DESIGN AND GENERATION OF EFFICIENT HARDWARE ACCELERATORS FOR SPARSE AND DENSE TENSOR COMPUTATIONS

A Dissertation

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Doctor of Philosophy

by

Nitish Kumar Srivastava

May 2020

DESIGN AND GENERATION OF EFFICIENT HARDWARE ACCELERATORS FOR SPARSE AND DENSE TENSOR COMPUTATIONS

Nitish Kumar Srivastava, Ph.D.
Cornell University 2020

Tensor algebra lives at the heart of big data applications. Where classical machine learning techniques such as embedding generation in recommender systems, dimensionality reduction and latent Dirichlet allocation make use of multi-dimensional tensor factorizations, deep learning techniques such as convolutional neural networks, recurrent neural networks and graph learning use tensor computations primarily in the form of matrix-matrix and matrix-vector multiplications. The tensor computations often used in many of these fields operate on sparse data where most of the elements are zeros. Traditionally, tensor computations have been performed on CPUs and GPUs, both of which have low energy-efficiency as they allocate excessive hardware resources to flexibly support various workloads. However, with the end of Moore's law and Dennard scaling, one can no longer expect more and faster transistors for the same dollar and power budget. This has led to an ever-growing need for energy-efficient and high-performance hardware that has resulted in a recent surge of interest in application-specific, domain-specific and behavior-specific accelerators, which sacrifice generality for higher performance and energy efficiency.

In this dissertation, I explore hardware specialization for tensor computations by building programmable accelerators. A central theme in my dissertation is determining common spatial optimizations, computation and memory access patterns, and building efficient storage formats and hardware for tensor computations. First, I present T2S-Tensor, a language and compilation framework for productively generating high-performance systolic arrays for dense tensor computations. Then I present a versatile accelerator, Tensaurus, that can accelerate both dense and mixed sparse-dense tensor computations. Here, I also introduce a new sparse storage format that allows accessing sparse data in a vectorized and streaming fashion and thus achieves high memory bandwidth utilization for sparse tensor kernels. Finally, I present a novel sparse-sparse matrix multiplication accelerator, MatRaptor, designed using a row-wise product approach. I also show how these different hardware specialization techniques outperform CPUs, GPUs and state-of-the-art accelerators in both energy efficiency and performance.

BIOGRAPHICAL SKETCH

Nitish Srivastava was born to Kanchan Srivastava and Kamal Srivastava in a small town Sitapur, Uttar Pradesh, India on December 21st, 1992. Nitish attended Sacred Heart Inter College in Sitapur from 1996–2008. During high school, in addition to his academic studies, Nitish was also interested in sketching and drawing, and has won several district-level drawing competitions with his sister. Nitish went on to pursue mathematics and science in the 11th and 12th grade (junior and senior high school) at Green Field Academy in Hargaon, from 2008–2010.

Determined to obtain an undergraduate degree in Electrical Engineering, Nitish enrolled at the Indian Institute of Technology, Kanpur, India. Nitish took fundamental courses in electrical engineering, physics and mathematics such as micro-electronics, digital VLSI, semiconductor physics, power systems, communication systems, signal processing, classical mechanics, electrodynamics, linear algebra, probability theory, differential equations and multi-variable calculus. Nitish also served as academic mentor, where he helped freshman undergraduates in understanding the key concepts in electrodynamics. Nitish was also secretary for the electronics club at IIT Kanpur, where he helped organize various intra and inter collegiate technical festivals such as Techkriti, Takneek and Electromania. Nitish completed his undergraduate degree in 2014 with the highest GPA in the graduating batch, consisting of all the engineering and science departments at IIT Kanpur, and was awarded with president's gold medal, proficiency medal and Prateek Mishra memorial gold medal.

After undegraduate, Nitish decided to pursue a Ph.D. degree in computer architecture and was offered a graduate fellowship at Cornell University in 2014. At Cornell University, Nitish began his graduate studies under the tutelage of Prof. Rajit Manohar and Prof. David Albonesi. By working with Prof. Rajit's group, he gained invaluable experience in topics such as asynchronous VLSI and circuit design. Soon after Prof. Rajit Manohar moved to Yale University, Nitish continued his Ph.D. at Cornell under the guidance of Prof. Zhiru Zhang and Prof. David Albonesi in topics related to energy-efficient computer architecture, programmability and design of specialized hardware, spatial compilers, and ASIC design.

This document is dedicated to all my teachers, to all my *gurus*.

ACKNOWLEDGEMENTS

This dissertation would not have been possible without the support, encouragement, and advice from many people. First and foremost, I would like to thank my advisors Zhiru Zhang and David Albonesi who have been true mentors and role models throughout my stay at Cornell. Zhiru's passion for research and his ability to grasp things in various different fields whether it be FPGAs, high-level synthesis, machine learning, programming languages, and computer architecture has always amazed and inspired me. Dave's critical thinking, ability to tease out the novel aspects of any research and passion for computer architecture has always inspired me. I would also like to thank Hongbo Rong, my mentor during my internship at Intel, whose skills in programming languages, compilers and spatial architectures helped me a lot in my Ph.D. He has been a great teacher and at the same time a really good friend to me. His constant support and advice has been really fruitful in my research. I would also like to thank the rest of my committee, Christopher Batten and Rajit Manohar, for helping me refine my research with thoughtful feedbacks. I have truly enjoyed working with Chris on HammerBlade, XLOOPS++ and pointer-chase prefetcher projects. Thanks Chris, for teaching me computer architecture and complex ASIC design. Those two courses motivated me to make energy efficient computing as the main focus of my PhD. Thanks Rajit, for teaching me asynchronous VLSI, which allowed me to think hardware design in a completely different way. Thanks to all the members of Zhang Research Group and Albonesi Research Group. Many thanks to Steve Dai who has been a great friend and collaborator. I really enjoyed working with him on the HLS face detection project. I also collaborated with Mathew Watkins, former Ph.D. student in Albonesi group, while working on the CGRA project. His mentorship and advice has been really valuable to me. Matt has been facing some serious health issues for the past one and a half year. I hope for the best for you Matt and hope you will get well soon. I collaborated with Hanchen Jin on the Tensaurus and Hammerblade projects. It was a lot of fun working with him. I also collaborated with Yuwei Hu, Chenhui Deng and Yichi Zhang on the Hammerblade project which was an amazing experience. I appreciate the friendship of Neeraj Kulkarni, who was always there as a really good friend. We did multiple course projects together and it was always fun working and brainstorming ideas with him. I would like to thank Shreesha Srinath and Skand Hurkat for being great mentors and friends during the times when I was struggling to find new research ideas. Despite, I did not collaborate with the rest of the Zhang and Albonesi Research Group, I really appreciate the friendship and support from Gai Liu, Ritchie Zhao, Ecenur Üstün, Abhinandan Majumdar, Yuan Zhou, Yi-Hsiang (Sean) Lai, Zhenghong (John) Jiang, Cunxi Yu and Jorden Dotzel. I would also like to thank all the friends at CSL.

This dissertation was supported in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, under NSF Awards #1453378, #1512937, #1909661, NSF/Intel CAPA Award #1723773, and by AFRL and DARPA under agreement number FA8650-18-2-7863.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

ALS	alternating least square
CGRA	coarse-grained reconfigurable architecture
CISR	compressed interleaved sparse row
CNN	convolution neural network
CPD	cannonical polyadic decomposition
CSC	compressed sparse column
CSR	compressed sparse row
C^{2}SR	cyclic channel sparse row
COO	co-ordinate
DSL	domain-specific language
DSP	digital signal processing
FIFO	first in first out
FMA	fused multiply add
FPGA	field-programmable gate array
FSM	finite state machine
GB/s	giga-bytes per second
GEMM	general matrix-matrix multiplication
GEMV	general matrix-vector multiplication
GFLOPS	giga floating point operations per second
GOP/s	giga operations per second
HBM	high bandwidth memory
HDL	hardware descriptive language
HOOI	higher-order orthogonal iterations
HPC	high-performance computing
IoT	internet of things
IR	intermediate representation
LIC	latency insensitive channel
LOC	lines of code
LUT	look-up table
MAC	multiply-accumulate
ML	machine learning
MTTKRP	matricized tensor times khatri-rao product
PE	processing engine
QoR	quality of result
RAM	random access memory
SCISR	scalable compressed interleaved sparse row
SIMD	single instruction multiple data
SpGEMM	sparse matrix-matrix multiplication
SpMM	sparse matrix dense matrix multiplication
SpMV	sparse matrix dense vector multiplication
T2S	temporal to spatial
TTM	tensor times matrix
TTMc	tensor times matrix chain

CHAPTER 1
INTRODUCTION

Data-intensive computing significantly impacts the world economy, from warehouse computing to personal mobile devices and Internet of Things (IoT). In this era of data explosion, machine learning (ML) techniques such as social networks, recommendation systems, computational advertising, and image recognition have become pervasive tools in emerging large-scale commercial applications. Tensor algebra lives at the heart of many of these big data applications. For example, multi-dimensional tensor factorizations are extensively deployed in classical ML techniques such as embedding generation in recommender systems, dimensionality reduction, and latent dirichlet allocation [SDLF+17, CMDL+15, VT02, PFS12]. More recent deep learning techniques such as convolutional neural networks (CNNs), recurrent neural networks (RNNs), and graph learning also make use of tensor computations primarily in the form of matrix-matrix and matrix-vector multiplications [CWV+14, VAG17, JYP+17]. The tensor computations in these fields often operate on sparse data where most of the elements are zeros. While the underlying computations in all these fields are some basic tensor computations, the sparsity of the tensors can vary significantly among different fields. For example, the graphs in graph learning applications such as Amazon co-purchase network consist of 400K nodes and 3.2M edges forming an adjacency matrix of 400K $\times$ 400K with a density of 0.002%. This is in contrast to sparse deep learning applications such as sparse convolutional neural networks where the weights and activations have a density of 20-40% and the graphs in Internet and social media where the density is in the order of 10^{-6}%. At the same time there are many other ML algorithms that mainly operate on dense tensors.

Traditionally, tensor computations have been performed on CPUs and GPUs [CHW+13,ESA+06, VSM11], both of which have low energy efficiency as they allocate excessive hardware resources to flexibly support various workloads. For many years, general-purpose processor architectures took advantage of Dennard scaling and Moore's Law. As per Moore's Law [M+65], new process technologies reduced feature size of the transistors by a factor of two every 18 months to two years for roughly the same dollar budget. This effectively resulted in exponentially more transistors on a chip for the same cost. Similarly, Dennard Scaling predicted a constant-factor scaling down of feature sizes, voltages, and gate capacitance every 18 months to two years, which led to faster transistors under roughly the same power budget [DGR+74]. Because of Moore's Law and Dennard Scaling, applications were transparently becoming faster and more energy efficient without any hardware/software specialization. However today, because of the end of Dennard scaling and an apparent slowing down of Moore's Law, one cannot get more and better transistors. This has caused a recent surge of interest in application- and domain-specific accelerators, which trade generality for higher performance and energy efficiency.

1.1 The Kernel-Sparsity Spectrum

Many real-world tensors are sparse and there are different types of tensor kernels. Fig. 1.1 shows where different application domains lie in the kernel-sparsity space. The x-axis of this spectrum represents density of the tensors in the log domain; The y-axis consists of three representative tensor kernels: matricized tensor times Khatri-Rao product (MTTKRP), matrix-matrix multiplication (MM) and matrix-vector multiplication (MV). As the figure shows, for the same tensor kernel the density of the tensor can vary significantly among the application domains. The existing works on tensor hardware either focus on a single kernel or a single application domain while making assumptions about the sparsity of the tensors [FOS+14,HLM+16,HKM+17,CES16, ZDZ+16,JYP+17]. The primary reason behind this is that different application domains have different requirements. The applications that operate on very sparse tensors require hardware that is optimized for off-chip and irregular memory accesses, while the applications consisting of dense tensor computations require high on-chip compute resources. I tackle the first challenge by designing sparse tensor formats, which enable efficient off-chip memory accesses for tensors with varying range of sparsity; and the second challenge by designing designing regular systolic arrays with SIMD vectorization to efficiently utilize the on-chip compute resources.

1.1.1 Challenges in Dense Tensor Acceleration

There has been a rich body of research on accelerating dense tensor computations. H.T. Kung and Charles E. Leiserson in their classic 1979 book on Systolic array for VLSI described how most of dense computations lie under the class of systolic algorithms and gave a mathematical foundation of designing systolic arrays for such algorithms [KL79]. Even after four decades, systolic arrays are still considered one of the most efficient ways of performing dense tensor computations. Many state-of-the-art dense tensor accelerators such as Google TPU [JYP+17] and NVIDIA Tensor Core [MDCL+18] are systolic arrays. The main challenge in the domain of dense tensor

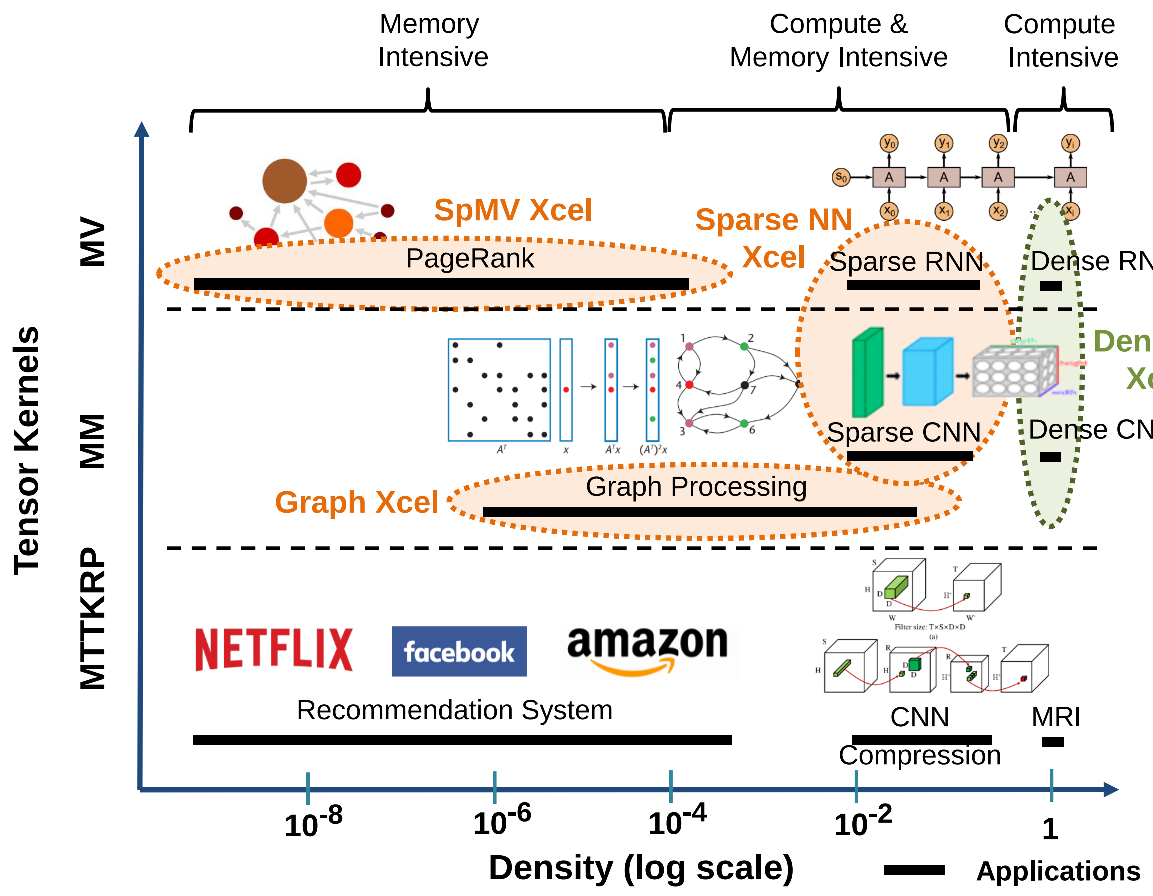


Figure 1.1: Kernel-Sparsity Spectrum of Few Tensor Kernels – The solid and black horizontal lines represent the application domains; the orange and green circles represent different application domains for which hardware accelerators have been built.

acceleration, however, is productivity. While the theories behind systolic algorithms and architectures are well understood, building high-performance systolic arrays requires significant efforts in coding and performance tuning. A recent work by Intel shows that it took around 18 months for industry experts to design a high-performance dense matrix-matrix multiplication accelerator using high-level synthesis (HLS) on an FPGA [Ron17a]. The reason behind this is there are numerous hardware optimizations such as tiling, data buffering, and I/O interfacing that need to be handled, apart from expressing the algorithm in a systolic manner, in order to get a high-performance design. These optimizations depend significantly on the amount of available on-chip compute and memory resources such arithmetic and logic units, registers and random-access memories (RAMs), and on the off-chip memory bandwidth. Thus, designing a highly-efficient dense tensor accelerator in a short amount of time still remains a challenge.

1.1.2 Challenges in Sparse Tensor Acceleration

The challenges faced in sparse tensor acceleration are primarily flexibility and efficiency. Unlike dense tensor acceleration, where the datasets are always dense and the hardware for a single tensor kernel such as dense matrix-matrix multiplication can be reused across multiple application domains (e.g., TPU [JYP$^+$17]), the same sparse tensor kernel requires different hardware accelerators for different application domains (e.g., SpMV accelerator by Fowers *et al.* [FOS$^+$14] for graphs and ESE by Han *et al.* [HKM$^+$17] for RNNs, both of which accelerate sparse-dense matrix-vector multiplication). This is because the density of the tensors involved in different sparse-tensor application can vary significantly and thus requires different density-dependent hardware and software optimizations such as sparse storage formats, tiling strategies and memory read/write vectorization. This makes the existing sparse tensor accelerators less flexible as they focus on a single tensor kernel or a single application domain while making assumptions about the sparsity of tensors involved in the computation. Thus, designing a hardware accelerator that can flexibly support different sparse tensor kernels with varying ranges of sparsity while being efficient in terms of both energy and performance is a major challenge in sparse tensor acceleration.

1.2 The Present and Future of Tensor Computations

The Moore's law and Dennard scaling have slowed down while at the same time data analytics and machine learning are continuously evolving. Below are some observations and predictions on how this might change the future of tensor computations.

Tensor computations will continue to gain importance. Today, tensor algebra lives at the heart of data analytics and machine learning. The primary reason behind this is that most of the real-world data whether that be the camera images, video recordings, temperature sensor data or digital circuits come in the form of tensors. Tensors are also a natural way of representing multi-dimensional vector spaces and thus form the basis of many scientific simulations. Both tensor algebra and tensor calculus have been thoroughly studied in the literature and thus tensor based computations are preferred methods in multiple domains. The forms of data that do not have obvious correspondence with tensors such as text documents, speech data, and graphs are often represented as tensors using some embedding techniques [MSC$^+$13, MCCD15, GL14]. Today, with social media and entertainment industry, humans are significantly relying on machines to provide recommendations for movies, clothes, food, and other consumer products. In the near future, machines may be intelligent enough to make decisions for humans, a glimpse of which can be seen with Amazon Echo Dot, OK Google and Siri.

Software for tensor computations will make significant advancements. The first dense matrix algebra library, basic linear algebra subroutines (BLAS) [bla], was developed in 1979, 25 years after the invention of formula translation language (FORTRAN) and 34 years after John von Neumann described modern programmable computers. Over the years, several other dense matrix algebra libraries have been created such as AMD Core Math Library (ACML) [acm], Automatically Tuned Linear Algebra Software (ATLAS) [atl], Intel Math Kernel Library (MKL) [mkl], and OpenBLAS [ope]. In the 1980s, sparse matrix algebra started garnering interest and the first sparse linear algebra library, Sparse BLAS [DHP02], was designed. Later Eigen [GJ+10], CSparse [Dav14], PETSc [BAA+19], OSKI [VDY05] and other sparse matrix algebra libraries were created. Around the same time researchers realized the power of multi-dimensional tensor algebra and multiple tensor libraries and toolbox such as TeLa [tel] and Cadabra [Pee07] were created. Recently, the growth in deep learning, social media, and human machine interactions resulted in a renewed interest in tensor algebra and many tensor computation libraries and languages such as Tensor Toolbox, GigaTensor [KPHF12], SPLATT [SRSK15], Halide [RKBA+13], TensorFlow [ABC+16], and TACO [KKC+17] were developed. Many of the recent work on tensor languages and libraries such as TensorFlow also not only support CPUs and GPUs but also specialized tensor hardware. As this is just the beginning of the era of machine learning and data analytics, more languages and libraries will be developed to meet the ever-growing needs of tensor computations. Further, with the advancements in tensor hardware these languages and libraries will have specialized backends to target different high-performance tensor hardware.

Tensor hardware will gain increasing importance. With the advent of the modern area of deep learning in the last decade, significant advancements have been made both in industry and academia to design energy-efficient and high-performance accelerators for deep learning. Google Tensor Processing Unit (TPU) [JYP+17], Microsoft Brainwave [CFO+18], and NVIDIA Tensor Core [MDCL+18] are a few examples of industry efforts while Eyeriss [CES16], ESE [HKM+17], EIE [HLM+16], SCNN [PRM+17], Cambricon-X [ZDZ+16] and DianNao [CCX+16] are some of the efforts made in academia. As deep learning is still in its nascent stage and significant advancements have to be made, tensor hardware will keep on gaining interest. Similar to deep learning, tensor factorizations primarily in the form of recommendation systems have also attracted a great deal of interest in the past few years. This emerging class of applications will require efficient processing and will create a need for more specialized tensor hardware.

1.3 Dissertation Overview

This dissertation presents hardware accelerators and storage formats for accelerating tensor computations. It has two thrusts that tackle two major problems in accelerator design for tensor computations: (1) creating a language and compilation framework for productively designing high-performance accelerators for dense tensor computations, and (2) building versatile accelerators that can accelerate dense and mixed sparse tensor computations. The first thrust provides abstractions, in the form of language primitives, for the computation patterns and spatial optimizations found in dense tensor computations and generating high-performance designs for reconfigurable spatial hardware. This allows programmers to design high-performance hardware accelerators for dense tensor computations in a productive manner. The second thrust extends the domain knowledge gained from (1) to design compute patterns, algorithms, spatial optimizations, and sparse storage formats for sparse tensor computations in order to design flexible and efficient hardware for sparse tensor computations.

Since tensors and tensor computations are the main focus of this dissertation, Chapter 2 presents an extensive background on tensors, tensor factorizations, applications of tensor factorizations, matrix kernels and their applications. The rest of the dissertation is organized as follows. Chapter 3 presents a language and compilation framework for productively generating high-performance systolic arrays for dense tensor computations on spatial architectures. Chapter 4 presents a versatile accelerator that can accelerate both dense and mixed sparse-dense tensor computations. It also proposes a new sparse storage format that allows accessing sparse data in a vectorized and streaming fashion and thus achieves high memory bandwidth utilization for sparse tensor kernels. Chapter 5 presents a novel sparse-sparse matrix multiplication accelerator designed using a row-wise product approach. This chapter also proposes a new sparse storage format that has the same benefits of the sparse storage format in Chapter 4 but also allows using the same format for the output matrix.

The key contributions of this dissertation are:

- Proposing a concise yet expressive programming abstraction that decouples spatial mapping from the functional specification of a tensor computation.
- Identifying a set of key spatial optimizations that are essential for creating high-performance spatial hardware for dense tensor computations.
- Proposing a language and compilation framework to productively generate high-performance systolic arrays for dense tensor computations.
- Proposing the first hardware accelerator designed for mixed sparse-dense tensor factorizations. This accelerator is both versatile and adaptable and supports several mixed sparse-dense matrix operations for a wide range of sparsity.
- Proposing a new sparse storage format, compressed interleaved sparse slice (CISS), which allows accessing sparse data in vectorized and streaming fashion and thus helps sparse accelerators achieve high memory bandwidth utilization and performance for sparse tensor kernels.
- Systematically analyzing different dataflows for sparse-sparse matrix multiplication by comparing and contrasting them against data reuse and on-chip memory requirements. It also shows that a row-wise product approach, which has not been explored in the design of sparse-sparse matrix multiplication accelerators, has the potential to outperform the existing approaches.
- Proposing C^{2}SR, a new hardware-friendly sparse storage format that allows different parallel processing engines to access the data in a vectorized and streaming manner leading to high utilization of the available memory bandwidth. This format solves the same challenges as in CISS format; however, unlike CISS that can be used only for sparse input matrices but not sparse output matrices, C^{2}SR can be used for both.

1.4 Collaboration, Previous Publications, and Funding

This dissertation has been made possible by the contributions of many people at Cornell University and Intel Parallel Computing Labs. My Ph.D. advisors, Profs. Zhiru Zhang and David

Albonesi, have been extremely helpful in providing insightful advice, feedback on research ideas, technical writing and presentations. Furthermore, Dr. Hongbo Rong of Intel has been a very valuable mentor and a collaborator for most of the works presented in this dissertation.

The T2S-Tensor project was done as a part of my internship at Intel Parallel Computing Labs under the mentorship of Dr. Rong. This work was later published in International Symposium on Field-Programmable Custom Computing Machines (FCCM) [SRB+19b]. Prithayan Barua, Guanyu Feng and Huanqi Cao had been great collaborators for this project. Additionally, Christohpher Hughes, Paul Petersen, and Pradeep Dubey had been really helpful in providing ideas and feedback for this work. Geoff Lowney, Tim Mattson, Vivek Sarkar and Wenguang Chen gave useful feedback for the paper. Yi-Hsiang Lai, Jie Wang, Weihao Zhao and Size Zheng were the first users of T2S-Tensor who extended this work for general systolic arrays in their work on T2S-Systolic. The Tensaurus project was done a Cornell University and was published in International Symposium on High-Performance Computer Architecture (HPCA) [SJS+20]. Hanchen Jin and Shaden Smith were great collaborators for this project, where Hanchen helped in designing the RTL designs for the accelerator and Shaden helped in providing key insights and sparse tensor datasets used for this work. Hongbo Rong had been really helpful in providing ideas and feedback for this work. Alex Coy, Parker Miller, Ayoub Benkhoris and Congyang Li helped in designing high-level synthesis and verilog models for this project. The MatRaptor project was done with the collaboration of Hanchen Jin and Jie Liu and is under submission.

Apart from the three major projects presented in this dissertation, during my Ph.D. I also designed real-time face-detection accelerator [SDMZ17] using high-level synthesis and provided a comprehensive case study to explore the flow from a pure software based implementation to an optimized C++ design suitable for HLS design flow. This accelerator design was later made a part of the Rosetta [ZGD+18], which is a realistic benchmark suite for software programmable FPGAs. During my first few years of Ph.D., I also worked on asynchronous circuit design and a proposed novel operation-dependent desynchronization technique [SM18], which desynchronizes the circuit and improves performance beyond the limits of synchronous design.

This research was funded in part by CRISP, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, under NSF Awards #1453378, #1512937, #1909661, NSF/Intel CAPA Award #1723773, and by AFRL and DARPA under agreement number FA8650-18-2-7863. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of AFRL and DARPA or the U.S. Government.

CHAPTER 2
BACKGROUND ON TENSOR COMPUTATIONS

This chapter provides a background on tensor computations. It first introduces tensor notations and then describes the most commonly used tensor computations such as tensor factorizations, matrix-matrix and matrix-vector multiplications.

2.1 Tensor Notations

A tensor is a generalization of a matrix to multiple dimensions. A scalar is a tensor of dimension zero, a vector is a tensor of dimension one and a matrix is a tensor of dimension two. We denote tensors with three or more dimensions using capital calligraphic letters (e.g., $\mathcal{A}$), matrices using boldface capital letters (e.g., $\mathbf{A}$), vectors using boldface letters (e.g., $\mathbf{a}$), and scalars using Greek letters (e.g. α).

The dimensions of a tensor are also called its modes and colon(:) is used to indicate all the elements of a mode. Thus a 3-dimensional (3-d) tensor is a tensor with 3 modes. Fibers are building blocks of tensors. A fiber is the result of holding all but one index constant. For a 3-d tensor $\mathcal{A}$, its fibers are $\mathcal{A}(:,j,k)$, $\mathcal{A}(i,:,k)$, and $\mathcal{A}(i,j,:)$. Similarly, for a matrix $\mathbf{A}$ its rows $\mathbf{A}(i,:)$ and columns $\mathbf{A}(:,j)$ are its fibers. A slice of a tensor is the resultant matrix by holding all but two indices constant. Slices of a 3-d tensor $\mathcal{A}$ would be $\mathcal{A}(i,:,:)$, $\mathcal{A}(:,j,:)$ and $\mathcal{A}(:,:,k)$.

2.2 Tensor Factorizations

Tensor factorization has two important applications in machine learning: *dimensionality reduction as a learning task*, and *compression of network layers* (e.g., convolutional layers) in deep neural networks.

2.2.1 Recommendation Systems

Matrix and tensor factorization have traditionally been used in recommender systems [SZL+05, KB06] to produce factor matrices that represent an *embedding* (e.g., of users and items in user-item pairs) into the reduced latent space. These factor matrices are then used to predict the relevance "score" of the interaction (e.g., between a user and an item) by simply taking the inner product of their vectors in the latent feature space. With the emergence of larger and more complex data sets, tensor decomposition offers several advantages over matrix- and neural network-based methods. While deep neural networks have produced accurate results in recommendation systems [ZZS+18, WFFW17], computer vision [S+15, HZRS16], and machine translation [VSP+17], they are also *expensive to train*, often requiring large quantities of labeled data, and have *limited interpretability*. In these respects, tensor decomposition can provide a *faster, yet competitive*, method for producing embedding for recommendation systems. Tensor decomposition also allows incorporating new data via coupled [BTK+14, DK17] and streaming methods [SHSK18, STF06], without the expensive re-factorization. It also has the advantage that the decomposed factors are more interpretable, as they often map easily to existing concepts [HGS14].

2.2.2 Model Compression

A recent study by Facebook [PNB+18] indicates that many important applications in ML: such as recommendations, computer vision, and language models: will soon see increased model complexity, and that a significant fraction of future demand is expected to come from workload corresponding to deep learning *inference*. Therefore, reducing their storage and computation cost will become critical, particularly since many inference workloads will require real-time response, or be computed on the *edge*. In convolutional neural networks in particular, convolution contributes to the bulk of all computation, and compressing this layer (i.e., a 4-D tensor) via tensor decomposition has already been shown to reduce the required data movement and computation, and improve the overall efficiency [CWZZ18, DZB+14, LGR+14, TXWW15, JVZ14].

The majority of prior studies have focused on matrix factorization. However, real-world data is becoming increasingly higher-order, electronic health records have dimensions that are as high as 85 [LCP+17], and keeping data in tensor form retains the multi-way relationship that may otherwise be lost when formulated as a matrix [KB06], and consequently, produces more accurate results [FO17].

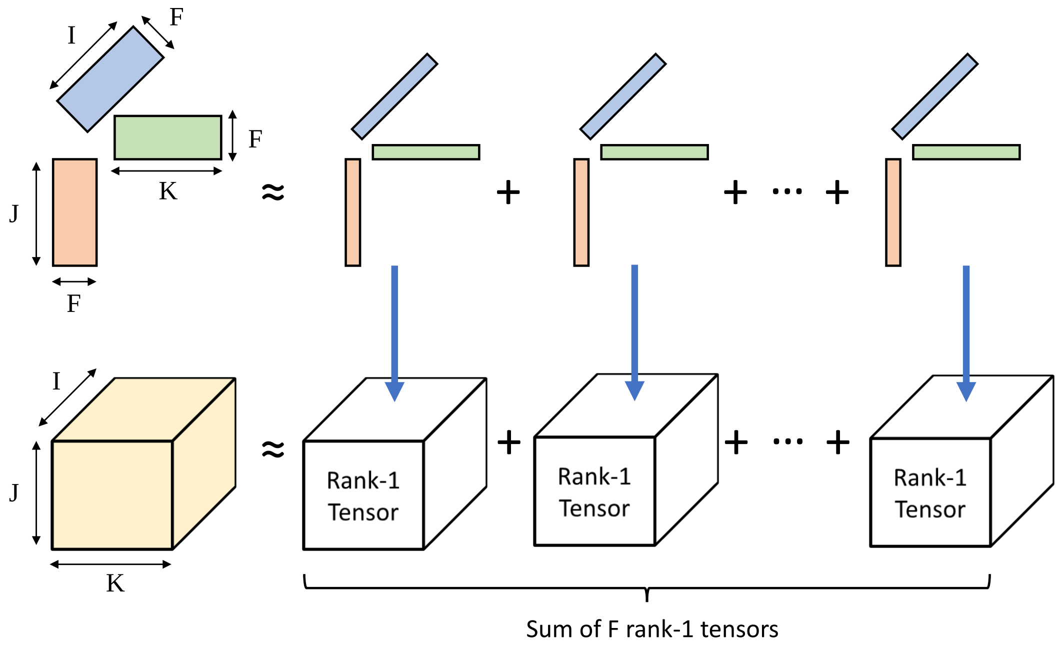


Figure 2.1: CP Decomposition of 3-D Tensor

2.3 Canonical Polyadic Decomposition

Canonical Polyadic Decomposition (CPD) is one of the most commonly used tensor factorization techniques and is used in fields such as recommender systems [SKB+12], signal processing [NS10] and psychometrics [CC70]. CPD is an extension of the Singular Value Decomposition (SVD) to tensors and decomposes a tensor into a summation of F rank-one tensors, where F can either be the rank of the tensor or some smaller integer if a low-rank approximation is desired. A rank-one tensor of order n is the outer product of n vectors. Eq. (2.1) shows how a 3-d tensor is approximated using the outer product of F rank-one tensors.

$$\boldsymbol{\mathcal{X}}(i,j,k) \approx \sum_{f=0}^{F-1} \mathbf{A}(i,f)\mathbf{B}(j,f)\mathbf{C}(k,f) \tag{2.1}$$

Determining the exact rank of a tensor is NP-hard [Hås90] and we are almost always interested in $F << max(I,J,K)$ for sparse tensors. When computing the rank-F CPD of a 3-d tensor, we wish to find factor matrices $\mathbf{A} \in \mathbb{R}^{I\times F}$, $\mathbf{B} \in \mathbb{R}^{J\times F}$, and $\mathbf{C} \in \mathbb{R}^{K\times F}$ where **A**, **B**, and **C** are typically dense regardless of the sparsity of $\boldsymbol{\mathcal{X}}$.

The method of Alternating Least Squares (ALS) as shown in Algorithm 1 is the most commonly used algorithm for computing the CPD. The computations in Algorithm 1 at line numbers 2, 5 and

Algorithm 1 CP-ALS for a mode-3 tensor

```
    Input: $\mathcal{X}$ : A 3rd order tensor F: The rank of approximation
    Output: CP decomposition A, B, C
 1: repeat
 2:     $\mathbf{A}(i,f) \leftarrow \sum_{j=0}^{J-1}\sum_{k=0}^{K-1} \mathcal{X}(i,j,k)\mathbf{B}(j,f)\mathbf{C}(k,f)$
 3:     $\mathbf{A} \leftarrow \mathbf{A}(\mathbf{B}^T\mathbf{B} - \mathbf{C}^T\mathbf{C})^{-1}$
 4:     Normalize columns of A
 5:     $\mathbf{B}(j,f) \leftarrow \sum_{i=0}^{I-1}\sum_{k=0}^{K-1} \mathcal{X}(i,j,k)\mathbf{A}(i,f)\mathbf{C}(k,f)$
 6:     $\mathbf{B} \leftarrow \mathbf{B}(\mathbf{A}^T\mathbf{A} - \mathbf{C}^T\mathbf{C})^{-1}$
 7:     Normalize columns of B
 8:     $\mathbf{C}(k,f) \leftarrow \sum_{i=0}^{I-1}\sum_{j=0}^{J-1} \mathcal{X}(i,j,k)\mathbf{A}(i,f)\mathbf{B}(j,f)$
 9:     $\mathbf{C} \leftarrow \mathbf{C}(\mathbf{A}^T\mathbf{A} - \mathbf{B}^T\mathbf{B})^{-1}$
10:     Normalize columns of C
11: until no improvement or maximum iterations reached
```

8 are matricized tensor times Khatri-Rao products (MTTKRP). MTTKRP is thus executed once per mode per iteration of ALS and is often the computational bottleneck of CPD.

2.4 Tucker Decomposition

The objective of Tucker decomposition is to model a tensor $\mathcal{X}$ with a set of orthonormal matrices and a core tensor. The orthonormal matrices are referred to as factor matrices. Eq. (2.2) shows how a 3-d tensor can be approximated using a core tensor $\mathcal{G}$ and the factor matrices **A**, **B** and **C**.

$$\mathcal{X}(i,j,k) \approx \sum_{f_k=0}^{F_k-1}\sum_{f_j=0}^{F_j-1}\sum_{f_i=0}^{F_i-1} \mathcal{G}(f_i,f_j,f_k)\mathbf{A}(i,f_i)\mathbf{B}(j,f_j)\mathbf{C}(k,f_k) \tag{2.2}$$

Several optimization algorithms have been developed to compute Tucker decomposition [DLDMV00a], [DLDMV00b], however, higher-order orthogonal iterations (HOOI) [DLDMV00b] is the most popular algorithm. HOOI is an iterative algorithm that cyclically updates each factor matrix and the core tensor until convergence. Most applications involving tensors perform a low-rank factorization, i.e., $max(F_i,F_j,F_k) << max(I,J,K)$ for mode-3 tensors.

The computations in Algorithm 2 at line numbers 2, 4 and 6 are tensor times matrix chain (TTMc). TTMc is thus executed once per mode per iteration of HOOI and is often the computational bottleneck of Tucker decomposition.

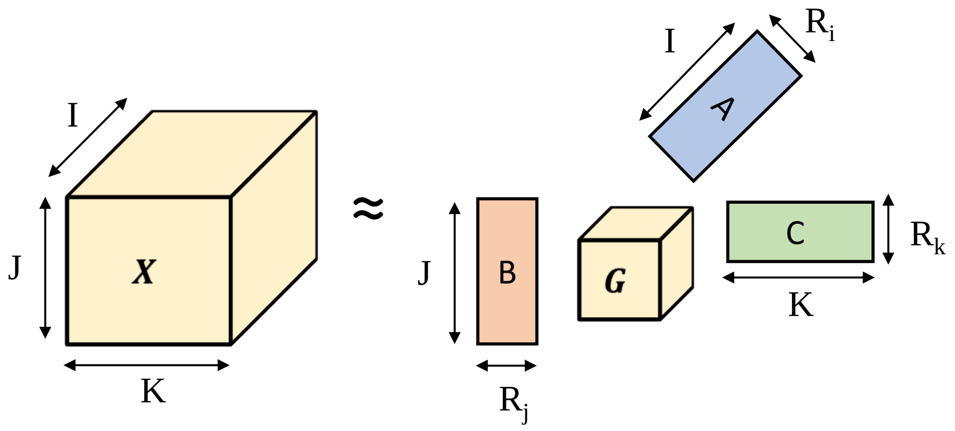


Figure 2.2: Tucker Decomposition of 3-d Tensor

Algorithm 2 HOOI for a mode-3 tensor

```
    Input: $\mathcal{X}$ : A 3rd order tensor $F_I$, $F_J$, $F_K$: The ranks of approximation
    Output: Tucker decomposition $\mathcal{G}$, $\mathbf{A}$, $\mathbf{B}$, $\mathbf{C}$
1: repeat
2:     $\mathcal{Y}_0(i, f_j, f_k) \leftarrow \sum_{j=0}^{J-1}\sum_{k=0}^{K-1} \mathcal{X}(i,j,k)\mathbf{B}(j,f_j)\mathbf{C}(k,f_k)$
3:     $\mathbf{A} \leftarrow F_I$ leading left singular vectors in mode-0 unfolding of $\mathcal{Y}_0$
4:     $\mathcal{Y}_1(f_i, j, f_k) \leftarrow \sum_{i=0}^{I-1}\sum_{k=0}^{K-1} \mathcal{X}(i,j,k)\mathbf{A}(i,f_i)\mathbf{C}(k,f_k)$
5:     $\mathbf{B} \leftarrow F_J$ leading left singular vectors in mode-1 unfolding of $\mathcal{Y}_1$
6:     $\mathcal{Y}_2(f_i, f_j, k) \leftarrow \sum_{i=0}^{I-1}\sum_{j=0}^{J-1} \mathcal{X}(i,j,k)\mathbf{A}(i,f_i)\mathbf{B}(j,f_j)$
7:     $\mathbf{C} \leftarrow F_K$ leading left singular vectors in mode-0 unfolding of $\mathcal{Y}_2$
8:     $\mathcal{G}(f_i, f_j, f_k) \leftarrow \sum_{k=0}^{K-1} \mathcal{Y}_2(f_i, f_j, k)\mathbf{C}(k,f_k)$
9: until no improvement or maximum iterations reached
```

2.5 Matricized Tensor Times Khatri-Rao Product (MTTKRP)

MTTKRP is the key computation kernel in the alternating least square (ALS) method, which is the most popular method for finding the factor matrices in CPD [SDLF+17, KB09]. The computation for MTTKRP consists of multiplication of a tensor with $N-1$ factor matrices, where N is the mode of the tensor, to produce an output matrix. Eq. (2.3) shows the MTTKRP kernel for a 3-d tensor along mode 0 (i), where $\cdot$ denotes multiplication. Since MTTKRP is used for both sparse and dense tensor factorizations [BHR18], I refer to MTTKRP on sparse tensors as SpMTTKRP and on dense tensors as DMTTKRP. The input matrices and the output matrix in both SpMTTKRP and DMTTKRP are dense. Even with very efficient data structures [SRSK15, LSV18], the arithmetic intensity of SpMTTKRP remains low, making this kernel memory bound [CLSS18].

The Hadamard product, denoted by $\circ$, is the element-wise multiplication of two matrices with the same dimensions. It is distributive and can be used to factor out the operand matrices in

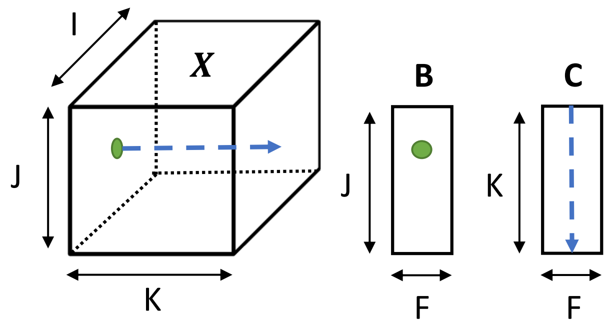


Figure 2.3: MTTKRP

MTTKRP [SRSK15] as shown in Eq. (2.4). Here the Hadamard product operates on two vectors instead of two matrices. Such factorization reduces the number of multiplications in DMTTKRP from $2I \cdot J \cdot K \cdot F$ to $I \cdot J \cdot F \cdot (K+1)$. Here I, J and K are the sizes of the three dimensions of the tensor and F is the desired rank for tensor factorization (normally in the order of 10s to 100s). Eq. (2.4) can be easily generalized to MTTKRP on tensors with more than three dimensions as shown in Eq. (2.5).

$$\mathbf{Y}(i,f) = \sum_{j=0}^{J-1} \sum_{k=0}^{K-1} \boldsymbol{\mathcal{A}}(i,j,k) * \mathbf{B}(j,f) * \mathbf{C}(k,f) \tag{2.3}$$

$$\mathbf{Y}(i,:) = \sum_{j=0}^{J-1} \mathbf{B}(j,:) \circ \left(\sum_{k=0}^{K-1} \boldsymbol{\mathcal{A}}(i,j,k) * \mathbf{C}(k,:) \right) \tag{2.4}$$

$$\mathbf{Y}(i_1,:) = \sum_{i_2} \mathbf{M}_2(i_2,:) \circ \ldots \circ \sum_{i_n} \boldsymbol{\mathcal{A}}(i_1,\ldots,i_n) \cdot \mathbf{M}_n(i_n,:) \tag{2.5}$$

Similarly, MTTKRP on sparse tensors with and without operand factoring can be represented as Eq. (2.6) and (2.7). The reductions in number of operations in the case of SpMTTKRP with operand factoring are similar to its dense counterpart [SRSK15].

$$\mathbf{Y}(i,f) = \sum_{\substack{j \in \{j \mid \exists k\, st. \\ \boldsymbol{\mathcal{A}}(i,j,k) \neq 0\}}} \sum_{\substack{k \in \{k \mid \\ \boldsymbol{\mathcal{A}}(i,j,k) \neq 0\}}} \boldsymbol{\mathcal{A}}(i,j,k) * \mathbf{B}(j,f) * \mathbf{C}(k,f) \tag{2.6}$$

$$\mathbf{Y}(i,:) = \sum_{\substack{j \in \{j \mid \exists k\, st. \\ \boldsymbol{\mathcal{A}}(i,j,k) \neq 0\}}} \mathbf{B}(j,:) \circ \left(\sum_{\substack{k \in \{k \mid \\ \boldsymbol{\mathcal{A}}(i,j,k) \neq 0\}}} \boldsymbol{\mathcal{A}}(i,j,k) * \mathbf{C}(k,:) \right) \tag{2.7}$$

2.6 Tensor Times Matrix Chain (TTMc)

TTMc involves a sequence of tensor times matrix operations along each mode, which compresses the tensor. The output of TTMc is another tensor compressed for all but one mode. Eq. (2.8) shows the TTMc kernel for a 3-d tensor along mode 0 (i). Similar to MTTKRP, TTMc is used for both dense and sparse tensors [CCJ$^+$17, BKK19, CLC18]. Accordingly, I refer to TTMc on sparse tensors as SpTTMc and on dense tensors as DTTMc. For both SpTTMc and DTTMc, the operand matrices and output tensor are dense.

