UC Merced
UC Merced Electronic Theses and Dissertations

Title
Tensor Computation Based on Heterogeneous Memory

Permalink
https://escholarship.org/uc/item/3d5162n8

Author
Liu, Jiawen

Publication Date
2022

Copyright Information
This work is made available under the terms of a Creative Commons Attribution License, availalbe at https://creativecommons.org/licenses/by/4.0/

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA, MERCED

Tensor Computation Based on Heterogeneous Memory

A dissertation submitted in partial satisfaction of the
requirements for the degree
Doctor of Philosophy

in

Electrical Engineering and Computer Science

by

Jiawen Liu

Committee in charge:

Dong Li, University of California Merced, Chair
Florin Rusu, University of California Merced
Hyeran Jeon, University of California Merced
Jiajia Li, College of William & Mary

Spring 2022

Copyright

Jiawen Liu, Spring 2022

All rights reserved.

The dissertation of Jiawen Liu is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Professor Dong Li, Chair

Professor Florin Rusu

Professor Hyeran Jeon

Professor Jiajia Li

University of California, Merced

Spring 2022

ABSTRACT OF THE DISSERTATION

Tensor Computation Based on Heterogeneous Memory

by

Jiawen Liu
Doctor of Philosophy in Electrical Engineering and Computer Science

University of California Merced, Spring 2022

Dong Li, University of California Merced, Chair

Tensors, which generalize matrices to more than two dimensions, are fundamental to many disciplines, such as scientific computing and machine learning. Improving the performance and scalability of tensor computation is essential to those domains. The recent advance of heterogeneous memory is promising to deliver large-scale, high-performance tensor computation. However, it is challenging to leverage memory heterogeneity because of performance disparity between memory components. Tensor computation, often characterized with irregular memory access patterns, large working set size, and unknown tensor dimension size, makes the usage of heterogeneous memory more challenging.

In this dissertation, we propose efficient and scalable heterogeneous memory systems for tensor computation to solve the challenges. The core innovation in our proposed systems is to introduce system-architecture-tensor co-designs, taking advantage of intersectional domain knowledge in runtime system policies, architecture characteristics, and tensor features. In particular, our approach takes into account runtime system policies (e.g., policies of data migration, prefetching, concurrency control), architecture characteristics (e.g., characteristics in emerging non-volatile memories, 3D-stacked memories, accelerators with massive parallelism), and tensor features (e.g., high data dimensionality, varying memory access patterns, irregular data distribution in the data structure) for tensor computation.

The evaluation results show that: (1) with evaluating various sparse tensor contraction datasets, our design brings 28−576 times speedup over the state-of-the-art sparse tensor contraction design; (2) with evaluating various sparse tensor contraction

sequence datasets, our design brings 327-7362 times speedup over the state-of-the-art work; (3) with evaluating various tensor-based neural network training workloads, our design achieves up to 24 times and 4 times better energy consumption compared to CPU and GPU respectively; (4) with evaluating various tensor-based neural network training workloads, our design achieves up to 50% (33% on average) performance improvement compared to the state-of-the-art TensorFlow.

ACKNOWLEDGEMENTS

First and foremost, I sincerely thank my advisor, Prof. Dong Li. Prof. Dong Li has been an exceptional advisor, and I have been very fortunate to receive his guidance for the five years of my Ph.D. journey. In retrospect, I learned from Prof. Dong Li how to define a problem in year one and two, solve this problem in year three and four, and spread the discovery in year five. In each step, Prof. Dong Li gave me extremely visionary advice, most generous support, and most sincere and constructive feedback. Prof. Dong Li's expertise and experience made his advice very insightful, and his enthusiastic of impactful research greatly motivated me. Prof. Dong Li's research foresight, technical depth, and commitment to the students is a valuable treasure for me. I had the unique privilege to have access to Prof. Dong Li's professional expertise and brilliant thinking. Prof. Dong Li offered me invaluable advice, diligently guided me through challenging problems, and taught me to perceive the philosophy. I feel very fortunate to have Prof. Dong Li be my advisor.

It has been an honor to work with many great collaborators outside UC Merced. I sincerely thank Prof. Jiajia Li. She was my mentor during my internship at Pacific Northwest National Laboratory (PNNL). Her ambition and foresight ignited my passion for pursuing the research in tensor computation. I sincerely thank Prof. Jishen Zhao at University of California, San Diego, and Prof. Dimitrios Nikolopoulos at Virginia Tech for their insightful discussions and priceless support for my research. I would also like to thank Dr. Gokcen Kestor at PNNL, Dr. Adnan Aziz at Meta, and Qingqing Zhou at Tencent America for mentoring me during my internships for valuable advise and guidance for my research.

I give sincere thanks to my mom and dad. I can't forget the encouragements I get from you when I came to US thousands of miles away from home, and I can't accomplish what I did without your love. Thank you for nurturing me and set me a great role model.

Finally, I would like to thank my wife, for your endless love and support. Thank you for being my best friend, my partner, and my compass when I got lost along my PhD journey. Thank you for studying with me, cooking with me, staying with me, and even discussing new research ideas with me. I am so fortune to have you with me, and am looking forward to our future adventures together.

TABLE OF CONTENTS

	Signature Page	iii
	Abstract	iv
	Acknowledgements	vi
	Table of Contents	vii
	List of Figures	x
	List of Tables	1
Chapter 1	Introduction	2
	1.1 Contributions	4
Chapter 2	Background	6
	2.1 Sparse Tensors	6
	2.2 Sparse Tensor Contraction	7
	2.3 Neural Network Training	8
	2.4 Dataflow-Based Machine Learning Framework	9
	2.5 Feasibility of Heterogeneous PIM Architecture	9
	2.6 Intel Optane DC Persistent Memory Module	9
Chapter 3	Sparta: Efficient and Parallel Sparse Tensor Contraction on Heterogeneous Memory Systems	11
	3.1 Motivation	11
	3.2 Design	14
	3.2.1 Sparse Tensor Contraction Algorithm	14
	3.2.2 Data Placement on Persistent Memory-based Heterogeneous Memory Systems	21
	3.3 Evaluation	26
	3.3.1 Evaluation Setup	26
	3.3.2 Overall Performance	27
	3.3.3 Performance Comparison to ITensor	29
	3.3.4 Thread Scalability	29
	3.3.5 Sparta on Heterogeneous Memory Systems	29
	3.4 Related Work	31
	3.5 Summary	32
Chapter 4	Athena: High-Performance Sparse Tensor Contraction Sequences on Heterogeneous Memory	33
	4.1 Motivation	33
	4.2 Design	37
	4.2.1 Algorithm Design	37

4.2.2 Data Management on PMM-based Heterogeneous Memory Systems 43
4.3 Evaluation 47
4.3.1 Evaluation Setup 47
4.3.2 Overall Performance 48
4.3.3 Optimization Analysis 49
4.3.4 Performance Comparison to ITensor 53
4.3.5 Application in Chemistry 53
4.4 Related Work 54
4.5 Summary 55

Chapter 5 Processing-in-Memory for Energy-efficient Neural Network Training: A Heterogeneous Approach 56
5.1 Motivation 56
5.1.1 NN Training Characterization 56
5.1.2 Software Design Challenges and Opportunities . . . 58
5.1.3 CPU vs. GPU – Where to Attach Heterogeneous PIMs? 60
5.2 Design 60
5.2.1 Heterogeneous PIM Architecture 61
5.2.2 Programming Model for Heterogeneous PIM 61
5.2.3 Runtime System Design 67
5.2.4 Implementation 69
5.3 Experimental Setup 72
5.3.1 Simulation Framework 72
5.3.2 Power and Area Modeling 73
5.3.3 Workloads 73
5.3.4 Real Machine Configurations 73
5.4 Evaluation 74
5.4.1 Execution Time Analysis 74
5.4.2 Energy Consumption Analysis 76
5.4.3 Comparison with Prior PIM-based NN Acceleration 76
5.4.4 Sensitivity Study 77
5.4.5 Evaluation of Software Impact 77
5.4.6 Mixed Workloads Analysis 79
5.4.7 Energy Efficiency Analysis 80
5.5 Related Work 81
5.5.1 Processing-in-memory for Machine Learning 81
5.5.2 Processing-in-memory for General Applications . . 82
5.5.3 Other Accelerator Optimization for Machine Learning. 82
5.6 Summary 83

Chapter 6 Runtime Concurrency Control and Operation Scheduling for High Performance Neural Network Training 84
6.1 Motivation 84
6.1.1 Performance Variance with Different Concurrency . 85
6.1.2 Impact of Input Data Size 86
6.1.3 Co-Running Operations 87
6.2 Design 88
6.2.1 Overview 88
6.2.2 Regression Model-Based Performance Model 90
6.2.3 Hill Climbing Algorithm-Based Performance Model 93
6.2.4 Runtime Scheduling 95
6.3 Experiment Setup 98
6.3.1 Training Models, Data Set and Framework 98
6.3.2 Hardware Platform 99
6.3.3 Controlling Intra-op Parallelism 99
6.4 Evaluation 99
6.4.1 Applying Concurrency Control for Individual Operations 100
6.4.2 Applying Operations Co-running 101
6.4.3 Applying Hyper-threading 101
6.4.4 Putting all together. 103
6.4.5 Comparing with the manual optimization. 104
6.5 Related Work 104
6.5.1 Performance Optimization for Dataflow-based Frameworks 104
6.5.2 Thread Concurrency Throttling 105
6.6 Summary 105

Chapter 7 Conclusion 106

Bibliography 109

LIST OF FIGURES

Figure 3.1: Workflow of the traditional SpTC-SPA and Sparta on $\mathcal{Z} = \mathcal{X} \times^{\{1,2\}}_{\{3,4\}} \mathcal{Y}$. 14
Figure 3.2: Percentage of execution time breakdown of *SpTC-SPA* (Algorithm 1). 19
Figure 3.3: Performance after placing a data object in PMM while leaving others in DRAM. The x axis shows the data object placed in PMM. "All in DRAM" means all data objects are placed in DRAM. . . . 23
Figure 3.4: Speedups of HtY+HtA (i.e., Sparta) and COOY+HtA over COOY+SPA (i.e., SpTC-SPA) for SpTCs on Chicago, NIPS, Uber, Vast and Uracil with 1-mode, 2-mode and 3-mode. 27
Figure 3.5: Speedups of Sparta over ITensor on Hubbard-2D model using different SpTC expression with different sparse input tensors. . . 28
Figure 3.6: Thread scalability of parallel Sparta on SpTCs on NIPS with 1-mode, Vast with 2-mode and NIPS with 3-mode. 28
Figure 3.7: Speedups of Sparta, IAL, Memory mode and Dram-only over Optane-only for SpTCs on Chicago*, NIPS*, Vast*, Flickr, Delicious and Nell-2 with 1-mode, 2-mode and 3-mode. 28
Figure 3.8: Memory bandwidth of Sparta, IAL, PMM Memory mode and Optane-only on Vast with 1-mode SpTC. 30
Figure 3.9: Peak memory consumption of SpTCs on Chicago*, NIPS*, Vast*, Flickr, Delicious and Nell-2 with 1-mode, 2-mode and 3-mode. . . 30

Figure 4.1: Workflow of Type 1 dependency of two SpTCs in Table 2.2, using shared hash table-represented sparse accumulator and hash table-represented sparse tensor summation (indicated by red arrows) 39
Figure 4.2: Workflow of the dynamic data placement based on data semantics. 42
Figure 4.3: Overall speedups of Athena over Sparta for SpTCSeq on 12 tensors. 48
Figure 4.4: Percentage of execution time breakdown of Athena. 49
Figure 4.5: Peak memory consumption of SpTCSeq on 12 tensors. 49
Figure 4.6: Speedups of Athena with hash-table represented summation over Sparta with traditional linear search-based summation. 50
Figure 4.7: "Stage Parallelism" and "Shared-HtA" optimization speedup over the "Sparta + Summation" as the baseline. 51
Figure 4.8: Speedups of Athena, IAL, Memory Mode and DRAM-only over PMM-only for SpTCSeq. 52
Figure 4.9: Speedups of Athena over ITensor on Hubbard-1D-T, Hubbard-1D-P, Hubbard-1D-Z and Hubbard-2D models using different SpTCSeq with different sparse input tensors. 53

Figure 5.1: Our profiling framework for profiling NN training workloads in TensorFlow. 57
Figure 5.2: Four categories of NN training operations. 57
Figure 5.3: Architecture overview of the proposed heterogeneous PIM. . . . 62

Figure 5.4: The process of executing NN training with our software framework design. . . . 63
Figure 5.5: Enabling OpenCL platform model on heterogeneous PIM systems. 64
Figure 5.6: An example of the recursive PIM kernel. . . . 65
Figure 5.7: Heterogeneous PIM implementation. . . . 71
Figure 5.8: Execution time breakdown of five NN models. . . . 72
Figure 5.9: Normalized dynamic energy of various NN models. . . . 74
Figure 5.10: Performance and energy comparison with Neurocube. . . . 75
Figure 5.11: Execution time breakdown of various NN models with 3D memory frequency scaling. . . . 76
Figure 5.12: Execution time with Progr PIM scaling. . . . 78
Figure 5.13: Execution time with and without RC and OP. . . . 79
Figure 5.14: Dynamic energy with and without RC and OP. . . . 79
Figure 5.15: Hardware utilization with and without RC and OP. . . . 79
Figure 5.16: Execution time of multiple NN training models with our design and sequential execution, respectively. . . . 79
Figure 5.17: Energy efficiency and power with 3D memory frequency scaling. 80

Figure 6.1: Performance variance of three operations with different intra-op parallelisms. The reported execution time is the total execution time of one thousand runs. . . . 85
Figure 6.2: Our runtime framework and its workflow. The notation "TS" is the total number of training steps. . . . 90
Figure 6.3: Quantifying the contribution of the four strategies. Comparing the performance of our runtime, manual optimization, and the recommendation by TensorFlow. . . . 100
Figure 6.4: The variance of the number of co-running operations along with the NN model training. The figures (a), (b) and (c) do not have Strategy 4 (but have Strategy 3); The figures (d), (e) and (f) have Strategy 4 in place. The red lines in the figures are the inter-op parallelism recommended by TensorFlow. . . . 103

LIST OF TABLES

Table 2.1: List of symbols and notation. 6
Table 2.2: Expression dependency between two SpTCs. 7

Table 3.1: Memory access patterns associated with data objects in six stages ("Ran" = Random; "Seq" = Sequential; "RW" = Read-Write; "RO" = Read-Only; "WO" = Write-Only). (Note that for Ht$\mathfrak{Z}$ in Summation, we employ temporal locality to always maintain the most frequently used bucket in DRAM. The memory access pattern of key-value nodes within the bucket is still random. For $\mathfrak{Z}_{input}$, the first two passes are "Seq, RO" and the last pass is "Ran, RO".) . . 23
Table 3.2: Characteristics of sparse tensors in the evaluation. 26

Table 4.1: Characteristics of sparse tensors in the evaluation 47
Table 4.2: A 10-SpTC sequence from a CCSD(T) model. 52

Table 5.1: Operation profiling results for three neural network models. "CI"= computation intensive; "MI"=memory intensive. 58
Table 5.2: Extending OpenCL for the heterogeneous PIM. 64
Table 5.3: Low-level APIs for PIMs. 68
Table 5.4: System configurations. 72

Table 6.1: Study the performance of NN models with different inter-op and intra-op parallelisms. The performance baseline for calculating speedup is the performance with the configuration recommended by the TensorFlow programming guide (68 threads for intra-op parallelism and 1 for inter-op parallelism). 86
Table 6.2: Study the impact of input data size on operation performance. The performance baseline for calculating performance variance is the performance with using 68 threads. The reported time is the total execution time of one thousand runs. 88
Table 6.3: Co-running two operations with three strategies. The performance baseline for calculating speedup is performance of serial execution of two operations. The reported time is the total execution time of one thousand runs. 88
Table 6.4: Prediction accuracy of a set of regression models. 92
Table 6.5: Performance prediction accuracy for four NN models based on the hill climbing-based performance model. 94
Table 6.6: Performance improvement of the top five most time-consuming operations in four NN models by recommendation and by applying Strategies 1 and 2. The performance baseline for calculating speedup is the performance with the configuration recommended by the TensorFlow programming guide (68 threads for intra-op parallelism and 1 for inter-op parallelism). 102

Chapter 1

Introduction

The tensor is a mathematical object that generalizes matrices. A tensor can have any dimension, and is ubiquitous in mathematics and sciences. The tensor is widely used in machine learning [1, 2, 3], data mining [4], social networks analytics [5, 6], signal processing [7], healthcare analytics [8, 9], and so on. Thus, improving the performance and efficiency of tensor computation is essential in many fields and realistic applications. Compared with computation on general data objects, tensor computation is more challenging due to the following reasons.

First, it is challenging to handle multi-index searches efficiently in tensor computation. The challenge includes performance and scalability issues associated with high data dimensionality, large memory consumption, and resource contention on cache hierarchy and memory bandwidth. For example, a sparse tensor contraction (SpTC) has indirect memory accesses to the second input tensor, caused by the non-zero indices of the first input tensor. The indirect memory accesses of the second input tensor and the sparse accumulator, which happen more often with the high-dimensional tensors, are not cache friendly.

Second, tensor computation can easily cause performance issue due to redundant computation and memory operations. For example, in the type that two SpTCs share an identical input tensor, the processing on this input in the second SpTC can be avoided. Because of the shared data objects, computation and memory access on intermediate data objects are also performed repeatedly. For example, in the type that the output tensor of the first SpTC becomes the input tensor in the second SpTC, the intermediate results in the accumulation of the first SpTC can be directly reused

to do the computation of the second SpTC, skipping multiple stages in the sequential execution. This performance issue becomes severer when the redundant computation and memory access dominate the execution. Moreover, the performance suffers when we perform an SpTC sequence with a larger number of contractions.

Third, tensor computation can involve large data sets and immense computation, which can cause frequent and expensive data movement across the memory hierarchy. For example, neural network (NN) training based on TensorFlow or Pytorch is implemented using tensors and tensor computation. NN training easily consumes hundreds of GB memory and billions of tensors [10]. Training NN models (especially those large natural-language-processing models, such as GTP-2 [11] and Megatron-LM [12]) can take tens of hours using tens of NVIDIA V100 GPU [13]. Training those models involves massive amount of tensor operations [14]. This leads to frequent data movement across the memory hierarchy. As NN models become deeper and larger, we expect that tensor computation in NN training will involve even larger data sets and more computation.

Fourth, thread-level concurrency control is challenging in tensor computation, because of diverse memory access patterns in tensors operations. We must efficiently manage thread-level concurrency for tensor operations to achieve efficient parallel execution. Some tensor operations do not have good scalability, because of caching effects and thread spawning overhead. Using a large number of threads on a many-core machine to run those tensor operations does not necessarily result in the best performance [15]. The problem of deciding thread-level concurrency for each tensor operation is coupled with the thread affinity problem (i.e., deciding the binding between threads and cores) [16], which makes this concurrency management problem even more challenging in tensor operations with diverse memory access patterns. Furthermore, we must decide how to co-run tensor operations. When tensor operations do not have unresolved dependency and each individual operation does not sufficiently utilize hardware resources (e.g., physical cores), co-running operations may improve memory bandwidth and thus improve performance. However, currently, there is no systematic approach to efficiently control tensor operations concurrency and schedule those large amounts of tensor operations with considering diverse memory access patterns. The existing runtime system frameworks simply use the same thread-level parallelism for all operations and schedule operations simply according to operation dependency [17].

In this dissertation, we aim to develop memory-oriented systems that overcome the above challenges for high-performance tensor computation.

1.1 Contributions

We briefly summarize the main contributions of this proposal as follows.

- We introduce Sparta, a high-performance sparse tensor contraction (SpTC) system for arbitrary-order element-wise SpTC by using multi-dimensional, efficient hash table representation for the accumulator and larger input tensor, and all-stage parallelization. Furthermore, we explore the emerging Optane-based heterogeneous memory to address memory capacity limitation suffered in the traditional tensor computation, and show that static data placement on heterogeneous memory can outperform dynamic data migration.

- We present Athena, a high-performance system for SpTC sequences. Athena introduces new data structures, leverages emerging Optane-based heterogeneous memory architecture, and adopts stage parallelism. In particular, Athena introduces shared hash table-represented sparse accumulator to eliminate unnecessary input processing and data migration; Athena uses a novel data-semantic guided dynamic migration solution to make the best use of the Optane-based heterogeneous memory for high performance; Athena also co-runs execution phases with different characteristics to enable high hardware utilization.

- We propose a heterogeneous processing-in-memory (PIM) architecture to reduce memory copy overhead between the host processor and the main memory. We further propose a programming model by extending OpenCL programming model in order to support effective and efficient memory synchronization between main memory and PIM. Moreover, we build a runtime system to dynamically schedule and pipeline tensor operations, based on profiled memory characteristics of tensor operations.

- We provide a novel and robust performance prediction model to predict performance of tensor operations by capturing diverse memory access patterns of tensor operations and cache sharing between threads. Moreover, we propose a

set of domain-specific operation scheduling policies to reduce data movement overhead with the consideration of non-uniform memory access.

Chapter 2

Background

2.1 Sparse Tensors

A tensor can be regarded as a multidimensional array. Each of its dimensions is called a *mode*, and the number of dimensions or modes is its *order*. For example, a matrix of order 2 means it has two modes (rows and columns). We represent tensors with calligraphic capital letters, e.g., $\mathcal{X} \in \mathbb{R}^{I \times J \times K \times L}$ (a tensor with four modes), and x_{ijkl} is its (i, j, k, l)-element. Table 2.1 summarizes notation and symbols for tensors.

Table 2.1: List of symbols and notation.

Symbols	Description
$\mathcal{X}, \mathcal{Y}, \mathcal{Z}$	Sparse tensors
$\mathcal{Z} = \mathcal{X} \times_{\{n\}}^{\{m\}} \mathcal{Y}$	Tensor contraction between two tensors
N_X	Tensor order of $\mathcal{X}$
I, J, K, L, I_n	Tensor mode sizes
nnz_X	#Non-zeros of the input tensor $\mathcal{X}$
N_F	#Mode-F^X sub-tensors of $\mathcal{X}$
nnz_F	The #Non-zeros of sub-tensors of $\mathcal{X}$
ptr_F	Pointers for mode-F^X sub-tensor locations of $\mathcal{X}$
C^X	A set of contract modes in $\mathcal{X}$, $\{n\}$ in $\times_{\{n\}}^{\{m\}}$ contraction
F^X	A set of free modes in $\mathcal{X}$, $\lvert F^X \rvert + \lvert C^X \rvert = N_X$
C_{nz}^X	Contract mode indices of a non-zero element in $\mathcal{X}$
F_{nz}^X	Free mode indices of a non-zero element in $\mathcal{X}$
val^X	A set of non-zero values in $\mathcal{X}$
val_{nz}^X	Value of a non-zero element in $\mathcal{X}$

Table 2.2: Expression dependency between two SpTCs.

Type	Feature	Expressions
1	Output as input	$\mathcal{Z}' = \mathcal{X} \times \mathcal{Y}$ and $\mathcal{Z}$ += $\mathcal{W} \times \mathcal{Z}'$
2	Identical input & Different output	$\mathcal{Z}$ += $\mathcal{X} \times \mathcal{Y}$ and $\mathcal{Z}'$ += $\mathcal{W} \times \mathcal{Y}$
3	Different input & Shared output	$\mathcal{Z}$ += $\mathcal{X} \times \mathcal{Y}$ and $\mathcal{Z}$ += $\mathcal{W} \times \mathcal{V}$
4	Identical input & Shared output	$\mathcal{Z}$ += $\mathcal{X} \times \mathcal{Y}$ and $\mathcal{Z}$ += $\mathcal{W} \times \mathcal{Y}$
5	Independent	$\mathcal{Z}$ += $\mathcal{X} \times \mathcal{Y}$ and $\mathcal{Z}'$ += $\mathcal{W} \times \mathcal{V}$

2.2 Sparse Tensor Contraction

Tensor contraction, a.k.a. Tensor-Times-Tensor (TTT) or mode-($\{n\}, \{m\}$) product [18], is an extension of matrix multiplication, denoted by

$$\mathcal{Z} = \mathcal{X} \times_{\{n\}}^{\{m\}} \mathcal{Y}, \tag{2.1}$$

where $\{n\}$ and $\{m\}$ are tensor modes to do contraction.

Example: $\mathcal{Z} = \mathcal{X} \times_{\{3,4\}}^{\{1,2\}} \mathcal{Y}$**.** This contraction operates on I_3 and I_4 in $\mathcal{X}$ and J_1 and J_2 in $\mathcal{Y}$ ($I_3 = J_1$) and ($I_4 = J_2$). All of the four modes are contract modes (annotated with $C_X = \{3, 4\}$ and $C_Y = \{1, 2\}$), and the other modes are free modes. This example's operation is formally defined as:

$$z_{i_1 i_2 j_3 j_4} = \sum_{i_3(j_1)=1}^{I_3(J_1)} \sum_{i_4(j_2)=1}^{I_4(J_2)} x_{i_1 i_2 i_3 i_4} x_{j_1 j_2 j_3 j_4}. \tag{2.2}$$

The number of modes of the output $\mathcal{Z}$, $N_Z = |F_X| + |F_Y| = (N_X - |C_X|) + (N_Y - |C_Y|)$. This is our walk-through example in the following discussion.

Element-wise Sparse Tensor Contraction. Element-wise sparse tensor contractions (SpTC) emerges in the applications of chemistry and physics [19, 20, 21, 22, 23, 24], where both input and output tensors have element-wise sparsity. Sparta [25] is the state-of-the-art algorithm for an arbitrary-order, element-wise SpTC. Sparta introduces a hash table-based representation for input sparse tensors and a sparse accumulator for a single element-wise sparse tensor contraction. The Sparta SpTC algorithm contains five stages: input processing, index search, accumulation, writeback, output sorting stages. Refer to [25] for more details.

Sparse Tensor Contraction Sequences Sparse tensor contractions sequence (SpTCSeq) is widely used in many methods. For example, SpTCSeq can be derived

from the well-known Coupled Cluster Single Double (Triple), CCSD(T) [26], in chemistry [24] and from the notable Hubbard model [27] in physics [19]. Within an SpTCSeq, multiple SpTCs could have dependence between each other or be independent, and they might share some identical tensors in different ways.

We summarize common expression types of SpTCSeq in Table 2.2. Five types could exist for two arbitrary SpTCs. In Type 1, the output tensor of the first SpTC, $\mathfrak{Z}'$, used as an input tensor of the second SpTC; In Type 2, both SpTCs have an identical input tensor $\mathfrak{Y}$; In Type 3, the two SpTCs use different input tensors but generate the same output tensor $\mathfrak{Z}$; In Type 4, both SpTCs use an identical input tensor $\mathfrak{Y}$ and share the output tensor $\mathfrak{Z}$; In Type 5, the two SpTCs are totally independent from each other. We quantify the occurrence percentage of the five types of SpTCSeq in CCSD(T) [26] from chemistry. Types 1-5 account for 85%, 9%, 6%, 8% and 91% of all SpTCSeq, respectively. Note that the sum of all types is more than 100%, because an SpTC equation could fall into more than one type. Besides, sparse tensor summation, the "+" operator in most expressions of Table 2.2, is common in sparse tensor contractions sequences (e.g., accounts for 90% in CCSD(T) [26] in chemistry).

2.3 Neural Network Training

NN training could be expensive, because it is an iterative process involving large training data sets. Many NN take hours or even days for training, even on the state-of-the-art GPU [28]. Although using GPU to train neural network is common, using multi/many-core processors (e.g., Intel Knights Landing) to train neural network is also becoming common [29, 30, 31, 32].

Training an NN often involves a large number of iterative steps (thousands and even millions of steps). In each step, a batch of samples is fed into the NN. Except the first step which is often used for performance profiling to determine appropriate data layout [33], initialize data based on device configuration, and estimate performance by a cost model empirically or analytically [34], all other steps have the same computation and memory access patterns. Performance of each step (particularly execution time and the number of main memory accesses) remains stable across steps. The above characteristics allow us to build performance models based on dynamic profiling of the first few steps and use the profiling results to improve performance of the following steps.

Note that the word "performance" in this paper refers to the execution time, not modeling accuracy of NN.

2.4 Dataflow-Based Machine Learning Framework

The state-of-the-art ML frameworks, such as TensorFlow, Caffe2 and MxNet, decompose an ML model into fine-grained operations. Similar to task-based parallel programming models [35] such operation-based ML frameworks greatly improve hardware utilization and system throughput [34]. Within a training step of NN training, there can be tens of different operations, and each operation can be invoked hundreds of times, each of which is an *operation instance*. Different instances of an operation can have different input data sizes.

TensorFlow allows users to control operation concurrency. The operation concurrency includes inter-op parallelism and intra-op parallelism. However, such control of operation concurrency has to be manually decided by the user. Furthermore, the intra-op parallelism is enforced uniformly on all operations, ignoring the scalability difference between operations.

2.5 Feasibility of Heterogeneous PIM Architecture

The logic layer of 3D memory stacks has area, power, and thermal limitations. But previous studies demonstrated the feasibility of adopting both fixed-function and programmable PIMs, while meeting these constraints [36]. We adopt similar methodologies to ensure the feasibility of our architecture implementation (Chapter 5.2.4).

2.6 Intel Optane DC Persistent Memory Module

The recent release of the Intel Optane DC Persistent Memory Module (PMM) is the first byte-addressed non-volatile memory (NVM) in the market. PMM can be configured to work in either *Memory* or *AppDirect* mode. In the Memory mode, DRAM is a hardware-managed, directly-mapped write-back cache to PMM and is transparent to applications. In the AppDirect mode, the placement of data objects on PMM and DRAM can be explicitly controlled by the programmers.

PMM can provide up to 6TB memory capacity on a single machine, but has 2.2-3.5$\times$ higher access latency and 2.7-6.2 $\times$ lower bandwidth than the traditional DRAM. Sparta [25] statically allocates data objects to either DRAM or PMM according to

their memory access patterns. However, this simple static data placement does not work best for all data objects, especially data objects with random read/write memory access. Our work leverages the AppDirect mode with dynamic data management and leads to better performance than the Memory mode.

Chapter 3

Sparta: Efficient and Parallel Sparse Tensor Contraction on Heterogeneous Memory Systems

3.1 Motivation

Tensors, especially those high-dimensional sparse tensors are attracting increasing attentions, because of their popularity in many applications. High-order sparse tensors have been studied well in tensor decomposition on various hardware platforms [37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49] with a focus on the product of a sparse tensor and a dense matrix or vector.

Nevertheless, the two sparse tensor contraction (SpTC), foundation for a spectrum of applications, such as quantum chemistry, quantum physics and deep learning [19, 20, 21, 22, 23, 24], are still lack of sufficient research, especially with element-/pair-wise sparsity. In essence, SpTC, a high-order extension of sparse matrix-matrix multiplication (SpGEMM), multiplies two sparse tensors along with their common dimensions. Efficient SpTC introduces multiple challenges.

First, the size and non-zero pattern of the output tensor are unknown before computation. Thus, memory allocation for the output tensor is difficult. Unlike operations such as a sparse tensor multiplies a dense matrix/vector where the size of the output data is predictable, the output tensor of an SpTC is usually sparse and the non-zero pattern (e.g., the number of non-zero elements and their distribution) is

unpredictable before the actual computation. Sparse data objects and unpredictable output size also exist in SpGEMM. Two popular approaches have been proposed to solve these issues for SpGEMM while are not efficient for SpTC. The first approach, using an extra symbolic phase [50] to predict the accurate output size and non-zero pattern, suffers from expensive pre-processing and is unaffordable in a dynamic sparsity environment. This issue is especially severe in SpTC, because an SpTC with the exactly same input is usually computed only once in a long sequence of tensor contractions [24]. However, with the symbolic approach, every SpTC is attached to both a symbolic phase and SpTC computation, which is very expensive, especially for large applications. The second approach makes a loose upper-bound prediction on the memory consumption of the output tensor. However, a tight prediction for SpTC of high-order tensors is very difficult because its more contract dimensions make the prediction less accurate based on the existed prediction algorithms [51, 52].

Second, irregular memory accesses along with multi-dimensional index search to the second input tensor and accumulator introduce performance problems. Similar to SpGEMM, SpTC has indirect memory accesses to the second input tensor, caused by the non-zero indices of the first input tensor. Take an SpGEMM $\mathbf{C} = \mathbf{A} \times \mathbf{B}$ as an example. A non-zero $\mathbf{A}(0, 1)$ gets, e.g. $\mathbf{B}(1, 1)$, to perform multiplication; while $\mathbf{A}(0, 10)$ computes with, e.g. $\mathbf{B}(10, 2)$. Those irregular memory accesses of $\mathbf{B}$ and the sparse accumulator, which happen more often with the high-dimensional tensors, are not cache friendly. In addition, index search and accumulator, which is used to address irregular memory accesses in SpTC, is more expensive than that in SpGEMM. Our evaluation shows that they takes 54% of SpTC performance on average.

Third, massive memory consumption caused by large input and output tensors and intermediate results creates pressure on the traditional DRAM-based machine. Sparse tensors from real-world applications easily consume a few to dozens of GB memory, while the output tensor could be even larger, because it contains more non-zero elements than any of the input sparse tensor. The intermediate results could be large as well, especially for multi-threading environment where each thread has its own intermediate results. Compared to the well-studied sparse tensor times dense matrices/vectors [43, 45, 46, 37], SpTC results in substantial memory consumption easily, which can be beyond typical DRAM capacity (up to a few hundreds of GB) on a single machine. However, expanding DRAM capacity is not cost effective, while

adding cheap but much slower SSD causes significant performance drop. This memory capacity problem is becoming more serious in those HPC applications with increasing dimension size in tensors [24, 19, 53, 54, 55, 56, 18].

To address the first two challenges, we propose Sparta (Algorithm 2) with performance optimizations executed in five stages: input processing, index search, accumulation, writeback, and output sorting. In particular, we employ dynamic arrays to accurately allocate memory space for the accumulator and output tensor to avoid the unknown output challenge. For multi-threading environment, we introduce a thread-private, dynamic object to store the output tensor from each thread for better parallelization. To address the irregular memory access challenge, we perform permutation and sorting on input sparse tensors before computation, thus significantly improve temporary locality of non-zeros in the first input tensor and spacial locality of non-zeros in the second input tensor. Furthermore, we adopt hash table-based approaches based on a large-number representation for the second tensor and accumulator to significantly speed up the process of multi-dimensional search in SpTC. With the above optimizations, Sparta substantially outperforms the traditional SpTC algorithm extended from SpGEMM. By evaluating real data from quantum chemistry and physics, our element-wise Sparta beats their block-sparse algorithms by 7.1× on average.

To address the third challenge, we explore the emerging persistent memory-based heterogeneous memory (HM). In particular, recent Intel Optane DC Persistent Memory Module (PMM) provides bandwidth and latency only slightly inferior to that of DRAM but with only half of the price. PMM often pairs with a small DRAM to build HM, where frequently accessed data objects placed in DRAM and the rest resided in PMM with several TBs of large memory capacity. It is performance-critical to decide the placement of data objects of SpTC (input and output tensors and intermediate results) on PMM-based HM, to make best use of DRAM's high bandwidth and low latency without causing frequent data movement between PMM and DRAM. We first characterize memory read/write patterns associated with those data objects in SpTC, and reveal the performance sensitivity of SpTC to the placement of those data objects on PMM and DRAM. Sparta then prioritizes the data placement between DRAM and PMM statically based on our knowledge on the SpTC algorithm and characterization of data objects for best performance. Sparta effectively avoids

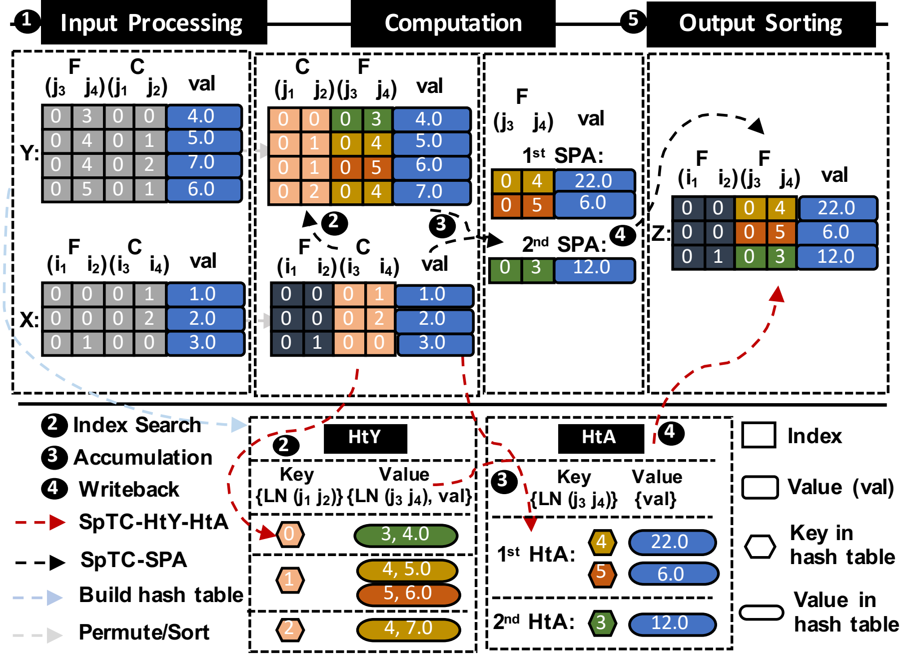


Figure 3.1: Workflow of the traditional SpTC-SPA and Sparta on $\mathcal{Z} = \mathcal{X} \times_{\{3,4\}}^{\{1,2\}} \mathcal{Y}$.

unnecessary data movement suffered in the traditional application-agnostic solutions (such as hardware-managed DRAM caching [57, 58, 59] or software-based page hotness tracking [60, 61, 62, 63, 64, 65, 66, 67]).

