

**ADVANCING COMPILER OPTIMIZATIONS FOR GENERAL-PURPOSE &
DOMAIN-SPECIFIC PARALLEL ARCHITECTURES**

A Dissertation
Presented to
The Academic Faculty

By

Prasanth Chatarasi

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

December 2020

Copyright © Prasanth Chatarasi 2020

**ADVANCING COMPILER OPTIMIZATIONS FOR GENERAL-PURPOSE &
DOMAIN-SPECIFIC PARALLEL ARCHITECTURES**

Approved by:

Dr. Vivek Sarkar, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Jun Shirako, Co-Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Dr. Tushar Krishna
School of Electrical and Computer
Engineering
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science
and Engineering
Georgia Institute of Technology

Date Approved: July 27, 2020

“Dream the impossible. Know that you are born in this world to do something wonderful and unique; don’t let this opportunity pass by. Give yourself the freedom to dream and think big.” — Sri Sri Ravi Shankar

*To universal consciousness,
To my family,
To my advisors, mentors, teachers, and friends.*

ACKNOWLEDGEMENTS

Taittiriya Upanishad, Shikshavalli I.20

मातृदेवो भव पितृदेवो भव आछार्यदेवो भव अतिथिदेवो भव

mātrudevo bhava pitrudevo bhava āchāryadevo bhava atithidevo bhava

“*Respects to Mother, Father, Guru, and Guest. They are all forms of God*”.

First and foremost, I would like to express my gratitude to my mother *Ch. Anjanee Devi* and my father *Dr. C. V. Subbaiah* for always being there for me, and loving me unconditionally throughout the situations of extreme happiness to sadness. Without the two of you, I don’t know where I would be. If I have learned anything while being away from you, it is that you are the most important people in my life, and I love you both more than anything.

I want to express my sincere appreciation to my guru (advisor) *Prof. Vivek Sarkar*, who always had time for me when I needed him – no matter whether the reason was a technical discussion, an administrative problem, academic development, or the planning of my next steps. There are no words that can express my gratitude towards your efforts. Thank you for believing in me and supporting me. Without you, I would not be able to handle everything that the graduate program and life threw. Having you backing me up 100% allows me to be at peace and do research. You’re much more than an advisor, and thanks for helping me in my personal life too. Thanks for being my inspiration in the path of being a useful scientific researcher to this world.

I am also very grateful to my co-advisor *Dr. Jun Shirako*, with whom I had many very fruitful discussions on various topics of this thesis. Thank you for always being supportive, even when I feel like I can’t do it. Thank you for being a humble teacher in explaining the answers to my questions – Arigatou Gozaimasu! The lunch sessions were great to discuss my ideas and build a relationship with you.

I want to thank the rest of my committee *Prof. Tushar Krishna, Prof. Santosh Pande,* and *Prof. Rich Vuduc* for agreeing to be part of my thesis committee. I thank you all for your time, feedback, and suggestions in significantly improving the thesis. I always had a great time interacting with you at the school corridors, labs, and conferences. I am very fortunate to collaborate with Tushar Krishna’s group, and thanks, Tushar, for being an integral part of my Ph.D. journey, including the job search.

I’m also very grateful to my collaborators, i.e., Albert Cohen from INRIA/Google AI, Tushar Krishna and Hyoukjun Kwon from Synergy Research Group, Angshuman Parashar and Micheal Pellauer from NVIDIA Architecture Research Group, Stephen Neuendorf-

fer, Samuel Bayliss, and Kees Vissers from Xilinx Research Labs, Karthik Murthy from Google AI, and John Mellor-Crummey from Rice University. Thanks for enriching my research experience in the Ph.D. journey, including asking the right questions, thinking critically, positioning the work, writing research papers, giving a broader perspective of the researcher, and so on. I also want to acknowledge the Habanero Extreme Scale Software Research Group, Synergy Research Group, and CRNCH Research Center (Jeffery Young and Jason Riedy) for the research interactions in my Ph.D. journey.

In addition to the collaborators mentioned above, I also want to acknowledge and appreciate my mentors, especially Dr. Milind Chabbi, Dr. Shams Imam, Dr. Deepak Majeti, and Dr. Rishi Surendran for all the helpful advice in both research and personal life. I cannot thank you enough for everything you taught me during my Ph.D. journey. I greatly value your kindness and the expertise you imparted to me as my mentors.

I want to acknowledge my professor *Prof. Kesav Nori* for piquing my curiosity about compilers during my undergraduate study at IIT Hyderabad. Without you, I would not be where I am today. Also, thanks are due to my bachelor thesis advisors Dr. Aditya Nori, Prof. M. V. Panduranga Rao and Prof. Bheemarjuna Reddy for giving me a research exposure in the undergraduate study. Thanks to all my professors who encouraged me to apply for graduate studies and quickly provide reference letters.

Thanks are due to my friends for all the support and encouragement during my stay at Georgia Tech. There are too many of you to mention, but I would especially like to thank Prithayan Barua, Pramod Chunduri, Poulami Das, Anirudh Jain, Hyoukjun Kwon, Ankush Mandal, Girish Mururu, Mayank Parasar, Sriraj Paul, Christopher Porter, Gururaj Saileshwar, Anand Samajdar, Srisehan Srikanth, Kavitha Sthanam, Lechen Yu, and Jimmy Zhong. Thanks for always being up for a good laugh over the years. I would also like to thank the School of Computer Science department staff members for all the help I received during my stay at GaTech, including Ruthie Book and Wanda Purinton.

Also, I want to acknowledge Sharon Riehl for being the coach in my leadership development as part of the Georgia Tech Leading Edge program and helping me to self-analyze and improve in many aspects, including establishing trust and taking and giving constructive feedback, and prioritization. Furthermore, an essential aspect of my Ph.D. journey is the association with the SKY meditation club at GaTech and also the Art of Living foundation. The association helped me much towards the spiritual path and also managing of my mental health – in that regards, I want to acknowledge many people, especially my teachers Sareena Nagpal, Rachana Gadhok, Kunwar Gadhok, Preeti Bhatt.

The acknowledgments never end without mentioning siblings, i.e., my elder sister Sree Pavani and my elder brother Sreenivas. First and for most, life may not be that exciting

without you. We may fight 50% of the time, but I love you a lot for being my best friends, toughening up during tough situations of my life, and celebrating with me during happy moments. As per a Vietnamese proverb, you both are as close as my hands and feet. Also, my life has been more beautiful with my two nieces, i.e., Venkata Tejaswini and Sai Sughandhini – I always felt content hearing the word “mamayya” from them.

“Gratitude to the entire creation for showering a lot of love, grace, and blessings!”.

TABLE OF CONTENTS

List of Tables	xiv
List of Figures	xvii
Summary	xxii
Chapter 1: Introduction	1
1.1 Thesis Statement	4
1.2 Contributions	4
1.3 Organization	8
Chapter 2: Background	10
2.1 Parallel Architectures	10
2.1.1 General-Purpose Parallel Architectures	10
Multi-Core Architectures	10
Vector Processing (SIMD) Architectures	12
2.1.2 Domain-Specific Parallel Architectures	13
Spatial DNN Accelerators	13
Specialized Vector Processing Units	14
Thread Migratory Architectures	15

2.2	High-Performance Applications	17
2.2.1	Scientific Computing Applications	17
2.2.2	Deep Learning	17
2.2.3	Graph Analytics	19
2.3	Summary	19
Chapter 3: PoPP: Polyhedral Optimizations of Explicitly-Parallel Programs . .		20
3.1	Abstract	20
3.2	Introduction	21
3.3	Background	22
3.3.1	Polyhedral Model	22
3.3.2	Explicit Parallelism	23
	Loop-level parallelism	24
	Task-level parallelism including dependences	24
3.4	Motivating Examples	24
3.4.1	2-D Jacobi	25
3.4.2	Particle Filter	26
3.5	Polyhedral optimizer for Parallel Programs (PoPP)	26
3.5.1	Conservative analysis	28
3.5.2	Extraction of happens-before relations	30
	Loop-level parallelism	30
	Task parallelism including dependences	31
3.5.3	Reflection of happens-before relations	33

3.5.4	PolyAST: a loop optimizer integrating polyhedral and AST-based transformations	34
3.6	Experimental Evaluation	35
3.6.1	Experimental setup	35
3.6.2	KASTORS Suite	39
3.6.3	Rodinia Suite	40
3.7	Limitations	41
3.8	Related Work	42
3.9	Summary	43
Chapter 4: PolySIMD: A Unified Approach to Variable Renaming for Enhanced Vectorization		45
4.1	Abstract	45
4.2	Introduction	46
4.3	Discussion on Variable Renaming Transformations	48
4.3.1	Source Variable Renaming (SoVR)	49
4.3.2	Sink Variable Renaming (SiVR)	50
4.3.3	Synergy between SoVR and SiVR	51
4.4	Motivating Example	51
4.5	Our Unified Approach to Variable Renaming	53
4.5.1	Dependence Cycles Finder	53
4.5.2	Bipartite Graph Constructor	55
4.5.3	Solver	55
4.5.4	Transformer	57

4.5.5	Bounding Additional Space	57
4.6	Performance Evaluation	58
4.6.1	Experimental Platforms	58
4.6.2	Benchmarks	59
4.6.3	Comparison with ICC	60
4.6.4	Comparison with Calland et al’s approach	62
4.6.5	Comparison with Chu et al’s approach	62
4.7	Related Work	63
4.8	Summary	64
Chapter 5: Marvel: A Data-centric Compiler for DNN Operators on Spatial Accelerators		66
5.1	Abstract	66
5.2	Introduction	67
5.3	Background	69
5.3.1	Spatial DNN Accelerators	70
5.3.2	MDC Notation	71
5.4	Conformable DNN Operators	73
5.5	Transformation	76
5.5.1	Data Mapping directives	77
5.6	Mapping Space Exploration	80
5.6.1	Solving off-chip mapping subspace	81
5.6.2	Solving on-chip mapping subspace	83
5.7	Evaluation	84

