and Yi-Cheng Tu. Efficient join algorithms for large database tables in a multi-GPU environment. *Proceedings of the VLDB Endowment*, 14(4):708–720, 2020.

[170] H. Rutishauser. The Jacobi method for real symmetric matrices. *Numerische Mathematik*, 9(1):1–10, Nov 1966.

[171] Fazle Sadi. Accelerating Sparse Matrix Kernels with Co-Optimized Architecture. `kilthub.cmu.edu/articles/thesis/Accelerating_Sparse_Matrix_Kernels_with_Co-Optimized_Architecture/7583975/1`, Jan 2019.

[172] Fazle Sadi, Joe Sweeney, Tze Meng Low, James C. Hoe, Larry Pileggi, and Franz Franchetti. Efficient SpMV Operation for Large and Highly Sparse Matrices Using Scalable Multi-Way Merge Parallelization. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 347–358, New York, NY, USA, 2019. Association for Computing Machinery.

[173] Alan Said and Alejandro Bellogín. Replicable evaluation of recommender systems. In *Proceedings of the 9th ACM Conference on Recommender Systems*, pages 363–364, 2015.

[174] Charbel Sakr, Naigang Wang, Chia-Yu Chen, Jungwook Choi, Ankur Agrawal, Naresh Shanbhag, and Kailash Gopalakrishnan. Accumulation bit-width scaling for ultra-low precision training of deep networks. *arXiv preprint arXiv:1901.06588*, 2019.

[175] Jason Sanders and Edward Kandrot. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional, 2010.

[176] Conrad Sanderson and Ryan Curtin. Armadillo: a template-based C++ library for linear algebra. *Journal of Open Source Software*, 1(2):26, 2016.

[177] Francesco Sgherzi, Alberto Parravicini, Marco Siracusa, and Marco D Santambrogio. Solving Large Top-K Graph Eigenproblems with a Memory and Compute-optimized FPGA Design. In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 78–87. IEEE, 2021.

[178] Yi Shan, Tianji Wu, Yu Wang, Bo Wang, Zilong Wang, Ningyi Xu, and Huazhong Yang. Fpga and GPU implementation of large scale SpMV. In *2010 IEEE 8th Symposium on Application Specific Processors (SASP)*, pages 64–70. IEEE, 2010.

[179] Anil Shanbhag, Samuel Madden, and Xiangyao Yu. A study of the fundamental performance characteristics of GPUs and CPUs for database analytics. In *Proceedings of the 2020 ACM SIGMOD international conference on Management of data*, pages 1617–1632, 2020.

[180] Guy Shani and Asela Gunawardana. Evaluating recommendation systems. In *Recommender systems handbook*, pages 257–297. Springer, 2011.

[181] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[182] Jieming Shi, Renchi Yang, Tianyuan Jin, Xiaokui Xiao, and Yin Yang. Realtime top-k personalized PageRank over large graphs on GPUs. *Proceedings of the VLDB Endowment*, 13(1):15–28, 2019.

[183] Amit Singhal et al. Modern information retrieval: A brief overview. *IEEE Data Eng. Bull.*, 24(4):35–43, 2001.

[184] M. Siracusa, M. Rabozzi, E. Del Sozzo, L. Di Tucci, S. Williams, and M. D. Santambrogio. A CAD-based methodology to optimize HLS code via the Roofline model. In *2020 IEEE/ACM ICCAD*, 2020.

[185] Nitish Srivastava, Hanchen Jin, Shaden Smith, Hongbo Rong, David Albonesi, and Zhiru Zhang. Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 689–702. IEEE, 2020.

[186] Markus Steinberger, Andreas Derlery, Rhaleb Zayer, and Hans-Peter Seidel. How naive is naive SpMV on the GPU? In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–8. IEEE, 2016.

[187] Junqing Sun, Gregory D Peterson, and Olaf O Storaasli. High-performance mixed-precision linear solver for FPGAs. *IEEE Transactions on Computers*, 57(12):1614–1623, 2008.

[188] Xiao Sun, Naigang Wang, Chia-Yu Chen, Jiamin Ni, Ankur Agrawal, Xiaodong Cui, Swagath Venkataramani, Kaoutar El Maghraoui, Vijayalakshmi Srinivasan, and Kailash Gopalakrishnan. Ultra-Low Precision 4-bit Training of Deep Neural Networks. In *NeurIPS*, 2020.

[189] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks. *Synthesis Lectures on Computer Architecture*, 15(2):1–341, 2020.