3.2 Design

3.2.1 Sparse Tensor Contraction Algorithm

This section introduces our SpTC algorithms, *SpTC-SPA* and Sparta, to address the challenges of unknown output and irregular memory accesses along with multi-dimensional index search.

Overview

Figure 3.1 depicts the workflow of our SpTC algorithm. Our algorithm has five stages: ❶ input processing, ❷ index search, ❸ accumulation, ❹ writeback, and ❺ output sorting, where ❶ and ❺ are called *input/output processing* collectively, and ❷, ❸ and ❹ are *computation* collectively. We describe the *input/output processing* stages in this section and the *computation* stages will be illustrated in Lines 13 and 19.

Input processing ❶. Figure 3.1 uses two tiny sparse tensors $\mathcal{X}$ and $\mathcal{Y}$ as input examples. When the modes of $\mathcal{X}$ or $\mathcal{Y}$ are not in the "correct mode order", permutation

Algorithm 1: *SpTC-SPA*: Sparse tensor contraction of *Example 2:* $\mathcal{Z} = \mathcal{X} \times_{\{3,4\}}^{\{1,2\}} \mathcal{Y}$, extended from SpGEMM [68] with sparse accumulator (SPA)

Input: Input tensors $\mathcal{X} \in \mathbb{R}^{I_1 \times I_2 \times I_3 \times I_4}$ and $\mathcal{Y} \in \mathbb{R}^{J_1 \times J_2 \times J_3 \times J_4}$, contract modes $C_X = \{3, 4\}$, $C_Y = \{1, 2\}$

Output: The output tensor $\mathcal{Z} \in \mathbb{R}^{I_1 \times I_2 \times J_3 \times J_4}$

```
1  Permute and sort X, Y if not yet;
2  for X(i1, i2, :, :) in X do
3  |  Initiate a sparse accumulator SPA
4  |  for Non-zero x(i1, i2, i3, i4) in X(i1, i2, :, :) do
5  |  |  for Non-zero y(i3, i4, j3, j4) in Y(i3, i4, :, :) do
6  |  |  |  v = x(i1, i2, i3, i4) × y(i3, i4, j3, j4)
7  |  |  |  if SPA(j3, j4) exists then
8  |  |  |  |  Accumulate SPA(j3, j4)+ = v
9  |  |  |  else
10 |  |  |  |  Append v to SPA
11 |  Write SPA back to Z(i1, i2, :, :)
12 Permute and sort Z as needed
13 return Z
```

and sorting are needed. "Correct mode order" means: The contract modes C_X ((i_3, i_4) in Figure 3.1) are the rightmost modes of $\mathcal{X}$ and C_Y ((j_1, j_2)) are the leftmost modes of $\mathcal{Y}$. $\mathcal{X}$ is first permuted to the "correct mode order" by exchanging mode indices, which is cheap for COO format [1]. Then according to the new mode order, all the non-zero elements of $\mathcal{X}$ are sorted using a quick sort algorithm with the complexity of $\mathcal{O}(nnz_X log(nnz_X)$ where nnz_X is the number of non-zero elements in $\mathcal{X}$. In Figure 3.1, $\mathcal{X}$ only needs sorting due to its correct mode order; permutation and sorting are both needed for $\mathcal{Y}$. Permutation and sorting are necessary to improve data locality for an efficient implementation of our SpTC algorithms.

Output sorting ❺. The output $\mathcal{Z}$ is not sorted from our algorithms' computation pattern (see Lines 13 and 19 for details). Depending on the needs, sorting could be acted on $\mathcal{Z}$ after the computation, using the quick sort algorithm. This could avoid its potential sorting when used as an input for the subsequent SpTC computations. In our algorithms, sorting on $\mathcal{Z}$ is by default.

[1] For example, to exchange modes i_1 and i_2, we only need to switch the pointers of $inds[1]$ and $inds[2]$.

Sparse Accumulator for High-order Sparse Tensors

Sparse accumulator (SPA) is a popular approach in sparse matrix-sparse matrix multiplication (SpGEMM) [68, 69], which uses a sparse representation to hold the indices and non-zero values of the current active matrix row to do accumulation and is conceptually parallel. We extend SPA to SpTC (named *SpTC-SPA*) for an arbitrary-order sparse tensor and any contraction operation. Figure 3.1 uses the fourth-order tensor contraction example to illustrate the five stages.

Index search ❷. Take $x(0, 1, 0, 0)$ in Figure 3.1 to illustrate, the indices $(0, 0)$ in mode-3 and 4 are used to search in $\mathcal{Y}$ for sub-tensor $\mathcal{Y}(0, 0, :, :)$ to multiply with. A linear search iterates non-zeros of $\mathcal{Y}$ until $\mathcal{Y}(0, 0, :, :)$ is found.Similarly in Algorithm 1, we loop all non-zeros of $\mathcal{X}$ by units of sub-tensors in Line 2. For each non-zero $x(i_1, i_2, i_3, i_4)$, we use the indices (i_3, i_4) to do linear search in $\mathcal{Y}$ to locate the sub-tensor $\mathcal{Y}(i_3, i_4, :, :)$ to perform multiplication. The linear search has the complexity $\mathcal{O}(nnz_Y)$ by searching all non-zeros of $\mathcal{Y}$ in the worst case. To solve multi-dimensional index search challenge, we will construct $\mathcal{Y}$ as a hash table in Line 19.

We explain the reason of using COO format in our algorithms by comparing with the popular compressed storage row (CSR) [70] and its generalization compressed sparse fiber (CSF) [40] formats. For example, we can direct locate row indices in a CSR-represented sparse matrix, but not column indices. Similarly, except the first mode, all the other contract modes have to do linear search as well in a CSF-represented sparse tensor. (Refer to [43, 40] for more details) Thus, index search on CSF-represented $\mathcal{Y}$ will not be significantly better than its COO representation.

Accumulation ❸. In Figure 3.1, if $y(0, 0, :, :)$ is found,
$x(0, 1, 0, 0)$ times every non-zero in $\mathcal{Y}(0, 0, :, :)$, and accumulate the result to SPA. For example, $z(0, 1, 0, 3)$ accumulates the product of $x(0, 1, 0, 0)$ and $y(0, 0, 0, 3)$. If $SPA(0, 3)$ is already exists, it adds this product; otherwise, the product along with its indices $(0, 3)$ are appended to the SPA.

In Algorithm 1, since every $\mathcal{X}(i_1, i_2, :, :)$ independently accumulates to $\mathcal{Z}(i_1, i_2, :, :)$, the sparse accumulator SPA is allocated for each sub-tensor of $\mathcal{X}$. For each non-zero $x(i_1, i_2, i_3, i_4)$, if found $\mathcal{Y}(i_3, i_4, :, :)$ in index search, all non-zeros in $\mathcal{Y}(i_3, i_4, :, :)$ are stored contiguously and have spacial data locality due to the permutation and sorting of $\mathcal{Y}$ in input processing. Since every non-zero in $\mathcal{Y}(i_3, i_4, :, :)$ compute with $x(i_1, i_2, i_3, i_4)$,

thus $\mathcal{X}$ gets temporary data locality. If $SPA(j_3, j_4)$ already exists, it adds up the product v; otherwise, v along with its indices (j_3, j_4) are dynamically appended to SPA. We also employ the linear search to locate $SPA(j_3, j_4)$ with the complexity $\mathcal{O}(|SPA|)$, the size of SPA. Once the traverse of all non-zeros in $\mathcal{X}(i_1, i_2, :, :)$ is done, SPA contains the final results of $\mathcal{Z}(i_1, i_2, :, :)$. The same multi-dimensional search challenge occurs in index search stage, which is optimized with hash table in Line 19.

Writeback ❹. Figure 3.1 shows the simple write-back stage by copying SPA values to $\mathcal{Z}(0, 1, :, :)$. In Line 19, we will introduce another temporary data for better parallelization and memory locality.

To solve the challenge of the unknown output size, traditionally two approaches, a two-phase method with symbolic and numeric phases [50] and a loose upper-bound size prediction [51, 52], have been investigated. Symbolic phase counts the number of non-zero elements of the output, which is expensive, then precise memory space is allocated to proceed the computation (numeric phase). A loose upper-bound size prediction uses probabilistic or upper bound methods to allocate large enough memory, which is more than sufficient, for the output. In *SpTC-SPA*, we use dynamic vectors for the SPA and output tensor, like progressive method [69] but more precise. The total time complexity of *SpTC-SPA* is

$$\begin{aligned} T_{SPA} = &\mathcal{O}(nnz_X log(nnz_X) + nnz_Y log(nnz_Y)) \\ &+ \mathcal{O}(2 \times nnz_X \times nnz_Y + nnz_Z) + \mathcal{O}(nnz_Z log(nnz_Z)) \end{aligned} \tag{3.1}$$

, where the three terms correspond to the time complexity of input processing, *computation* with index search, accumulation, and writeback, and output sorting. Figure 3.2 illustrates the execution time breakdown of the stages of *SpTC-SPA*. This evaluation matches theoretical analysis in Eq. (3.1), the SpTC time is dominated by computation stage. Stages ❶ and ❺, shown together as *input/output processing*, takes less than 1% of the algorithm. Compared to the two-phase method, our *SpTC-SPA* approach highly reduces the input processing time; while compared to the prediction methods, *SpTC-SPA* can highly reduce SPA and the output space. Thus, our *SpTC-SPA* algorithm is a good baseline for SpTCs, by following the spirit of SpGEMM SPA approach with dynamic, precise memory allocation, and good data locality, to support arbitrary-order sparse tensors and any tensor contraction operations. Stages ❷ and

Algorithm 2: *Sparta*: Sparta sparse tensor contraction for arbitrary-order data.

Input: Input tensors $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_{N_X}}$ and $\mathcal{Y} \in \mathbb{R}^{J_1 \times \cdots \times J_{N_Y}}$, contract modes C_X, C_Y
Output: The output tensor $\mathcal{Z}$

```
1  Permute and sort X if needed;
2  Obtain N_F, |F^X|, sub-tensors of X, and its ptr_F;
3  Convert Y to HtY with LN(C^Y) as keys and (LN(F^Y), val^Y) as values;
4  // Compute: Z = X ×_{C_X}^{C_Y} Y
5  for f in 1, ..., N_F do
6      Initiate thread-local HtA with F^Y as keys
7      for nz in ptr_F[f], ..., ptr_F[f+1] do
8          if LN(C^X_nz) is not found in HtY then
9              continue
10         for (LN(F^Y_nz), val^Y_nz) in (LN(F^Y), V^Y) of HtY do
11             v = val^X_nz * val^Y_nz
12             if LN(F^Y_nz) is found in HtA then
13                 Accumulate val^HT_nz += v
14             else
15                 Insert (LN(F^Y_nz), v) to HtA
16     Form (F^X_nz, F^Y_nz) as coordinates and val^HT_nz as non-zero value and append to
       Z_local
17 Gather thread-local Z_local independently to Z
18 Permute and sort Z if needed
19 return Z
```

❸ are the performance bottlenecks in Figure 3.2 for all of our test cases and from Equation (3.1). We will emphasize optimizing these two stages in Lines 13 and 19.

Hash table-represented Sparse Tensor

To address the problems of multi-dimensional index search and inherit good data locality from *SpTC-SPA*, we propose the hash table-represented input tensor $\mathcal{Y}$ with specifications for sparse tensors for the *index search* stage.

Figure 3.1 depicts the process of converting $\mathcal{Y}$ represented in COO format into a hash table HtY with a large-number representation and its usage in the example SpTC. The index search for $\mathcal{Y}(0,0,:,:)$ uses $\mathcal{X}$'s contract indices $(0,0)$, which is taken as the keys of HtY naturally. Since we need to keep the information of free indices of $\mathcal{Y}$, $(0,3)$, and non-zero values 4.0 for the next stage ❸, the tuple $((0,3),4.0)$ is put to the values of HtY. Since the keys of HtY are index tuples, as the tensor order grows, it is difficult and time-consuming to do key matching on multi-dimensional tuples.

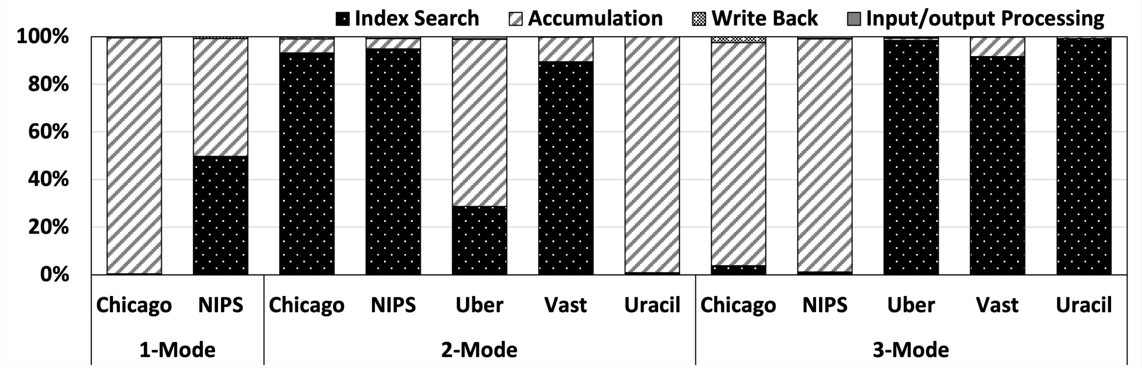


Figure 3.2: Percentage of execution time breakdown of *SpTC-SPA* (Algorithm 1).

We introduce a *large-number representation*, noted as the LN function in Figure 3.1, which converts a sparse index tuple to a large index in a dense pattern. For example, $(0,3)$ tuple is converted to $3 = 0 \times J_4 + 3$. Unique identifiers are extremely important for a fast hash table search. This large-number representation obtains unique numbers for every tuple of keys in HtY, hence the index search becomes faster on HtY by doing integer comparison for keys. To create HtY from $\boldsymbol{\mathcal{Y}}$ in COO format, we use separate chaining hash table [71] with given-sized buckets to distribute the keys. Compared to COO format, the contract indices have no duplication due to the unique key feature of a hash table, which reduces the index search space. To maintain the good spacial data locality from Algorithm 1, for the non-zeros having the same key in $\boldsymbol{\mathcal{Y}}$, we adopt dynamic array to construct the values of HtY.

The creation and usage of HtY for an arbitrary-order SpTC with random contract modes C_X and C_Y are illustrated in Algorithm 2. The three for-loops are in the same order with those in Algorithm 1. The first and second loop sub-tensors in $\boldsymbol{\mathcal{X}}$ and non-zeros in the sub-tensor using ptr_F to indicate locations respectively. The indices of contract modes C_Y and the tuple of free modes and non-zero value (F^Y, val^Y) are taken as the keys and values of HtY respectively in Line 3. For each non-zero element nz, we search $LN(C^X_{nz})$, the large-number representation of the contract indices C_X of $\boldsymbol{\mathcal{X}}$, in HtY (Line 8). Compared to the linear search in *SpTC-SPA* with the complexity $\mathcal{O}(nnz_Y)$, the time complexity of hash table search on HtY is significantly reduced to $\mathcal{O}(1)$ [71]. We also optimize input processing, the COO-to-hashtable conversion is faster than permutation and sorting of $\boldsymbol{\mathcal{Y}}$, $\mathcal{O}(nnz_Y)$ versus $\mathcal{O}(nnz_Y log(nnz_Y))$.

Our proposed hash table-represented sparse tensor with the large-number compressed keys highly improves the SpTC performance by efficiently addressing multi-

dimensional index search issue and maintain temporary and spacial data locality. To reduce the frequency of index search, we always treat the larger input tensor as $\mathcal{Y}$ in our SpTC algorithms.

Hash table-based Sparse Accumulator

Hash table [50, 72, 73, 74], hashmap [75], and heap [76] are popular data structures to represent the accumulator in state-of-the-art SpGEMM research, where hash table performs the best from prior evaluations [50]. As mentioned in Line 13 and Figure 3.2, stage ❸ in *SpTC-SPA* could dominate the performance of an SpTC. To more efficiently accumulate the intermediate results, we propose a hash table-based accumulator HtA, shown in Figure 3.1. We take the free indices of $\mathcal{Y}$, $(0,3)$, as a key and refer to the intermediate result as the values of the hash table. Separate chaining hash table and the large-number representation LN are also adopted here for fast key matching and hash search.

We observe the key of HtA ($(0,3)$ in Figure 3.1) is the same with the free indices of $\mathcal{Y}$ in the value tuples of HtY (also $(0,3)$). To avoid the key conversion for HtA, we convert the free indices of $\mathcal{Y}$ to the large-number representation in stage ❶ (Line 3 in Algorithm 2). We directly retrieve the keys from the values of HtY, avoiding mode indices-key conversion between HtY and the accumulator HtA during computation. As depicted in Figure 3.1 and Algorithm 2, the accumulation performs similar to *SpTC-SPA* but on hash table HtA instead.

By far, we form the Sparta SpTC algorithm (Algorithm 2). Compared to *SpTC-SPA*, we replace $\mathcal{Y}$ and SPA with two hash table HtY and HtA with large-number representation respectively. Sparta solves the multi-dimensional index search challenge, get faster processing for input $\mathcal{Y}$, extract unnecessary index computation/conversion out of the computation, while maintain the good data locality shown in *SpTC-SPA*, to reduce the SpTC execution time. The total time complexity of Sparta:

$$\begin{aligned} T_{Sparta} &= \mathcal{O}(nnz_X log(nnz_X) + nnz_Y) \\ &+ \mathcal{O}(2 \times nnz_X \times nnz_{Favg} + nnz_Z) + \mathcal{O}(nnz_Z log(nnz_Z)) \end{aligned} \tag{3.2}$$

, where nnz_{Favg} is the average size of all sub-tensors (e.g. $\mathcal{Y}(j_1, j_2, :, :)$ in Algorithm 1). The three terms correspond to the time complexity of stages ❶, *computation*

with ❷, ❸, and ❹, and ❺. Eq. (3.2) shows that depending on different sparse tensors, the SpTC time could be dominated by different stages.

Parallelization

We parallelize all the five stages of *SpTC-SPA* and Sparta algorithms. For stage ❶, since permutation takes negligible time, we parallelize the quick sort algorithm using OpenMP tasks, which is also used in stage ❺. Sparta has the COO-to-hashtable representation for $\mathcal{Y}$ in stage ❶, we parallelize sub-tensors of $\mathcal{Y}$ and use locks on the buckets of HtY to ensure correct insertion and updates. Since the separate chaining hash table relatively evenly distributes the search requests, locks on multi-threading gets an acceptable performance (7.8× speedup on average over a sequential version using 12 threads in our experiments).

In computation, we parallelize the outermost loop for sub-tensors of $\mathcal{X}$ (Line 2 in Algorithm 1 and Line 5 in Algorithm 2). Thus, the sparse accumulator SPA in *SpTC-SPA* and hash table accumulator HtA in Sparta are both thread-private and each thread can do accumulation independently. Due to the dynamic output structure, directly write the intermediate thread-local SPA or HtA results to $\mathcal{Z}$ is not feasible. We introduce thread-local dynamic Z_{local} in Algorithm 2 to write the intermediate results. After one thread completes its execution, we have the size of Z_{local} which can be used to allocate the space for $\mathcal{Z}$. Then each thread could writes its Z_{local} to $\mathcal{Z}$ in a parallel pattern. The introduction of Z_{local} helps to solve the unknown output challenge in multi-threading parallel environment and improves stage ❹ with the cost of the affordable thread-local storage Z_{local}.

3.2.2 Data Placement on Persistent Memory-based Heterogeneous Memory Systems

We discuss our approaches to leveraging HM to address the memory capacity bottleneck of SpTC.

Characterization Study

To motivate our solution of data placement on heterogeneous memory, we characterize memory accesses of major data objects of Sparta (Algorithm 2), in terms of access patterns (sequential/random and read/write) in Table 3.1. Five stages, input processing ❶, computation (combined ❷ index search, ❸ accumulation, ❹ writeback)

and output sorting ❺, are considered with six major data objects, i.e., the two input tensors ($\mathcal{X}$ and $\mathcal{Y}$), the hash table-represented second input tensor (HtY), thread-local hash table-based accumulator (HtA), the thread-local temporary data ($\mathcal{Z}_{local}$), and the output tensor ($\mathcal{Z}$).

We study the performance impact of the placement of six data objects on tensor Nell-2 with 2-Mode contraction in Figure 3.3, by evaluating Sparta on a server with an HM with PMM and DDR4. We use the execution time to reflect the underneath PMM and DRAM memory characteristics and an accurate performance behavior of SpTC. Our baseline is the Sparta execution time when residing all data in DRAM, which achieves the fastest performance on the HM. We perform six tests: each one by placing only one data object in PMM, while leaving the others stay in DRAM. We have three interesting observations that guide our data placement for Sparta.

Observation 1: Performance difference between read and write matters a lot to performance of Sparta. For example, the memory access pattern associated with $\mathcal{Y}$ in the stage ❶ is sequential read-only, and placing it on PMM causes ignorable performance loss; In contrast, the memory access pattern associated with $\mathcal{Z}_{local}$ in the stage ❸ is sequential write-only, and placing it on PMM causes 12.9% performance loss. The bandwidth difference between read and write on PMM is about 3×, which leads to the difference in Sparta's performance.

Observation 2: Sequential and random accesses have large performance difference. For example, the memory access pattern associated with $\mathcal{Y}$ in the stage ❶ is sequential read-only, and placing it on PMM causes ignorable performance loss; In contrast, the memory access pattern associated with HtY in the stage ❷ is random read-only, and placing it on PMM causes 30.8% performance loss. The performance difference between sequential and random accesses on PMM is due to the unique architecture of PMM (e.g., the combining buffer in devices [77]); Sequential accesses also makes hardware prefetching more effective for improving data locality.

Observation 3: The performance of Sparta is not sensitive to the placement of some data objects on PMM. For exampling, placing $\mathcal{X}$ and $\mathcal{Y}$ on PMM, Sparta has ignorable performance loss, because of the memory access patterns discussed in the above two observations.

The first two observations are unique to PMM compared to traditional DRAM. Read and write, sequential and random accesses both has small performance difference

Table 3.1: Memory access patterns associated with data objects in six stages (”Ran” = Random; ”Seq” = Sequential; ”RW” = Read-Write; ”RO” = Read-Only; ”WO” = Write-Only). (Note that for Ht$\mathfrak{Z}$ in Summation, we employ temporal locality to always maintain the most frequently used bucket in DRAM. The memory access pattern of key-value nodes within the bucket is still random. For $\mathfrak{Z}_{input}$, the first two passes are “Seq, RO” and the last pass is “Ran, RO”.)

Stages	Data Objects							
	$\mathfrak{X}$	$\mathfrak{Y}$	HtY	HtA	$\mathfrak{Z}_{local}$	$\mathfrak{Z}$	$\mathfrak{Z}_{input}$	Ht$\mathfrak{Z}$
Input Processing ❶	Ran, RW	Seq, RO	Ran, RW	-	-	-	-	-
Index Search ❷	Seq, RO	-	Ran, RO	-	-	-	-	-
Accumulation ❸	-	-	-	Ran, RW	Seq, WO	-	-	-
Writeback ❹	-	-	-	-	Seq, RO	Seq, WO	-	-
Output Sorting ❺	-	-	-	-	-	Ran, RW	-	-
Summation ❻	-	-	-	-	-	Seq, RO	Ran, RO	Ran, RW

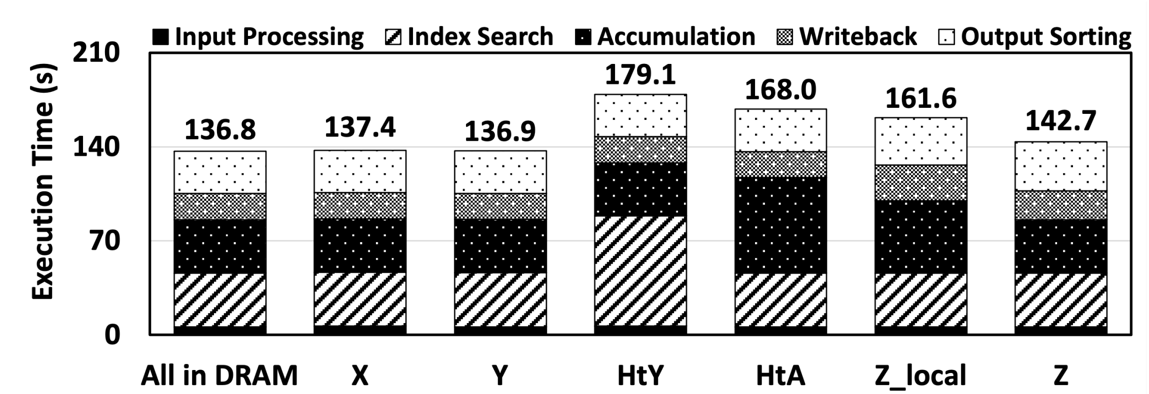


Figure 3.3: Performance after placing a data object in PMM while leaving others in DRAM. The x axis shows the data object placed in PMM. “All in DRAM” means all data objects are placed in DRAM.

in DRAM. We get the same observations for other 14 datasets.

Data Placement Strategy

Driven by the characterization results, we use the following data placement strategy. $\mathfrak{X}$ and $\mathfrak{Y}$ is always on PMM, because of the observation 3. For the other four data objects, we decide their placement in DRAM, following the priority of $HtY \succ HtA \succ \mathfrak{Z}_{local} \succ \mathfrak{Z}$. For each of the four data objects, we make best efforts to place it into DRAM. This means that given a data object, if there is remaining DRAM space after excluding that consumed by the data objects with higher priority, then that object is placed into DRAM as much as possible; If there is no remaining DRAM space, that object is placed into PMM.

To implement the above data placement strategy, we must estimate the memory consumption of the four data objects, to decide whether they should be placed into

DRAM or not. We discuss it as follows.

The placement of HtY. We estimate the memory consumption of HtY using Equation 3.3 based on tensor information and knowledge on data structures used in HtY. In Equation 3.3, $Size_{HtY}$ is the memory consumption of HtY; $Size_{ep}$, $Size_{idx}$ and $Size_{val}$ are the size of the entry pointer for a bucket in HtY, the size of an index, and the size of a value, respectively; $\#Buckets_{HtY}$ is the number of buckets in HtY; $nnz_{\mathcal{Y}}$ is the number of non-zero elements in $\mathcal{Y}$; N_Y is the number of modes of $\mathcal{Y}$.

$$\begin{aligned} Size_{HtY} = \; & Size_{ep} \cdot \#Buckets_{HtY} + nnz_{\mathcal{Y}} \cdot (Size_{idx} \cdot N_Y \\ & + Size_{val} + Size_{ep}) \end{aligned} \tag{3.3}$$

Equation 3.3 includes the memory consumption for metadata (i.e., the pointers pointing to each bucket in the hash table, modeled as $Size_{ep} \cdot \#Buckets_{HtY}$); Equation 3.3 also includes the memory consumption for storing all non-zero elements of $\mathcal{Y}$ in HtY, each of which consumes memory for an index, a value, and a pointer pointing to another element, modeled as $Size_{idx} \cdot N_Y + Size_{val} + Size_{ep}$.

To use Equation 3.3, we must know $nnz_{\mathcal{Y}}$ and $\#Buckets_{HtY}$. $nnz_{\mathcal{Y}}$ as a tensor feature is typically known; $\#Buckets_{HtY}$ is defined by the user, and hence is known.

The placement of HtA. We use Equation 3.4 to estimate the memory consumption of HtA. While Equation 3.3 estimates the exact memory consumption, Equation 3.4 gives an upper bound on the memory consumption ($Size_{HtA}$).

$$\begin{aligned} Size_{HtA} = Size_{ep} \cdot \#Buckets_{HtA} + nnz^X_{Fmax} \cdot nnz^Y_{Fmax} \cdot (Size_{idx} \\ \cdot |F^Y| + Size_{val} + Size_{ep}) \end{aligned} \tag{3.4}$$

In Equation 3.4, $|F^Y|$ is the number of free modes of $\mathcal{Y}$; nnz^X_{Fmax} is the maximum size of all non-zero sub-tensors $\mathcal{X}(F^X, :, \ldots, :)$; nnz^Y_{Fmax} represents the maximum size of all non-zero sub-tensors $\mathcal{Y}(C^Y, :, \ldots, :)$. The product of nnz^X_{Fmax} and nnz^Y_{Fmax} gives an upper bound on the number of non-zero elements stored in HtA.

Equation 3.4 gives an upper bound, because we do not know the exact number of non-zero elements in $\mathcal{Y}$ that have the same contract indices as those in $\mathcal{X}$; We use the

maximum number to give an upper bound and ensure there is enough space allocated in DRAM for HtA. Using the upper bound does not cause significant waste of DRAM space, because HtA per thread is usually 10-50 MB (even with the largest dataset using 768GB memory in our evaluation). Given tens of threads in a machine, the upper bound takes only a few GB of DRAM, which is typically a small portion of DRAM space in an HPC server.

To use Equation 3.4, we must know nnz^X_{Fmax} and nnz^Y_{Fmax}. nnz^X_{Fmax} and nnz^Y_{Fmax} are known after the stage ❶, and the dynamic allocation of HtA can happen after the stage ❶ but before the stage ❷ where HtA is accessed. Hence, Equation 3.4 can be used to effectively direct data placement. In addition, DRAM is evenly partitioned between threads for placing HtA per thread, in order to avoid load imbalance.

The placement of $\mathfrak{Z}_{local}$. The memory consumption of $\mathfrak{Z}_{local}$ can be estimated after HtA is filled (Line 16 in Algorithm 2) and before memory allocation for $\mathfrak{Z}_{local}$ happens. The memory consumption of $\mathfrak{Z}_{local}$ is equal to the size of HtA plus the size of $F^X_{nz} \cdot nnz_{HtA}$, where F^X_{nz} refers to free indices of a non-zero element in $\mathfrak{X}$ and nnz_{HtA} is the number of non-zero elements in HtA. In addition, DRAM is evenly partitioned between threads for placing $\mathfrak{Z}_{local}$ per thread, in order to avoid load imbalance.

The placement of $\mathfrak{Z}$. The size of $\mathfrak{Z}$ is the summation of the size of $\mathfrak{Z}_{local}$ in each thread. The size of $\mathfrak{Z}$ is estimated in Line 17 in Algorithm 2, before memory allocation for $\mathfrak{Z}$ happens.

Static placement vs. dynamic migration. The data placement strategy in Sparta is static, which means a data object, once placed in DRAM or PMM, is not migrated to PMM or DRAM in the middle of execution. The traditional solutions are application agnostic and dynamic. They track page (or data) access frequency [60, 61, 62, 63, 64, 65, 66, 67] or manage DRAM as a hardware cache for PMM [57, 58, 59, 78] to decide the placement of data objects on DRAM and PMM. The traditional solutions, once determining frequently accessed data (hot data), dynamically migrate hot or cold data between DRAM and PMM for high performance. However, those dynamic migration solutions cannot work well in our case because they can cause unnecessary data movement. For example, the performance of Sparta is not sensitive to the placement of $\mathfrak{X}$ and $\mathfrak{Y}$ on PMM and DRAM, because of their sequential read patterns. The dynamic solutions can unnecessarily migrate them to DRAM for high performance. For another example, HtY has a random access

pattern. Any dynamic migration solution cannot effectively capture its pattern and hence causes unnecessary data migration. Our evaluation results in the evaluation show that two dynamic migration solutions (i.e., hardware-based Memory mode and software-based IAL [79]) perform worse than Sparta by 10.7% (up to 28.3%) and 30.7% (up to 98.5%) respectively.

Other datasets. We evaluate 15 datasets in total, and 11 of them shows the same priority for data placement (i.e., $HtY \succ HtA \succ \mathfrak{Z}_{local} \succ \mathfrak{Z}$). However, there are four cases showing different priority (i.e., $HtA \succ HtY \succ \mathfrak{Z}_{local}$ and $\mathfrak{Z}$). For those uncommon cases, we can use the same methodology to determine data placement; Our methods to determine the sizes of the data objects are still valid.

3.3 Evaluation

3.3.1 Evaluation Setup

Platforms. The experiments in Sections 3.3.2, 3.3.3 and 3.3.4 are run on a Linux server consisting of 96 GB DDR4 memory and Intel Xeon Gold 6126 CPU including 12 physical cores with 2.6 GHz frequency on one socket. The experiments in Section 3.3.5 are run on an Intel Optane Linux server containing Intel Xeon Cascade-Lake CPU including 24 physical cores with 2.3 GHz frequency. Each socket has 6 × 16 GB of DRAM and 6 × 128 GB Intel Optane DIMMs. All implementations (Sparta and other approaches) are compiled by gcc-7.5 and OpenMP 4.5 with -O3 optimization option. All experiments were conducted on a single socket with one thread per physical core. Each workload was run 10 times and we report the average execution time.

Datasets and expression. We use sparse tensors, derived from real-world applica-

Table 3.2: Characteristics of sparse tensors in the evaluation.

Tensors	Order	Dimensions	#Nonzeros	Density
Nell-2	3	$12K \times 9K \times 28K$	76M	2.4×10^{-5}
NIPS	4	$2K \times 3K \times 14K \times 17K$	3M	1.8×10^{-6}
Uber	4	$183 \times 24 \times 1K \times 1K$	3M	2×10^{-4}
Chicago	4	$6K \times 24 \times 77 \times 32$	5M	1×10^{-2}
Uracil	4	$90 \times 90 \times 174 \times 174$	10M	4.2×10^{-2}
Flickr	4	$320K \times 28M \times 2M \times 731$	113M	1.1×10^{-14}
Delicious	4	$533K \times 17M \times 2M \times 1K$	140M	4.3×10^{-15}
Vast	5	$165K \times 11K \times 2 \times 100 \times 89$	26M	8×10^{-7}

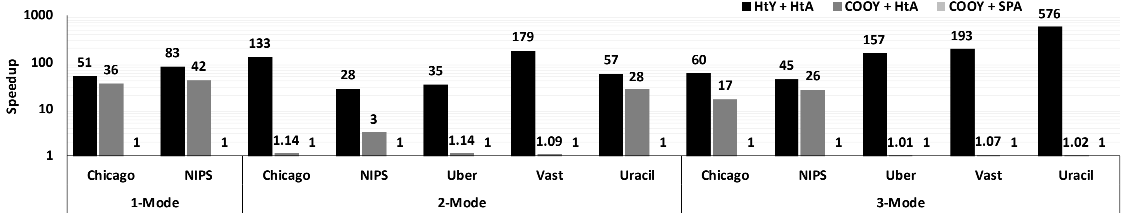


Figure 3.4: Speedups of HtY+HtA (i.e., Sparta) and COOY+HtA over COOY+SPA (i.e., SpTC-SPA) for SpTCs on Chicago, NIPS, Uber, Vast and Uracil with 1-mode, 2-mode and 3-mode.

tions, that appear in Table 3.2, ordered by modes and nonzero density. The tensors are included in FROSTT [80]. Tensor Uracil [81, 24] is from a real-world CCSD model in quantum chemistry, formed by cutting off values smaller than 1×10^{-8} verified by chemists.

For some SpTC, the memory requirement is larger than the system memory capacity. We do not evaluate the performance of those SpTC. For a tensor with different expression, we use a "*" to distinguish. For example, Chicago and Chicago* are the same tensors with different expression. Sparta includes five stages, ❶ input processing, computation (combined ❷, ❸, ❹), and ❺ output sorting.

3.3.2 Overall Performance

Figure 3.4 shows the performance comparison of using HtY+HtA (i.e., Sparta), COOY+HtA and COOY+SPA (i.e., SpTC-SPA) on tensors Chicago, NIPS, Uber, Vast and Uracll with 1-mode, 2-mode and 3-mode SpTC respectively. In Figure 3.4, we observe that HtY+HtA significantly outperforms COOY+HtA with $1.4 - 565\times$ performance improvement. The results show that HtY is much efficient than COOY. Also, we found that COOY+HtA significantly outperforms COOY-+SPA with $1\% - 42\times$ performance improvement. The results expose that HtA is much efficient than SPA.