The Kronecker product, denoted by $\otimes$, is the generalization of the vector outer product to matrices and tensors. It is also distributive and can be used to factor out the operand matrices in TTMc as shown in Eq. (2.9). Such factorization reduces the number of multiplications in DTTMc from $2 \cdot I \cdot J \cdot K \cdot F_1 \cdot F_2$ to $I \cdot J \cdot (K \cdot F_2 + F_1 \cdot F_2)$. Here F_1 and F_2 are in the order of 10s to 100s. Eq. (2.9) can also be easily generalized to tensors with more than three dimensions as shown in Eq. (2.10).

$$\boldsymbol{\mathcal{Y}}(i, f_1, f_2) = \sum_{j=0}^{J-1} \sum_{k=0}^{K-1} \boldsymbol{\mathcal{A}}(i,j,k) * \mathbf{B}(j, f_1) * \mathbf{C}(k, f_2) \tag{2.8}$$

$$\boldsymbol{\mathcal{Y}}(i,:,:) = \sum_{j=0}^{J-1} \mathbf{B}(j,:) \otimes \left(\sum_{k=0}^{K-1} \boldsymbol{\mathcal{A}}(i,j,k) * \mathbf{C}(k,:) \right) \tag{2.9}$$

$$\boldsymbol{\mathcal{Y}}(i_1,:,...,:) = \sum_{i_2} \mathbf{M}_2(i_2,:) \otimes ... \otimes \sum_{i_n} \boldsymbol{\mathcal{A}}(i_1,...,i_n) \cdot \mathbf{M}_n(i_n,:) \tag{2.10}$$

Similarly, TTMc on sparse tensor with and without operand factoring can be represented as Eqs. (2.11) and (2.12) and the reductions in number of operations performed for SpTTMc in the case of operand factoring are similar to the ones for DTTMc [SK17].

$$\boldsymbol{\mathcal{Y}}(i, f_1, f_2) = \sum_{\substack{j \,\in\, \{j \,|\, \exists\, k\; st. \\ \boldsymbol{\mathcal{A}}(i,j,k) \neq 0\}}} \; \sum_{\substack{k \,\in\, \{k\,| \\ \boldsymbol{\mathcal{A}}(i,j,k) \neq 0\}}} \boldsymbol{\mathcal{A}}(i,j,k) * \mathbf{B}(j, f_1) * \mathbf{C}(k, f_2) \tag{2.11}$$

$$\boldsymbol{\mathcal{Y}}(i,:,:) = \sum_{\substack{j \,\in\, \{j \,|\, \exists\, k\; st. \\ \boldsymbol{\mathcal{A}}(i,j,k) \neq 0\}}} \mathbf{B}(j,:) \otimes \left(\sum_{\substack{k \,\in\, \{k\,| \\ \boldsymbol{\mathcal{A}}(i,j,k) \neq 0\}}} \boldsymbol{\mathcal{A}}(i,j,k) * \mathbf{C}(k,:) \right) \tag{2.12}$$

2.7 Matrix-Matrix Multiplication

Matrix-matrix multiplication multiplies two matrices **A** and **B** to produce an output matrix **C**. Matrix-matrix multiplication where both **A** and **B** are dense is called general matrix-matrix multiplication (GEMM); the case where **A** is sparse and **B** is dense is called sparse-dense matrix-matrix multiplication (SpMM); and the case where both **A** and **B** are sparse is called generalized sparse matrix-matrix multiplication or sparse-sparse matrix-matrix multiplication (SpGEMM). GEMM and SpMM are building blocks of many algorithms such as graph learning [HYL17, KW16] and CNNs [CWV⁺14], and SpGEMM is a key kernel in algorithms such as graph contraction [GRS08]. In the literature, researchers have used the term SpMM for both sparse-dense and sparse-sparse matrix multiply; however, for clarity, I will use the term SpMM for sparse-dense, and SpGEMM for sparse-sparse matrix-matrix multiply. Eqs. (2.13), (2.14) and (2.15) show the computations for GEMM, SpMM and SpGEMM.

$$\mathbf{Y}(i,f) = \sum_{j=0}^{J-1} \mathbf{A}(i,j) * \mathbf{B}(j,f) \tag{2.13}$$

$$\mathbf{Y}(i,f) = \sum_{\substack{j \in \{j \mid \\ \mathbf{A}(i,j) \neq 0\}}} \mathbf{A}(i,j) * \mathbf{B}(j,f) \tag{2.14}$$

$$\mathbf{Y}(i,f) = \sum_{\substack{j \in \{j \mid \\ \mathbf{A}(i,j) \neq 0\} \\ \mathbf{B}(j,f) \neq 0\}}} \mathbf{A}(i,j) * \mathbf{B}(j,f) \tag{2.15}$$

2.7.1 Convolutional Layers as Matrix Multiplication

A CNN is a directed acyclic graph of many computation layers. Each layer of a CNN takes input data, also known as the input feature map, performs some computations, and outputs an output feature map. The computational bottleneck of a CNN are the convolutional (CONV) layers, which perform high dimensional convolution. Eq. (2.16) shows the computation within a single CONV layer.

$$\mathcal{O}[n][r_o][c_o] = \sum_{m,k_y,k_x} \mathcal{I}[m][r_o+k_y][c_o+k_x] * \mathcal{W}[n][m][k_y][k_x] \tag{2.16}$$

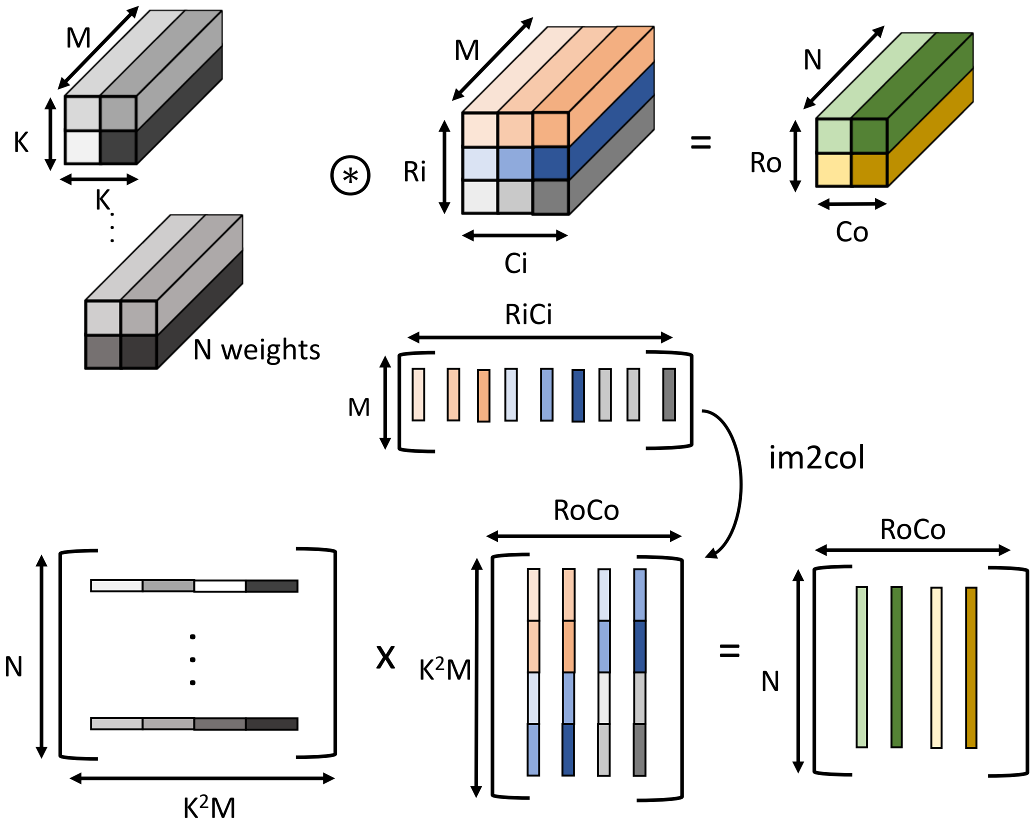


Figure 2.4: Convolution Layer as Matrix-Matrix Multiplication

Here $\mathcal{O}$ and $\mathcal{I}$ are 3-dimensional tensors for the output feature and input feature maps, respectively, and $\mathcal{W}$ is a 4-dimensional tensor for the weights. The computation in CONV layers is easily mapped to matrix-matrix multiplication as shown in Fig. 2.4 by organizing the weights and input feature maps as matrices [CPS06].

2.8 Matrix-Vector Multiplication

Matrix-vector multiplication involves multiplication of a matrix with a vector to produce an output vector. Matrix-vector multiplication involving a dense matrix and a dense vector is known as general matrix-vector multiplication (GEMV), and a sparse matrix and a dense vector is known as sparse-dense matrix-vector multiplication (SpMV). GEMV and SpMV are used in applications such as PageRank [BP98], RNNs, minimal spanning tree, single-source shortest path and ML algorithms such as support vector machine [NMM15] and text analytics [MNV$^+$17]. Eq. (2.17) and (2.18) show the computations for GEMV and SpMV.

$$\mathbf{y}(i) = \sum_{j=0}^{J-1} \mathbf{A}(i,j) * \mathbf{b}(j) \tag{2.17}$$

$$\mathbf{y}(i) = \sum_{\substack{j \,\in\, \{j \mid \\ \mathbf{A}(i,j) \neq 0\}}} \mathbf{A}(i,j) * \mathbf{b}(j) \tag{2.18}$$

CHAPTER 3
T2S-TENSOR: PRODUCTIVELY GENERATING HIGH-PERFORMANCE SPATIAL HARDWARE FOR DENSE TENSOR COMPUTATIONS

This chapter explores hardware acceleration of dense tensor computations on spatial hardware such as FPGAs and CGRAs. T2S-Tensor provides a language and compilation framework for productively generating high-performance systolic arrays for dense tensor kernels on spatial architectures. T2S-Tensor decouples a functional specification from a spatial mapping, allowing programmers to quickly explore various spatial optimizations for the same function. The actual implementation of these optimizations is left to a compiler. Thus, productivity and performance are achieved at the same time.

3.1 Introduction

High-performance computing (HPC) on spatial architectures tends to be limited by very low design productivity — it is not unusual for industry experts to spend several months or even more than a year to deliver one seemingly simple kernel with good quality of results (QoRs). While HPC programming is presumably challenging on any architecture including CPUs/GPUs, it is especially painful on spatial architectures like field-programmable gate arrays (FPGAs) due to their much longer compile time and primitive debugging support [Ron17b]. In addition, FPGA designs often require construction of specialized user-managed on-chip memory hierarchy, which significantly increases the programming complexity. Coarse-grain reconfigurable architectures (CGRAs) have been proposed to address the slow compilation problem, but how to efficiently program CGRAs remains an open question.

To address these long-standing challenges, I propose a novel programming system consisting of a language and compiler, named T2S(Temporal To Spatial)-Tensor, for productively generating high-performance spatial hardware for dense tensor computations. I observe that *a dense tensor computation suitable for a spatial architecture is usually a dataflow function, and a high-performance spatial design partitions the computation into many sub-computations. These sub-computations are distributed over the spatial architecture, and are connected with channels, i.e. FIFOs. They run in parallel, and are individually optimized by a series of loop and data transfor-*

mations. Based on this observation, T2S-Tensor enables programmers to describe a tensor computation in a functional notation, followed by a spatial mapping that describes compute partition and loop and data transformation optimizations. A compiler then composes these optimizations and synthesizes them to run on a spatial architecture. Thus, T2S-Tensor allows programmers to succinctly *specify* different optimizations and leave the actual implementation of the optimizations to the compiler.

Well-known dense tensor kernels include, for example, general matrix multiply (GEMM), tensor times matrix (TTM), matricized tensor times khatri-rao product (MTTKRP), and tensor times matrix-chain (TTMc) as described in Chapter 2. These dense tensor kernels have regular memory access patterns and high parallelism, making them a good match for spatial architectures. Yet how to productively accelerate the tensor kernels on spatial architectures for high performance remains a challenge. First, tensors often have many dimensions and a tensor kernel can involve many tensors. Second, every tensor kernel has many possible designs. Implementing any design efficiently on spatial hardware like FPGAs usually takes significant engineering effort.

In this chapter, I show that T2S-Tensor enables dense tensor kernels to be succinctly expressed and effectively optimized.

The major technical contributions of this chapter are as follows:

- I propose a concise yet expressive programming abstraction that decouples a spatial mapping from the functional specification of a dense tensor computation. The spatial mapping can direct the compiler to realize many sophisticated optimizations, achieving productivity and performance at the same time.
- I identify a set of key compiler optimizations that are essential for creating high-performance spatial hardware for dense tensor computations, and implement them in a comprehensive compilation framework. The compiler further provides composability of these optimizations, where various combinations of transformations can be applied and the compiler automatically generates the correct low-level code.
- I demonstrate the efficacy of T2S-Tensor approach on an Arria-10 FPGA as well as a research CGRA by generating OpenCL and assembly code, respectively for the two architectures. The GEMM implementations using T2S-Tensor achieve 88% and 92% of the performance achieved by codes that were manually written, highly optimized, and extensively tuned by

experts, with only around 3% of engineering time (two weeks vs. 18 months on the FPGA, and three days vs. three months on the CGRA, respectively). The T2S-Tensor GEMM implementation on the Arria-10 FPGA is also 76% faster than a (tuned) NDRange-style OpenCL implementation [Inta].

- Using this language, dense tensor factorization kernels, MTTKRP, TTM, and TTMc were also implemented in a productive and high-performance manner, and for the first time on spatial architectures.

The rest of the chapter is organized as follows: Section 3.2 gives a brief overview of the programming model; in Section 3.3 I illustrate the spatial optimizations in this system with GEMM as a working example; the compiler flow and optimizations are described in Section 3.4, followed by experimental results in Section 3.5, and related work in Section 3.6. Finally, Section 3.7 concludes this chapter.

3.2 T2S-Tensor: Overview of the Programming Model

A T2S-Tensor program is a specification embedded in C++. It consists of two parts: a temporal definition and a spatial mapping. The former defines a tensor computation functionally, while the latter specifies how to map the computation to a spatial architecture. In this chapter, I describe the programming abstraction with small intuitive examples to facilitate understanding. The same principle can be applied to construct very sophisticated spatial designs.

Fig. 3.1(a) shows a trivial computation that computes a function `B` with an input vector `A`. To map it to a spatial architecture, I isolate two functions: `A_loader` for loading the input values and `B_unloader` for saving the computed values from function `B`. This is called *compute partition*, which leads to a spatial layout shown in Fig. 3.1(b).

A corresponding T2S-Tensor specification is shown in Fig. 3.1(c). Lines 1-4 are the temporal definition expressing the original computation, where `A` is a one-dimensional single-precision floating-point vector, `i` is a variable, and `B` is a function that is to be run on the device. The upper loop bound of 100 is set upon execution, which is not shown.

Lines 5-7 are the spatial mapping. `A_loader` and `B_unloader` are the two new functions on the device, isolated out of function `B` as a producer of the input values in `A` and a consumer of the computed values in B, respectively. A more detailed intermediate representation (IR) after compute partition is shown in Fig. 3.1(d), where RCH and WCH are primitives of reading and writing a channel. All the functions have exactly the same loop structure. However, function `A_loader` does nothing but loads the values of A and sends them to a channel, `channel1`. Function B reads data from `channel1`, performs computation, and sends the results to `channel2`. Function `B_unloader` reads data from `channel2`, and stores it into memory.

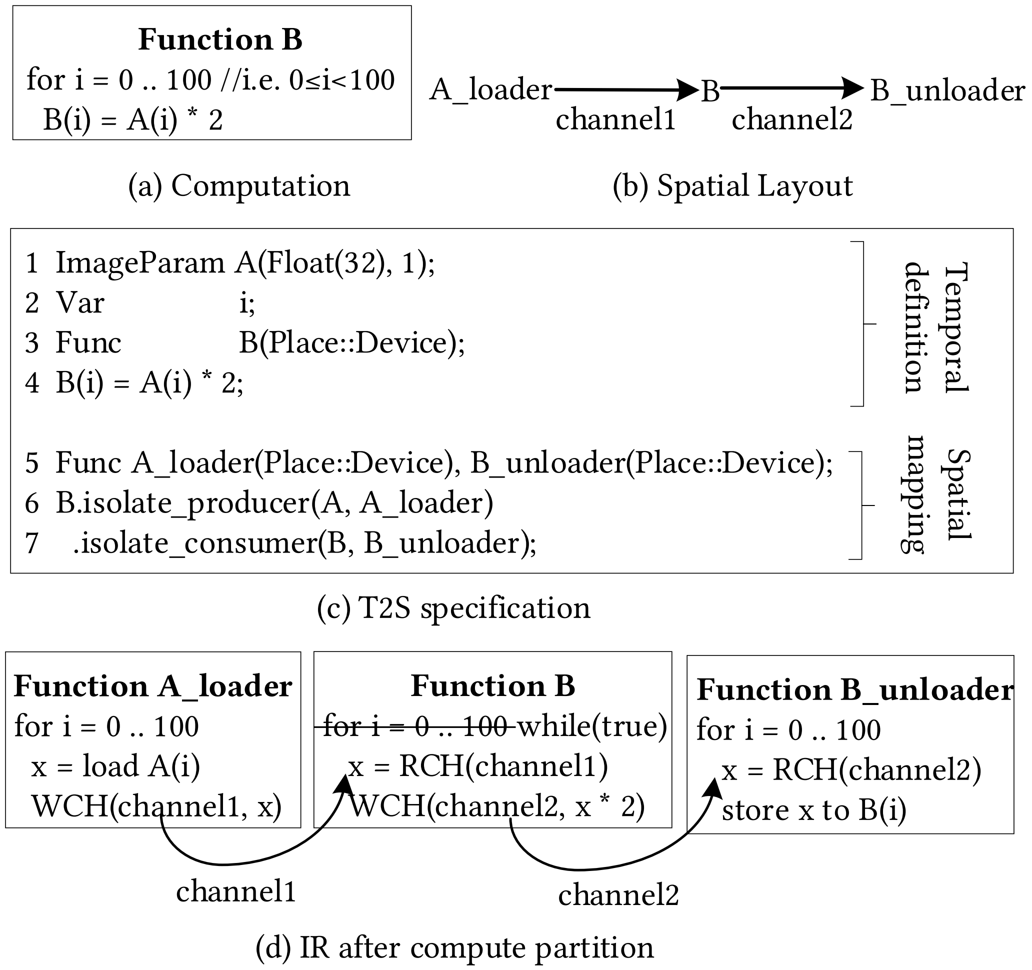
The compute partition maintains the semantics of the original computation. Its purpose is to partition a computation into sub-computations, which in turn become accessible to optimizations and can be specialized individually (Section 3.4). For example, in the middle box of Fig. 3.1(d), the compiler finds that the loop variable `i` is no longer used, and all the inputs are from `channel1`. Therefore, the compiler may automatically replace the loop with an infinite loop, `while(true)`, and the loop automatically executes (or stops) if data are (not) available in `channel1`. This optimization is called *loop infinitization*.

Assuming a temporal definition, usually including simple math equations, is correctly specified by the programmer, the compiler checks the spatial directives to ensure that they do not violate the semantics of the temporal definition. This provides a correctness guarantee of the generated hardware, and composability of the directives.

T2S-Tensor is built on Halide [RKBA$^+$13], a domain-specific language (DSL) for image processing on CPUs and GPUs. A key strength of Halide is to decouple a functional specification from optimizations. Many important loop-nest optimizations (e.g., loop reordering and tiling) can be easily specified in Halide. T2S-Tensor extends Halide to spatial architectures with the following optimizations:

- *Spatial layout*: compute partition.
- *Loop transformations*: loop unrolling[1], flattening, perfectization, infinitization, and removal.
- *Data transformations*: data forwarding, scattering, gathering, and vectorization.
- *Data caching*: single/double buffer insertion.
- *Control*: overlapping draining and filling of a pipeline, and drain signal generation.

[1]Different from unrolling on CPUs/GPUs, unrolling a loop of *n* iterations on a spatial architecture will create *n* Processing Elements (PEs) in hardware, each for an iteration, and input/output channels will also be unrolled (Section 3.4).



```
Function B
for i = 0 .. 100 //i.e. 0≤i<100
  B(i) = A(i) * 2
```

(a) Computation

(b) Spatial Layout

```
ImageParam A(Float(32), 1);
Var        i;
Func       B(Place::Device);
B(i) = A(i) * 2;

Func A_loader(Place::Device), B_unloader(Place::Device);
B.isolate_producer(A, A_loader)
  .isolate_consumer(B, B_unloader);
```

(c) T2S specification

```
Function A_loader
for i = 0 .. 100
  x = load A(i)
  WCH(channel1, x)
```

```
Function B
~~for i = 0 .. 100~~ while(true)
  x = RCH(channel1)
  WCH(channel2, x * 2)
```

```
Function B_unloader
for i = 0 .. 100
  x = RCH(channel2)
  store x to B(i)
```

(d) IR after compute partition

Figure 3.1: An Illustration of T2S-Tensor Approach

3.3 T2S-Tensor: High-Level Illustration of the Optimizations

This section introduces the spatial optimizations that can be specified with T2S-Tensor, using a high-performance FPGA-targeted GEMM design as the driving example. For ease of understanding, the optimizations are illustrated at a high level, and the compiler details are left to Section 3.4. As the optimizations can be freely composed together, one may use these optimizations to specify many different designs for the same workload. The design illustrated here is thus only one of the possibilities. In general, programmers can start from a simple design with no or few optimizations, and gradually evolve it into more complicated designs by specifying more optimizations.

3.3.1 Temporal Definition

Consider matrix multiplication `C = A * B`. A temporal definition for this is shown in Fig. 3.2(a). Here `A` and `B` are single-precision floating-point matrices, `i` and `j` are variables, and `k` is a reduction domain of `[0, K * KK * KKK)` (Lines 1-3). In this example, `I, II, III, J, JJ, JJJ, K, KK`, and `KKK` represent some compile-time constants, and the input matrices are assumed to have a row-major storage format.

The output function `C` includes an *initial definition* (Line 5), and an *update definition* (Line 6) referred to as `C.update()`. From now on, this section will focus on transforming the update definition for high performance. For convenience, function `C` will refer to the update definition of function `C`, unless stated otherwise.

With the temporal definition, one gets a familiar three-level loop nest as shown in the left side of Fig. 3.2(b). This initial implementation does not exploit data locality, which is critical for achieving high performance. To optimize for locality, Lines 9-10 tile every loop twice to generate a nine-level loop nest, as shown in the right side of Fig. 3.2(b).

3.3.2 Evolving the Initial Specification into a High-Performance Spatial Design

Now let us see how the basic tiled loop nest (the right side of Fig. 3.2(b)), which has no notion of “space” at all, can be transformed to a high-performance spatial design. Fig. 3.3 abstractly visualizes the major spatial optimizations, with the details to be described in the next section.

Compute partition (Fig. 3.3(a)): It isolates from function `C` the loading of matrix `A` into another function named `A_feeder`. These two functions are automatically connected via a channel after the isolation.

Loop unrolling (Fig. 3.3(b)): It unrolls loops `ii` and `jj` of function `C`. This results in an `II` $\times$ `JJ` array of PEs for function `C`. The input channel from `A_feeder` is also replicated accordingly.

Data forwarding (Fig. 3.3(c)): `A(_i, _k)` can be shared when multiplied with `B(_k, _j)`, `B(_k, _j + 1)`, etc. to compute `C(_i, _j)`, `C(_i, _j + 1)`, etc. To minimize redundant memory accesses, let `A_feeder` send the data of matrix `A` to the boundary `C` PEs (`jj = 0`), which receive and forward the data to their neighbor PEs in `jj` direction. The neighbors receive and

```
ImageParam A(Float(32), 2), B(Float(32), 2);
Var        i, j;
RDom       k(0, K * KK * KKK);
Func       C;
C(i, j)  = 0;
C(i, j) += A(i, k) * B(k, j);

Var ii, jj, iii, jjj;
RDom kk, kkk;
C.update().tile(k, j, i, kk, jj, ii, KK * KKK, JJ * JJJ, II * III)
          .tile(kk, jj, ii, kkk, jjj, iii, KKK, JJJ, III);
```

(a) Temporal definition

```
for i = 0 .. I * II * III
  for j = 0 .. J * JJ * JJJ
    for k = 0 .. K * KK * KKK
      C(i, j) += A(i, k) * B(k, j)
```

$\xrightarrow{\text{tiling}}$

```
for i = 0..I
 for j = 0..J
  for k = 0..K
   for ii = 0..II
    for jj = 0..JJ
     for kk = 0..KK
      for iii = 0..III
       for jjj = 0..JJJ
        for kkk = 0..KKK
         _i = i*II*III + ii*III + iii
         _j = j*JJ*JJJ + jj*JJJ + jjj
         _k = k*KK*KKK + kk*KKK + kkk
         C(_i, _j) += A(_i, _k) * B(_k, _j)
```

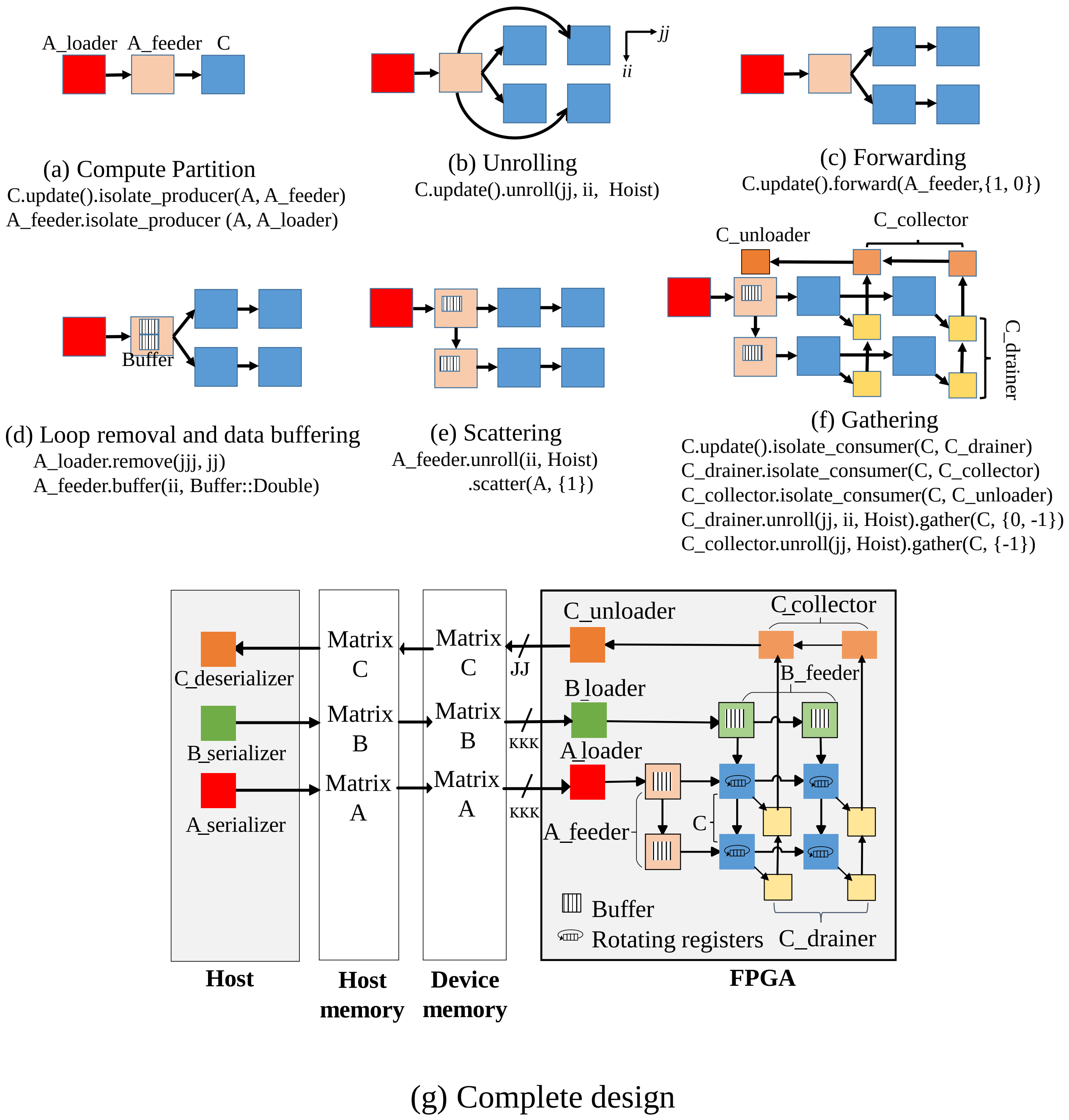
(b) Function `C` before and after tiling

Figure 3.2: Temporal Definition of Matrix-Matrix Multiply, Including Tiling

forward the data to their own neighbors in `jj` direction, and so on. This effectively allows the `jj` loop in `A_feeder` to be removed, reducing the total off-chip accesses of matrix `A` by `JJ` times.

Buffering (Fig. 3.3(d)): It further isolates the accesses of matrix `A` from `A_feeder` to another function `A_loader`. To further minimize redundant memory accesses, the `jjj` loop in `A_loader` can be removed, because it does not appear in the subscripts of the reference to matrix `A`. Such a loop is called a *reuse loop*. This further reduces the total accesses of matrix `A` by `JJJ` times.

However, because `A_loader` sends less data to its consumer after the loop removal, the data stream produced is not what is expected by the consumer. To restore the correct data stream, a (double) buffer is inserted in the consumer, `A_feeder`, at an appropriate loop level so that `A_feeder` accepts the data from `A_loader`, stores the data in an on-chip buffer, and sends the buffered data



(a) Compute Partition

```
C.update().isolate_producer(A, A_feeder)
A_feeder.isolate_producer (A, A_loader)
```

(b) Unrolling

```
C.update().unroll(jj, ii,  Hoist)
```

(c) Forwarding

```
C.update().forward(A_feeder,{1, 0})
```

(d) Loop removal and data buffering

```
A_loader.remove(jjj, jj)
A_feeder.buffer(ii, Buffer::Double)
```

(e) Scattering

```
A_feeder.unroll(ii, Hoist)
        .scatter(A, {1})
```

(f) Gathering

```
C.update().isolate_consumer(C, C_drainer)
C_drainer.isolate_consumer(C, C_collector)
C_collector.isolate_consumer(C, C_unloader)
C_drainer.unroll(jj, ii, Hoist).gather(C, {0, -1})
C_collector.unroll(jj, Hoist).gather(C, {-1})
```

(g) Complete design

Figure 3.3: Illustrating Some Major Optimizations With GEMM as an Example. For Each Optimization, the Corresponding T2S-Tensor Specification is Also Shown

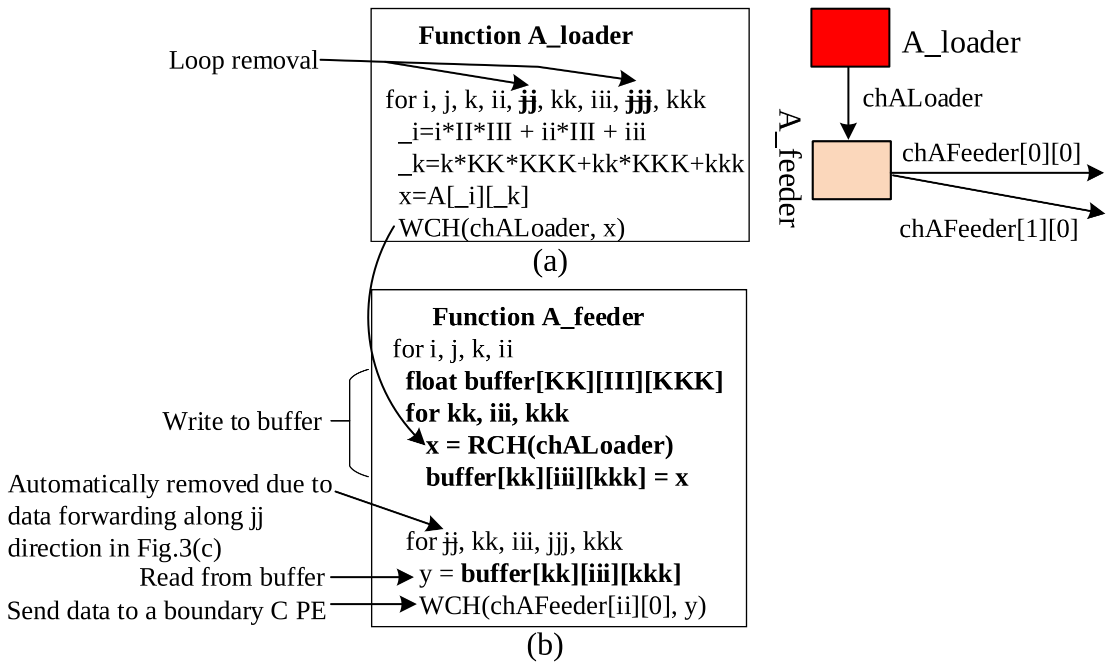


Figure 3.4: Loop Removal and Buffer Insertion

for `JJJ` times to `C` in the right order. This restores the correct dataflow relationship that was broken by loop removal. Fig. 3.4 shows the IR after loop removal in `A_loader` and buffer insertion in `A_feeder`.

The loop level in a consumer at which the buffer is inserted must enclose the outermost loop that is removed in the producer but not in the consumer. In case of `A_loader` → `A_feeder`, this loop is `ii` and hence a buffer is inserted at loop level `ii` in `A_feeder`.

How about removing loop `j` from `A_loader`, which is also a reuse loop? That will further reduce the total accesses of matrix `A` by `J` times. This is feasible with a bigger on-chip buffer. However, in this example I assume that loop `j` is not removed due to insufficient on-chip buffering.

Data scattering (Fig. 3.3(e)): A single `A_feeder` transferring data to all the boundary PEs of `C` will not scale for bigger FPGAs. Unroll loop `ii` in `A_feeder` such that every `A_feeder` PE feeds a boundary `C` PE (`jj = 0`). The first `A_feeder` PE keeps the data needed by the first boundary PE of `C`, and streams the rest of the data to the next `A_feeder` PE, which works similarly. Fig. 3.8 shows the IR and the corresponding dataflow for data scattering.

Data gathering (Fig. 3.3(f)): Directly writing the results to the memory from each of the `C` PEs requires a large cross-bar, which can impact timing. To avoid this, I isolate from `C` the consumer of the computed `C` values to a function, `C_drainer`. Then I isolate from `C_drainer` the consumer of the computed `C` values to another function, `C_collector`. I unroll `C_drainer` into a 2-dimensional array of PEs, each accepting the results of a `C` PE. Each column of drainer PEs relays the results upward. I unroll `C_collector` into a 1-dimensional array of PEs, each reading the data from a column of drainer PEs, and merging them with the data from its right, and then sending the results to its left.

Data vectorization is not illustrated, but is important for performance. That is, I load matrix `A` in vectors of data, vectorize related channels and operations, and save the results in vectors as well. For the GEMM example, load vectorization can be specified by `C.update().vload(A)` before compute partition so that after partition, all the isolated functions are automatically vectorized; finally, store vectorization can be specified by `C_collector.vstore(C)`.

Besides the optimizations that can be specified, there are optimizations that are automatically done by the compiler. For example, after flattening a loop nest into a single loop, the loop may be pipelined efficiently, and hence the T2S-Tensor compiler automatically flattens nested loops.

The optimizations can also be specified to compose some other optimizations, for example, data serialization and de-serialization. In GEMM, matrix `A` is not visited sequentially (one can determine this from the tiled loops in Fig. 3.2(b)). To maximize the usage of memory bandwidth, I isolate from `A_loader` another function `A_serializer`. Unlike all the functions seen so far that are on the device, `A_serializer` can be declared to run on the host. `A_serializer` reads matrix `A` once, and sends the values via a *memory channel* to `A_loader` that is on the device. The memory channel is a virtual FIFO that the T2S-Tensor system automatically constructs with the host and device memory. This enables `A_loader` to visit the device memory completely sequentially. Similarly, `C_unloader` can send the results using a memory channel to a new function `C_deserializer` on the host that stores the results into the host memory in the right order (i.e., data de-serialization).

So far this section only describes optimizations for matrix `A`. Similar optimizations can be applied to matrix `B` to get the complete design shown in Fig. 3.3(g).

```
C(j, i) = 0.0f;
C(j, i) += A(k, i) * B(j, k);
C.tile(j, i, jj, ii, JJ, II).tile(jj, ii, jjj, iii, JJJ, III);
C.update(0).tile(k, j, i, kk, jj, ii, KK, JJ, II).tile(kk, jj, ii, kkk, jjj, iii, KKK, JJJ, III);
C.update(0).isolate_producer_chain(A, A_serializer, A_loader, A_feeder)
           .isolate_producer_chain(B, B_serializer, B_loader, B_feeder)
           .isolate_consumer_chain(C, C_drainer, C_collector, C_unloader, C_deserializer);
A_serializer.sread().swrite();
B_serializer.sread().swrite();
C.update(0).vread({A,B});
C.update(0).unroll(ii)
           .unroll(jj)
C.update(0).forward(A_feeder, { 0, 1 })
           .forward(B_feeder, { 1, 0 });
A_serializer.remove(jjj, jj, j);
A_loader.remove(jjj, jj);
A_feeder.buffer(ii, true).unroll(ii);
B_serializer.remove(iii, ii, i);
B_loader.remove(iii, ii);
B_feeder.buffer(k, true).unroll(jj);
A_feeder.scatter(A, {1});
B_feeder.scatter(B, {1});
C_drainer.unroll(ii).unroll(jj).gather(C, jjj, { 1, 0 });
C_collector.unroll(jj).gather(C_drainer, jjj, { 1 });
```

Figure 3.5: T2S-Tensor Code for High-Performance SGEMM Design

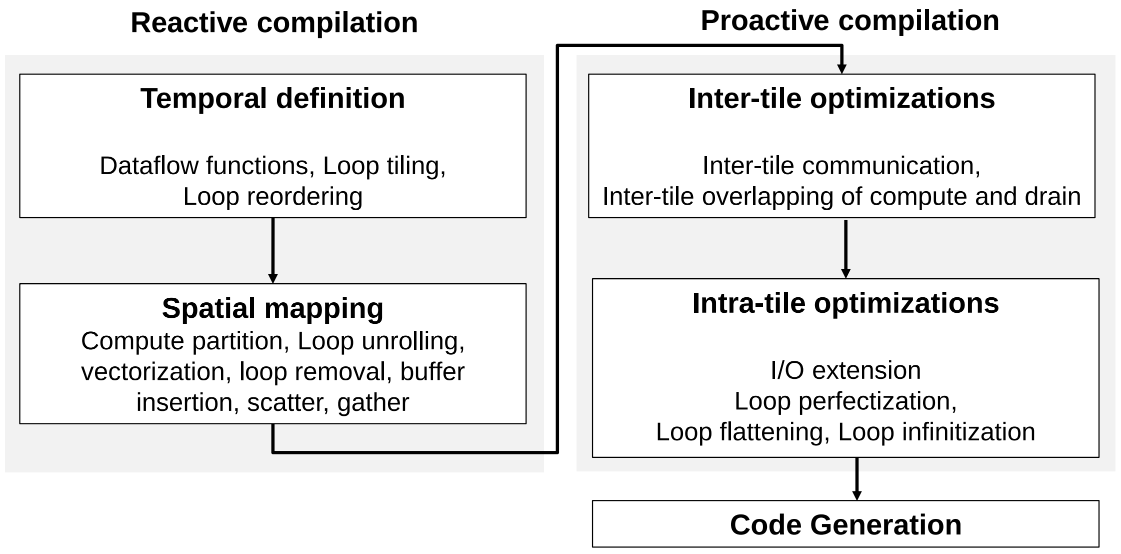

Reactive compilation
Proactive compilation
Temporal definition
Dataflow functions, Loop tiling, Loop reordering
Spatial mapping
Compute partition, Loop unrolling, vectorization, loop removal, buffer insertion, scatter, gather
Inter-tile optimizations
Inter-tile communication, Inter-tile overlapping of compute and drain
Intra-tile optimizations
I/O extension
Loop perfectization,
Loop flattening, Loop infinitization
Code Generation

Figure 3.6: T2S-Tensor Compiler Flow

3.4 T2S-Tensor: Compiler Flow and Optimizations

T2S-Tensor leverages the Halide compiler [RKBA$^+$13] and extends it to spatial architectures. Fig. 3.6 shows the overall flow of the T2S-Tensor compiler. The compiler works in two modes: first in reactive, then proactive mode.

In the reactive mode, the compiler reads a programmer's specification, which includes a temporal definition and spatial mapping. The compiler builds an IR according to the temporal definition, and transforms the IR according to the spatial mapping. The compiler is reactive, since its work is purely directed by the specification.

After applying user-specified optimizations, the compiler automatically switches into the proactive mode, and performs optimizations transparent to the programmer. Finally, it generates code for the target architecture.

The following subsections describe the compiler in more detail.

3.4.1 Reactive compilation

The compiler builds up an IR according to the temporal definition. The temporal definition defines a computation with some dataflow functions, and often, also specifies how to tile or reorder the loops of the functions for better data locality or reuse.

The compiler then transforms, or equivalently annotates, the IR according to the spatial mapping. Typically, the spatial mapping specifies how to partition the computation, which loops to unroll, which loops to remove, how to manipulate data on the device including buffering, forwarding, gathering and scattering, and vectorizing of loads/stores from/to the off-chip device memory.

Compute partition – The compiler copies the IR to a new IR as a producer (or consumer), prunes the new IR so that it contains only the part of the original tree that generates the input (or consumes the output) values of the original tree. The compiler creates a channel and substitutes two nodes in the IRs with the channel, such that the values flow between the two IRs through the channel. The new IR inherits the annotations of the original IR, which implies that the loop nests the IRs encode have the same loop structure and are optimized (e.g., tiled) in the same way. This ensures the values are communicated in the right order over the newly-created channel. If the new IR is a consumer, reduction loops are automatically removed, as the consumer is supposed to consume the final results after reduction is done.

Loop unrolling – The unrolled loops are annotated as `unrolled` and hoisted to the outermost levels: all other loops become the body of the unrolled loops. Later in code generation, every iteration of the unrolled loops is turned into a hardware PE. With loop unrolling, channels in the IR have to be unrolled as well. Suppose there is a channel `ch` between a producer and a consumer. If loop x in the producer is unrolled `X` times, the channel is split correspondingly into an array of `X` number of channels, such that instead of writing to channel `ch`, the x'th PE writes to the x'th channel in the new channel array, denoted by `ch[`x`]`. The consumer is also changed to read from `ch[`x`]`. More generally, given a pair of producer and consumer, where loops `{x,y}` are unrolled in the producer and loops `{y,z}` are unrolled in the consumer, the compiler unrolls the channel between them `X` $\times$ `Y` $\times$ `Z` times (product of the trip counts of the union of the unrolled loops from both the producer and consumer).