5.7.1	Evaluation on CONV2D	85
5.7.2	Evaluation on GEMM	89
5.7.3	Evaluation on MLP and LSTM	90
5.8	Related Work	91
5.9	Summary	93
Chapter 6: Vyasa: A High-Performance Vectorizing Compiler for Tensor Con-		
volutions on the Xilinx AI Engine		94
6.1	Abstract	94
6.2	Introduction	94
6.3	Background	97
6.3.1	Tensor Convolutions	97
6.3.2	Xilinx AI Engine	98
6.4	Our Approach	100
6.4.1	Translating into Triplet Representation	101
6.4.2	Lazy Stores Optimization	103
6.4.3	Exploiting Vector Register Reuse & Realizing Unaligned Loads and Scalar Broadcast	104
6.4.4	2D Vector SIMD Datapath	106
6.4.5	Code Generation	108
6.4.6	Auto-tuner	108
6.5	Experiments	109
6.5.1	CONV2D in Computer Vision	110
6.5.2	CONV2D in Deep Learning	113

6.5.3	CONV3D	115
6.6	Related Work	116
6.7	Summary	117
Chapter 7: Compiler Optimizations for Graph Analytics on a Thread Migratory Architecture (EMU)		118
7.1	Abstract	118
7.2	Introduction	118
7.3	Compiler Transformations	120
7.3.1	Node/Loop Fusion	120
7.3.2	Edge Flipping	121
7.3.3	Use of Remote Updates	121
7.4	Experiments	122
7.4.1	Experimental Setup	122
7.4.2	Conductance algorithm	124
7.4.3	Single Source Shortest Path using Bellman-Ford's Algorithm (SSSP-BF)	125
7.4.4	Triangle Counting Algorithm	128
7.5	Related Work	129
7.6	Summary	130
Chapter 8: Conclusions and Future Directions		132
References		157

LIST OF TABLES

3.1	Details of architectures used for experiments.	35
3.2	Sequential execution times of KASTORS and Rodinia on Intel Westmere and IBM Power 8 systems along with problem sizes. Intel ICC-14.0 compiler doesn't support OpenMP 4.0 task depend constructs. So, no execution time is reported for KASTORS on Intel platform with ICC compiler. Transformations exposed by PoPP: Permutation (P), Fusion (F), Skewing (S), Tiling (T), Doacross pipelined parallelism (D), No further optimizations (-). Manual modifications performed before passing to PoPP: Replace complex if-statements by closures i.e., outlined functions (OF), Delinearization on task-depend variables (D), Function inlining (F), Annotated inner loop as parallel (AP), Annotated inner loop as parallel with array reductions (APR), Annotated with task-depend constructs (AT), Removal of printf statements (R), No modifications (-).	36
4.1	A comparison between SoVR and SiVR transformations related to the space requirements and additional stores, loads introduced by these transformations in one iteration of the target loop. * – Additional scalar loads/stores for SiVR transformation may go negative in case of renaming scalars. . . .	51
4.2	Bipartite graph constructed on the dependence graph of the original program in Figure 4.2.	54
4.3	Summary of SIMD architectures and compiler flags used in our experiments. SP refers to Single Precision floating point operands, VPU refers to a KNL Vector Processing Unit, and SM refers to a GPU Streaming Multi-processor.	59

4.4	Summary of the 11 benchmarks from the TSVC suite used in our evaluation, including the number of statements, number of dependences, and number of elementary cycles per benchmark (excluding self-loop cycles). The benchmarks were executed using $N = 2^{25}$ and $T = 200$ as input parameters. Number of SiVR and SoVR transformations performed by <i>PolySIMD</i> for the 11 benchmarks, and also the overall compilation times required. Coincidentally, none of these benchmarks triggered a case in which both SiVR and SoVR transformations had to be performed.	60
4.5	Speedups on the Intel KNL processor and NVIDIA Volta accelerator using <i>PolySIMD</i> on seven benchmarks from the eleven benchmarks relative to past approaches, i.e., Calland et al. [29] and Chu et al. [30]. We excluded the remaining four benchmarks from the table since our results were similar to both of the past works.	62
5.1	Conformability of the popular DNN operators onto the MDC notation (Y/N refers to YES/NO).	76
5.2	DNN primitive operators, occurrences, and MDC conformability in the MLPerf [147] DNN models, VGG16, and AlexNet models.	77
5.3	Accelerator setups in our evaluation.	84
5.4	The statistics (min/avg/max) of the CONV2D mapping space in our evaluation and the resultant mapping subspaces after decoupling and pruning strategies.	87
5.5	Two layers from VGG16 and MobileNetV2 for brief discussion on our approach generated mappings; Level-3 tile sizes and degree of parallelism are part of the mappings identified by our approach on Platform P2.	88
5.6	Description of the GEMM workloads taken from the recent work in [154].	89
5.7	Description of the MLP and LSTM workloads taken from the Interstellar work in [136].	90
5.8	Comparison of our MDC notation with prior compilers in terms of expressiveness, mapping notation, and the presence of accurate cost models.	92
6.1	Triplet representation of the loop body in Figure 6.3(c)	103
6.2	Triplet representation after the lazy stores optimization	104

6.3	Triplet representation after addressing unaligned loads, scalar broadcast, and exploiting vector register reuse.	106
6.4	Triplet representation after fusing the logical 1D vector multiplications and finding the data selection parameters	108
6.5	The AI Engine configuration used in our evaluation.	110
6.6	CONV2D workloads of Computer Vision used in our evaluation and optimal schedules from auto-tuner	112
6.7	CONV2D workloads of deep learning used in our evaluation (variable names described in Section 6.3) and optimal schedules.	113
7.1	Specifications of a single node of the Emu system.	122
7.2	Experimental evaluation of three graph algorithms (Conductance, SSSP-BF and Triangle counting) on the RMAT graphs from scales 6 to 14 specified by Graph500. Transformations applied on the algorithms: Conductance/SSSP-BF/Triangle counting: (Node fusion)/(Edge flipping and Remote updates)/(Remote updates). The evaluation is done a single node of the Emu system described in Table 7.1. Note that we had intermittent termination issues while running SSSP-BF from scale 13-14 on the Emu node, and hence we omitted its results.	123

LIST OF FIGURES

1.1	40 years of process performance (taken from [6, 7]).	1
2.1	An abstract overview of a general-purpose multi-core processor having multiple CPUs with a memory hierarchy (figure source: [38]).	11
2.2	An abstract overview of a general-purpose SIMD architecture (figure source: [39]).	12
2.3	An abstract overview of a spatial DNN accelerator model which is pervasive in many state-of-the-art accelerators [40, 41, 42, 43, 44].	13
2.4	An abstract overview of a specialized vector processing unit in the Xilinx AI Engine.	15
2.5	An abstract overview of a thread migratory architecture in the EMU (figure source: [53]).	16
3.1	2-D Jacobi kernel from KASTORS suite.	25
3.2	Particle filter kernel from Rodinia suite	27
3.3	Overview of our approach	28
3.4	Happens-before relations for the Jacobi program in Figure 3.5(a) due to task-spawn, task-wait, and sequential ordering	31
3.5	Overall explanation of our framework on Jacobi benchmark from KASTORS suite.	32
3.6	Evaluation of the KASTORS suite (using GCC compiler). Sequential times are reported in Table 3.2. Original benchmark speedup is compared against with optimized codes from PoPP with/ without considering happens-before (HB) relations.	37

3.7	Evaluation of the Rodinia suite (using Intel compiler) on Intel Westmere with 12 cores. Sequential times are reported in Table 3.2. Original benchmark speedup is compared against with optimized codes from PoPP with/without considering happens-before (HB) relations.	37
3.8	Evaluation of Rodinia suite (using GCC compiler) on Intel Westmere with 12 cores. Sequential times are reported in Table 3.2. Original benchmark speedup is compared against with optimized codes from PoPP with/ without considering happens-before (HB) relations.	38
3.9	Evaluation of the Rodinia suite (using GCC compiler) on IBM Power8 with 24 cores. Sequential times are reported in Table 3.2. Original benchmark speedup is compared against with optimized codes from PoPP with/ without considering happens-before (HB) relations.	38
4.1	An example to illustrate SoVR and SiVR transformations.	48
4.2	A running example from [29] whose dependence graph consists of three cycles $c1/c2/c3$: $s1-s3-s2-s4-s1/s1-s3-s4-s1/s1-s2-s4-s1$ which prohibit vectorization. The table also lists dependence graphs and transformed codes after applying past approach [29] and our integrated approach on the original program.	52
4.3	Workflow of <i>PolySIMD</i> implemented as an extension to the PPCG [114]. . .	53
4.4	Speedups using <i>PolySIMD</i> on the eleven benchmarks from the TSVC suite, compiled using the Intel’s ICC v17.0 product compiler and running on a single core of Intel Knights Landing processor.	59
5.1	Overview of the design-time flow for computer architects developing new accelerators, and the compilation flow for ML programmers leveraging the accelerators. Scope of this work is the mapping explorer and the loop optimizer in the above diagram.	67
5.2	Abstract spatial accelerator model which is pervasive in many state-of-the-art accelerators [40, 46, 144, 43].	70
5.3	A broader overview of a spatial accelerator system having multiple accelerators in the form of clusters.	70
5.4	A mapping of the CONV1D in the MDC notation along with the visualization of its data mappings.	72