We observe that the performance improvement of Sparta over COOY-SPA on Uracil with 3-mode is larger than others. This is because the execution time of stage ❷ dominates the total execution time (99.3%) and the total execution time is relatively large (1072 seconds) than others. Based on the time complexity difference between HtY and COOY in stage ❷, the larger execution time SpTC spends, the larger performance improvement Sparta can achieve. In Figure 3.2, the total execution

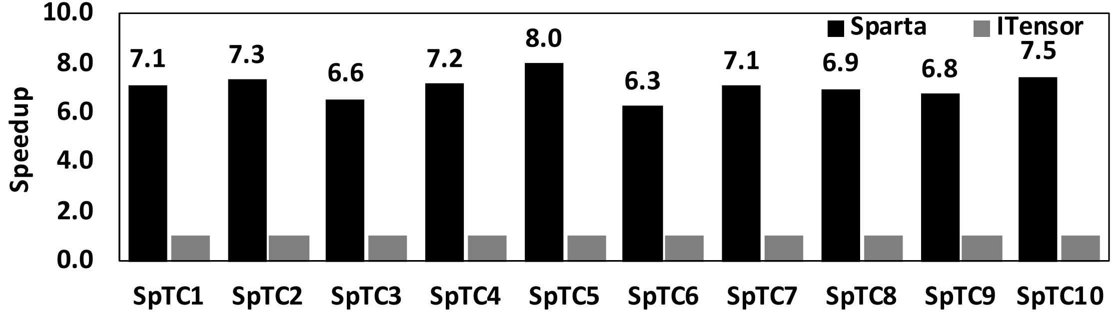


Figure 3.5: Speedups of Sparta over ITensor on Hubbard-2D model using different SpTC expression with different sparse input tensors.

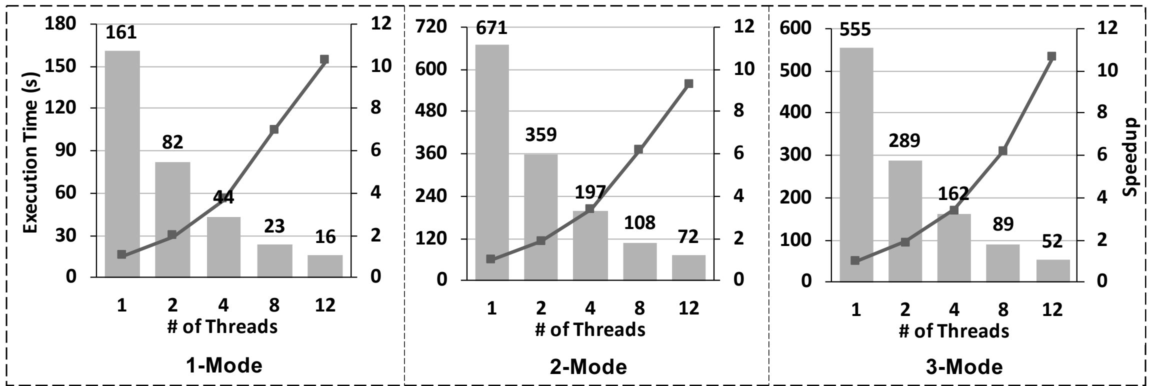


Figure 3.6: Thread scalability of parallel Sparta on SpTCs on NIPS with 1-mode, Vast with 2-mode and NIPS with 3-mode.

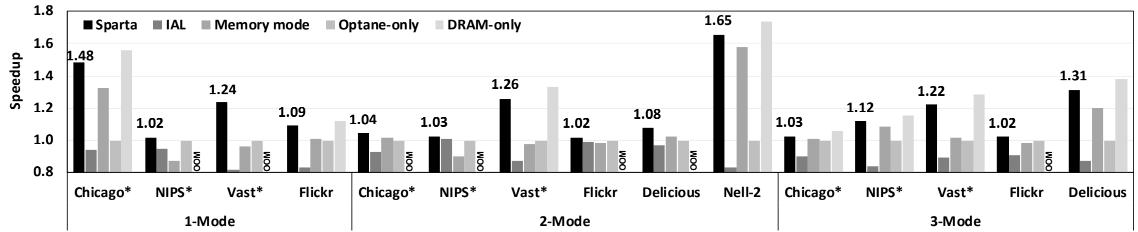


Figure 3.7: Speedups of Sparta, IAL, Memory mode and Dram-only over Optane-only for SpTCs on Chicago*, NIPS*, Vast*, Flickr, Delicious and Nell-2 with 1-mode, 2-mode and 3-mode.

is dominated by stages ❷ and ❸ in COOY-SPA (99.6%). Since the execution time of ❷ and ❸ is highly reduced by Sparta, the execution time of stages ❷ and ❸ might not be the bottleneck of an SpTC. In our experiments with Sparta, the time in ❷ accounts for 4.7%; the time of stage ❸ is 61.6%; the time of stage ❹ is 9.6%; the stage ❶ accounts for 3.3% and ❺ is 20.8%.

3.3.3 Performance Comparison to ITensor

In this experiment, we compare the performance of Sparta and ITensor. ITensor [19] is a state-of-the-art library for multi-threading, block-sparse tensor contraction on a single machine, which is the most related to Sparta among other works. ITensor is configured with its best configurations described in its repository [82]. SpTC expressions with different tensors (SpTC1 to SpTC10) are from a well-known quantum physics model (Hubbard-2D) [27] in ITensor [82], and those tensors are formed by cutting off values smaller than 1×10^{-8} verified by physicists. We choose ITensor as a representative for comparison rather than others (such as libtensor [55], TiledArray [53], CTF [54] and TACO [83]), because libtensor only supports sequential block-wise SpTC [55] while TiledArray and CTF are distributed, and TACO does not support high-order SpTC yet. Figure 3.5 shows the performance comparison between Sparta and ITensor. We observe that Sparta significantly outperforms ITensor with 7.1× performance improvement on average. We demonstrate that Sparta can also be employed for applications featured with block-wise SpTC.

3.3.4 Thread Scalability

Figure 3.6 shows the performance of parallel Sparta over the sequential version. Sparta achieves 10.2×, 9.3× and 10.7× speedup on NIPS with 1-mode, Vast with 2-mode and NIPS with 3-mode using 12 threads. Different stages have different thread scalability. Evaluation with 15 datasets using Sparta, the average speedup of parallel execution over sequential execution achieves: 10.4 × in stage ❷; 10.9 × in stage ❸; 9.5 × in stage ❹; 6.8 × in stage ❶ and 6.2 × in ❺. Though the thread scalability of stages ❶ and ❺ are not as good as the computation stages (❷,❸,❹), the SpTC is always dominated by the computation stages. Thus, Sparta achieves high overall thread scalability.

3.3.5 Sparta on Heterogeneous Memory Systems

Now we study the performance of Sparta on HM, compared with a state-of-the-art solution for HM management (i.e., IAL (Improved Active List) [79]), hardware-managed cache approach (i,e, PMM Memory mode), Optane-only (i.e., AppDirect mode with assigning all data objects to Optane) and DRAM-only (i.e., assign all data objects to DRAM). IAL is configured with its best configurations based on the IAL

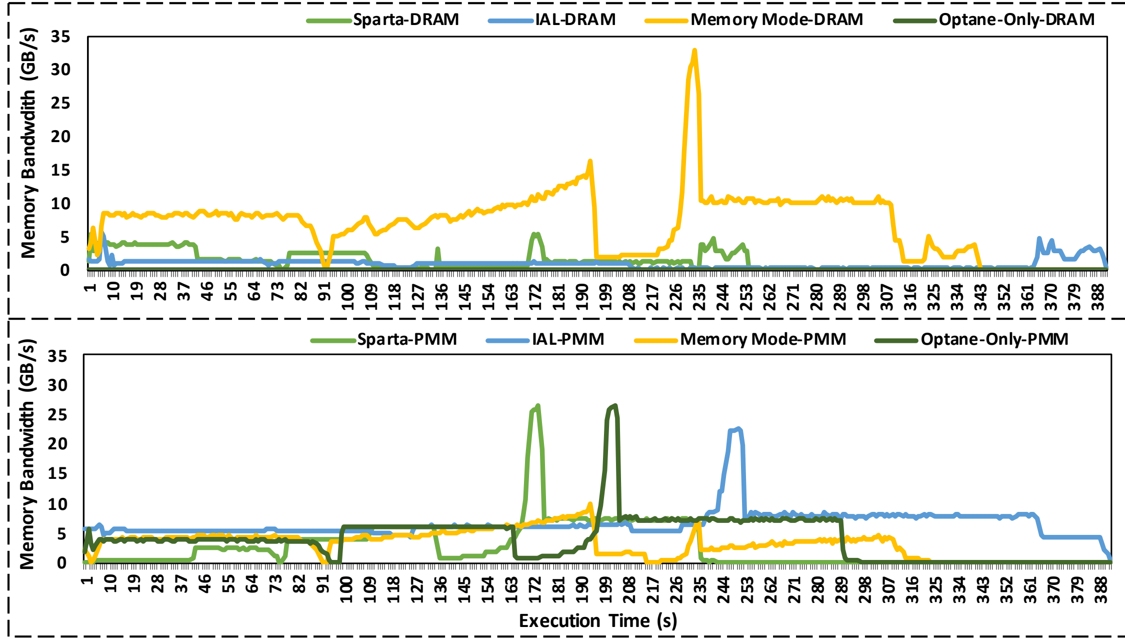


Figure 3.8: Memory bandwidth of Sparta, IAL, PMM Memory mode and Optane-only on Vast with 1-mode SpTC.

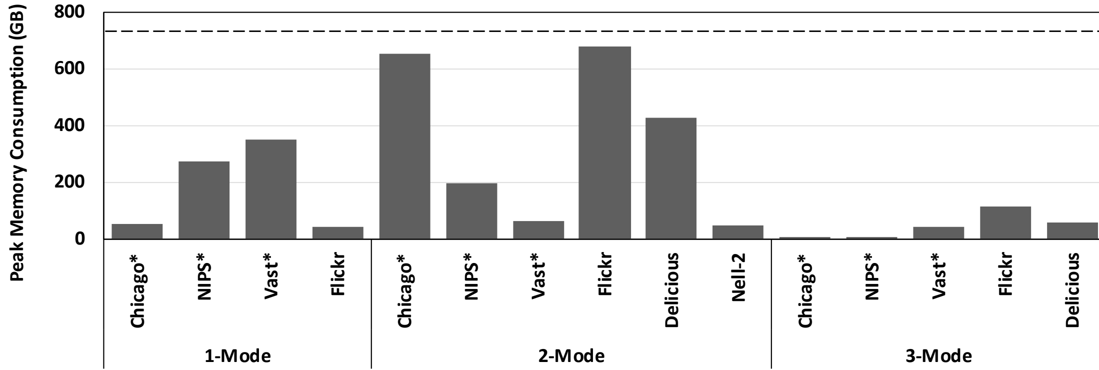


Figure 3.9: Peak memory consumption of SpTCs on Chicago*, NIPS*, Vast*, Flickr, Delicious and Nell-2 with 1-mode, 2-mode and 3-mode.

repository [84]. Figure 3.9 shows the peak memory consumption of SpTCs in the experiment.

As shown in Figure 3.7, Sparta outperforms IAL with 30.7% performance improvement on average (up to 98.5%). Also, Sparta achieves 10.7% (up to 28.3%) and 17% (up to 65.1%) performance improvement on average than PMM Memory mode and Optane-only approaches respectively. Furthermore, Sparta is comparable to DRAM-only approach with only 6% performance loss. For some SpTC (e.g., Chicago* with 3-mode), because the memory bandwidth requirement is small, the performance

difference between Sparta and Optane-only is small. For example, if we assign all data objects to DRAM (i.e., DRAM-only, the configuration with the best performance) on Chicago* with 3-mode, the performance improvement is only 6% over Optane-only.

In Figure 3.8, we observe that the average PMM memory bandwidth of IAL is larger than Sparta. This is because IAL causes undesirable data movement and such data movement causes higher PMM memory bandwidth. The average DRAM memory bandwidth of PMM memory mode is larger than Sparta because PMM Memory mode manages DRAM as a hardware cache for PMM and unnecessarily migrates data objects to DRAM for high performance without being able to be aware of access patterns of data objects.

3.4 Related Work

Tensor contraction. Tensor contraction has a long history in scientific computing in chemistry, physics, and mechanics. Dense tensor contraction has been studied for decades on diverse hardware platforms [85, 86, 87, 88, 89, 90, 54, 91, 92, 93, 94, 95]. The state-of-the-art sparse tensor contractions emphasize on block-sparse tensor contractions, between two tensors with non-zero dense blocks. The general approaches extract dense block-pairs of the two input tensors, then do multiplication by calling dense BLAS linear algebra and have the output tensor pre-allocated from domain knowledge or a symbolic phase [96, 97, 98, 99], such as libtensor [55, 56], TiledArray, and Cyclops Tensor Framework [100]. Our work proposes an efficient element-sparse tensor contraction and shows its performance advantages if a practical cutoff value gets quantum chemistry or physics data below 5% non-zero density. This work will be valuable for deep learning after introducing sparsity from model or data compression.

Sparse tensor formats. Researchers are making continuous effort on developing sparse tensor formats for high-order data, including compressed sparse fiber (CSF) [40], balanced and mixed-mode CSF (BCSF, MM-CSF) [39, 38], flagged COO (F-COO) [42], and hierarchical coordinate (HiCOO) [37] for general sparse tensors, and mode-generic and -specific formats for structured sparse tensors [101]. We choose COO format in this work as a start because CSF format also needs expensive search to locate $\boldsymbol{\mathcal{Y}}$ due to multi-dimensionality. Our hashtable-represented $\boldsymbol{\mathcal{Y}}$ is a new approach to compress a sparse tensor customized to a tensor contraction. This work is orthogonal to the tensor format works and will adopt a more compressed format for the sparse tensor $\boldsymbol{\mathcal{X}}$

according to SpTC operations.

Sparse matrix-matrix multiplication. Sparse matrix-matrix multiplication (SpGEMM) has been well-studied [74, 73, 72, 51, 76, 50, 75, 102, 68, 83]. Our hash table implementations can be improved from more advanced algorithms in [72, 73, 74, 103].

Data management on heterogeneous memory systems attracts a lot of attention recently. Many research efforts [60, 61, 62, 63, 64, 65, 66, 67] use a software-based solution to track data objects or page hotness to decide data placement on HM; Many research efforts [57, 58, 59, 78] use a hardware-based solution to profile memory accesses and decide data placement on HM. All of those solutions use dynamic migration and are application-agnostic. Sparta is different from them in terms of static data placement and application awareness.

3.5 Summary

SpTC plays an important role in many applications. However, how to efficiently implementing SpTC faces multiple challenges, such as unpredictable output size, time-consuming process to handle irregular memory accesses, and massive memory consumption. In this paper, we introduce Sparta, a high performance SpTC algorithm to address the above challenges based on the innovation of leveraging new data representation, data structures and emerging HM architecture. Sparta shows superior performance: evaluating with 15 datasets, we show that Sparta brings $28 - 576\times$ speedup over the traditional sparse tensor contraction; With our algorithm- and memory heterogeneity-aware data management, Sparta brings extra performance improvement on HM built with DRAM and PMM over a state-of-the-art software-based data management solution, a hardware-based data management solution and PMM-only by 30.7% (up to 98.5%), 10.7% (up to 28.3%) and 17% (up to 65.1%) respectively.

Chapter 4

Athena: High-Performance Sparse Tensor Contraction Sequences on Heterogeneous Memory

4.1 Motivation

Tensors, especially those high-dimensional sparse tensors, are attracting increasing attentions, because of their popularity in many applications. High-order sparse tensors have been studied well in tensor decomposition on various hardware platforms [37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49] with a focus on the product of a sparse tensor and a dense matrix or vector. The two sparse tensor contraction (SpTC) has been studied well [96, 53, 97, 98, 99, 25, 53, 100, 55, 56] where block-wise sparsity is the main focus. As the needs of element-/pair-wise sparsity emerge in applications from chemistry, physics and deep learning [19, 20, 21, 22, 23, 24], the recent work [25] studied element-wise SpTC. In essence, SpTC, a high-order extension of sparse matrix-matrix multiplication (SpGEMM), multiplies two sparse tensors along with their common dimensions.

Nevertheless, SpTC commonly is shown as sequences in quantum chemistry, quantum physics and deep learning [19, 20, 21, 22, 23, 24] as a foundation of coupled cluster single double (Triple), CCSD(T) [26], high-order tensor decomposition methods, etc. An SpTC sequence (SpTCSeq) performs a sequence of sparse tensor contractions which could be independent or have different dependency types. While sequences of

tensor contraction have been studied [88] with a focus on independent contractions and limited dependency types, such as identical input and shared outputs , an SpTCSeq is still lack of sufficient research for element-wise contractions and other dependency types. For example, Type 1 dependency, the output tensor of an SpTC taken as an input in another SpTC, occurs in 85% contractions in CCSD(T) from the NWChem library and has not been studied yet. However, multiple challenges impede obtaining high performance for a whole SpTC sequence.

First, redundant computation and memory traffic in an SpTCSeq lead to performance issues since it shares data objects across different SpTCs. As shown in Table 2.2, an SpTCSeq could include four dependency types and some data objects are shared across different SpTCs. Computation and memory traffic on the shared data objects are performed repeatedly. For example, in the type that two SpTCs share an identical input tensor, the processing on this input in the second SpTC can be avoided. Because of the shared data objects, computation and memory access on intermediate data objects are also performed repeatedly. For example, in the type that the output tensor of the first SpTC becomes the input tensor in the second SpTC, the intermediate results in the accumulation of the first SpTC can be directly reused to do the computation of the second SpTC, skipping multiple stages in the sequential execution. (Details in Line 26) This performance issue becomes severer when the redundant computation and memory access dominate the execution. Moreover, the performance suffers when we perform an SpTCSeq with a larger number of contractions.

Second, large memory consumption from large input and output tensors and intermediate results causes a memory capacity issue and creates pressure on the traditional DRAM-based machine. Sparse tensors from real-world applications easily consume a few to dozens of GB memory, while the output tensor could be even larger when generated with more non-zero elements than any of the input. The intermediate results could be large as well, especially in multi-threading environment where each thread has its local intermediate results. Compared to the well-studied sparse tensor times dense matrices/vectors [43, 45, 46, 37], SpTC results in substantial memory consumption easily, which can be beyond typical DRAM capacity (up to a few hundreds of GB) on a single machine. The memory capacity problem in an SpTCSeq becomes much more serious than in an individual SpTC. This memory capacity problem is especially pronounced in those HPC applications with increasing

dimension sizes of tensors [24, 19, 53, 54, 55, 56, 18]. Expanding DRAM capacity is not cost-effective, while adding cheap but much slower Solid-State Drive (SSD) causes significant performance drop.

Third, an SpTCSeq suffers from inefficient hardware utilization due to the diverse computation and memory patterns of different stages in an SpTC. For example, the accumulation stage in an SpTC with tensor Disilane, the average memory bandwidth is only 19.3% of the peak memory bandwidth, while the average CPU utilization is 71.9%; for index search stage (proposed in [25]) in an SpTC with the same tensor Disilane, the average CPU utilization is only 34.2% while the memory bandwidth is 44.1%. Given a 2-SpTC sequence, if we simply run the SpTCs sequentially, the hardware (e.g., computing units or memory bandwidth) is not fully utilized; If we co-run stages in the same intensive pattern, e.g., both memory-intensive, the SpTCSeq suffers from resource (e.g., memory bandwidth) contention; How to efficiently arrange stages across SpTCs in a sequence to achieve efficient hardware utilization without resource contention is challenging.

To address the above challenges, we propose Athena, a high-performance framework for SpTC sequences. To address the first challenge, we introduce shared hash table-represented sparse accumulator. In particular, given two SpTCs, we adopt hash table-represented sparse accumulator with reusing intermediate results in the first SpTC and then perform index search in the second SpTC to eliminate finishing up stages of the first SpTC and the starting expense of the second SpTC. Moreover, we retain shared data objects across SpTCs to eliminate unnecessary input processing and data migration. We also introduce a hash table-represented sparse tensor summation to significantly increase the performance of summation stages which are widely used in SpTC sequences. Athena effectively avoids redundant computation and memory operations in an SpTCSeq with shared data objects.

To address the second challenge, we explore the persistent memory-based heterogeneous memory (HM). In particular, the emerging Intel Optane DC Persistent Memory Module (PMM) provides up to 9TB memory capacity per node, which can be leveraged to address the memory capacity problem faced by SpTCSeq. PMM has slightly inferior bandwidth and latency (compared to DRAM) but with much lower price. As a result, PMM is often paired with a small DRAM, such that frequently accessed data objects can be placed into DRAM while the rest reside in PMM with

large memory capacity. PMM and DRAM builds a heterogeneous memory system.

The PMM-based HM raises a question on how to perform an SpTCSeq given limited DRAM space for high performance. Effectively placing data objects of an SpTCSeq in DRAM and PMM for high performance is critical to use PMM to address the memory capacity problem faced by SpTCSeq. To decide data placement on HM, the traditional solutions track page (or data) access frequency [60, 61, 62, 63, 64, 65, 66, 67, 104, 105] or manage DRAM as a hardware cache for PMM [57, 58, 59, 78], and then reactively place frequently accessed data objects into DRAM subject to the DRAM capacity constraint. However, those solutions are application-agnostic, and cause unnecessary and frequent data movement because of short-term variance in memory access patterns. The static data placement strategy [25] places data objects in DRAM or PMM without triggering dynamic migration in the middle of application execution. However, this strategy lacks the flexibility of handling irregular memory access patterns but with certain temporal locality.

Athena addresses the above problem by introducing a data-semantics guided data placement. This solution strikes a good balance between the static and dynamic data placement. In particular, it leverages data semantics to guide dynamic data placement. Instead of tracking the number of memory accesses at runtime as in the traditional dynamic data placement, we use the algorithm knowledge to reason the numbers of memory accesses (or hotness) at data object level during the construction of critical data structures in SpTCSeq, and then associate those numbers with data objects. After using the data semantics to identify those data objects, Athena is able to use hotness information to guide dynamic data placement.

To address the third challenge, we introduce stage parallelism for an SpTCSeq. We first characterize computation and memory behaviors of different stages in an SpTCSeq. Next, we co-run those stages in an SpTCSeq with respect to their integer operations (IOP)-, floating point operations (FLOP)-, or memory-intensive patterns, to avoid resource contentions and meanwhile improve the utilization of CPU and memory bandwidth. Hyperthreading technique is used for data prefetching and higher memory bandwidth usage to gain better overlapping between two stages. For exascale problems deployed in a distributed environment, Athena could help to reduce the number of nodes needed for computation due to its capability to solve large sparse tensors on each single node.

Our main contributions are summarized as follows:

- We introduce the first, high-performance SpTCSeq system for element-wise sparse tensor contraction sequence, named Athena.
- We explore the emerging PMM-based HM to address memory capacity limitation suffered in the tensor computations, and use algorithm knowledge and data semantics to guide dynamic data placement.
- Evaluating with 12 datasets, we show that Athena brings 327-7362 $\times$ speedup over the state-of-the-art SpTC algorithm. With the dynamic data placement guided by data semantics, Athena brings performance improvement on HM built with DRAM and PMM over a state-of-the-art software-based data management solution, a hardware-based data management solution, and PMM-only by 1.58$\times$ (up to 2.09$\times$), 1.82$\times$ (up to 2.58$\times$) and 2.34$\times$ (up to 2.94$\times$) respectively.

4.2 Design

4.2.1 Algorithm Design

This section introduces our SpTCSeq algorithm for the five dependency types in Table 2.2 and efficient sparse tensor summation.

Hash Table-Represented Sparse Tensor Summation

A general process of two element-wise sparse tensor summation is as follows. Given a sparse output tensor $\mathfrak{Z}_{pre}$ produced from previous SpTCs or other operations (hidden in the "+=" operator in Table 2.2) and a sparse output tensor $\mathfrak{Z}$ in the current SpTC, the sparse tensor summation performs three steps. First, a non-zero element along with its indices from $\mathfrak{Z}_{pre}$ is selected. Next, the summation searches the corresponding non-zero element(s) in $\mathfrak{Z}$ with the exact same tuple of indices. Finally, if the particular non-zero element in $\mathfrak{Z}$ with the same indices is found, $\mathfrak{Z}$ is updated with the sum of the two non-zero values under the tuple of indices. Otherwise, the non-zero element in $\mathfrak{Z}_{pre}$ is appended to $\mathfrak{Z}$ as a new element. Meanwhile, the non-zero elements of $\mathfrak{Z}$ which are not updated during the summation remain the same. This process is expensive in the searching step due to multi-dimensionality of the tuple of

Algorithm 3: *Sparta*: sparse tensor contraction sequence for arbitrary-order data in the expression $\mathcal{Z} = \mathcal{X} \times_{C_X}^{C_Y} \mathcal{Y} \times_{F_Y}^{C_W} \mathcal{W} + \mathcal{Z}_{pre}$.

Input: Input tensors $\mathcal{X} \in \mathbb{R}^{I_1 \times \cdots \times I_{N_X}}$, $\mathcal{Y} \in \mathbb{R}^{J_1 \times \cdots \times J_{N_Y}}$, and $\mathcal{W} \in \mathbb{R}^{J_1 \times \cdots \times J_{N_W}}$, contract modes C_X, C_Y, C_W, and output tensor $\mathcal{Z}_{pre}$ produced from previous SpTCs

Output: The output tensor $\mathcal{Z}$

```
1  Permute and sort X if needed;
2  Obtain N_F, |F^X|, sub-tensors of X, and its ptr_F;
3  Convert Y to HtY and W to HtW
4  for f in 1,...,N_F do
5      Initiate thread-local HtA and Shared-HtA
6      for nz in ptr_F[f],...,ptr_F[f+1] do
7          Index_Search(c_nz^X, HtY)
8          Accumulation(f_nz^Y, v^X, v^Y, v^HtA)
9      for (key,v^HtA) in HtA do
10         if key is not found in HtW then
11             continue
12         for (LN(f_nz^W), v^W) in (LN(F^W), V^W) of HtW do
13             if LN(f_nz^W) is found in HtW then
14                 Accumulate v^SHtA += v^HtA * v^W
15             else
16                 Insert (LN(f_nz^W), v^HtA * v^W) to Shared-HtA
17     Form (f_nz^X, f_nz^W) as coordinates and v^SHtA as non-zero value and append to
       Z_local
18 Gather thread-local Z_local independently to Z
19 Convert Z to HtZ with M^Z as keys,
20 for nz in Z_pre do
21     if LN(m_nz^Zpre) is not found in HtZ then
22         Append (LN(m_nz^Zpre), v^Zpre) to HtZ
23     else
24         Append (LN(m_nz^Zpre), v^Zpre + v^Z) to HtZ
25 Convert HtZ to Z, and permute/sort Z if needed
26 return Z
```

indices as keys and dynamically updating $\mathcal{Z}$ especially with appending new non-zero elements from $\mathcal{Z}_{pre}$.

To address the above problems, we propose the hash table-represented sparse tensor summation for an SpTC. Figure 4.1 depicts our proposed approach as Summation, stage 5 (The rest stages will be explained in Line 26). It is extremely time-consuming to perform key matching on multi-dimensional tuples, especially when the tensor order is large high [25]. We adopt the hash table representation from the work [25] by first

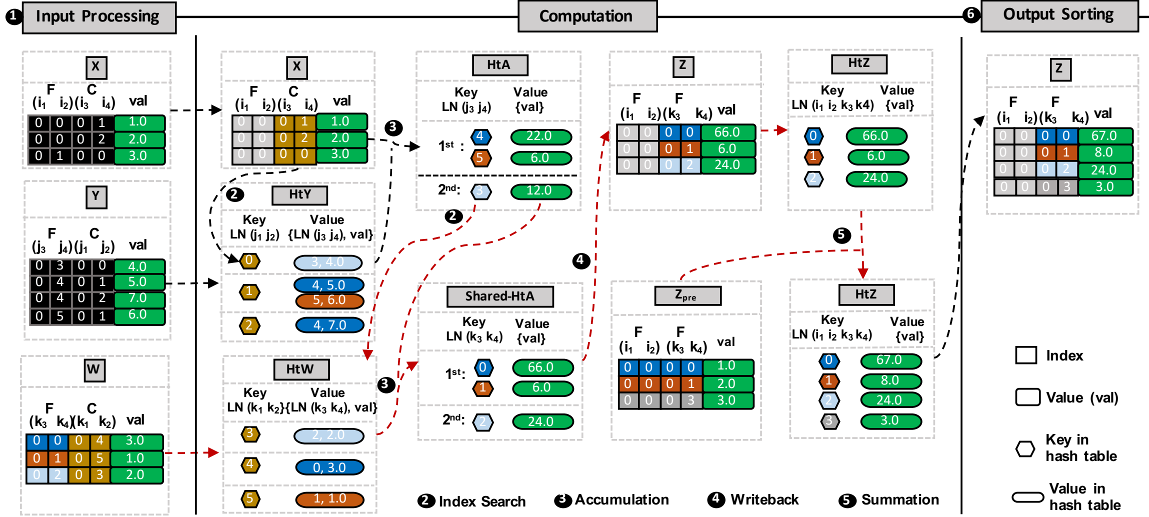


Figure 4.1: Workflow of Type 1 dependency of two SpTCs in Table 2.2, using shared hash table-represented sparse accumulator and hash table-represented sparse tensor summation (indicated by red arrows)

converting the sparse output tensor $\mathcal{Z}$ in COO format to a hash table-represented $Ht\mathcal{Z}$. The index tuples of $\mathcal{Z}$ are taken as the keys of HtZ naturally, different from the key construction in Sparta [25]. A large-number representation, noted as the LN function in Figure 4.1, is also leveraged to convert a sparse index tuple to a large and unique index. The index search is improved by 1) reduced searching space of the unique index keys of the hash table; 2) pinpointing the targeted index much faster than the traditional linear search approach with a constant algorithm complexity. To fast update $\mathcal{Z}$ and maintain good spacial data locality, we adopt dynamic arrays to construct the values of HtZ for the non-zeros having the same key. $Ht\mathcal{Z}$ is then converted back to $\mathcal{Z}$ as the final output. As shown in line 19 to 24 in Algorithm 3, Athena converts $\mathcal{Z}$ to HtZ and then iterates all non-zeros in $\mathcal{Z}_{pre}$. If the nnz index is not found in HtZ, Athena appends the key-value pair to HtZ. Otherwise, Athena appends the index along with the summed values to HtZ.

Our hash table-represented sparse tensor summation extends to support the fused multiplication and summation and plays a critical role for performance when the size of output tensor is similar to or even larger than input tensors in an SpTC.

Shared Hash Table-Represented Sparse Accumulator

We observe that the traditional approach for Type 1 dependency (Output as input) in Table 2.2 leads to repeated and inefficient computations and data movement

because the two SpTCs share some intermediate data objects. To address this problem, we introduce a shared hash table-represented sparse accumulator (named *Shared-HtA*). Figure 4.1 depicts the workflow of our Shared-HtA design. The first SpTC, $\boldsymbol{\mathcal{Z}}' = \boldsymbol{\mathcal{X}} \times \boldsymbol{\mathcal{Y}}$, follows the five-stage computation proposed in the work [25]: input processing, index search, accumulation, writeback, output sorting, stages 1-4 and 6. The hash table-represented summation is the new stage 5.

Once the index search and accumulation stages of the first SpTC are completed, we treat the free modes F_Y of $\boldsymbol{\mathcal{Y}}$ in HtA to be the contract modes of the second SpTC. We then employ F_Y as the key to search the corresponding contract modes C_W in HtW in another index search stage for the second SpTC, $\boldsymbol{\mathcal{Z}} \mathrel{+}= \boldsymbol{\mathcal{W}} \times \boldsymbol{\mathcal{Z}'}$. For example, in Figure 4.1, 3 is used as the key in the 2nd HtA to search the corresponding contract modes (3) in HtW. Next, we generate the Shared-HtA to store the intermediate results during the accumulation stage of the second SpTC (2 and 24.0). Once the index search and accumulation of $\boldsymbol{\mathcal{X}}$ with the same free modes are completed (i.e., index search and accumulation stages in both black and red arrows in Figure 4.1) , the intermediate results of Shared-HtA are converted back to COO format ($(0, 2)$ tuple) and appended to $\boldsymbol{\mathcal{Z}}$ along with its accumulated result (24.0) for summation and output sorting stages. Finally, the intermediate Shared-HtA is released to save memory space. This approach also leverages the same large-number representation approach in the work [25] and converts the input tensor $\boldsymbol{\mathcal{W}}$ of the second SpTC to the hash table representation HtW. The full algorithm of Shared-HtA is illustrated in line 4 - 18 in Algorithm 3. Athena first completes the index search and accumulation in the first SpTC in line (5-8). Athena then iterates each key-value pair in HtA and leverages Shared-HtA to calculate and store the intermediate results (line 9-17). Finally, Athena gather thread-local $\boldsymbol{\mathcal{Z}}_{local}$ independently to $\boldsymbol{\mathcal{Z}}$ (line 18).

By employing the Shared-HtA, we eliminate multiple time consuming stages: appending intermediate results in the accumulation stage, writeback and output sorting stages of the first SpTC and input permutation/sorting in the input processing stage of the first SpTC's output and large-number conversion in the index search stage of the second SpTC. Therefore, our proposed Shared-HtA avoids the repeated computation and eliminate unnecessary data movement and hence significantly improves the performance and memory efficiency for an SpTCSeq in Type 1.

Types 2-4 dependency in Table 2.2 is less frequently occurred compared to Type

1. Type 4 has been studied in previous research [106], which converted to $\mathcal{Z} \mathrel{+}= (\mathcal{X} + \mathcal{W}) \times \mathcal{Y}$ to improve performance through contraction fusion and replacing one tensor product with a summation operation. For an SpTCSeq in Types 2 and 3 dependency, we could utilize a simple strategy to avoid redundant data movement between DRAM and storage (or PMM) if the DRAM space is adequate. For Type 2, after completing the first SpTC, the identical input tensor $\mathcal{Y}$ remains in DRAM; similarly for the shared output tensor $\mathcal{Z}$ for Type 3.

Stage Parallelism

We propose stage parallelism to better utilize machine resources, such as CPU and memory bandwidth. This is different from the prior research [106, 88] which uses strategies, like compiler-based tensor contraction expression generator [106] or hand-tuned optimization [88], to obtain independent expressions of an SpTCSeq in an optimal order following dependency types. After resolving different types of dependent SpTCSeq, we treat an SpTCSeq in Types 1-5 as a single simple/compound, independent SpTC in this section.

Stage characterization. We first explore the characteristics of the six stages (i.e., input processing, index search, accumulation, writeback, summation and output sorting) in the SpTC algorithm (Algorithm 3) and categorize them into IOP-, FLOP-, and memory-intensive behaviors. We calculate the compulsory number of integer operations (IOPs), floating point operations (FLOPs), and memory traffic of the stages. The word "compulsory" means the minimum requirements of operations or memory traffic assuming an infinite cache size, which gives a fundamental idea of an algorithm behavior and has been used in performance model analysis [107]. Diverse IOPs, FLOPs, and memory traffic behaviors have been observed in the six stages.

In particular, we observe that three stages, namely input processing, index search and output sorting, dominated by integer operations (IOPs). Sorting and/or permutation, the primary components of input processing and output sorting stages, have frequent index comparison and exchanging. Index search performs search on index tuples, thus only IOPs are needed. These stages are referred to as *IOP-intensive stages*. Accumulation and summation stages, referred to as *FLOP-intensive stages*, consist of the core floating point operations of a tensor contraction. The writeback stage has pure memory access, named *memory-intensive stages*. IOP-, FLOPS-, and

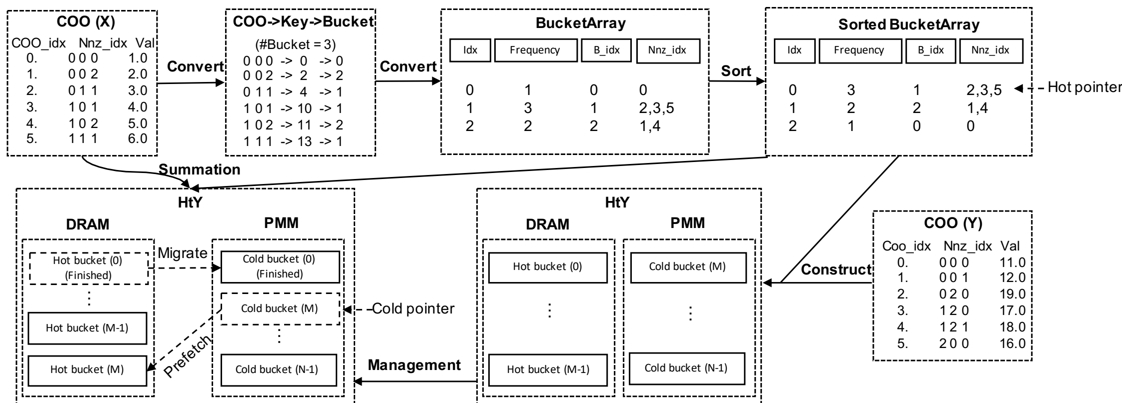


Figure 4.2: Workflow of the dynamic data placement based on data semantics.

memory-intensive stages utilize different computing units or memory components to fulfill, which makes it possible to parallelize them from multiple SpTCs to improve hardware utilization. The above observations on stage characterization drive our design.