[190] Fengqin Tang, Chunning Wang, Jinxia Su, and Yuanyuan Wang. Spectral clustering-based community detection using graph distance and node attributes. *Computational Statistics*, 35(1):69–94, 2020.

[191] Joel A Tropp and Anna C Gilbert. Signal recovery from random measurements via orthogonal matching pursuit. *IEEE Transactions on information theory*, 53(12):4655–4666, 2007.

[192] Frederick Tung, Alexander Wong, and David A Clausi. Enabling scalable spectral clustering for image segmentation. *Pattern Recognition*, 43(12):4069–4076, 2010.

[193] Yaman Umuroglu, Nicholas J Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. Finn: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 65–74, 2017.

[194] Yaman Umuroglu and Magnus Jahre. An energy efficient column-major backend for FPGA SpMV accelerators. In *2014 IEEE 32nd International Conference on Computer Design (ICCD)*, pages 432–439. IEEE, 2014.

[195] Yaman Umuroglu and Magnus Jahre. A vector caching scheme for streaming FPGA SpMV accelerators. In *International Symposium on Applied Reconfigurable Computing*, pages 15–26. Springer, 2015.

[196] Yaman Umuroglu and Magnus Jahre. Random access schemes for efficient FPGA SpMV acceleration. *Microprocessors and Microsystems*, 47:321–332, 2016.

[197] Javier Vázquez-Castillo, Alejandro Castillo-Atoche, Roberto Carrasco-Alvarez, Omar Longoria-Gandara, and Jaime Ortegón-Aguilar. Fpga-Based Hardware Matrix Inversion Architecture Using Hybrid Piecewise Polynomial Approximation Systolic Cells. *Electronics*, 9(1):182, 2020.

[198] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[199] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 071. IOP Publishing, 2005.

[200] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. Intel math kernel library. In *High-Performance Computing on the Intel® Xeon Phi*, pages 167–188. Springer, 2014.

[201] Erwei Wang, James J Davis, Ruizhe Zhao, Ho-Cheung Ng, Xinyu Niu, Wayne Luk, Peter YK Cheung, and George A Constantinides. Deep Neural Network Approximation for Custom Hardware: Where We've Been, Where We're Going. *ACM Computing Surveys (CSUR)*, 52(2):1–39, 2019.

[202] T. Wang, L. Guo, G. Li, J. Li, R. Wang, M. Ren, and J. He. Implementing the Jacobi Algorithm for Solving Eigenvalues of Symmetric Matrices with CUDA. In *2012 IEEE Seventh International Conference on Networking, Architecture, and Storage*, pages 69–78, 2012.

[203] Yang Wang, Chen Zhang, Zhiqiang Xie, Cong Guo, Yunxin Liu, and Jingwen Leng. Dual-side Sparse Tensor Core. *arXiv preprint arXiv:2105.09564*, 2021.

[204] Yangzihao Wang, Andrew Davidson, Yuechao Pan, Yuduo Wu, Andy Riffel, and John D Owens. Gunrock: A high-performance graph processing library on the GPU. In *ACM SIGPLAN Notices*, volume 51, page 11. ACM, 2016.

[205] Zeke Wang, Hongjing Huang, Jie Zhang, and Gustavo Alonso. Shuhai: Benchmarking High Bandwidth Memory On FPGAS. In *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 111–119. IEEE, 2020.

[206] Haroon Waris, Chenghua Wang, Weiqiang Liu, and Fabrizio Lombardi. Axsa: On the Design of High-Performance and Power-Efficient Approximate Systolic Arrays for Matrix Multiplication. *Journal of Signal Processing Systems*, 8 2020.

[207] Hao Wei, Jeffrey Xu Yu, Can Lu, and Xuemin Lin. Speedup graph processing by graph ordering. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1813–1828, 2016.

[208] Brian Wheatman and Helen Xu. Packed compressed sparse row: A dynamic graph representation. In *2018 IEEE High Performance extreme Computing Conference (HPEC)*, pages 1–7. IEEE, 2018.

[209] Xilinx. Large FPGA Methodology Guide. `www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug872_largefpga.pdf`, 2012.

[210] Xilinx. Ultrascale Architecture Configurable Logic Block. `www.xilinx.com/support/documentation/user_guides/ug574-ultrascale-clb.pdf`, 2017.

[211] Xilinx. *Alveo U280 Data Center Accelerator Card*, 11 2019. Rev. 1.2.1.

[212] Xilinx. *Alveo U280 Data Center Accelerator Cards Data Sheet*, 5 2020. Rev. 1.3.