5.5	The dimension dependence graph (DDG) of simple operators such as CONV1D and stencil satisfying the rule R3, and an example violating the rule R3. $d_o/d_I/d_w$: tensor dimension variables corresponding to the output, input, and weight tensors.	74
5.6	A brief overview of the mapping expressed in the loop-nest form of CONV1D, and its translation into the MDC notation with data mapping directives. . . .	79
5.7	An overview of our approach along with pruning strategies for searching mapping space of convolutions. The pruning strategies in green color preserve optimal mappings, whereas the strategies in red color may prune optimal.	81
5.8	Performance comparison of Marvel generated mappings with the mappings of dMazeRunner-like optimizer [135] and Interstellar-like optimizer [136] relative to the roof-line peaks of the AlexNet and VGG-16 models on both the platforms (P1 and P2).	85
5.9	Runtime and energy comparison of Marvel generated mappings with the popular mapping styles such as row-stationary (RS) from Eyeriss [40], weight-stationary from DLA [46], output-stationary from ShiDianNao [145] for the AlexNet [64], VGG-16 [65], ResNet-50 [137], MobileNet-V2 [138] models on both the platforms (P1 and P2).	86
5.10	Performance comparison of Marvel generated mappings with the mappings of dMazeRunner-like optimizer [135], and Interstellar-like optimizer [136] relative to the roof-line peaks of the GEMM workloads in Table 5.6 and LSTM, MLP in Table 5.7 on both the platforms (P1 and P2).	91
5.11	Comparison of Marvel with prior compiler approaches for spatial accelerators (mRNA [139], Zhang et al. [132], Ma et al. [131], Auto-TVM [76], dMazeRunner [135], Interstellar [136], TimeLoop [133]) for the mapping space exploration of DNN operators. Our approach (Marvel) supports any operator conformable with the MDC notation.	93
6.1	A pictorial overview of the key architectural features of the Xilinx AI Engine, i.e., 2D vector SIMD datapath and shuffle network.	99
6.2	Workflow of our approach (Vyasa) which is implemented as an extension to the Halide framework [70].	101
6.3	Algorithmic description of the convolution of a 4x3 filter over an input 2D image in the Halide language [70]. $A(a:b,c)$ is a short hand vector notation for denoting a contiguous slice from $A(a,c)$ to $A(b,c)$ in one direction. . . .	102

6.4	A pictorial overview of the convolution of 4x3 filter based on the schedule described in Figure 6.3(b) at the loop iterations $x = 0$ and $y = 0$	102
6.5	Reuse graph corresponding to the vector loads of the tensor I in Table 6.2, and its connected components to construct larger vector loads.	105
6.6	An overview of the two fused vector operations (a and b) over the vector registers V1, V2 for input and weights, respectively of the running example shown in Table 6.2 at $x=0$ and $y=0$. The shuffle network of the AI Engine helps each multiplier of the 16 lanes and 2 columns of the 2D SIMD datapath to choose required elements from the vector registers.	107
6.7	A snippet of the generated 16-bit vector code for the running example in Figure 6.3. VLOAD/VMUL/VMAC/VSTORE refers to vector load, vector multiplication, vector multiply-and-accumulate, and vector store. SELECT symbolically represents the data selection over a vector register for the i^{th} row and j^{th} column of 2D datapath multipliers.	109
6.8	Comparison of our approach with auto-tuner against the available expert-written codes for CONV2D operation with 3x3 and 5x5 filters.	110
6.9	Roof-line graphs of four workloads considered in Figure 6.8, where each data point is a schedule explored by the auto-tuner.	111
6.10	Performance of our approach generated codes for CONV2D workloads of Computer Vision over filter sizes from 2 to 11.	112
6.11	Performance of our approach generated codes for CONV2D workloads (shown in Table 6.7) of Deep Learning.	114
6.12	Data-layouts of input and weight tensors of the 16-bit REG-3x3 workload (Table 6.7), to enable the fusion of 1D logical vector multiplications along the channels, thereby avoiding the padding required for weights.	114
6.13	Performance of our approach generated codes for CONV3D workloads with weight sizes as 3x3x3, 5x5x5, 7x7x7.	116
7.1	Overview of a single Emu node (figure source: [53]), where a dotted circle represents a nodelet. Note that, the co-location of the narrow channel memory unit (NCDRAM) with gossamer cores makes the overall Emu system a near memory system.	120
7.2	Speedup over the original conductance algorithm on a single Emu node (8 nodelets) and % reductions in thread migrations after applying loop fusion. .	125

7.3 % reductions in thread migrations of SSSP-BF algorithm after applying edge flipping with regular atomic updates and with remote atomic updates on a single node (8 nodelets) of Emu Chick. 127

7.4 Speedup of SSSP-BF algorithm on a single Emu node (8 nodelets) after applying edge flipping with regular atomic updates and with remote updates. 127

7.5 Speedup over the original triangle counting implementation on a single Emu node (8 nodelets) and % reductions in thread migrations after using remote atomic updates. 129

SUMMARY

Computer hardware is undergoing a major disruption as we approach the end of Moore’s law, in the form of new advancements to general-purpose and domain-specific parallel architectures. Contemporaneously, the demand for higher performance is broadening across multiple application domains ranging from scientific computing applications to deep learning and graph analytics. These trends raise a plethora of challenges to the de-facto approach to achieving higher performance, namely application development using high-performance libraries. Some of the challenges include porting/adapting to multiple parallel architectures, supporting rapidly advancing domains, and also inhibiting optimizations across library calls. Hence, there is a renewed focus on advancing optimizing compilers from industry and academia to address the above trends, but doing so requires enabling compilers to work effectively on a wide range of applications and also to exploit current and future parallel architectures better. As summarized below, this thesis focuses on compiler advancements for current and future hardware trends.

First, we observe that software with explicit parallelism for general-purpose multi-core CPUs and GPUs is on the rise, but the foundation of current compiler frameworks is based on optimizing sequential code. Our approach uses explicit parallelism specified by the programmer as logical parallelism to refine the conservative dependence analysis inherent in compilers (arising from the presence of program constructs such as pointer aliasing, unknown function calls, non-affine subscript expressions, recursion, and unstructured control flow). This approach makes it possible to combine user-specified parallelism and compiler-generated parallelism in a new unified polyhedral compilation framework (PoPP).

Second, despite the fact that compiler technologies for automatic vectorization for general-purpose vector processing (SIMD) units have been under development for over four decades, there are still considerable gaps in the capabilities of modern compilers to perform automatic vectorization. One such gap can be found in the handling of loops with dependence cycles that involve memory-based anti (write-after-read) and output (write-after-write) dependences. A significant limitation in past work is the lack of a unified formulation that synergistically integrates multiple storage transformations to break the cycles and further unify the formulation with loop transformations to enable vectorization. To address this limitation, we propose the PolySIMD approach.

Third, the efficiency of domain-specific spatial accelerators for Deep Learning (DL) solutions depends heavily on the compiler’s ability to generate optimized mappings or code for various DL operators (building blocks of DL models, e.g., CONV2D, GEMM) on the accelerator’s compute and memory resources. However, the rapid emergence of new op-

erators and new accelerators pose two key challenges/requirements to the existing compilers: 1) Ability to perform fine-grained reasoning of various algorithmic aspects of the new operators and also complex hardware structures of the new accelerators to achieve peak performance, and 2) Ability to quickly explore the enormous space of possible mappings involving various partitioning schemes, loop transformations, and data-layout choices, yet achieving high-performance and energy efficiency. To address these challenges, we introduced a data-centric compiler “Marvel” for optimizing DL operators onto flexible spatial accelerators. We also introduced a high-performance vectorizing compiler “Vyasa” for optimizing tensors convolutions on specialized SIMD units of Xilinx AI Engine.

Finally, with the emergence of a domain-specific thread migratory architecture (EMU) to address the locality wall, we developed thread-migration aware compiler optimizations to enhance the performance of graph analytics on the EMU machine. Our preliminary evaluation of compiler optimizations such as node fusion and edge flipping demonstrate a significant benefit relative to the original programs.

CHAPTER 1

INTRODUCTION

Traditionally, computational science and engineering (CSE) applications such as large-scale climate prediction, computational fluid dynamics, high-dimensional tensor contractions, have dominated the need for performance to advance research in science and engineering. However, the demand for high performance is broadening across multiple application domains with the advent of big data and machine learning in the current era. For example, large-scale graph processing has become an essential application domain for high performance because of the prevalence of large graph data size from social networks [1, 2]. In the last few years, Deep Learning (DL) became a promising solution to many of the learning tasks such as real-time speech recognition, computer vision, health intelligence, self-driving cars [3, 4, 5]. These DL solutions also require higher performance because of their longer training time and also tight latency constraints in the inference.

Parallel computer architectures refer to a class of computer processors with multiple compute units connected via interconnection networks and a memory hierarchy. Some of the popular parallel computer architectures are multi-core CPUs, SIMD units, and GPUs. Figure 1.1 presents a quick forty-year overview of the processor performance, and we briefly describe the parallel architecture evolution from 2004.

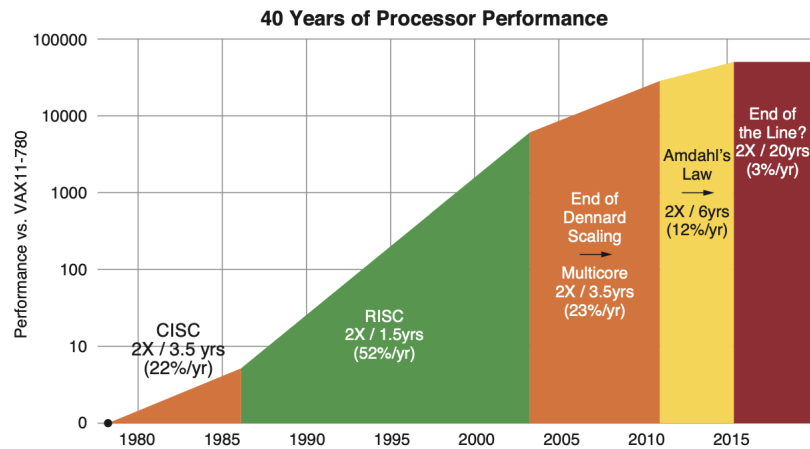


Figure 1.1: 40 years of process performance (taken from [6, 7]).