Concurrency control. Based on the stage characterization study and the fact that an SpTCSeq includes a large amount of independent SpTCs (accounting for 91% of all SpTCs in a chemistry application), we propose *stage parallelism* to improve hardware utilization for high performance. In particular, given an SpTCSeq, Athena co-runs an IOP-, FLOP-, or memory-intensive stage in an SpTC with alternative intensive stages in another SpTC.

Athena employs hyper-threading to co-run the stages with different intensive behaviors. This means that a memory-intensive stage and a compute (IOP/FLOP)-intensive stage or an IOP-intensive stage and a FLOP-intensive stage share a physical CPU core and use two hyperthreads to co-run. Because of the complementary characteristics of the two stages, using hyperthreading to co-run them increase instruction throughput (hence increasing CPU utilization). Athena co-runs two SpTCs but not more at the same time, because of the following reasons. (1) We conduct 32 tests using 12 input problems ranging from small to large datasets (see Table 4.1), and find that co-running two compute stages in a core using hyperthreading leads to at least 94.1% CPU utilization, which is sufficiently high; The co-run between an IOP-intensive stage and a FLOP-intensive stage is sufficient because of their compute-intensive feature. More than two SpTCs may incur instruction pipeline stall due to the limited integer or floating point function units. (2) Our tests also show that using one thread to

run a memory-intensive stages consumes at least 60.3% of peak memory bandwidth. Hence, co-running a memory-intensive stage with another compute-intensive stage is enough to improve the utilization of memory bandwidth. In general, the accurate number of SpTCs to co-run is determined by the CPU utilization of individual stages and heavily relying on input and output data.

4.2.2 Data Management on PMM-based Heterogeneous Memory Systems

We leverage the heterogeneous memory system to address the memory capacity bottleneck in an SpTCSeq.

Static Data Placement

We consider eight major data objects in the six stages of an individual SpTC. The eight major data objects are the two input tensors ($\mathfrak{X}$ and $\mathfrak{Y}$), the hash table-represented second input tensor (HtY), the thread-local hash table-based accumulator (HtA), the thread-local temporary data ($\mathfrak{Z}_{local}$), the output tensor ($\mathfrak{Z}_{pre}$) produced from previous SpTCs, the output tensor ($\mathfrak{Z}$) in the current SpTC, and the hash table-represented output tensor (HtZ).

Athena uses the static data placement strategy [25] to decide the placement of $\mathfrak{X}$, $\mathfrak{Y}$, HtA, $\mathfrak{Z}_{local}$ and $\mathfrak{Z}$ on DRAM and PMM for individual SpTCs. This strategy considers memory access patterns associated with each data object, and places them in DRAM or PMM without migration in the middle of an SpTC execution. This strategy leads to higher performance than dynamic data placement, because of the avoidance of unnecessary data movement, discussed in [25]. In particular, for each SpTC, Athena places $\mathfrak{X}$, $\mathfrak{Y}$ and $\mathfrak{Z}_{pre}$ on PMM, because memory accesses to them are sequential and read-only in computation. Such a memory access pattern does not lead to big performance difference between placing data objects on DRAM and PMM, because of effective hardware prefetching and higher PMM performance in read (refer to [25] for details). Athena places HtA, $\mathfrak{Z}_{local}$ and $\mathfrak{Z}$ in DRAM, following the priority of $HtA \succ \mathfrak{Z}_{local} \succ \mathfrak{Z}$, according to the performance variance when moving them from PMM to DRAM (a data object causing higher variance has a higher priority). For large data objects such as HtA, $\mathfrak{Z}_{local}$ and $\mathfrak{Z}$, Athena makes the best efforts to place them on DRAM. This means that given a data object, if there is remaining DRAM

space after excluding the memory consumed by data objects with higher priority, that data object is placed into DRAM as much as possible; If there is no remaining DRAM space, that data object is placed into PMM.

Athena is different from Sparta [25] in terms of data placement from the following perspectives. First, Athena manages data objects from all SpTCs together. This means that when the DRAM space is not large enough to save all data objects, not only data objects in an individual SpTC are managed following the priority discussed above, but also all data objects across SpTCs are managed following the above priority. This cross-SpTCs static data placement is feasible, because the sizes of data objects can be estimated [25] and the execution order of the six stages is known. For the data objects with the same priority in different SpTCs, Athena gives higher DRAM priority to those SpTCs with smaller memory footprint. This is because the SpTC with less memory footprint tends to have shorter execution time and hence can release the DRAM space to other SpTCs sooner.

Second, Athena dynamically migrates HtY and HtZ between DRAM and PMM, instead of using the static data placement in Sparta. This is because the two data objects have a large amount of random memory accesses. For example, the memory read/write accesses to HtY and HtZ account for 45% and 27% of all memory accesses in an SpTCSeq with the tensor Disilane (see Table 4.1 for Disilane). Placing them to DRAM can lead to significant performance improvement. However, the two data objects are the largest ones among all data objects and using the static data placement places most of data in PMM, which causes large performance loss. Athena uses a dynamic data placement strategy based on data semantics to place hot data from the two data objects into DRAM as much as possible, discussed as follows.

Dynamic Data Placement based on Data Semantics

Dynamic data placement has been employed to enable high performance on heterogeneous memory [60, 61, 62, 63, 64, 65, 66, 67, 57, 58, 59, 78, 104, 105]. Most of those solutions are application agnostic, which means that they track page (or data) access frequency [60, 61, 62, 63, 64, 65, 66, 67] or manage DRAM as a hardware cache for PMM [57, 58, 59, 78] without the knowledge of data semantics. However, the data semantics gives critical indications on memory access patterns, which is useful to direct data placement and avoid unnecessary data movement. Leveraging data

semantics to direct data placement has recently been used in data analytics workloads (e.g., traffic analysis) [108]. We study how to use data semantics to build HtY and HtZ and direct data placement in an SpTCSeq.

HtY and HtZ have random memory access patterns but still have hot non-zero elements frequently accessed. Those non-zero elements can be eliminated out of DRAM because of short-term variance in memory access patterns, if we use application-agnostic solutions. Using data semantics we can keep hot non-zero elements in DRAM to address the above problem. Furthermore, using data semantics allows us to know in advance which non-zero elements will be accessed, enabling effective prefetching from PMM to DRAM.

The existing HtY and HtZ built from $\mathfrak{Y}$ and $\mathfrak{Z}$ are based on the hash table [25], which is difficult to get the number of accesses for each element in advance to direct data placement, and the access order of non-zero elements in the hash table is also random, making prefetching difficult. Hence, we introduce a new method that exposes element hotness, during the construction of the hash table-based HtY and HtZ. As a result, using the semantics of HtY and HtZ, the data hotness is associated with data, allowing Athena to implement dynamic data placement and prefetching.

Figure 4.2 depicts the workflow of our design. Our design has four steps: bucket conversion, bucket sorting, hash table construction, and semantics-guided dynamic data placement.

Bucket conversion. Figure 4.2 uses tiny sparse tensors $\mathfrak{X}$ and $\mathfrak{Y}$ as an example. Using the method in [25], Athena first converts indices tuple of non-zero elements to keys based on the large-number representation function (LN), in order to make the key of each element unique. But different from [25], after the above conversion, Athena uses a common hash function (the Jenkins hash function) to distribute indices to different buckets. The number of buckets equals to the number of non-zero elements in $\mathfrak{Y}$ (or $\mathfrak{Z}$).

Bucket sorting. In the bucket conversion step, Athena records the number of non-zero elements in each bucket, which indicates the number of accesses to each bucket. The number of accesses to each bucket can be determined based on the number of non-zero elements, because SpTCSeq iterates non-zero elements in $\mathfrak{X}$ (or $\mathfrak{Z}_{pre}$) and then performs index search in HtY (or HtZ). The numbers of non-zero elements collected from buckets form a bucket array. The size of the bucket array is

determined by the number of non-zero elements in $\mathcal{Y}$ (or $\mathcal{Z}$). Athena sorts the bucket array in an decreasing order. The sorting is necessary to enable quick identification of bucket hotness.

Hash table construction. Athena constructs the hash table-represented sparse tensor HtY from $\mathcal{Y}$ using the existing approach. But different form it, during the hash table construction, Athena traverses the sorted bucket array from the most accessed bucket (i.e., the bucket 0) and puts them into DRAM one by one till DRAM runs out of space. At that point, the remaining buckets, including those with the number of accesses as zero, are placed into PMM.

During the bucket placement on HM, Athena leverages a simple analytical model to estimate the memory requirement of each bucket: $Size_{idx} \cdot N_{\mathcal{X}} + Size_{val} + Size_{ep}$, in which $Size_{idx}$, $Size_{val}$ and $Size_{ep}$ are the size of an index, the size of a value, and the size of the entry pointer pointing to the bucket, respectively; $N_{\mathcal{X}}$ is the number of modes of $\mathcal{X}$.

Data-semantics guided data management. During the computation stages, Athena maintains two helper threads to manage data between DRAM and PMM. The first helper thread is referred to as the *migration thread*. Whenever an element is accessed, the number of accesses for the corresponding bucket in the bucket array is reduced by one. Once the number of accesses for a hot bucket in DRAM becomes zero, meaning that the bucket will not be accessed any more, Athena put the hot bucket ID to an FIFO queue for the migration thread to move to PMM. The migration thread continuously checks the FIFO queue to migrate the bucket to PMM.

The second helper thread is referred to as the *prefetching thread*. When there is DRAM space for HtY (or HtZ) in DRAM, the prefetching thread migrates the hottest bucket from PMM to DRAM before it is needed by computation.

The semantics guided data management in Athena significantly improves the performance by directing data placement based on the expected data hotness/coldness using data semantics.

4.3 Evaluation

4.3.1 Evaluation Setup

Platforms. We use an Intel Optane (PMM) Linux server, equipped with an Intel Xeon Cascade-Lake CPU including 24 physical cores at 2.3 GHz frequency. The CPU is attached with 6× 16 GB of DRAM and 6× 128 GB Intel PMM DIMMs. All implementations (Athena and other approaches) are compiled by gcc-7.5 and OpenMP 4.5 with `-O3`. All experiments were conducted on a single socket with one thread per physical core. Similar to recent work [25, 62, 63, 109], we use one-socket evaluation to highlight data movement between DRAM and PMM. Each workload is run 10 times and we report the average execution time.

Datasets We use sparse tensors summarized in Table 4.1. Those tensors are derived from real-world applications. Six tensors are derived from the well-known Coupled Cluster Singles and Doubles with perturbative triples correction, CCSD(T) [26] from chemistry [24]; Four tensors are derived from the notable Hubbard model from quantum physics in ITensor [19]; Two tensors are from large sparse tensor collection FROSTT [80]. Tensors in chemistry and physics are constructed by cutting off magnitude values smaller than 1×10^{-8} verified by domain scientists. We evaluate a real-world chemistry and four physics applications with Athena in Section 4.3.5 and Section 4.3.4 separately to study the effectiveness of Athena. We use a 4-SpTC sequence in Types 1 and 5 dependencies for each experiment to benchmark the performance if not mentioned otherwise. Section 4.3.5 will show the applications of

Table 4.1: Characteristics of sparse tensors in the evaluation

Domains	Tensors	Order	Dimensions	#Non-zeros	Density
Chemistry	Benzene	4	336 × 336 × 42 × 42	4M	1.9×10^{-2}
	Cytosine	4	400 × 400 × 58 × 58	19M	3.4×10^{-2}
	Disilane	4	270 × 270 × 34 × 34	4M	4.2×10^{-2}
	Guanine	4	280 × 280 × 78 × 78	32M	6.6×10^{-2}
	Siosi3	4	64 × 64 × 186 × 186	6M	4.0×10^{-2}
	Uracil	4	90 × 90 × 174 × 174	10M	4.2×10^{-2}
Physics	Hubbard-1D-P	5	4 × 4 × 93 × 36 × 432	0.3M	6.3×10^{-3}
	Hubbard-1D-T	5	131 × 4 × 413 × 36 × 4	0.4M	5.1×10^{-3}
	Hubbard-1D-Z	5	4 × 129 × 184 × 24 × 4	0.1M	5.2×10^{-3}
	Hubbard-2D	5	4 × 4 × 111 × 24 × 528	0.3M	6.6×10^{-3}
Others	NIPS	4	2K × 3K × 14K × 17K	3M	1.8×10^{-6}
	Vast	5	165K × 11K × 2 × 100 × 89	26M	8.0×10^{-7}

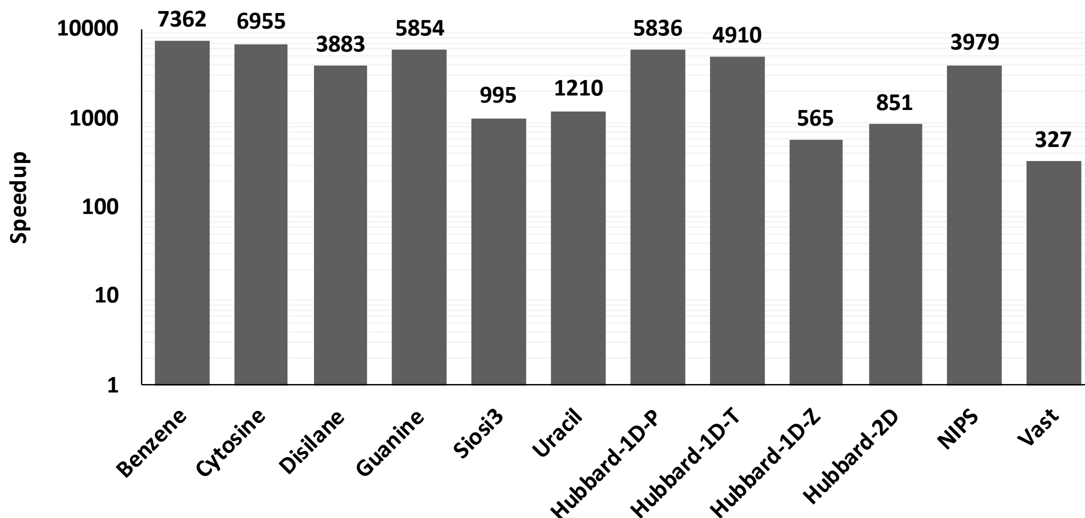


Figure 4.3: Overall speedups of Athena over Sparta for SpTCSeq on 12 tensors.

chemistry using a real SpTCSeq with ten SpTCs. Eight tensors exceed the DRAM capacity (96 GB) on our platform, which indicates the necessity of using PMM.

4.3.2 Overall Performance

In this experiment, to study the performance of Athena, we compare Athena with Sparta [25], the state-of-the-art element-wise sparse tensor contraction framework for an individual SpTC on heterogeneous memory. In general, as shown in Figure 4.3, Athena achieves 327-7362 $\times$ speedups over Sparta for SpTC sequences on 12 real-world tensors. Hash table-based sparse tensor summation contributes the most, 42-838 $\times$; while shared sparse accumulator and stage parallelism methods obtains 2.67-6.82 $\times$ and 1.21-1.46$\times$ speedup respectively. Performance analysis for every proposed optimization will be given in Section 4.3.3.

Figure 4.4 depicts the performance breakdown of Athena. Index search and accumulation stages are the most expensive stages for most tensors, which are in the computation part of an SpTC. Some tensors (e.g., Vast and Nell2) spend more time in output sorting than input processing stage, while some tensors (e.g., Hubbard-1D-T, Hubbard-1D-Z, Hubbard-2D) are vice versa, though they both use sorting and permutation algorithms. This is determined by the output tensor size versus the input tensors. For example, the size of the output tensor in Vast is 21$\times$ larger than the size of input tensor, while the size of the output tensor in Hubbard-1D-Z is 78% of the input tensor.

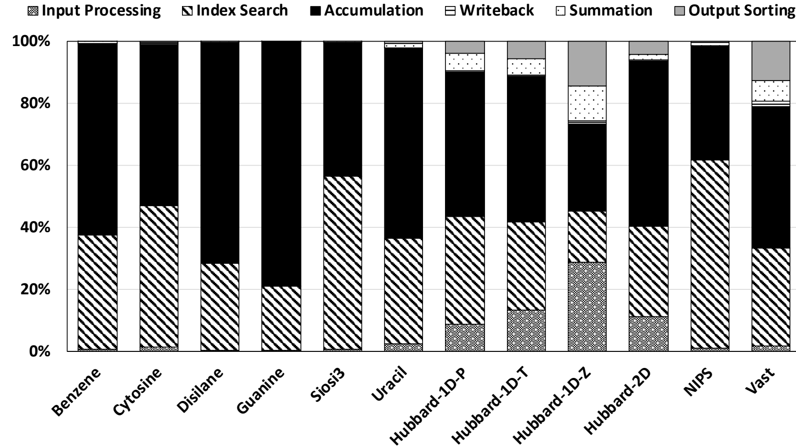


Figure 4.4: Percentage of execution time breakdown of Athena.

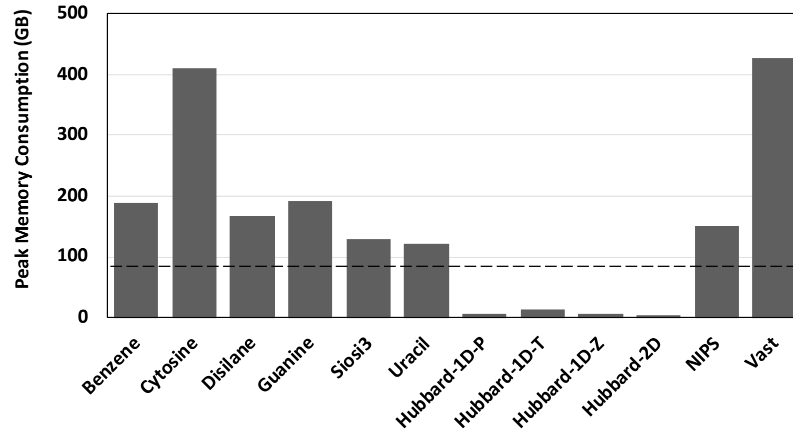


Figure 4.5: Peak memory consumption of SpTCSeq on 12 tensors.

Figure 4.5 shows the peak memory consumption of SpTC sequences in the experiment. Eight tensors consume more than DRAM space (96 GB), which cannot be performed without PMM memory. This indicates the large data used in applications and the necessity of using PMM. For even larger problems deployed in a distributed environment, Athena could help to reduce the number of nodes needed for computation due to the usage of the large PMM capacity.

4.3.3 Optimization Analysis

Hash table-based sparse tensor summation. Figure 4.6 shows the performance of using hash table-based sparse tensor summation on the 12 tensors respectively. In Figure 4.6, we observe that Athena significantly outperforms Sparta by 42-838 $\times$. The results show that our proposed hash table-based sparse tensor summation in

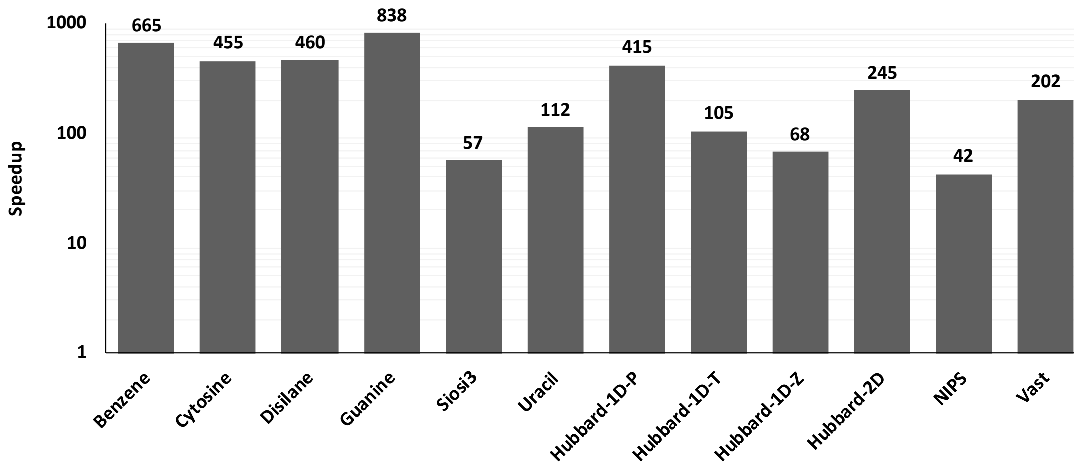


Figure 4.6: Speedups of Athena with hash-table represented summation over Sparta with traditional linear search-based summation.

Athena is more efficient than the traditional linear search-based summation.

Shared sparse accumulator. The shared sparse accumulator design in Athena reuses intermediate results of an SpTCSeq in Type 1 expression dependency to avoid redundant computation and memory operations and retains shared data objects to eliminate unnecessary input processing and data migration. Figure 4.7 shows the performance of using the shared sparse accumulator design ("Shared-HtA" in gray bars) in Athena compared to the sequential execution of the 4-SpTC sequence. We observe that Athena with the shared sparse accumulator design greatly outperforms the sequential execution by 2.67-6.82 $\times$, where Siosi3 obtains 6.82$\times$ and Hubbard-1D-Z is 2.67$\times$. Because the performance improvement of shared sparse accumulator derives from eliminating the redundant computation and memory operations in some stages, the performance improvement of leveraging shared sparse accumulator depends on the weights of those stages (i.e., for the first SpTC, the process of intermediate results appending in the accumulation stage, writeback stage and output sorting stage; for the second SpTC, the process of input permutation/sorting in the input processing stage and the process of large-number conversion in the index search stage). For example, those stages account for 85.3% of the total execution time in the Siosi3 while only account for 62.5% of the total execution time in the Hubbard-1D-Z.

Stage parallelism. Given a 4-SpTC sequence, the stage parallelism design in Athena co-runs stages in diverse patterns, IOP-, FLOP- and memory-intensive between two consecutive independent SpTCs. Figure 4.7 shows the performance of using the

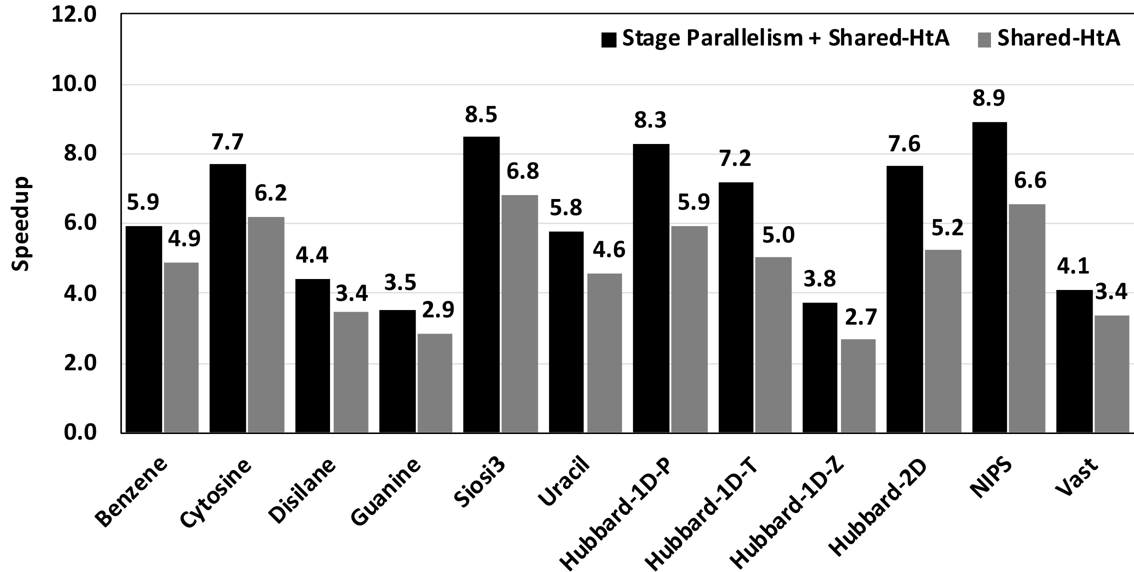


Figure 4.7: "Stage Parallelism" and "Shared-HtA" optimization speedup over the "Sparta + Summation" as the baseline.

stage parallelism in Athena compared to its sequential execution for this SpTCSeq. We observe that our proposed stage parallelism outperforms the sequential execution using Athena with 21%-46% performance improvement. Athena with the stage parallelism improves 14-19% CPU utilization and 12-24% memory bandwidth compared to the sequential execution. The performance improvement in different tensors varies because the execution time of overlapped stages varies. For example, the stage parallelism gains 17.6% performance improvement on Vast while 31.5% on Disilane. Assume the ideal case without considering the potential resource contention of co-running a 4-SpTC sequence, the upper bound of performance improvement in Vast could achieve 21.2% and in Disilane is 37.8%. Our stage parallelism is quite close to ideal upper bound. The performance of the ideal case is measured by separately running stages in the critical path. For some small sparse input tensors, the thread scalability is poor due to the inadequate parallelism in the index search and accumulation stages. Stage parallelism can bring extra performance improvement in this case. For example, stage parallelism for the four small tensors in physics, having the least non-zeros in all 12 tensors, brings 11% to 22% extra performance improvement than the other eight.

Data management on PMM-based heterogeneous memory systems. We study the performance of employing the semantics guided data management on HM, compared with a state-of-the-art solution for HM management (i.e., IAL

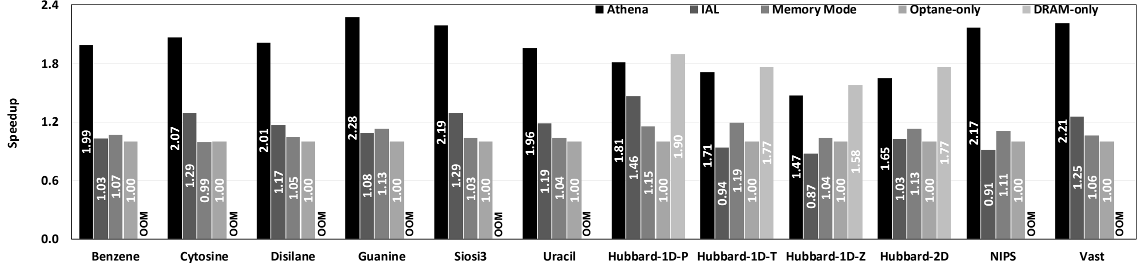


Figure 4.8: Speedups of Athena, IAL, Memory Mode and DRAM-only over PMM-only for SpTCSeq.

(Improved Active List) [79]), the hardware-managed cache approach (i,e, PMM Memory Mode), PMM-only (i.e., the AppDirect mode assigning all data objects to PMM) and DRAM-only (i.e., assign all data objects to DRAM). IAL is configured with its best configurations based on the IAL repository [84].

As shown in Figure 4.8, Athena with the semantics guided data management design outperforms IAL by 1.58× on average (up to 2.09×). Also, Athena achieves 1.82× (up to 2.58×) and 2.34× (up to 2.94×) performance improvement on average than PMM Memory Mode and PMM-only respectively. For some tensors (e.g., Hubbard-1D-Z), because the average memory bandwidth requirement is relatively smaller compared to others, the performance difference between Athena and PMM-only is small (47% improvement). For example, with Hubbard-1D-Z, if we place all data objects to DRAM (i.e., DRAM-only), the performance improvement is only 58%, compared to PMM-only.

We observe that the average PMM memory bandwidth of IAL is larger than that of Athena. This is because IAL causes undesirable data movement that consumes higher PMM memory bandwidth. The average DRAM memory bandwidth of PMM

Table 4.2: A 10-SpTC sequence from a CCSD(T) model.

$K[h4,h3,h1,h2]+=-0.125*L[p1,p2,h3,h4]*M[p1,p2,h1,h2]$
$N[p3,p4,h1,h2]+=1.0*K[h4,h3,h1,h2]*M[p3,p4,h4,h3]$
$O[p1,h3,p4,h2]+=0.5*L[p2,p4,h3,h1]*M[p1,p2,h1,h2]$
$N[p3,p4,h1,h2]+=1.0*O[p4,h4,p1,h1]*M[p3,p1,h4,h2]$
$P[p1,h3,p4,h2]=-0.5*L[p2,p4,h3,h1]*Q[p2,p1,h1,h2]$
$R[p3,p4,h1,h2]+=-1.0*P[p4,h4,p1,h1]*Q[p1,p3,h4,h2]$
$S[p1,h3,p4,h2]+=0.5*T[p2,p4,h3,h1]*U[p1,p2,h1,h2]$
$V[p3,p4,h2,h1]+=1.0*S[p4,h4,p1,h1]*Q[p3,p1,h2,h4]$
$R[p3,p4,h1,h2]+=1.0*S[p4,h4,p1,h1]*U[p3,p1,h4,h2]$
$V[p3,p4,h2,h1]+=-1.0*P[p4,h4,p1,h1]*M[p3,p1,h4,h2]$

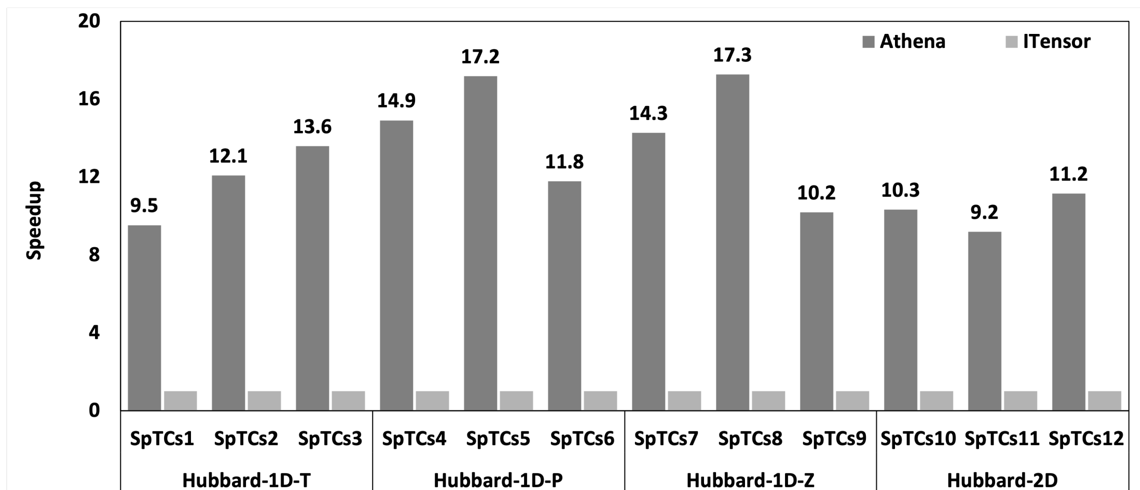


Figure 4.9: Speedups of Athena over ITensor on Hubbard-1D-T, Hubbard-1D-P, Hubbard-1D-Z and Hubbard-2D models using different SpTCSeq with different sparse input tensors.

memory mode is larger than that of Athena, because PMM Memory Mode manages DRAM as a hardware cache for PMM and unnecessarily prefetches data objects to DRAM for high performance without being able to be aware of semantic hotness of data objects.

4.3.4 Performance Comparison to ITensor

In this experiment, we compare the performance of Athena and ITensor, which is a state-of-the-art library for block-sparse, multi-threading tensor contraction on a single machine. As applications in ITensor only include independent SpTCs (Type 5) without summation, we employ stage parallelism and semantic-hotspot-based data management in Athena and compare the performance of Athena and with ITensor. SpTCs with different tensors (SpTCs1 to SpTCs12) are from well-known quantum physics models (Hubbard-1D-T, Hubbard-1D-P, Hubbard-1D-Z and Hubbard-2D) [27] in ITensor [82] with cutting off values smaller than 1×10^{-8}. Figure 4.9 shows the performance comparison between Athena and ITensor. We observe that Athena significantly outperforms ITensor with 12.6× performance improvement on average.

4.3.5 Application in Chemistry

We study the performance of Athena on a real-world SpTC sequence from NWChem in chemistry. NWChem is a well-known computational chemistry library for quantum chemical and molecular dynamics functionality [24]. We select a 10-SpTC

sequence derived from CCSD(T) [26]. The 10-SpTC sequence is concluded in Table 4.2 and cover 5 different expression types. We compare the performance of Athena to Sparta [25] on this sequence. Athena achieves 6232× speedup over Sparta combining all our designs. In particular, Athena achieves 635× speedup with hash table-based sparse tensor summation; 1.9× with semantic-hotspot-based data management; 4.3× with shared sparse accumulator; 1.2× with stage parallelism.

4.4 Related Work

Sparse tensor contraction. Dense tensor contraction has been studied for decades on diverse hardware platforms [85, 86, 87, 88, 89, 90, 54, 91, 92, 93, 94, 95], in scientific computing including chemistry, physics, and mechanics. The state-of-the-art studies focus on block-sparse tensor contractions with dense blocks in tensors. The conventional approaches first extract dense block-pairs of the two input tensors, and then perform multiplication by calling dense BLAS linear algebra. Finally, those approaches pre-allocate the output tensor using domain knowledge or a symbolic phase approach [96, 53, 97, 98, 99], such as TiledArray [53], Cyclops Tensor Framework [100], and libtensor [55, 56]. The state-of-the-art work Sparta focuses on element-wise sparse tensor contractions [25], solving the high dimensionality challenges through hash table-based approaches and addressing the unknown output tensor and irregular memory access challenges by dynamic allocation, permutation and sorting. Athena develops element-wise sparse tensor contraction by optimizing tensor summation as well, frequently occurred in contraction sequences.

Sparse tensor contraction sequences. Sparse tensor contraction often occurs as sequences in a spectrum of applications, such as quantum chemistry, quantum physics and deep learning [19, 20, 21, 22, 23]. Some existing work optimizes tensor computation sequences. AutoHOOT [106] decomposes a dense tensor contraction workload into task sequences and overlaps the computation and communication task sequences to reduce the communication overhead in a distributed execution. DLTC [88] takes input tensor computation sequences and generates optimized derivative sequences by automatic differentiation. TensorFlow [110] leverages a directed acyclic graph to represent the computation and data flow of tensor-based operator sequences and co-run the tensor-based operator sequences in an FIFO method. Athena is different from them in terms of leveraging the domain-knowledge of SpTC sequences to achieve

high performance.

Data management on heterogeneous memory systems. Heterogeneous memory management attracted plenty of research efforts in recent years [65, 61, 62, 60, 111, 112, 113]. These works explore various page-level data placement polices on HM based on main memory access profiling result. Thermostat [60] uses sampling-based profiling to track page table and migrates hot pages into DRAM. RAMinate [61], Heteros [62], Yan et al. [65] propose the state-of-the-art memory management solutions for general purpose which guides page placement based on an existing Linux page replacement mechanism. Application-specific HM management solutions [114, 115, 64, 63, 77, 116, 117, 118, 119, 120, 121, 122] leverage domain knowledge to further improve performance. MyNVM [115] proposes a software-managed multi-level caches policy to treat DRAM and NVM as caches for hard drives. Sparta [25] leverages application awareness and static data placement to avoid unnecessary data movement. Athena is different from these works in terms of exposing data semantics and dynamically managing data objects across SpTCs.

4.5 Summary

Efficiently computing sparse tensor contraction sequences (SpTCSeq) is critical to many applications. However, it is challenging, due to its redundant computation and memory operations, massive memory consumption, and inefficient utilization of hardware. In this paper, we explore solutions to address those challenges based on algorithm knowledge and characterization of workloads in SpTCSeq. We introduce Athena, a high performance framework for SpTC sequences. Athena is based on a set of novelty in data structures, runtime techniques, and emerging Optane-based memory architecture. Evaluating with 12 datasets, we show that Athena brings significant speedup (327-7362 $\times$) over the state-of-the-art SpTC algorithm. Athena also showcases its effectiveness in quantum chemistry and physics applications. For exascale problems deployed in a distributed environment, Athena could help to reduce the number of nodes needed for computation due to its capability to solve large sparse tensors on each single node.

Chapter 5

Processing-in-Memory for Energy-efficient Neural Network Training: A Heterogeneous Approach

5.1 Motivation

We motivate our software/hardware coordinated design by discussing the challenges of accelerating machine learning training workloads. We employ three widely used CNN training models – VGG-19 [123], AlexNet [124], and DCGAN [125] – as examples in this chapter. However, our observations can also be applied to various other training workloads (Chapter 5.4).

5.1.1 NN Training Characterization

In order to understand the characteristics of NN training workloads, we develop a profiling framework (Figure 5.1) by leveraging TensorBoard [126] and Intel VTune [127] to collect software and hardware counter information of training operations. Measuring the number of main memory accesses of individual operations during training can be inaccurate due to the extra cache misses imposed by simultaneously executing operations. As such, we disable inter-operation parallelism to ensure characterization accuracy of individual operations.

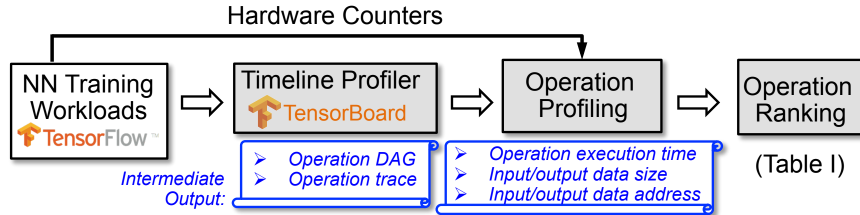


Figure 5.1: Our profiling framework for profiling NN training workloads in TensorFlow.