Loop removal and data buffering – Loop removal is usually accompanied by data buffering, as discussed in Section 3.3. The compiler removes a loop in a producer IR, and inserts a buffer at a loop level in a consumer IR. The compiler calculates the buffer size with the references of the data in the body of the loop level in the consumer IR, and change all the references in the body to refer to the buffer elements. Fig. 3.4 shows the loop removal and buffer insertion for `A_loader` and `A_feeder`, respectively. Here, loops `jj,jjj` are removed from `A_loader` and a buffer of size $KK \times III \times KKK$ is inserted in `A_feeder`.

Data forwarding

Data forwarding is also called dependence localization [LVMQ91]. The channels are renamed so that a producer sends data only to the boundary PEs of its consumer, and every boundary PE broadcasts the data it receives to the other PEs along a given direction as shown in Fig. 3.7.

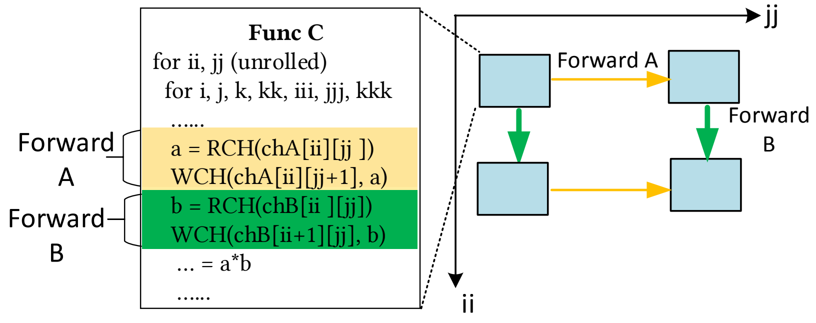
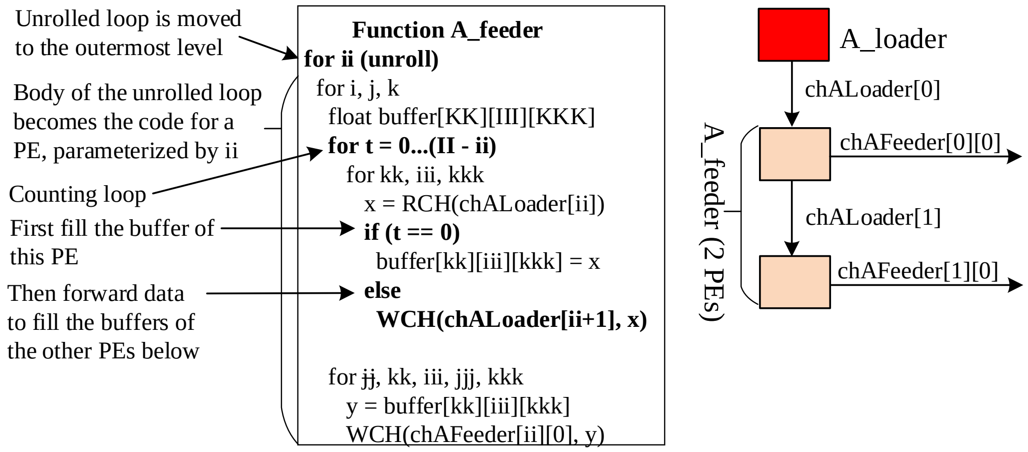


Figure 3.7: IR for Data Forwarding in C PEs

Data Scattering

For a PE to scatter data along a direction, the compiler automatically inserts a counting loop `t`, counting the total data received by the PE. Depending on the value of `t`, the received data are either used by the current PE, or forwarded to the next PE along the scattering direction. Fig. 3.8 shows the IR before and after applying data scattering optimization.



```
Function A_feeder
for ii (unroll)
  for i, j, k
    float buffer[KK][III][KKK]
    for t = 0...(II - ii)
      for kk, iii, kkk
        x = RCH(chALoader[ii])
        if (t == 0)
          buffer[kk][iii][kkk] = x
        else
          WCH(chALoader[ii+1], x)

    for ~~jj~~, kk, iii, jjj, kkk
      y = buffer[kk][iii][kkk]
      WCH(chAFeeder[ii][0], y)
```

Unrolled loop is moved to the outermost level
Body of the unrolled loop becomes the code for a PE, parameterized by ii
Counting loop
First fill the buffer of this PE
Then forward data to fill the buffers of the other PEs below
A_loader
chALoader[0]
chAFeeder[0][0]
A_feeder (2 PEs)
chALoader[1]
chAFeeder[1][0]

Figure 3.8: IR Change After Applying Data Scattering Optimization

Data Vectorization Loads and stores can be vectorized to access the device memory in chunks of multiple contiguous data elements. This reduces the number of memory accesses and improves the achieved memory bandwidth. Currently, only the data accesses in an innermost loop are allowed for vectorization. Since the data is communicated via channels, the compiler increases the width of the channels to match the width of vectorized data, and inserts a load before or a store after the innermost loop to read/write the data as a vector. All the operations in the innermost loop are changed to operate on the vector.

Data Gathering Similar to scattering, the compiler inserts a counting loop `t` at a specified loop level, and depending on the value of `t`, a PE decides to accept incoming data from the previous PE for the same function along the given direction, or from a producer function.

When the counting loop `t` is the innermost loop, and if it has been specified to vectorize the incoming data, the compiler does not generate an explicit counting loop, but instead, creates a small vector for each PE, and a PE writes the data into this vector before sending the vector to the next PE. This makes the gathering network faster, and eventually allows an entire vector to be written back to the device memory, reducing the number of memory accesses and improving memory bandwidth. This vectorized data gathering is a trade-off as it increases the width of the channels, which can also result in more resource usage.

Fig. 3.9 shows two gathering networks: one formed by `C_drainer` PEs, which has an explicit timing loop, another by `C_collector` PEs, which has vectorized gathering.

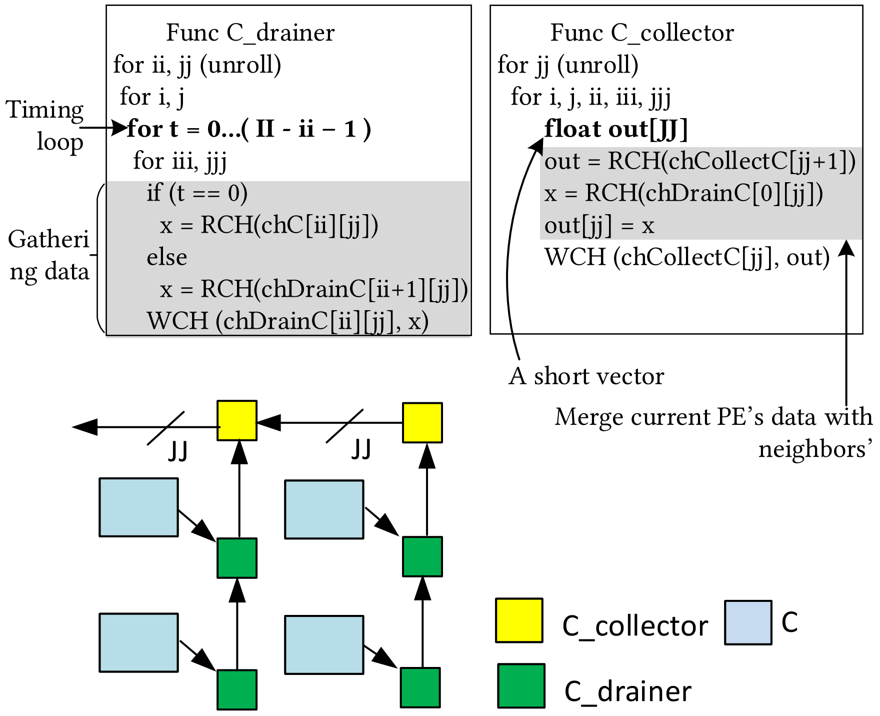


Figure 3.9: IR Change for Data Gathering

3.4.2 Proactive compilation

Usually, a problem is too big to fit on the on-chip memory, and thus has to be partitioned into tiles, and computed tile by tile, as specified by the programmer. In the proactive mode, the compiler automatically performs optimizations between tiles and within a tile, and finally generates code for the target architecture.

Overlapping drain and compute – In general, a systolic array of PEs (like the `C` PEs in GEMM) computes one tile of results, drains them, and computes the next tile of results, and so on. It is desirable to overlap the draining of one tile and the computation of the next tile. The compiler identifies all the reduction loops and inserts a local buffer right before the outermost reduction loop. This buffer contains the results for the current tile. The size of the buffer is calculated from the memory footprint of the results in the body of the outermost reduction loop. Then the compiler generates a drain signal and inserts code in the innermost loop to drain and re-initialize one buffer element every iteration while computing results for next tile at the same time. If the buffer has unit-stride cyclic access pattern, the buffer can be optimized into a rotating register file. Rotating registers remove the area overhead due to address calculation. The compiler changes the memory accesses to the buffer so that a read/write access occur only at the first/last element of the buffer. Then the compiler inserts code for rotation of the buffer after the access.

Loop perfectization – As many HLS compilers cannot pipeline a non-perfect loop nest efficiently, it is important to convert an imperfect loop nest into a perfect loop nest. The compiler moves an operation at an outer loop level into an inner level by predication. When there is more than one inner loop at the same level, the inner loops will be merged together as a single loop whose trip count is the sum of their individual trip counts, and the loop variable is used to construct a finite state mahcine (FSM) that controls the execution among the original inner loops' operations. Fig. 3.10 illustrates how an imperfect loop nest by transformed into a perfect loop nest in the T2S-Tensor compiler.

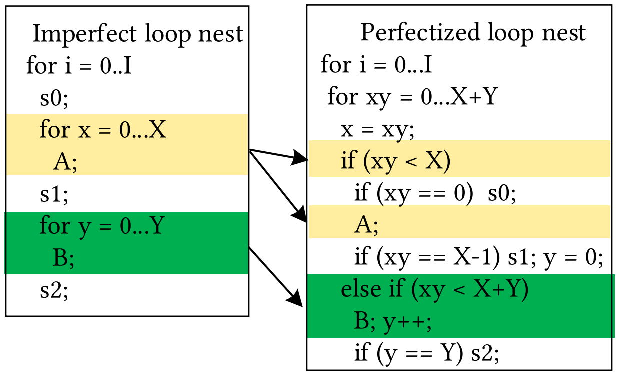


Figure 3.10: IR Change for Loop Perfectization

Loop flattening – As HLS compilers usually pipeline the innermost loop only, if the innermost loop has fewer number of iterations, the startup overhead for the pipeline can significantly degrade the performance. Hence, it is advisable to flatten a loop nest into a single loop to increase the number of iterations of the new innermost loop and amortize the startup overhead.

In the T2S-Tensor compilation flow, loop flattening happens automatically after loop perfectization. The compiler merges all the original loops into a single loop whose trip count is the multiplication of all the original loops' trip counts. Then the compiler inserts code to extract the original loop variables from the flattened loop variable. Fig. 3.11(a,b) show loop flattening where all loops' trip counts are powers of two, and Fig. 3.11(c,d) show loop flattening when all the loops' trip counts are not powers of two. Bit masks and shifts are used to extract the original loop variables, which are efficient for FPGAs. When an original loop's trip count is not a power of two, the flattened loop variable will "jump" to the next power-of-two value when that original loop is done, as shown in the last statement of Fig. 3.11(d).

So far, the compiler only supports loop flattening for the loops with constant trip counts. This does not effect the quality of the generated code, since after tiling, all loops but few outermost loops have constant trip counts.

```
  Unflattened
for x = 0..8
  for y = 0...4
    for z = 0...16
      A
```
(a)

```
  Flattened
for xyz = 0..512
  x = xyz >> 6
  y = (xyz & 63) >>4
  z = (xyz & 15)
  A
```
(b)

```
  Unflattened
for x = 0..4
  for y = 0...3
    for z = 0...4
      A
```
(c)

```
  Flattened
for xyz = 0..64
  x = xyz >> 4
  y = (xyz & 15) >> 2
  z = (xyz & 3)
  A
  if (xyz == 3*4)
    xyz += 4;
```
(d)

Figure 3.11: IR Change for Loop Flattening

Loop infinitization – As HLS compilers need to generate a finite state machine for each `for` loop, which adds significant area overhead, it is desirable to convert `for` loops to an infinite `while(1)` loop whenever possible. The compiler converts `for` loops to an infinite `while(true)` loop when all the input values in a PE are from channels, and the loop variable is no longer used anywhere. The infinite loop's execution is then controlled by data availability of the input channels. The T2S-Tensor compiler checks the following constraints to determine the feasibility of infinitizing a `for` loop:

1 Loops with loop variables being used in the loop body cannot be infinitized. In Fig. 3.12(a), loop `i` cannot be infinitized.

2 All the loops enclosing an infinitized loop have to be infinitized. In Fig. 3.12(a), as loop `i` cannot be infinitized, loop `j` cannot be infinitized, either.

3 Loops in the functions that are either the first producer or the very last consumer on the device cannot be infinitized. Infinitizing such loops may make the computation never finish. For example, `A_loader`, `B_loader`, and `C_unloader` in Fig. 3.3(g) cannot be infinitized.

4 Imperfect loops cannot be infinitized. In Fig. 3.12(b), loop `j` and `k` cannot be infinitized separately, as infinitizing loop `j` will prevent loop `k` from being executed and infinitizing loop `k` prevents further executions of loop `j` for a different `i`.

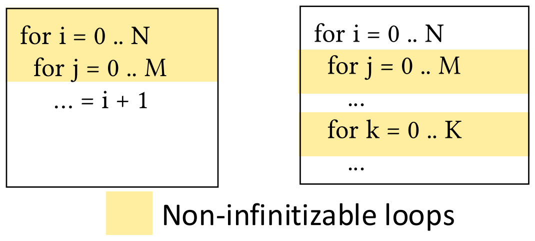


Figure 3.12: Loops That Cannot be Infinitized

Code generation – Finally, the compiler generates the target code. Currently, the compiler generates Intel (Altera) OpenCL code for Intel FPGAs, and assembly for a research CGRA. The OpenCL codes are further compiled by Intel (Altera) HLS compiler into bitstreams for FPGA. For CGRA, the assembly codes are place-and-routed by an assembler and simulated by a cycle-accurate simulator.

3.5 T2S-Tensor: Evaluation

I designed and wrote T2S-Tensor specifications for four important tensor kernels, including GEMM, MTTKRP, TTM and TTMc. Their definitions are shown in Table 3.1. GEMM is a core computation in many fields. MTTKRP is the computational bottleneck in canonical polyadic decomposition (CPD), and TTM and TTMc are the bottlenecks in Tucker decomposition algorithms.

Table 3.1: Definitions of the Tensor Kernels

GEMM	$C(i,j) += A(i,k) * B(k,j)$
MTTKRP	$D(i,j) += A(i,k,l) * B(k,j) * C(l,j);$
TTM	$C(i,j,k) += A(i,j,l) * B(l,k)$
TTMc	$D(i,j,k) += A(i,l,m) * B(l,j) * C(m,k)$

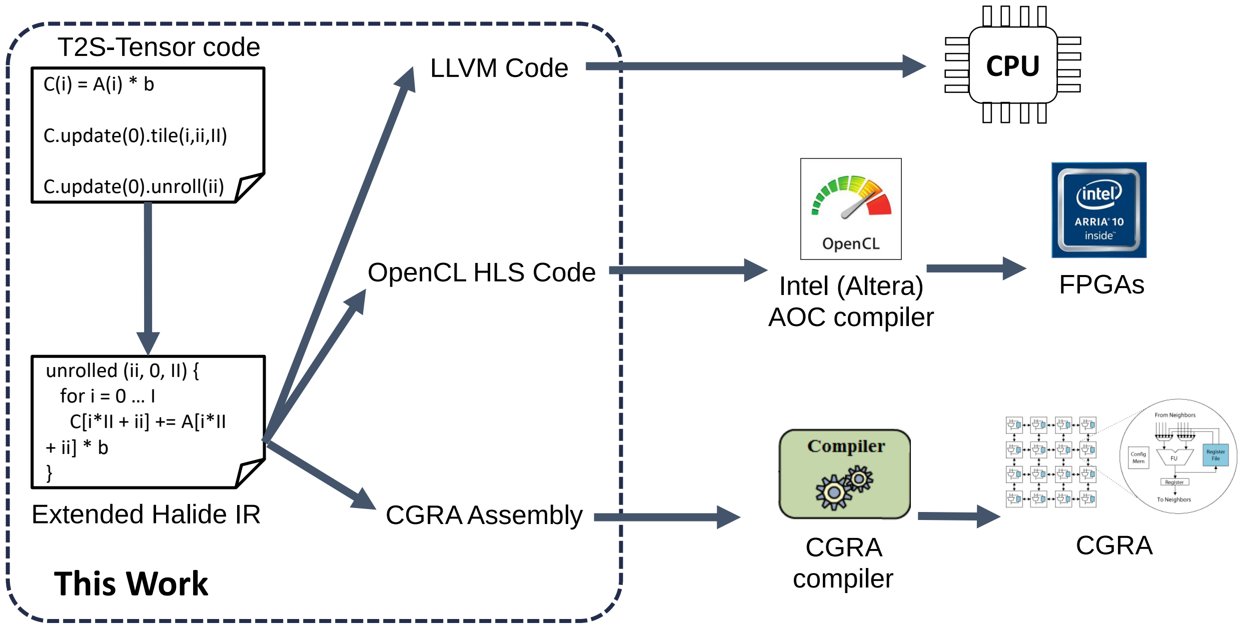


Figure 3.13: Complete Design Flow

The designs try to match the underlying hardware architectures for the maximum efficiency, and thus for the same kernel, its designs for an FPGA and the CGRA are different. While all four tensor kernels are dominated by multiply-add operations, they vary in data reuse and compute patterns, and must be individually designed for the best performance. Therefore I have created eight different designs, which is in fact a proof of the flexibility and generality of the proposed approach.

For FPGA experiments, I use the Intel vLab Academic Cluster [Intb] to access one Xeon CPU and an Arria 10 GX FPGA (10AX115N2F40E2LG). For CGRA experiments, I use a research CGRA based on the triggered-instruction architecture (TIA) [PPA$^+$15]. All the designs use the machines' native precisions, i.e., single and double-precision floats for the FPGA and CGRA, respectively.

For either architecture, a high-performance ninja implementation of GEMM was available — for FPGA it is a production design and implemented in OpenCL; for CGRA it is in assembly. Both implementations were manually written and highly optimized by industry experts, and have been extensively tuned for performance. These two designs took 18 and 3 months to develop, respectively. For FPGA, I also compare with an NDRange-style OpenCL implementation, a tutorial HLS design from Intel.

For the other workloads, namely MTTKRP, TTM and TTMc, I am not aware of any existing implementations on any spatial architectures. Since GEMM is a highly parallel and compute-intensive application, it provides an estimate of how close one can get to the peak throughput of the target hardware. Hence, I choose to use the performance of the ninja GEMM as a "yardstick" to roughly estimate if the achieved throughput of a tensor kernel is high.

For FPGA experiments, I further report the absolute performance (in GFLOPS) and resource usage. For CGRA experiments, I report performance relative to the ninja implementation of GEMM. In addition, I report productivity in terms of the engineering time and lines of code (LOC).

3.5.1 Overall Performance and Productivity

For most of the workloads on either architecture, using a small fraction of lines of code, the performance achieved by T2S-Tensor implementations is very close to or sometimes even higher than that of the ninja implementation of GEMM.

On the FPGA, the T2S-Tensor designs achieve an absolute throughput of 549, 700, 562 and 738 GFLOPS for GEMM, MTTKRP, TTM and TTMc, respectively. On average, the T2S-Tensor specifications have 29 LOC and achieve 637 GFLOPS. On the CGRA, the designs achieve 92%-104% of the performance of the fine-tuned GEMM using an average 38 LOC. Roughly, it took me approximately two weeks on the FPGA (due to long compile time for FPGA) and three days on the CGRA for engineering a workload, which is only 3% of the time spent by the experts who implemented the ninja GEMM.

Table 3.2: Comparison Between an NDRange OpenCL Baseline, T2S-Tensor and Ninja Implementations for GEMM on Arria-10 FPGA

	Baseline	**T2S-Tensor**	**Ninja**
LOC	70	20	750
Systolic Array Size	—	10×8	10×8
Vector Length	16×float	16×float	16×float
# Logic Elements	131K (31%)	214K (50%)	230K (54%)
# DSPs	1,032 (68%)	1,282 (84%)	1,280 (84%)
# RAM Blocks	1,534 (57%)	1,384 (51%)	1,069 (39%)
Frequency (MHz)	189	215	245
Throughput (GFLOPs)	311	549	626

3.5.2 Evaluation on the FPGA

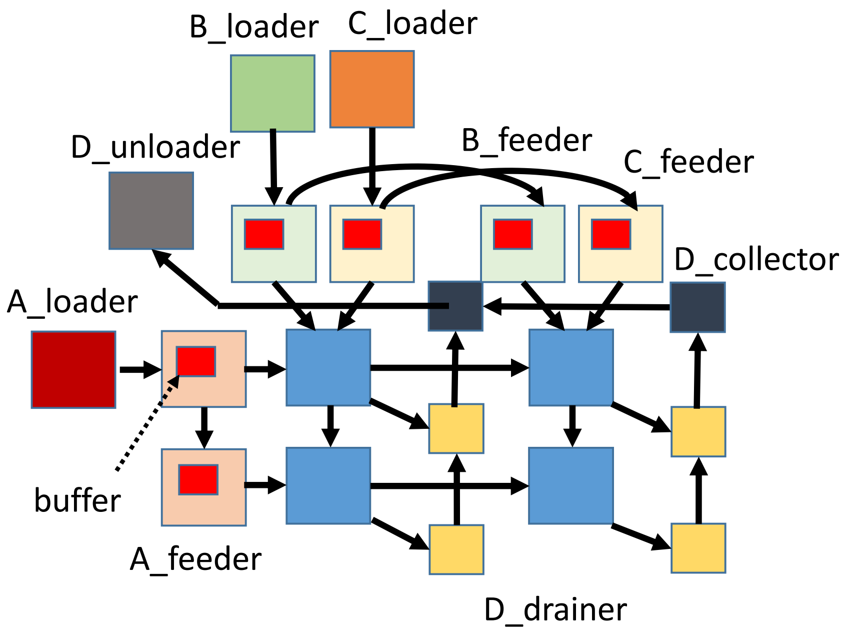


Figure 3.14: MTTKRP Design for FPGA

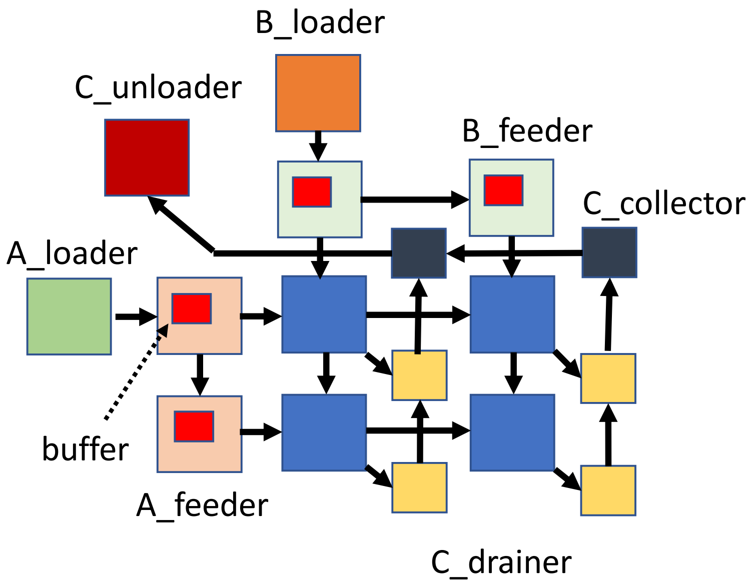


Figure 3.15: TTM Design for FPGA

Table 3.2 compares the NDRange-style OpenCL baseline [Inta], T2S-Tensor and ninja GEMM implementations. I have tuned the parameters of the baseline for the specific FPGA and show its best performance. Overall, T2S-Tensor GEMM design achieves 1.76× speedup over the baseline, and 88% of the performance of the manually written and highly optimized ninja implementation.

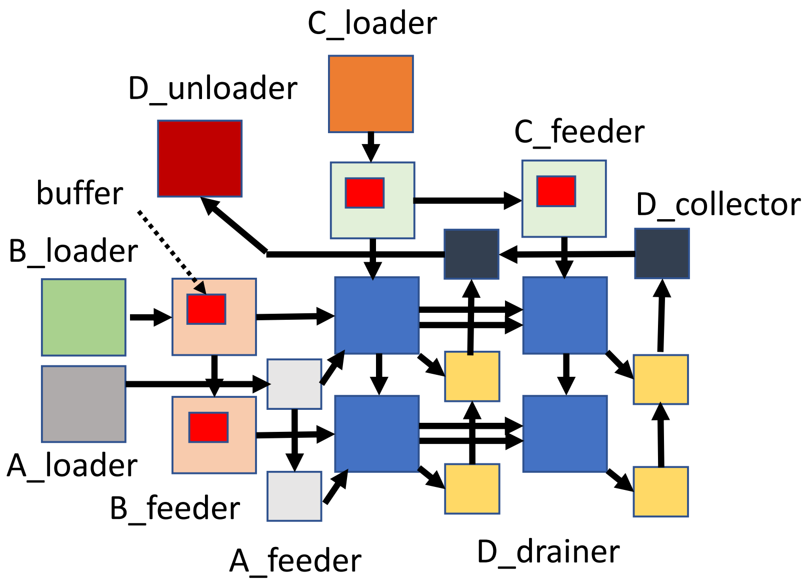


Figure 3.16: TTMc Design for FPGA

As one can see from Table 3.2, the baseline design is not able to efficiently utilize the DSP resources and achieve a lower clock frequency. The design uses NDRange kernels consisting of a 2-dimensional array of work-items (threads), each loading one data element from each of the input matrices and storing them in on-chip buffers, and computing one element of a tile of the output matrix. While these work-items are scheduled dynamically on the FPGA, only a subset of work-items can be executed simultaneously in lock step. The on-chip buffer requirement, however, is determined by the total number of work-items which leads to poor utilization of on-chip buffers and making it harder to fit a bigger design on the FPGA. In contrast, T2S-Tensor generates a systolic array that requires buffers only for the data that are being processed in the hardware and hence RAM usage remains well within the limits.

The T2S-Tensor GEMM design for the FPGA has been shown in Fig. 3.3(g). Here the parameters are set such that it has ten rows and eight columns of PEs in the core systolic array, i.e., `II`=10 and `JJ`=8. The loads are streamed in the unit of 16 floats every memory access, i.e., `KKK`=16. Due to vectorized gathering along the systolic array columns, 8 floats are stored to the memory simultaneously. Each PE computes 1024 results with `III`=`JJJ`=32. Similarly, for MTTKRP, TTM and TTMc each loop is tiled three times and two loops in the nested loop structure are unrolled to get a 2-D systolic array. Table 3.3 shows the parameters for all the four designs. Data loads from the memory and computation within each PE is vectorized by vectorizing the innermost loop. For vectorizing stores, vectorized gathering is applied in each design and hence the second dimension of the systolic array determines the vector length for stores.

For MTTKRP, all the four loops on `i`, `j`, `k`, and `l` are tiled three times, and `ii` and `jj` loops are unrolled. The final design consists of a 8×9 systolic array, i.e., `II`=8 and `JJ`=9. Loads and stores are done in the unit of 16 floats and nine floats respectively. Each PE computes 256 results with `III`=`JJJ`=16. To enable rotating registers for the compute buffers in PEs `KKK` is set to 1, which ensures that consecutive elements of the compute buffer are accessed in consecutive cycles. Since the MTTKRP computation, $D(i,j) = \sum_{k,l} A(i,k,l) * B(k,j) * C(l,j)$, can also be expressed as $D(i,j) = \sum_k B(k,j) * (\sum_l A(i,k,l) * C(l,j))$, which reduces the number of MAC (multiply-accumulate) operations at the cost of larger on-chip storage for intermediate results, I manually perform this optimization in the innermost loop of the generated OpenCL code, which is a tiny code change. This optimization reduces the number of operations to be performed in the hardware, which results in a high DSP utilization leading to a throughput even better than GEMM ninja implementation.

For TTM, loops `i`, `j`, `k` and `l` are tiled three times, loops `jj` and `kk` are unrolled with `JJ`=8 and `KK`=11 to form an 8×11 systolic array. Loads and stores are in units of 16 and 11 floats, respectively. Each PE computes 1024 results with `III`=`JJJ`=8 and `KKK`=16. To enable rotating registers, `II` is set to 1.

For TTMc, loops `i`, `j`, `k`, `l`, `m` are tiled three times, and `jj` and `kk` are unrolled loops with `JJ`=8 and `KK`=10, implementing an 8×10 systolic array. Loads are in units of 16 floats with `MMM`=16, and stores are done in units of ten floats. A PE computes 256 results with `III`=16 and `JJJ`=`KKK`=4. To enable rotating registers, `II` and `LLL` are set to 1. Similar to MTTKRP, TTMc i.e., $D(i,j,k) = \sum_{l,m} A(i,l,m) * B(l,j) * C(m,k)$, can be expressed as $D(i,j,k) = \sum_l B(l,j) * (\sum_m A(i,l,m) * C(m,k))$, and I manually implemented the optimization in the generated OpenCL code.

The designs for these workloads are shown in Fig. 3.14, 3.15 and 3.16 with their resource usage and performance shown in Table 3.4.

3.5.3 Evaluation on the CGRA

Fig. 3.17 shows the design for GEMM on CGRA. First, the triple loop nest is tiled, similar to the FPGA design, so that a tile fits into the CGRA. Then for a tile, loader PEs read matrix `A` from external memory, and send data to a feeder. The feeder has a "virtual buffer" inside,

Table 3.3: Parameters for GEMM, MTTKRP, TTM and TTMc Designs

	GEMM	MTTKRP	TTM	TTMc
i, j, k, l, m	32,32,32,-,-	8,4,16,16,-	8,4,4,16,-	8,4,4,4,4
ii, jj, kk, ll, mm	10,8,32,-,-	8,9,16,1,-	1,8,11,4,-	1,8,10,32,2
iii, jjj, kkk, lll, mmm	32,32,16,-,-	16,16,1,16,-	8,8,16,16,-	16,4,4,1,16
Unrolled Loops	*ii, jj*	*ii, jj*	*jj, kk*	*jj, kk*
LOC	20	28	30	37
Systolic Array Size	8×10	8×9	8×11	8×10
Compute Vector Length	16×float	16×float	16×float	16×float
Load Vector Length	16×float	16×float	16×float	16×float
Store Vector Length	10×float	9×float	11×float	10×float

Table 3.4: Evaluation Results of MTTKRP, TTM and TTMc in T2S-Tensor on FPGA

	# Logic Elements	# DSPs	# RAM Blocks	Frequency (MHz)	Throughput (GFLOPs)
MTTKRP	228K (53%)	1,224 (81%)	1,526 (56%)	204	700
TTM	265K (64%)	1,416 (93%)	2,394 (88%)	201	562
TTMc	229K (54%)	1,368 (90%)	1,679 (62%)	205	738

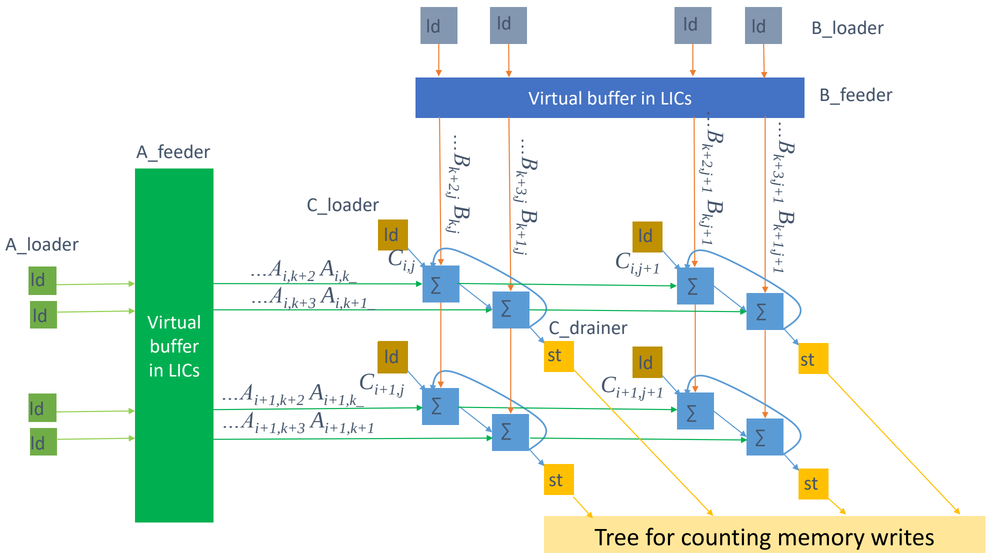

ld
B_loader
Virtual buffer in LICs
B_feeder
A_feeder
C_loader
A_loader
Virtual buffer in LICs
$...B_{k+2,j}\ B_{k,j}$
$...B_{k+3,j}\ B_{k+1,j}$
$...B_{k+2,j+1}\ B_{k,j+1}$
$...B_{k+3,j+1}\ B_{k+1,j+1}$
$C_{i,j}$
$C_{i,j+1}$
$C_{i+1,j}$
$C_{i+1,j+1}$
$...A_{i,k+2}\ A_{i,k}$
$...A_{i,k+3}\ A_{i,k+1}$
$...A_{i+1,k+2}\ A_{i+1,k}$
$...A_{i+1,k+3}\ A_{i+1,k+1}$
Σ
C_drainer
st
Tree for counting memory writes

Figure 3.17: GEMM Design for CGRA

which is functionally equivalent to a single buffer based on scratchpad memory, but is implemented using latency-insensitive channels (LICs) [FAP+12, PPA+15]. This basic construct is important to performance: without it, GEMM loses almost 60% of performance. The same mechanism is built for matrix B. The reduction loop k is unrolled so that several PEs are accumulating partial results

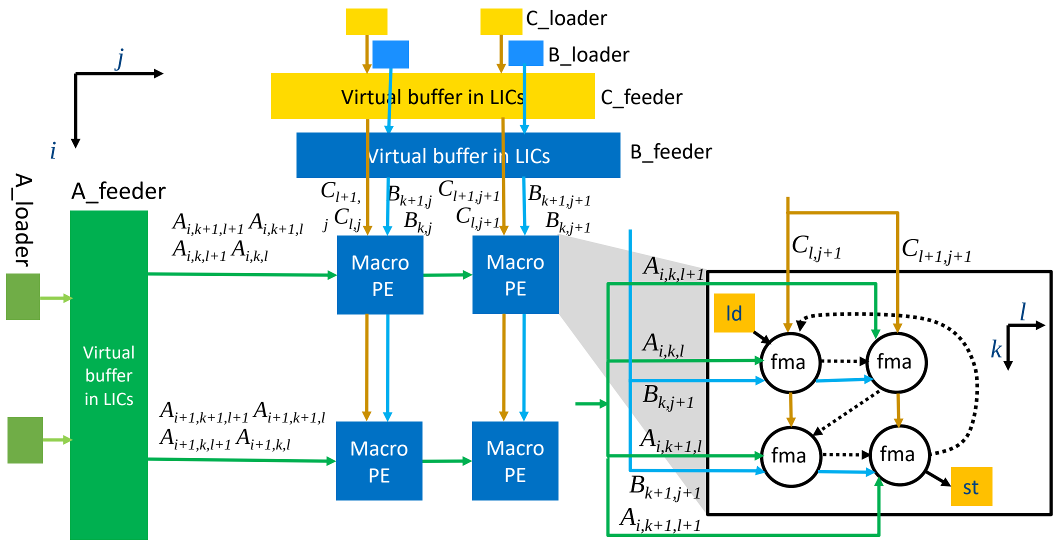


Figure 3.18: TTM Design for CGRA

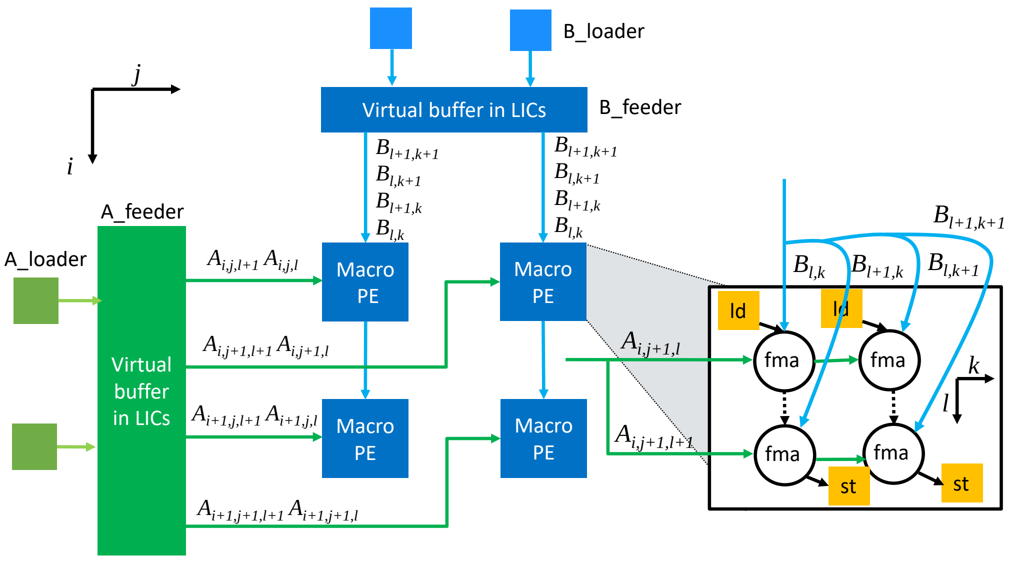


Figure 3.19: TTM Design for CGRA

of the same `C(i, j)` with different parts of the matrix `A` and `B` data. The initial values of `C(i, j)` are loaded from memory, and the final values are stored back to memory. One can visually see the big differences of this design from its FPGA counterpart. Unlike the FPGA design which is a 2-D systolic array, the CGRA design is 3-D systolic. This difference is due to the difference between the architectures — unrolling a loop results in a PE for each iteration of the unrolled loop; if the unrolled loop has inner loops not unrolled, a finite state machine (FSM) has to be constructed for each PE, which is easy to implement on an FPGA using LUTs, but is difficult to implement on a CGRA due to the limited number of control flow PEs. Hence, I fully unroll the innermost

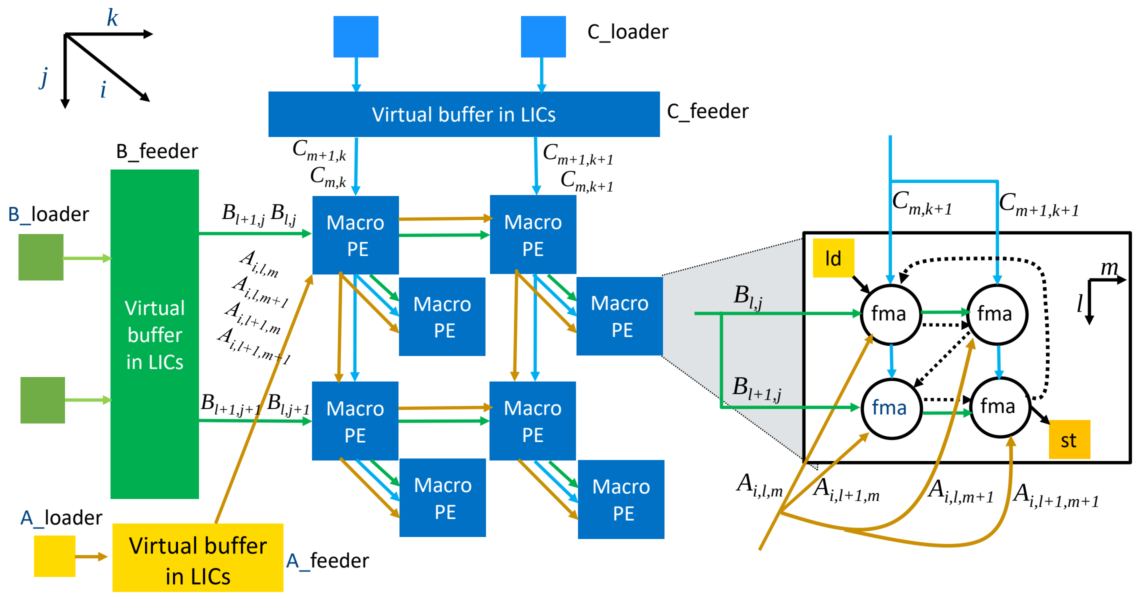


Figure 3.20: TTMc Design for CGRA

loop levels for the CGRA design to reduce the number of FSMs. This however increases the inter-PE data communication channels which are expensive on FPGAs but since the CGRA uses LICs, which are efficient for fine-grained data communication, the energy and performance cost is minimal.

Fig. 3.18 shows the MTTKRP design. Similar to the GEMM design, there are loaders and feeders (with virtual buffers) for all the input matrices. However, this is a 4-D systolic array where `i` and `j` dimensions correspond to the elements in the output matrix, while the `k` and `l` dimension correspond to reductions. Similarly, Fig. 3.19 and Fig. 3.20 are 4-D and 5-D systolic arrays for TTM and TTMc where (`i`, `j`, `k`) correspond to output tensor elements for both TTM and TTMc, and (`l`) and (`l`, `m`) are the reduction loops for TTM and TTMc, respectively.

Table 3.5 shows the LOC, performance and area results for all the four kernels. All the kernels have utilized 100% fused-multiply add (FMA), except TTMc, which uses 95% FMAs, achieving a performance very close to the ninja GEMM implementation. The average LOC for these designs is 38 while the ninja GEMM implementation uses 2,280 LOC. The usage of all the other resources for these designs are well within the hardware resource constraints.

Table 3.5: **Evaluation Results of T2S-Tensor Designs on the Research CGRA**

	LOC	Throughput w.r.t Ninja GEMM	FMA usage
GEMM	40	92%	100%
MTKRP	32	99%	100%
TTM	47	104%	100%
TTMc	38	103%	95%

3.6 Related Work

This section surveys a subset of the representative work on hardware description languages (HDLs), HLS languages, DSLs, and their associated compilers.

HDLs (e.g., Verilog and VHDL) describe a circuit at the register-transfer level (RTL) with explicit timing [TM08, Nav97]. They can be compared to "assembly languages". HLS languages have a higher abstraction. They accept an algorithmic description of a desired behavior without clock-level timing [GDWL12, CM08, CCA+11, CLN+11, BMG16]. T2S-Tensor code is more succinct and at a higher abstraction level than an HLS program. T2S-Tensor code controls a compiler to generate details, instead of letting the programmer directly write the details. Languages like Chisel [BVR+12], PyMTL [LZB14] and BlueSpec [Nik08] raise the level of hardware design abstraction by introducing concepts like object orientation, functional programming and guarded atomic actions in hardware design. Hot & Spicy [SMSF18] is another tool suite which translates python functions to HLS-suitable C functions and thus integrates FPGA accelerators in Python applications.

DSLs also have a higher abstraction level than HLS languages [VDKV00]. Such languages express and optimize an algorithm in predefined domain-specific patterns, and lower the patterns into an HDL [RKBA+13, PBY+17, KFP+18, LSY12, HBD+14, GLN+14]. The system presented in this Chapter extends Halide [RKBA+13], a DSL for image processing on CPUs/GPUs, to spatial architectures. Most of the optimizations I implemented are new to Halide as they are specific to spatial architectures. Some of them, like loop unrolling, extend the existing Halide implementation. Halide-HLS [PBY+17] is another spatial extension of Halide, where a dataflow graph of functions is specified to offload to an FPGA, with line buffers to optimize the communication. It currently focuses on image processing applications and has not implemented any of the tensor kernels I have.