Parallel architectures evolution from 2004: The inability to increase single CPU clock frequency within a power budget because of the breakdown of Dennard’s scaling (power wall [8]) has resulted in the emergence of *multi-core processors*, and enabled Moore’s law

(doubling transistor count per every two years) [9] to extend further. Each of these cores employs either simultaneous multi-threading (SMT) [10] or fine-grained multi-threading [11] to provide thread-level parallelism. Also, *vector processing units*, i.e., Single Instruction Multiple Data (SIMD) units, are integrated with the individual CPU cores to provide data-parallelism.

The multi-core solution was a partial response to address the end of Dennard's scaling, and the solution quickly resulted in the utilization wall¹, i.e., the percentage of a multi-core chip that can actively switch drops exponentially due to power constraints. As a result, multiple approaches have emerged to address the utilization wall. Some of the notable approaches are: 1) *Light-weight and simpler general-purpose cores* (e.g., Intel KNL, GPG-PUs), 2) *Domain-specific accelerators* tailored to optimizing a set of specific computations (e.g., DNN accelerators, specialized SIMD units), and 3) *Heterogeneous processors* incorporating both light-weight general-purpose cores and domain-specific accelerators (e.g., Xilinx Versal [7]).

Unlike processor architectures, recently, applications have started posing walls to achieve higher performance on the existing parallel architectures, for example, locality wall [14] with the rise of memory-intensive applications (e.g., large scale graph analytics), where computation is dominated by data access, movement, and little reuse during the computation. To address the locality wall, multiple approaches have started emerging, and some of the noted approaches are: 1) *Migratory thread-based architecture* (e.g., EMU [15]) with near-memories, 2) *3D stacked near-memory systems* (e.g., HMC Micron [16]), and 3) *Custom accelerators for domain-specific memory-intensive applications* (e.g., ExTensor [17] for sparse linear algebra).

General-Purpose and Domain-Specific Parallel Architectures: Since we are reaching the end of Moore's law, significant disruption is underway in computer hardware as processors strive to extend, and go beyond, the end-game of Moore's Law. The Industry and Academia are pushing the boundaries of maximum peak performance and energy efficiency of parallel hardware by 1) making the general-purpose processors lightweight and increasing its count on a chip, and 2) building custom domain-specific accelerators to accelerate specific sets of computations.

Majorly, there are three approaches to achieving the higher performance of applications on parallel architectures – 1) Ninja (expert) programmers, 2) High-performance libraries, and 3) Optimizing compilers. Ninja programmers have an in-depth knowledge of architec-

¹The utilization wall is also referred to as dark silicon [12, 13] in the literature.

ture, including various hardware intricacies, and optimize a given application very well on that particular architecture. However, this approach requires a significant investment effort to port it to other architectures, and this becomes even more daunting with the proliferation of more architectures as we approach towards the end of Moore's law.

Library-based approach, the de-facto approach, lets a programmer/developer compose an application using high-performance library primitives, which are generally provided by the hardware vendors (e.g., Intel MKL) and tuned for their particular architectures. However, this approach can yield sub-optimal performance of the entire application, because these library calls are black boxes to programmers/compiler to perform optimizations across library calls. Also, with applications rapidly evolving (e.g., DNN models and their operators) and their input sizes varying drastically (e.g., DNN layer sizes), it is tough for the hardware vendors to cope with such a rapid change and provide a library that can provide high-performance across all the inputs (e.g., Intel MKL DGEMM kernel can yield lower performance on skewed matrix shapes).

Optimizing compilers is another approach to achieve performance across a variety of parallel architectures and includes optimizations to maximize parallelism and memory locality using loop dependence analysis and transformations [18, 19, 20, 21, 22, 23, 24]. Using this approach, programmers/application developers can become less concerned about intricacies of the performance optimizations and underlying architecture details, unlike Ninja programmers. Since the optimizing compilers generally have back-ends for multiple target architectures (e.g., LLVM), the portability issues to other architectures can be alleviated using this approach. Also, optimizing compilers can customize their optimization techniques based on application input sizes, and also help in optimizing the rapidly evolving applications/kernels (e.g., DNN models and their operators), unlike the library-based approaches [25]. Even though there are numerous benefits of using the optimizing compilers relative to Ninja programmers and library-based approaches, optimizing compilers still require advancements in program analysis, code transformations, and code generation to exploit better the advancements in both the general-purpose and domain-specific parallel architectures as part of the hardware disruption.

1.1 Thesis Statement

”Given the increasing demand for performance across multiple application domains and the major disruptions in future computer hardware as we approach the end of Moore’s Law, our thesis is that advances in compiler optimizations, via newer analyses, transformations, mapping space exploration strategies, and code generation techniques, are critical for enabling a wide range of applications to exploit future advances in both general-purpose and domain-specific parallel architectures.”

1.2 Contributions

In this section, we outline our contributions in advancing compiler optimizations via newer analyses, transformations, mapping space exploration strategies, and code generation techniques to achieve higher performance on general-purpose and domain-specific parallel architectures.

1) Polyhedral optimizations of explicitly-parallel programs for general-purpose multi-core processors: Most of the compiler frameworks treat explicit-parallelism of an input program either as unknown library calls or ignore them and perform dependence analysis to enable optimizations. Additionally, the compilers perform conservative dependence analysis in the presence of unanalyzable constructs such as pointer aliasing, unknown function calls, non-affine expressions, recursion, and unstructured control flow. These unanalyzable constructs can limit the applicability of transformations even though they are legal to apply. Our work is motivated by the observation that software with explicit parallelism is on the rise. Our approach uses the explicit parallelism specified by the programmer as a logical parallelism. Then, our approach refines the conservative dependences with partial execution order from the explicit parallelism to enable a broader set of loop transformations, compared to what might have been possible if the input program is sequential.

A summary of our approach (PoPP – Polyhedral optimizer for Parallel Programs) [26, 27] is as follows. We first enable conservative dependence analysis of a given region of code by introducing dummy variables that can work with any polyhedral tool that supports *access functions*. After obtaining conservative dependences, the Fourier-Motzkin elimination method is used to remove all dummy variables. Next, we identify happens-before relations from the explicitly parallel constructs, notably parallel loops and tasks, and intersect with conservative dependences. The resulting set of dependences is then passed on to a polyhedral optimization tool, such as PolyAST, to enable the transformation of explicitly-parallel programs with unanalyzable data accesses.

We evaluate our approach using twelve OpenMP benchmark programs from the KAS-TORS and Rodinia benchmark suites. We show that 1) these benchmarks contain unanalyzable data accesses that prevent polyhedral frameworks from performing exact dependence analysis, 2) explicit parallelism can help mitigate the imprecision, and 3) polyhedral transformations with the resulting dependences can further improve the performance of manually-parallelized OpenMP programs. Our experimental results show performance improvements for these OpenMP programs on a 12-core Intel Westmere platform and a 24-core IBM Power8 platform.

2) Unification of multiple storage transformations with loop optimizations for enhanced vectorization on general-purpose vector processors (SIMD/GPUs): Despite the fact that compiler technologies for automatic vectorization have been under development for over four decades, there are still considerable gaps in the capabilities of modern compilers to perform automatic vectorization for SIMD units. One such gap can be found in the handling of loops with dependence cycles that involve memory-based anti (write-after-read) and output (write-after-write) dependences. Past approaches, such as variable renaming and variable expansion, break such dependence cycles by either eliminating or repositioning the problematic memory-based dependences. However, the past work suffers from three key limitations: 1) Lack of a unified framework that synergistically integrates multiple storage transformations, 2) Lack of support for bounding the additional space required to break memory-based dependences, and 3) Lack of support for integrating these storage transformations with other code transformations (e.g., statement reordering) to enable vectorization.

We address the three limitations above by integrating both Source Variable Renaming (SoVR) and Sink Variable Renaming (SiVR) transformations into a unified formulation, and by formalizing the “cycle-breaking” problem as a minimum weighted set cover optimization problem. To the best of our knowledge, our work (PolySIMD) [28] is the first to formalize an optimal solution (reflecting best execution time) for cycle breaking that simultaneously considers both SoVR and SiVR transformations, thereby enhancing vectorization and reducing storage expansion relative to performing the transformations independently.

We implemented our approach in PPCG, a state-of-the-art optimization framework for loop transformations, and evaluated it on eleven kernels from the TSVC benchmark suite. Our experimental results show a geometric mean performance improvement of $4.61\times$ on an Intel Xeon Phi (KNL) machine relative to the optimized performance obtained by Intel’s ICC v17.0 product compiler. Further, our results demonstrate a geometric mean performance improvement of $1.08\times$ and $1.14\times$ on the Intel Xeon Phi (KNL) and Nvidia Tesla

V100 (Volta) platforms relative to past work that only performs the SiVR transformation [29], and of $1.57\times$ and $1.22\times$ on both platforms relative to past work on using both SiVR and SoVR transformations [30].

3) Data-centric compiler for deep learning operators onto domain-specific DNN spatial accelerators: The efficiency of a spatial DNN accelerator depends heavily on the compiler and its cost model ability to generate optimized mappings for various operators of DNN models on the accelerator’s compute and memory resources. A significant difference between the compilers for spatial accelerators and CPUs/GPUs is the need for “accurate” cost models for finding optimal mappings reflecting best latency and energy efficiency. This is because spatial accelerator’s performance is sensitive to the mapping parameters, for, e.g., a small change in the tile size or degree of parallelism would drastically change the latency or energy efficiency numbers. However, existing cost models lack a formal boundary over the operators for precise and tractable analysis, which poses adaptability challenges for new DNN operators. To address this challenge, we leverage the recently introduced Maestro Data-Centric (MDC) notation. We develop a formal understanding of DNN operators whose mappings can be described in the MDC notation because any mapping adhering to the notation is always analyzable by the MDC’s cost model. Furthermore, we introduce a transformation for translating mappings into the MDC notation for exploring the mapping space.