Execution Time	Memory Access %	Example Operations
Long	Low	*Conv2D* in VGG-19
Long	High	*Conv2DBackpropFilter* in VGG-19
Short	High	*Slice* in DCGAN
Short	Low	*Reshape* in AlexNet

Figure 5.2: Four categories of NN training operations.

Table 5.1 illustrates our profiling results of top five most time-consuming and memory-intensive operations, respectively, with three training models. Each model has tens of different types of operations and requires thousands of iterative steps to train; In each step, each type of operation can be invoked up to tens of times. We only show results within one training step. But the characteristics remain stable across training steps.

We make three key observations. First, only several operations dominate training execution time. For example, top five operations in VGG-19 model consume over 95% of total execution time. Second, the most time-consuming operations are also the most memory intensive. In fact, the top five most time-consuming operations contribute to over 98% of total main memory accesses across all three models. We further classify operations into four classes, shown in Figure 5.2. The first class of operations is compute intensive, and does not have to be offloaded to PIMs, but we can offload them when there are idling hardware units in PIMs. The second class of operations is our target to offload to PIMs. The third class is unusual, and the fourth class does not have big performance impact on model training. *The above two observations motivate us to adopt a PIM architecture to accelerate NN training in order to reduce data movement between the host processor and the main memory.*

Third, time-consuming and memory-intensive operations require heterogeneous computation types. It appears that many of such operations are multiplication and addition (e.g., MatMul) or can be decomposed so (e.g., Conv2D). This is inline with previous works on machine learning acceleration [128, 129]. Yet, significant

Table 5.1: Operation profiling results for three neural network models. “CI”= computation intensive; “MI”=memory intensive.

VGG-19					
Top 5 CI Ops	Execution Time(%)	#Invocation	Top 5 MI Ops	#Main Memory Access(%)	#Invocation
1. Conv2DBackpropFilter	40.15	16	1. Conv2DBackpropFilter	42.52	16
2. Conv2DBackpropInput	32.68	15	2. BiasAddGrad	35.68	16
3. BiasAddGrad	11.92	16	3. Conv2DBackpropInput	21.06	15
4. Conv2D	10.34	16	4. MaxPoolGrad	0.22	16
5. MaxPoolGrad	1.49	16	5. Relu	0.14	19
Other 13 ops	3.37	232	Other 13 ops	0.38	229
AlexNet					
Top 5 CI Ops	Execution Time(%)	#Invocation	Top 5 MI Ops	#Main Memory Access(%)	#Invocation
1. Conv2DBackpropFilter	33.64	5	1. BiasAddGrad	44.64	3
2. Conv2DBackpropInput	33.46	4	2. Conv2DBackpropInput	36.61	4
3. MatMul	13.54	6	3. Conv2DBackpropFilter	14.79	5
4. Conv2D	10.48	5	4. Relu	1.20	8
5. BiasAddGrad	4.62	3	5. Conv2D	0.46	5
Other 13 ops	4.26	121	Other 13 ops	2.30	119
DCGAN					
Top 5 CI Ops	Execution Time(%)	#Invocation	Top 5 MI Ops	#Main Memory Access(%)	#Invocation
1. Conv2DBackpropFilter	19.98	4	1. Conv2DBackpropFilter	37.21	4
2. Conv2DBackpropInput	17.18	4	2. Conv2DBackpropInput	28.09	4
3. MatMul	14.28	12	3. Slice	17.18	14
4. Conv2D	10.53	4	4. Conv2D	5.45	4
5. Mul	9.89	84	5. Mul	2.22	84
Other 47 ops	28.14	821	Other 47 ops	9.85	819

amount of top time-consuming and memory-intensive operations cannot simply be implemented by pure multiplication and addition. For instance, Relu is an activation function that incorporates conditional statement; MaxPool is a sample-based discretization process; ApplyAdam is a first-order gradient-based optimization of stochastic objective functions. Complex operations, such as Conv2DBackpropFilter and Conv2DBackpropInputs, include other logic and computations beyond multiplication and addition. Such non-multiply-add operations can consume over 40% of total execution time. Furthermore, studies on modern multi-tenancy [130] and multimodel training [131] workloads also demonstrate such heterogeneous computation requirement. *This observation motivates us to adopt a heterogeneous PIM architecture that combines fixed-function logic and programmable cores.*

Most previous works on PIM adopt either fixed-function [128] or programmable [129] computation components in the logic layer of 3D die-stacked memory. In the following, we discuss feasibility, challenges, opportunities of accelerating NN training with software/hardware co-design of heterogeneous PIM.

5.1.2 Software Design Challenges and Opportunities

There are three challenges for the software design: (1) How do we enable high productivity of system programmers and ease-of-adoption of PIM-based NN training accelerators? (2) How do we develop a unified programming model that can efficiently accommodate the host processor, fixed-function PIMs, and programmable PIMs? (3)

How do we balance hardware utilization at runtime?

One candidate baseline programming model is OpenCL [132], which is widely used in accelerator-based heterogeneous computing platforms (e.g., GPU and FPGA). We adopt OpenCL, due to its portability, expressiveness, and ability to enable high programming productivity to support programming on heterogeneous systems (details are discussed in Chapter 5.2.2). However, it is not straightforward to adopt OpenCL for NN model training on the heterogeneous PIM architecture. (1) How do we map the platform model of OpenCL to the heterogeneous PIM architecture? (2) Given the execution model of OpenCL with limited considerations on hardware utilization, how do we make the best use of CPU (the host processor) and different types of PIMs? (3) Given the memory model of OpenCL with limited considerations on synchronization between hardware units, how do we meet the requirement of frequent synchronizations from NN operations?

Trade-offs between parallelism and programmability. Fixed-function PIMs typically offer high computation parallelism by executing fine-grained, simple operations distributed across massive amount of logic units. However, they are less flexible than programmable PIMs that can be programmed to accommodate a large variety of operations. Furthermore, fixed-function PIMs can impose high performance overhead by (i) frequent operation-spawning and (ii) host-PIM synchronization. Programmable PIMs typically execute coarse-grained code blocks with less frequent host-PIM synchronization. However, the limited number of computational units in programmable PIMs can lead to much lower parallelism than in fixed-function PIMs.

Opportunities in runtime system scheduling. Substantial opportunities exist in leveraging system-level software to optimize resource sharing among various system components. The heterogeneity of our architecture introduces requirements on scheduling model-training operations across the host processor (CPU), fixed-function PIMs and programmable PIMs, based on the dynamic utilization of compute resources on these system components. Yet, we observe that NN training workloads tend to have repeatable (hence predictable) computation behavior over the execution time. As such, system software can accurately predict and dynamically schedule the operations by profiling the resource utilization of various compute elements in the first few steps of modeling training. Such dynamic profiling-based scheduling can achieve the best utilization of computation resources, while improving energy efficiency.

5.1.3 CPU vs. GPU – Where to Attach Heterogeneous PIMs?

Today, NN-training workloads can be executed on both CPU- and GPU-based systems. Recent silicon interposer technology allows both types of systems to adopt 3D die-stacked memories closely integrated with logic components. For example, modern GPU device memories [133] are implemented by high-bandwidth memory technology. High-end CPU servers integrate high-bandwidth memories using the DRAM technology adopted from hybrid memory cubes.

Our heterogeneous PIMs are logic components closely integrated with die-stacked memories. Therefore, they are generally applicable to both CPU or GPU systems. However, this work focuses on the software design for heterogeneous PIMs attached on CPU systems, due to the constraint of current GPU systems. Today, GPU systems often fuse and organize computation kernels into NN layers rather than fine-grained operations, because of the inefficiency of compute preemption and thread scheduling. This significantly limits the flexibility of operation scheduling on GPU.

The NVIDIA Volta GPU provides certain support for fine-grained acceleration of NN training operations, yet only available with limited number of threads. Modern CPU systems are easy to access and program; this enables easy-to-adopt and flexible programming abstraction and system library functions.

5.2 Design

To address the aforementioned challenges, we propose a software/hardware co-design of heterogeneous PIM framework to accelerate NN training. Our design consists of a heterogeneous PIM architecture, an extended OpenCL programming model, and a runtime system. Figure 5.3 depicts our architecture configuration. Figure 5.4 shows the process of building and executing NN training with our software framework. Given an OpenCL kernel to implement an operation, our system extracts code sections from the kernel and compiles them into a set of binaries to run on CPU, programmable PIM, and fixed-function PIMs, respectively. After the training workload starts to execute, our runtime scheduler profiles the first step of training to obtain operation characterization. It then performs dynamic scheduling of operations across CPU, programmable PIM, and fixed-function PIMs in the rest of training steps. Our runtime system incorporates two key components: (i) an operation-pipeline scheme, which

allows multiple NN operations to co-run on PIMs to improve hardware utilization and (ii) a recursive operation-execution scheme, which allows the programmable PIM to call fixed-function PIMs to improve hardware utilization and avoid frequent synchronization between CPU and PIMs.

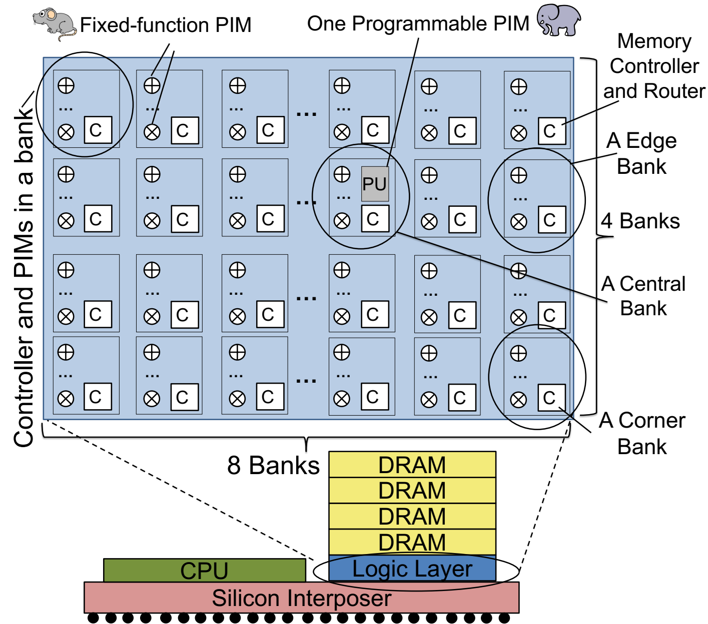
Software/hardware co-design principles. Our software design supports our hardware configuration in the following manner. First, our software design offers a portable programming model across the host processor, fixed-function PIMs, and the programmable PIM. Our programming model provides a unified abstract to program various PIMs, which need to be programmed in separate manners in conventional systems. Our runtime scheduling scheme effectively optimizes PIM hardware utilization. Our runtime system also enables recursive calls between the programmable PIM and fixed-function PIMs. Our architecture design supports our software design in two ways: our heterogeneous PIM architecture enables efficient NN training acceleration by exploiting the heterogeneous characteristics of software operations; We employ a set of hardware registers to track PIM hardware utilization information, which is required by our runtime scheduling.

5.2.1 Heterogeneous PIM Architecture

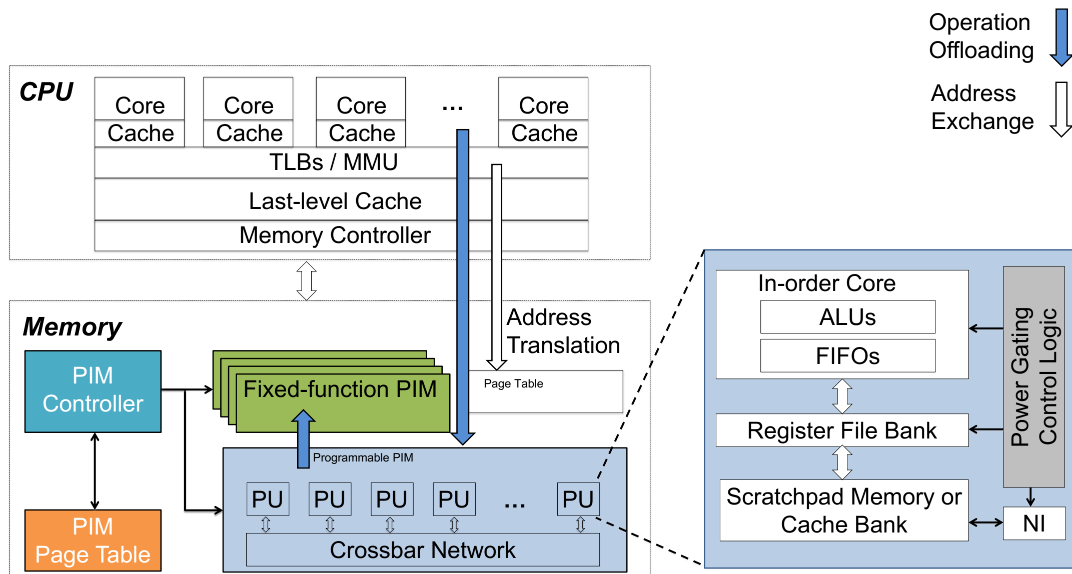
To accommodate various types of operations that are likely to execute on PIMs, we adopt a heterogeneous PIM architecture consisting of (i) a programmable PIM, which is an ARM core and (ii) massive fixed-function PIMs, which are adders and multipliers distributed across all memory banks. While our design can be used with various 3D die-stacked memory devices, we employ a 32-bank memory stack (where a bank is a vertical slice in the stack) as an example in this work. Figure 5.3 depicts our architecture configuration. Chapter 5.2.4 describes hardware implementation details.

5.2.2 Programming Model for Heterogeneous PIM

We extend the OpenCL programming model to program the heterogeneous PIM. OpenCL has been widely employed to enable program portability across accelerator-based, heterogeneous computing platforms (e.g., GPU and FPGA). We use OpenCL because of the following reasons. First, by treating the fixed-function PIMs and programmable PIM as accelerators, the semantics of OpenCL naturally fit into the heterogeneous PIM environment. Second, writing a program for the heterogeneous



(a) Heterogeneous PIM architecture overview.



(b) Operation offloading between CPU and PIM. (c) Architecture of programmable PIM.

Figure 5.3: Architecture overview of the proposed heterogeneous PIM.

PIM based on an abstract and unified hardware model in OpenCL, the programmer can write the program just once but run it on a variety of PIMs. Therefore, by using OpenCL, we can hide hardware variety of the heterogeneous PIM from system programmers, improve their productivity, and enable code portability.

Other programming models, such as OpenACC [134, 135] and OpenMP [136],

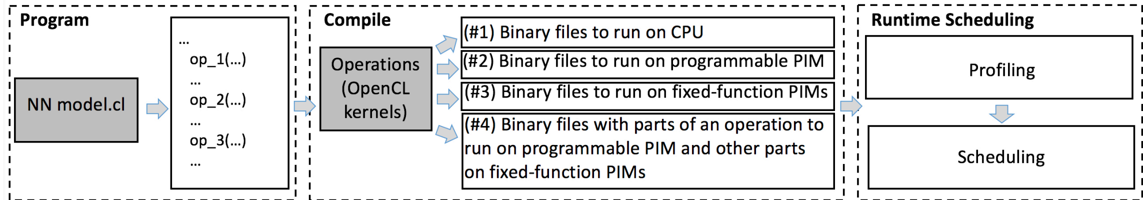


Figure 5.4: The process of executing NN training with our software framework design.

can also hide hardware heterogeneity and reduce programmers' burden. However, these are higher-level programming models, which rely on compilers to transform programs into a lower-level programming model, such as OpenCL, to enable code portability. We focus on OpenCL in our study, because it provides a foundation for those higher-level programming models.

Overview of our programming model. Table 5.2 summarizes our extension to OpenCL. Our platform model includes multiple types of heterogeneous devices. Such platform model is driven by the characteristics of NN training operations. Our execution model adds (i) recursive kernel invocation to enable kernel invocation between PIMs to support complex NN operations (e.g., Conv2DBackpropFilter) and (ii) operation pipeline to improve hardware utilization for small NN operations with limited parallelism (e.g., Slice). Finally, we extend the memory model to support a single global memory shared between the host processor and accelerators. We also add explicit synchronization across different PIMs and CPU (host processor) to enforce execution orders across NN operations.

OpenCL background. The existing OpenCL adopts a host-accelerator platform model as shown in Figure 5.5(a). A host processor connects to one or more compute devices (i.e., accelerators). A compute device is divided into one or more compute units, each of which is further divided into one or more processing elements (PE). An OpenCL program consists of kernels for compute devices and a host program. The host program runs on CPU and enqueues commands to a command-queue attached to a compute device.

In order to employ OpenCL programming model on the heterogeneous PIM system, we investigate how to map the heterogeneous PIM system onto the OpenCL model, and extend the OpenCL model for efficient runtime scheduling. In the following, we discuss our mapping method from the perspectives of platform model, execution model, and memory model. Table 5.2 summarizes our programming model extension.

Table 5.2: Extending OpenCL for the heterogeneous PIM.

	Native OpenCL	Extensions for Heterogeneous PIM
Platform model	Host + accelerators (e.g., host + GPU).	Host + two types of accelerators (fixed-function PIMs and programmable PIM) driven by the characteristics of NN training.
Execution model	Host submits work to accelerators.	• Host submits work to accelerators; • Accelerators submit work to accelerators (i.e., recursive kernel invocation); • Work execution pipeline (i.e., operation pipeline); • Work scheduling based on dynamic profiling.
Memory model	• Multiple types of memory with a relaxed consistency model; • The global memory is not shared; • No defined synchronization across accelerators.	• A single global memory with a relaxed consistency model; • The global memory is shared; • Explicit synchronization across PIMs and CPU.

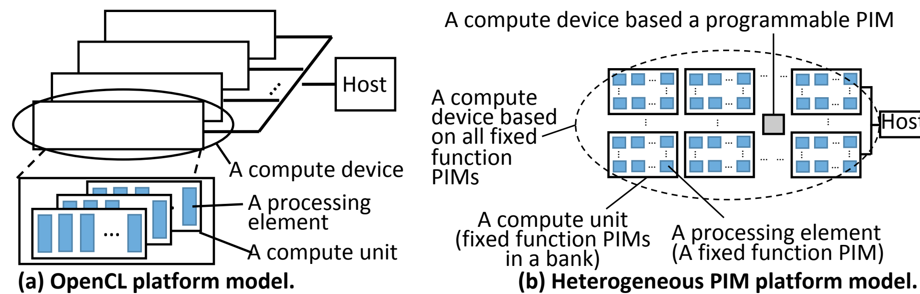


Figure 5.5: Enabling OpenCL platform model on heterogeneous PIM systems.

Heterogeneous PIM platform model. Figure 5.5(b) illustrates our platform model. A large number of fixed-function PIMs provide massive parallelism for data processing. Each fixed-function PIM is a PE (in the OpenCL jargon). All fixed-function PIMs in all memory banks form a compute device. All fixed-function PIMs in a bank form a compute unit. Each programmable PIM is a compute device; each core of the programmable PIM is a PE. Hence, within the context of OpenCL, a heterogeneous PIM system has heterogeneous compute devices. We expose fixed-function PIM and programmable PIM as distinct compute devices to give control flexibility to the runtime system for operation scheduling. An OpenCL operation can be offloaded to any compute device that supports the operation execution.

Execution model. Tasks (i.e., operations in NN model training) to be launched on any PIM are represented as kernels managed by a host program, as in a traditional OpenCL program. If the task includes instructions that cannot be executed on the fixed-function PIM, then the task will not be scheduled by the OpenCL runtime to run on the fixed-function PIM. Otherwise, a task can run anywhere (CPU, fixed-function

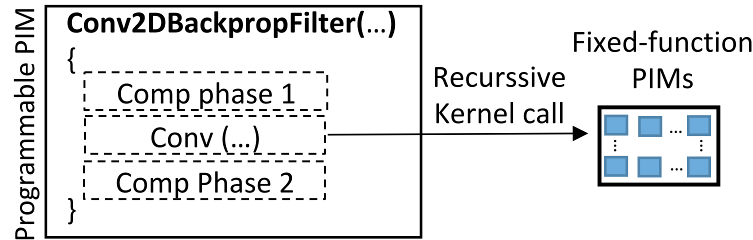


Figure 5.6: An example of the recursive PIM kernel.

PIM, and programmable PIM). The OpenCL runtime (on CPU) is in charge of task scheduling between different PIMs and CPU. Leveraging low-level APIs (Chapter 5.2.4) and hardware registers, the runtime can determine whether a specific PIM is busy and whether a specific task is completed. We describe the scheduling algorithm in Chapter 5.2.3. Binary files for a task to run on CPU, fixed-function PIM, or programmable PIM are generated during the compilation stage. Given an OpenCL kernel for a task, we generate four binary files as shown in Figure 5.4. Chapter 5.2.4 discusses details of binary generation.

Binaries (#3) and (#4) in Figure 5.4 allow *recursive PIM kernel*, a new execution scheme for our heterogeneous PIM design. A kernel in the programmable PIM can trigger data processing with fixed-function PIMs. This is supported by the programmable PIM runtime and implemented by calling small kernels loadable on fixed-function PIMs. By combining multiple kernels into a single kernel, the recursive PIM kernel scheme reduces overhead of kernel spawning and synchronization between the host and PIMs. Figure 5.6 shows an example that further explains the recursive PIM kernel. In the example, we illustrate an NN operation, Conv2DBackpropFilter, which is offloaded to the programmable PIM as a kernel; the kernel includes computation phases 1 and 2 that cannot be offloaded to the fixed-function PIMs. Conv2DBackpropFilter includes convolution computation ("Conv(...)" in the figure); The programmable PIM offloads this portion of computation to fixed-function PIM as a smaller kernel. The computation phases 1, 2 and convolution are combined as a single recursive PIM kernel, which reduces the synchronization between CPU and PIMs.

In general, the four binary files provide convenience for scheduling on CPU, the fixed-function PIMs and programmable PIM, and hence allows the runtime to maximize utilization of CPU and PIMs.

Memory model. The existing OpenCL defines four distinct memory regions in a

compute device: global, constant, local, and private. On a heterogeneous PIM system, only a single global memory (i.e., the main memory) exists. In addition, the global memory is shared between CPU and PIMs, and addressed within a unified physical address space. This memory model requires synchronization at multiple points: (1) between CPU and PIMs; and (2) between different PIMs. The synchronization is necessary to avoid data race and schedule operations.

To implement effective synchronization, we employ the programmable PIM to drive the synchronization and avoid frequent interrupts to CPU. In particular, for synchronization between CPU and PIMs, the programmable PIM checks the completion of operations offloaded to PIMs (either programmable or fix function PIMs) and sends the completion information to CPU. For synchronization between different PIMs, the programmable and fix function PIMs synchronize through global variables in main memory.

Between CPU and PIMs, we introduce explicit synchronization points to synchronize the accesses to shared variables. To the host processor, the whole set of fixed-function PIMs or the programmable PIM appear as another processor. We employ standard synchronization schemes (e.g., barriers and locks), similar to the ones in a shared-memory multiprocessor. For fixed-function PIMs, their operations are atomic and the synchronization points are not expected in the middle of operations. For programmable PIMs, the synchronization points can be in the middle of a kernel. This is feasible based on global lock variables shared between CPU and PIMs. To support memory consistency, we adopt a relaxed memory consistency model, which aims to improve performance and reduce hardware complexity. In particular, an update to a memory location by a fixed-function PIM is not visible to all the other fixed-function PIMs at the same time. Instead, the local view of memory from each fixed-function PIM is only guaranteed to be consistent right after the kernel call to fixed-function PIMs. Between the fixed-function PIMs and programmable PIM, we employ the same consistency scheme: updates to memory locations by the entire set of fixed-function PIMs are not visible until the end of the kernel call to the fixed-function PIMs.

Because of our shared memory model, there is no data copy overhead before and after PIM kernel calls. Based on the above synchronization schemes, PIM kernel calls can be launched asynchronously to overlap computation on CPU and PIMs.

Support for easy program maintenance. To use the extended OpenCL programming model, operations need to be re-written using OpenCL. To write OpenCL code for operations, one can use OpenACC directives and compilers [134, 135] to automatically transform the original code into OpenCL code. This can significantly simplify the programming work. Furthermore, the number of operations for machine learning models is limited (tens of operations). Hence, using OpenCL to implement those machine learning operations is feasible. Other than that, however, the higher level software components (e.g., most of the middleware components, operation APIs, and Python syntax for using machine learning models) remain the same. This enables easy maintenance of machine learning frameworks.

5.2.3 Runtime System Design

Our runtime system is in charge of scheduling operations to fixed-function PIMs, programmable PIM, and CPU. To minimize NN training time, the runtime strives to maximize utilization of PIMs and CPU to optimize system throughput. The runtime schedules operations based on the following two steps.

Step 1: profiling. The runtime profiles performance of all operations on CPU. The profiling happens in only one step of NN model training. NN model training typically has a large amount of iterative steps (thousands and even millions of steps). Using one step for profiling has ignorable impact on performance. In addition, all steps almost have the same classes of operations; performance of operations (particularly execution time and the number of main memory access) remains stable across steps. Therefore, one step is sufficient for profiling purpose. During profiling, the runtime executes operations one by one in CPU, collecting execution time and the number of main memory access level cache misses of each operation with hardware counters. Based on the profiling results in the step, the runtime employs the following algorithm to determine the candidate operations to be offloaded to PIMs.

To determine the candidate operations, the runtime sorts operations into two lists (in descending order) based on execution time and the number of main memory accesses, respectively. Each operation in each of the two lists is correlated to an index, i.e., each operation has two indexes. With each operation, the runtime calculates a global index by adding these two indexes. Based on the global indexes, the runtime sorts operations into a global list. The runtime chooses top operations in the global

Table 5.3: Low-level APIs for PIMs.

Name	Description
int pim_fix(int* pim_ids, void* args, void* ret, size_t num_pim)	Asks specific fixed-function PIMs to work with input arguments `args` and return results `ret` and a work ID.
int pim_prog(int pim_id, pim_program kernel, void* args, int* args_offset, void* ret, size_t ret_size)	Asks a programmable PIM to work on a kernel (an operation) and return a work ID.
int pim_status(int pim_id)	Checks whether a specific PIM is busy.
int work_query(int work_id)	Checks whether a specific operation is completed.
void work_info(int work_id, int* pim_ids, int* data_loc)	Queries the computation location (pim_ids) and input/output data location (i.e, which DRAM banks) for a specific operation.

list to offload to PIMs. Those top operations account for x% of total execution time of one step (x = 90 in our evaluation). The above algorithm is inspired by feature selection process in machine learning [137]. The goal of this algorithm is to select those operations that are both time-consuming and have a large number of main memory accesses.

Step 2: scheduling. Given the candidate operations to offload, the runtime makes the scheduling decision based on the following three principles.

- Scheduling operations to execute on fixed-function PIMs as much as possible.
- Scheduling operations to execute on PIMs (not CPU) as much as possible. In case all fixed-function or programmable PIMs are busy, the runtime will schedule the candidate operations to execute on CPU;
- Scheduling needs to respect data dependency across operations.

The first principle favors fixed-function PIMs over other compute units, because fixed-function PIMs are more energy efficient and typically performs faster with higher parallelism than other compute units. The second principle avoids CPU idling and introduces parallelism between CPU and PIMs. The third principle ensures execution correctness. Each operation defined in the machine learning frameworks typically has explicit input and output data objects (e.g., Tensors in TensorFlow), which provides convenience in tracking data dependencies across operations.

Operation pipeline. The above scheduling algorithm and principles enable operation pipeline to maximize hardware utilization. In particular, when an operation in a step cannot fully utilize fixed-function PIMs, our runtime schedules an operation in the next step to execute a portion of its computation by utilizing idling fixed-function PIMs as long as the two operations do not depend on each other.

In essence, these two operations can enable a pipelined execution manner. For

instance, in AlexNet, a single convolution operation with a filter size of 11×11 consumes 121 multiplication and 120 addition (241 fixed-function PIMs in total). In case we have 444 fixed-function PIMs in total (Chapter 5.2.4), the utilization of fixed-function PIMs is only 54%. To improve hardware utilization, the runtime can schedule multiplication and addition from an operation (or operations) in the next step to execute on fixed-function PIMs. Once the convolution operation in the current step is completed, the partially executed operation(s) from the next step can immediately utilize the newly released fixed-function PIMs to improve hardware utilization and performance. This indicates that an operation can dynamically change its usage of PIMs, depending on the availability of PIMs. Such dynamic nature of operation execution is feasible based on a runtime system running on the programmable PIM (Chapter 5.2.4 presents implementation details).

5.2.4 Implementation

Low-level APIs for PIM Runtime System

We introduce several low-level API functions for fixed-function and programmable PIMs. These API functions allow direct control of individual PIMs, and provide foundation for our runtime. The API achieves the following functionality: (1) offloading a specific operation into specific PIM(s); (2) tracking the status of PIMs, including examining whether a PIM is busy or not; (3) querying the completion of a specific operation; (4) querying the computation location (i.e., which PIM) and input/output data location (i.e, which DRAM banks) for a specific operation. Table 5.3 summarizes our API functions.

OpenCL Binary Generation

To schedule operations to execute on CPU, fixed-function PIMs, or programmable PIM, we generate four binary files (Figure 5.4). In order to generate the binary file (#3) that corresponds to a portion of a large operation (an OpenCL kernel) to execute on fixed-function PIMs (e.g., the convolution within the operation Conv2DBackpropFilter), we first extract code sections from the corresponding OpenCL kernel. We then transform these code sections into a set of small kernels to execute on fixed-function PIMs. Finally we compile them into binary file (#3). In the original OpenCL kernel, these extracted code regions are replaced with the kernel calls and then compiled into

binary file (#4) to execute on the programmable PIM. Binary files (#1) and (#2) are generated during the regular compilation stage.

Runtime Implementation

Our runtime consists of two components, which execute on the CPU and the programmable PIM, respectively.

The runtime on CPU. To support our runtime scheduling, we extend the runtime system of TensorFlow by adding approximately 2000 lines of code. The runtime on CPU schedules operations on CPU and PIMs, based on hardware utilization information provided by the low-level APIs. It does not support the implementation of recursive PIM kernels. In other words, the runtime on CPU is only responsible for offloading a kernel – which can have a part of its computation offloadable to fixed-function PIMs – to the programmable PIM. Our modifications to TensorFlow runtime include (1) device initialization and characterization using OpenCL intrinsics; (2) creating a device context and instance for a PIM device; (3) providing a new OpenCL device abstraction to other components of Tensorflow; (4) a mechanism to communicate with the runtime on the programmable PIM. This is one-time modification to Tensorflow, but can support various PIM hardware configurations without involving system programmers' future efforts.

The runtime on programmable PIM. The runtime on the programmable PIM supports recursive PIM kernels and operation pipeline. In particular, a kernel with a part of its computation replaced with kernel calls to fixed-function PIMs is handled by the runtime on the programmable PIM, which automatically offloads the computation to fixed-function PIMs. In order to keep track of the dynamic utilization of fixed-function PIMs, our runtime on the programmable PIM records the numbers of additions and multiplications already completed in each operation offloaded to the programmable PIM, as well as the remaining additions and multiplications.

Hardware Implementation

Figure 5.3 and Figure 5.7 illustrate our hardware implementation. The programmable PIM employs an ARM Cortex-A9 processor with four 2GHz cores. Each core has an in-order pipeline. In individual NN training models, operations that are potentially offloaded to the programmable PIM (e.g., ApplyAdam, MaxPooling, and ReLU) are typically not executed at the same time. Therefore, we only adopt

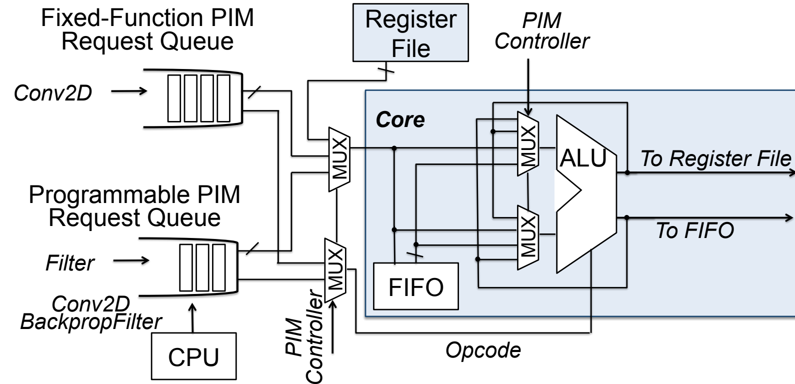


Figure 5.7: Heterogeneous PIM implementation.

one programmable PIM in our design. Even if we simultaneously train multiple NN models, the chance of having multiple operations to use the programmable PIM at the same time is low according to our evaluation with mixed workload analysis.

Because a significant portion of NN training operations can be decomposed to addition and multiplications, we implement our fixed-function PIMs as 32-bit floating point multipliers and adders. We implement equal numbers of multipliers and adders in the pool of fixed-function PIMs. Our low-level APIs allow us to map operations to fixed-function PIMs that are in the same bank as input data of the operations. In addition, we accommodate random memory access pattern in NN computation by adopting buffering mechanisms [128]. We determine the fixed-function PIM configurations by employing a set of architectural level area, power, and thermal modeling tools, including McPAT [138] and HotSpot [139], to perform design space exploration of the logic die of 3D DRAM. Based on our study, the total number of allowed fixed-function PIMs is limited by the area of the logic die. With our baseline 3D DRAM configuration, we can distribute 444 fixed-function PIMs (pairs of multipliers and adders) across the 32 banks in the logic die. It is impossible to distribute these fixed-function PIMs evenly to each bank. We consider the placement of the fixed-function PIMs on 32 banks based on the following policy: we place more fixed-function PIMs on edge and corner banks than on central banks (Figure 5.3 (a)). The rationale behind is that the banks at the edge and corner have better thermal dissipation paths than central banks. Therefore, these banks can support higher computation density.

Furthermore, we employ a set of registers as shown in Figure 5.7. Each register

Table 5.4: System configurations.

CPU	Intel Xeon E5-2630 V3@2.4GHz
Main memory	16GB DDR4
Operating system	Ubuntu 16.04.2
GPU	NVIDIA GeForce GTX 1080 Ti (Pascal)
GPU cores	28 SMs, 128 CUDA cores per SM, 1.5GHz
L1 cache	24KB per SM
L2 cache	4096KB
Memory interface	8 memory controllers, 352-bit bus width
GPU main memory	11GB GDDR5X

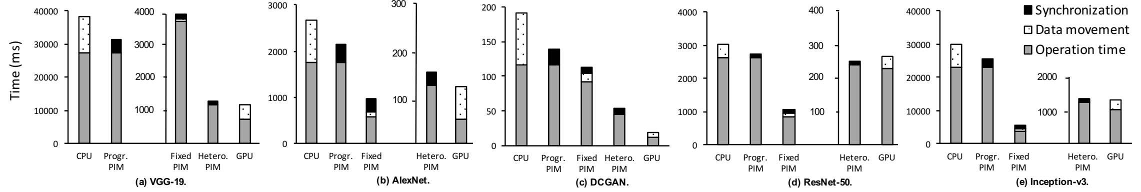


Figure 5.8: Execution time breakdown of five NN models.

indicates the idling of either a bank of fixed-function PIMs or the programmable PIM. The registers allow our software runtime scheduler to query the completion of any computation and decide the idleness of processing units.

5.3 Experimental Setup

5.3.1 Simulation Framework

In order to evaluate the performance of our design, we model fixed-function PIM and programmable PIM architectures, respectively, using Synopsys Design Compiler [140] and PrimeTime [141] with Verilog HDL. We adopt HMC 2.0 [142] timing parameters and configurations for our evaluation of 3D memory stack. Baseline memory frequency is set to 312.5 MHz, which is the same as HMC 2.0 specification [142]. This is also used as the working frequency of our heterogeneous PIM. We employ a trace generator developed on Pin [143] to collect instruction trace, when running our OpenCL kernel binaries on CPU. We develop a python-based, trace-driven simulation framework based on our design to evaluate the execution time of various training workload traces. Our simulator also incorporates our runtime scheduling mechanisms.

5.3.2 Power and Area Modeling

We adopt 10nm technology node for the host CPU and the logic die of the PIMs; 25nm technology node for the DRAM dies. We measure CPU and GPU power with VTune [144] and *nvidia-smi*, respectively. Our power model considers whole system power when we evaluate the power of heterogeneous-PIM-based systems, including CPU and the memory stack. We calculate the power and area of the programmable PIM using McPAT [138]. We evaluate the power and area of fixed-function PIMs using Synopsys Design Compiler [140] and PrimeTime [141].

5.3.3 Workloads

We evaluate various training models, including VGG-19 [123], AlexNet [124], Deep Convolutional Generative Adversarial Networks (DCGAN)) [125], ResNet-50 [28], Inception-v3 [145], Long Short Term Memory (LSTM) with dropout [146] and Word2vec [147]. LSTM and Word2vec are evaluated in Section 5.4.6. The rest models are widely used in recent studies on CNN training and image classification.