[213] Xilinx. Axi High Bandwidth Memory Controller v1.0 LogiCORE IP Product Guide. `www.xilinx.com/support/documentation/ip_documentation/hbm/v1_0/pg276-axi-hbm.pdf`, 2020.

[214] Xilinx. *Vivado Design Suite User Guide*, 1 2020. Rev. v2019.2.

[215] Xilinx. *Vitis SPARSE Library*, 6 2021. Retrieved on 2021-08-24.

[216] Ichitaro Yamazaki, Hartwig Anzt, Stanimire Tomov, Mark Hoemmen, and Jack Dongarra. Improving the performance of CA-GMRES on multicores with multiple GPUs. In *IPDPS 2014*, Phoenix, AZ, 05-2014 2014. IEEE.

[217] Da Yan, Wei Wang, and Xiaowen Chu. Demystifying tensor cores to optimize half-precision matrix multiply. In *2020 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 634–643. IEEE, 2020.

[218] Carl Yang, Aydın Buluç, and John D Owens. Design principles for sparse matrix multiplication on the GPU. In *European Conference on Parallel Processing*, pages 672–687. Springer, 2018.

[219] Carl Yang, Aydin Buluc, and John D Owens. Graphblast: A high-performance linear algebra-based graph framework on the GPU. *arXiv preprint arXiv:1908.01407*, 2019.

[220] Dingqing Yang, Amin Ghasemazar, Xiaowei Ren, Maximilian Golub, Guy Lemieux, and Mieszko Lis. Procrustes: a dataflow and accelerator for sparse deep neural network training. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 711–724. IEEE, 2020.

[221] Renchi Yang, Jieming Shi, Xiaokui Xiao, Yin Yang, and Sourav S Bhowmick. Homogeneous network embedding for massive graphs via reweighted personalized PageRank. *arXiv preprint arXiv:1906.06826*, 2019.

[222] Yifan Yang, Joel S Emer, and Daniel Sanchez. Spzip: Architectural Support for Effective Data Compression In Irregular Applications. In *2021 ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pages 1069–1082. IEEE, 2021.

[223] Zi Yin and Yuanyuan Shen. On the dimensionality of word embedding. In *Advances in Neural Information Processing Systems*, pages 887–898, 2018.

[224] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 6848–6856, 2018.

[225] Yunming Zhang, Mengjiao Yang, Riyadh Baghdadi, Shoaib Kamil, Julian Shun, and Saman Amarasinghe. Graphit: A high-performance graph DSL. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):121, 2018.

[226] Zhekai Zhang, Hanrui Wang, Song Han, and William J Dally. Sparch: Efficient architecture for sparse matrix multiplication. In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 261–274. IEEE, 2020.

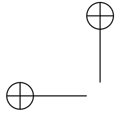
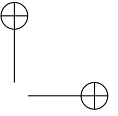
[227] Bolong Zheng, Zhao Xi, Lianggui Weng, Nguyen Quoc Viet Hung, Hang Liu, and Christian S Jensen. Pm-LSH: A fast and accurate LSH framework for high-dimensional approximate NN search. *Proceedings of the VLDB Endowment*, 13(5):643–655, 2020.

[228] Qingmei Zhou, Xin Chen, and Jiuya Zhang. Spectral Clustering-Based Matrix Completion Method for Top-n Recommendation. In *Proceedings of the 2019 5th International Conference on Computing and Data Engineering*, ICCDE' 19, pages 1–6, New York, NY, USA, 2019. Association for Computing Machinery.

[229] Shijie Zhou, Charalampos Chelmis, and Viktor K Prasanna. Optimizing memory performance for FPGA implementation of PageRank. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6. IEEE, 2015.

[230] Shijie Zhou, Kartik Lakhotia, Shreyas G Singapura, Hanqing Zeng, Rajgopal Kannan, Viktor K Prasanna, James Fox, Euna Kim, Oded Green, and David A Bader. Design and implementation of parallel PageRank on multicore platforms. In *2017 IEEE High Performance Extreme Computing Conference (HPEC)*, pages 1–6. IEEE, 2017.

[231] Xiaowei Zhu, Wenguang Chen, Weimin Zheng, and Xiaosong Ma. Gemini: A computation-centric distributed graph processing system. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 301–316, 2016.



[232] Ling Zhuo and Viktor K Prasanna. Sparse matrix-vector multiplication on FPGAs. In *Proceedings of the 2005 ACM/SIGDA 13th international symposium on Field-programmable gate arrays*, pages 63–74, 2005.