Searching for the optimal mappings reflecting best latency and energy efficiency is challenging because of the large space of mappings, and this challenge gets exacerbated with new operators and diverse accelerator configurations. To address this challenge, we propose a decoupled off-chip/on-chip approach that decomposes the mapping space into off-chip and on-chip subspaces, and first optimizes the off-chip subspace followed by the on-chip subspace. The motivation for this decomposition is to dramatically reduce the size of the search space and prioritize the optimization of off-chip data movement, which is 2-3 orders of magnitude more than the on-chip data movement. We implemented our approach in a tool called *Marvel*, and another significant benefit of our approach is that it applies to any DNN operator conformable with the MDC notation. In addition, our approach works by leveraging two state-of-the-art cost models to explore the two subspaces – a classical distinct-block (DB) locality cost model for the off-chip subspace, and a state-of-the-art DNN accelerator behavioral cost model, MAESTRO, for the on-chip subspace.

Overall, our approach reduced the mapping space by an $O(10^{10})$ factor for the four major CNN models (AlexNet, VGG16, ResNet50, MobileNetV2), while generating mappings that demonstrate a geometric mean performance improvement of $10.25\times$ higher throughput

and $2.01\times$ lower energy consumption compared with three state-of-the-art mapping styles from past work. We also evaluated our approach over the GEMM, LSTM, and MLP workloads and compared them with the optimizers from past work.

4) High-performance vectorizing compiler for tensor convolutions on the Xilinx AI Engine (domain-specific 2D SIMD processor): There is a strong resurgence of interest in improving vector processing (SIMD) units due to the significant energy efficiency benefits of using SIMD parallelism. There is an emphasis on specializing SIMD units to improve further energy efficiency benefits for specific domains such as Machine learning, Computer Vision, and 5G Wireless. An important specialization, which is referred to as "2D vector SIMD datapath" [31, 32, 33], is the ability of each vector lane to execute more than one scalar operation and to chain the results from one operation to another. Another specialization includes the removal of expensive data permutation units (e.g., shuffle units) [34, 35] and instead introduce sophisticated, programmable interconnection networks (a.k.a shuffle networks) between the SIMD datapath and vector register file to support the required data permutation patterns [36, 33]. Xilinx's AI Engine is a recent industry example of energy-efficient vector processing that includes novel support for 2D SIMD datapaths and shuffle interconnection network. The current approach to programming the AI Engine relies on a C/C++ API for vector intrinsics. While an advance over assembly-level programming, it requires the programmer to specify a number of low-level operations based on detailed knowledge of the hardware.

To address these challenges, we introduce *Vyasa*, a new programming system that extends the Halide DSL compiler to generate code for the AI Engine automatically. We evaluated *Vyasa* on 36 CONV2D and 6 CONV3D workloads and achieved geometric means of 7.6 and 23.3 MACs/cycle for 32-bit and 16-bit operands (which represent 95.9% and 72.8% of the peak performance respectively). For 4 of these workloads for which expert-written codes were available to us, *Vyasa* demonstrated a geometric mean performance improvement of $1.10\times$ with $50\times$ smaller code relative to the expert-written codes. Further, our compiler-generated code achieved a geometric mean performance improvement of $1.134\times$ relative to expert-written codes available for two workloads.

5) Thread-migration aware compiler optimizations for graph analytics on thread-migratory domain-specific hardware (EMU): Unlike dense linear algebra applications, graph applications typically suffer from poor performance because of 1) inefficient utilization of memory systems through random memory accesses to graph data, and 2) overhead of executing atomic operations. Hence, there is a rapid growth in improving software and

hardware platforms to address the above challenges. One such improvement in the hardware is a realization of the Emu system, a thread migratory and near-memory processor introduced to address applications domains having weak-locality. In the Emu system, a thread responsible for computation on a datum is automatically migrated over to a node where the data resides without any intervention from the programmer. The idea of thread migrations is very well suited to graph applications as memory accesses of the applications are irregular. However, thread migrations can hurt graph applications' performance if overhead from the migrations dominates the benefits achieved through migrations.

In this preliminary study [37], we explore two high-level compiler optimizations, i.e., loop fusion and edge flipping, and one low-level compiler transformation leveraging hardware support for remote atomic updates to address overheads arising from thread migration, creation, synchronization, and atomic operations. We performed a preliminary evaluation of these compiler transformations by manually applying them on three graph applications over a set of RMat graphs from Graph500 – Conductance, Bellman-Ford's algorithm for the single-source shortest path problem, and Triangle Counting. Our evaluation targeted a single node of the Emu hardware prototype and has shown an overall geometric mean reduction of 22.08% in thread migrations.

1.3 Organization

The rest of the dissertation is organized as follows.

- Chapter 2 briefly describes high-performance applications and also both general-purpose and domain-specific parallel architectures considered in our thesis.
- Chapter 3 describes our work (PoPP) on using explicit parallelism to refine conservative dependence analysis and to enable a broader set of transformations for enhanced parallelization on general-purpose multi-core CPUs.
- Chapter 4 describes our work (PolySIMD) on synergistically integrating multiple storage transformations to break cycles in dependence graphs, and further unification with loop transformations to enable vectorization on general-purpose vector (SIMD/GPUs) processors.
- Chapter 5 describes our work (Marvel) introducing data-centric compiler for mapping computationally expensive deep learning primitives onto the domain-specific DNN spatial accelerators.

- Chapter 6 describes our work (Vyasa) introducing a domain-specific compiler to automate the generation of high-performance vector code for tensor convolutions, while exploiting the unique capabilities of the Xilinx AI Engine (domain-specific 2D SIMD processor) without requiring manual effort in development and tuning.
- Chapter 7 describes our preliminary evaluation of thread-migration aware compiler optimizations for graph algorithms on the thread-migratory (EMU) system introduced for accelerating weak-locality application domains.
- Finally, Chapter 8 present our conclusions and directions for future research.

CHAPTER 2

BACKGROUND

In this chapter, we provide a brief overview of the high-performance applications and also both general- purpose and domain-specific parallel architectures in the dissertation.

2.1 Parallel Architectures

Parallel computer architectures refer to a class of computer processors with multiple compute units connected via interconnection networks and a memory hierarchy. Some of the popular parallel computer architectures are multi-core CPUs and GPUs. As we are reaching the end of Moore's law, computer architects are pushing the boundaries of maximum peak performance and energy efficiency of parallel hardware by 1) making the general-purpose processors light-weight and increasing the count of the processors on a chip, and 2) building custom domain-specific accelerators to accelerate specific sets of computations. In this section, we provide a brief overview of both general-purpose and domain-specific parallel architectures that are considered in this dissertation.

2.1.1 General-Purpose Parallel Architectures

General-purpose parallel architectures are programmable devices having multiple compute units that can be used in a variety of applications, not limited to a specific application/domain. These architectures/processors have generic datapaths with large register files and general functional units. Also, these processors are designed to be reasonably good at performance nearly for any application. These processors often involve multiple levels of the memory hierarchy (including caches) to make memory access faster.

Multi-Core Architectures

With the challenges from the lack of more abundant instruction-level parallelism (ILP) in the applications and the breakdown of Dennard's scaling (power wall), a significant trend in the development of processors was more than one processing core on a single integrated circuit. These processors are called as multi-core processors or chip multi-processors. Each of these cores can employ either simultaneous multi-threading (SMT) to effectively use the core resources from instructions of multiple threads, or fine-grained/temporal multi-threading where only one thread of instructions can execute at a time and

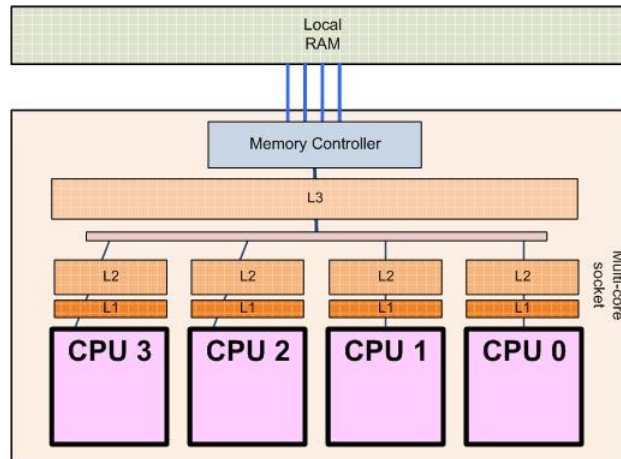


Figure 2.1: An abstract overview of a general-purpose multi-core processor having multiple CPUs with a memory hierarchy (figure source: [38]).

share the resources in a round-robin fashion. The multi-core processors typically have multiple memory hierarchy levels to address memory wall problem and make memory accesses faster. For example, each of the cores in a processor typically has a private L1 cache for both instructions and data, a distributed L2 cache shared by multiple cores, and a single larger L3 cache shared by all cores. The processor may also implement different cache coherence protocols such as bus-based or directory-based protocols to maintain coherency of data present at multiple locations in the cache hierarchy. Furthermore, multiple of these cores or multi-core processors can be grouped into a socket and distribute its main memory (DRAM) across the sockets, making the processors non-uniform (NUMA machines), e.g., IBM Power 9 machine.

The multi-core solution was a partial response to address the end of Dennard’s scaling. The solution quickly resulted in the utilization wall, i.e., the percentage of a multi-core chip that can actively switch drops exponentially due to power constraints, which is referred to as dark silicon or Utilization wall [12, 13]. To address the utilization wall, the general-purpose cores are becoming light-weight and more straightforward, for, e.g., removing energy-expensive branch prediction units, changing from out-of-order execution to in-order execution. Also, these processors have an increasing number of cores, and for instance, Intel KNL processor has close to 60 cores. GPGPU architectures also belong to this class of many-core processors, with each core being light-weight and more straightforward.

Programming: Traditionally, there have been two different approaches in programming multi-core architectures: the automatic parallelization and the explicitly-parallel programming approach. In the automatic parallelization approach, the programmer provides a portion of a sequential program or a high-level specification of a computation. Then, the compiler identifies parallelism available in the program and generates parallel codes

for a broad range of architectures. The alternate approach is to write explicitly-parallel programs, in which the programmer specifies the logical parallelism and explicit synchronizations in the source program, and the compiler extracts the parallelism subset that is best suited for a given target platform. In this approach, the programmer takes care of providing the parallelism required for performance, and then the compiler takes care of generating low-level code for the architecture.