Training Datasets. We employ ImageNet as training data set of VGG-19, AlexNet, ResNet-50, and Inception-V3. ImageNet is a large image dataset with millions of images belonging to thousands of categories. DCGAN employs MNIST dataset [148]. LSTM adopts Penn Tree Bank (PTB) [146] dataset. Word2vec employs "questions-words" dataset [149] in TensorFlow.

Training framework and batch Size. We adopt TensorFlow [110] as our training framework. We adopt default batch sizes of each training model in TensorFlow. The batch size of VGG-19, AlexNet and Inception-v3 is 32. The batch size of Word2vec and ResNet-50 is 128. DCGAN has a batch size of 64. LSTM employs a batch size of 20.

5.3.4 Real Machine Configurations

To compare performance and energy efficiency of heterogeneous PIM with GPU and CPU, we run the training models on (1) NVIDIA GeForce GTX 1080 Ti graphic card [150] and (2) CPU listed in Table 5.4. Our GPU-based training evaluations adopt CUDA 8 [151] and NVIDIA cuDNN 6.0 library [152]. GPU utilizations of each training model in TensorFlow are: Inception-v3 (average: 62%; peak: 100%); ResNet-50 (average: 44%; peak: 58%); AlexNet (average: 30%; peak: 34%); VGG-19

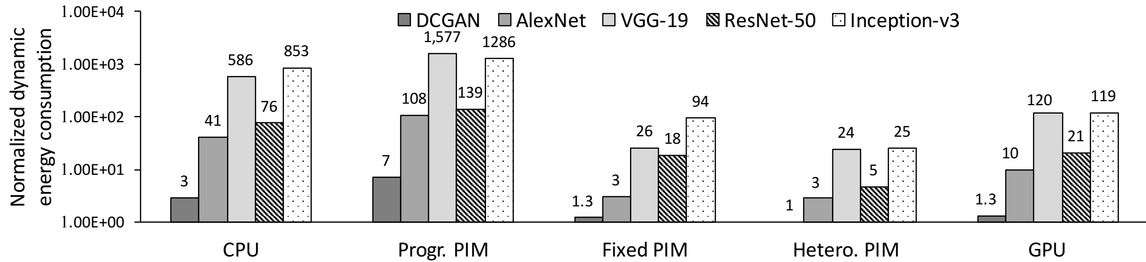


Figure 5.9: Normalized dynamic energy of various NN models.

(average: 63%; peak: 84%); DCGAN (average: 28%; peak 42%. We use NVIDIA's profiling tool [153] and Intel's VTune [127] to collect performance and power statistics.

5.4 Evaluation

Our experiments compare among the following five configurations, including our design.

- **CPU** – Executing all training operations on CPU;
- **GPU** – Executing all training operations on GPU;
- **Progr PIM** – Programmable PIMs only, which executes all operations on as many ARM-based programmable cores as needed by workloads (without our runtime scheduling);
- **Fixed PIM** – Fixed-function PIMs only, which executes the operations that can be offloaded on fixed-function PIM and other operations on CPU (without our runtime scheduling);
- **Hetero PIM** – Our heterogeneous PIM design (including our runtime scheduling).

5.4.1 Execution Time Analysis

Figure 5.8 shows execution (training) time of various NN training models. We break down the execution time into synchronization time, data movement time and operation time (i.e., computation time in CPU, GPU or PIMs). For GPU-based systems, the data movement time is the time for data transfer between main memory and GPU global memory. Certain amount of data transfer time is overlapped with GPU computation, e.g. by copying a minibatch of images to the GPU memory, while

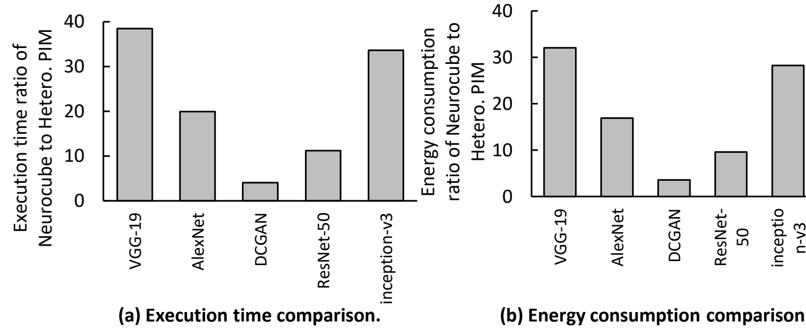


Figure 5.10: Performance and energy comparison with Neurocube.

the computation on GPU is processing another minibatch. Our breakdown only shows the data transfer time that is not hidden by the computation. For PIM-based systems, the data movement time is the time for data transfer between CPU and the main memory. Our runtime scheduling allows operations to execute concurrently on CPU and PIMs.

We observe that PIM-based designs (including Fixed PIM, Progr PIM and Hetero PIM) perform much better than CPU, with 19%-28× performance improvement. Compared with Progr PIM and Fixed PIM, our design has 2.5×-23× and 1.4×-5.7× performance improvement, respectively. PIM-based designs also significantly reduce data movement overhead, compared to CPU and GPU. Overall, Hetero PIM leads to the lowest synchronization and data movement overhead among all configurations.

The performance benefit of Hetero PIM stands out with larger training models and larger working sets due to (i) more reduction in data movement and (ii) higher parallelism between host CPU and PIMs introduced by more offloadable operations. DCGAN has smaller model and working set than others. Therefore, Hetero PIM appears to result in worse performance than GPU with DCGAN; yet, compared with other configurations, our design still significantly improves performance. ResNet is a large training model with large working sets. As a result, Hetero PIM leads to better performance than GPU with ResNet. With other training models, Hetero PIM leads to performance close to (within 10% of) GPU. GPU has good performance because of its massive thread-level parallelism. Our design leads to much better performance than all other configurations.

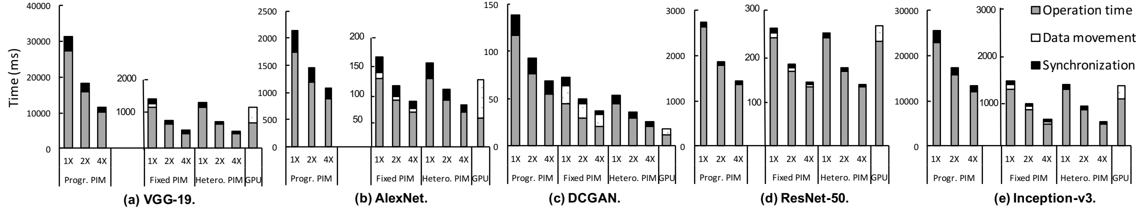


Figure 5.11: Execution time breakdown of various NN models with 3D memory frequency scaling.

5.4.2 Energy Consumption Analysis

Figure 5.9 shows the dynamic energy consumption of the five NN models with five different configurations. The energy consumption results are normalized to the results of Hetero PIM. We observe substantial energy benefit of using our design: it consumes 3×-24× and 1.3×-5× less energy than CPU and GPU, respectively. CPU consumes higher dynamic energy than Hetero PIM, Fixed PIMs, and GPU, even though its power consumption is the lowest among all of these configurations (note that we take CPU power into account when we calculate the power of PIMs and GPU, in order to evaluate full-system power consumption). This is because CPU has the longest execution (training) time. Furthermore, we notice that the dynamic energy consumption of Progr PIM is higher than that of other configurations, because the speed of Progr PIM is only slightly faster than that of CPU, yet the dynamic power of Progr PIM is higher than that of CPU due to the additional processing units in Progr PIM. Overall, Hetero PIM leads to the lowest dynamic energy consumption across all configurations.

5.4.3 Comparison with Prior PIM-based NN Acceleration

Figure 5.10 shows a quantitative comparison between our design and a recent PIM-based NN accelerator design, Neurocube [129] (qualitative comparison is in Chapter 5.5). Neurocube also reduces data movement overhead and improves energy efficiency by using PIM technology. However, our work outperforms Neurocube in terms of performance and energy efficiency. With highly compute-intensive models, such as VGG-19 and Inception-V3, our design achieves much higher performance and energy-efficiency improvement than Neurocube. Even with less compute-intensive models, such as DCGAN, our work can achieve at least 3× higher performance and

energy efficiency than Neurocube. The reason for the improvement is two-fold: (1) Neurocube only adopts programmable PIMs, while our design employs energy-efficient, highly-parallel fixed-function PIMs to accelerate fine-grained operations; (2) Our design employs runtime scheduling that effectively optimizes hardware utilization (evaluated in Section 5.4.5).

5.4.4 Sensitivity Study

Frequency Scaling. We adopt three different frequencies for fixed-function PIMs and programmable PIM: their original frequencies (1×), doubling of their frequencies (2×) and quadrupling of their frequencies (4×). We use a phase-locked loop module to change the frequency. We study execution (training) time with the different frequencies.

Figure 5.11 shows the results. We observe that with higher frequency, the heterogeneous PIM performs better than GPU. With 2× frequency, Hetero PIM performs 36% and 17% better than GPU, with VGG-19 and AlexNet, respectively. With 4× frequency, Hetero PIM performs 37% and 60% better than GPU, with VGG-19 and AlexNet respectively. We also observe that the synchronization and data movement overheads are reduced, when using higher frequencies.

Programmable PIM Scaling. We employ three different configurations for Hetero PIM, while keeping the area of logic die in the memory stack unchanged. We scale the number of Progr PIM (ARM cores) from one to two to 16, while the rest of the logic die area is used to implement Fixed PIM. The three configurations are labeled as *1P*, *4P* and *16P*, respectively.

Figure 5.12 shows our results. The figure reveals that the performance difference between the three configurations is relatively small. The performance difference between *16P* and *1P* is 12%–14%. The reason is two-fold: (1) One Progr PIM is sufficient for the NN models to schedule and pipeline operations; (2) Using more Progr PIMs loses more Fixed PIMs, given the constant area in the logic layer of memory stacks.

5.4.5 Evaluation of Software Impact

We isolate the the impact of our software (runtime) techniques from that of Hetero PIM hardware. We aim to provide more insightful analysis on the effectiveness

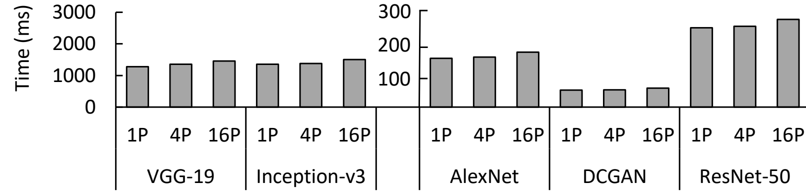


Figure 5.12: Execution time with Progr PIM scaling.

of software/hardware co-design. In particular, we study execution time, energy and utilization of Fixed PIM with and without the recursive PIM kernel call (RC) and operation pipeline (OP) – our two major runtime techniques. *Without RC and OP, we also compare Hetero PIM hardware design with Fixed PIM and Progr PIM, in terms of execution time and energy. This comparison allows us to study the impact of Hetero PIM architecture with the absence of our runtime techniques.*

Execution time analysis. As shown in Figure 5.13, Hetero PIM without runtime scheduling performs better than Progr PIM and Fixed PIM by up to 8.5×. This demonstrates the necessity of using Hetero PIM architecture. However, comparing with Fixed PIM, the performance benefit of Hetero PIM hardware is not significant (7%-30%). After incorporating the runtime scheduling techniques, the performance of Hetero PIM is improved by up to 3.8×. This result demonstrates the necessity of using an efficient runtime to maximize the benefit of Hetero PIM architecture.

Energy analysis. Figure 5.14 shows our energy results normalized to the energy of Hetero PIM with RC and OP. We have similar observations as the execution time analysis: Hetero PIM without runtime scheduling performs better than Progr PIM and Fixed PIM by up to 2.7×. With RC and OP, we further reduce the energy of Hetero PIM by up to 3.9×.

PIM utilization analysis. Figure 5.15 shows our utilization results. With RC only, the utilization of Fixed PIM in Hetero PIM is improved by up to 66% (VGG-19); With OP, the utilization of Fixed PIM is further improved by up to 18% (AlexNet); With RC and OP, the utilization of Fixed PIM is close to 100%. The reason for the poor hardware utilization with neither RC nor OP is the lack of scheduling for the operations that do not have sufficient parallelism or cannot be completely offloaded to Fixed PIM.

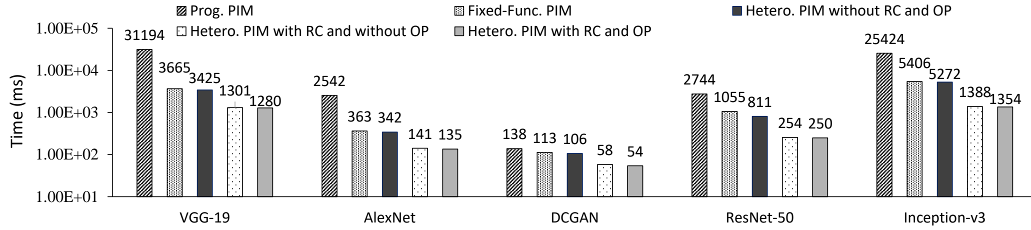


Figure 5.13: Execution time with and without RC and OP.

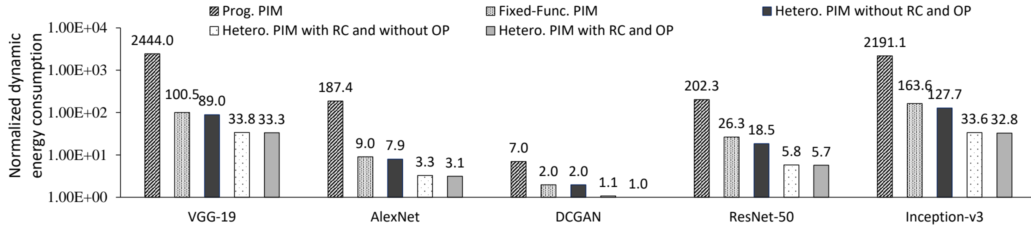


Figure 5.14: Dynamic energy with and without RC and OP.

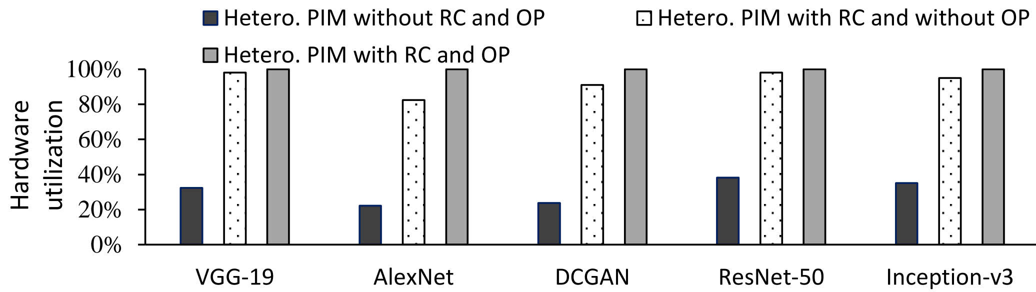


Figure 5.15: Hardware utilization with and without RC and OP.

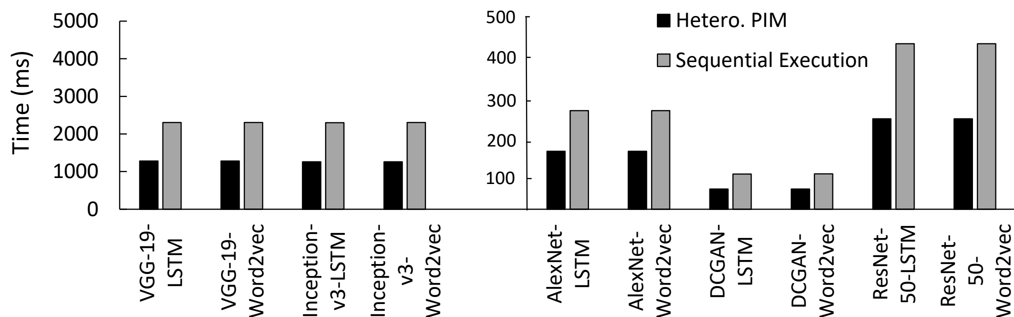


Figure 5.16: Execution time of multiple NN training models with our design and sequential execution, respectively.

5.4.6 Mixed Workloads Analysis

We also evaluate the case, when multiple models co-run in the same system [154]. We co-run two NN training models: a CNN model and a non-CNN model. The CNN

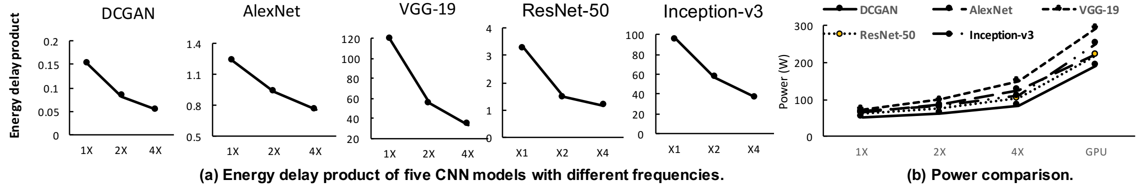


Figure 5.17: Energy efficiency and power with 3D memory frequency scaling.

model can execute on CPU and PIMs, subject to our runtime scheduling; the non-CNN model executes on CPU or the programmable PIM, when they are idle. Figure 5.16 shows the results of six co-run cases. In each case, "Hetero. PIM" indicates that we simultaneously execute both models, with the total execution time matched (i.e., when the CNN model executes for one step, the non-CNN model can execute one or multiple steps because the latency of the CNN model in one step can be longer than the non-CNN model in one step); "Sequential Execution" indicates that we execute the two models one after another in serial.

The results show that Hetero. PIM achieves 69%-83% performance improvement compared with Sequential Execution. Such improvement comes from high utilization of CPU and the programmable PIM in our design. With Sequential Execution, there can be no operations available to execute even though CPU and the programmable PIM are idle due to dependency between operations within the same model. Hetero. PIM avoids hardware idling, because operations across different models have no dependency and can execute simultaneously.

5.4.7 Energy Efficiency Analysis

We study energy efficiency of the PIMs with different frequencies as in Chapter5.4.4. We use energy-delay-product (EDP) as the metric to evaluate energy efficiency. Figure 5.17 (a) shows the results. The figure reveals that the most energy efficient point is not the original frequency for the five models. The 4× frequency is the most energy efficient for the five models. The tradeoff between energy consumption and execution time leads to such results. Thus, we conclude that higher frequency tends to be more energy efficient for NN model training. Figure 5.17 (b) compares power consumption between GPU and Hetero PIM with different frequencies. In general, GPU is very power hungry. It consumes 1.5× to 2.6× more power than Hetero PIM with high frequency (4×). Compared with GPU, Hetero PIM can be highly power

efficient.

5.5 Related Work

To our knowledge, this is the first work to propose a software/hardware co-design of a heterogeneous-PIM-based acceleration framework for NN training. Whereas previous PIM-based accelerator designs [155, 156, 157, 158, 159, 160, 161, 128] investigated the mapping of workloads on either fixed-function or programmable PIMs, it is unclear how to coordinate software and hardware designs to best utilize PIM technologies to support the heterogeneity requirement of NN training workloads.

5.5.1 Processing-in-memory for Machine Learning

Recent PIM-based machine learning accelerator designs strive to leverage the memory cells of nonvolatile memory technologies to execute NN inference operations [128, 162, 163, 164]. However, NN training typically incorporates substantial complex operations as we identified. It is difficult to accommodate these complex operations in previous processing-in-memory-cell designs. Azarkhish et al. [165] and Schuiki et al. [166] adopt RISC-V cores [167] and a streaming coprocessor in die-stacked DRAM to accelerate convolution networks or SGD. However, the RISC-V cores are merely used to control the arithmetic elements in the streaming coprocessor. Furthermore, both designs require users to modify code and perform tiling based on new APIs. Schuiki et al.'s study [166] only focuses on a specific operation (SGD). Azarkhish et al.'s design [165] primarily aims at inference and requires data to be carefully laid out in memory with 4D tiling. This constraint on data layout leads to inefficient training, because intermediate activations after each layer need to be re-tiled [166]. Neurocube [129] accelerates CNN inference and training by integrating programmable processing elements in the logic layer of 3D die-stacked DRAM. However, using programmable PIMs alone cannot provide the massive parallelism and execution efficiency enabled by heterogeneous PIMs. Furthermore, the aforementioned previous studies do not consider dynamic runtime scheduling of operations. Our experiment results demonstrate an efficient heterogeneous PIM design with runtime scheduling.

5.5.2 Processing-in-memory for General Applications

Fujiki et al. [161] proposed a ReRAM-based in-memory processor architecture and data-parallel programming framework. The study introduces a compact instruction set for memory array with processors. The programming framework combines dataflow and vector processing, employs TensorFlow input, and generates code for in-memory processors. Our work also employs TensorFlow, but optimizes operations scheduling and introduces PIM heterogeneity. Ahn et al. [156] explores mapping of PIM operations based on data locality of applications, while we schedule operations in multiple dimensions – hardware utilization, data locality, and data dependency. Ahn et al. [160] introduced PIM for parallel graph processing. The design offers an efficient communication method between memory partitions and develops prefetchers customized for memory access patterns of graph processing. Other works introduce PIM architectures based on 3D-stacked memory. For example, Zhang et al. [168] presented an architecture for programmable, GPU-accelerated, in-memory processing implemented using 3D die-stacking. The throughput-oriented nature of GPU architectures allows efficient utilizaztion of high memory bandwidth provided by 3D-stacked memory, while offering the programmability required to support a broad range of applications. Akin et al. [157] presented a set of mechanisms that enable efficient data reorganization in memory using 3D-stacked DRAM. However, the aforementioned studies cannot efficiently accelerate NN training workloads, because they cannot fully accommodate the heterogeneous computing requirement in NN training. Furthermore, these studies do not consider efficient programming model and runtime system to accommodate the hardware heterogeneity as explored in our study.

5.5.3 Other Accelerator Optimization for Machine Learning.

Recent works explored software- and hardware-based approaches for a variety of inference acceleration [169, 170, 171, 172, 173]. Most of these works focused on improving performance and energy efficiency of NN inference. However, training is much more compute and memory intensive than inference. The data movement overhead in training is much more significant. Several prior studies [174, 175, 176] investigated architecture design for NN training. However, these studies focus on addressing the memory capacity constraint issues caused by a large amount of feature

maps generated in CNN training. The data movement bottleneck is not fully explored.

5.6 Summary

In this chapter, we propose a software and hardware co-design of heterogeneous PIM approach, combining the power of programmable PIM and fixed-function PIMs, for NN training. Our software design enables (1) a portable and unified programming model across CPU, fixed-function PIMs, and programmable PIM; (2) runtime scheduling that effectively optimizes PIM hardware utilization and maximizes NN-operation-level parallelism. Our design not only allows natively training models to execute on heterogeneous PIM, but also enables easy maintenance of machine learning frameworks. Our design achieves significant improvement in performance and energy efficiency with various NN training workloads.

Chapter 6

Runtime Concurrency Control and Operation Scheduling for High Performance Neural Network Training

6.1 Motivation

We study the performance characteristics of operations to motivate our concurrency control and operation scheduling. We perform our study from three perspectives: (1) Operation performance variance with different thread-level parallelisms; (2) Impact of the input data size on operation performance; (3) Performance impact of co-running operations.

Hardware platform. We use Intel Knights Landing (KNL) processor (Xeon Phi 7250) as a manycore example in the rest of the work. Several leadership supercomputers are based on KNL, including Cori at Lawrence Berkeley National Lab and Theta at Argonne National Lab. KNL provides strong computation capabilities to train and deploy neural networks [177]. We use KNL processors at Cori for our study.

A KNL processor can contain 68 cores, each of which supports four hardware threads (in total 272 hardware threads). 68 cores are organized into 34 tiles (i.e., two cores per tile). Two cores in the same tile share a 1 MB L2 cache (the last level cache). KNL has a 16GB on-package high-bandwidth memory. This memory can

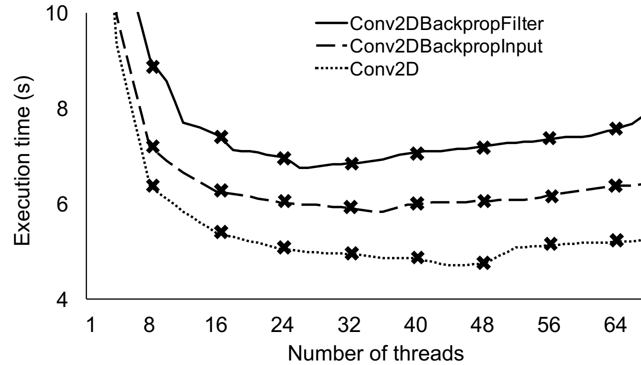


Figure 6.1: Performance variance of three operations with different intra-op parallelisms. The reported execution time is the total execution time of one thousand runs.

be configured as a transparent, direct-mapped hardware cache. This configuration is called "cache mode". The cache mode is the most common mode in a KNL-based HPC. All the tests in this work use the cache mode of KNL. With the cache mode, all data sets of NN models in our tests are placed in the high-bandwidth memory and there is no effect of non-uniform memory access (NUMA).

We use TensorFlow (v1.9) in our study. We develop a performance profiling framework by leveraging TensorBoard [126] and Intel VTune [144] to collect timing and hardware counter information of operations. The default intra-op and inter-op parallelisms in Tensorflow are set as the number of logical cores of the hardware platform (272 in KNL). However, the TensorFlow performance guide [17] recommends the user to set the inter-op parallelism as the number of sockets (which is one in our platform) and set the intra-op parallelism as the number of physical cores, which is 68 in our platform.

6.1.1 Performance Variance with Different Concurrency

We change the intra-op and inter-op parallelisms when running a couple of NN models (particularly ResNet-50 and DCGAN). Table 6.1 summarizes the results. There is a significant performance variance across different cases. The default case (68 threads for intra-op parallelism and 1 for inter-op parallelism), which is our baseline, does not result in the best performance. There is up to 28% performance difference between the default case and the most performant case, as shown in Table 6.1.

Furthermore, we change the number of threads to run individual operations (i.e., not the whole NN model). When running each operation with multiple threads, we

Table 6.1: Study the performance of NN models with different inter-op and intra-op parallelisms. The performance baseline for calculating speedup is the performance with the configuration recommended by the TensorFlow programming guide (68 threads for intra-op parallelism and 1 for inter-op parallelism).

Parallelism		ResNet-50		DCGAN	
Inter-op	Intra-op	Time (ms)	Speedup	Time (ms)	Speedup
1	34	1414	0.98	484	1.21
1	68	1382	1.00	524	1.00
1	136	2257	0.61	1045	0.50
2	34	1088	1.27	411	1.28
2	68	1213	1.14	501	1.04
2	136	4017	0.34	1238	0.42
4	34	1169	1.18	434	1.21
4	68	3048	0.45	565	0.93
4	136	4782	0.29	1469	0.36

put those threads with data sharing into the same tile for best performance (threads resident in the same tile share the last level cache). We do not use hyper-threading for the tests. When we run those individual operations, we develop a series of scripts to run them as standalone operations to avoid any performance interference between operations as in the NN model training.

Figure 6.1 shows the execution times of three operations, $Conv2DBackpropFilter$, $Conv2D-$ $BackpropInput$ and $Conv2D$ with different number of threads. The three operations are common and among the most time-consuming operations in NN training [178].

For those three operations, we use certain input data sizes in the NN model Inception-v3 [145].

Figure 6.1 reveals that we achieve the best performance, when we use 26, 36 and 45 threads to run the three operations respectively. There is up to 17.3% performance difference between the default concurrency (i.e., 68 threads) and the best case. The scalability of the three operations with the given input data size is limited on KNL due to thread spawning overhead and non-parallelizable code regions.

Observation 1. The intra-op parallelism must be chosen differently for different operations, in order to achieve the best performance of individual operations.

6.1.2 Impact of Input Data Size

An NN model can involve many instances of an operation in a training step. Different instances of the operation can use different input data sizes. For example, in Inception-v3, the operation $Conv2DBackpropFilter$ has 42 instances in a training

step, each of which uses different input data sizes.

We study three operations from Inception-v3, which are *Conv2DBackpropFilter*, *Conv2D− BackpropInput* and *Conv2D*. We change the input data sizes of the three operations. For each input data size, we change the intra-op parallelism to find the best performance. Table 6.2 shows the results. The table shows that as we change the input data sizes, we need to use different numbers of threads to achieve the best performance. For example, for *Conv2DBackpropFilter* with the input data size *par_input* as (32,8,8,384), we need to use 26 threads to achieve the best performance, while with the input data size *par_input* as (32,17,17,384) and *par_input* (32,8,8,2048), we need to use 42 and 68 threads, respectively.

Observation 2. The best concurrency (in terms of the optimal number of threads per operation) changes, as we change the input data size of the operation.

6.1.3 Co-Running Operations

We study the performance of co-running operations. We use three strategies to run two operations. First, we run them in serial, and each operation uses 68 threads. This strategy would be used by the TensorFlow runtime by default. Second, we leverage two hardware threads (hyper-threading) in each core to allow the two operations to co-occupy 68 cores (i.e., each operation uses 68 cores and there is one hardware thread per core for each operation). Third, we evenly partition 68 cores between the two operations (i.e., each operation uses 34 core; only one hardware thread per core). The performance of co-running the two operations is the time span from launching them to finishing both of them.

Table 6.3 summarizes the results of co-running *Conv2DBackpropFilter* and *Conv2DBackprop− Input*. The input size for the operations is *par_input* (32,8,8,2048). Given this input size, the number of threads to achieve the best performance for the two operations is 68.

Table 6.3 reveals that the third strategy achieves the best performance. Comparing with the first strategy, the third one has 38% performance improvement, although individual operations have performance loss (25% and 17% for *Conv2DBackpropFilter* and *Conv2DBackpropInput* respectively). Hyper-threading (the second strategy) is helpful in this study: we have 3% performance improvement (comparing with the first strategy).

Table 6.2: Study the impact of input data size on operation performance. The performance baseline for calculating performance variance is the performance with using 68 threads. The reported time is the total execution time of one thousand runs.

Operation Type	Input data size	Time (s)	Intra-Op Parallelism	Performance Variance
Conv2DBackpropFilter	(32,8,8,384)	7.2	26	17.3%
	(32,17,17,384)	11.1	42	10.2%
	(32,8,8,2048)	20.3	68	0%
Conv2DBackpropInput	(32,8,8,384)	5.8	36	9.8%
	(32,17,17,384)	8.7	56	2.3%
	(32,8,8,2048)	19.6	68	0%
Conv2D	(32,8,8,384)	4.7	45	11.1%
	(32,17,17,384)	7.4	63	3.5%
	(32,8,8,2048)	14.8	66	2.0%

Table 6.3: Co-running two operations with three strategies. The performance baseline for calculating speedup is performance of serial execution of two operations. The reported time is the total execution time of one thousand runs.

Strategies	#Threads	Time (s)	Speedup
Serial execution	68	41.1	1
Co-run with hyper-threading	68+68	39.9	1.03
Co-run with threads control	34+34	29.8	1.38

Observation 3. Co-running operations are helpful for overall performance improvement, even though individual operations may have performance loss when co-running them.

6.2 Design

6.2.1 Overview

Our motivation examples demonstrate the necessity of dynamically changing concurrency (intra-op and inter-op parallelisms) and scheduling operations to reduce training time. Driven by the motivation examples, we use a performance model-driven approach to extend the TensorFlow runtime. Figure 6.2 generally depicts our runtime and its workflow.

In particular, we explore two performance models to predict performance (execution time) of each operation with different intra-op parallelisms. We study the performance modeling accuracy and model portability across architectures and operation implementations. We further extend the TensorFlow runtime to schedule operations to enable co-running operations.

Our two performance models are based on dynamic profiling. The performance models use a few training steps (the profiling steps) to profile operations and then

make performance prediction on operations with various intra-op parallelisms.

Our first performance model uses machine learning models (a set of regression models) to make the performance prediction. Those models use characteristics of operations as input features. We characterize computation and memory access patterns of operations by collecting performance events during the profiling steps. However, the machine learning models have low prediction accuracy because execution times of some operations are short and collecting performance events with hardware counters within such short times is not accurate. Furthermore, the regression models are architecture dependent and have to be re-trained on a new platform.

Our second performance model is based on the hill climbing algorithm [179]. Our hill climbing algorithm aims to find the best performance (the shortest execution) and corresponding number of threads to run an operation with a given input data size. The algorithm starts with a certain number of threads to run the operation, then attempts to find another number of threads with a shorter execution time by making an incremental change to the number of threads. If the change produces a shorter execution time, another incremental change is made to the number of threads until no further execution time is reduced. When running the hill climbing algorithm, the runtime tests a few cases (i.e., the profiling cases) and measure their execution times. To predict the performance of any untested case, we use linear interpolation to predict the performance of the untested case based on the measured performance of two profiling cases. The hill climbing algorithm-based performance model is lightweight and accurate. Different from the first performance model, it is architecture-independent and require no knowledge of operations. Hence, we use the second performance model to guide the runtime.

Our runtime system decides (1) the optimal intra-op parallelism for each operation and (2) which operations to co-run to increase system utilization. The performance model determines the optimal number of threads for an operation that results in the shortest execution time. To avoid frequent changes of operation concurrency, which may lead to sub-optimal overall performance, we optimize operation concurrency for the largest input size, which yield overall better performance. Furthermore, our runtime decides which operations should co-run and how they should interleave. We analyze a set of candidate execution scenarios and select the one that best suits the current execution flow to increase overall system throughput and hardware utilization

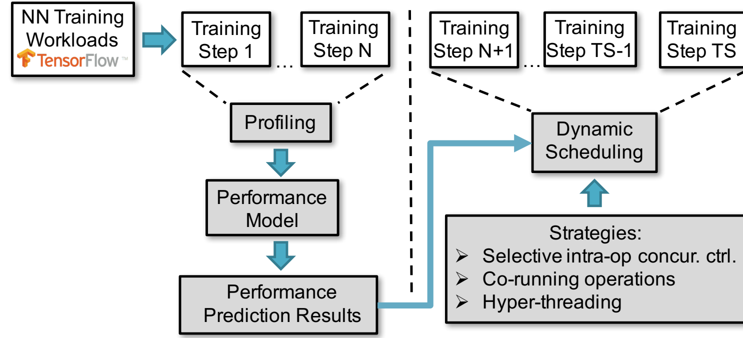


Figure 6.2: Our runtime framework and its workflow. The notation "TS" is the total number of training steps.

under the constraints of available computing resources. Our runtime system also leverages hyper-threading to allow multiple operations to share the same physical cores to improve system throughout. In the following, we describe our design in detail.

6.2.2 Regression Model-Based Performance Model

Our first performance models, which are regression-based, predict performance for 68 cases, each of which has a specific number of threads. For each case, we have a performance model to make prediction (68 performance models in total). For each prediction case, there is only one thread per core. We do not predict the cases with multiple threads per core (i.e., using hyper-threading), because hyper-threading often causes performance slowdown, when running a single operation.

Among the 68 prediction cases, 34 of them have at most one thread in each tile. In other words, those 34 cases do not have any cache sharing between threads. The remaining 34 cases have either two threads or no threads in each tile. In other words, those 34 cases have cache sharing between threads. For those 34 cases, we only use even number of threads. We do not consider odd number of threads, because that makes some tile have only one thread, causing load imbalance between tiles.

Note that we use 68 performance models to predict 68 cases. We do not try to build a single model to predict performance of the optimal case (the case with the shortest execution time), because the runtime system needs to know the performance of many cases to decide which operations to co-run. We also do not try to build a single model to predict performance of 68 cases, because of the complexity of model training and low prediction accuracy (as low as 25% according to our study).

Each performance model collects a set of workload features as model input, using

a few training steps. We consider *thread affinity* while collecting features. Thread affinity decides the binding between threads and cores. For those threads with large data sharing, we want to bind them into the same tile, such that those threads can reuse data in the L2 cache of the tile. Given the number of threads per tile and total number of threads to run an operation, different thread affinity can result in different performance. Our model aims to predict performance with the best thread affinity.

When running operations with a specific number of threads and measuring the execution times of those operations, we carefully choose which two threads should share a tile. In particular, we put the threads with continuous IDs into the same tile. For example, threads 1 and 2 share a tile, and threads 3 and 4 share a tile. This method is based on the following observation: The multi-threading mechanism (i.e., OpenMP) used in TensorFlow on KNL is implemented in the Intel MKL-DNN library, and this mechanism parallelizes operations by assigning iterations of the major computation loop to threads in order, and neighbor iterations tend to access the same data set, hence the threads with continuous IDs that work on the neighbor iterations tend to have data sharing.

The above method provides a lightweight and practical solution to enforce thread affinity for best performance. There are other solutions that involve compiler and runtime analysis [180], but they are expensive.

Feature Selection. We use performance events collectible by hardware counters, and the execution time of the operation, as features. In total there are 27 features.

On KNL, there are 26 performance events collectible by hardware counters. Using all of them as features is problematic due to the following reasons. First, those performance events cannot be collected at the same time. We need at least four training steps to collect those events separately, which increases the number of training steps for profiling. Second, some features are not informative, discriminating and independent. For example, the number of branch instructions and number of conditional branch instructions are correlated and redundant, and should not be selected together.