Spatial [KFP$^+$18] is a language whose abstraction level is higher than T2S-Tensor. For example, for matrix multiply, Spatial specifies the multiplication between two values from two matrices, and lets the compiler determine an efficient systolic array for the computation. In comparison, T2S-Tensor lets the programmer specify a systolic array in detail. This simplifies the compiler due to the diversity of the possible systolic arrays for a single computation, and the difficulty for a compiler to statically determine the best choice. I have attempted to make a direct comparison with Spatial on the GEMM kernel. Unfortunately, the current Spatial provides a more robust flow for Xilinx FPGAs but fails in compilation when I target the Intel device. Nevertheless, I notice that the reported GEMM performance in the Spatial paper [KFP$^+$18] is much lower than what I have achieved. HeteroCL [LCH$^+$19] decouples algorithm specification from compute, data type and memory customizations. It also provides an abstraction level higher than T2S-Tensor where the compiler automatically determines the systolic array for GEMM using PolySA [CW18] framework.

3.7 Conclusions

I proposed a language and compilation framework that decouples algorithm from hardware specialization. In this framework, the programmers specify a tensor algorithm in a simple functional form followed by the spatial optimizations to map the algorithm on a spatial hardware. With this system, several important dense tensor kernels achieved high performance with high productivity. These kernels achieved a performance close to the manually implemented designs from industry experts with significantly low design effort and time.

CHAPTER 4
TENSAURUS: A VERSATILE ACCELERATOR FOR MIXED SPARSE-DENSE TENSOR COMPUTATIONS

This chapter explores hardware acceleration of both dense and mixed sparse-dense tensor computations. It uses the domain knowledge gained from Chapter 3 to design a hardware accelerator that can accelerate both dense and sparse tensor factorizations. This is done by co-designing the hardware and a sparse storage format, which allows accessing the sparse data in a vectorized and streaming manner and achieves high memory bandwidth utilization. For designing such an accelerator, I extract a common computation pattern that is found in matrix and tensor computation kernels presented in Chapter 2 and implement it in the hardware. By designing the hardware based on this common compute pattern, I can not only accelerate tensor factorizations but also mixed sparse-dense matrix operations.

4.1 Introduction

Many of the real world tensors and matrices are sparse, containing many zeros and thus can be accelerated by sparse tensor computations. Traditionally, both dense and sparse tensor factorizations and sparse matrix computations have been done on CPUs and GPUs, both of which have low energy-efficiency as they allocate excessive hardware resources to flexibly support various workloads [CHW+13, ESA+06, VSM11]. Hardware accelerators solve this energy efficiency problem at the expense of flexibility. With the recent surge in deep learning hardware and parameter pruning techniques, many hardware accelerators have been proposed for sparse matrix computations; however, these accelerators are designed for low sparsity and hence do not generalize to the broad domain of sparse matrix computations that involves matrices from different domains with a varying range of sparsity. Nevertheless, these sparse matrix accelerators cannot perform the common tensor kernels in tensor factorizations. Although algebra on multi-dimensional tensors can be reduced to matrix-vector/matrix-matrix operations, it has been shown that doing so increases the amount of intermediate storage and often leads to unnecessary computations [SRSK15]. Thus, the main focus of this chapter is to design a single accelerator that can accelerate all the dense and mixed sparse-dense tensor and matrix kernels mentioned in Chapter 2. There are two key challenges with designing such an accelerator. First, many of the real-world tensors such as Netflix movie ratings [BL$^+$07], never-ending language learning (NELL) [CBK$^+$10] and Amazon product reviews [ML13] are highly sparse, which makes tensor computations memory bound. Second, the compute and memory access patterns of different tensor and matrix kernels can be very different, which makes it necessary to reduce these computations into some basic operations in order to implement them in hardware.

In this work, I propose *Tensaurus*, a new hardware accelerator for highly efficient processing of mixed sparse and dense tensor computations. Tensaurus accelerates a computation pattern that I observe in common across different tensor factorization kernels as well as several widely used matrix operations. It further exploits a new sparse storage format that I introduce to allow accessing of sparse data in a vectorized and streaming manner to achieve high memory bandwidth utilization. Thus, with the co-design of the accelerator architecture and sparse storage, Tensaurus is both *versatile* and *adaptable*. It is versatile in the sense that it can accelerate both tensor factorizations and common matrix operations such as matrix-matrix and matrix-vector multiplications, which are key compute primitives in many ML applications. It can also easily *adapt* to different levels of sparsity found in these computations.

To the best of my knowledge, no prior work has proposed a hardware accelerator for sparse tensor factorizations and previous efforts have focused on dense tensor factorizations (e.g., T2S-Tensor [SRB$^+$19a] and [ZZZ19]).

The key technical contributions of this chapter are:

- This work is the first to propose a hardware accelerator that is capable of accelerating not only sparse tensor factorizations, but also dense tensor factorizations and other common mixed sparse-dense (sparse-dense and dense-dense) matrix operations for a wide range of sparsity.
- I introduce a new sparse storage format, *compressed interleaved sparse slice* (*CISS*), which allows accessing sparse data in a vectorized and streaming manner and thus achieves high memory bandwidth utilization and performance for sparse tensor kernels.
- Tensaurus achieves significant speedup and energy reduction for tensor factorizations over the state-of-the-art CPU and GPU implementations. Tensaurus also demonstrates higher performance and energy efficiency compared to CPU, GPU, and the state-of-the-art hardware accelerator for sparse CNNs.

- Tensaurus also performs consistently better than the state-of-the-art sparse neural network accelerator, Cambricon-X, on sparse-dense matrix multiplication for a wide range of sparsity.

4.2 Compute Pattern

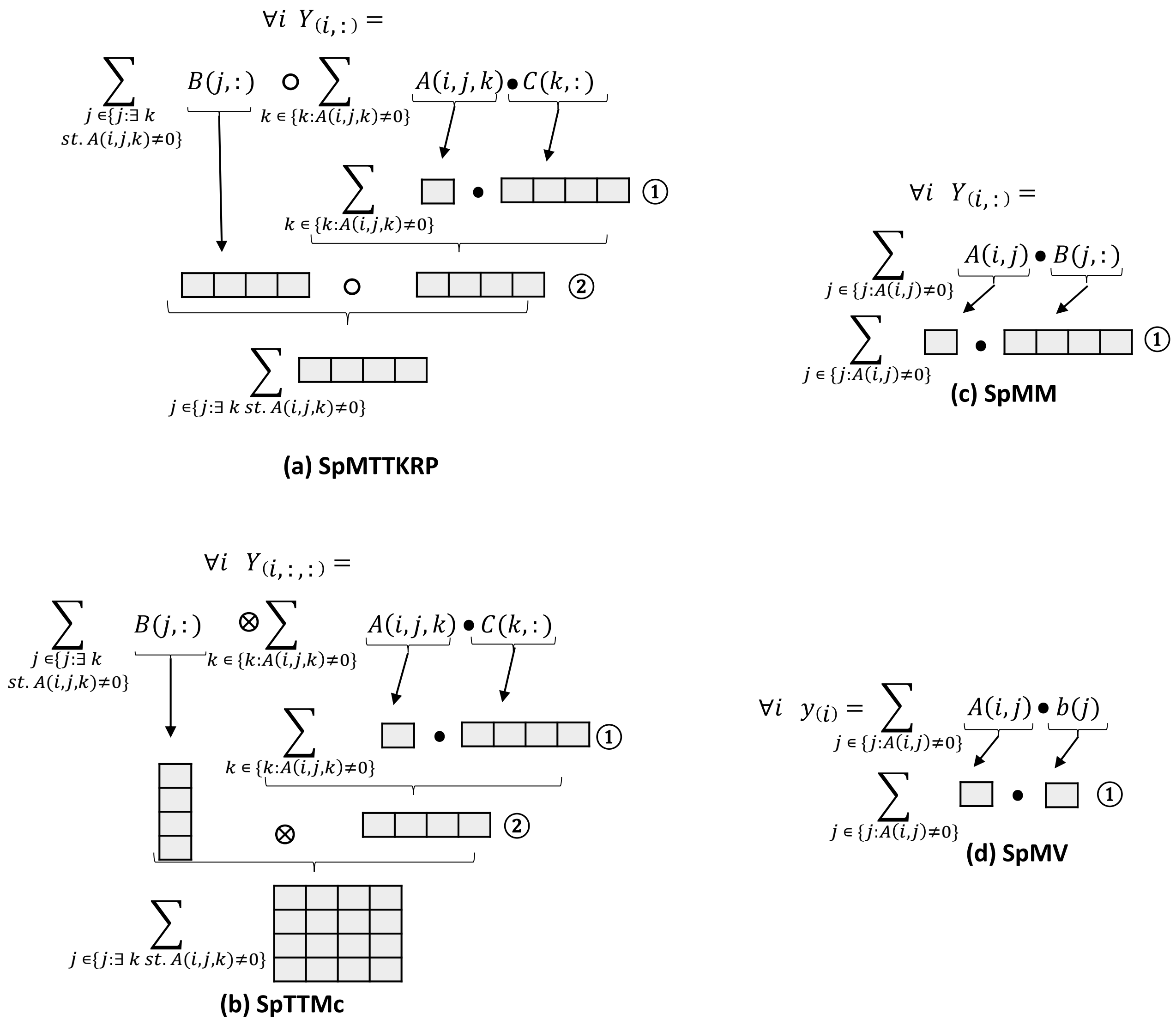


Figure 4.1: SpMTTKRP, SpTTMc, SpMM and SpMV Expressed Using the SF^3 Compute Pattern

I observe that a common compute pattern can be extracted across all the aforementioned kernels, namely, MTTKRP, TTMc, matrix-matrix multiplication, and matrix-vector multiplication. I name this compute pattern as *scalar-fiber product followed by fiber-fiber products (SF^3)* and it is expressed in the following form:

$$\mathbf{fibers}_{out} \;=\; \sum_{D_1} \mathbf{fiber}_1 \; op \; \sum_{D_0} \big(\mathbf{scalar} \;\cdot\; \mathbf{fiber}_0\big) \tag{4.1}$$

Table 4.1: Mapping of DMTTKRP, SpMTTKRP, DTTMc, SpTTMc, GEMM, SpMM, GEMV and SpMV Kernels to the SF^3 Compute Pattern

	$\mathbf{fibers}_{out}$	$\mathbf{fiber}_1$	$\mathbf{D}_1$	op	**scalar**	$\mathbf{fiber}_0$	$\mathbf{D}_0$
DMTTKRP	$\mathbf{Y}(i,:)$	$\mathbf{B}(j,:)$	$[0,J)$	$\circ$	$\boldsymbol{\mathcal{A}}(i,j,k)$	$\mathbf{C}(k,:)$	$[0,K)$
SpMTTKRP	$\mathbf{Y}(i,:)$	$\mathbf{B}(j,:)$	$\{j \mid \exists\, k\ st.\ \boldsymbol{\mathcal{A}}(i,j,k) \neq 0\}$	$\circ$	$\boldsymbol{\mathcal{A}}(i,j,k)$	$\mathbf{C}(k,:)$	$\{k \mid \boldsymbol{\mathcal{A}}(i,j,k) \neq 0\}$
DTTMc	$\boldsymbol{\mathcal{Y}}(i,:,:)$	$\mathbf{B}(j,:)$	$[0,J)$	$\otimes$	$\boldsymbol{\mathcal{A}}(i,j,k)$	$\mathbf{C}(k,:)$	$[0,K)$
SpTTMc	$\boldsymbol{\mathcal{Y}}(i,:,:)$	$\mathbf{B}(j,:)$	$\{j \mid \exists\, k\ st.\ \boldsymbol{\mathcal{A}}(i,j,k) \neq 0\}$	$\otimes$	$\boldsymbol{\mathcal{A}}(i,j,k)$	$\mathbf{C}(k,:)$	$\{k \mid \boldsymbol{\mathcal{A}}(i,j,k) \neq 0\}$
GEMM	$\mathbf{Y}(i,:)$	NA	NA	NA	$\mathbf{A}(i,j)$	$\mathbf{B}(j,:)$	$[0,J)$
SpMM	$\mathbf{Y}(i,:)$	NA	NA	NA	$\mathbf{A}(i,j)$	$\mathbf{B}(j,:)$	$\{j \mid \mathbf{A}(i,j) \neq 0\}$
GEMV	$\mathbf{y}(i)$	NA	NA	NA	$\mathbf{A}(i,j)$	$\mathbf{b}(j)$	$[0,J)$
SpMV	$\mathbf{y}(i)$	NA	NA	NA	$\mathbf{A}(i,j)$	$\mathbf{b}(j)$	$\{j \mid \mathbf{A}(i,j) \neq 0\}$

Here, $\mathbf{fibers}_{out}$ represent one or more output fibers of a tensor, $\mathbf{fibers}_0$ and $\mathbf{fibers}_1$ represent a single fiber from two tensors, **scalar** represents a scalar value from a tensor, op is either a Hadamard product ($\circ$) or a Kronecker product ($\otimes$), and D_1 and D_0 are domains over which the two summations are performed.

Table 4.1 shows how DMTTKRP, DTTMc, GEMM and GEMV map to SF^3 compute pattern. For DMTTKRP, $\mathbf{fibers}_{out}$ is a row from $\mathbf{Y}$ matrix, $\mathbf{fiber}_1$ is a row from $\mathbf{B}$ matrix, op is $\circ$, **scalar** is data value from $\boldsymbol{\mathcal{A}}$ tensor and $\mathbf{fiber}_0$ is a row from $\mathbf{C}$ matrix. For DTTMc, all the notations are the same as DMTTKRP, except $\mathbf{fibers}_{out}$ are more than one fiber (one slice $F_1 \times F_2$) from $\boldsymbol{\mathcal{Y}}$ tensor and op is $\otimes$. For GEMM, the **scalar** is data from $\mathbf{A}$ matrix, $\mathbf{fiber}_0$ is a row from $\mathbf{B}$ matrix, $\mathbf{fibers}_{out}$ is a row from $\mathbf{Y}$ matrix and op and $\mathbf{fiber}_1$ are not applicable (NA). Matrix-vector multiplication is same as matrix-matrix multiplication except $\mathbf{fibers}_{out}$ and $\mathbf{fibers}_0$ consist of only one element. Since all the computations in Table 4.1 are dense, the domains D_0 and D_1 for each computation are also continuous ranges.

Fig. 4.1 shows how SpMTTKRP, SpTTMc, SpMM and SpMV map to the SF^3 compute pattern in Eq. (4.1). Here the mapping of each sparse kernel to Eq. (4.1) is the same as that of their dense counterparts except the domains D_0 and D_1 are non-continuous ranges and determined by the position of non-zero entries in the sparse tensor. Although shown for 3-d tensor, the SF^3 compute pattern can be easily extended to tensors of any dimensions.

The formulation in Eq. (4.1) exhibits coarse-grained parallelism, where different output fibers can be computed in parallel and fine-grained single instruction multiple data (SIMD) parallelism where the computation of a single fiber can be performed in a vectorized manner.

4.3 Sparse Format

Sparse tensor computations require a sparse storage format that is efficient for both load balancing and parallel data accesses in order to accelerate them on spatial hardware. The existing sparse storage formats such as compressed sparse row (CSR) [BFF+09], compressed sparse fiber (CSF) [SRSK15], co-ordinate (COO) and their variants [ZDZ+16, PBP+18] allocate data needed by different processing elements (PEs) at far away locations in memory, resulting in low memory bandwidth utilization.

Figs. 4.2a and 4.2b show a sparse tensor and how it is stored in an extended CSR format. In this format, all the non-zero entries in the tensor are stored contiguously in the memory along with their mode 1 and mode 2 indices j and k. An array of slice pointers whose length is equal to the number of slices in the tensor (4 in this example) stores the pointers to the beginning of each slice in the array of non-zero entries. Fig. 4.2c depicts cycle-by-cycle execution of two PEs, each of which reads a different slice of the tensor from the memory in a streaming fashion. As it can be seen in each cycle the two PEs read the data from non-contiguous memory locations.

The *compressed interleaved sparse row* (CISR) format proposed by Fowers *et al.* [FOS+14] tackles this issue by storing the sparse data accessed by different PEs at the same time in contiguous memory locations; however, this format requires centralized row decoding, lock-step execution, and applies only to matrices. Partly inspired by CISR, I propose a new sparse storage format called *compressed interleaved sparse slice (CISS)*. With this new format, I overcome the limitations of CISR and extend it to tensors with more than two dimensions. Fig. 4.2d shows the tensor in Fig. 4.2a stored in the CISS format, which consists of an array of CISS entries. Each entry is $(dw + 2 \cdot iw) \cdot P$ bits long, where dw (data width) and iw (index width) are the bitwidths of the non-zero elements and their mode indices, respectively, and P is the number of PEs in hardware. For each PE, a CISS entry consists of three fields: nnz (non-zero data value), i/j (mode 0 or mode 1 index) and k (mode 2 index). Since nnz is supposed to carry only non-zero data values, a 0 in nnz

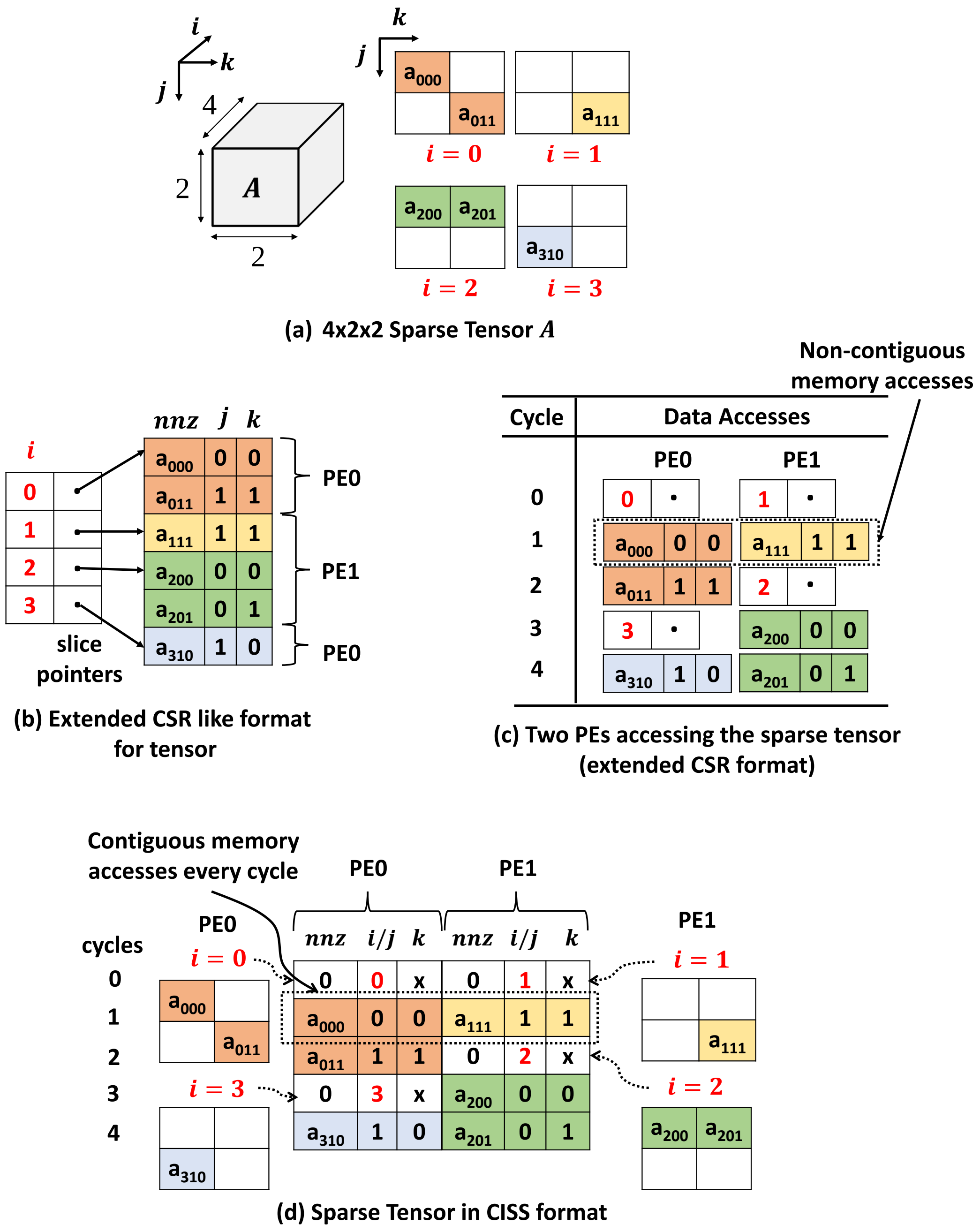

i
k
j
4
2
A
2
a000
a011
a111
a200
a201
a310
i = 0
i = 1
i = 2
i = 3
(a) 4x2x2 Sparse Tensor A
nnz j k
slice pointers
PE0
PE1
PE0
(b) Extended CSR like format for tensor
Non-contiguous memory accesses
Cycle
Data Accesses
PE0
PE1
(c) Two PEs accessing the sparse tensor (extended CSR format)
Contiguous memory accesses every cycle
cycles
nnz i/j k
x
(d) Sparse Tensor in CISS format

Figure 4.2: Storage Formats – (a) a $4 \times 2 \times 2$ sparse tensor $\mathcal{A}$; (b) the sparse tensor stored in extended CSR format. Here the slice pointers point to beginning of a slice in the array consisting of non-zero data elements and their mode 1 (m_1) and mode 0 (m_1) indices. The data in sparse tensor is split across two PEs, where PE0 accesses data from slice 0 and slice 3, and PE1 accesses data from slice 1 and slice 2; (c) data accessed by different PEs in different cycles while computing fibers of output matrix $\mathbf{Y}(i,:)$ in SpMTTKRP. Here the data accessed by different PEs in the same cycle is stored in non-contiguous memory locations; and (d) the sparse tensor stored in the CISS format. Here a single CISS entry contains the data from both the PEs and thus memory accesses for both the PEs in every cycle are done at contiguous memory locations. "**x**" denotes don't cares.

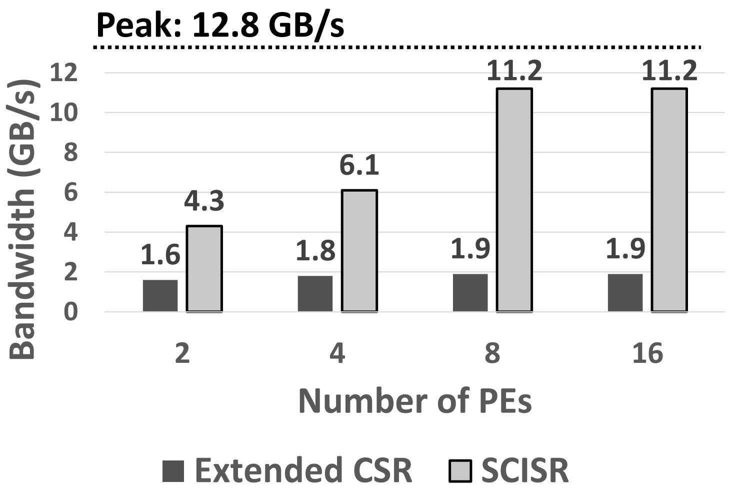


Figure 4.3: Memory Bandwidth Comparison – Achieved bandwidth comparison between CSR and CISS for different numbers of PEs. Here the utilized bandwidth for CSR saturates at 1.9 GB/s for 8 PEs, while CISS is able achieve 70% of the peak bandwidth.

indicates that the i/j field consists of i value and a non-zero in *nnz* indicates that i/j consists of a j value.

To store a sparse tensor in CISS format, first each PE is assigned a slice of the tensor corresponding to its ID. For example, in Fig. 4.2d PE0 is assigned slice 0 and PE1 is assigned slice 1 in the first cycle. The slice indices for each PE are written to the i/j and the *nnz* is set to 0. In the next few cycles, the CISS entries for a PE are filled with the non-zero entries from the slice by assigning the non-zero data elements to *nnz*, mode 1 indices to i/j and mode 2 indices to k. When all the non-zero elements in the slice assigned to a PE are scheduled, the next available slice is assigned to that PE and its slice index and data values are inserted into the array of CISS entries. For example, in Fig. 4.2d when all the non-zero entries from the slice $i = 1$ are inserted into the array, the next available slice is slice $i = 2$ and hence it gets assigned to PE1.

Since CISS assigns the non-zeros accessed from different PEs at the same time in contiguous memory locations, it achieves high *spatial locality* and *memory bandwidth utilization*. The CISS format also schedules the next available slice of the tensor to the PE with the least data that ensures a *load balanced* schedule where each PE is assigned a similar number of non-zero entries. Although described for 3-d tensors, the CISS format can be easily generalized to 2-d matrices and tensors with more than three dimensions.

Fig. 4.3 shows the achieved bandwidth when multiple PEs stream a 3-d tensor stored in extended CSR and CISS formats from the off-chip memory with a peak bandwidth of 16 GB/s. As it can be seen, the achieved bandwidth for extended CSR saturates at 1.9 GB/s for 8 PEs while CISS achieves a bandwidth of 11.2 GB/s, very close to the peak bandwidth.

4.4 Tensaurus Microarchitecture

In this section, using SpMTTKRP as a driving example, I explain the implementation of the SF^3 compute pattern described in Section 4.2.

Fig. 4.4 shows the execution of SpMTTKRP kernel using the same tensor as in Fig. 4.2a on two PEs. The slices $i = 0$ and $i = 3$ are assigned to PE0 and slices $i = 1$ and $i = 2$ are assigned to PE1 (same as in Fig. 4.2). Each PE reads a non-zero data element a_{ijk} from the sparse tensor $\boldsymbol{\mathcal{A}}$, the k^{th} row from matrix $\mathbf{C}$ ($\mathbf{c}_{k:}$) and performs a scalar-vector multiplication to produce $\mathbf{t}^{k}_{ij:}$. All the partial results $\mathbf{t}^{k}_{ij:}$ for different values of k are accumulated together and then multiplied by the j^{th} row of matrix $\mathbf{B}$ ($\mathbf{b}_{j:}$) to produce the partial results $\mathbf{y}^{j}_{i:}$. All the $\mathbf{y}^{j}_{i:}$ vectors are then summed together for different values of j to produce the i^{th} row of the output matrix $\mathbf{Y}$. From this example, it can be seen that the core operations in the SpMTTKRP are scalar-vector multiplication ($a_{ijk} \cdot \mathbf{c}_{k:}$), element-wise vector-vector multiplication ($\mathbf{t}^{k}_{ij:} \circ \mathbf{b}_{j:}$) and element-wise vector-vector addition ($\mathbf{t}^{k}_{ij:} + \mathbf{t}^{k'}_{ij:}$ and $\mathbf{y}^{j}_{i:} + \mathbf{y}^{j'}_{i:}$). There are also two types of intermediate results produced in the SpMTTKRP computation: $\mathbf{t}^{k}_{ij:}$ and $\mathbf{y}^{j}_{i:}$.

For the SF^3 compute pattern in Eq. (4.1), the scalar-vector multiplications arise from $\mathbf{scalar} \cdot \mathbf{fiber}_0$; the element-wise vector-vector multiplication (VVMUL) operations arises from op; and the element-wise vector-vector addition (VVADD) arises from the two summations over D_0 and D_1. The intermediate results in Eq. (4.1) correspond to the partial results of the computations $\mathbf{scalar} \cdot \mathbf{fiber}_0$ and $\mathbf{fiber}_1\ op \sum_{D_0} \mathbf{scalar} \cdot \mathbf{fiber}_0$. Thus, the micro-architecture of a PE in Tensaurus can be designed using these three major operations: scalar-vector multiplication, VVMUL and VVADD and two sets of storage registers for the two kinds of partial results. For scalability, I can further split a single vector operation into multiple small vector operations, each of vector length VLEN.

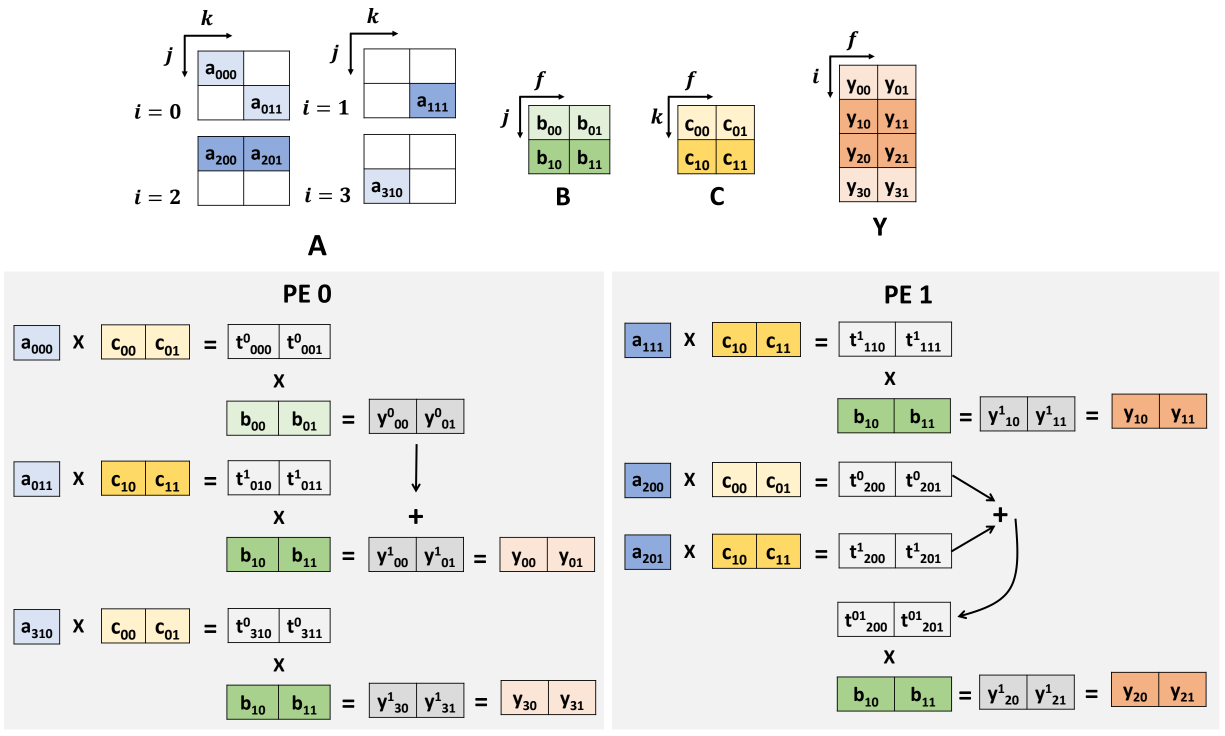


Figure 4.4: Cycle by Cycle Execution of SpMTTKRP on Two PEs

4.4.1 Implementation Details of Tensaurus

Fig. 4.5 shows the architecture of *Tensaurus*, which consists of a 2-D array ($r \times c$) of compute PEs, a tensor load unit (TLU), a matrix load unit (MLU), an array of scratchpad memories (SPM) and a matrix store unit (MSU). The TLU reads the first operand tensor, which is either stored in CISS format for sparse-dense kernels or dense format for dense-dense kernels, from the main memory. The MLU reads the dense operand matrices from the main memory and sends them to the SPMs. The SPMs receive the matrix data from the MLU and cache them in the double buffers. Each PE gets the data from the TLU and the SPM, performs VVMUL and VVADD operations in SIMD fashion (with vector length as VLEN) and accumulates the results in either a temporary shift register (TSR) or output shift register (OSR) depending on the type of accumulation. When the partial results for the current input tiles are completely evaluated, the PE drains the result to the MSU. Although each PE accumulates all the partial sums in local shift registers for the current input tiles, different input tiles may still update the same output element. Hence the MSU accumulates the drained results from the compute PEs and stores them in an output double buffer to perform reductions across different tiles. When all the input tiles for an output tile are processed, the MSU writes the results from the output buffer to main memory.

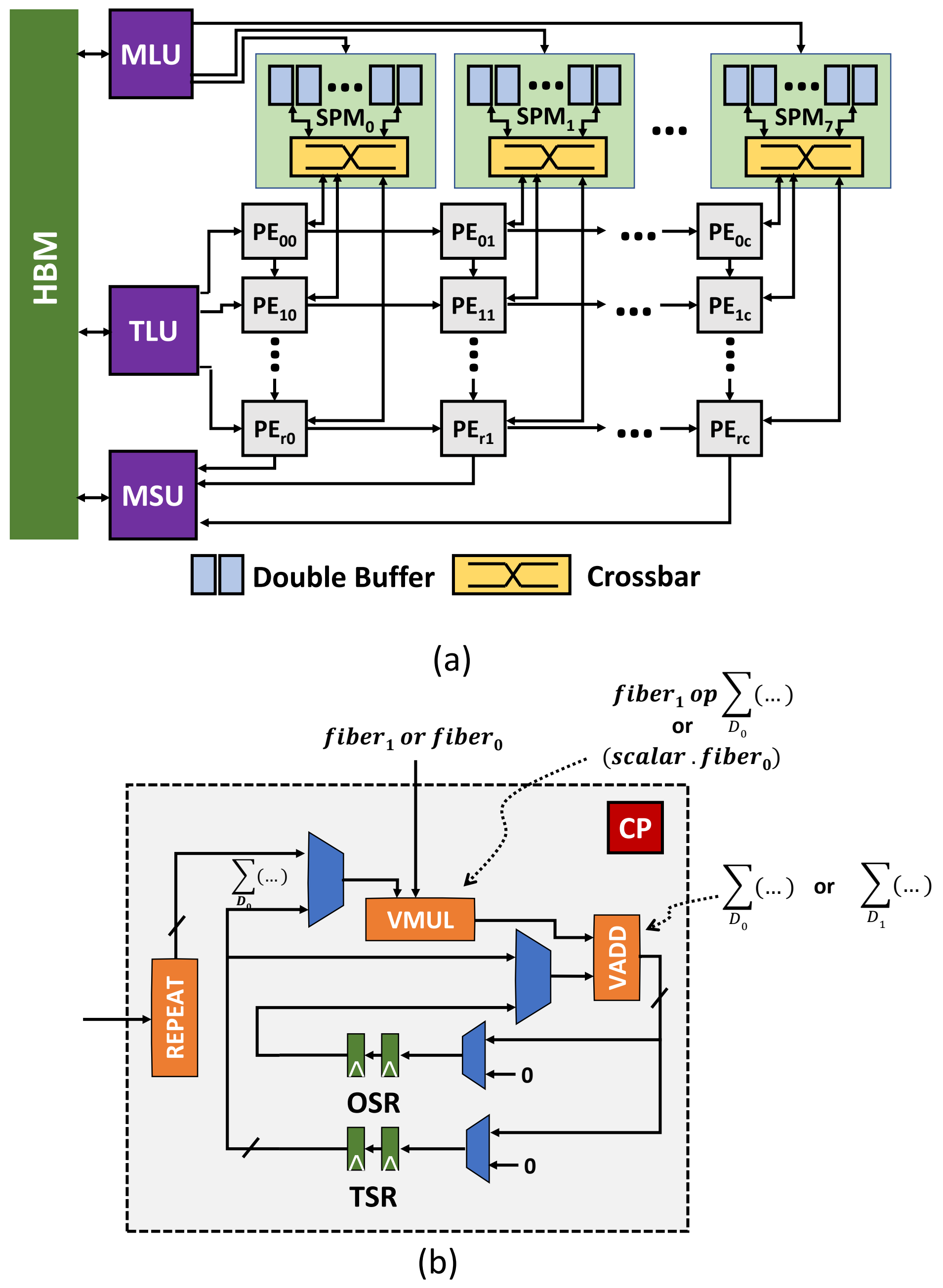


Figure 4.5: An Overview of Tensaurus Architecture – (a) A High Bandwidth Memory (HBM) is connected to the MLU, TLU and MSU via an arbiter. The MLU is connected to an array of SPMs using on-chip FIFOs. Each SPM consists of an array of *r* double buffers each of which is connected to the crossbar. The TLU is connected to PEs in the first column of 2-d systolic array to supply the sparse/dense data from the first operand matrix/tensor. Each PE is connected to the PE in the same row for systolic communications and to the crossbar in the same column to request the data from SPM. A systolic network from PEs going downwards connects PEs with the MSU to drain the results. (b) The architecture of a single PE which consists of a control processor (CP) implemented as an FSM (finite state machine), a repeat unit, VVMUL unit, VVADD unit, TSR and OSR shift-registers.

Tensor Load Unit

The TLU reads the data for the first kernel operand from the main memory, which is either a sparse tensor stored in CISS format or a dense tensor stored in dense format. The read data is then pushed to the hardware queues connecting the TLU and the boundary PEs. The queues between the TLU and the boundary PEs ensure that the TLU and PEs can work asynchronously. To enable non-blocking memory accesses, the TLU is capable of handling out-of-order memory responses. It tries to send a load request to the main memory every clock cycle and pushes the request ID to a hardware queue in-order. When the response for a memory request arrives (out-of-order), the response data is written to the hardware queue and the corresponding entry is marked as completed. In each cycle, the TLU polls the head of the hardware queue and pops the data if marked as completed and sends it to the boundary PEs. Since the CISS format ensures that the data accessed by the PEs at the same time is stored contiguously in the memory, the TLU sends wide read requests (one CISS entry) to the main memory to saturate the memory bandwidth.

Matrix Load Unit

The MLU reads the data for the dense operand matrices from the main memory. Similar to the TLU, it consists of hardware queues to perform out-of-order memory reads. The MLU reads data in chunks of $c \cdot VLEN \cdot dw$ bits from the main memory and sends the data to the SPMs.

Scratchpad Memories

The SPMs are responsible for two major tasks: they read the matrix data from MLU and store it in the local buffers, and they serve a read request from the PEs in the corresponding column of the PE array. To avoid serialization between read and write to the buffers, the local buffers are implemented as double buffers. Each SPM receives data in chunks of $VLEN \cdot dw$ bits from the MLU and stores it into one of the buffers. The read and write port width of each buffer is $VLEN \cdot dw$ bits. Inside an SPM, a tile of a matrix is banked such that consecutive rows from the matrix are assigned to different buffers. Since a PE can request the data from any row of the matrix, a crossbar is used to connect different buffers to the PEs as shown in Fig. 4.5.

For SpMTTKRP and DMTTKRP, each SPM stores a tile of both the dense operand matrices **B** and **C**. For SpTTMc and DTTMc, only the SPM in the first column stores a tile of both the first and second dense operand matrices while the rest of the SPMs store only the tiles of the second operand matrix **C**. Thus, the SPM in the first column has $2\times$ the amount of buffer capacity as compared to other SPMs. For SpMM and GEMM, each SPM stores tiles of the dense operand matrix **B**; and for SpMV and GEMV, only the SPM in the first column of the PE array is active, and stores a tile of dense input vector **b**.

Compute PEs

The compute PEs are designed for efficient computation of the SF^3 compute pattern in Eq. (4.1). Fig. 4.5b shows the design of a single compute PE. It consists of a control processor (CP), two shift-registers (temporary shift register (TSR) and output shift register (OSR)), a VVMUL unit, and a VVADD unit. The VVMUL and VVADD units process vectors of size VLEN. The TSR and OSR consist of TLEN and OLEN number of shift-registers, each of which is $VLEN \cdot dw$ bit wide. The TSR is used to store the result of $\sum_{D_0} \mathbf{scalar} \cdot \mathbf{fiber}_0$ in Eq. (4.1) and the OSR stores the partial sums for the output $\mathbf{fibers}_{out}$. Since for SpMM and SpMV, $\mathbf{fiber}_1$ and op are not applicable, the TSR is not used and the OSR stores the value of the computation $\sum_{D_0} \mathbf{scalar} \cdot \mathbf{fiber}_0$, which also is the $\mathbf{fibers}_{out}$. The PEs in the same row form a systolic array where the PEs on the left boundary read the $(scalar, j, k)$ triplets (in the case of SpMTTKRP, DMTTKRP, SpTTMc and DTTMc) and $(scalar, j)$ pairs (in the case of SpMM, GEMM, SpMV and GEMV), and forward them to the PEs in the same row. Here the $scalar$ is the non-zero element from the sparse tensor operand.

For SpMTTKRP, each PE requests the data from the k^{th} row of the **C** matrix from the SPM. The SPM receives the request and sends VLEN elements from that row to the PE. The PE then replicates the $scalar$ to perform a VVMUL operation corresponding to ① in Fig. 4.1a and accumulates the results in the TSR. When all the non-zero entries in $\boldsymbol{\mathcal{A}}(i, j, :)$ are processed, the PE requests the SPM for the data from the j^{th} row of matrix **B**, performs a VVMUL operation with the partial results in the TSR followed by a VVADD with the partial results in the OSR ②, and accumulates the result in the OSR. When all the non-zeros in the i^{th} slice of the tensor ($\boldsymbol{\mathcal{A}}(i, :, :)$) are processed, the results in the OSR are sent to the MSU, which writes them to the output buffer. Fig. 4.6 shows the execution of SpMTTKRP kernel for the sparse tensor in Fig. 4.2a on a 2×2 PE array. Here, the dense matrices are tiled along the columns, banked along rows and stored in different local buffers (VLEN is assumed to be one).

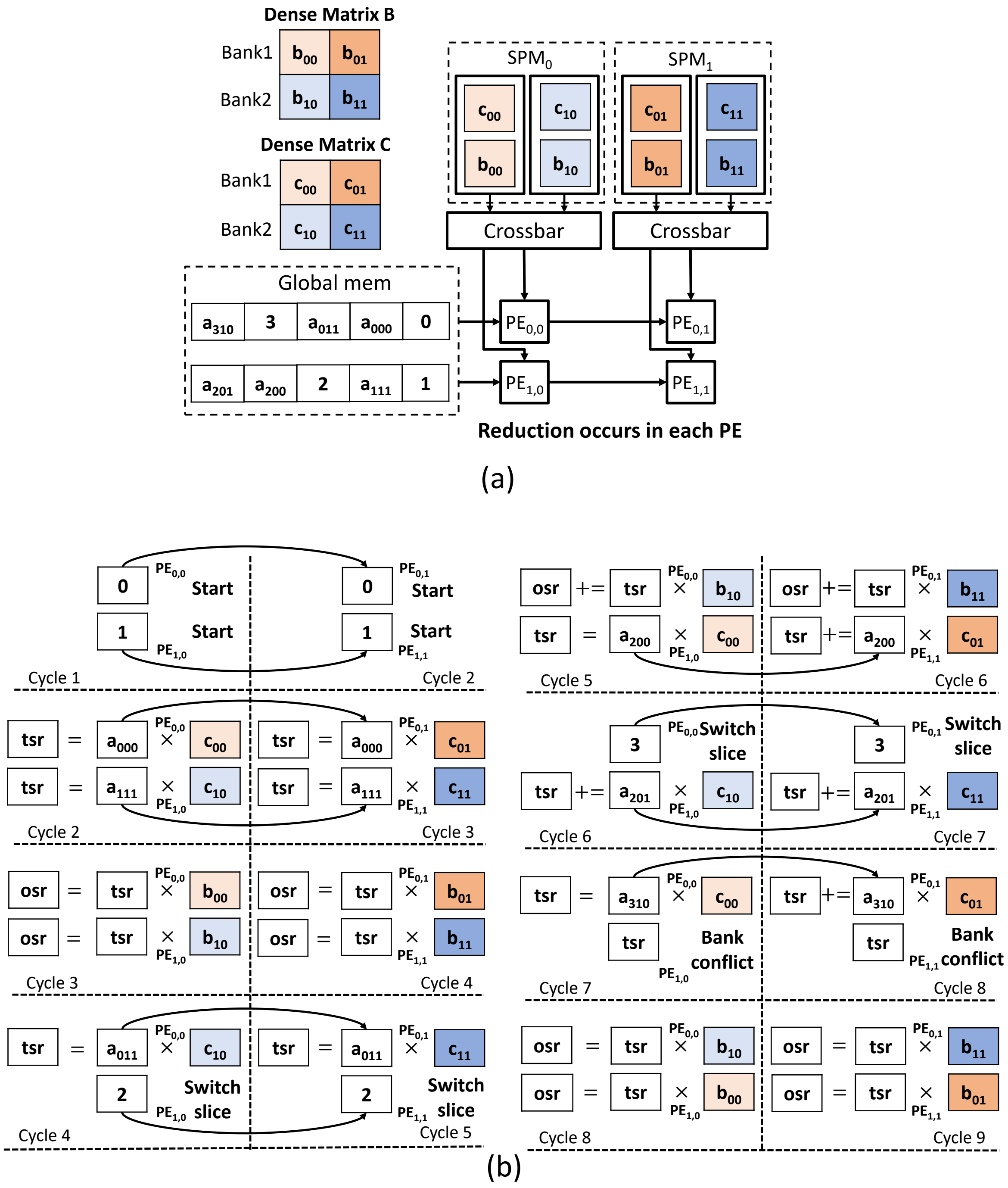


Figure 4.6: SpMTTKRP on Tensaurus – (a) 2×2 PE array in Tensaurus where dense matrix B is tiled, banked and stored in the buffers of different SPMs. The data from sparse tensor $\mathcal{A}$ is stored in the memory in CISS format (only data values shown); (b) cycle by cycle execution of SpMTTKRP kernel on the 2×2 PE array. Here VLEN is assumed to be one.