Vector Processing (SIMD) Architectures

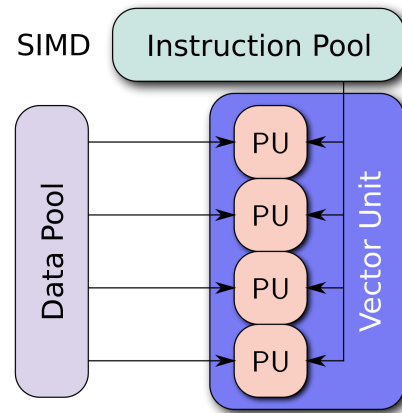


Figure 2.2: An abstract overview of a general-purpose SIMD architecture (figure source: [39]).

Single Instruction Multiple Data (SIMD) architectures are a class of parallel architectures introduced for vector supercomputers to exploit data-level parallelism for higher performance with better energy efficiency. In SIMD architectures, multiple processing elements perform the same operation on multiple data elements simultaneously. Also, these SIMD architectures can only be used by a single thread of instructions at any point in time, unlike multi-core processors that can simultaneously run multiple threads of instruction streams. However, modern multi-core processors often include multiple SIMD architectures (for, e.g., two SIMD units per single core in Intel KNL) in each core to improve the performance of data-parallel applications such as multimedia and machine learning. GPGPU architectures can also be viewed as a group of vector processing units since all threads in a GPU block execute in a SIMD fashion.

Programming: In general, application programmers rely on automatic vectorizing compilers to generate vector code automatically from a high-level programming language or annotations, because it is a very time-consuming task to deal with vector intrinsics, memory alignment, and also portability issues to other SIMD architectures. However, vector hardware vendors do provide low-level vector intrinsics/APIs to explicitly program.

2.1.2 Domain-Specific Parallel Architectures

Like general-purpose parallel architectures, domain-specific parallel architectures are also programmable devices with multiple compute units, but these architectures are limited to specific applications/domains. In general, the domain-specific architectures have custom datapaths with scratchpad buffers and specific functional units tailored for the specific domains. These processors are also designed to be extremely efficient in performance and energy for the specific applications/domains.

Spatial DNN Accelerators

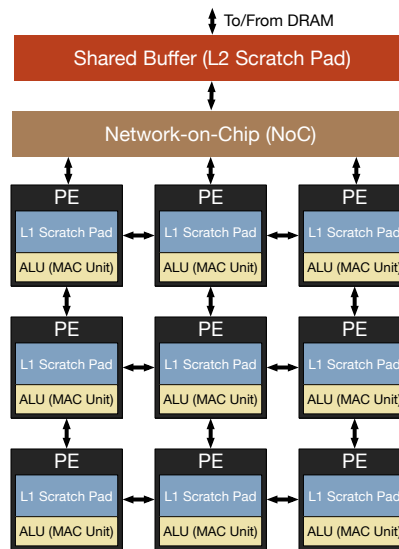


Figure 2.3: An abstract overview of a spatial DNN accelerator model which is pervasive in many state-of-the-art accelerators [40, 41, 42, 43, 44].

Spatial DNN accelerators based on ASICs and FPGAs have emerged to address extreme demands on performance and energy-efficiency of Deep Neural Networks (DNNs) [40, 45, 46, 42, 41, 43]. Such accelerators are built using an array of processing elements (PEs) to provide high parallelism and use direct communication instead of via shared memory for energy-efficiency. An abstract model of spatial accelerators is shown in Figure 2.3.

Each PE of an accelerator consists of a single/multiple ALU(s) dedicated to multiply-accumulate operations (MACs) and a local scratchpad (L1 buffer). Also, accelerators employ various network-on-chip (NoCs) for direct communication among PEs and between PE array and L2 scratchpad buffer. The interconnection network often supports multicasting data to multiple PEs, which can reduce the total number of data reads from L2 buffer to PEs. Unlike GPU cores, PEs can communicate with adjacent PEs (data forwarding) using an NoC, which can significantly reduce the energy consumption for expensive

L2 buffer accesses. Accelerators also typically employ a large shared L2 scratchpad buffer to stage data from DRAM and results from PE arrays. Note that both L1 and L2 scratchpad buffers are software-controlled memories, i.e., programmer directly controls the contents of each memory block, unlike cache memories, it implicitly stores data with contiguous addresses within a block for the spatial locality. This is because the memory traffic in accelerators is known in advance, unlike it is arbitrary in multi-core systems, allowing fine-tuning of memory block contents for further optimization. Many spatial accelerators can be further interconnected together to create a scale-out system [47].

Systolic arrays [43] are also popular DNN accelerators, which entirely relies on the point-to-point connection among adjacent PEs for input data distribution and partial sum accumulations. That is, systolic arrays distribute input data and accumulate partial sums via store-and-forward. Typically, systolic arrays are two dimensional, and each dimension is used for data forwarding and partial sum accumulation, respectively. Although systolic arrays can provide high throughput and energy efficiency, they lack flexibility in its data flow due to their rigid NoC architecture. Such inflexibility allows limited data flow style, leading to low compute unit utilization depending on the layer type and dimensions. Therefore, in this dissertation, we focus on spatial accelerators providing more flexibility from NoC to explore massive benefits from data flow/schedule optimizations.

Programming: Programmers rely on mappers (compilers) to effectively map deep learning computations and generate hardware-compatible code for a given spatial accelerator. The output code involves the configuration for each processing element and scratchpad.

Specialized Vector Processing Units

There is a strong resurgence of interest in improving vector processing (SIMD) units due to the significant energy efficiency benefits of using SIMD parallelism. These benefits increase with widening SIMD vectors, reaching vector register lengths of 2048 bits in the scalable vector extension of the Armv8 architecture [48]. There is an emphasis on specializing SIMD units to improve further energy efficiency benefits for specific domains such as Machine learning, Computer Vision, and 5G Wireless. An important specialization, which is referred to as “2D vector SIMD datapath” [31, 32, 33], is the ability of each vector lane to execute more than one scalar operation and to chain the results from one operation to another. Another specialization includes the removal of expensive data permutation units (e.g., shuffle units) [34, 35] and instead introduce sophisticated, programmable interconnection networks (a.k.a shuffle networks) between the SIMD datapath and vector register file to support the required data permutation patterns [36, 33].

A recent industry example with these specializations is the Xilinx Versal AI Engine [49],

a high-performance VLIW SIMD core which can deliver performance comparable to traditional FPGA solutions for Computer Vision, Deep Learning, and 5G wireless domains, but with 50% less power consumption and up to eight times more compute capacity per silicon area [49]. A high-level overview of the AI Engine architecture can be seen in Figure 2.4.

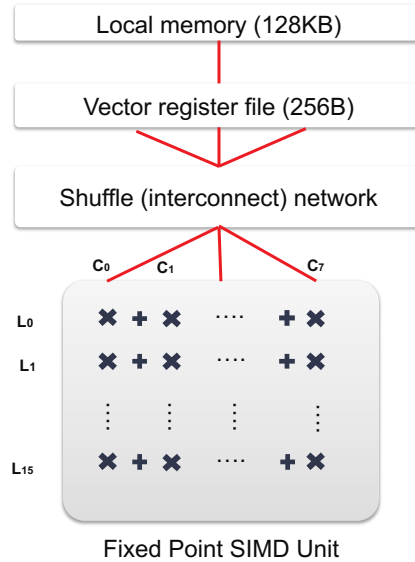


Figure 2.4: An abstract overview of a specialized vector processing unit in the Xilinx AI Engine.

Programming: These high-performance specialized AI Engines are programmed using the C/C++ programming language with optional pragmas. Currently, the AI Engine compilers do not advertise support for auto-vectorization, and application programmers write vectorized code explicitly using architecture-specific vector intrinsics. However, the AI Engine compilers have support for automatic software pipelining [50] of innermost loops to exploit instruction-level parallelism.

Thread Migratory Architectures

Since graph applications are typically cache-unfriendly and are not well supported by existing traditional architectures, there is growing attention being paid by the architecture community to innovate suitable architectures for such applications. One such innovation is the Emu system, a highly scalable near memory system with support for migrating threads without programmer intervention [15]. The system is designed to improve the performance of data-intensive applications that exhibit weak locality, i.e., from irregular and cache-unfriendly memory access, which are often found in graph analytics [51] and sparse matrix algebra operations [52].

An Emu system consists of multiple Emu nodes interconnected by a fast-rapid IO network, and each node (shown in Figure 2.5) contains *nodelets*, stationary cores and migra-

tion engines. Each *nodelet* consists of a Narrow Channel DRAM (NCDRAM) memory unit and multiple Gossamer cores, and the co-location of the memory unit with the cores makes the overall Emu system a near-memory system. Although each nodelet has a different physical co-located memory unit, the Emu system provides a logical view of the entire memory via the partitioned global address space (PGAS) model with memory contributed by each nodelet. Each gossamer core of a nodelet is a general-purpose, simple, pipelined processor with no support for data caches and branch prediction units. The core is also capable of supporting 64 concurrent threads using fine-grain multi-threading. A key aspect of the Emu system is thread migration by hardware, i.e., a thread is migrated on a remote memory read by removing thread context from the nodelet and transmitting the thread context to a remote nodelet without programmer intervention. As a result, each nodelet requires multiple queues such as service, migration, and run queues to process threads spawned locally (using *spawn* instruction) and also migrated threads.