We employ the decision tree estimator to select features. We choose four features: number of CPU cycles, number of last level cache misses, number of last level cache accesses and number of level 1 cache hits. We also normalize the numbers of performance events by the total number of instructions to make the feature values independent of

Table 6.4: Prediction accuracy of a set of regression models.

#Sample (N)	Metrics	Gradient Boosting	K-Neighbors	TSR	OLS	PAR
1	Accuracy	61%	56%	37%	27%	18%
	R^2	0.961	0.818	0.779	0.981	0.196
4	Accuracy	57%	67%	17%	21%	14%
	R^2	0.957	0.592	0.539	0.951	0.175
8	Accuracy	51%	56%	26%	31%	18%
	R^2	0.972	0.589	0.965	0.977	0.177
16	Accuracy	34%	26%	13%	14%	11%
	R^2	0.959	0.585	0.852	0.892	0.159

total number of instructions. The normalization makes the performance model usable for workloads with different number of instructions.

Feature collection. We collect features using N sample cases. Each sample case uses a specific number of threads to run a training step. All operations in this training step use the same number of intra-op parallelism. In the training step, we run the operations in serial to avoid performance interference among multiple operations and ensure accuracy of feature collection.

We choose sample cases by evenly sampling the search space of possible intra-op parallelisms with the consideration of cache sharing. Using those sample cases is meant to be representative of all cases.

To decide the number of sample cases (N), we change N to study its impact on modeling accuracy. The results are summarized in Table 6.4. The results reveal that N has a significant impact on modeling accuracy, but a large N is not helpful for improving modeling accuracy. Also, using a large N can cause large runtime overhead, because of frequent counting performance events for a large number of operations. In our test with ResNet-50, when $N = 16$, the runtime overhead is up to 20%.

Regression models. We experiment with ten regression models and compare their accuracy, including random forest, k-nearest neighbors, gradient boosting, ϵ-support vector machine regressions (ϵ-SVR) with linear, poly and RBF kernels, decision tree, Bayesian automatic relevance determination (ARD), ordinary least squares (OLS), passive aggressive regression (PAR), multiple layer perceptron (MLP) with sgd, lbfgs and adam kernels and Theil Sen Regression (TSR).

Training Data Set. For training data set, we collect operation information from three common NN models with TensorFlow (particularly ResNet-50 with CIFAR-10 dataset, DCGAN [125] with MNIST dataset and Inception-v3 with ImageNet dataset. To increase training data set, we vary batch size from 16 to 256. When we run those

operations in the three NN models , we develop scripts to run them as standalone operations, similar to what we do in the motivation examples (Section 6.1).

Model Testing. We test model accuracy with DCGAN. Table 6.4 shows the results. We use two metrics, modeling accuracy and R^2 (the coefficient of determination). The modeling accuracy is defined as $1 - \frac{1}{n_t}\sum\left|\frac{\hat{y}_i - y_i}{y_i}\right|$ where n_t is the size of the test data set, and $\hat{y}_i$ and y_i are the predicted and actual execution times, respectively.

Table 6.4 shows that the regression-based performance models do not present good accuracy for the selection of operation concurrency. Using the most accurate regression model (k-neighbors) to direct NN model training (ResNet-50 in particular), we have performance loss (30%).

We attribute those prediction inaccuracy to possible inaccuracy in hardware counters to collect performance events. Using hardware counters can be inaccurate. Furthermore, the regression model-based performance models are architecture-dependent. The regression models need to be re-trained on a platform with different hardware counters.

Because of the above reasons, we propose to use a hill climbing algorithm to direct the selection of intra-op parallelism for operations.

6.2.3 Hill Climbing Algorithm-Based Performance Model

We describe our hill climbing algorithm as follows. Similar to the regression-based performance models, we use N training steps to run operations in serial with different number of threads. In particular, we first use one thread to run each operation and measure execution time in one step. Then, we increase the number of threads by x (named as *interval*) to run each operation and measure its execution time. By increasing the number of threads, the execution time can decrease. We further to increase the number of threads by x in the following steps, until one of the following two cases happens: (1) the execution time increases; (2) we reach the maximum number of cores to run threads. If (1) happens, then we stop changing the number of threads for this operation and claim that we find the best number of threads to run the operation in the last time step. If (2) happens, then the best number of threads to run the operation is the maximum number of cores.

We consider thread affinity in the above hill climbing algorithm. In particular, given a specific number of threads to run an operation, we run the operation twice

Table 6.5: Performance prediction accuracy for four NN models based on the hill climbing-based performance model.

	Intervals			
Models	2	4	8	16
ResNet-50	98.13%	95.45%	83.42%	31.12%
DCGAN	97.16%	94.43%	51.54%	10.14%
Inception-v3	97.91%	94.22%	73.21%	21.21%
LSTM	95.56%	90.45%	41.34%	11.03%

with two training steps: one step with cache sharing between threads, and the other without cache sharing between threads.

The output of the above hill climbing algorithm includes not only the shortest execution time and corresponding number of threads, but also the execution time of those sampling cases in the N training steps. To predict the performance of those cases that are not tested in the N steps, we simply use linear interpolation. For example, if we measure the execution time of using one and four threads for an operation ($x = 3$ in this example), then the execution time of using two and three threads will be approximated based on a linear interpolation between the execution times of using one and four threads.

N (the number of training steps to run sample cases) is related to x. Assuming that the maximum number of cores is C, then N is at most $C/x \times 2$ (we have "2", because we consider both cache-sharing and no-cache-sharing cases.)

Performance prediction accuracy. We run ResNet-50, DCGAN, Inception-v3 and LSTM. and use the hill climbing algorithm-based performance model to predict performance of those cases not executed in the N steps. We change x from 2, 4, 8 to 16. Table 6.5 shows the prediction accuracy. The prediction accuracy is the average prediction accuracy for all operations. In general, we achieve very high prediction accuracy (up to 98.13% with $x = 2$ and 95.45% with $x = 4$), much higher than regression model-based performance models (Section 6.2.2).

Discussion. Using the hill climbing algorithm has two potential problems. First, the "shortest execution time" found by the hill climbing algorithm may be a "local optimum" solution, not a "global optimum" solution. However, after extensive evaluation of operation performance (1025 operations in four NN models) with different number of threads, we observe that the local optimum is always the global optimum. As the number of threads changes, the variance of execution time is shown as a convex function.

Second, if the interval x is large, it is possible that the hill climbing algorithm may skip the optimum. For example, assuming that the hill climbing algorithm has tested the case of eight threads, $x = 4$, and the optimum is the case of 10 threads, then the hill climbing algorithm will only test the case of 12 threads and skip the optimum. The case of 12 threads is incorrectly selected as the optimum. However, our evaluation reveals that the optimum found by the hill climbing is pretty close to the real optimum. With the evaluation of four NN models (ResNet-50, DCGAN, Inception-v3 and LSTM) and $x = 4$, the performance difference between the two optimums is less than 2%.

In conclusion, the performance model based on hill climbing is a practical and effective approach for performance profiling and prediction. Comparing with the regression model-based performance models, the hill climbing has the following advantages: (1) No need of performance model training; (2) architecture independence; (3) no need of considering operation characteristics, hence can accommodate any future change of operations in TensorFlow; and (4) better accuracy.

6.2.4 Runtime Scheduling

The runtime decides (1) intra-op parallelism for each operation, and (2) which operations to co-run. The existing runtime system in TensorFlow employs a first-in-first-out policy to schedule operations: The operations that are ready to run are simply executed in the order they put into the operation queue. All operations use the same intra-op parallelism and inter-op parallelism defined by the user before the training starts. Such scheduling strategy loses performance without sufficient consideration of operation scalability and hardware utilization. Our runtime avoids this problem and schedules operations based on the following strategies.

Strategy 1: Deciding intra-op parallelism for individual operations based on the performance model. After running the hill climbing algorithm in the first few steps, the runtime runs each instance of each operation using the number of threads that can lead to the shortest execution time. This indicates that different operations may use different number of threads; This also indicates that different instances of an operation with different input data sizes may also use different numbers of threads.

Strategy 2: Avoiding frequent change of operation concurrency. In

practice, Strategy 1 might not lead to better performance than the execution with the default TensorFlow configuration. The reason is because of frequent change of operation concurrency, which causes cache thrashing and large thread management overhead (e.g., thread spawning or binding to cores). In Strategy 2, the runtime avoids frequent change of operation concurrency. In particular, the operation, no matter what input data size it uses, always use the same number of threads, but different operations can still use different number of threads. The number of threads to run the operation is determined by the operation instance with the largest input data size (the most time-consuming instance), such that the execution time of this operation instance is the shortest.

Strategy 3: Co-running operations to maximize hardware utilization. To decide which operations should co-run and how they should co-run, we use the following algorithm. For any operation ready to run, we use three different numbers of threads as *candidates* to run the operation (The "three" is an empirical number). The three candidates should be the most performant ones (i.e., the ones with the shortest execution times). Whenever some physical cores are idling, either because an operation is just finished or because we just start the training, we examine those operations ready to run. For each of those operations, we check if any of its three candidates can fit into the idling cores without decreasing system throughput. We decide whether system throughput will be decreased by ensuring that the candidate does not take longer execution time than ongoing operations in busy cores.

For example, an operation ready to run has three candidates, which are (1) using 18 threads (no cache sharing) that takes 1.5 seconds; (2) using 20 threads (no cache sharing) that takes 1.3 seconds; and (3) using 16 threads (no cache sharing) that takes 2.1 seconds. We have 20 idling cores, and the remaining 48 cores run an ongoing operation that needs 1.9 seconds to finish. We choose the candidate (1) to co-run with the ongoing operation, because it takes shorter execution time than the ongoing operation (1.5 vs. 1.9 seconds), and can fit into the 20 idling cores. We do not use 20 threads to fit into the 20 idling cores, because using 18 threads can release two idling cores to run another operation and we want to *maximize operations co-running to increase system throughput*. An argument to support using 20 threads is that we finish the operation earlier and then run another operation. However, according to our experience, maximizing operations co-running (using 18 threads) is helpful to system

throughput and hence more beneficial for performance. Note that the above execution times for operations are predicted based on the performance model.

If we cannot find any operation that can fit into the idling cores without decreasing system throughput, we choose the most time-consuming operation to run.

Strategy 3 should not conflict with Strategy 2. If the number of threads to run an operation based on Strategy 3 is quite different from the number of threads chosen by Strategy 2 (the difference in the number of thread is larger than 2 and "2" is an empirical value), then we will use the number of threads chosen by Strategy 2 to run the operation. This method avoids disruptive changes to intra-op parallelism for each operation.

Strategy 3 is lightweight and can make a quick decision on how to co-run operations, such that the runtime overhead is small. Based on our profiling on four neural networks (ResNet-50, DCGAN, Inception-v3 and LSTM), we seldom have more than five operations ready to run. Hence, Strategy 3 does not need to explore a lot of operations to make the decision.

Strategy 4: Leveraging hyper-threading to run multiple operations. Some scalable operations can take all 68 cores and never allow any operation to co-run. However, we find that running small operations using hyper-threading along with the time-consuming, scalable operations can be beneficial for performance. This means that the small operations share physical cores with the time-consuming operations, enabling another type of co-run.

At runtime, when the runtime finds an operation using 68 cores, the runtime then tries to co-run small operations. The small operations are defined as those operations that have shortest serial-execution time in the operation-ready queue.

Putting all together. The runtime uses Strategies 1-2 to decide the number of threads to run for each operation based on the performance model. This can be done right after running the hill climbing algorithm in the first few training steps (the profiling steps). After that, the runtime decides how to co-run operations based on Strategies 3-4. The runtime repeatedly uses the four strategies until all operations are finished. Note that to minimize runtime overhead, some decisions based on Strategy 3 to co-run operations can be reused without repeatedly running Strategy 3.

Discussion. Our performance model is used to predict performance for individual operations, and does not capture performance interference between operations when

co-running them. Hence, when we use the performance model to direct operations to co-run, the performance loss of individual operations can be unexpected low because of performance interference. Our runtime can record such cases and avoid co-running such operations in the future train steps. In practice, we do not find significant performance slowdown in individual operations when co-running them.

6.3 Experiment Setup

6.3.1 Training Models, Data Set and Framework

We employ CIFAR-10, MNIST, ImageNet and PTB training dataset for ResNet-50, DCGAN, Inception-v3 and LSTM respectively. The batch sizes of ResNet-50, DCGAN, Inception-v3 and LSTM are 64, 64, 16 and 20, respectively. We adopt TensorFlow (v1.9) as our NN training framework. We use the implementation of ResNet-50, Inception-v3 and LSTM from the TensorFlow software package [181] and DCGAN from [182]. In TensorFlow, the default intra-op and inter-op parallelisms are set as the number of logical cores of the hardware platform (272 in KNL). As discussed in Chapter 6.1, the TensorFlow performance guide recommends to set the inter-op parallelism as the number of sockets (which is one in our platform) and set the intra-op parallelism as the number of physical cores, which is 68 in our platform. Since the performance of the TensorFlow default configuration is much worst (more than 10 times slower) than the recommended configuration from the TensorFlow performance guide, we use the *recommended configuration* as the baseline in our evaluation. The performance with the recommended configuration is annotated as "Recommendation" in Figure 6.3 and Table 6.6.

The performance reported in this chapter is the execution time of one training step. Recall that the performance of one training step remains stable across training steps, hence the execution time of one training step is good for performance evaluation. In addition, there is no accuracy loss in NN models with our runtime, because our runtime does not make any change to the input data sizes of operations, does not change any NN model parameters, and does not violate any dependency between operations.

6.3.2 Hardware Platform

We use a machine with an Intel Knights Landing (KNL) processor (Xeon Phi 7250) at the Cori supercomputer at Lawrence Berkeley National Lab as our test platform. Section 6.1 has more details for KNL.

6.3.3 Controlling Intra-op Parallelism

On Intel KNL, TensorFlow uses operations implemented in both MKL-DNN and Eigen. Dynamically changing intra-op parallelism for those operations implemented in the Eigen causes large runtime overhead (larger than 10%), because the Eigen decomposes an operation into a large number of tasks, and changing intra-op parallelism of an operation causes frequent task-pushing into and task-popping out of a queue associated with each thread. MKL-DNN uses OpenMP threads to parallelize operations, and there is negligible overhead to change intra-op parallelism for those operations implemented in MKL-DNN. Hence, in our evaluation, we only change intra-op parallelism for those operations implemented in MKL-DNN. Those operations take more than 70% of total NN training time.

To enable dynamic change of intra-op parallelism for a few operations (e.g., batch_normalization in DCGAN), we have to make small changes to the operation implementation. For example, we have to allocate a larger memory space for some variables during operation initialization. However, the changes are minor and have no impact on operation performance.

In general, the implementation of our runtime incurs limited overhead (less than 1%). Also, the number of profiling steps is small (less than 0.05% of total training steps). Hence, the profiling overhead is negligible.

6.4 Evaluation

Figure 6.3.d compares the performance of our runtime system with that of the recommended TensorFlow configuration (labeled as "recommendation") and of manual optimization. For manual optimization, we manually change intra-op and inter-op parallelisms uniformly enforced on all operations, aiming to find the best configuration. The manual optimization is not a scalable solution, because we have to exhaustively test every possible combination of intra-op and inter-op parallelisms to find the best

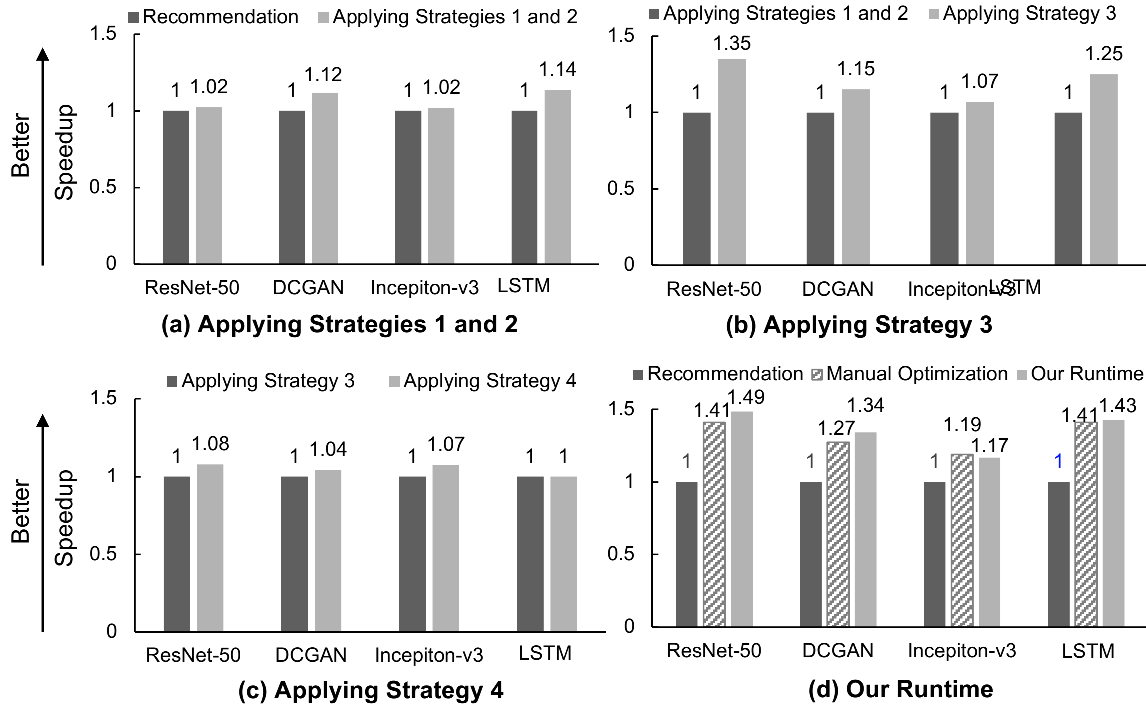


Figure 6.3: Quantifying the contribution of the four strategies. Comparing the performance of our runtime, manual optimization, and the recommendation by TensorFlow.

configuration.

Figure 6.3.d reveals that our runtime leads to the best performance in all tests. Our runtime performs at least 17% (Inception-v3) and up to 49% (ResNet-50) better than the recommendation. Our runtime performs even better (at least 2%) than the manual optimization for three NN models (ResNet-50, DCGAN and LSTM), and performs similar (2% worse) to the manual optimization (Inception-v3).

The above results demonstrate the superior performance of our runtime system. To further understand the performance contributions of four runtime strategies, we apply them one by one. The results are shown in Figure 6.3.a-Figure 6.3.c.

6.4.1 Applying Concurrency Control for Individual Operations

Figure 6.3 shows that applying Strategies 1 and 2 alone, we have 14% performance improvement for LSTM, 12% for DCGAN and 2% for ResNet-50 and Inception-v3.

Table 6.6 shows the execution times of the top five most time-consuming operations of four NN models with the recommended TensorFlow configuration and with Strategies

1 and 2 in place. The table reveals that we have better performance for all operations, up to 34% performance improvement.

Some operations do not have performance improvement after applying Strategies 1 and 2, however these operations (e.g., Conv2D in ResNet-50) with our runtime can use less number of threads than with the recommendation, while achieving the same performance. Using less number of threads introduces opportunities to co-run operations.

6.4.2 Applying Operations Co-running

To isolate the effects of co-running operations from Strategy 4, we apply Strategy 3 after using Strategies 1 and 2 without Strategy 4.

Figure 6.3.b shows the results. The performance reported in the figure is normalized by the performance of applying Strategies 1 and 2. By using Strategy 3, ResNet-50 achieves 35% performance improvement. LSTM achieves 25% performance improvement. DCGAN and Inception-v3 achieve 15% and 7% performance improvement, respectively.

6.4.3 Applying Hyper-threading

To isolate the effects of Strategy 4 from other strategies, we apply Strategy 4 after applying Strategy 3 (this implicitly indicates that we also apply Strategies 1 and 2).

Figure 6.3.c shows the results. The performance reported in the figure is normalized by the performance of applying Strategies 3. ResNet-50, DCGAN, and Inception-v3 achieve 8%, 4%, and 7% performance improvement, respectively. LSTM has no performance improvement, because almost no operation in LSTM needs all of cores to achieve best performance, hence there is few opportunity to apply Strategy 4.

To further study the effectiveness of Strategy 4, we record the number of co-running operations along with the NN training. In particular, whenever there is an operation finished or launched, we record the number of co-running operations at the moment. Finishing or launching an operation is an *event*. Figure 6.4 shows the number of co-running operations whenever an event happens. There are a large number of events (sometimes millions of events), in just one training step. Presenting the number of co-running operations for all events makes the figure very intensive and difficult

Table 6.6: Performance improvement of the top five most time-consuming operations in four NN models by recommendation and by applying Strategies 1 and 2. The performance baseline for calculating speedup is the performance with the configuration recommended by the TensorFlow programming guide (68 threads for intra-op parallelism and 1 for inter-op parallelism).

Operations	Execution Time (ms)		Speedup
	Recommendation	Applying Strategies 1 and 2	
ResNet-50			
Conv2DBackpropFilter	158	146	1.08
InputConversion	131	122	1.07
Tile	107	105	1.02
Mul	103	100	1.03
ToTf	79	78	1.01
Inception-v3			
AvgPool	759	730	1.04
Tile	539	532	1.01
Conv2DBackpropFilter	479	475	1.01
MaxPooling	455	422	1.08
InputConversion	416	413	1.01
DCGAN			
Conv2DBackpropInput	164	144	1.14
Conv2DBackpropFilter	133	110	1.21
ApplyAdam	84	72	1.17
BiasAddGrad	26	23	1.17
FusedBatchNorm	15	14	1.03
LSTM			
SparseSoftmaxCross	11.71	8.76	1.34
BiasAddGrad	2.03	1.98	1.03
Mul	1.36	1.09	1.25
AddN	1.02	0.87	1.17
MatMul	0.95	0.93	1.02

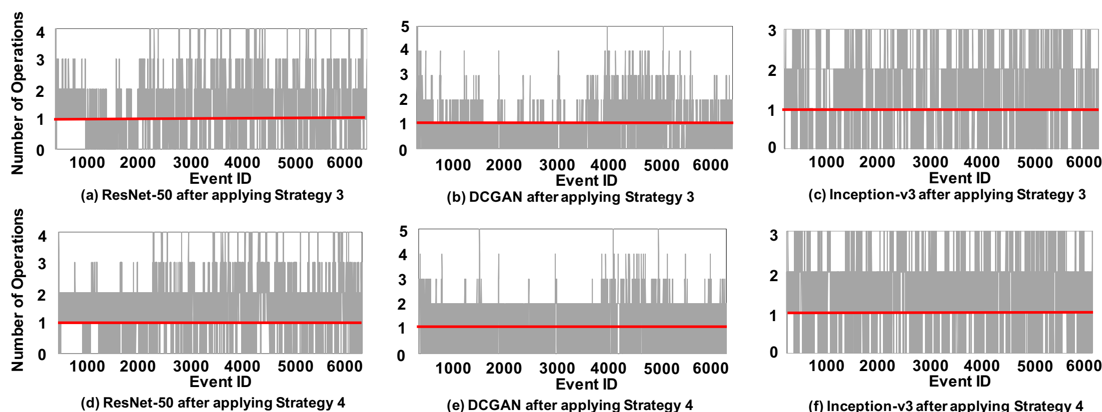


Figure 6.4: The variance of the number of co-running operations along with the NN model training. The figures (a), (b) and (c) do not have Strategy 4 (but have Strategy 3); The figures (d), (e) and (f) have Strategy 4 in place. The red lines in the figures are the inter-op parallelism recommended by TensorFlow.

to read. Hence, we present the number of co-running operations for 6000 events in Figure 6.4. The events happen in the middle of one step. Figure 6.4 does not show the results for LSTM, because there is no change in co-running operations after applying Strategy 4.

Figure 6.4 shows that with Strategy 4 in place, the number of co-running operations is larger than that without Strategy 4 (but with Strategy 3). The average number of co-running operations for 6000 events with Strategy 4 in place for three NN models are 1.89, 2.04, and 1.74, while without Strategy 4 (but with Strategy 3), the average number is 1.61, 1.62, and 1.52. Hence, Strategy 4 enables a larger number of co-running operations.

In general, we notice both Strategies 3 and 4 can dynamically change the number of co-running operations, instead of fixing the number of inter-op parallelism as in the traditional TensorFlow (shown as red lines in Figure 6.4).

6.4.4 Putting all together.

Figure 6.3.d shows the performance after applying all strategies together and compares it with the performance of the recommendation and manual optimization.

We observed that ResNet-50 achieves the largest performance improvement (49%) among the four NN models. Such a large performance improvement largely comes from applying Strategy 3. Many operations in ResNet-50 are not scalable, which brings a lot

of opportunities to apply Strategy 3 to co-run operations. Furthermore, ResNet-50 has many small operations which can run together with those time-consuming operations, by applying hyper-threading (Strategy 4).

6.4.5 Comparing with the manual optimization.

Figure 6.3.d compares the performance of the manual optimization and our runtime. We observed that the performance of ResNet-50, DCGAN and LSTM by our runtime can achieve 8%, 7% and 2% performance improvement than the manual optimization, respectively. Our experiments show that for ResNet-50, manual optimization sets intra-op and inter-op parallelisms as 16 and 4. For DCGAN, manual optimization sets them as 34 and 2. For LSTM, manual optimization sets them as 2 and 2.

For Inception-v3, our runtime performs 2% worse than the manual optimization. The manual optimization sets intra-op and inter-op parallelisms as 68 and 2, respectively. Such configuration is closing to the configurations chosen by our runtime for most of operations. Hence our runtime performs similar to manual optimization. Our runtime has slight performance loss (2%). We suspect that the slight performance loss comes from changing intra-op parallelism across operations.

6.5 Related Work

6.5.1 Performance Optimization for Dataflow-based Frameworks

Recent works explore performance optimization for dataflow-based frameworks [183, 184, 185, 186, 178]. Mirhoseini et al. [183, 184] propose a method that first schedules the operations to groups and then places those groups onto devices. Hafner et al. [185] allow the TensorFlow execution engine to parallelize computation to improve training performance. Liu et al. [178] propose a software and hardware co-design of heterogeneous processing-in-memory system that schedules NN training operations across compute resources to improve hardware utilization.

Our work is different from the existing efforts. We propose runtime scheduling strategies that co-run operations to improve hardware utilization and system throughput on manycore platforms. We also explore performance modeling to predict

performance of operations with various intra-op parallelisms, which is not explored in the existing efforts.

6.5.2 Thread Concurrency Throttling

Previous work explores dynamic thread concurrency throttling to achieve the optimal performance [187, 188, 189]. Pusukuri et al. [188] develop a framework to dynamically determine an appropriate number of threads that identifies near optimal number of threads with OpenMP to achieve the optimal performance. Sanzo et al. [187] proposes a self-regulation approach that predicts the scalability of applications to improve performance.

Our concurrency throttling approach differs from them, in that we not only study concurrency for individual operations, but also study inter-op concurrency control by co-running operations with various runtime scheduling strategies.

6.6 Summary

The new generation of ML frameworks such as TensorFlow and PyTorch embraces a dataflow model and represents computation by a directed graph composed of operations. Training an NN model based on such ML frameworks can generate a lot of operations, which brings challenges to manage them for best performance. We expect such challenges will be more pronounced in the future NN models. In this work, we study how to automatically decide intra-op parallelism for each operation and how to co-run operations to improve performance. We use a performance model-driven approach to guide the runtime system to parallelize and schedule operations. Guided by the performance model, we introduce a set of practical and effective scheduling strategies. Applying the performance model and scheduling strategies to the TensorFlow runtime, we achieve great performance improvement. Our work reveals many opportunities to improve the performance of NN training through concurrency control and operation scheduling.

Chapter 7

Conclusion

In this dissertation, we introduce a high performance SpTC algorithm to address the above challenges based on the innovation of leveraging new data representation, data structures and emerging HM architecture. We then present a high performance framework for SpTC sequences, which is based on a set of novelty in data structures, runtime techniques, and emerging Optane-based memory architecture. Also, we have demonstrated a software/hardware co-design of a heterogeneous PIM framework for high performance, memory-oriented and energy-efficient tensor-based NN training. Finally, we present an end-to-end tensor-based runtime system for efficient concurrency control and operation scheduling towards tensor-based NN training.

Future Work. Sparse tensor computations have inputs that include a majority of zeroes. In order to minimize the inefficiency of storing and computing zero-valued data, applications store only the nonzero data, with auxiliary data structures to search for the locations of these values. As a result, sparse tensor computation often exhibits unpredictable memory-access patterns that include indirection through the auxiliary data structures. Hence, in many modern computer architectures, the performance of sparse tensor computations is dominated by the movement of data across nodes and memory hierarchy. However, it is challenging to achieve high-performance and efficient data movement for sparse tensor computations, because of hardware heterogeneity and high dimensionality.

In order to address the above challenges, I plan to design a framework on heterogeneous memory and computing systems to optimize how sparse tensors are organized in memory, how sparse tensor computations are structured to minimize the

movement of data, and how the correlation between computation and movement of data makes the most efficient use of the heterogeneous hardware. The novel and most significant aspects of my research agenda include: (1) a sparse tensor compression technique that can be aware of and adaptive to features of diverse architectures to increase performance and efficiency of sparse tensor computations; (2) a memory manager to efficiently migrate sparse tensors in heterogeneous memories with the consideration of memory properties (e.g., latency, bandwidth and capacity); and (3) a runtime system to automatically schedule and optimize data- and computation-dependency for sparse tensor computations.

It remains future work to investigate tensor computation in disaggregated data centers. Cloud computing has become one of the most important aspects of how our society operates today. Cloud computing is supported by many large data centers around the world, each containing thousands of machines. Resource disaggregation has recently been proposed as a means of improving data centers' memory and processor utilization, energy efficiency, and reliability by physically separating computing, memory, and storage resources into disaggregated components and connecting those components via a network fabric. However, such a physical separation poses three significant performance issues for tensor computations. First, when accessing disaggregated memory and storage resources frequently, there is a severe problem issue because of the unawareness of data location and data transfer overhead among the disaggregated resources via a network fabric. Second, because of the unique features of tensor computations (e.g., high data dimensionality and varying memory access patterns), existing approaches for disaggregated data centers that are tensor-agnostic can easily cause severe spatial and temporal locality issues in disaggregated compute, memory, and storage resources, and hence cause severe performance loss. Third, diverse memory characteristics (e.g., asymmetric latency, bandwidth, capacity, and persistency) in disaggregated memory resources incur significant performance loss because existing approaches do not consider those diverse memory characteristics in disaggregated data centers.

To address the performance issues, this future work aims to build tensor-aware, in-network memory management for tensor computations in disaggregated data centers. In this research project, I will (1) exploit emerging programmable network hardware (e.g., SmartNIC) in disaggregated data centers to facilitate performant

access to disaggregated resources connecting by a network fabric; (2) build unified memory abstraction based on unique tensor characteristics at a datacenter scale, where each disaggregated resource can access tensors effectively; (3) propose a scalable disaggregated memory scheduler to manage memory migration across disaggregated resources based on characteristics of diverse disaggregated memory types and runtime memory statistics.

Bibliography

[1] Kensho Hara, Hirokatsu Kataoka, and Yutaka Satoh. Learning spatio-temporal features with 3d residual networks for action recognition. In *Proceedings of the IEEE International Conference on Computer Vision Workshops*, pages 3154–3160, 2017. 1

[2] Yoon Kim. Convolutional neural networks for sentence classification. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1746–1751, 2014. 1

[3] Joseph Redmon, Santosh Divvala, Ross Girshick, and Ali Farhadi. You only look once: Unified, real-time object detection. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 779–788, 2016. 1

[4] Joon Hee Choi and S Vishwanathan. Dfacto: Distributed factorization of tensors. In *Advances in Neural Information Processing Systems*, pages 1296–1304, 2014. 1

[5] Evangelos E Papalexakis, Christos Faloutsos, and Nicholas D Sidiropoulos. Parcube: Sparse parallelizable tensor decompositions. In *Joint European Conference on Machine Learning and Knowledge Discovery in Databases*, pages 521–536. Springer, 2012. 1

[6] Evangelos E Papalexakis, Christos Faloutsos, and Nicholas D Sidiropoulos. Parcube: Sparse parallelizable candecomp-parafac tensor decomposition. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 10(1):1–25, 2015. 1

[7] Animashree Anandkumar, Rong Ge, Daniel Hsu, Sham M Kakade, and Matus Telgarsky. Tensor decompositions for learning latent variable models. *Journal of Machine Learning Research*, 15:2773–2832, 2014. 1

[8] Joyce C Ho, Joydeep Ghosh, and Jimeng Sun. Marble: high-throughput phenotyping from electronic health records via sparse nonnegative tensor factorization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 115–124, 2014. 1

[9] Yichen Wang, Robert Chen, Joydeep Ghosh, Joshua C Denny, Abel Kho, You Chen, Bradley A Malin, and Jimeng Sun. Rubik: Knowledge guided tensor

factorization and completion for health data analytics. In *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 1265–1274, 2015. 1

[10] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018. 1

[11] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. Language models are unsupervised multitask learners. *OpenAI Blog*, 1(8):9, 2019. 1

[12] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using gpu model parallelism. *arXiv preprint arXiv:1909.08053*, 2019. 1

[13] Nvidia data center deep learning product performance. https://developer.nvidia.com/deep-learning-performance-training-inference. 1

[14] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, 2017. 1

[15] Tim Cramer, Dirk Schmidl, Michael Klemm, and Dieter an Mey. Openmp programming on intel r xeon phi tm coprocessors: An early performance comparison. In *Proc. Many Core Appl. Res. Community (MARC) Symp*, pages 38–44, 2012. 1

[16] Matthew Curtis-Maury, Filip Blagojevic, Christos D Antonopoulos, and Dimitrios S Nikolopoulos. Prediction-based power-performance adaptation of multithreaded scientific codes. *IEEE Transactions on Parallel and Distributed Systems*, 19(10):1396–1410, 2008. 1

[17] TensorFlow Performance Guide. `https://www.tensorflow.org/performance/performance_guide`. 1, 6.1

[18] Andrzej Cichocki. Era of big data processing: A new approach via tensor networks and tensor decompositions. *CoRR*, abs/1403.2048, 2014. 2.2, 3.1, 4.1

[19] Matthew Fishman, Steven R White, and E Miles Stoudenmire. The ITensor software library for tensor network calculations. *arXiv preprint arXiv:2007.14822*, 2020. 2.2, 2.2, 3.1, 3.3.3, 4.1, 4.3.1, 4.4

[20] Christoph Koppl and Hans-Joachim Werner. Parallel and low-order scaling implementation of hartree–fock exchange using local density fitting. *Journal of chemical theory and computation*, 12(7):3122–3134, 2016. 2.2, 3.1, 4.1, 4.4

[21] Christoph Riplinger, Peter Pinski, Ute Becker, Edward F Valeev, and Frank Neese. Sparse maps—a systematic infrastructure for reduced-scaling electronic structure methods. ii. linear scaling domain based pair natural orbital coupled cluster theory. *The Journal of chemical physics*, 144(2):024109, 2016. 2.2, 3.1, 4.1, 4.4

[22] Lingjie Li, Wenjian Yu, and Kim Batselier. Faster tensor train decomposition for sparse data. *arXiv preprint arXiv:1908.02721*, 2019. 2.2, 3.1, 4.1, 4.4

[23] Chase Roberts, Ashley Milsted, Martin Ganahl, Adam Zalcman, Bruce Fontaine, Yijian Zou, Jack Hidary, Guifre Vidal, and Stefan Leichenauer. Tensornetwork: A library for physics and machine learning. *arXiv preprint arXiv:1905.01330*, 2019. 2.2, 3.1, 4.1, 4.4

[24] Edoardo Apra, Eric J Bylaska, Wibe A De Jong, Niranjan Govind, Karol Kowalski, Tjerk P Straatsma, Marat Valiev, HJJ van Dam, Yuri Alexeev, James Anchell, et al. Nwchem: Past, present, and future. *The Journal of chemical physics*, 152(18):184102, 2020. 2.2, 2.2, 3.1, 3.3.1, 4.1, 4.3.1, 4.3.5

[25] Jiawen Liu, Jie Ren, Roberto Gioiosa, Dong Li, and Jiajia Li. Sparta: High-performance, element-wise sparse tensor contraction on heterogeneous memory. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021. 2.2, 2.6, 4.1, 26, 4.2.2, 4.3.1, 4.3.2, 4.3.5, 4.4

[26] T Daniel Crawford and Henry F Schaefer. An introduction to coupled cluster theory for computational chemists. *Reviews in computational chemistry*, 14:33–136, 2000. 2.2, 4.1, 4.3.1, 4.3.5

[27] Tilman Esslinger. Fermi-hubbard physics with atoms in an optical lattice. *Annu. Rev. Condens. Matter Phys.*, 1(1):129–152, 2010. 2.2, 3.3.3, 4.3.4