For SpTTMc, each PE acts in the exact same way as in SpMTTKRP; however, when all the non-zero entries in $\mathscr{A}(i, j, :)$ are processed and it has requested the data from the j^{th} row of matrix

B, instead of directly performing a VVMUL with the TSR as in case of SpMTTKRP, it streams the values from the j^{th} row of matrix **B** one by one. It then replicates these values to perform a VVMUL with the TSR and accumulates the results in one of the shift-registers in the OSR. The number of shift registers in the OSR (*OLEN*) is thus set to be VLEN. This approach in effect computes the outer-product of the row from matrix **B** and the value in the TSR ②.

For SpMM, each PE sends the column index j to its SPM, which reads VLEN elements from the j^{th} row of matrix B and sends them to the PE. The PE then replicates the *scalar* value from sparse matrix, performs a VVMUL operation corresponding to ① in Fig. 4.1c and accumulates the result in the OSR. When all the non-zero entries in the current row of the sparse matrix $\mathbf{A}(i,:)$ are processed, the PE drains the OSR data to the MSU.

For SpMV, since the second operand is a dense vector instead of a dense matrix, only the first column of PEs in the systolic array is active. Each PE sends the column index j of the non-zero entry $\mathbf{A}(i,j)$ to the SPM similar to SpMM. However, this time the SPM reads only one element from the j^{th} index of the dense vector $\mathbf{b}$ and sends it to the PE. The PE then performs a scalar multiplication between the *scalar* value from sparse matrix **A** and the dense vector element, corresponding to ① in Fig. 4.1d, and accumulates the result in a single register of the OSR. When all the non-zero entries in the current row of the sparse matrix are processed, the PE drains the OSR data to the MSU.

For dense operations (DMTTKRP, DTTMc, GEMM and GEMV), the TLU reads the data in dense format, constructs a CISS representation on the fly and sends it to the PEs. The PEs and other units remain unaware that they are performing a dense computation. Since in the case of dense operations each PE from the same column would request the same entry from the SPM, these requests can get serialized by the crossbar. To avoid such serialization, for dense operations, only the PE in the first row is responsible for sending row addresses to the SPM and the crossbar broadcasts the response to all the PEs in the same column.

Matrix Store Unit

The MSU is responsible for receiving the partial results from the PEs and accumulating them in the output buffer. The MSU drains the output buffer to the main memory when all the input tiles corresponding to an output tile have been processed. Although buffering the intermediate results in the output buffer reduces the number of off-chip memory accesses, it also limits the tile size of the sparse input tensor. Since for very sparse tensors the benefit of storing the intermediate results in the buffer is outweighed by the larger tile size of the tensor (as it results in more reuse of the dense operand matrices), the MSU can be configured to directly accumulate the results in the main memory.

4.5 Tensaurus Experimental Setup

4.5.1 Simulation Infrastructure

To evaluate the performance of Tensaurus, I model the architecture of Tensaurus consisting of TLU, MLU, SPMs, PEs, MSU and HBM using the gem5 simulator [BBB$^+$11]. The gem5 model uses an 8×8 PE array with $VLEN = 4$. Each SPM except the first consists of a double buffer of size $2 \times 16KB$ where each side of the double buffer is divided into 8 $2KB$ banks. The SPM in the first column has a double buffer of size $2 \times 32KB$ divided into 16 $2KB$ banks. The MSU consists of an output double buffer of size $2 \times 128KB$, which is further divided into 8 $16KB$ banks. For HBM, I use the gem5 model, which supports up to 8 128-bit physical channels, runs at 1GHz clock frequency and provides a peak memory bandwidth of 128 GB/s. Tensaurus is attached to a CPU as a co-processor, where the CPU executes instructions to configure Tensaurus to run a specific tensor kernel. The configuration instructions configure Tensaurus for: (1) mode of operation like SpMTTKRP, SpMM, etc. and (2) size of tensors and matrices.

4.5.2 Measurements

The PEs and the crossbar were implemented in RTL using PyMTL [LZB14], synthesized using the Synopsys Design Compiler and TSMC 28nm library and placed-and-routed using Cadence Innovus. For the SPM and MSU, since the majority of area and power is dominated by scratchpads, I used CACTI 7.0 [MBJ09] to model SRAM latencies, area and power. For the TLU, I pessimistically assumed the same area and power as a single PE. Table 4.2 shows the area and power breakdown of different components of the design. For HBM I use the energy numbers from [Shi16].

Table 4.2: Area and Power Breakdown of Tensaurus

Component	Area(mm^2)	%	Power (mW)	%
PE	0.625	27.2 %	402.30	40.9 %
Xbar	0.066	2.8 %	24.27	2.5%
SPM	0.832	36.2 %	296.05	30.1 %
MSU	0.759	33.0 %	247.03	25.2 %
TLU	0.009	0.4 %	6.28	0.6%
MLU	0.009	0.4 %	6.28	0.6 %
Total	2.3	100 %	982.21	100 %

4.5.3 Baselines

I compare Tensaurus against four baselines: CPU, GPU, Cambricon-X [ZDZ+16] and T2S-Tensor [SRB+19a].

CPU: I use SPLATT [SRSK15] and Sparse BLAS [DHP02] to evaluate our benchmarks on a single core of an Intel(R) Xeon(R) CPU E7-8867 running at 2.40 GHz with 32 KB L1 cache, 256 KB L2 cache and 45 MB of L3 cache. For energy estimates I use McPAT 1.3 [LAS+09] CPU energy models.

GPU: I use ParTI [LMV18, MLW+19] and cuSPARSE [NCVK10] to evaluate the benchmarks on a modern GPU Titan Xp, which has GDDR5x DRAM with a peak bandwidth of 547.6 GB/s and a peak 32-bit performance of 12.15 TFLOP/s. I use CUDA 9.1 for programming the GPU. For power estimation, I use thermal design power (TDP) from the GPU datasheet. I do not use GPUWattch [LHE+13] integrated with GPGPU-Sim [BYF+09] since it does not provide support for the latest GPUs and CUDA versions above 4.0.

Accelerator: For SpMM, I also compare Tensaurus against the Cambricon-X [ZDZ+16] state-of-the-art CNN accelerator, which uses sparse weights and dense activations. I implement the architecture of Cambricon-X in gem5 and scale it to have the same bitwidth, clock frequency, number of multiply-accumulate (MAC) units, size of on-chip RAM and DRAM bandwidth as our accelerator. For energy comparisons, I use the power numbers from [ZDZ+16], which are in 65nm technology node and scale them to 28nm using [tecc, tece]. The dynamic power can be estimated as $\alpha f C V_{dd}^2$, where α is the switching activity, f is the clock frequency, C is the total capacitance and V_{dd} is the supply voltage. Since α remains the same for the two technology nodes (65nm and 28nm) and the capacitance scales with the area of the design, I use the square of the ratio of Contacted Gate Poly Pitch CPP (which is technology dependent) from [tece] and [tecc] as a scaling factor for C. I further use the ratio of V_{dd} from [tece] and [tecc] and frequency numbers from [ZDZ$^+$16] to scale V_{dd}^2 and f. For DRAM energy, I measure the number of DRAM accesses from the simulator and use the HBM energy from [Shi16].

For DMTTKRP, DTTMc and GEMM I compare Tensaurus against T2S-Tensor [SRB$^+$19a], which implements these kernels on an FPGA. I scale the design in T2S-Tensor to use the same number of MAC units and clock frequency as Tensaurus.

Table 4.3: Tensors with Their Dimensions, Number of Non-Zeros (nnz), Density and Problem Domain

Tensor	Dimensions	nnz	Density	Domain
nell-2	12K × 9K × 28K	77M	2.5e-5	NLP
netflix	480K × 18K × 2K	100M	5.7e-6	Rec. Sys.
poisson3D	3K × 3K × 3K	99M	3.6e-3	Synthetic

Table 4.4: Weight Matrices from AlexNet and VGG-16 with Their Dimensions, Number of Non-Zeros (nnz) and Density

	Layer	Dim.	Density	Layer	Dim.	Density
AlexNet	c1	96 × 363	0.84	c2	256 × 1200	0.38
	c3	384 × 2304	0.35	c4	384 × 1728	0.37
	c5	256 × 1728	0.37	fc6	9216 × 4096	0.09
	fc7	4096 × 4096	0.09	fc8	4096 × 1000	0.25
VGG-16	c1_1	64 × 27	0.58	c1_2	64 × 576	0.22
	c2_1	128 × 1152	0.34	c2_2	128 × 1152	0.36
	c3_1	256 × 1152	0.53	c3_2	256 × 2304	0.24
	c3_3	256 × 2304	0.42	c4_1	512 × 2304	0.32
	c4_2	512 × 4608	0.27	c4_3	512 × 4608	0.34
	c5_1	512 × 4608	0.35	c5_2	512 × 4608	0.29
	c5_3	512 × 4608	0.36	fc6	25088 × 2096	0.01
	fc7	4096 × 4096	0.02	fc8	4096 × 1000	0.09

4.5.4 Datasets

For SpMTTKRP and SpTTMc, I use the tensor datasets shown in Table 4.3. The `nell-2` tensor is a snapshot of the never ending language learner knowledge base that attempts to create a computer system that learns how to read the web [CBK$^+$10]. The `netflix` dataset is taken from

Table 4.5: Matrices from SuiteSparse [DH11] with Their Dimensions, Number of Non-Zeros (nnz), Density and Problem Domain

Matrix	Dim	nnz	Density	Domain
amazon0312	401K $\times$ 401K	3.2M	1.9e-5	Copurchase network
m133-b3	200K $\times$ 200K	801K	2.0e-5	Combinatorics
scircuit	171K $\times$ 171K	959K	3.2e-5	Circuit simulation
p2pGnutella31	63K $\times$ 63K	148K	3.7e-5	p2p network
offshore	260K $\times$ 260K	4.2M	6.2e-5	EM Problem
cage12	130K $\times$ 130K	2.0M	1.1e-4	Weighted graph
2cubes-sphere	101K $\times$ 101K	1.6M	1.5e-4	EM Problem
filter3D	106K $\times$ 106K	2.7M	2.4e-4	Reduction problem
emailEnron	36.7K $\times$ 36.7K	368K	2.7e-4	Email network
citeseer	3.3K $\times$ 3.3K	4.7K	4.2e-4	Graph Learning
cora	2.7K $\times$ 2.7K	5.3K	7.2e-4	Graph Learning
wikiVote	8.3K $\times$ 8.3K	104K	1.5e-3	Wikipedia network
poisson3Da	14K $\times$ 14K	353K	1.8e-3	Fluid Dynamics

the Netflix prize competition [BL+07] and poisson3D is taken from [SK17]. The `nell-2` and `netflix` are public datasets and taken from Smith *et al.* [SCL+17].

For SpMM, I use the pruned models for AlexNet and VGG-16 [HPTD15]. I did not use any of the newer CNN models for this study as their pruned weights are not publicly available. Table 4.4 shows the sparse weight matrices in these CNN models with their size and densities. For SpMM, I also use the sparse matrices from SuiteSparse [DH11] and graph benchmarks from GraphSAGE [HYL17]. Table 4.5 shows the sparse matrices from SuiteSparse and GraphSAGE. For SpMV, I use the same matrices from SuiteSparse and GraphSAGE as in SpMM.

4.6 Tensaurus Evaluation

4.6.1 Roofline Evaluation

Figs. 4.7, 4.8 and 4.9 show the throughput of SpMTTKRP, DMTTKRP, SpTTMc, DTTMc, SpMM and GEMM, under the roofline [WWP09] of Tensaurus. The x-axis shows operation intensity, which is the number of operations (multiply and add) performed for each byte of data accessed from the off-chip memory. The y-axis shows the performance in GOP/s. The horizontal line towards the right of the plot shows the peak attainable performance from the design when the operation intensity is high (kernel is compute bound) and the inclined line (with slope 1) towards the left shows the peak attainable performance when the operation intensity is low (kernel is memory bound). The point where these two lines intersect is called the *knee point*. The value of throughput for operation intensity of 1 represents the peak memory bandwidth (in GB/s). The gap between the roofline and the achieved performance of a kernel indicates the inefficiencies within the hardware. Our design consists of an 8×8 PE array, each with 4 SIMD MAC units and hence it has $8 \times 8 \times 4 \times 2 = 512$ scalar multipliers and adders. Since Tensaurus is simulated at 2GHz clock frequency, the scratchpads are assumed to be synchronous where each PE spends every other clock cycle to access the scratchpads instead of doing a MAC. Thus, the peak attainable throughput of Tensaurus is $512 \times 2 \times 0.5 = 512$ GOP/s. For peak memory bandwidth, I use the peak bandwidth of HBM which is 128 GB/s.

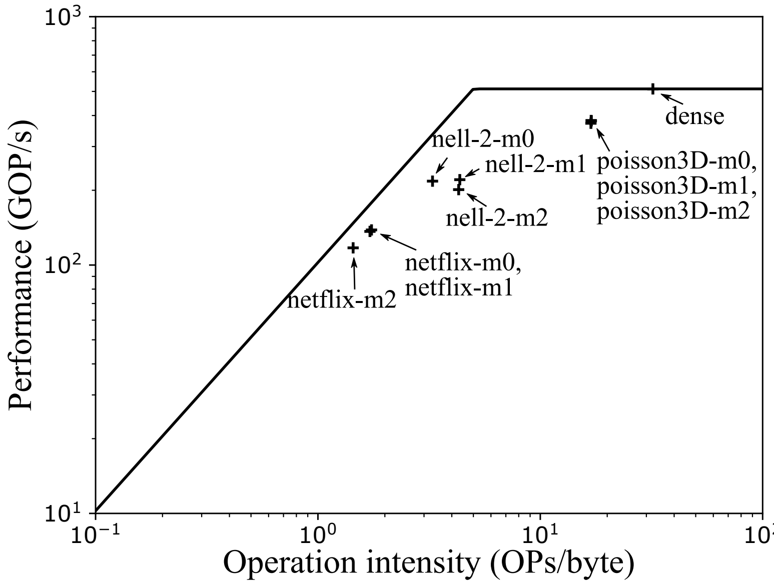


Figure 4.7: Roofline for SpMTTKRP

Fig. 4.7 shows the achieved throughput for SpMTTKRP along all the three modes of the three tensors shown in Table 4.3. Here, SpMTTKRP on all the tensors except `poisson3D` is memory bound while `poisson3D` is compute bound. This is because among all the tensors `poisson3D` has the highest density. For all the SpMTTKRP kernels Tensaurus is able to perform close to the peak throughput.

Fig. 4.8 shows the achieved throughput for SpTTMc for the three tensors along each mode. Here, `nell-2-m0`, `nell-2-m1` and `poisson3D` are compute bound while the others are memory bound and the achieved performance on each kernel is very close to the peak throughput. It can also

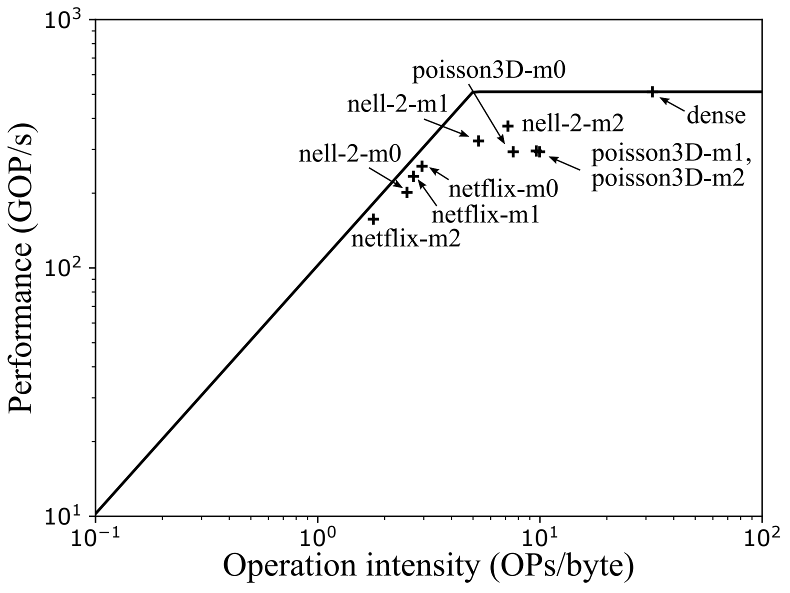


Figure 4.8: Roofline for SpTTMc

be seen from Fig. 4.7 and 4.8 that the operation intensity for the same tensor along the same mode is higher for SpTTMc as compared to SpMTTKRP. The reason behind this is SpTTMc performs outer-product as an intermediate operation as shown in Fig. 4.1d which has more MAC operations compared to the Hadamard product as in Fig. 4.1c.

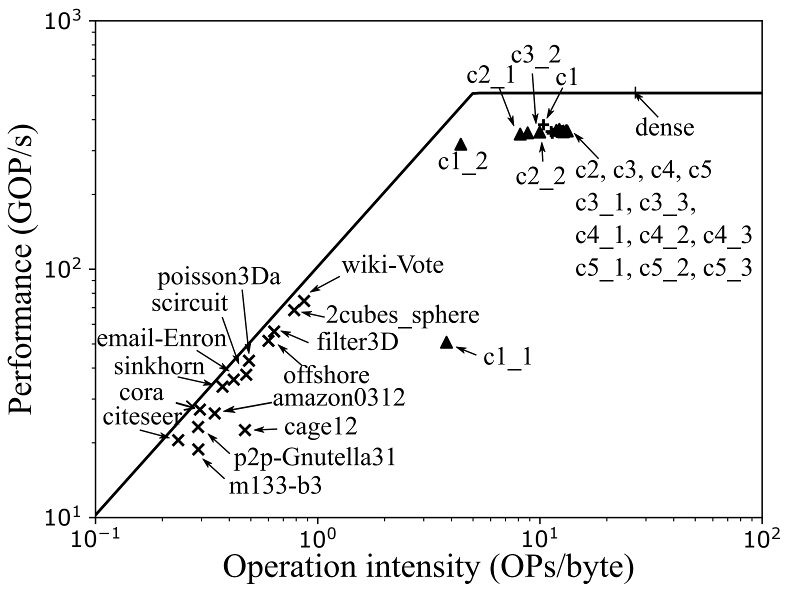


Figure 4.9: Roofline for SpMM

Fig. 4.9 shows the achieved throughput for sparse conv layers in AlexNet and VGG-16 and sparse matrices from SuiteSparse [DH11] and GraphSAGE [HYL17] for the SpMM kernel. For all the layers except c1_1 and c1_2 the achieved throughput is very close to the peak throughput. For c1_1 and c1_2, since the sparse weight matrices are very small (Table 4.4), the scratchpads and MAC units in Tensaurus are underutilized. For the SuiteSparse and GraphSAGE matrices since the densities of these matrices are very low (Table 4.5), the SpMM kernel is memory bound and Tensaurus achieves very close to the peak throughput in the memory bound region.

Figs. 4.7, 4.8 and 4.9, also show the achieved throughput for dense MTTKRP, TTMc and GEMM on our accelerator (labeled as “dense"). It can be seen that all the dense kernels are compute bound and our accelerator achieves close to the peak throughput for each of them.

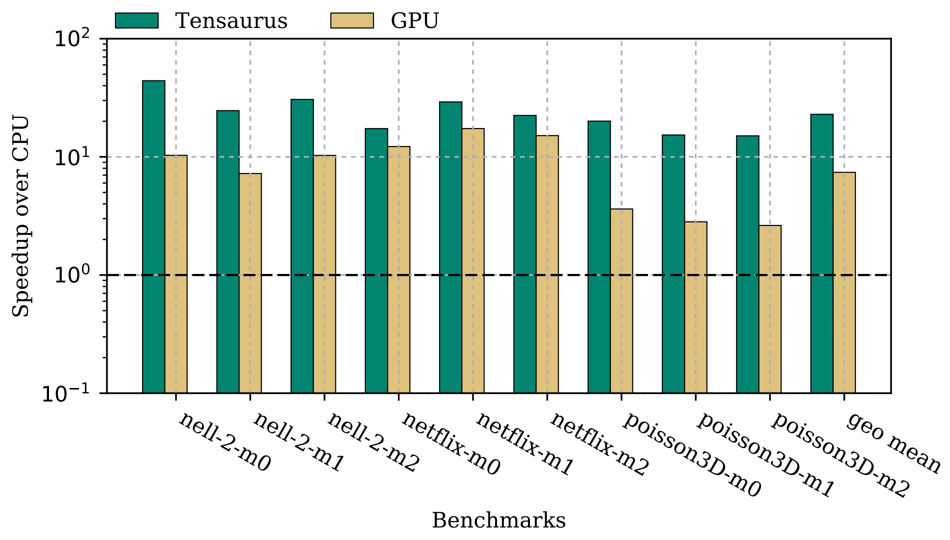
4.6.2 Performance Evaluation

Fig. 4.10(a) shows the speedup of Tensaurus and GPU (ParTI) on SpMTTKRP for the three tensors along each mode over the CPU (SPLATT) baseline. Tensaurus achieves a geomean speedup of 22.9$\times$ over CPU and 3.1$\times$ over GPU.

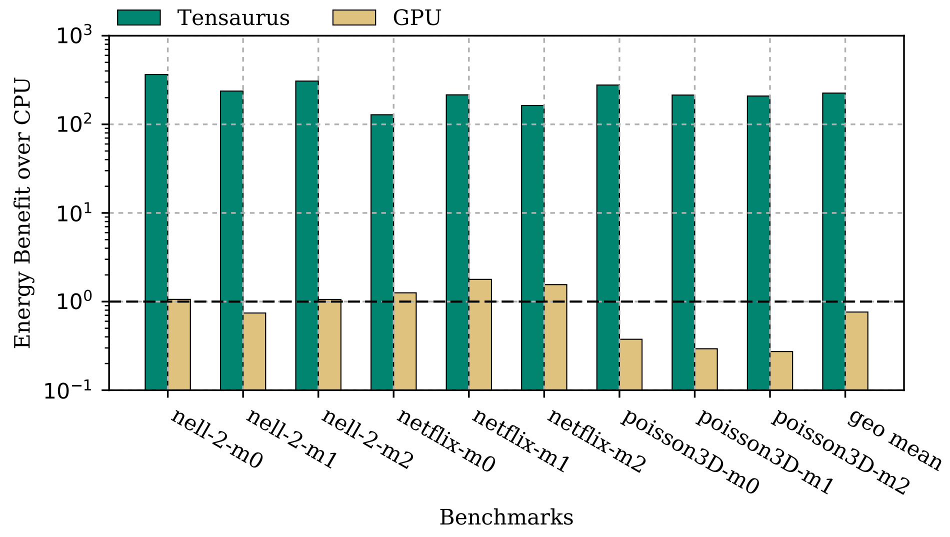
Fig. 4.11(a) shows the speedup of Tensaurus and GPU (ParTI) for SpTTMc on three tensors along each mode over the CPU (SPLATT) baseline. Here, Tensaurus achieves 6$\times$ speedup over CPU. However, Tensaurus achieves 0.1$\times$ of the performance of the GPU baseline. This slowdown compared to GPU (PaRTI) is because in PaRTI a significant portion of the SpTTMc algorithm runs on the host CPU, but for comparison with Tensaurus I do not take into account the CPU execution time. After taking CPU execution time into account, Tensaurus would achieve a 5$\times$ speedup over GPU. Unlike SpMTTKRP, where the speedup of Tensaurus is more than 20$\times$ over CPU, I also achieve a lower speedup of 6$\times$ in the case of SpTTMc. The reason behind this is SpTTMc benefits significantly from the operand factorization. A smaller tile size and on-chip memory limits operand factoring opportunities, and Tensaurus uses just 512KB of on-chip memory as compared to 45MB of L3 cache in the case of the CPU.

Figs. 4.12(a) and 4.13(a) show the speedup of Tensaurus, GPU (cuSPARSE) and Cambricon-X over the CPU (SparseBLAS) baseline for AlexNet and VGG-16. For most of the conv layers Tensaurus performs better than CPU, GPU and Cambricon-X. On average, Tensaurus is 349.2$\times$, 1.8$\times$ and 1.9$\times$ faster than CPU, GPU and Cambricon-X, respectively.

Fig. 4.14(a) shows the speedup of Tensaurus, GPU and Cambricon-X over CPU baseline for benchmarks from SuiteSparse and GraphSAGE. Unlike CNNs where the matrices have low sparsity (high density), these matrices have really high sparsity (low density). Tensaurus performs



(a) **Speedup for SpMTTKRP**

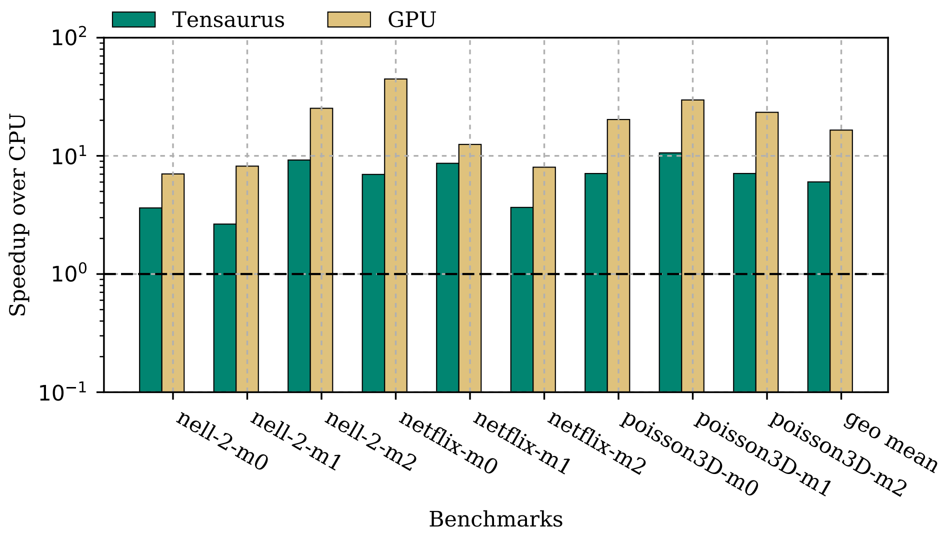


(b) **Energy Benefit for SpMTTKRP**

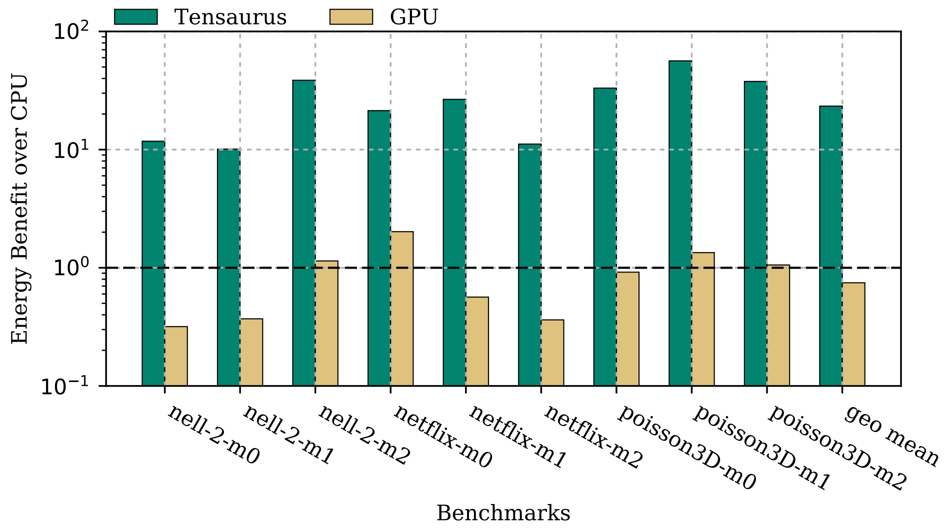
Figure 4.10: Speedup and Energy Benefit of Tensaurus, GPU (PaRTI) and Tensaurus-Dense Over CPU (SPLATT) Baseline for SpMTTKRP on Sparse Tensors in Table 4.3

better than Cambricon-X on all the matrices and often beats GPU. Overall, Tensaurus achieves 125.8× and 119.7× speedup over CPU and Cambricon-X, respectively and achieves 0.87× of the performance of GPU for these matrices.

To further analyse the performance of Tensaurus for different densities of sparse matrix, I generate synthetic matrices and measure SpMM performance for Tensaurus and all the baselines. Fig. 4.15 shows the speedup of Tensaurus, GPU, and Cambricon-X over the CPU baseline for different densities for sparse matrices. As it can be seen, Tensaurus performs consistently better



(a) **Speedup for SpTTMc**

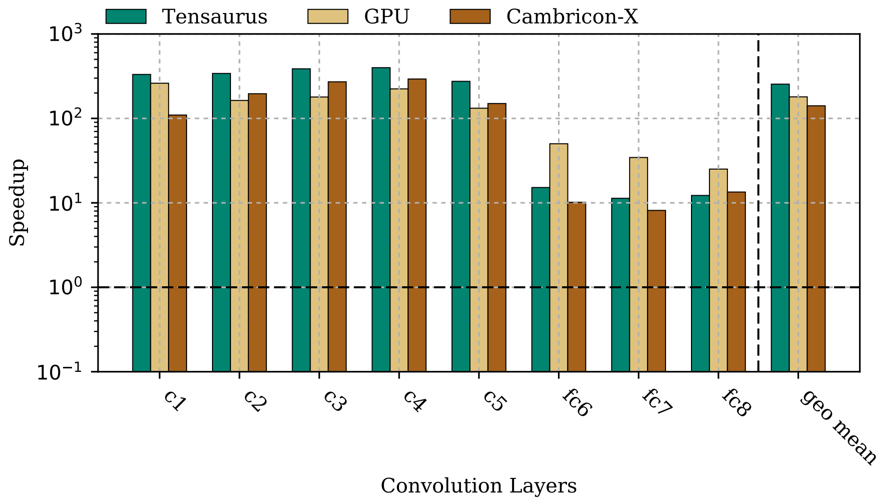


(b) **Energy Benefit for SpTTMc**

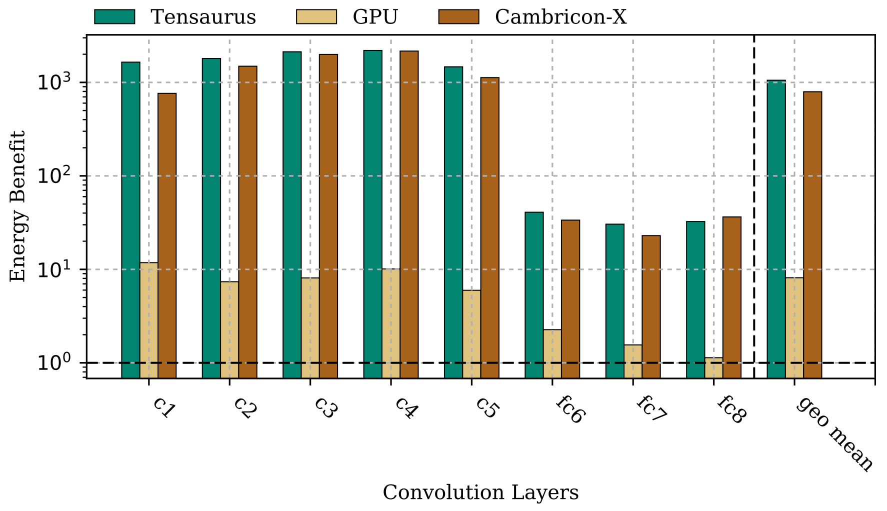
Figure 4.11: Speedup and Energy Benefit of Tensaurus, GPU (PaRTI) and Tensaurus-Dense Over CPU (SPLATT) Baseline for SpTTMc on Sparse Tensors in Table 4.3.

than all the baselines and the performance of GPU is very similar to the performance of Tensaurus. I further observe that the performance benefit of storing the partial results of a tile in output buffer of the MSU is outweighed by having large tiles of sparse matrix and directly storing them in memory for densities below 0.0003, and hence I turn off the output buffer in the MSU for all the experiments where densities are less than 0.0003.

Fig. 4.16(a) shows the speedup of Tensaurus, GPU (cuSPARSE) and Cambricon-X over CPU (SparseBLAS) baseline for benchmarks from SuiteSparse and GraphSAGE for the SpMV kernel. Overall Tensaurus achieves 7.7× and 0.45× speedup over CPU and GPU. Since SpMV is highly



(a) **Speedup Over CPU**



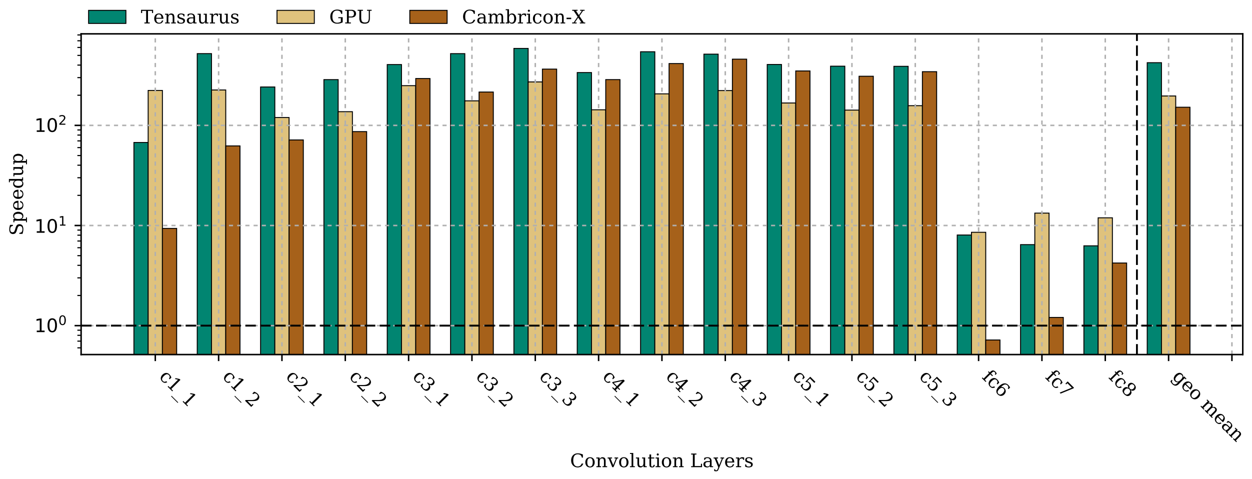
(b) **Energy Benefit Over CPU**.

Figure 4.12: Speedup and Energy Benefit of Tensaurus, GPU (cuSPARSE) and Cambricon-X over CPU (Sparse BLAS) Baseline for SpMM (Conv Layers) and SpMV (Fully Connected Layers) on Sparse Matrices from AlexNet

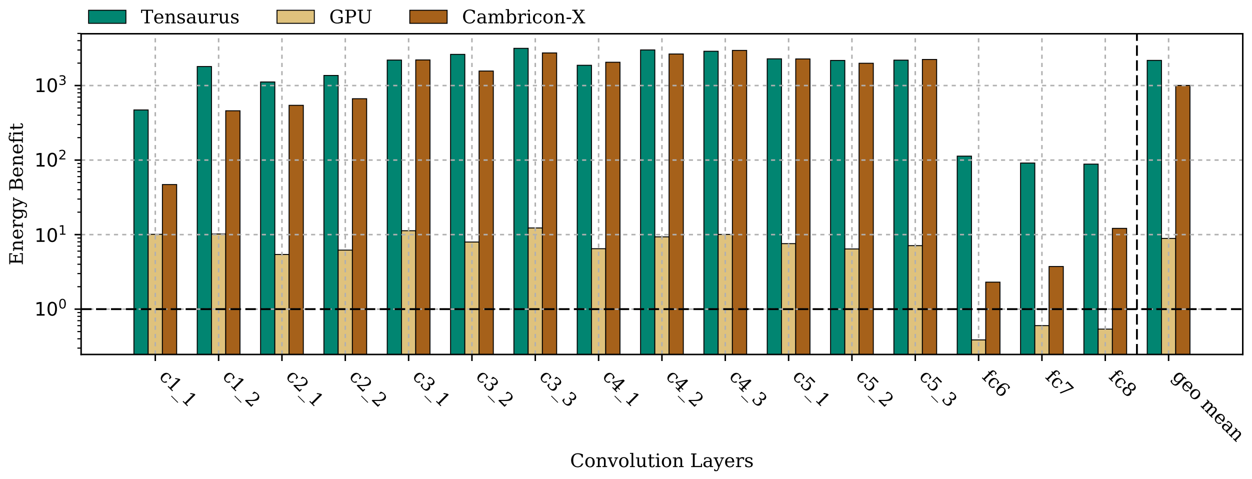
memory bound and GPU has 5× more bandwidth and more on-chip memory the performance of SpMV is better on GPU as compared to Tensaurus.

Table 4.6: Comparison between the performance of Tensaurus-dense and T2S-Tensor [SRB$^+$19a]

Benchmark	**Throughput (GOP/s)**		**Speedup**
	Tensaurus-dense	**T2S-Tensor**	
DMTTKRP	511.9	986.3	0.52×
DTTMc	498.9	926.6	0.54×
GEMM	506.5	1019.8	0.49×



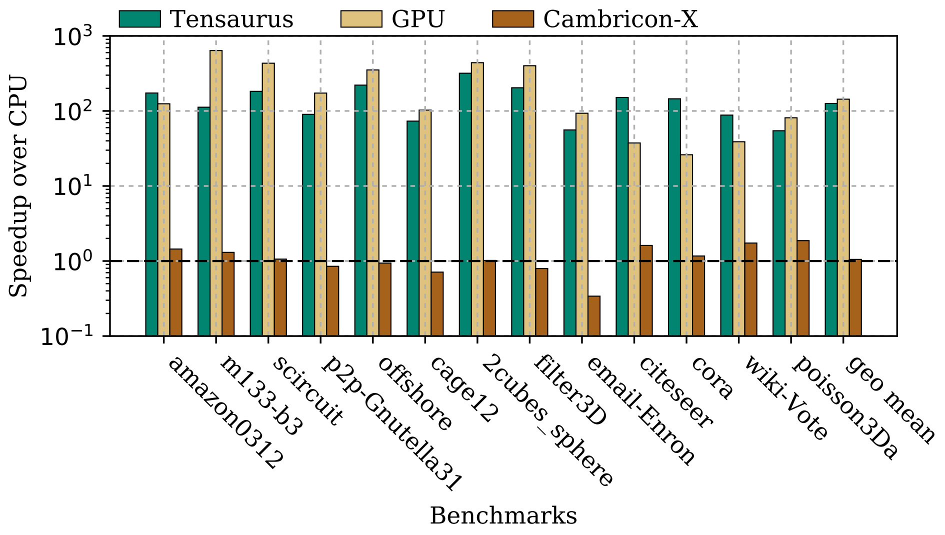
(a) **Speedup over CPU**



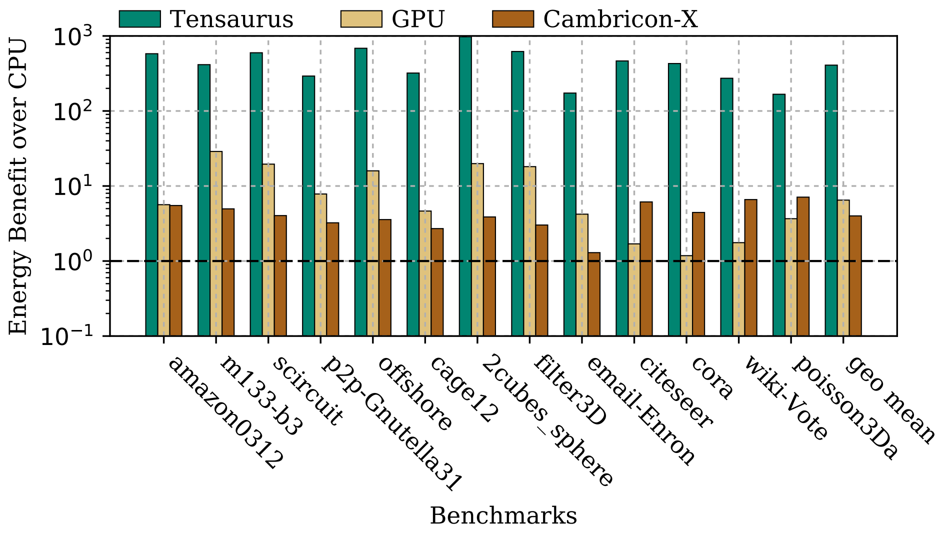
(b) **Energy Benefit over CPU**.

Figure 4.13: Speedup and Energy Benefit of Tensaurus, GPU (cuSPARSE) and Cambricon-X over CPU (Sparse BLAS) Baseline for SpMM (Conv Layers) and SpMV (Fully Connected Layers) on Sparse Matrices from VGG-16

I further compare the performance of DMTTKRP, DTTMc and GEMM with T2S-Tensor. Table 4.6 shows the throughput of our accelerator in dense mode (Tensaurus-dense) compared to T2S-Tensor. As it can be seen for DMTTKRP, DTTMc and GEMM, Tensaurus-dense achieves close to 0.5× of the performance of T2S-Tensor, which is a pessimistic estimate since I assume perfect scaling for T2S-Tensor.



(a) **Speedup for SpMM**



(b) **Energy Benefit for SpMM**.

Figure 4.14: Speedup and Energy Benefit of Tensaurus, GPU (cuSPARSE) and Cambricon-X over CPU (Sparse BLAS) Baseline for SpMM on Sparse Matrices from SuiteSparse and GraphSAGE

4.6.3 Energy

Fig. 4.10(b) and 4.11(b) show the energy benefit of our accelerator and GPU over the CPU baseline for SpMTTKRP and SpTTMc on three tensors along each mode. Overall, our accelerator is 223.2× and 292.8× more energy efficient than CPU and GPU for SpMTTKRP and 23.2× and 30.9× more energy efficient than CPU and GPU for SpTTMc.

Fig. 4.12(b) and 4.13(b) shows the energy benefit of our accelerator, GPU and Cambricon-X over the CPU baseline for AlexNet and VGG-16. On average, our accelerator is 1983.7×,

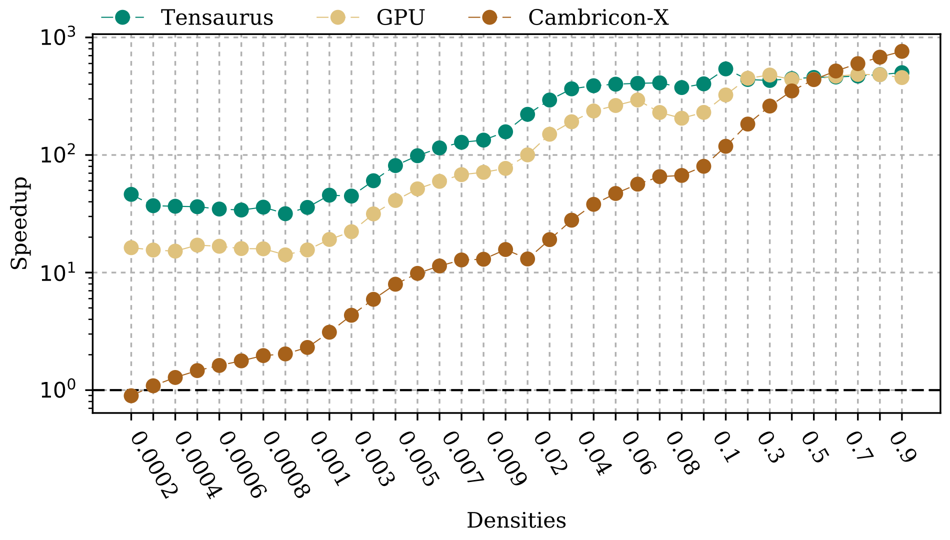


Figure 4.15: Speedup of Tensaurus, GPU (cuSPARSE) and Cambricon-X over CPU (Sparse BLAS) Baseline for SpMM on Synthetic Matrices with Density Varying from 0.0001 to 0.9

226.6× and 1.7× more energy efficient than CPU, GPU and Cambricon-X. Fig. 4.14(b) shows the energy benefit of our accelerator for SpMM on SuiteSparse and GraphSAGE matrices. Overall our accelerator is 405.6×, 62.5× and 101.5× more energy efficient than CPU, GPU and Cambricon-X.