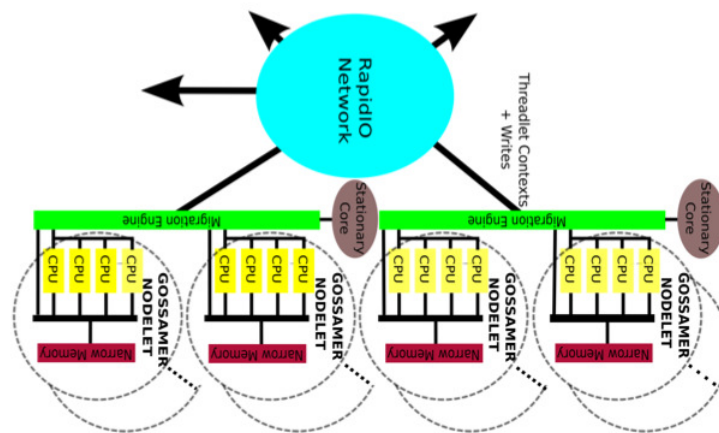


Figure 2.5: An abstract overview of a thread migratory architecture in the EMU (figure source: [53]).

Programming: The Emu system supports the Cilk parallel programming model for thread spawning and synchronization using `cilk_spawn`, `cilk_sync` and `cilk_for` constructs [54]. Since the Emu hardware automatically takes care of thread migration and management; hence the Cilk run-time is discarded in the toolchain. Also, it is essential to note that appending a `cilk_spawn` keyword before a function invocation to launch a new task is directly translated to the `spawn` instruction of the Emu ISA during the compilation. The Emu system provides libraries for data allocation and distribution over multiple nodelets, and intrinsic functions for atomic operations and migrating thread control functions. Also, there has been significant progress in supporting standard C libraries.

2.2 High-Performance Applications

2.2.1 Scientific Computing Applications

Scientific computing involves the development of computational models and simulations to solve problems science (e.g., biological, physical, and social) and engineering to understand them better. Some of the scientific computing applications include large-scale climate prediction, computational fluid dynamics, high-dimensional tensor contractions, and traditionally the scientific applications have dominated the need for performance to advance research in science and engineering.

Programming: Optimizing compilers [55, 56, 20, 21] involving automatic parallelization and vectorization was one of the major approaches in optimizing the scientific applications for higher-performance. But, with the limitations of automatic parallelization and vectorization arising from unanalyzable regions of code at compile time, writing explicitly-parallel programs using parallel programming models (e.g., OpenMP [57], Chapel [58], Cilk [59], and X10 [60]) along with libraries became the dominant approach for scientific applications to get better performance. However, domain-specific compilers (e.g., [61, 62]) have recently become popular to program particular domains of scientific application, because of its ability to exploit domain-specific properties in customizing optimizations for the domain and also the target hardware.

2.2.2 Deep Learning

Deep learning (DL) has become a fundamental technology for many emerging applications such as autonomous driving, translation, and image classification, with accuracy close to and even surpassing humans. The applications involve different Deep Neural Network models (DNNs) for different tasks in an application. Convolution Neural Networks (CNNs) are one of the most popular DNNs for image recognition [3]. Among many layers in CNN models, convolution layers account for more than 90% of overall computation [63, 40], dominating overall latency and energy consumption in inferences. A convolution is a mathematical operation that computes the amount of overlap of a function g as it is shifted over another function f , and it is symbolically represented as $f \circ g$. In this description, we restrict our attention to describing CONV2D, a popular convolution operator widely used in Deep learning [64, 65, 3, 66, 67, 68] and Computer Vision [69, 70, 71, 72]. In these domains, the function f and g are referred to as the “input” tensor (a.k.a image/activations) and “weight” tensor (a.k.a filters/kernels), respectively. The CONV2D deals with three four-dimensional tensors, i.e., Output (O), Weight (W), and Input (I), whose dimensions are described below.

Tensor	Dim1	Dim2	Dim3	Dim4
Output (O)	Width (X)	Height (Y)	Channels (K)	Batch (N)
Weight (W)	Width (R)	Height (S)	Channels (C)	Batch (K)
Input (I)	Width (X')	Height (Y')	Channels (C)	Batch (N)

The mathematical expression of the CONV2D operations is shown below, where f refers to stride factor.

$$O(x, y, k, n) = \sum_c^C \sum_s^S \sum_r^R W(r, s, c, k) \times I(x \times f + r, y \times f + s, c, n)$$

The convolutions used in Computer Vision are special cases of the CONV2D operator, where each tensor has only the first two dimensions (width and height) and stride factor set to one. However, there exists a wide variety of filter sizes (ranging from 2 to 11) used in many different image processing operators, such as Gaussian smoothing and edge detection [69]. A wide variety of other specialized variations of the CONV2D operator are used in Convolutional Neural Networks such as point-wise, depth-wise separable, and spatially separable convolutions. These variations can be viewed as constraints on the regular CONV2D operator, and these are shown below.

Operator	Constraints on CONV2D
Point-wise (PW)	Filter width = Filter height = 1
Fully-connected (FC)	Filter width = Input width Filter height = Input height
Spatially separable (SS)	Filter width = 1 or Filter height = 1
Depth-wise separable (DS)	Input channels = Filter channels = 1

Furthermore, the CONV2D operator can also be used to describe other DNN operators such as LSTMs [73] used in Recurrent Neural Networks. Even though we briefly described the CONV2D operator and its variations, our approach applies to other convolution operators such as CONV1D and CONV3D.

Programming: High-performance tensor convolutions are generally realized through high-performance libraries such as Intel MKL/DNNL [74] for Intel platforms, NVIDIA cuDNN [75] for NVIDIA platforms, and also use of domain-specific compilers such as Halide [70], TVM [76] for a variety of target platforms including CPUs, GPUs, and FPGAs.

2.2.3 Graph Analytics

Graph analytics or Graph algorithms are used to uncover insights about information stored in graph representation, i.e., to determine the strength and direction of relationships between objects in a graph [77]. Graph analytics applications include performing the breadth-first search traversal, shortest path solution, finding connected components, and page rank. In the last decade, large-scale graph processing has become a relevant application domain for high performance because of the prevalence of graph data in real-world applications such as social networks and their rapidly increasing size [1, 2].

Programming: In general, high-performance graph algorithms are programmed using frameworks such as Google’s Pregel [78] for distributed systems, and also using high-performance libraries such as nvGraph [77] for GPUs. Also, domain-specific compilers such as Green-Marl [79] and GraphIt [80] are also becoming a popular approach to generate efficient high-performant code from a high-level specification of graph algorithms for a variety of target hardware platforms.

2.3 Summary

From this chapter, we can observe two trends: the computer hardware is undergoing through a significant disruption to improve computing capabilities as we approach the end of Moore’s Law, for, e.g., in the form of light-weight multi-cores, specialized SIMD units, spatial accelerators, and thread migratory hardware. Also, the demand for higher performance is broadening across multiple application domains, and also these application domains are evolving at a rapid pace. The above trends pose a plethora of challenges to application development using high-performance libraries (as can be seen from this chapter on programming), which is the de-facto approach to achieving higher performance. Some of the challenges in using library-based approaches are porting/adapting to multiple emerging parallel architectures, supporting rapidly advancing domains, and inhibiting optimizations across library calls. Hence, there is a renewed focus on optimizing compilers (including domain-specific compilers) from industry and academia to address the above trends. However, it requires advancements in enabling them to a wide range of applications and better-exploiting current and future parallel architectures, which is the focus of this dissertation, and the rest of the thesis chapters focus on advancements to the optimizing compilers.

CHAPTER 3

POPP: POLYHEDRAL OPTIMIZATIONS OF EXPLICITLY-PARALLEL PROGRAMS

3.1 Abstract

The polyhedral model is a powerful algebraic framework that has enabled significant advances to analysis and transformation of sequential affine (sub)programs, relative to traditional AST-based approaches. However, given the rapid growth of parallel software, there is a need for increased attention to using polyhedral frameworks to optimize explicitly parallel programs. An interesting side effect of supporting explicitly parallel programs is that doing so can also enable optimization of programs with unanalyzable data accesses within a polyhedral framework. In this thesis, we address the problem of extending polyhedral frameworks to enable analysis and transformation of programs that contain both explicit parallelism and unanalyzable data accesses. As a first step, we focus on OpenMP loop parallelism and task parallelism, including task dependences from OpenMP 4.0.

A summary of our approach [26, 27] is as follows. We first enable conservative dependence analysis of a given region of code by introducing dummy variables that can work with any polyhedral tool that supports *access functions*. After obtaining conservative dependences, the Fourier-Motzkin elimination method is used to remove all dummy variables. Next, we identify happens-before relations from the explicitly parallel constructs, notably parallel loops and tasks, and intersect them with the conservative dependences. The resulting set of dependences is then passed on to a polyhedral optimization tool, such as PolyAST, to enable transformation of explicitly-parallel programs with unanalyzable data accesses.

We evaluate our approach using twelve OpenMP benchmark programs from the KASTORS and Rodinia benchmark suites. We show that 1) these benchmarks contain unanalyzable data accesses that prevent polyhedral frameworks from performing exact dependence analysis, 2) explicit parallelism can help mitigate the imprecision, and 3) polyhedral transformations with the resulting dependences can further improve the performance of manually-parallelized OpenMP programs. Our experimental results show performance improvements for these OpenMP programs on a 12-core Intel Westmere platform and a 24-core IBM Power8 platform.

3.2 Introduction

A key challenge for optimizing compilers is to keep up with the increasing complexity related to locality and parallelism in modern computers, especially as computer vendors head towards new designs for extreme-scale processors and exascale systems [81]. Classical AST-based optimizers typically focus on one particular objective at a time, such as vectorization, locality or parallelism. In contrast, polyhedral transformation frameworks support complex sequences of transformations of perfectly/imperfectly nested loops in a unified formulation. The advantages of this unified formulation are seen in polyhedral optimizers, such as PLuTo [20, 21] and PolyAST [22]. It has even been extended and specialized to integrate SIMD constraints [23]. Polyhedral frameworks achieve this generality in transformations by restricting the class of programs that do not have *unanalyzable* control or data flow. In the original formulation of polyhedral frameworks, all array subscripts, loop bounds, and branch conditions in *analyzable* programs were required to be affine functions of loop index variables and global parameters. However, decades of research since then have led to a great expansion of programs that can be considered analyzable by polyhedral frameworks. The main remaining constraints stem from restrictions on various program constructs including pointer aliasing, unknown function calls, non-affine expressions, recursion, and unstructured control flow.