[28] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016. 2.3, 5.3.3

[29] Samyam Rajbhandari, Yuxiong He, Olatunji Ruwase, Michael Carbin, and Trishul Chilimbi. Optimizing cnns on multicores for scalability, performance and goodput. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2017. 2.3

[30] Amrita Mathuriya, Deborah Bard, Peter Mendygral, Lawrence Meadows, James Arnemann, Lei Shao, Siyu He, Tuomas Karna, Daina Moise, Simon J Pennycook, et al. Cosmoflow: Using deep learning to learn the universe at scale. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 9. ACM, 2018. 2.3

[31] Yang You, Aydın Buluç, and James Demmel. Scaling deep learning on gpu and knights landing clusters. In *Proceedings of the International Conference for High*

Performance Computing, Networking, Storage and Analysis, page 9. ACM, 2017. 2.3

[32] Thorsten Kurth, Jian Zhang, Nadathur Satish, Evan Racah, Ioannis Mitliagkas, Md Mostofa Ali Patwary, Tareq Malas, Narayanan Sundaram, Wahid Bhimji, Mikhail Smorkalov, et al. Deep learning at 15pf: supervised and semi-supervised classification for scientific data. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 7. ACM, 2017. 2.3

[33] Chao Li, Yi Yang, Min Feng, Srimat Chakradhar, and Huiyang Zhou. Optimizing memory efficiency for deep convolutional neural networks on gpus. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, 2016. 2.3

[34] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. 2.3, 2.4

[35] Hans Vandierendonck, Polyvios Pratikakis, and Dimitrios S Nikolopoulos. Parallel programming of general-purpose programs using task-based programming models. In *3rd USENIX workshop on Hot Topics in Parallelism (HotPar 2011)*. USENIX Association, 2011. 2.4

[36] Yuxiong Zhu, Borui Wang, Dong Li, and Jishen Zhao. Integrated thermal analysis for processing in die-stacking memory. In *Proceedings of the Second International Symposium on Memory Systems*, pages 402–414, 2016. 2.5

[37] Jiajia Li, Jimeng Sun, and Richard Vuduc. HiCOO: Hierarchical storage of sparse tensors. In *Proceedings of the ACM/IEEE International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*, Dallas, TX, USA, November 2018. 3.1, 3.4, 4.1

[38] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Prasant Singh Rawat, Sriram Krishnamoorthy, and P. Sadayappan. An efficient mixed-mode representation of sparse tensors. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, pages 49:1–49:25, New York, NY, USA, 2019. ACM. 3.1, 3.4, 4.1

[39] Israt Nisa, Jiajia Li, Aravind Sukumaran-Rajam, Richard Vuduc, and P Sadayappan. Load-balanced sparse mttkrp on gpus. In *2019 IEEE International*

Parallel and Distributed Processing Symposium (IPDPS), pages 123–133. IEEE, 2019. 3.1, 3.4, 4.1

[40] Shaden Smith, Niranjay Ravindran, Nicholas Sidiropoulos, and George Karypis. SPLATT: Efficient and parallel sparse tensor-matrix multiplication. In *Proceedings of the 29th IEEE International Parallel & Distributed Processing Symposium*, IPDPS, 2015. 3.1, 13, 3.4, 4.1

[41] Jiajia Li, Bora Uçar, Ümit V. Çatalyürek, Jimeng Sun, Kevin Barker, and Richard Vuduc. Efficient and effective sparse tensor reordering. In *Proceedings of the ACM International Conference on Supercomputing*, ICS '19, pages 227–237, New York, NY, USA, 2019. ACM. 3.1, 4.1

[42] B. Liu, C. Wen, A. D. Sarwate, and M. M. Dehnavi. A unified optimization approach for sparse tensor operations on GPUs. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 47–57, Sept 2017. 3.1, 3.4, 4.1

[43] Shaden Smith and George Karypis. Accelerating the Tucker decomposition with compressed sparse tensors. In *European Conference on Parallel Processing*. Springer, 2017. 3.1, 13, 4.1

[44] Venkatesan T. Chakaravarthy, Jee W. Choi, Douglas J. Joseph, Prakash Murali, Shivmaran S. Pandian, Yogish Sabharwal, and Dheeraj Sreedhar. On optimizing distributed Tucker decomposition for sparse tensors. In *Proceedings of the 32nd ACM International Conference on Supercomputing*, ICS '18, 2018. 3.1, 4.1

[45] Jiajia Li, Jee Choi, Ioakeim Perros, Jimeng Sun, and Richard Vuduc. Model-driven sparse cp decomposition for higher-order tensors. In *2017 IEEE international parallel and distributed processing symposium (IPDPS)*, pages 1048–1057. IEEE, 2017. 3.1, 4.1

[46] O. Kaya and B. Uçar. Parallel Candecomp/Parafac decomposition of sparse tensors using dimension trees. *SIAM Journal on Scientific Computing*, 40(1):C99–C130, 2018. 3.1, 4.1

[47] Ioakeim Perros, Evangelos E. Papalexakis, Fei Wang, Richard Vuduc, Elizabeth Searles, Michael Thompson, and Jimeng Sun. SPARTan: Scalable PARAFAC2 for large & sparse data. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, pages 375–384, New York, NY, USA, 2017. ACM. 3.1, 4.1

[48] Shaden Smith and George Karypis. A medium-grained algorithm for distributed sparse tensor factorization. In *Parallel and Distributed Processing Symposium (IPDPS), 2016 IEEE International*. IEEE, 2016. 3.1, 4.1

[49] Jiajia Li, Yuchen Ma, Chenggang Yan, and Richard Vuduc. Optimizing sparse tensor times matrix on multi-core and many-core architectures. In *Proceedings*

of the Sixth Workshop on Irregular Applications: Architectures and Algorithms, IAˆ3 '16, pages 26–33, Piscataway, NJ, USA, 2016. IEEE Press. 3.1, 4.1

[50] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. High-performance sparse matrix-matrix products on intel knl and multicore architectures. In *Proceedings of the 47th International Conference on Parallel Processing Companion*, pages 1–10, 2018. 3.1, 13, 19, 3.4

[51] Rasmus Resen Amossen, Andrea Campagna, and Rasmus Pagh. Better size estimation for sparse matrix products. In *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques*, pages 406–419. Springer, 2010. 3.1, 13, 3.4

[52] Edith Cohen. On optimizing multiplications of sparse matrices. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 219–233. Springer, 1996. 3.1, 13

[53] Chong Peng, Justus A Calvin, Fabijan Pavosevic, Jinmei Zhang, and Edward F Valeev. Massively parallel implementation of explicitly correlated coupled-cluster singles and doubles using tiledarray framework. *The Journal of Physical Chemistry A*, 120(51):10231–10244, 2016. 3.1, 3.3.3, 4.1, 4.4

[54] Edgar Solomonik, Devin Matthews, Jeff Hammond, and James Demmel. Cyclops tensor framework: Reducing communication and eliminating load imbalance in massively parallel contractions. In *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pages 813–824. IEEE, 2013. 3.1, 3.3.3, 3.4, 4.1, 4.4

[55] Samuel Manzer, Evgeny Epifanovsky, Anna I Krylov, and Martin Head-Gordon. A general sparse tensor framework for electronic structure theory. *Journal of chemical theory and computation*, 13(3):1108–1116, 2017. 3.1, 3.3.3, 3.4, 4.1, 4.4

[56] Evgeny Epifanovsky, Michael Wormit, Tomasz Kuś, Arie Landau, Dmitry Zuev, Kirill Khistyaev, Prashant Manohar, Ilya Kaliman, Andreas Dreuw, and Anna I Krylov. New implementation of high-level correlated methods using a general block tensor library for high-performance electronic structure calculations. *Journal of computational chemistry*, 34(26):2293–2309, 2013. 3.1, 3.4, 4.1, 4.4

[57] Luiz E. Ramos, Eugene Gorbatov, and Ricardo Bianchini. Page Placement in Hybrid Memory Systems. In *International Conference on Supercomputing (ICS)*, May 2011. 3.1, 3.2.2, 3.4, 4.1, 4.2.2

[58] H. Yoon, J. Meza, R. Ausavarungnirun, R. A. Harding, and O. Mutlu. Row buffer locality aware caching policies for hybrid memories. In *2012 IEEE 30th International Conference on Computer Design (ICCD)*, 2012. 3.1, 3.2.2, 3.4, 4.1, 4.2.2

[59] Wei Wei, Dejun Jiang, Sally A. McKee, Jin Xiong, and Mingyu Chen. Exploiting Program Semantics to Place Data in Hybrid Memory. In *PACT*, 2015. 3.1, 3.2.2, 3.4, 4.1, 4.2.2

[60] Neha Agarwal and Thomas F. Wenisch. Thermostat: Application-transparent page management for two-tiered main memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017*, pages 631–644, 2017. 3.1, 3.2.2, 3.4, 4.1, 4.2.2, 4.4

[61] Takahiro Hirofuchi and Ryousei Takano. Raminate: Hypervisor-based virtualization for hybrid main memory systems. In *Proceedings of the Seventh ACM Symposium on Cloud Computing*, SoCC '16, pages 112–125, New York, NY, USA, 2016. ACM. 3.1, 3.2.2, 3.4, 4.1, 4.2.2, 4.4

[62] S. Kannan, A. Gavrilovska, V. Gupta, and K. Schwan. Heteroos — os design for heterogeneous memory management in datacenter. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 521–534, June 2017. 3.1, 3.2.2, 3.4, 4.1, 4.2.2, 4.3.1, 4.4

[63] K. Wu, Y. Huang, and D. Li. Unimem: Runtime Data Management on Non-Volatile Memory-based Heterogeneous Main Memory. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, 2017. 3.1, 3.2.2, 3.4, 4.1, 4.2.2, 4.3.1, 4.4

[64] Kai Wu, Jie Ren, and Dong Li. Runtime Data Management on Non-Volatile Memory-Based Heterogeneous Memory for Task Parallel Programs. In *ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis*, 2018. 3.1, 3.2.2, 3.4, 4.1, 4.2.2, 4.4

[65] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble page management for tiered memory systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, pages 331–345, New York, NY, USA, 2019. ACM. 3.1, 3.2.2, 3.4, 4.1, 4.2.2, 4.4

[66] Subramanya R. Dulloor, Amitabha Roy, Zheguang Zhao, Narayanan Sundaram, Nadathur Satish, Rajesh Sankaran, Jeff Jackson, and Karsten Schwan. Data Tiering in Heterogeneous Memory Systems. In *European Conference on Computer Systems*, 2016. 3.1, 3.2.2, 3.4, 4.1, 4.2.2

[67] Seongdae Yu, Seongbeom Park, and Woongki Baek. Design and Implementation of Bandwidth-aware Memory Placement and Migration Policies for Heterogeneous Memory Systems. In *International Conference on Supercomputing (ICS)*, 2017. 3.1, 3.2.2, 3.4, 4.1, 4.2.2

[68] Fred G Gustavson. Two fast algorithms for sparse matrices: Multiplication and permuted transposition. *ACM Transactions on Mathematical Software (TOMS)*, 4(3):250–269, 1978. 1, 13, 3.4

[69] John R Gilbert, Cleve Moler, and Robert Schreiber. Sparse matrices in matlab: Design and implementation. *SIAM Journal on Matrix Analysis and Applications*, 13(1):333–356, 1992. 13

[70] Richard Wilson Vuduc and James W Demmel. *Automatic performance tuning of sparse matrix kernels*, volume 1. University of California, Berkeley Berkeley, CA, 2003. 13

[71] Wikipedia. Hash table. https://en.wikipedia.org/wiki/Hash_table, Dec 2020. 19

[72] Yusuke Nagasaka, Satoshi Matsuoka, Ariful Azad, and Aydın Buluç. Performance optimization, modeling and analysis of sparse matrix-matrix products on multi-core and many-core processors. *Parallel Computing*, 90:102545, 2019. 19, 3.4

[73] Yusuke Nagasaka, Akira Nukada, and Satoshi Matsuoka. High-performance and memory-saving sparse general matrix-matrix multiplication for nvidia pascal gpu. In *2017 46th International Conference on Parallel Processing (ICPP)*, pages 101–110. IEEE, 2017. 19, 3.4

[74] Pham Nguyen Quang Anh, Rui Fan, and Yonggang Wen. Balanced hashing and efficient gpu sparse general matrix-matrix multiplication. In *Proceedings of the 2016 International Conference on Supercomputing*, pages 1–12, 2016. 19, 3.4

[75] Mehmet Deveci, Christian Trott, and Sivasankaran Rajamanickam. Performance-portable sparse matrix-matrix multiplication for many-core architectures. In *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 693–702. IEEE, 2017. 19, 3.4

[76] Ariful Azad, Grey Ballard, Aydin Buluc, James Demmel, Laura Grigori, Oded Schwartz, Sivan Toledo, and Samuel Williams. Exploiting multiple levels of parallelism in sparse matrix-matrix multiplication. *SIAM Journal on Scientific Computing*, 38(6):C624–C651, 2016. 19, 3.4

[77] Jian Yang, Juno Kim, Morteza Hoseinzadeh, Joseph Izraelevitz, and Steve Swanson. An empirical guide to the behavior and use of scalable persistent memory. In *18th USENIX Conference on File and Storage Technologies (FAST 20)*, 2020. 3.2.2, 4.4

[78] Jiawen Liu, Hengyu Zhao, Matheus A Ogleari, Dong Li, and Jishen Zhao. Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 655–668. IEEE, 2018. 3.2.2, 3.4, 4.1, 4.2.2

[79] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Nimble Page Management for Tiered Memory Systems. In *ASPLOS*, 2019. 3.2.2, 3.3.5, 4.3.3

[80] Shaden Smith, Jee W Choi, Jiajia Li, Richard Vuduc, Jongsoo Park, Xing Liu, and George Karypis. Frostt: The formidable repository of open sparse tensors and tools, 2017. 3.3.1, 4.3.1

[81] Evgeny Epifanovsky, Karol Kowalski, Peng-Dong Fan, Marat Valiev, Spiridoula Matsika, and Anna I Krylov. On the electronically excited states of uracil. *The Journal of Physical Chemistry A*, 112(40):9983–9992, 2008. 3.3.1

[82] Matthew Fishman, Steven R. White, and E. Miles Stoudenmire. ITensor: A C++ library for efficient tensor network calculations. Available from `https://github.com/ITensor/ITensor`, August 2020. 3.3.3, 4.3.4

[83] Fredrik Kjolstad, Shoaib Kamil, Stephen Chou, David Lugato, and Saman Amarasinghe. The tensor algebra compiler. *Proc. ACM Program. Lang.*, 1(OOPSLA):77:1–77:29, October 2017. 3.3.3, 3.4

[84] Zi Yan, Daniel Lustig, David Nellans, and Abhishek Bhattacharjee. Repository of Nimble Page Management for Tiered Memory Systems in ASPLOS2019. Available from `https://github.com/ysarch-lab/nimble_page_management_asplos_2019`, July 2020. 3.3.5, 4.3.3

[85] Jinsung Kim, Aravind Sukumaran-Rajam, Vineeth Thumma, Sriram Krishnamoorthy, Ajay Panyala, Louis-Noël Pouchet, Atanas Rountev, and Ponnuswamy Sadayappan. A code generator for high-performance tensor contractions on gpus. In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 85–95. IEEE, 2019. 3.4, 4.4

[86] Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, and P Sadayappan. Analytical cache modeling and tilesize optimization for tensor contractions. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–13, 2019. 3.4, 4.4

[87] Jinsung Kim, Aravind Sukumaran-Rajam, Changwan Hong, Ajay Panyala, Rohit Kumar Srivastava, Sriram Krishnamoorthy, and Ponnuswamy Sadayappan. Optimizing tensor contractions in ccsd (t) for efficient execution on gpus. In *Proceedings of the 2018 International Conference on Supercomputing*, pages 96–106, 2018. 3.4, 4.4

[88] Pai-Wei Lai, Kevin Stock, Samyam Rajbhandari, Sriram Krishnamoorthy, and Ponnuswamy Sadayappan. A framework for load balancing of tensor contraction expressions via dynamic task partitioning. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10, 2013. 3.4, 4.1, 26, 4.4

[89] Albert Hartono, Qingda Lu, Thomas Henretty, Sriram Krishnamoorthy, Huaijian Zhang, Gerald Baumgartner, David E Bernholdt, Marcel Nooijen, Russell Pitzer, J Ramanujam, et al. Performance optimization of tensor contraction expressions for many-body methods in quantum chemistry. *The Journal of Physical Chemistry A*, 113(45):12715–12723, 2009. 3.4, 4.4

[90] Alexander A Auer, Gerald Baumgartner, David E Bernholdt, Alina Bibireata, Venkatesh Choppella, Daniel Cociorva, Xiaoyang Gao, Robert Harrison, Sriram Krishnamoorthy, Sandhya Krishnan, et al. Automatic code generation for many-body electronic structure methods: the tensor contraction engine. *Molecular Physics*, 104(2):211–228, 2006. 3.4, 4.4

[91] Edgar Solomonik, Devin Matthews, Jeff R Hammond, John F Stanton, and James Demmel. A massively parallel tensor contraction framework for coupled-cluster computations. *Journal of Parallel and Distributed Computing*, 74(12):3176–3190, 2014. 3.4, 4.4

[92] So Hirata. Tensor contraction engine: Abstraction and automated parallel implementation of configuration-interaction, coupled-cluster, and many-body perturbation theories. *The Journal of Physical Chemistry A*, 107(46):9887–9897, 2003. 3.4, 4.4

[93] Devin Matthews. High-performance tensor contraction without BLAS. *CoRR*, abs/1607.00291, 2016. 3.4, 4.4

[94] Jean Kossaifi, Yannis Panagakis, Anima Anandkumar, and Maja Pantic. TensorLy: Tensor learning in Python. *CoRR*, abs/1610.09555, 2018. 3.4, 4.4

[95] Y. Shi, U. N. Niranjan, A. Anandkumar, and C. Cecka. Tensor contractions with extended BLAS kernels on CPU and GPU. In *2016 IEEE 23rd International Conference on High Performance Computing (HiPC)*, pages 193–202, Dec 2016. 3.4, 4.4

[96] Daniel Kats and Frederick R Manby. Sparse tensor framework for implementation of general local correlation methods. *The Journal of Chemical Physics*, 138(14):144101, 2013. 3.4, 4.1, 4.4

[97] David Ozog, Jeff R Hammond, James Dinan, Pavan Balaji, Sameer Shende, and Allen Malony. Inspector-executor load balancing algorithms for block-sparse tensor contractions. In *2013 42nd International Conference on Parallel Processing*, pages 30–39. IEEE, 2013. 3.4, 4.1, 4.4

[98] Ilia Sivkov, Patrick Seewald, Alfio Lazzaro, and Jürg Hutter. DBCSR: A blocked sparse tensor algebra library. *arXiv preprint arXiv:1910.13555*, 2019. 3.4, 4.1, 4.4

[99] Thomas Hérault, Yves Robert, George Bosilca, Robert Harrison, Cannada Lewis, and Edward Valeev. *Distributed-memory multi-GPU block-sparse tensor*

contraction for electronic structure. PhD thesis, Inria-Research Centre Grenoble–Rhône-Alpes, 2020. 3.4, 4.1, 4.4

[100] Ryan Levy, Edgar Solomonik, and Bryan K Clark. Distributed-memory dmrg via sparse and dense parallel tensor contractions. *arXiv preprint arXiv:2007.05540*, 2020. 3.4, 4.1, 4.4

[101] M. Baskaran, B. Meister, N. Vasilache, and R. Lethin. Efficient and scalable computations with sparse tensors. In *High Performance Extreme Computing (HPEC), 2012 IEEE Conference on*, pages 1–6, Sept 2012. 3.4

[102] Weifeng Liu and Brian Vinter. An efficient gpu general sparse matrix-matrix multiplication for irregular data. In *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pages 370–381. IEEE, 2014. 3.4

[103] Baotong Lu, Xiangpeng Hao, Tianzheng Wang, and Eric Lo. Dash: scalable hashing on persistent memory. *arXiv preprint arXiv:2003.07302*, 2020. 3.4

[104] Jie Ren, Jiaolin Luo, Kai Wu, Minjia Zhang, Hyeran Jeon, and Dong Li. Sentinel: Efficient tensor migration and allocation on heterogeneous memory systems for deep learning. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 598–611. IEEE, 2021. 4.1, 4.2.2

[105] Jie Ren, Minjia Zhang, and Dong Li. HM-ANN: Efficient Billion-Point Nearest Neighbor Search on Heterogeneous Memory. In *Neurips*, 2020. 4.1, 4.2.2

[106] Linjian Ma, Jiayu Ye, and Edgar Solomonik. Autohoot: Automatic high-order optimization for tensors. *arXiv preprint arXiv:2005.04540*, 2020. 26, 4.4

[107] Samuel Webb Williams. *Auto-tuning performance on multicore computers*. University of California, Berkeley Berkeley, CA, 2008. 26

[108] Zhenyuan Ruan, Malte Schwarzkopf, Marcos K. Aguilera, and Adam Belay. AIFM: High-Performance, Application-Integrated Far Memory. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2020. 4.2.2

[109] Kai Wu, Jie Ren, and Dong Li. Runtime data management on non-volatile memory-based heterogeneous memory for task-parallel programs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis*, page 31. IEEE Press, 2018. 4.3.1

[110] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 265–283, 2016. 4.4, 5.3.3

[111] Jie Ren, Samyam Rajbhandari, Reza Yazdani Aminabadi, Olatunji Ruwase, Shuangyan Yang, Minjia Zhang, Dong Li, and Yuxiong He. Zero-offload: Democratizing billion-scale model training. In *USENIX Annual Technical Conference (ATC)*, 2021. 4.4

[112] Zhen Xie, Wenqian Dong, Jie Liu, Ivy Peng, Yanbao Ma, and Dong Li. Md-hm: Memoization-based molecular dynamics simulations on big memory system. In *Proceedings of the 35th ACM International Conference on Supercomputing*, 2021. 4.4

[113] Jie Ren, Kai Wu, and Dong Li. Exploring non-volatility of non-volatile memory for high performance computing under failures. In *2020 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 237–247. IEEE, 2020. 4.4

[114] Chenxi Wang, Huimin Cui, Ting Cao, John Zigman, Haris Volos, Onur Mutlu, Fang Lv, Xiaobing Feng, and Guoqing Harry Xu. Panthera: Holistic memory management for big data processing over hybrid memories. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, 2019. 4.4

[115] Assaf Eisenman, Darryl Gardner, Islam AbdelRahman, Jens Axboe, Siying Dong, Kim Hazelwood, Chris Petersen, Asaf Cidon, and Sachin Katti. Reducing DRAM Footprint with NVM in Facebook. In *Proceedings of the Thirteenth EuroSys Conference*, 2018. 4.4

[116] Gurbinder Gill, Roshan Dathathri, Loc Hoang, Ramesh Peri, and Keshav Pingali. Single machine graph analytics on massive datasets using intel optane dc persistent memory, 2019. 4.4

[117] Yu Chen, Ivy B. Peng, Zhen Peng, Xu Liu, and Bin Ren. Atmem: Adaptive data placement in graph applications on heterogeneous memories. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization*, CGO 2020, 2020. 4.4

[118] R. Madhava Krishnan, Jaeho Kim, Ajit Mathew, Xinwei Fu, Anthony Demeri, Changwoo Min, and Sudarsun Kannan. Durable transactional memory can scale with timestone. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '20, 2020. 4.4

[119] Se Kwon Lee, Jayashree Mohan, Sanidhya Kashyap, Taesoo Kim, and Vijay Chidambaram. Recipe: Converting concurrent dram indexes to persistent-memory indexes. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, 2019. 4.4

[120] Jie Liu, Jiawen Liu, Wan Du, and Dong Li. Performance analysis and characterization of training deep learning models on mobile device. In *2019 IEEE 25th*

International Conference on Parallel and Distributed Systems (ICPADS), pages 506–515. IEEE, 2019. 4.4

[121] Bang Di, Jiawen Liu, Hao Chen, and Dong Li. Fast, flexible, and comprehensive bug detection for persistent memory programs. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 503–516, 2021. 4.4

[122] Jiawen Liu, Zhen Xie, Dimitrios Nikolopoulos, and Dong Li. {RIANN}: Real-time incremental learning with approximate nearest neighbor on mobile devices. In *2020 {USENIX} Conference on Operational Machine Learning (OpML 20)*, 2020. 4.4

[123] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014. 5.1, 5.3.3

[124] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012. 5.1, 5.3.3

[125] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *CoRR*, abs/1511.06434, 2015. 5.1, 5.3.3, 6.2.2

[126] TensorBoard: Visualizing learning, https://www.tensorflow.org/programmers_gu ide/summa ries_and_tensorboard. 5.1.1, 6.1

[127] Intel, Vtune user's guide, https://software.intel.com/en-us/get-started-with-vtune/. 5.1.1, 5.3.4

[128] Ping Chi, Shuangchen Li, Cong Xu, Tao Zhang, Jishen Zhao, Yongpan Liu, Yu Wang, and Yuan Xie. PRIME: A novel processing-in-memory architecture for neural network computation in ReRAM-based main memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 27–39, 2016. 5.1.1, 5.1.1, 5.2.4, 5.5, 5.5.1

[129] Duckhwan Kim, Jaeha Kung, Sek Chai, Sudhakar Yalamanchili, and Saibal Mukhopadhyay. Neurocube: A programmable digital neuromorphic architecture with high-density 3D memory. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 380–392, 2016. 5.1.1, 5.1.1, 5.4.3, 5.5.1

[130] Scott Boag, Parijat Dube, Benjamin Herta, Waldemar Hummer, Vatche Ishakian, Jayaram K. R., Michael Kalantar, Vinod Muthusamy, Priya Nagpurkar, and Florian Rosenberg. Scalable Multi-Framework Multi-Tenant Lifecycle Management of Deep Learning Training Jobs. In *Workshop on ML Systems at NIPS'17*, 2017. 5.1.1

[131] MultiModel: Multi-task machine learning across domains, https://ai.googleblog.com/2017/06/multimodel-multi-task-machine-learning.html. 5.1.1

[132] OpenCL. http://www.khronos.org/opencl/. 5.1.2

[133] NVIDIA, TITAN Xp, https://www.nvidia.com/en-us/geforce/products/10series/titan-xp/. 5.1.3

[134] Openarc. https://ft.ornl.gov/research/openarc. 5.2.2, 5.2.2

[135] Ipmacc compiler. https://github.com/lashgar/ipmacc. 5.2.2, 5.2.2

[136] OpenMP Forum. OpenMP Fortran application program interface, version 1.1. http://www.openmp.org, 1999. 5.2.2

[137] Hans van Halteren, Jakub Zavrel, and Walter Daelemans. Improving accuracy in word class tagging through the combination of machine learning systems. *Computational linguistics*, 27(2):199–229, 2001. 5.2.3

[138] Sheng Li, Jung Ho Ahn, Richard D. Strong, Jay B. Brockman, Dean M. Tullsen, and Norman P. Jouppi. McPAT: An integrated power, area, and timing modeling framework for multicore and manycore architectures. In *Proceedings of the 42Nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 469–480, 2009. 5.2.4, 5.3.2

[139] Kevin Skadron, Mircea R. Stan, Karthik Sankaranarayanan, Wei Huang, Sivakumar Velusamy, and David Tarjan. Temperature-aware microarchitecture: modeling and implementation. *ACM Transactions on Architecture and Code Optimization*, 1(1):94–125, 2004. 5.2.4

[140] Synopsys. Design compiler. https://www.synopsys.com/support/training/rtl-synthesis/design-compiler-rtl-synthesis.html. 5.3.1, 5.3.2

[141] Synopsys. Primetime. https://www.synopsys.com/support/training/signoff/primetime1-fcd.html. 5.3.1, 5.3.2

[142] HMCC. Hybrid memory cube specification 2.0. http://http://www.hybridmemorycube.org/. 5.3.1

[143] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. 5.3.1

[144] James Reinders. Vtune performance analyzer essentials. *Intel Press*, 2005. 5.3.2, 6.1

[145] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2016. 5.3.3, 6.1.1

[146] Wojciech Zaremba, Ilya Sutskever, and Oriol Vinyals. Recurrent neural network regularization. *arXiv preprint arXiv:1409.2329*, 2014. 5.3.3

[147] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013. 5.3.3

[148] Yann LeCun and Corinna Cortes. MNIST handwritten digit database. *Computational linguistics*, 2010. 5.3.3

[149] Tensorflow, questions-words dataset, http://download.tensorflow.org/data/questions-words.txt. 5.3.3

[150] NVIDIA, GeForce GTX 1080 Ti, https://www.nvidia.com/en-us/geforce/products/. 5.3.4

[151] NVIDIA CUDA. `http://www.nvidia.com/cuda`. 5.3.4

[152] cudnn. `https://developer.nvidia.com/cudnn`. 5.3.4

[153] NVIDIA, Profiler user's guide, http://docs.nvidia.com/cuda/profiler-users-guide/. 5.3.4

[154] Tian Li, Jie Zhong, Ji Liu, Wentao Wu, and Ce Zhang. Ease.ml: Towards multi-tenant resource sharing for machine learning workloads. *Proc. VLDB Endow.*, 11(5), 2018. 5.4.6

[155] Gabriel H. Loh, Nuwan Jayasena, Mark H. Oskin, Mark Nutter, David Roberts, Mitesh Meswani, Dongping Zhang, and Mike Ignatowski. A processing-in-memory taxonomy and a case for studying fixed-function PIM. In *WoNDP*, pages 1–6, 2013. 5.5

[156] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, pages 336–348, 2015. 5.5, 5.5.2

[157] Berkin Akin, Franz Franchetti, and James C. Hoe. Data reorganization in memory using 3D-stacked DRAM. In *Proceedings of the Annual International Symposium on Computer Architecture*, pages 131–143, 2015. 5.5, 5.5.2

[158] Lifeng Nai and Hyesoon Kim. Instruction offloading with HMC 2.0 standard: A case study for graph traversals. In *Proceedings of the 2015 International Symposium on Memory Systems*, pages 258–261, 2015. 5.5

[159] Yasuko Eckert, Nuwan Jayasena, and Gabriel H. Loh. Thermal feasibility of die-stacked processing in memory. In *WoNDP*, pages 1–6, 2014. 5.5

[160] Junwhan Ahn, Sungpack Hong, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. A scalable processing-in-memory accelerator for parallel graph processing. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, pages 105–117, 2015. 5.5, 5.5.2

[161] Daichi Fujiki, Scott Mahlke, and Reetuparna Das. In-memory data parallel processor. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '18, pages 1–14, New York, NY, USA, 2018. ACM. 5.5, 5.5.2

[162] Ali Shafiee, Anirban Nag, Naveen Muralimanohar, Rajeev Balasubramonian, John Paul Strachan, Miao Hu, R Stanley Williams, and Vivek Srikumar. ISAAC: A convolutional neural network accelerator with in-situ analog arithmetic in crossbars. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 14–26, 2016. 5.5.1

[163] Peiqi Wang, Yu Ji, Chi Hong, Yongqiang Lyu, Dongsheng Wang, and Yuan Xie. SNrram: an efficient sparse neural network computation architecture based on resistive random-access memory. In *Proceedings of the 55th Annual Design Automation Conference*, page 106. ACM, 2018. 5.5.1

[164] Shaahin Angizi, Zhezhi He, Adnan Siraj Rakin, and Deliang Fan. CMP-PIM: an energy-efficient comparator-based processing-in-memory neural network accelerator. In *Proceedings of the 55th Annual Design Automation Conference*, pages 105–110. ACM, 2018. 5.5.1

[165] Erfan Azarkhish, Davide Rossi, Igor Loi, and Luca Benini. Neurostream: Scalable and energy efficient deep learning with smart memory cubes. *IEEE Transactions on Parallel and Distributed Systems*, 29(2):420–434, 2018. 5.5.1

[166] Fabian Schuiki, Michael Schaffner, Frank K. Gürkaynak, and Luca Benini. A scalable near-memory architecture for training deep neural networks on large in-memory datasets. *arXiv*, abs/1803.04783, 2018. 5.5.1

[167] RISC-V: The free and open RISC instruction set architecture. `https://riscv.org/`. 5.5.1

[168] Dongping Zhang, Nuwan Jayasena, Alexander Lyashevsky, Joseph L. Greathouse, Lifan Xu, and Michael Ignatowski. TOP-PIM: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing*, pages 85–98, 2014. 5.5.2

[169] Brandon Reagen, Paul Whatmough, Robert Adolf, Saketh Rama, Hyunkwang Lee, Sae Kyu Lee, José Miguel Hernández-Lobato, Gu-Yeon Wei, and David

Brooks. Minerva: Enabling low-power, highly-accurate deep neural network accelerators. In *Proceedings of the 43rd International Symposium on Computer Architecture*, pages 267–278, 2016. 5.5.3

[170] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks. In *ISCA*, pages 367–379. IEEE, 2016. 5.5.3

[171] Hadi Esmaeilzadeh, Adrian Sampson, and Luis Ceze et al. Neural acceleration for general-purpose approximate programs. In *MICRO*, pages 449–460. IEEE Computer Society, 2012. 5.5.3

[172] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. Fused-layer CNN accelerators. In *MICRO*, pages 1–12. IEEE, 2016. 5.5.3

[173] Y. Chen, T. Luo, S. Liu, S. Zhang, L. He, J. Wang, L. Li, T. Chen, Z. Xu, N. Sun, and O. Temam. DaDianNao: A machine-learning supercomputer. In *IEEE/ACM International Symposium on Microarchitecture*, 2014. 5.5.3

[174] Minsoo Rhu, Natalia Gimelshein, and Jason Clemons et al. vDNN: Virtualized deep neural networks for scalable, memory-efficient neural network design. In *MICRO*, 2016. 5.5.3

[175] Minsoo Rhu, Mike O'Connor, Niladrish Chatterjee, Jeff Pool, Youngeun Kwon, and Stephen W Keckler. Compressing DMA engine: Leveraging activation sparsity for training deep neural networks. In *HPCA*, pages 78–91, 2018. 5.5.3

[176] Animesh Jain, Amar Phanishayee, Jason Mars, Lingjia Tang, and Gennady Pekhimenko. Gist: Efficient data encoding for deep neural network training. In *ISCA*, pages 1–14, 2018. 5.5.3

[177] TensorFlow at NERSC. `http://www.nersc.gov/users/data-analytics/data-analytics-2/deep-learning/using-tensorflow-at-nersc/`. 6.1

[178] Jiawen Liu, Hengyu Zhao, Matheus Almeida Ogleari, Dong Li, and Jishen Zhao. Processing-in-memory for energy-efficient neural network training: A heterogeneous approach. In *Proceedings of the 51th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 1–14. ACM, 2018. 6.1.1, 6.5.1

[179] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003. 6.2.1

[180] Abdelhafid Mazouz, Denis Barthou, et al. Performance evaluation and analysis of thread pinning strategies on multi-core platforms: Case study of spec omp applications on intel architectures. In *High Performance Computing and Simulation (HPCS), 2011 International Conference on*, pages 273–279. IEEE, 2011. 6.2.2

[181] TensorFlow Models. `https://github.com/tensorflow/models`. 6.3.1

[182] A tensorflow implementation of Deep Convolutional Generative Adversarial Networks. `https://github.com/carpedm20/DCGAN-tensorflow`. 6.3.1

[183] Azalia Mirhoseini, Hieu Pham, Quoc V Le, Benoit Steiner, Rasmus Larsen, Yuefeng Zhou, Naveen Kumar, Mohammad Norouzi, Samy Bengio, and Jeff Dean. Device placement optimization with reinforcement learning. *arXiv preprint arXiv:1706.04972*, 2017. 6.5.1

[184] Azalia Mirhoseini, Anna Goldie, Hieu Pham, Benoit Steiner, Quoc V Le, and Jeff Dean. A hierarchical model for device placement. *arXiv preprint arXiv:1406.3897*, 2018. 6.5.1

[185] Danijar Hafner, James Davidson, and Vincent Vanhoucke. Tensorflow agents: Efficient batched reinforcement learning in tensorflow. *CoRR*, abs/1709.02878, 2017. 6.5.1

[186] Yuanxiang Gao, Li Chen, and Baochun Li. Spotlight: Optimizing device placement for training deep neural networks. In *International Conference on Machine Learning*, pages 1662–1670, 2018. 6.5.1

[187] Pierangelo Di Sanzo, Francesco Del Re, Diego Rughetti, Bruno Ciciani, and Francesco Quaglia. Regulating concurrency in software transactional memory: An effective model-based approach. In *Self-adaptive and self-organizing systems, 2013 ieee 7th international conference on*, pages 31–40, 2013. 6.5.2

[188] Kishore Kumar Pusukuri, Rajiv Gupta, and Laxmi N Bhuyan. Thread reinforcer: Dynamically determining number of threads via os level monitoring. In *Workload Characterization (IISWC), 2011 IEEE International Symposium on*, pages 116–125. IEEE, 2011. 6.5.2

[189] Diego Rughetti, Pierangelo Di Sanzo, Bruno Ciciani, and Francesco Quaglia. Machine learning-based self-adjusting concurrency in software transactional memory systems. In *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*, pages 278–285. IEEE, 2012. 6.5.2