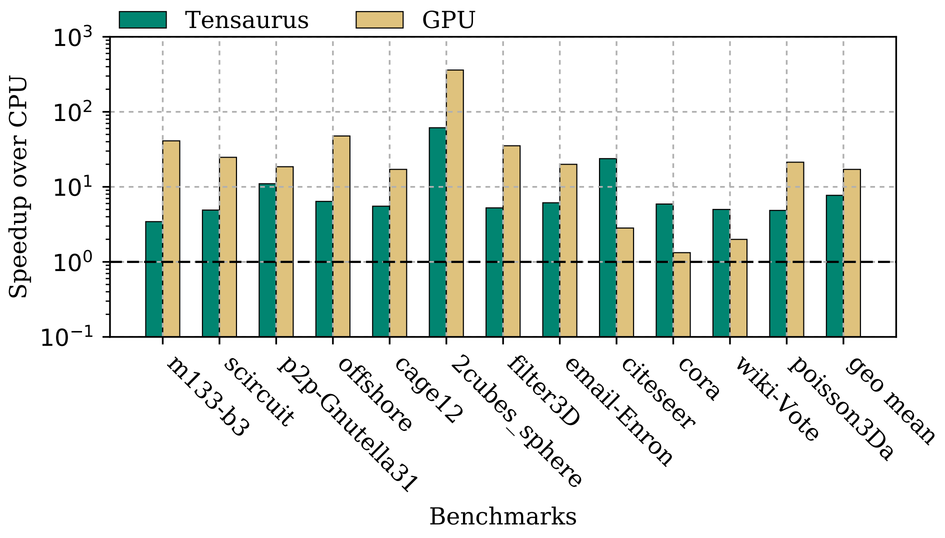
For dense GEMM, I compare the performance per Watt with Google TPU [JYP+17]. TPU is implemented in the same technology node 28nm, achieves 92 TOPS/s for 8-bit precision (23 TOPS/s for 32-bit precision) and has a TDP of 75W, thus achieving a performance/watt of 0.31 TOPS/s/Watt. In contrast, Tensaurus has a power consumption of 1.26 W and achieves a throughput of 0.5 TOPS/s with a performance/watt ratio of 0.39 TOPS/s/Watt which is very similar to TPU.

Fig. 4.16(b) shows the energy benefit of our accelerator and GPU over the CPU baseline for SpMV on matrices from SuiteSparse. For SpMV, our accelerator is 46.4× and 60.1× more energy efficient than CPU and GPU.

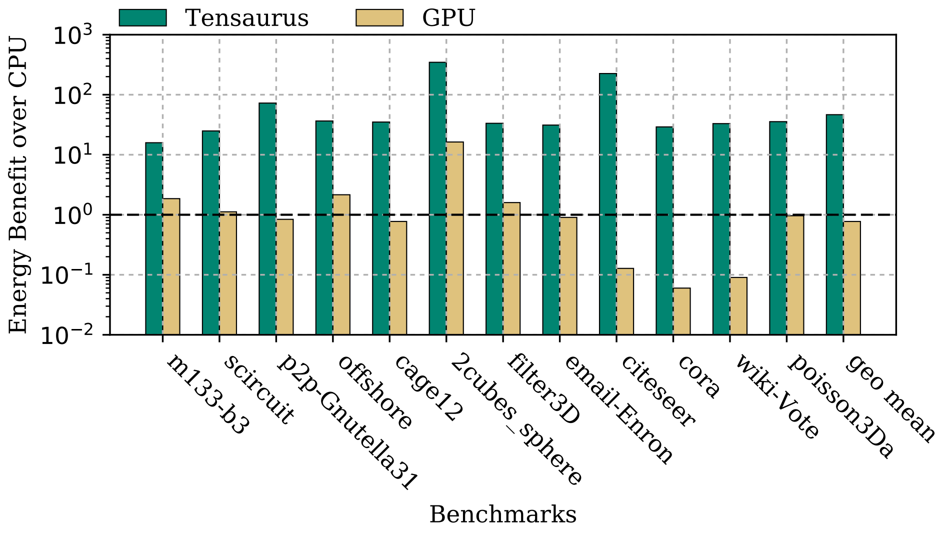
4.7 Related Work

4.7.1 Sparse Storage Formats

Many sparse storage formats have been propose in the literature. CSR (Compressed Sparse Row), CSC (Compressed Sparse Column) and COO (Co-ordinate) are the most commonly used



(a) **Speedup for SpMV**



(b) **Energy Benefit for SpMV**

Figure 4.16: Speedup and Energy Benefit of Tensaurus, GPU (cuSPARSE) and Cambricon-X over CPU (Sparse BLAS) Baseline for SpMV on Sparse Matrices from SuiteSparse and GraphSAGE

sparse storage formats for CPUs. Liu *et al.* [LWSD17] propose a sparse tensor storage format F-COO, which is similar to the co-ordinate format and used it for GPUs. CSF [SRSK15] and Hi-COO [LSV18] are other sparse tensor storage formats that are based on CSR and COO, respectively. OuterSPACE [PBP^{+}18] uses a variant of CSR and CSC formats called CR and CC for sparse-sparse matrix-matrix multiplication (SpGEMM). For machine learning hardware, researchers have proposed multiple variants of CSR and CSC formats. For example, Cambricon-X [ZDZ^{+}16] proposes a modification of CSR format where the non-zeros are are compressed and stored in contiguous memory and index vectors are used to decode the row and column indices.

EIE [HLM$^+$16] uses a variant of CSC storage format where instead of storing the row indices they store the number of zeros before a non-zero element. However, since these works focus on deep learning, especially CNNs, their sparse storage format is specialized for low sparsity (high density).

4.7.2 Software

TACO [KKC$^+$17] is language and compiler framework to generate high-performance code for sparse matrix and tensor kernels for CPUs. Kjolstad *et al.* [KAKA18] introduced workspace optimizations in TACO to implement operand factoring optimizations in tensor kernels. SPLATT [SRSK15, SK17] introduced a C library implementing SpMTTKRP and SpTTMc with shared memory parallelization. Bhaskaran *et al.* [BHP$^+$17] introduce various techniques to reduce memory usage and execution time for sparse tensor factorization algorithms. Ballard *et al.* [BHR18] and Choi *et al.* [CLC18] propose methods to perform DMTTKRP and DTTMc on CPU and GPU, respectively. Blanco *et al.* [BLD18] described a scalable cloud based sparse tensor factorization algorithm for distributed tensor factorizations.

4.7.3 Hardware

Srivastava *et al.* [SRB$^+$19a] propose a language and compilation framework called T2S-Tensor to generate high performance hardware for dense tensor computations such as GEMM, DMTTKRP and DTTMc. Zhang *et al.* [ZZZ19] design a hardware accelerator for DTTMc. Hegde *et al.* [HAMP$^+$19] designed a hardware accelerator called ExTensor for sparse tensor algebra using the ideas of merge lattice proposed in TACO [KKC$^+$17]. This work, however, does not accelerate sparse tensor factorizations as they involve more than two operands and their performance are significantly effected by operand factoring optimizations. ExTensor also uses significantly more on-chip storage than Tensaurus (30 MB as compared to 0.5 MB). This allows ExTensor to read an entire tile of the sparse input tensor on-chip in a streaming manner, avoiding the need for a special sparse storage format. However, this comes at the cost of area and energy where ExTensor is 40$\times$ larger than Tensaurus for the same technology node. Kanellopoulos *et al.* propose a hardware-software cooperative mechanism to accelerate sparse matrix operations. For SpMM and GEMM, prior works involving ASIC implementations include Cambricon-

X [ZDZ$^+$16], Cambricon-S [ZDG$^+$18], Cnvlutin [AJH$^+$16], SCNN [PRM$^+$17] [AKM$^+$18], and OuterSPACE [PBP$^+$18]. Cambricon-X [ZDZ$^+$16] and Cambricon-S [ZDG$^+$18] implement hardware accelerators for SpMM and SpGEMM in CNNs where either the weight matrices or both weight matrices and neurons are sparse. SCNN [PRM$^+$17] designs a SpGEMM accelerator for CNNs which can exploit the sparsity in both weights and neurons. OuterSPACE [PBP$^+$18] and [AKM$^+$18] design accelerator designs for SpGEMM. EIE [HLM$^+$16] proposes the SpMSpV (sparse matrix sparse vector multiplication) accelerator for fully connected layers in CNN and show significant performance gains over CPU and GPU. TPU [JYP$^+$17] implemented a 2-d systolic array for GEMM. Prior work involving FPGA implementations for sparse-dense and sparse-sparse matrix-matrix and matrix-vector accelerators include [LXH$^+$19], ESE [HKM$^+$17], [ZP05] and [FOS$^+$14]. Lu *et al.* [LXH$^+$19] design a CNN accelerator with sparse weights. ESE [HKM$^+$17] proposes an FPGA-accelerator for SpMV in LSTMs. Prasanna *et al.* [ZP05] and Fowers *et al.* [FOS$^+$14] propose SpMV accelerators for very sparse matrices.

4.8 Conclusions

In this work, I propose a new sparse storage format which allows accessing sparse data in a vectorized manner and co-design a hardware accelerator for sparse and dense tensor factorizations. I also extract a common computation pattern among different tensor factorization algorithms, which form the basis for many matrix operations and design a suitable hardware implementation. With such hardware software co-design I achieve significant speedup and energy benefits over multiple hardware and software baselines.

CHAPTER 5
MATRAPTOR: A SPARSE-SPARSE MATRIX MULTIPLICATION ACCELERATOR BASED ON ROW-WISE PRODUCT

This chapter explores the hardware acceleration of sparse-sparse matrix-matrix multiplication (SpGEMM), which is by far the most important sparse-sparse tensor kernel. In this chapter, I present MatRaptor, a novel SpGEMM accelerator that is high performance and highly resource efficient. Unlike conventional methods using inner or outer product as the meta operation for matrix multiplication, the approach presented in this chapter is based on the row-wise product, which offers a better trade-off in terms of data reuse and on-chip memory requirements, and achieves higher performance for large sparse matrices. I further propose a new hardware-friendly sparse storage format, which allows parallel compute engines to access the sparse data in a vectorized and streaming fashion, leading to high utilization of memory bandwidth.

5.1 Introduction

Sparse-sparse matrix-matrix multiplication (SpGEMM) is a key computational primitive in many important application domains such as graph analytics, machine learning, and scientific computation. More concretely, SpGEMM is a building block for many graph algorithms such as graph contraction [GRS08], recursive formulations of all-pairs shortest-paths algorithms [DN07], peer pressure clustering [Sha07], cycle detection [YZ04], Markov clustering [VD00], triangle counting [ABG15], and matching algorithms [RV89]. It has also been widely used in scientific computing such as multigrid interpolation/restriction [BM$^+$00], Schur complement methods in hybrid linear solvers [YL10], colored intersection searching [KSV06], finite element simulations based on domain decomposition [HHM12], molecular dynamics [IOM95], and interior point methods [KGK94].

SpGEMM often operates on very sparse matrices. One such example is the Amazon co-purchase network [LAH07], which is a graph where each product is represented as a node and the likelihood of two products being bought together is represented as an edge. This network consists of 400K nodes and 3.2M edges forming an adjacency matrix of 400K $\times$ 400K with a density of 0.002%. With such a high sparsity, the SpGEMM computation becomes highly memory-bound and requires effective utilization of memory bandwidth to achieve high performance. Traditionally, SpGEMM computations have been performed on CPUs and GPUs [NCVK10, DHP02, WZS+14], both of which have low energy efficiency as they allocate excessive hardware resources to flexibly support various workloads. Hardware accelerators tackle this energy efficiency problem through specialization. However, there are three key challenges in designing an accelerator for SpGEMM computation: (1) The inner product and outer product algorithms, which perform well for dense matrix multiplication, are not necessarily the ideal algorithms for SpGEMM due to the low densities of the sparse matrices involved in the computation; (2) The SpGEMM computation is highly memory-bound and the traditional sparse storage formats such as CSR, CSC and COO perform random data accesses when used with multiple compute units in parallel; this results in low memory bandwidth utilization; and (3) The output of SpGEMM is also a sparse matrix for which the number of non-zeros are not known ahead of the time, and hence contiguous data allocation for different rows/columns that are being computed in parallel requires synchronization.

In this chapter, I analyze the different dataflows for SpGEMM and compare them in terms of data reuse and on-chip memory requirements. I argue that a row-wise product approach has the potential to outperform other approaches for SpGEMM for very sparse matrices. I further propose a new sparse storage format named *cyclic channel sparse row (C^2SR)*, which enables efficient parallel accesses to the main memory with multiple channels. Using the row-wise product approach and the new sparse format, I describe the design of MatRaptor, a highly efficient accelerator architecture for SpGEMM. Based on experiments performed on a diverse set of sparse matrices, MatRaptor can achieve significant performance improvement over alternative solutions based on CPUs, GPUs, and the state-of-the-art SpGEMM accelerator OuterSPACE [PBP+18].

The key technical contributions of this chapter are as follows:

- I systematically analyze different dataflows for SpGEMM by comparing and contrasting them against data reuse and on-chip memory requirements. I show that a *row-wise product* approach, which has not been explored in the design of SpGEMM accelerators, has the potential to outperform the existing approaches.
- I introduce C^2SR, a new hardware-friendly sparse storage format that allows different parallel processing engines (PEs) to access the data in a vectorized and streaming manner leading to high utilization of the available memory bandwidth.

- I design a novel SpGEMM accelerator named MatRaptor, which efficiently implements the row-wise product approach and fully exploits the C^{2}SR format to achieve high performance. The experiments using gem5 show that MatRaptor is 1.7$\times$ faster than OuterSPACE on average with 12.3$\times$ higher energy efficiency. MatRaptor is also 31.3$\times$ smaller in terms of area and consumes 7.2$\times$ less power compared to OuterSPACE.

5.2 Analysis of SpGEMM Dataflows

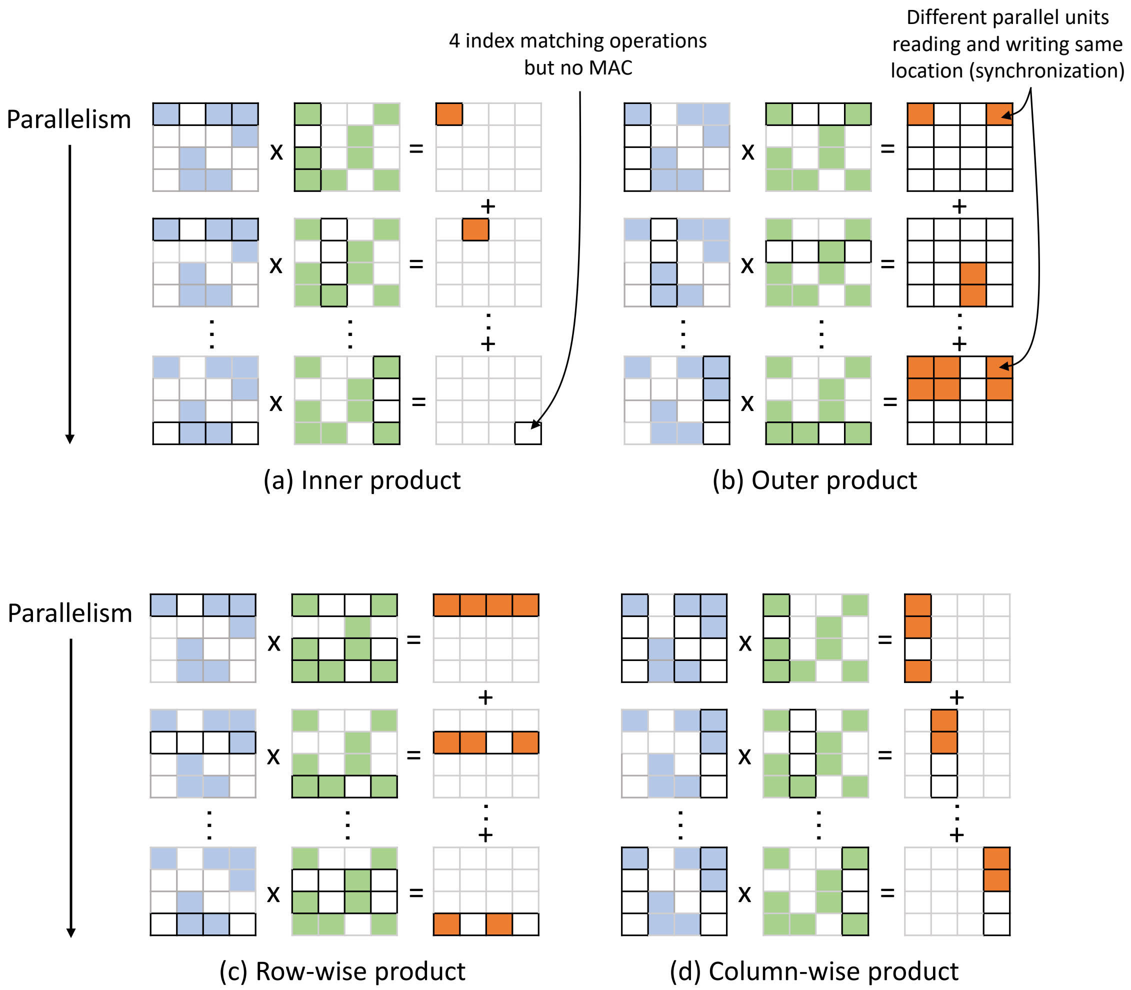

Parallelism
4 index matching operations but no MAC
Different parallel units reading and writing same location (synchronization)
(a) Inner product
(b) Outer product
Parallelism
(c) Row-wise product
(d) Column-wise product

Figure 5.1: Four Different Ways of Computing SpGEMM Kernel – (a) Inner product approach; (b) Outer product approach; (c) Row-wise product approach; and (d) Column-wise product approach. The non-zero elements in the two input matrices are shown in blue and green colors, the non-zero elements in the output matrix are shown in orange color, and the elements of the matrices involved in the computation are shown with dark borders.

In matrix multiplication, since each of the input matrices can be accessed in either a row-major order or column-major order, there are four possible ways to perform matrix multiplication — inner product (row times column), outer product (column times row), row-wise product (row times row), and column-wise product (column times column) as shown in Fig. 5.1. In the following subsections, I discuss each of these four approaches in terms of the data reuse and their on-chip memory requirements. I define data reuse as the number of multiply-accumulate (MAC) operations performed when a single byte of data is read/written from/to the memory. For the sake of simplicity, I assume that (1) each of the input and the output matrices have dimensions of $N \times N$; (2) both the input matrices have nnz number of non-zeros; (3) the output matrix has nnz' number of non-zeros; and (4) the number of non-zero elements for each row/column are the same and equal to $\frac{nnz}{N}$ for input matrices and $\frac{nnz'}{N}$ for the output matrix.

5.2.1 Inner Product

This is arguably the most widely-known approach for computing matrix multiplication, where a dot product is performed between a sparse row from the first matrix and a sparse column from the second matrix as shown in Eq. (5.1). With this approach, I can parallelize the computation of multiple dot products across different PEs. Fig. 5.1a shows the inner product approach for SpGEMM computation and the parallelization strategy.

$$C[i,j] = \sum_{k=0}^{N} A[i,k] * B[k,j] \tag{5.1}$$

This approach reads a row of sparse matrix A and column of sparse matrix B each of which has $\frac{nnz}{N}$ non-zeros, and performs index matching and MACs. As the number of non-zeros in the output matrix is nnz', the probability that such index matching produces a useful output (i.e., any of the two indices actually matched) is $\frac{nnz'}{N^2}$. Thus the data reuse for the inner product approach is $O(\frac{nnz'/N^2}{nnz/N})$, which is $O(\frac{nnz'}{nnz}.\frac{1}{N})$. Since N can be very large and nnz' is similar to nnz, the data reuse of the inner product approach is very low for large matrices. The on-chip memory requirements for this approach is $O(\frac{nnz}{N})$. Since N can be of the order of $100K - 10M$ and $\frac{nnz}{N}$ is of the order of $10 - 100$, as can be seen from Table 4.5 the data reuse for inner product approach is low; however, the on-chip memory requirements are also low.

Thus the inner product approach has three major disadvantages for SpGEMM:

1. The two input matrices need to be stored in different formats, one row major and another column major.
2. It attempts to compute each element of the output matrix. However, in the case of SpGEMM, the output matrix is typically also sparse, which leads to a significant amount of wasted computation. For example, in Fig. 5.1 when the inner product is performed between the last row of matrix A and last column of matrix B, none of the indices match and the output is a zero.
3. It performs a dot product of two sparse vectors, which is very inefficient since it requires index matching where a MAC is performed only when the indices of the non-zero elements from the two vectors match. For example, in Fig. 5.1a the inner product of first row of A and second column of B requires three index matching operations but performs only one MAC.

5.2.2 Outer Product

With this approach, an outer product is performed between a sparse column of the first matrix and a sparse row of the second matrix to produce partial results for the entire output matrix as shown in Eq. (5.2). The outer product approach parallelizes the computation of different outer products across different PEs. Fig. 5.1b shows the outer product approach and the parallelization strategy.

$$C[:,:] = \sum_{k=0}^{N} A[:,k] * B[k,:] \tag{5.2}$$

This approach reads a column and a row of the sparse input matrices A and B, each with $\frac{nnz}{N}$ non-zeros and performs an outer product with $(\frac{nnz}{N})^2$ MAC operations. Thus the data reuse for the outer product approach is $O(\frac{nnz}{N})$. The on-chip memory requirement for the outer product approach is $O(\frac{nnz}{N} + nnz')$, where the first term is the on-chip storage required for rows/columns of input matrices and the second term is the on-chip requirement for the output matrix. Thus, using typical ranges of $\frac{nnz}{N}$ from 10-100 and for nnz' from 100K-10M, the outer product approach reuses the data read/written from/to the memory 10-100 time. However, it requires an on-chip memory size of hundreds of mega-bytes.

The outer product algorithm solves two major problems with the inner product approach by computing only non-zero entries of the output matrix and not performing any index matching. However, it has three major disadvantages:

1. The two input matrices still need to be stored in different formats, one column major and another row major.
2. Multiple PEs produce the partial sums for the entire output matrix, as shown in Fig. 5.1b, which requires synchronization among them. This limits the scalability of the hardware.
3. For output reuse the partial sums are typically stored on-chip. This requires a large buffer since the partial sums from the entire output matrix are produced for each outer product. For example, OuterSPACE [PBP^{+}18], which employs the outer product approach, uses 0.5MB of on-chip memory (256KB of scratchpads within the PEs, 256KB of L0 caches and 16KB of victim caches). Yet it still cannot hold all the partial sums on the chip as the size of partial sums varies from 10-100MB.

5.2.3 Row-Wise Product

In the row-wise product approach (also known as Gustavson's algorithm [Gus78]), all the non-zero elements from a single row of matrix A are multiplied with the non-zero entries from the corresponding rows of matrix B, where the row index of matrix B is determined by the column index of non-zero values from matrix A. The results are accumulated in the corresponding row of the output matrix as shown in Eq. (5.3). Multiple rows from the output matrix can be computed in parallel. Fig. 5.1c illustrates the row-wise product approach and the parallelization strategy. Fig. 5.2 gives a more concrete example that illustrates the computation of SpGEMM with two PEs using the row-wise product approach. Here, each of the two PEs reads entire rows of the `A` and `B` matrices and writes entire rows of the output matrix `C`.

$$C[i,:] = \sum_{k=0}^{N} A[i,k] * B[k,:] \tag{5.3}$$

With this approach, a PE reads a scalar value from matrix A and a row of matrix B with $\frac{nnz}{N}$ non-zeros and perform an scalar-vector product with $(\frac{nnz}{N})$ MAC operations. While the data reuse is low, the on-chip memory requirement for this approach is only $O(\frac{nnz}{N} + \frac{nnz'}{N})$. Here the two terms represent the on-chip storage required by the non-zeros from a row of B and a row of the output matrix, respectively. Thus, using typical ranges of $\frac{nnz}{N}$ and $\frac{nnz'}{N}$ from 10-100, the row-wise product approach only requires an on-chip buffer size of a few kilo-bytes. The key advantages of using the row-wise product are as follows:

- **Consistent Formatting**: The row-wise product accesses both the input matrices and the output matrix in row-major order that allows using the same format for the inputs and outputs. Since many algorithms such as graph contractions require a chain of matrix multiplications having a consistent format in essential.
- **Elimination of Index Matching**: The core computation in this approach is the scalar-vector product; hence it computes only non-zero entries of the output matrix and does not require inefficient index matching of the inputs as in the case of the inner product approach.
- **Elimination of Synchronization**: This approach allows multiple PEs to compute different rows of the sparse output matrix and hence there is no need to synchronize the reads and writes to the output memory.
- **Low On-Chip Memory Requirements** Since a single output row is computed at a time, the partial sums for only a single output row needs to be stored on-chip. The required output buffer size is in the order of $O(\frac{nnz'}{N})$. This is contrast with the outer product approach, which requires the entire output matrix to be stored on chip with a buffer size of $O(nnz')$. As N is typically very large (on the order of 100K–10M), the on-chip memory savings from the row-wise product approach over outer product approach are significant.

  The row-wise product approach also has some disadvantages: (a) on-chip data-reuse for for matrix `B` is low as compared to outer product; and (b) just like other approaches row-wise product needs to cope with the load imbalance issue among multiple PEs. The low on-chip data reuse has more impact on the performance when the density of matrix `B` is high. This, however, is not the case with most of the SpGEMM algorithms where both the operand matrices are highly sparse. The load imbalance issue can mostly be solved using a round-robin row allocation strategy discussed in Section 5.4.1.

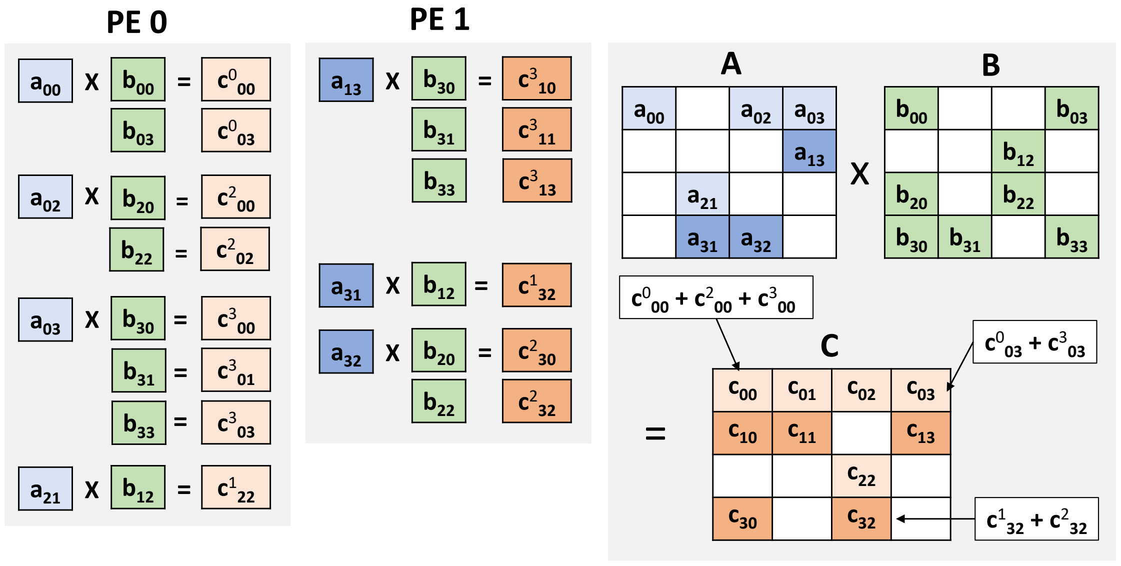


Figure 5.2: Parallelization of the Row-Wise Product on two PEs – PE0 is assigned to rows 0 and 2 of input matrix `A` and computes rows 0 and 2 of output matrix `C`; PE1 is assigned rows 1 and 3 of the matrix `A` and computes rows 1 and 3 of the matrix `C`; the matrix `B` is shared between the two PEs.

5.2.4 Column-Wise Product

In the column-wise product approach, all the non-zero elements from a single column of matrix B are multiplied with the non-zero entries from the corresponding columns of matrix A and the results are accumulated in the corresponding column of the output matrix as shown in Eq. (5.4). Fig. 5.1d shows the column-wise product approach and the parallelization strategy.

$$C[:,j] = \sum_{k=0}^{N} A[:,k] * B[k,j] \tag{5.4}$$

This approach is similar to the row-wise product approach and has the same data reuse and on-chip memory requirements as the row-wise product. The rest of the chapter focuses on row-wise product approach for SpGEMM as it has low on-chip memory requirements and does not lose much in terms of data reuse compared to the outer product approach, especially for very sparse large matrices.

5.3 Sparse Matrix Format

By using the row-wise product, one can achieve high compute utilization and low on-chip storage requirements. In this section, I further propose a new hardware-friendly sparse storage format to achieve high utilization of the off-chip memory bandwidth, which is essential for high-performance SpGEMM computation. For scalability and high performance, the on/off-chip memories are often divided into smaller memory banks to achieve lower memory latency and higher bandwidth. I abstractly represent such memory banks as *channels* in this discussion and assume that data is interleaved in these channels in a cyclic manner. These channels can be later mapped to different DRAM channels or scratchpad banks.

5.3.1 Limitations of the Traditional CSR Format

Fig. 5.3 shows the same sparse matrix `A` as in Fig. 5.2 using its dense form (Fig. 5.3a) and the conventional compressed sparse row (CSR) format (Fig. 5.3b). Here the CSR format consists of (1) an array of (value, column id) pairs for the non-zero elements in the matrix and (2) an array of row indices, where the i^{th} index points to the first non-zero element in the i^{th} row of the matrix. In Fig. 5.3, I illustrate how the (value, column id) array can be mapped to a memory with two channels. Fig. 5.3e shows how two PEs read the data from matrix `A` and Fig. 5.4 shows how these PEs write the output matrix `C` using the row-wise product approach depicted in Fig. 5.2. I assume that the channel interleaving is 4 elements and each PE sends 4-element wide requests to the memory. As shown in Fig. 5.3e and Fig. 5.4, the CSR format has several major limitations:

1. A vectorized memory request can be split among different channels, leading to non-vectorized memory access within each channel.
2. Multiple PEs may send memory read requests to the same channel resulting in memory channel conflicts.
3. A vectorized memory read can read wasteful data that does not belong to that PE.
4. For writing the output matrix, each PE writing the i^{th} row to the memory needs to wait for all the PEs writing rows $< i$ to finish, which leads to synchronization issues.

5.3.2 The Proposed C^{2}SR Format

To overcome the aforementioned limitations, I propose a new sparse storage called *channel cyclic sparse row* (C^{2}SR), where each row is assigned a fixed channel in a cyclic manner. Fig. 5.3c shows the cyclic assignment of rows to the channels and Fig. 5.3d shows the corresponding C^{2}SR

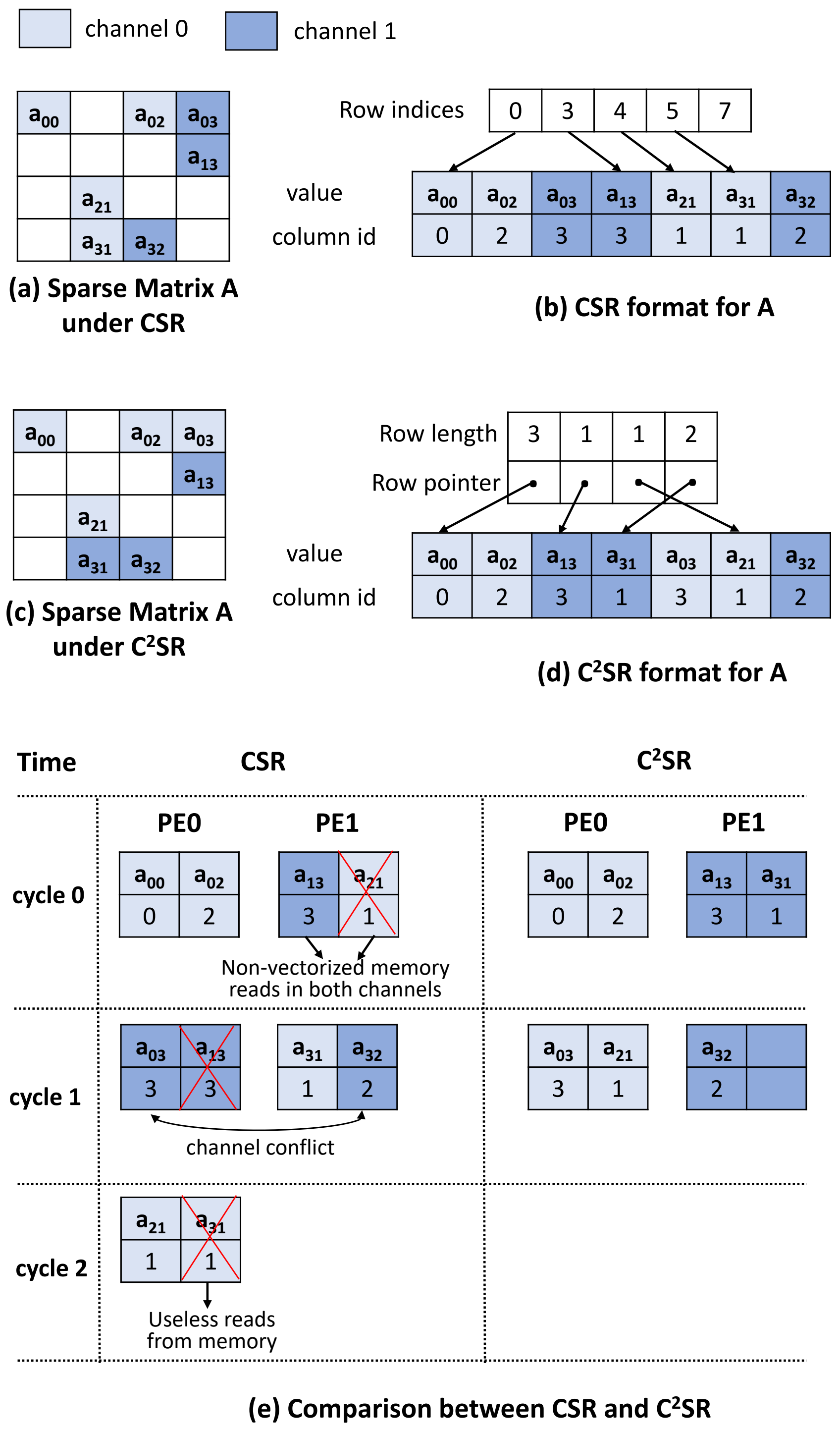


Figure 5.3: Comparison of Sparse Storage Formats – (a) shows the sparse matrix A in its dense representation; (b) shows matrix A stored in the CSR storage format and assignment of non-zero elements to two channels using 4-element channel interleaving; (c) shows the same sparse matrix A in dense representation where each row is mapped to a unique channel; (d) shows the sparse matrix A in C^{2}SR format; and (e) shows the cycle by cycle execution of two PEs where PE0 and PE1 read rows 0 and 2, and rows 1 and 3 of matrix A, respectively.

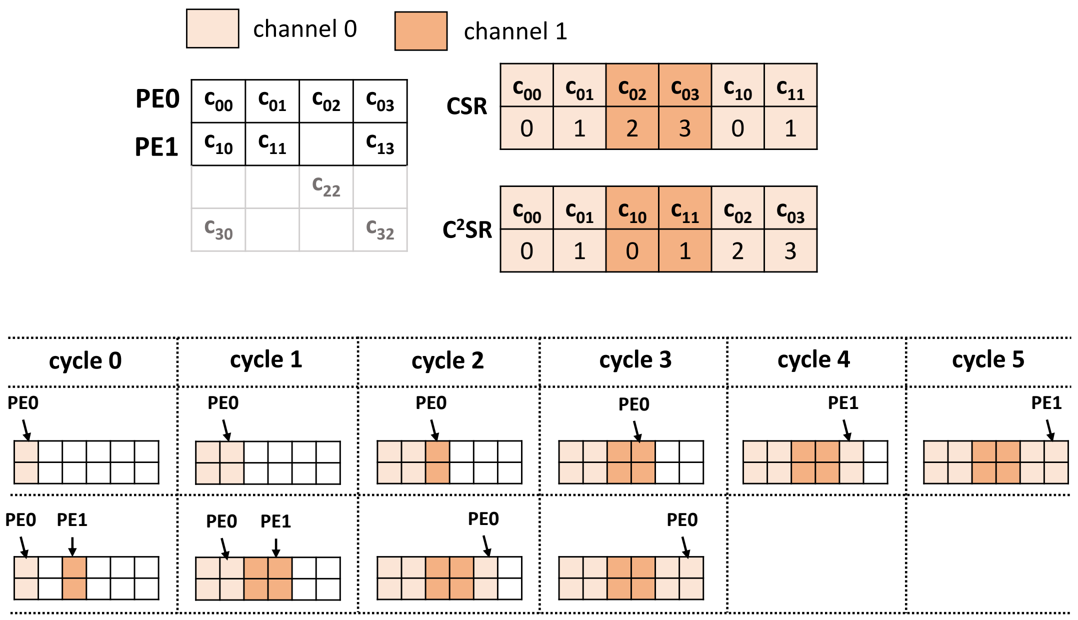


Figure 5.4: Comparison of Sparse Storage Formats CSR and C^{2}SR for Output Matrix

format. This new format consists of an array of (row length, row pointer) pairs and an array of (value, column id) pairs. The (value, column id) array stores the non-zero elements from the sparse matrix along with their column ids. The i^{th}entry in the (row length, row pointer) array stores the number of non-zeros in the i^{th} row and the pointer to the first non-zero element from that row in the (value, column id) array. To store a sparse matrix in C^{2}SR format, first each row is assigned to a fixed channel in a cyclic manner and then for each channel all non-zero elements are written serially to the memory locations corresponding to that channel in the (value, column id) array. For example in Fig. 5.3c, rows 0 and 2 are assigned to channel 0 and hence their non-zero elements are stored at the locations corresponding to channel 0 in the (value, column id) array in Fig. 5.3d. The reading of matrix `A` and writing to output matrix `C` are shown in Fig. 5.3e and 5.4. The C^{2}SR storage format has the following three key properties:

- **No Channel Conflicts:** Each row is assigned to a unique channel, which means that the rows mapped to different channels do not have memory channel conflicts and can be accessed in parallel. This is in contrast to CSR format, where different rows are not necessarily mapped to distinct channels and can result in memory channel conflicts.
- **Vectorized and Streaming Memory Reads:** All the rows mapped to a channel are stored sequentially in that channel, resulting in high spatial locality when accessing these rows in a row major order.

- **Parallel Writes to the Output Matrix:** The rows of the sparse matrix mapped to a channel can be written to that channel without requiring any information about the rows mapped to other channels. For example in Fig. 5.4, with C^2SR format PE0 and PE1 do not wait for each other and can write to the results to their corresponding channel in parallel. While using CSR format, PE1 needs to wait for PE0 to finish so that it can determine the next available memory location to write the output row.

The first two properties result in high inter-channel memory bandwidth since all the channels can work in parallel and high intra-channel memory bandwidth as the data accesses are done sequentially within a channel. These two properties help C^2SR achieve a bandwidth very close to the peak memory bandwidth, which is essential for the memory bound SpGEMM kernel. The third property allows multiple PEs to write the output elements without requiring any synchronization and thus leads to better utilization of the PEs.

5.4 MatRaptor Architecture

This section details the implementation of the row-wise product approach for SpGEMM, which improves over other conventional SpGEMM methods through: (1) consistent formatting, (2) elimination of index matching, (3) elimination of synchronization, and (4) low on-chip memory requirements. In Section 5.4.1, I first describe the two major operations in the row-wise product implementation; multiplication and merge, where for merge operation an efficient sorting technique using queues is introduced. In Section 5.4.2, I provide the details of the MatRaptor accelerator architecture, with a focus on describing the operation of the compute PE.

5.4.1 Row-wise Product Implementation

An important consideration for sparse kernels such as SpGEMM is load balancing among the PEs. Many real-world sparse matrices follow the power-law distribution, in which some of the rows can be completely dense while others may be almost empty. This can result in load balancing issues in the cases when: (i) different PEs work in lock step, meaning all the PEs finish processing one row each before starting to process the next row; and (ii) the rows of a sparse matrix mapped to one PE have significantly more non-zero elements than the number of non-zeros in the rows mapped to

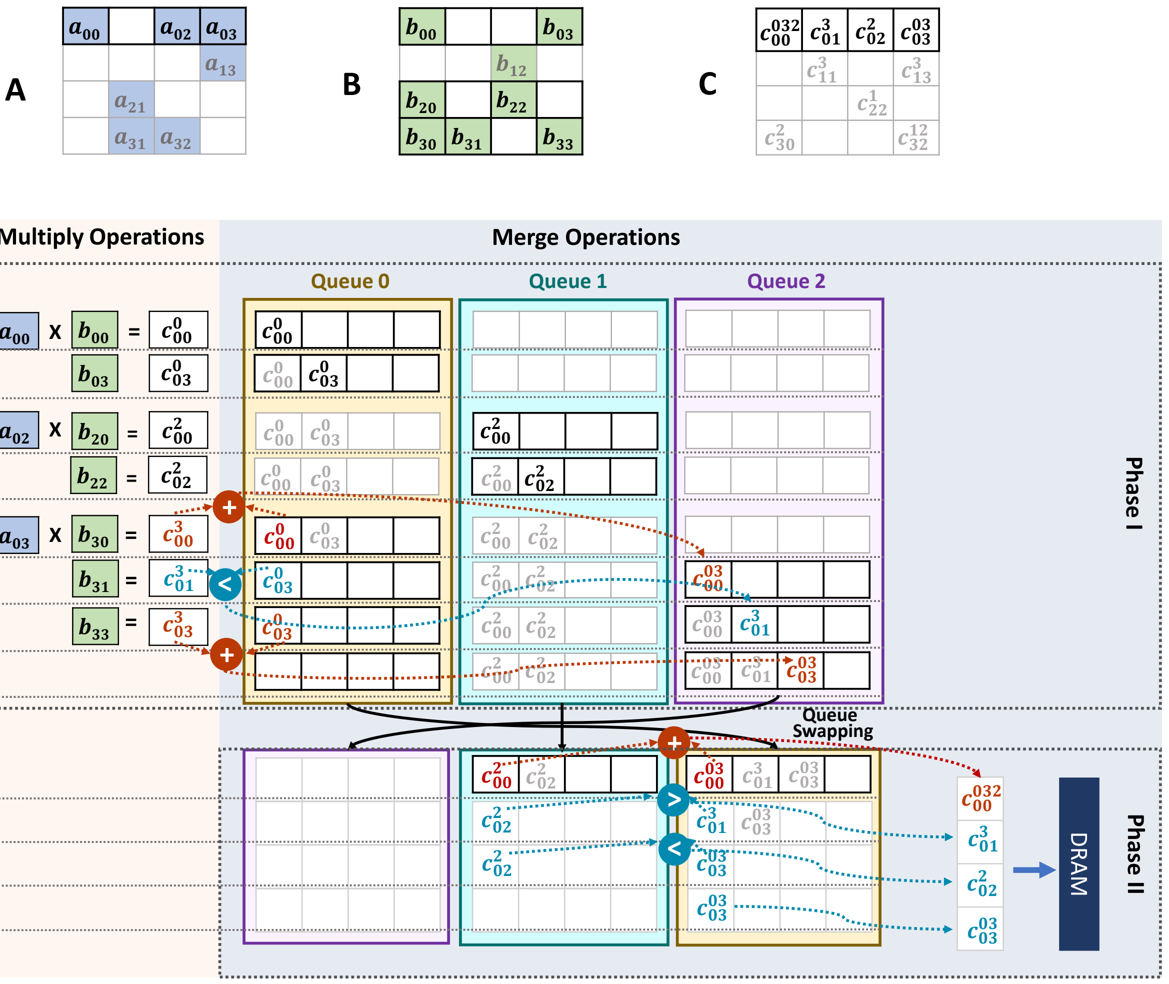


Figure 5.5: Illustration of Multiply and Merge Operations Involved in Computing the Results for a Single Row of the Output Matrix – Phase I corresponds to the cycles when the multiplications are performed and the result of the multiplication is merged with the $(data, col\ id)$ values in one of the queues; and Phase II corresponds to the cycles when the $(data, col\ id)$ values in different queues are merged together and streamed out to the DRAM.

a different PE. MatRaptor solves (i) by allowing all the PEs to work completely asynchronously. Such asynchronous execution is made possible by the C^{2}SR format, which partitions the address space of the output matrix between different PEs and allows the PEs to independently produce the results of the output matrix. MatRaptor tackles (ii) by doing a round robin allocation of rows to different PEs. This effectively ensures that for the sparse matrices with few high-density regions, the non-zero elements in these regions are approximately uniformly split among different PEs.

Fig. 5.2 shows the parallelization of SpGEMM using the row-wise product approach for two PEs. Here, rows 0 and 2 of the input matrix `A` are assigned to PE0, which is responsible for computing rows 0 and 2 of output matrix `C`. Rows 1 and 3 of the input matrix `A` are assigned to PE1, which is responsible for computing the corresponding rows of the output matrix C. The input matrix B is shared between both PEs. The PEs read the input matrices A and B stored in the memory in C^{2}SR format, perform the SpGEMM computation, and write the non-zero elements of the output matrix C to the memory using the same C^{2}SR format.

The SpGEMM computation within a PE consists of two major operations: (1) *multiply operations*, which correspond to the scalar-vector multiplications shown in Fig. 5.5; and (2) *merge operations*, which involves sorting and accumulations. From now onwards, I will use the following notations: (1) The non-zero elements from the input matrices A and B will be represented as a_{ik} and b_{kj} where the subscripts represent the row index and the column index of the matrix elements, respectively; (2) A non-zero element of the output matrix will be represented as either c_{ij}^{k} or $c_{ij}^{k_0,k_1,...k_m}$ where $c_{ij}^{k} = a_{ik} * b_{kj}$ and $c_{ij}^{k_0,k_1,...k_m} = c_{ij}^{k_0} + c_{ij}^{k_1} + ... + c_{ij}^{k_m}$. The subscripts i and j represent the row index and the column index of the non-zero element in matrix C.

Multiply Operations. For the multiply operations, a PE reads non-zero elements a_{ik} from the i^{th} row of matrix A assigned to it. For each a_{ik}, it then reads all the non-zero elements $\{b_{kj_1}, b_{kj_2}, b_{kj_3}, ...\}$ from the k^{th} row of matrix B and performs a scalar-vector multiplication to produce the partial results $\{c_{ij_1}^{k}, c_{ij_2}^{k}, c_{ij_3}^{k}, ...\}$ for the i^{th} row of matrix C. For example in Fig. 5.2, PE0 reads the non-zero element a_{00} and the 0^{th} of matrix B and performs scalar-vector multiplication on $\{b_{00}, b_{03}\}$ to produce the partial results $\{c_{00}^{0}, c_{03}^{0}\}$ for the 0^{th} row of matrix C.

Merge Operations. The partial result vectors for the i^{th} row of the output matrix C need to be merged to produce the final results. This requires sorting all the partial sum vectors with respect to their column indices j and then accumulating the partial results that have the same column index. One naïve solution is to collect all the partial sum vectors and sort them using sorting hardware. However, such sorting hardware would be inefficient as it would not make use of the property that each of the partial sum vectors is already sorted with respect to the column indices.

A better approach to solve such a sorting problem is by using multiple queues. In this approach, each PE has Q $>$ 2 queues, where each queue maintains the invariant that its entries are always sorted with respect to column indices. Out of all the queues, all except one queue act as primary queues while the remaining one acts as a helper queue. For each row of the output matrix, the first (Q$-$1) partial sum vectors are written to one of the primary queues such that there is only one partial sum vector per queue. After the first (Q$-$1) partial result vectors, each partial result vector is merged with the queue with the least number of entries and the results are written to the helper queue. While merging, if the column indices in the partial result vector and the top of the queue match, the entry is removed from both the partial result vector and the queue and their sum is pushed to the helper queue. If the column indices do not match, then the one (either partial sum vector or queue) with the smaller column index is popped and the value is pushed to the helper queue. After the merge is complete, the helper queue is swapped with the primary queue involved in the merge, and the primary queue becomes the new helper queue. This process continues until the row index of the non-zero element from `A` changes.

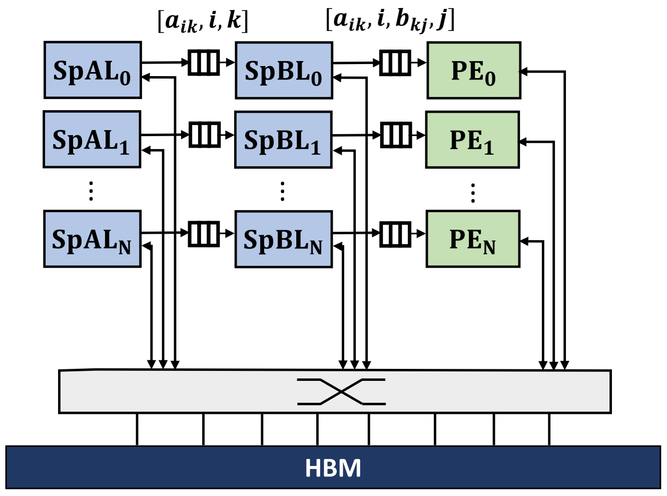
Fig. 5.5 shows the merge part of the computation with three queues. Initially, the first two queues are the primary ones and the last is a helper, and the partial result vectors $\{c^0_{00}, c^0_{03}\}$ and $\{c^2_{00}, c^2_{02}\}$ are inserted into queues 0 and 1. Then the partial sum vector $\{c^3_{00}, c^3_{01}, c^3_{03}\}$ is merged with queue 0 as it has the least number of entries and the results are written to the helper queue.

When the row index of the non-zero element from `A` changes, then the entries in all the queues need to be merged and written back to the main memory. To merge the data in all the queues, the queue with the smallest column index is popped and the data is streamed to the main memory. In the case when multiple queues have the same minimum column index at the top of the queue, all such queues are popped and the sum of the popped elements is streamed to the main memory, as shown in Fig. 5.5. After the last non-zero element from the first row of matrix `A` is processed, queue 0 and queue 1 are merged and the results are streamed to the DRAM.

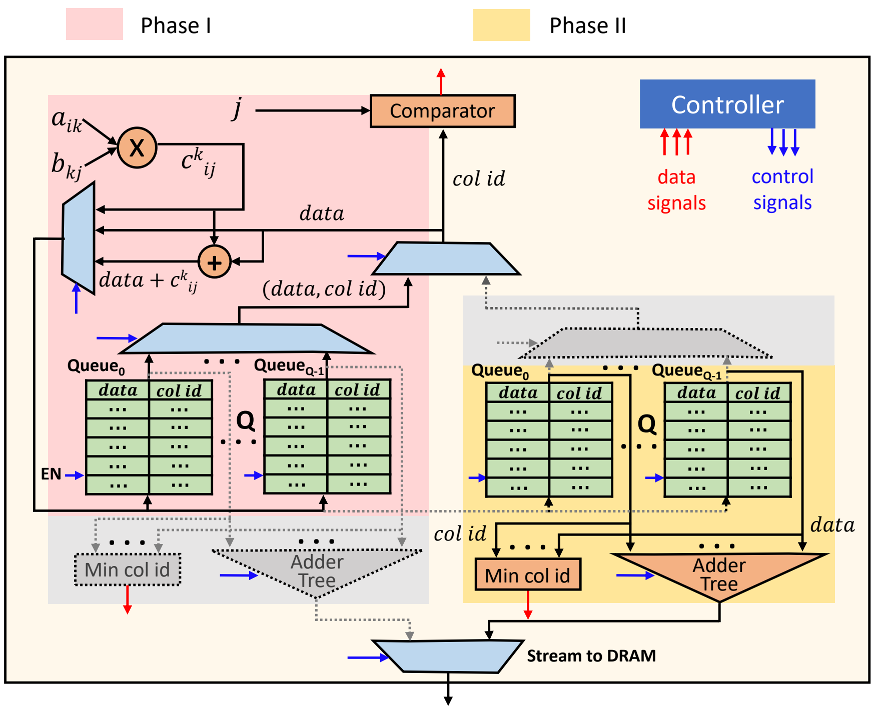
5.4.2 Architectural Details of MatRaptor

Fig. 5.6a shows the micro-architecture of MatRaptor. It consists of Sparse Matrix A Loaders (SpAL), Sparse Matrix B Loaders (SpBL), and the compute PEs. SpALs, SpBLs, and the PEs implement a one-dimensional systolic array with N rows. The rows of the input matrix `A` are assigned to the rows of the systolic array in a round-robin fashion.

Each SpAL reads the non-zero elements a_{ik} from a row of matrix `A` and sends it along with its row and column indices to SpBL. SpBL uses the column index k received from SpAL to fetch the non-zero elements from k^{th} row of matrix `B` (i.e., b_{kj}), and sends a_{ik}, b_{kj}, i and j to the main compute PEs. The PE performs multiplication and merge computations where it multiplies a_{ik} and b_{kj} and merges the results and writes them to the main memory. The following subsections describe each component of MatRaptor micro-architecture in more detail.



(a)



(b)

Figure 5.6: MatRaptor Architecture – (a) shows the MatRaptor microarchitecture consisting of Sparse A Loaders (SpALs), Sparse B Loaders (SpBLs), Processing Elements (PEs), system crossbar and high-bandwidth memory (HBM); (b) shows the microarchitecture of a single PE consisting of multpliers, adders, comparator and queues on the left performing phase I of the multiply and merge operations, and the queues on the right, adder tree and minimum column index logic performing phase II of the merge operations.

SpAL

The SpAL is configured with the number of rows N in the sparse matrix `A` and the pointer to the beginning of the arrays containing the (row length, row pointer) pairs in the C^{2}SR storage of matrix `A`. SpAL first sends a memory read request to fetch the (row length, row pointer) pair for a row of matrix `A`. Then it sends multiple memory read requests using the row pointer to fetch the (value, column id) pairs in that row. To achieve high memory bandwidth, SpAL sends wide memory read requests for (value, column id) pairs such that size of the memory request is the same as the channel interleaving size and thus implements vectorized memory reads. SpAL also implements outstanding requests and responses queue to avoid stalling for the memory responses and thus is able send and receive memory requests and responses in a pipelined manner. Once a (value, column id) pair is read from the memory, SpAL sends the values along with its row and column indices i.e. (a_{ik}, i, k) to SpBL.

SpBL

The SpBL receives (a_{ik}, i, k) values from the SpAL and sends a memory read request for the (row length, row pointer) pair in the k^{th} row of matrix `B`. It then uses the row pointer to send multiple memory read requests for the (value, column id) pairs in k^{th} row of the `B` matrix. Similar to SpAL, SpBL also loads the (value, column id) pairs in vectorized streaming manner and maintains outstanding requests and responses queue. When a (value, column id) pair is read from the memory, SpBL sends the values a_{ik}, b_{kj}, i and j to the PE.

PEs

Each PE receives (a_{ik}, b_{kj}, i, j) values from the SpBL and performs the multiply and merge operations. Fig. 5.6b shows the design of a single PE. It consists of a multiplier to calculate a_{ik} times b_{kj} and produce the partial result c_{ij}^{k}. It also consists of two sets of Q queues, where each queue contains $(data, col\ id)$ pairs. The reason behind having two set of queues is that the merge operations in Fig. 5.5 can be divided into two phases: *Phase I*, when the multiplications are performed and the result of the multiplication is merged with the $(data, col\ id)$ values in one of the queues; and *Phase II*, when all the partial sums for a single output row have been written to one of the queues and the $(data, col\ id)$ values in different queues are merged together and streamed out to the DRAM.

Since Phase II stalls the multiply operations, this can lead to poor utilization of the multipliers. With two sets of queues, when Phase I is completed, the multipliers can start computing the results for the next output row in a different set of queues while the results from the current queues are merged and written to the DRAM. With this kind of double buffering, Phase I and Phase II can be performed in parallel, which results in higher compute utilization.

All the queues within a set are connected to a multiplexer, which is used to select the queue with the least number of entries. The queues within a set are also connected to an adder tree and minimum column index selection logic. The minimum column index logic outputs a Q-bit signal where each bit represents whether the corresponding queue has the minimum column index. The output of the minimum column index logic is then sent to the controller, which configures the adder tree to accumulate the *data* values from the selected queues. The controller also pops an element from each of these queues. Fig. 5.6b shows the PE when the set of the queues on the left are involved in Phase I computation and the set of queues on the right are involved in Phase II of the computation. The inactive components from both the sets are shown with gray color and dotted lines.

If the number of rows of the systolic array is an integer multiple of the number of channels, then each row of the systolic array will read/write the elements of matrix `A/C` from/to a unique channel; however, multiple rows of the systolic array can access the data from the same channel. If the number of channels is an integer multiple of the number of rows of the systolic array, then no two rows of the systolic array will share a channel while a single row of systolic array will be assigned more than one channel. For the cases when the number of rows in the systolic array and the number of channels are the same, each row of the systolic array is assigned one channel.

5.5 Experimental Setup

5.5.1 Simulation Infrastructure

To evaluate the performance of our accelerator, I model a MatRaptor architecture consisting of SpALs, SpBLs, PEs, and HBM using the gem5 simulator [BBB$^+$11]. I implement the systolic array with 8 rows to match the number of channels in the HBM. Each PE consists of 10 queues that are implemented as SRAMs and where each queue is 8KB in size. The memory request and response queues are implemented with 64 entries. For HBM, I used the gem5 memory model for HBM, which supports up to 8 128-bit physical channels, runs at 1GHz clock frequency and provides a peak memory bandwidth of 128 GB/s.

We attach MatRaptor to a host CPU as a co-processor where both the CPU and MatRaptor share the same off-chip memory. For the host CPU we use the RISCV minor CPU model in gem5. We add support for custom instructions to send/receive messages between the CPU and MatRaptor in the CPU model and the RISCV gcc compiler. When the CPU executes these custom instructions in the program binary, it sends the size of matrices and the pointer to the sparse data to the accelerator, which then performs the SpGEMM computation.

5.5.2 Measurements

I implemented the PEs in RTL using PyMTL [LZB14], translated them to Verilog, synthesized them using the Synopsys Design Compiler for synthesis and Cadence Innovus for place-and-route, targeting a TSMC 28nm library. I modeled the latency, area and power of the queues in the merge logic and outstanding request and responses queues using CACTI 7.0 [MBJ09]. For the SpALs and SpBLs, since the area and power are dominated by outstanding memory requests and response queues, I use the area and power numbers for these queues from CACTI and add 10% overhead for the control logic. Table 5.1 shows the area and power breakdown of different components of the design. For HBM I use the energy numbers from [Shi16]. Overall the area of our accelerator is $2.2mm^2$, which is $31.3\times$ smaller than the area of OuterSPACE ($70.2mm^2$ after technology scaling). The main reason behind this is our PEs and on-chip memory are much simpler than the PEs, scratchpads and caches in OuterSPACE.

5.5.3 Baselines

I compare our design against three baselines: CPU, GPU, and OuterSPACE [PBP+18].

CPU: I use the Intel Math Kernel Library (MKL) to evaluate the benchmarks on both single thread and multiple threads (12 threads) of Intel Xeon E5-2699 v4 server-class CPU, which is manufactured using 14nm technology node, runs at 2.20 GHz and has 32 KB L1 cache, 256 KB L2 cache,

Table 5.1: Area and Power Breakdown of MatRaptor

Component	Area (mm^2)	%	Power (mW)	%
PE	1.981	88.44 %	1050.57	78.46 %
– Logic	0.080	3.61 %	43.08	3.22 %
– Sorting Queues	1.901	84.83 %	1007.49	75.24 %
SpAL	0.129	5.78 %	144.15	10.77 %
SpBL	0.129	5.78 %	144.15	10.77 %
Total	2.241	100 %	1338.89	100 %

and 55 MB of L3 cache. We kept the number of CPU threads in the multi-threaded version the same as the one used in OuterSPACE [PBP$^+$18] for their CPU baseline. The CPU uses DDR4 with 4 DRAM channels and supports a peak memory bandwidth of 76.8 GB/s. Since, SpGEMM is primarily memory-bound and the Intel CPU supports a peak memory bandwidth of only 76.8 GB/s while HBM used for MatRaptor has a peak bandwidth of 128 GB/s, we also scale the performance and energy efficiency of the CPU accordingly to 128 GB/s for comparison purposes. For energy estimation of CPU and DRAM, we use the energy numbers from McPAT [LAS$^+$09] and [Bal14].

GPU: I use cuSPARSE [NCVK10] to evaluate the benchmarks on a modern GPU NVIDIA Titan Xp, which has GDDR5x DRAM with a peak bandwidth of 547.6 GB/s and a peak 32-bit performance of 12.15 TFLOP/s. I use CUDA 9.1 for programming the GPU. For power estimation, I use the TDP number from the GPU datasheet.

Accelerator: I also compare our work against OuterSPACE, the state-of-the-art SpGEMM accelerator, which uses the outer product approach. I obtained the performance numbers for all benchmarks from the authors of OuterSPACE and used those numbers for comparison. For energy comparisons, I used the power numbers from [PBP$^+$18] which are in 32nm technology node and scale them to 28nm.

5.5.4 Technology Node Scaling

We scale the energy numbers for all these baselines to 28nm technology node. The dynamic power can be estimated as $\alpha f C V_{dd}^2$, where α is the switching activity, f is the clock frequency, C is the total capacitance and V_{dd} is the supply voltage. As α remains the same between technology nodes, capacitance C scales proportional to the square of Contacted Gate Poly Pitch (CPP) and V_{dd} is different for different technology nodes, we use the ratio of the square of CPP as a scaling factor for C and the ratio of V_{dd} to scale the power and energy numbers. We obtain the CPP and V_{dd} values for different technology nodes from their technical specifications [tecd, tecc, tecb, teca].

Table 5.2: Matrices from SuiteSparse [DH11] with Their Dimensions, Number of Non-Zeros (nnz), Density and Problem Domain

Matrix	Short name	Dim	nnz	$\frac{nnz}{N}$	Density	Domain
web-Google	wg	916K × 916K	5.1M	5.6	6.1e-6	Web Graph
mario002	m2	390K × 390K	2.1M	5.4	1.3e-5	2D/2D problem
amazon0312	az	401K × 401K	3.2M	8.0	1.9e-5	Copurchase network
m133-b3	mb	200K × 200K	801K	4.0	2.0e-5	Combinatorics
scircuit	sc	171K × 171K	959K	5.6	3.2e-5	Circuit simulation
p2pGnutella31	pg	63K × 63K	148K	2.4	3.7e-5	p2p network
offshore	of	260K × 260K	4.2M	16.3	6.2e-5	EM Problem
cage12	cg	130K × 130K	2.0M	15.6	1.1e-4	Weighted graph
2cubes-sphere	cs	101K × 101K	1.6M	16.2	1.5e-4	EM Problem
filter3D	f3	106K × 106K	2.7M	25.4	2.4e-4	Reduction problem
emailEnron	ee	36.7K × 36.7K	368K	10.0	2.7e-4	Email network
ca-CondMat	cc	23K × 23K	187K	8.1	3.5e-4	Condensed Matter
wikiVote	wv	8.3K × 8.3K	104K	12.5	1.5e-3	Wikipedia network
poisson3Da	p3	14K × 14K	353K	26.1	1.8e-3	Fluid Dynamics

5.5.5 Datasets

For benchmarks, I used the same matrices as in OuterSPACE [PBP+18], which are taken from SuiteSparse [DH11] as shown in Table 5.2. Since OuterSPACE evaluates the performance of SpGEMM by multiplying a sparse matrix with itself (`C = A×A`), I used the same approach for our evaluation to perform a fair comparison. However, since many of the real-world applications such as graph contractions involve the sparse matrix multiplication of two different matrices, I did a performance evaluation by using different combinations of `A` and `B` matrices from Table 5.2. For this, I selected the top-left 10K × 10K submatrices from all the matrices in Table 5.2 so that the two matrices have same size but different sparsity structure representative of the real matrices. This technique has been adopted from prior work on characterizing SpGEMM performance on GPUs [KTH+17].

5.6 Evaluation

5.6.1 Bandwidth Utilization

To compare the bandwidth utilization of CSR and C^{2}SR I simulate 2, 4 and 8 PEs reading a sparse matrix from the memory in a streaming fashion. For CSR, I assume that each PE reads 8-byte data elements from the memory to avoid sending vectorized memory requests that map to different channels and cause memory alignment problem. For C^{2}SR, each PE sends a 64-byte wide streaming memory requests. For all the simulations I assume the number of PEs to be the same as the number of DRAM channels. Fig. 5.7 shows the achieved bandwidth with CSR and C^{2}SR formats. As it can be seen from the figure, the achieved bandwidth from C^{2}SR format is significantly higher than the achieved bandwidth from CSR and is also close to the theoretical peak memory bandwidth.

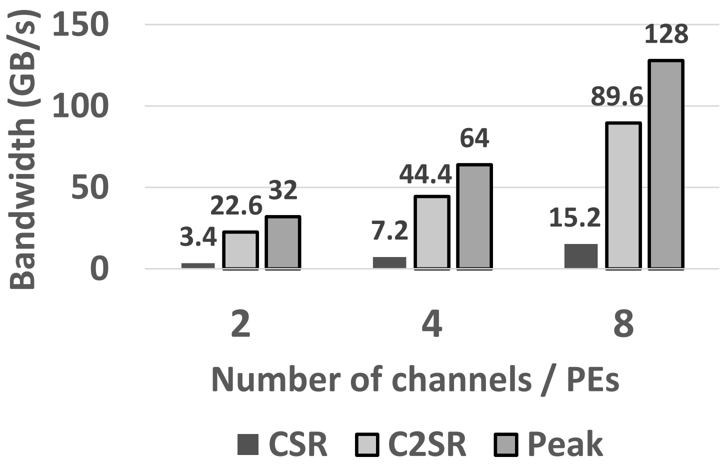


Figure 5.7: Achieved Memory Bandwidth with CSR and C^{2}SR – The number of PEs is assumed to be same as the number of channels.

5.6.2 Roofline Evaluation

Figure 5.8 shows the throughput of SpGEMM under the roofline [WWP09] of MatRaptor. The x-axis shows operation intensity, which is number of operations (multiply and add) performed for each byte of data accessed from the off-chip memory. The y-axis shows the performance in GOP/s. The horizontal line towards the right of the plot shows the peak attainable performance from the design when the operation intensity is high (kernel is compute bound) and the inclined line (with slope 1) towards the left shows the peak attainable performance when the operation intensity is

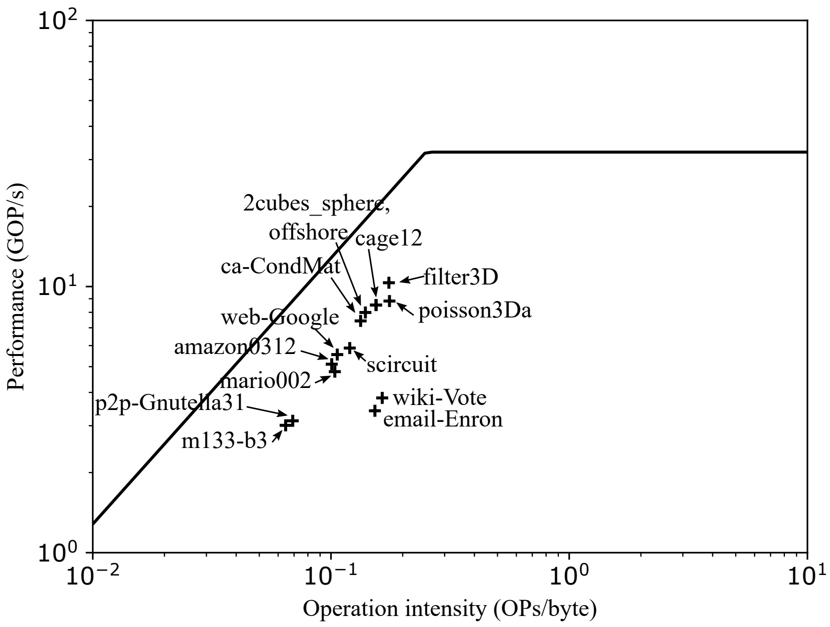
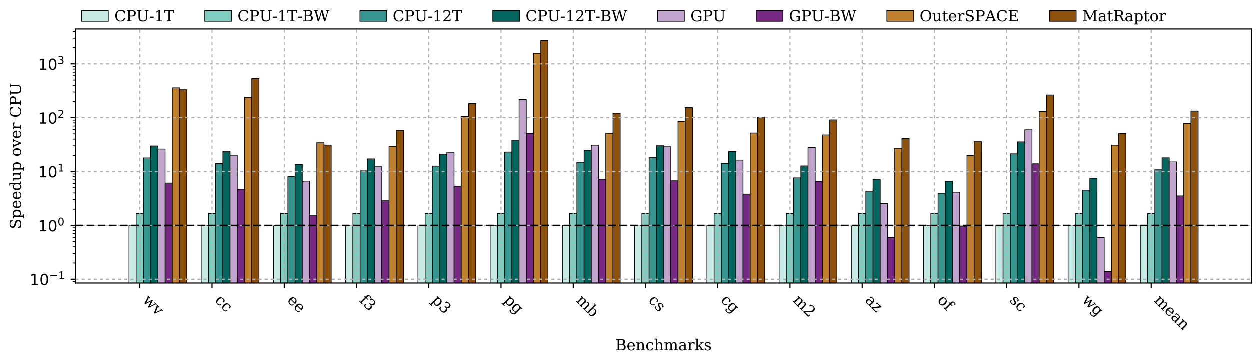


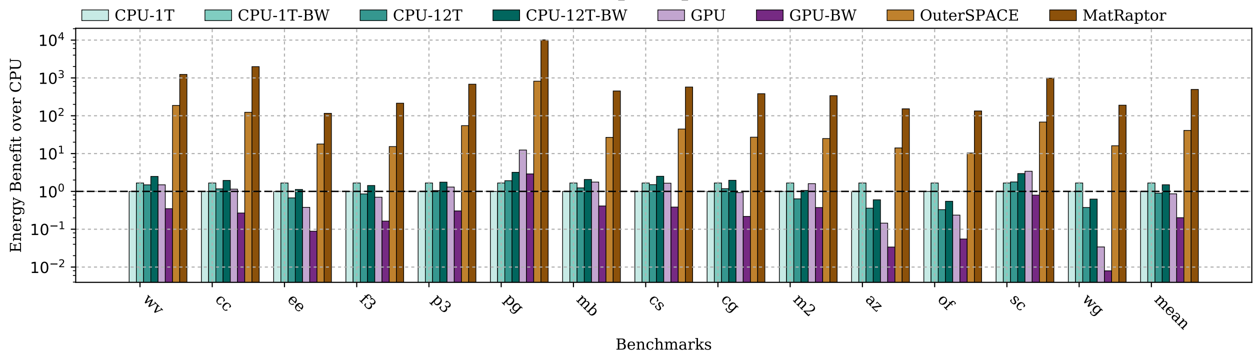
Figure 5.8: Performance of SpGEMM Under the Roofline of MatRaptor for A×A

low (kernel is memory bound). The point where these two lines intersect is called the *knee point*. The intercept of the slanted line (value of throughput for operation intensity of 1) represents the peak memory bandwidth (in GB/s). The gap between the roofline and the achieved performance of a kernel indicates the inefficiencies within the hardware and the algorithm. The MatRaptor design consists of 8 PEs, each with one MAC unit and hence the accelerator has $8 \times 2 = 16$ multipliers and adders. Since I simulate the accelerator design for a 2GHz clock frequency the peak attainable throughput is $16 \times 2 = 32$ GOP/s. For peak memory bandwidth, I use the peak bandwidth of HBM, which is 128 GB/s.

Fig. 5.8 shows the achieved throughput for SpGEMM for A $\times$ A computation for all the matrices in Table 5.2. It can be seen from the roofline, the throughput for each of the benchmark is very close to the peak performance and all the benchmarks lie in the memory bound region. The gap between the peak performance and the attained performance is due to the memory accesses to the second input matrix. As in the row-wise product approach only the first input matrix and the output matrix are partitioned among different PEs while the second input matrix is shared between different PEs. This results in memory channel conflicts and lowers the achieved memory bandwidth.



(a) Speedup

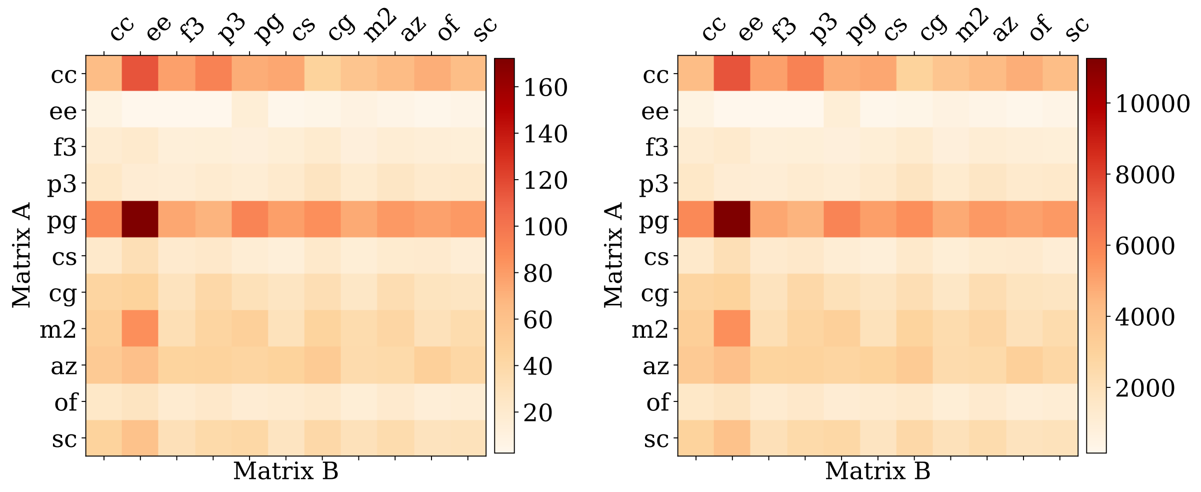


(b) Energy benefit

Figure 5.9: Speedup for **`A×A`** **over CPU –** CPU-1T = single-thread CPU; CPU-1T-BW = single-thread CPU with bandwidth normalization; CPU-12T = CPU with 12 threads; CPU-1T-BW = CPU with 12 threads and bandwidth normalization; GPU; GPU-BW = GPU with bandwidth normalization; OuterSPACE and MatRaptor. The dotted line indicates the CPU-1T baseline in both plots.

5.6.3 Performance Evaluation

Fig. 5.9(a) shows the performance of CPU (single-/multi-threaded and without/with bandwidth normalization), GPU (without/with bandwidth normalization), OuterSPACE [PBP+18] and MatRaptor over the single-threaded CPU baseline for `A` × `A` SpGEMM computation. MatRaptor outperforms CPU and GPU for all benchmarks. Compared to OuterSPACE the performance of MatRaptor is better for all the benchmarks except Wiki-Vote and Email-Enron, for which the performance of the two are very similar. The main reason behind this is that these matrices have small sizes compared to other matrices and hence the on-chip memory for the output matrix in OuterSPACE is sufficient to store the partial results, which results in similar performance of OuterSPACE and MatRaptor. In terms of geometric mean speedup, MatRaptor achieves 158.7×, 95.2×, 16.5×, 9.9×, 6.9×, 29.5 and 1.7× speedup over single-threaded CPU without and with



(a) Speedup of MatRaptor over GPU (b) Energy benefit of MatRaptor over GPU

Figure 5.10: Speedup of MatRaptor over GPU-CuSPARSE for `A`$\times$`B` **–** `A` and `B` are top-left 10K$\times$10K tiles of different matrices from the dataset.

bandwidth normalization, multi-threaded CPU without and with bandwidth normalization, GPU without and with bandwidth normalization, and OuterSPACE, respectively.

Fig. 5.10(a) shows the speedup of MatRaptor over GPU with bandwidth normalization for `A` $\times$ `B` SpGEMM computation. It can be seen from these figures that for GPU the performance of MatRaptor is better in all the cases. Overall, MatRaptor achieves 27.8$\times$ speedup over GPU with bandwidth normalization for `A` $\times$ `B` computation.

5.6.4 Energy

Fig. 5.9(b) shows the energy benefit of CPU (single-/multi-threaded and without/with bandwidth normalization), GPU (without/with bandwidth normalization), OuterSPACE [PBP+18] and MatRaptor over the single-threaded CPU baseline for `A` $\times$ `A` SpGEMM computation. In terms of geometric mean energy benefit, MatRaptor achieves 495.9$\times$, 297.6$\times$, 551.5$\times$, 330.9$\times$, 574.7$\times$, 2458.9$\times$ and 12.3$\times$ energy benefit over single-threaded CPU without and with bandwidth normalization, multi-threaded CPU without and with bandwidth normalization, GPU without and with bandwidth normalization, and OuterSPACE, respectively.

Fig. 5.10(b) show the energy benefit of MatRaptor over GPU with bandwidth normalization for `A` $\times$ `B` SpGEMM computation. Overall MatRaptor achieves 1815.9$\times$ energy improvement over GPU.

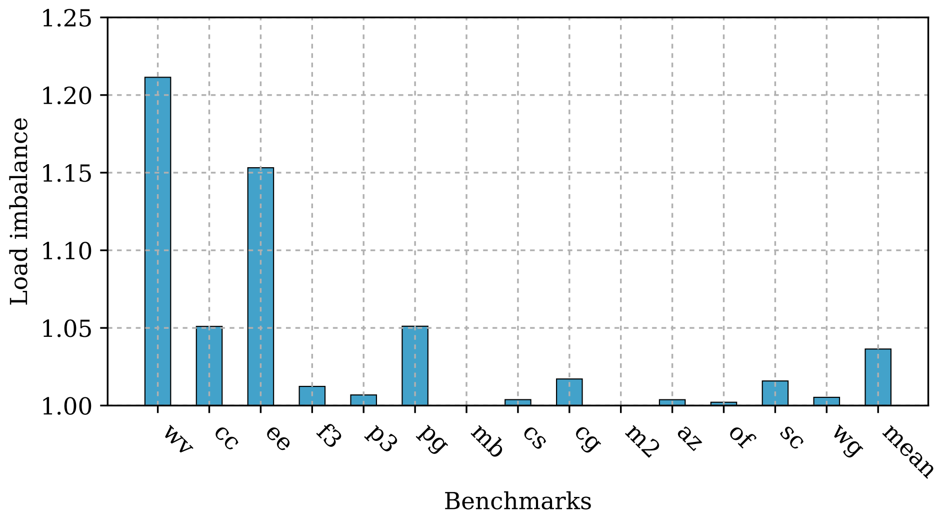


Figure 5.11: Load Imbalance — measured as the ratio of the maximum and minimum number of non-zeros of matrix A assigned to the PEs.

5.6.5 Load Imbalance

To measure the load imbalance due to the power-law distribution of the sparse matrices as discussed in Section 5.4.1, we determine the total number of non-zeros of matrix A assigned to each PE by C^2SR format and plot the ratio of maximum and minimum number of non-zeros in these PEs. The minimum value of such ratio is 1, which means no load imbalance and a higher ratio means a higher load imbalance. Fig. 5.11 shows the load imbalance; except for `wv` and `ee` the load imbalance for all the benchmarks is less than 5%. For `wv` and `ee` the load imbalance is higher because these matrices are small and thus a round-robin row assignment to PEs is not very effective.

5.7 Discussion on Sparse Format

The current implementation assumes that the number of virtual channels used to create C^2SR matches the number of physical channels in the DRAM, which results in highly efficient SpGEMM processing. To make the sparse format portable across different platforms, the sparse matrix can be stored in CSR format and converted to C^2SR (or vice versa) by a target-specific software library or dedicated logic in the accelerator. The complexity of such conversion is O(nnz) which is much lower than that of SpGEMM O(nnz*nnz/N).

More importantly, the cost of format conversion gets amortized due to: (1) In SpGEMM, the rows of matrix B are read multiple times while for the format conversion they are read only once; (2) For algorithms like local graph clustering, the same sparse matrix is reused multiple times; (3) Many other algorithms such as graph contractions perform a chain of matrix multiplications and thus the output matrix becomes the input of another SpGEMM without requiring additional format conversion.

To evaluate the performance overhead of CSR to C^2SR conversion and vice versa, we designed a simple hardware unit that reads the sparse matrix in CSR format and stores it back to memory in C^2SR. According to our results, the format conversion takes on average 12% of the SpGEMM execution time on MatRaptor.

5.8 Related Work

5.8.1 Sparse Storage Formats

Many sparse storage formats have been proposed in the literature. CSR (Compressed Sparse Row), CSC (Compressed Sparse Column) and COO (Co-ordinate) are the most commonly used sparse storage formats for CPUs. Liu *et al.* [LWSD17] proposes a sparse tensor storage format F-COO, which is similar to the co-ordinate format and used it for GPUs. CSF [SRSK15] and Hi-COO [LSV18] are other sparse tensor storage formats that are based on CSR and COO, respectively. For machine learning hardware, researchers have proposed multiple variants of CSR and CSC formats. For example, Cambricon-X [ZDZ$^+$16] introduce a modification of CSR format where the non-zeros are are compressed and stored in contiguous memory and index vectors are used to decode the row and column indices. EIE [HLM$^+$16] uses a variant of the CSC storage format where instead of storing the row indices they store the number of zeros before a non-zero element. However, since these works focus on deep learning, especially CNNs, their sparse storage format is specialized for low sparsity (high density). For SpGEMM involving matrices with high sparsity, OuterSPACE [PBP$^+$18] uses a variant of CSR and CSC formats called CR and CC. However, to solve the issues related to channel conflicts and memory misalignment, it uses caches instead to directly accessing the DRAM from the hardware and thus spends 18$\times$ more area for on-chip memory compare to MatRaptor. The area of the hardware is important especially when using HBM since the hardware is placed on logic die of HBM which has limited number of transistors. Fowers *et al.* [FOS+14] propose a sparse storage format called *compressed interleaved sparse row* (CISR), which also maps different PEs to different DRAM channels for sparse matrix dense vector multiplication (SpMV). However, this format can only be used for the input matrices and not for the output matrix as generating the output in this format requires information about all the non-zero elements in the output matrix.

5.8.2 CPU/GPU Acceleration

Akbudak *et al.* [AA17] design hypergraph and bipartite graph models for 1D row-wise partitioning of matrix A in SpGEMM to evenly partition the work across threads. Saule *et al.* [SKÇ13] investigate the performance of the Xeon Phi coprocessor for SpMV computation. Sulatycke *et al.* [SG98] propose a sequential cache-efficient algorithm and illustrated high performance than existing algorithms for sparse matrix multiplication for CPUs. The works involving GPU acceleration of SpGEMM computation include [DOB15,GHS+15,LV14,MIK12]. Kiran *et al.* [MIK12] explore the load-balancing problem that only considers the band matrices. Weifeng and Brian [LV14] apply the techniques such as GPU merge path algorithm and memory pre-allocation to improve the performance and the storage issue. Felix *et al.* [GHS+15] reduce the overhead of memory access by merging several sparse rows using the main kernel. Steven *et al.* [DOB15] decompose the SpGEMM operations and leverage bandwidth saving operations like layered graph model. They also perform the SpGEMM in a row-wise product method to balance the workload and improve the performance.

5.8.3 Custom Accelerators

For SpGEMM, prior works involving FPGA implementations for sparse-dense and sparse-sparse matrix-matrix and matrix-vector accelerators include [LXH+19], ESE [HKM+17], [ZP05] and [FOS+14]. Lu *et al.* [LXH+19] proposes a CNN accelerator with sparse weights. ESE [HKM+17] proposes an FPGA-accelerator for SpMV in LSTMs. Prasanna *et al.* [ZP05] and Fowers *et al.* [FOS+14] designs SpMV accelerators for very sparse matrices. Lin *et al.* [LWS13] proposed an FPGA-based architecture for sparse matrix-matrix multiplication.

Prior works involving ASIC implementations include Cambricon-S [ZDG$^+$18], Cnvlutin [AJH$^+$16], SCNN [PRM$^+$17], [AKM$^+$18], OuterSPACE [PBP$^+$18] and ExTensor [HAMP$^+$19]. Cambricon-S [ZDG$^+$18] implements hardware accelerator for SpGEMM in CNNs where both weight matrices and neurons are sparse. SCNN [PRM$^+$17] proposes a SpGEMM accelerator for CNNs which can also exploit the sparsity in both weights and neurons. Anders *et al.* [AKM$^+$18] designs accelerator designs for SpGEMM. EIE [HLM$^+$16] proposes SpMSpV (sparse matrix sparse vector multiplication) accelerator for fully connected layers in CNN and show significant performance gains over CPU and GPU. However, all these works focus on deep learning applications where the density is really high. TPU [JYP$^+$17] implemented a 2-d systolic array for GEMM.

OuterSPACE [PBP$^+$18] and ExTensor [HAMP$^+$19] are two recent works that propose hardware accelerators for SpGEMM computation on very sparse matrices. However, OuterSPACE applies the outer product approach and ExTensor applies the inner product approach for SpGEMM, the inefficiencies of which have been discussed in Section 5.2. Yavits and Ginosar [YG17] explores a juxtaposed resistive content addressable memory (CAM) and RAM-based SpGEMM. Zhu *et al.* [ZGS$^+$13] introduces a 3D-stacked logic-in-memory system by placing logic layers between DRAM dies to accelerate a 3D-DRAM system for sparse data access and built a custom CAM architecture to speed-up the index-alignment process of column-wise product approach.

5.9 Conclusion

In this chapter, I present a novel row-wise product based accelerator (MatRaptor) for SpGEMM, which achieves high performance and energy-efficiency over CPU, GPU and the state-of-the-art SpGEMM accelerator OuterSPACE. It also has $7.2\times$ lower power consumption and $31.3\times$ smaller area compared to OuterSPACE. To achieve this, I introduce a new hardware-friendly sparse storage format named C^{2}SR, which improves the memory bandwidth utilization by enabling vectorized and streaming memory accesses. I also implement a novel sorting hardware to merge the partial sums in the SpGEMM computation and prototyped and simulated MatRaptor using gem5 on a diverse set of matrices.

CHAPTER 6
CONCLUSION

This dissertation presents hardware accelerators, a spatial language for generating hardware and storage formats for accelerating sparse and dense tensor computations. The problem of accelerating tensor computations in hardware is approached from two angles: creating a a general framework for productively designing high-performance accelerators for dense tensor computations, and building versatile accelerators that can accelerate some of the common dense and sparse tensor computations. As Moore's Law and Dennard scaling slow down and the computation requirements increase with surge in machine learning, there is an urgent need of energy-efficient and high-performance hardware. This dissertation addresses this challenge by designing energy-efficient and high-performance accelerators for tensor computations, which are the integral part of most machine learning applications.

6.1 Dissertation Summary and Contributions

Chapters 1 and 2 motivated the need for energy efficient hardware for tensor computations. In the current era of data explosion, machine learning techniques have become pervasive tools in emerging large-scale commercial applications and tensor algebra is at the heart of these machine learning techniques. As Moore's law and Dennard scaling slow down and computation demands increase, energy-efficient and high-performance hardware for tensor computations is needed. In this dissertation, I provided an extensive background on tensors, tensor factorizations, applications of tensor factorizations, matrix kernels and their applications. Both sparse tensor kernels and sparse storage formats form an important part of this dissertation. The co-design of sparse storage formats and hardware for sparse tensor kernels can address the needs of high-performance and energy-efficient tensor computations.

Chapter 3 of this dissertation explored the hardware acceleration of dense tensor computations on spatial hardware such as FPGAs and CGRAs. I provided a language and compilation framework for productively generating high-performance systolic arrays for dense tensor kernels on spatial architectures. The proposed language decouples a functional specification from a spatial mapping, and allows programmers to quickly explore various spatial optimizations for the same tensor kernel. The techniques proposed in this chapter such as unrolling, double buffering, vectorization and data serialization provide a key set of hardware optimizations that are essential for generating high-performance accelerators for dense tensor computations, which are later generalized to the mixed sparse-dense and dense tensor computations in subsequent chapters.

Chapter 4 explored the hardware acceleration of both dense and mixed sparse-dense tensor computations and used the domain knowledge gained from Chapter 3. Unlike hardware for dense tensor computations, a key requirement for achieving high-performance for sparse tensor kernels is a sparse storage format which can allow efficient memory accesses for parallel compute units in the hardware. In this chapter, I co-designed a hardware and a sparse storage format, which allows accessing the sparse data in vectorized and streaming fashion and maximizes the utilization of the memory bandwidth. For designing such an accelerator, I extracted a common computation pattern that is found in major matrix and tensor computation kernels and implemented it in the hardware. Thus by designing the hardware based on this common compute pattern, I could not only accelerate tensor factorizations but also mixed sparse-dense matrix operations.

Chapter 5 further explored the hardware acceleration of sparse-sparse matrix-matrix multiplication (SpGEMM), which is by far the most important sparse-sparse tensor kernel. I presented MatRaptor, a novel SpGEMM accelerator that is high performance and highly resource efficient. Unlike conventional methods using inner or outer product as the meta operation for matrix multiplication, in this chapter, I use a row-wise product approach, which offers a better trade-off in terms of data reuse and on-chip memory requirements, and achieves higher performance for large sparse matrices. I further proposed a new hardware-friendly sparse storage format, which allows parallel compute engines to access the sparse data in a vectorized and streaming fashion, leading to high utilization of memory bandwidth.

To reiterate the major contributions of this dissertation are:

- It proposes a concise yet expressive programming abstraction that decouples spatial mapping from the functional specification of a tensor computation.
- It identifies a set of key spatial optimizations that are essential for creating high-performance spatial hardware for dense tensor computations.
- It proposes a language and compilation framework to productively generate high-performance systolic arrays for dense tensor computations.

- It proposes the *first* hardware accelerator designed for mixed sparse-dense tensor factorizations. This accelerator is both versatile and adaptable and supports several mixed sparse-dense matrix operations for a wide range of sparsity.
- It proposes a new sparse storage format, scalable compressed interleaved sparse row (SCISR), which allows accessing sparse data in vectorized and streaming fashion and thus helps sparse accelerators in achieving high memory bandwidth utilization and performance for sparse tensor kernels.
- It systematically analyzes different dataflows for sparse-sparse matrix-matrix multiplication by comparing and contrasting them against data reuse and on-chip memory requirements. It also shows that a row-wise product approach, which has not been explored in the design of sparse-sparse matrix-matrix multiplication accelerators, has the potential to outperform existing approaches.
- It proposes C^{2}SR, a new hardware-friendly sparse storage format that allows different parallel processing engines to access the data in a vectorized and streaming manner leading to high utilization of the available memory bandwidth. This format solves the same challenges as in SCISR format; however, unlike SCISR which can be used only for sparse input matrices but not sparse output matrices, C^{2}SR can be used for both.

6.2 Future Work

6.2.1 T2S-Tensor for sparse tensors

T2S-Tensor allows expressing a varying range of tensor kernels and performs spatial optimizations to generate high-performance systolic arrays. However, it only focuses on dense tensor computations as the programming abstractions; the intermediate representation and many of the code transformations are based on Halide [RKBA$^+$13] that was primarily built for optimizing image processing pipelines. Recently, TACO [KKC$^+$17], a domain-specific language (DSL) for sparse tensor computations, was proposed in which the sparse algorithms can be expressed in the same way as in Halide along with a new way of representing sparse storage formats. It uses a notion of merge lattices to automatically generate the CPU code for sparse tensor kernels. There is great opportunity in implementing the ideas of lattice merging from TACO in T2S-Tensor to add support for sparse tensor computation. One interesting work that emerged after TACO is ExTensor [HAMP$^+$19], which uses the ideas of lattice merging to create a domain-specific accelerator for sparse matrix kernels. There is an already existing idea of enabling sparse matrix-vector computation in T2S-Tensor language [Ron18]; however, it has not yet been realized in the language and the compiler.

6.2.2 Processing In Memory for Tensaurus

Tensaurus accelerates dense and mixed sparse-dense tensor computations. As it can be seen from the results for the sparse tensor benchmarks, most of the kernels are memory bound with low data-reuse and hence can benefit significantly from a memory system that provides a higher memory bandwidth. Near-memory computing (NMC) is an attractive solution for such high bandwidth requirements. NMC aims at processing the data close to where it resides. This data-centric approach places the compute units close to the data and attempts to minimize expensive data movements. Three-dimensional memory stacking is the true enabler for NMC as it allows stacking of logic and memory together using through-silicon vias (TSVs). Micron's Hybrid Memory Cube (HMC) [Paw11], High Bandwidth Memory (HBM) [LKK$^+$14] from AMD and Hynix, and Samsung's Wide I/O [KOL$^+$11] are few examples of such memory technologies.

6.2.3 Accelerating Sparse Convolution Neural Networks on MatRaptor

MatRaptor accelerates sparse-sparse matrix multiplication using a row-wise product approach. Such an approach is more efficient compared to the outer-product approach when the operand matrices are large and very sparse as the disadvantage corresponding to low data-reuse is compensated by the low on-chip memory requirements. However, for sparse deep learning applications such as convolutional neural networks that involve sparse-sparse matrix multiplication the size and the sparsity of the operand matrices is low, which makes the outer-product approach more efficient solution. An interesting direction would be to build a reconfigurable accelerator for sparse-sparse matrix multiplication which can operate in two modes: (1) using a row-wise product approach for large and very sparse matrices, and (2) using an outer-product approach for small and less sparse matrices.

BIBLIOGRAPHY

[AA17] K. Akbudak and C. Aykanat. "Exploiting Locality in Sparse Matrix-Matrix Multiplication on Many-Core Architectures". *IEEE Trans. on Parallel and Distributed Systems*, 2017.

[ABC$^+$16] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. TensorFlow: A System for Large-Scale Machine Learning.". *USENIX Symp. on Operating Systems Design and Implementation*, 2016.

[ABG15] A. Azad, A. Buluç, and J. Gilbert. "Parallel Triangle Counting and Enumeration Using Matrix Algebra". *Workshop in Int'l Symp. on Parallel and Distributed Processing*, 2015.

[acm] "AMD Core Math Library". *https://developer.amd.com/amd-aocl/amd-math-library-libm/.*

[AJH$^+$16] J. Albericio, P. Judd, T. Hetherington, T. Aamodt, N. E. Jerger, and A. Moshovos. "Cnvlutin: Ineffectual-Neuron-Free Deep Neural Network Computing". *ACM SIGARCH Computer Architecture News*, 2016.

[AKM$^+$18] M. Anders, H. Kaul, S. Mathew, V. Suresh, S. Satpathy, A. Agarwal, S. Hsu, and R. Krishnamurthy. "2.9 TOPS/W Reconfigurable Dense/Sparse Matrix-Multiply Accelerator with Unified INT8/INTI6/FP16 Datapath in 14NM Tri-Gate CMOS". *IEEE Symp. on VLSI Circuits*, 2018.

[atl] "Automatically Tuned Linear Algebra Software". *http://math-atlas.sourceforge.net/.*

[BAA$^+$19] S. Balay, S. Abhyankar, M. Adams, J. Brown, P. Brune, K. Buschelman, L. Dalcin, A. Dener, V. Eijkhout, W. Gropp, et al. "PETSc Users Manual". *Argonne National Laboratory*, 2019.

[Bal14] R. Balasubramonian. `https://my.eng.utah.edu/~cs7810/pres/14-7810-02.pdf`, 2014.

[BBB$^+$11] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti, et al. "The gem5 Simulator". *ACM SIGARCH Computer Architecture News*, 2011.

[BFF$^+$09] A. Buluç, J. T. Fineman, M. Frigo, J. R. Gilbert, and C. E. Leiserson. "Parallel Sparse Matrix-Vector and Matrix-Transpose-Vector Multiplication Using Compressed Sparse Blocks". *Int'l Symp. on Parallelism in Algorithms and Architectures*, 2009.

[BHP$^+$17] M. Baskaran, T. Henretty, B. Pradelle, M. H. Langston, D. Bruns-Smith, J. Ezick, and R. Lethin. "Memory-Efficient Parallel Tensor Decompositions". *IEEE High Performance Extreme Computing Conference (HPEC)*, 2017.

[BHR18] G. Ballard, K. Hayashi, and K. Ramakrishnan. "Parallel Nonnegative CP Decomposition of Dense Tensors". *Int'l Conf. on High Performance Computing (HiPC)*, 2018.

[BKK19] G. Ballard, A. Klinvex, and T. G. Kolda. "TuckerMPI: A Parallel C++/MPI Software Package for Large-Scale Data Compression via the Tucker Tensor Decomposition". *arXiv preprint arXiv:1901.06043*, 2019.

[BL$^+$07] J. Bennett, S. Lanning, et al. "The Netflix Prize". *Proceedings of KDD Cup and Workshop*, 2007.

[bla] "Basic Linear Algebra Subprograms". *http://www.netlib.org/blas/*.

[BLD18] Z. Blanco, B. Liu, and M. M. Dehnavi. "CSTF: Large-Scale Sparse Tensor Factorizations on Distributed Platforms". *Int'l Conf. on Parallel Processing*, 2018.

[BM$^+$00] W. L. Briggs, S. F. McCormick, et al. "A Multigrid Tutorial". *Siam*, 2000.

[BMG16] T. Becker, O. Mencer, and G. Gaydadjiev. "Spatial Programming with OpenSPL". *FPGAs for Software Programmers*, 2016.

[BP98] S. Brin and L. Page. "The Anatomy of a Large-Scale Hypertextual Web Search Engine". *Computer Networks and ISDN Systems*, 1998.

[BTK$^+$14] A. Beutel, P. P. Talukdar, A. Kumar, C. Faloutsos, E. E. Papalexakis, and E. P. Xing. "FlexiFaCT: Scalable Flexible Factorization of Coupled Tensors on Hadoop". *SDM*, 2014.

[BVR$^+$12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović. "Chisel: Constructing Hardware in a Scala Embedded Language". *Design Automation Conf. (DAC)*, 2012.

[BYF$^+$09] A. Bakhoda, G. L. Yuan, W. W. Fung, H. Wong, and T. M. Aamodt. "Analyzing CUDA Workloads Using a Detailed GPU Simulator". *Int'l Symp. on Performance Analysis of Systems and Software*, 2009.

[CBK$^+$10] A. Carlson, J. Betteridge, B. Kisiel, B. Settles, E. R. Hruschka Jr., and T. M. Mitchell. "Toward an Architecture for Never-Ending Language Learning.". *AAAI*, 2010.

[CC70] J. D. Carroll and J.-J. Chang. "Analysis of Individual Differences in Multidimensional Scaling via an N-way Generalization of “Eckart-Young” Decomposition". *Psychometrika*, 1970.

[CCA$^+$11] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. H. Anderson, S. Brown, and T. Czajkowski. "LegUp: High-Level Synthesis for FPGA-based Processor/Accelerator Systems". *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2011.

[CCJ$^+$17] V. T. Chakaravarthy, J. W. Choi, D. J. Joseph, X. Liu, P. Murali, Y. Sabharwal, and D. Sreedhar. "On Optimizing Distributed Tucker Decomposition for Dense Tensors". *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2017.

[CCX$^+$16] Y. Chen, T. Chen, Z. Xu, N. Sun, and O. Temam. "DianNao Family: Energy-Efficient Hardware Accelerators for Machine Learning". *Communications of the ACM*, 2016.

[CES16] Y.-H. Chen, J. Emer, and V. Sze. "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks". *ACM SIGARCH Computer Architecture News*, 2016.

[CFO$^+$18] E. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. Caulfield, T. Massengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, et al. "Serving DNNs in Real Time at Datacenter Scale with Project Brainwave". *IEEE Micro*, 2018.

[CHW$^+$13] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. "Deep Learning with COTS HPC Systems". *Int'l Conf. on Machine Learning*, 2013.

[CLC18] J. Choi, X. Liu, and V. Chakaravarthy. "High-performance dense tucker decomposition on GPU clusters". *Int'l Conf. for High Performance Computing, Networking, Storage, and Analysis*, 2018.

[CLN$^+$11] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. "High-Level Synthesis for FPGAs: From Prototyping to Deployment". *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2011.

[CLSS18] J. Choi, X. Liu, S. Smith, and T. Simon. "Blocking Optimization Techniques for Sparse Tensor Computation". *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2018.

[CM08] P. Coussy and A. Morawiec. "High-Level Synthesis: from Algorithm to Digital Circuit". *Springer Science & Business Media*, 2008.

[CMDL$^+$15] A. Cichocki, D. Mandic, L. De Lathauwer, G. Zhou, Q. Zhao, C. Caiafa, and H. A. Phan. "Tensor Decompositions for Signal Processing Applications: From Two-Way to Multiway Component Analysis". *IEEE Signal Processing Magazine*, 2015.

[CPS06] K. Chellapilla, S. Puri, and P. Simard. "High Performance Convolutional Neural Networks for Document Processing". *Int'l Workshop on Frontiers in Handwriting Recognition*, 2006.

[CW18] J. Cong and J. Wang. "PolySA: Polyhedral-based Systolic Array Auto-compilation". *Int'l Conf. on Computer-Aided Design (ICCAD)*, 2018.

[CWV$^+$14] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer. "cuDNN: Efficient Primitives for Deep Learning". *arXiv preprint arXiv:1410.0759*, 2014.

[CWZZ18] Y. Cheng, D. Wang, P. Zhou, and T. Zhang. "Model Compression and Acceleration for Deep Neural Networks: The Principles, Progress, and Challenges". *IEEE Signal Processing Magazine*, 2018.

[Dav14] T. Davis. "CSparse: A Concise Sparse Matrix Package", 2014.

[DGR$^+$74] R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. "Design of Ion-Implanted MOSFET's With Very Small Physical Dimensions". *IEEE Journal of Solid-State Circuits*, 1974.

[DH11] T. A. Davis and Y. Hu. "The University of Florida Sparse Matrix Collection". *ACM Trans. on Mathematical Software (TOMS)*, 2011.

[DHP02] I. S. Duff, M. A. Heroux, and R. Pozo. "An Overview of the Sparse Basic Linear Algebra Subprograms: The New Standard from the BLAS Technical Forum". *ACM Trans. on Mathematical Software (TOMS)*, 2002.

[DK17] L. De Lathauwer and E. Kofidis. "Coupled Matrix-Tensor Factorizations — The Case of Partially Shared Factors". *Asilomar Conf. on Signals, Systems, and Computers*, 2017.

[DLDMV00a] L. De Lathauwer, B. De Moor, and J. Vandewalle. "A Multilinear Singular Value Decomposition". *SIAM Journal on Matrix Analysis and Applications*, 2000.

[DLDMV00b] L. De Lathauwer, B. De Moor, and J. Vandewalle. "On The Best Rank-1 and Rank-(r 1, r 2,..., rn) Approximation of Higher-Order Tensors". *SIAM Journal on Matrix Analysis and Applications*, 2000.

[DN07] P. D'alberto and A. Nicolau. "R-Kleene: A High-Performance Divide-And-Conquer Algorithm for the All-Pair Shortest Path for Densely Connected Networks". *Algorithmica*, 2007.

[DOB15] S. Dalton, L. Olson, and N. Bell. "Optimizing Sparse Matrix—Matrix Multiplication for the GPU". *ACM Trans. on Mathematical Software (TOMS)*, 2015.

[DZB$^+$14] E. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus. "Exploiting Linear Structure Within Convolutional Networks for Efficient Evaluation". *Int'l Conf. on Neural Information Processing Systems (NIPS)*, 2014.

[ESA$^+$06] H. Esmaeilzadeh, P. Saeedi, B. N. Araabi, C. Lucas, and S. M. Fakhraie. "Neural Network Stream Processing Core (NnSP) for Embedded Systems". *Int'l Symp. on Circuits and Systems*, 2006.

[FAP$^+$12] K. E. Fleming, M. Adler, M. Pellauer, A. Parashar, A. Mithal, and J. Emer. "Leveraging Latency-insensitivity to Ease Multiple FPGA Design". *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2012.

[FO17] E. Frolov and I. Oseledets. "Tensor Methods and Recommender Systems". *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 2017.

[FOS$^+$14] J. Fowers, K. Ovtcharov, K. Strauss, E. S. Chung, and G. Stitt. "A High Memory Bandwidth FPGA Accelerator for Sparse Matrix-Vector Multiplication". *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2014.

[GDWL12] D. D. Gajski, N. D. Dutt, A. C. Wu, and S. Y. Lin. "High-Level Synthesis: Introduction to Chip and System Design". *Springer Science & Business Media*, 2012.

[GHS$^+$15] F. Gremse, A. Hofter, L. O. Schwen, F. Kiessling, and U. Naumann. "GPU-Accelerated Sparse Matrix-Matrix Multiplication by Iterative Row Merging". *SIAM Journal on Scientific Computing*, 2015.

[GJ$^+$10] G. Guennebaud, B. Jacob, et al. "Eigen". *URl: http://eigen. tuxfamily. org*, 2010.

[GL14] Y. Goldberg and O. Levy. "word2vec Explained: deriving Mikolov et al.'s negative-sampling word-embedding method". *arXiv preprint arXiv:1402.3722*, 2014.

[GLN$^+$14] N. George, H. Lee, D. Novo, T. Rompf, K. J. Brown, A. K. Sujeeth, M. Odersky, K. Olukotun, and P. Ienne. "Hardware System Synthesis from Domain-Specific Languages". *Int'l Conf. on Field Programmable Logic and Applications (FPL)*, 2014.

[GRS08] J. R. Gilbert, S. Reinhardt, and V. B. Shah. "A Unified Framework for Numerical and Combinatorial Computing". *Computing in Science & Engineering*, 2008.

[Gus78] F. G. Gustavson. "Two Fast Algorithms for Sparse Matrices: Multiplication and Permuted Transposition". *ACM Trans. on Mathematical Software (TOMS)*, 1978.

[HAMP$^+$19] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher. "ExTensor: An Accelerator for Sparse Tensor Algebra". *Int'l Symp. on Microarchitecture (MICRO)*, 2019.

[Hås90] J. Håstad. "Tensor Rank is NP-Complete". *Journal of Algorithms*, 1990.

[HBD$^+$14] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan. "Darkroom: Compiling High-level Image Processing Code into Hardware Pipelines". *ACM Trans. on Graphics*, 2014.

[HGS14] J. C. Ho, J. Ghosh, and J. Sun. "Marble: High-throughput Phenotyping from Electronic Health Records via Sparse Nonnegative Tensor Factorization". *Int'l Conf. on Knowledge Discovery and Data Mining*, 2014.

[HHM12] V. Hapla, D. Horák, and M. Merta. "Use of Direct Solvers in TFETI Massively Parallel Implementation". *Int'l Workshop on Applied Parallel Computing*, 2012.

[HKM$^+$17] S. Han, J. Kang, H. Mao, Y. Hu, X. Li, Y. Li, D. Xie, H. Luo, S. Yao, Y. Wang, et al. "ESE: Efficient Speech Recognition Engine with Sparse LSTM on FPGA". *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[HLM$^+$16] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally. "EIE: Efficient Inference Engine on Compressed Deep Neural Network". *Int'l Symp. on Computer Architecture (ISCA)*, 2016.

[HPTD15] S. Han, J. Pool, J. Tran, and W. Dally. "Learning Both Weights and Connections for Efficient Neural Network". *Advances in Neural Information Processing Systems*, 2015.

[HYL17] W. Hamilton, Z. Ying, and J. Leskovec. "Inductive Representation Learning on Large Graphs". *Advances in Neural Information Processing Systems*, 2017.

[HZRS16] K. He, X. Zhang, S. Ren, and J. Sun. "Deep Residual Learning for Image Recognition". *Int'l Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[Inta] Intel. Intel Design Examples. Https://www.intel.com/content/www/us/en/programmable/support/support-resources/design-examples/design-software/opencl/matrix-multiplication.html.

[Intb] Intel. vLab Academic Cluster. Https://wiki.intel-research.net.

[IOM95] S. Itoh, P. Ordejón, and R. M. Martin. Order-N Tight-Binding Molecular Dynamics on Parallel Computers. *Computer physics communications*, 1995.

[JVZ14] M. Jaderberg, A. Vedaldi, and A. Zisserman. "Speeding up Convolutional Neural Networks with Low Rank Expansions.". *arXiv preprint arXiv:1405.3866*, 2014.

[JYP$^+$17] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, R. Boyle, P.-l. Cantin, C. Chao, C. Clark, J. Coriell, M. Daley, M. Dau, J. Dean, B. Gelb, T. V. Ghaemmaghami, R. Gottipati, W. Gulland, R. Hagmann, C. R. Ho, D. Hogberg, J. Hu, R. Hundt, D. Hurt, J. Ibarz, A. Jaffey, A. Jaworski, A. Kaplan, H. Khaitan, D. Killebrew, A. Koch, N. Kumar,

S. Lacy, J. Laudon, J. Law, D. Le, C. Leary, Z. Liu, K. Lucke, A. Lundin, G. MacKean, A. Maggiore, M. Mahony, K. Miller, R. Nagarajan, R. Narayanaswami, R. Ni, K. Nix, T. Norrie, M. Omernick, N. Penukonda, A. Phelps, J. Ross, M. Ross, A. Salek, E. Samadiani, C. Severn, G. Sizikov, M. Snelham, J. Souter, D. Steinberg, A. Swing, M. Tan, G. Thorson, B. Tian, H. Toma, E. Tuttle, V. Vasudevan, R. Walter, W. Wang, E. Wilcox, and D. H. Yoon. "In-Datacenter Performance Analysis of a Tensor Processing Unit". *Int'l Symp. on Computer Architecture (ISCA)*, 2017.

[KAKA18] F. Kjolstad, P. Ahrens, S. Kamil, and S. Amarasinghe. "Sparse Tensor Algebra Optimizations with Workspaces". *arXiv preprint arXiv:1802.10574*, 2018.

[KB06] T. Kolda and B. Bader. "The TOPHITS Model for Higher-order Web Link Analysis". *Workshop on Link Analysis, Counterterrorism and Security*, 2006.

[KB09] T. G. Kolda and B. W. Bader. "Tensor Decompositions and Applications". *SIAM Review*, 2009.

[KFP$^+$18] D. Koeplinger, M. Feldman, R. Prabhakar, Y. Zhang, S. Hadjis, R. Fiszel, T. Zhao, L. Nardi, A. Pedram, C. Kozyrakis, and K. Olukotun. "Spatial: A Language and Compiler for Application Accelerators". *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2018.

[KGK94] G. Karypis, A. Gupta, and V. Kumar. "A Parallel Formulation of Interior Point Algorithms". *IEEE Conf. on Supercomputing*, 1994.

[KKC$^+$17] F. Kjolstad, S. Kamil, S. Chou, D. Lugato, and S. Amarasinghe. "The Tensor Algebra Compiler". *Intl'l Conf. on Object-Oriented Programming, Systems, Languages, and Applications*, 2017.

[KL79] H. Kung and C. E. Leiserson. "Systolic arrays (for VLSI)". *Sparse Matrix Proceedings*, 1979.

[KOL$^+$11] J.-S. Kim, C. S. Oh, H. Lee, D. Lee, H. R. Hwang, S. Hwang, B. Na, J. Moon, J.-G. Kim, H. Park, et al. "A 1.2 V 12.8 GB/s 2 Gb Mobile Wide-I/O DRAM With 4x128 I/Os Using TSV Based Stacking". *IEEE Journal of Solid-State Circuits*, 2011.

[KPHF12] U. Kang, E. Papalexakis, A. Harpale, and C. Faloutsos. "Gigatensor: Scaling Tensor Analysis up by 100 times-Algorithms and Discoveries". *Int'l Conf. on Knowledge Discovery and Data Mining*, 2012.

[KSV06] H. Kaplan, M. Sharir, and E. Verbin. "Colored Intersection Searching via Sparse Rectangular Matrix Multiplication". *Int'l Symp. on Computational geometry*, 2006.

[KTH$^+$17] S. E. Kurt, V. Thumma, C. Hong, A. Sukumaran-Rajam, and P. Sadayappan. "Characterization of Data Movement Requirements for Sparse Matrix Computations on GPUs". *Int'l Conf. on High Performance Computing (HiPC)*, 2017.

[KW16] T. N. Kipf and M. Welling. "Semi-Supervised Classification with Graph Convolutional Networks". *arXiv preprint arXiv:1609.02907*, 2016.

[LAH07] J. Leskovec, L. A. Adamic, and B. A. Huberman. "The Dynamics of Viral Marketing". *ACM Trans. on the Web (TWEB)*, 2007.

[LAS$^+$09] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi. "McPAT: An Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures". *Int'l Symp. on Microarchitecture (MICRO)*, 2009.

[LCH$^+$19] Y.-H. Lai, Y. Chi, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, and Z. Zhang. "HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing". *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2019.

[LCP$^+$17] J. Li, J. Choi, I. Perros, J. Sun, and R. Vuduc. "Model-Driven Sparse CP Decomposition for Higher-Order Tensors". *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2017.

[LGR$^+$14] V. Lebedev, Y. Ganin, M. Rakhuba, I. V. Oseledets, and V. S. Lempitsky. "Speeding-up Convolutional Neural Networks Using Fine-tuned CP-Decomposition". *arXiv preprint arXiv:1412.6553*, 2014.

[LHE$^+$13] J. Leng, T. Hetherington, A. ElTantawy, S. Gilani, N. S. Kim, T. M. Aamodt, and V. J. Reddi. "GPUWattch: Enabling Energy Optimizations in GPGPUs". *ACM SIGARCH Computer Architecture News*, 2013.

[LKK$^+$14] D. U. Lee, K. W. Kim, K. W. Kim, H. Kim, J. Y. Kim, Y. J. Park, J. H. Kim, D. S. Kim, H. B. Park, J. W. Shin, et al. "25.2 A 1.2 V 8Gb 8-channel 128GB/s High-Bandwidth Memory (HBM) Stacked DRAM with Wffective Microbump I/O Test Methods Using 29nm Process and TSV". *Int'l Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, 2014.

[LMV18] J. Li, Y. Ma, and R. Vuduc. "ParTI! : A Parallel Tensor Infrastructure for multicore CPUs and GPUs", 2018.
URL `https://github.com/hpcgarage/ParTI`

[LSV18] J. Li, J. Sun, and R. Vuduc. "HiCOO: Hierarchical Storage of Sparse Tensors". *Int'l Conf. for High Performance Computing, Networking, Storage and Analysis*, 2018.

[LSY12] W. Luzhou, K. Sano, and S. Yamamoto. "Domain-Specific Language and Compiler for Stencil Computation on FPGA-Based Systolic Computational-memory Array". *Int'l Conf. on Reconfigurable Computing: Architectures, Tools and Applications*, 2012.

[LV14] W. Liu and B. Vinter. "An Efficient GPU General Sparse Matrix-Matrix Multiplication for Irregular Data". *Int'l Symp. Parallel and Distributed Processing*, 2014.

[LVMQ91] H. Le Verge, C. Mauras, and P. Quinton. "The ALPHA Language and its use for the Design of Systolic Arrays". *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 1991.

[LWS13] C. Y. Lin, N. Wong, and H. K.-H. So. "Design Space Exploration for Sparse Matrix-Matrix Multiplication on FPGAs". *Int'l Journal of Circuit Theory and Applications*, 2013.

[LWSD17] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi. "A Unified Optimization Approach for Sparse Tensor Operations on GPUs". *Int'l Conf. on Cluster Computing (CLUSTER)*, 2017.

[LXH$^+$19] L. Lu, J. Xie, R. Huang, J. Zhang, W. Lin, and Y. Liang. "An Efficient Hardware Accelerator for Sparse Convolutional Neural Networks on FPGAs". *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[LZB14] D. Lockhart, G. Zibrat, and C. Batten. "PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research". *Int'l Symp. on Microarchitecture (MICRO)*, 2014.

[M$^+$65] G. E. Moore et al. "Cramming More Components onto Integrated Circuits". *McGraw-Hill New York, NY, USA*, 1965.

[MBJ09] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. "CACTI 6.0: A Tool to Model Large Caches". *HP laboratories*, 2009.

[MCCD15] T. Mikolov, K. Chen, G. S. Corrado, and J. A. Dean. "Computing numeric representations of words in a high-dimensional space". *Google Patents, US Patent 9,037,464*, 2015.

[MDCL$^+$18] S. Markidis, S. W. Der Chien, E. Laure, I. B. Peng, and J. S. Vetter. "Nvidia Tensor Core Programmability, Performance & Precision". *Workshop in Int'l Parallel and Distributed Processing Symposium*, 2018.

[MIK12] K. Matam, S. R. K. B. Indarapu, and K. Kothapalli. "Sparse Matrix-Matrix Multiplication on Modern Architectures". *Int'l Conf. on High Performance Computing*, 2012.

[mkl] "Intel Math Kernel Library". *https://software.intel.com/en-us/mkl*.

[ML13] J. McAuley and J. Leskovec. "Hidden Factors and Hidden Topics: Understanding Rating Dimensions with Review Text". *ACM Conf. on Recommender systems*, 2013.

[MLW+19] Y. Ma, J. Li, X. Wu, C. Yan, J. Sun, and R. Vuduc. "Optimizing Sparse Tensor Times Matrix on GPUs". *Journal of Parallel and Distributed Computing*, 2019.

[MNV+17] A. K. Mishra, E. Nurvitadhi, G. Venkatesh, J. Pearce, and D. Marr. "Fine-Grained Accelerators for Sparse Machine Learning Workloads". *Asia and South Pacific Design Automation Conf. (ASP-DAC)*, 2017.

[MSC+13] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. "Distributed Representations of Words and Phrases and Their Compositionality". *Int'l Conf. on Neural Information Processing Systems (NIPS)*, 2013.

[Nav97] Z. Navabi. "VHDL: Analysis and Modeling of Digital Systems". *McGraw-Hill, Inc.*, 1997.

[NCVK10] M. Naumov, L. Chien, P. Vandermersch, and U. Kapasi. "CuSPARSE Library". *GPU Technology Conference*, 2010.

[Nik08] R. S. Nikhil. "Bluespec: A General-Purpose Approach to High-Level Synthesis based on Parallel Atomic Transactions". *High-Level Synthesis*, 2008.

[NMM15] E. Nurvitadhi, A. Mishra, and D. Marr. "A Sparse Matrix Vector Multiply Accelerator for Support Vector Machine". *Int'l Conf. on Compilers, Architecture and Synthesis for Embedded Systems*, 2015.

[NS10] D. Nion and N. D. Sidiropoulos. Tensor Algebra and Multidimensional Harmonic Retrieval in Signal Processing for MIMO Radar. *IEEE Trans. on Signal Processing*, 2010.

[ope] "An Optimized BLAS Library". *https://www.openblas.net/*.

[Paw11] J. T. Pawlowski. "Hybrid Memory Cube (HMC)". *IEEE Hot Chips Symposium (HCS)*, 2011.

[PBP+18] S. Pal, J. Beaumont, D.-H. Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. Mudge, and R. Dreslinski. "OuterSPACE: An Outer Product Based Sparse Matrix Multiplication Accelerator". *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2018.

[PBY+17] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz. "Programming Heterogeneous Systems from an Image Processing DSL". *ACM Trans. on Architecture and Code Optimization (TACO)*, 2017.

[Pee07] K. Peeters. Cadabra: A Field-Theory Motivated Symbolic Computer Algebra System. *Computer Physics Communications*, 2007.

[PFS12] E. E. Papalexakis, C. Faloutsos, and N. D. Sidiropoulos. "ParCube: Sparse Parallelizable Tensor Decompositions". *Joint European Conf. on Machine Learning and Knowledge Discovery in Databases*, 2012.

[PNB$^+$18] J. Park, M. Naumov, P. Basu, S. Deng, A. Kalaiah, D. S. Khudia, J. Law, P. Malani, A. Malevich, N. Satish, J. Pino, M. Schatz, A. Sidorov, V. Sivakumar, A. Tulloch, X. Wang, Y. Wu, H. Yuen, U. Diril, D. Dzhulgakov, K. M. Hazelwood, B. Jia, Y. Jia, L. Qiao, V. Rao, N. Rotem, S. Yoo, and M. Smelyanskiy. "Deep Learning Inference in Facebook Data Centers: Characterization, Performance Optimizations and Hardware Implications". *arXiv preprint arXiv:1811.09886*, 2018.

[PPA$^+$15] M. Pellauer, A. Parashar, M. Adler, B. Ahsan, R. Allmon, N. Crago, K. Fleming, M. Gambhir, A. Jaleel, T. Krishna, D. Lustig, S. Maresh, V. Pavlov, R. Rayess, A. Zhai, and J. Emer. "Efficient Control and Communication Paradigms for Coarse-Grained Spatial Architectures". *ACM Transactions on Computer Systems (TOCS)*, 2015.

[PRM$^+$17] A. Parashar, M. Rhu, A. Mukkara, A. Puglielli, R. Venkatesan, B. Khailany, J. Emer, S. W. Keckler, and W. J. Dally. "SCNN: An Accelerator for Compressed-Sparse Convolutional Neural Networks". *Int'l Symp. on Computer Architecture (ISCA)*, 2017.

[RKBA$^+$13] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe. "Halide: A Language and Compiler for Optimizing Parallelism, Locality, and Recomputation in Image Processing Pipelines". *ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI)*, 2013.

[Ron17a] H. Rong. "Programmatic Control of a Compiler for Generating High-Performance Spatial Hardware". *arXiv preprint arXiv:1711.07606*, 2017.

[Ron17b] H. Rong. Programmatic Control of a Compiler for Generating High-performance Spatial Hardware. *arXiv preprint arXiv:1711.07606*, 2017.

[Ron18] H. Rong. Expressing Sparse Matrix Computations for Productive Performance on Spatial Architectures. *arXiv preprint arXiv:1810.07517*, 2018.

[RV89] M. O. Rabin and V. V. Vazirani. "Maximum Matchings in General Graphs Through Randomization". *Journal of Algorithms*, 1989.

[S$^+$15] C. Szegedy, , , P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. "Going Deeper with Convolutions". *Int'l Conf. on Computer Vision and Pattern Recognition (CVPR)*, 2015.

[SCL$^+$17] S. Smith, J. W. Choi, J. Li, R. Vuduc, J. Park, X. Liu, and G. Karypis. "The Formidable Repository of Open Sparse Tensors and Tools". `http://frostt.io/tensors/`, 2017.

[SDLF$^+$17] N. D. Sidiropoulos, L. De Lathauwer, X. Fu, K. Huang, E. E. Papalexakis, and C. Faloutsos. "Tensor Decomposition for Signal Processing and Machine Learning". *IEEE Trans. on Signal Processing*, 2017.

[SDMZ17] N. Srivastava, S. Dai, R. Manohar, and Z. Zhang. "Accelerating Face Detection on Programmable SoC Using C-Based Synthesis". *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2017.

[SG98] P. D. Sulatycke and K. Ghose. "Caching-Efficient Multithreaded Fast Multiplication of Sparse Matrices". *Int'l Symp. on Parallel and Distributed Processing*, 1998.

[Sha07] V. B. Shah. "An Interactive System for Combinatorial Scientific Computing With an Emphasis on Programmer Productivity". *University of California, Santa Barbara*, 2007.

[Shi16] A. Shilov. "JEDEC Publishes HBM2 Specification". `http://www.anandtech.com/show/9969/jedec-publisheshbm2-specification`, 2016.

[SHSK18] S. Smith, K. Huang, N. D. Sidiropoulos, and G. Karypis. "Streaming Tensor Factorization for Infinite Data Sources". *Int'l Conf. on Data Mining*, 2018.

[SJS$^+$20] N. Srivastava, H. Jin, S. Smith, H. Rong, D. Albonesi, and Z. Zhang. "Tensaurus: A Versatile Accelerator for Mixed Sparse-Dense Tensor Computations.". *Int'l Symp. on High-Performance Computer Architecture (HPCA)*, 2020.

[SK17] S. Smith and G. Karypis. "Accelerating the Tucker Decomposition with Compressed Sparse Tensors". *European Conference on Parallel Processing*, 2017.

[SKB$^+$12] Y. Shi, A. Karatzoglou, L. Baltrunas, M. Larson, A. Hanjalic, and N. Oliver. "TFMAP: Optimizing MAP for Top-n Context-Aware Recommendation". *Int'l Conf. on Research and Development in Information Retrieval*, 2012.

[SKÇ13] E. Saule, K. Kaya, and Ü. V. Çatalyürek. "Performance Evaluation of Sparse Matrix Multiplication Kernels on Intel Xeon Phi". *Int'l Conf. on Parallel Processing and Applied Mathematics*, 2013.

[SM18] N. Srivastava and R. Manohar. "Operation-Dependent Frequency Scaling Using Desynchronization". *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, 2018.

[SMSF18] S. Skalicky, J. Monson, A. Schmidt, and M. French. "Hot & Spicy: Improving Productivity with Python and HLS for FPGAs". *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2018.

[SRB$^+$19a] N. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. Albonesi, V. Sarkar, W. Chen, P. Petersen, G. Lowney, A. H. Herr, C. Hughes, T. Mattson, and P. Dubey. "T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations". *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[SRB$^+$19b] N. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. Albonesi, V. Sarkar, W. Chen, P. Petersen, et al. "T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations". *IEEE Symp. on Field Programmable Custom Computing Machines (FCCM)*, 2019.

[SRSK15] S. Smith, N. Ravindran, N. D. Sidiropoulos, and G. Karypis. "SPLATT: Efficient and Parallel Sparse Tensor-Matrix Multiplication". *Int'l Parallel and Distributed Processing Symp. (IPDPS)*, 2015.

[STF06] J. Sun, D. Tao, and C. Faloutsos. "Beyond Streams and Graphs: Dynamic Tensor Analysis". *Int'l Conf. on Knowledge Discovery and Data Mining*, 2006.

[SZL$^+$05] J.-T. Sun, H.-J. Zeng, H. Liu, Y. Lu, and Z. Chen. "CubeSVD: A Novel Approach to Personalized Web Search". *Int'l Conf. on World Wide Web*, 2005.

[teca] 14 nm lithography process. `https://en.wikichip.org/wiki/14_nm_lithography_process`.

[tecb] 16 nm lithography process. `https://en.wikichip.org/wiki/16_nm_lithography_process`.

[tecc] 28 nm Lithography Process. `https://en.wikichip.org/wiki/28_nm_lithography_process`.

[tecd] 32 nm lithography process. `https://en.wikichip.org/wiki/32_nm_lithography_process`.

[tece] 65 nm Lithography Process. `https://en.wikichip.org/wiki/65_nm_lithography_process`.

[tel] "TeLa the Tensor Language". *https://space.fmi.fi/prog/tela.html*.

[TM08] D. Thomas and P. Moorby. "The Verilog® Hardware Description Language". *Springer Science & Business Media*, 2008.

[TXWW15] C. Tai, T. Xiao, X. Wang, and E. Weinan. "Convolutional Neural Networks with Low-Rank Regularization". *arXiv preprint arXiv:1511.06067*, 2015.

[VAG17] A. Vasudevan, A. Anderson, and D. Gregg. "Parallel Multi Channel Convolution Using General Matrix Multiplication". *Int'l Conf. on Application-Specific Systems, Architectures and Processors (ASAP)*, 2017.

[VD00] S. M. Van Dongen. *Graph clustering by flow simulation*. Ph.D. Thesis, 2000.

[VDKV00] A. Van Deursen, P. Klint, and J. Visser. "Domain-Specific Languages: An Annotated Bibliography". *ACM Sigplan Notices*, 2000.

[VDY05] R. Vuduc, J. W. Demmel, and K. A. Yelick. "OSKI: A Library of Automatically Tuned Sparse Matrix Kernels". *Journal of Physics: Conference Series*, 2005.

[VSM11] V. Vanhoucke, A. Senior, and M. Z. Mao. "Improving the Speed of Neural Networks on CPUs". *Workshop on Deep Learning and Unsupervised Feature Learning, NIPS*, 2011.

[VSP$^+$17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. "Attention is All You Need". *arXiv preprint arXiv:1706.03762*, 2017.

[VT02] M. A. O. Vasilescu and D. Terzopoulos. "Multilinear Analysis of Image Ensembles: Tensorfaces". *European Conf. on Computer Vision*, 2002.

[WFFW17] R. Wang, B. Fu, G. Fu, and M. Wang. "Deep & Cross Network for Ad Click Predictions". *Proceedings of the ADKDD*, 2017.

[WWP09] S. Williams, A. Waterman, and D. Patterson. "Roofline: An Insightful Visual Performance Model for Multicore Architectures". *Commun. ACM*, 2009.

[WZS$^+$14] E. Wang, Q. Zhang, B. Shen, G. Zhang, X. Lu, Q. Wu, and Y. Wang. "Intel Math Kernel Library", 2014.

[YG17] L. Yavits and R. Ginosar. "Sparse Matrix Multiplication on CAM Based Accelerator". *arXiv preprint arXiv:1705.09937*, 2017.

[YL10] I. Yamazaki and X. S. Li. "On Techniques to Improve Robustness and Scalability of a Parallel Hybrid Linear Solver". *Int'l Conf. on High Performance Computing for Computational Science*, 2010.

[YZ04] R. Yuster and U. Zwick. "Detecting Short Directed Cycles Using Rectangular Matrix Multiplication and Dynamic Programming". *ACM-SIAM Symp. on Discrete Algorithms*, 2004.

[ZDG$^+$18] X. Zhou, Z. Du, Q. Guo, S. Liu, C. Liu, C. Wang, X. Zhou, L. Li, T. Chen, and Y. Chen. "Cambricon-S: Addressing Irregularity in Sparse Neural Networks through A Cooperative Software/Hardware Approach". *Int'l Symp. on Microarchitecture (MICRO)*, 2018.

[ZDZ$^+$16] S. Zhang, Z. Du, L. Zhang, H. Lan, S. Liu, L. Li, Q. Guo, T. Chen, and Y. Chen. "Cambricon-x: An Accelerator for Sparse Neural Networks". *Int'l Symp. on Microarchitecture (MICRO)*, 2016.

[ZGD$^+$18] Y. Zhou, U. Gupta, S. Dai, R. Zhao, N. Srivastava, H. Jin, J. Featherston, Y.-H. Lai, G. Liu, G. A. Velasquez, et al. "Rosetta: A Realistic High-Level Synthesis Benchmark Suite for Software Programmable FPGAs". *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2018.

[ZGS$^+$13] Q. Zhu, T. Graf, H. E. Sumbul, L. Pileggi, and F. Franchetti. "Accelerating Sparse Matrix-Matrix Multiplication with 3D-Stacked Logic-In-Memory Hardware". *IEEE High Performance Extreme Computing Conference (HPEC)*, 2013.

[ZP05] L. Zhuo and V. K. Prasanna. "Sparse Matrix-Vector Multiplication on FPGAs". *Int'l Symp. on Field-Programmable Gate Arrays (FPGA)*, 2005.

[ZZS$^+$18] G. Zhou, X. Zhu, C. Song, Y. Fan, H. Zhu, X. Ma, Y. Yan, J. Jin, H. Li, and K. Gai. "Deep Interest Network for Click-Through Rate Prediction". *Int'l Conference on Knowledge Discovery &#38; Data Mining*, 2018.

[ZZZ19] K. Zhang, X. Zhang, and Z. Zhang. "Tucker Tensor Decomposition on FPGA". *arXiv preprint arXiv:1907.01522*, 2019.