Our work is motivated by the observation that software with explicit parallelism is on the rise. It can be used to enable larger set of polyhedral transformations (by mitigating conservative dependences), compared to what might have been possible if the input program is sequential. Our work focuses on explicitly-parallel programs that specify potential logical parallelism, rather than actual parallelism. Thus, explicit parallelism is simply a specification of a partial order, traditionally referred to as a happens-before relations [82]. Dependences can only occur among statement instances that are ordered by the happens-before relations. Hence, we can reduce spurious dependences arising from the unanalyzable constructs by intersecting happens-before relations with conservative dependences.

In this work, we restrict our attention to explicitly-parallel programs that satisfy the *serial elision* property, i.e., the property that removal of all parallel constructs results in a sequential program that is a valid (albeit inefficient) implementation of the parallel program semantics [54]. We observe that loop-level and task-level parallelism form the core of modern parallel programming languages, such as OpenMP [57], Chapel [58], Cilk [59], and X10 [60]. So, we focus our attention on loop-level and task-level constructs in OpenMP that satisfy the serial elision property, while deferring support for SPMD constructs that do not satisfy this property to future work.

A summary of our approach is as follows. We first enable conservative dependence analysis of a given region of code. Next, we identify happens-before relations from the explicitly parallel constructs and intersect with the conservative dependences. Finally, the resulting set of dependences is passed to the polyhedral transformation tools, such as PLuTo [20, 21] and PolyAST [22], to enable the transformations of explicitly-parallel programs with unanalyzable data accesses. To the best of our knowledge, our work is the first to enable the polyhedral transformations of explicitly parallel OpenMP programs by combining the classical dependence analysis with happens-before analysis for explicit parallelism¹.

The rest of the chapter is organized as follows. Section 3.3 summarizes background material, and Section 3.4 motivates the problem using OpenMP benchmark programs. Section 3.5 provides an overview of our approach for enabling polyhedral transformations of explicitly parallel programs; we refer to our framework as the Polyhedral optimizer for Parallel Programs (PoPP). Section 3.6 presents experimental results to evaluate our approach on OpenMP benchmarks from the Kastors [83] and the Rodinia [84] benchmark suites on a 12-core Intel Westmere processor and a 24-core IBM Power8 system.

3.3 Background

We start with a brief overview of the polyhedral model, the basis of the proposed optimizing framework. Next, we briefly summarize explicit-parallelism including loop-level and task-level parallelism in the context of OpenMP [57], which is a widely used shared memory parallel programming model.

3.3.1 Polyhedral Model

The polyhedral model is a flexible representation for perfect and imperfect loop nests with static predictable control. Loop nests amenable to this algebraic representation are called *Static Control Parts* (SCoPs). It consists of a set of consecutive statements, and each statement contains three elements namely iteration domain, access relations, and schedule. The loop bounds, branch conditions, and array subscripts in the SCoP need to be affine functions of iterators and global parameters. A code region that does not strictly satisfy the above requirements can be also represented in the polyhedral model via conservative estimations. **Iteration domain**, \mathcal{D}_S : A statement S enclosed by ‘ m ’ loops is represented by a m -dimensional polytope, referred to as an iteration domain of the statement [85]. Each element in the iteration domain of the statement is regarded as a statement instance.

¹An earlier version of this work was informally presented at the IMPACT’15 workshop [26], which is a forum that does not include formal proceedings.

Access relation: Each array expression in the statement is expressed through an access relation in the SCoP. An access relation maps the statement instance to one or more array elements [86]. It can conservatively support non-affine array expressions by mapping them to multiple array elements, perhaps even the entire range of the array. An example of a non-affine array access is shown below. The array reference to \mathbf{x} is an indirect access via $\text{col}[j]$ and is considered to read the entire range of $\mathbf{x}[*]$ to enable conservative estimations. In contrast, an access function maps a statement instance to a single array element, and cannot support non-affine accesses as a result.

```

1 for(i = 0; i < n; i++)
2   for(j = index[i]; j < index[i+1]; j++)
3     y[i] += A[j]*x[col[j]];

```

Schedule: is a function which associates a logical execution date (a timestamp) to each instance of a given statement. In the case of multidimensional schedules, this timestamp is a vector. In the program, statement instances will be executed according to the increasing lexicographic order of their timestamp.

Dependence Polyhedra, $\mathcal{P}^{S \rightarrow \mathcal{T}}$: captures all possible dependences between statements S and \mathcal{T} . Two statement instances \vec{X}_s and \vec{X}_t , which belong to the iteration domains of statements S and \mathcal{T} respectively, are said to be in dependence if they access the same array location and at least one of them is a write. Multiple dependence polyhedra may be required to capture all dependent instances between two statements (scalars are simply treated as zero-dimensional arrays). For a given schedule, *depth* of a dependence polyhedron indicates the loop nest level where its dependence is carried. In other words, depth is the first non-zero dimension of the corresponding dependence vector.

A dependence polyhedron captures exact dependence information when each of the access relations is an access function or if the access relation models an exact read/write of an array range, e.g., a memset of an entire array. However, dependence polyhedra can be over-estimated due to conservative access relations when array subscripts include unanalyzable accesses.

3.3.2 Explicit Parallelism

The major difference between sequential programs and explicitly-parallel programs is that sequential programs specify a total execution order, whereas the execution of an explicitly-parallel program can be viewed as a partial order, which is traditionally referred to as a happens-before relation. We briefly summarize the loop-level and task-level constructs in the context of OpenMP [57].

Loop-level parallelism

The OpenMP loop construct, `#pragma omp for`, is specified immediately before a for loop. This construct indicates that the iterations of the loop can be executed in parallel, which guarantees no happens-before relations among iterations. A barrier, i.e., an all-to-all synchronization point, is implied immediately after the parallel loop region.

The `private(op: list)` clause, which is attached to a for loop construct, indicates that each OpenMP thread has its own private copies of the variables specified in *list*.

Task-level parallelism including dependences

The OpenMP task construct, `#pragma omp task`, is specified on a code region and indicates the creation of an asynchronous task to process the region. Synchronization among the parent task and its child tasks (i.e., tasks spawned by the parent task) is supported by the `taskwait` construct, `#pragma omp taskwait`. This directive specifies a synchronization point at which the encountering task waits for all its child tasks to complete. Synchronization among the sibling tasks with the same parent task is supported by the `depend(type: vars)` clauses attached on a task construct. Here, *type* is `in`, `out`, or `inout` to imply read, write, or read-and-write access on *vars*, which is a list of variables that can include arrays². The ordering constraints enforced by the `depend` clauses are as follows:

- *in dependence-type*. The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `out` or `inout` `depend` clause.
- *out and inout dependence-types*. The generated task will be a dependent task of all previously generated sibling tasks that reference at least one of the list items in an `in`, `out`, or `inout` `depend` clause.

A task can start its execution only when all the dependent tasks have completed. These dependences on previous generated tasks enforce *serial elision* property. More details on these constructs can be found in [87].

3.4 Motivating Examples

To motivate the proposed approach, we discuss two explicitly parallel kernels with data accesses that are likely to be considered unanalyzable by many existing polyhedral frameworks. The first example uses C nested arrays which may have an unrestricted pointer

²Any type or rank of arrays are permitted; in Figure 3.1, 1-D array of type `double (*)[ny]` are used.

aliasing, in general. The second example uses linearized (non-affine) array subscripts that would require a de-linearization analysis to make them analyzable by polyhedral frameworks.

3.4.1 2-D Jacobi

```

1 jacobi (double *u_, double *unew_, double *f_) {
2 double (*f)[nx][ny] = (double (*)(nx)[ny])f_;
3 double (*u)[nx][ny] = (double (*)(nx)[ny])u_;
4 double (*unew)[nx][ny] = (double (*)(nx)[ny])unew_;

6 #pragma omp parallel
7 #pragma omp single
8 {
9 for (int it = itold + 1; it <= itnew; it++) {
10 for (int i = 0; i < nx; i++) {
11 #pragma omp task depend(out: u[i]) depend(in: unew[i])
12 for (int j = 0; j < ny; j++) {
13 (*u)[i][j] = (*unew)[i][j];
14 } }
15 for (int i = 0; i < nx; i++) {
16 #pragma omp task depend(out: unew[i]) depend(in: f[i], u[i-1], ←
    u[i], u[i+1])
17 for (int j = 0; j < ny; j++) {
18 if (i == 0 || j == 0 ||
19 i == nx - 1 || j == ny - 1) {
20 (*unew)[i][j] = (*f)[i][j];
21 } else {
22 (*unew)[i][j] = 0.25 * ((*u)[i-1][j]
23 + (*u)[i][j+1] + (*u)[i][j-1]
24 + (*u)[i+1][j] + (*f)[i][j] * dx * dy);
25 } } } }
26 #pragma omp taskwait
27 } }

```

Figure 3.1: 2-D Jacobi kernel from KASTORS suite.

The first example (in Figure 3.1) is a 2-dimensional Jacobi computation from the KASTORS suite [83]. This computation is parallelized using the OpenMP 4.0 task construct with depend clauses. Even though the loop nest has affine accesses on arrays `u` and `unew`, the possible aliasing of the flat array pointers can prevent a sound compiler analysis from detecting the exact cross-iteration dependences. However, the happens-before relations described through the `task depend` clauses (lines 13-14, lines 19-20) indicate uniform dependence patterns only among neighboring iterations (i.e., `u[i-1]`, `u[i]`, and `u[i+1]`),

which enable skewing, tiling, and doacross pipelined parallelization. Section 3.6 shows how these transformations improve the data locality and the parallelism granularity and contribute the overall performance. However, there exist speculative approaches that add code to the program to check if all referenced arrays of a loop nest do not overlap and to generate optimized variants that can be selected at runtime [88].

3.4.2 Particle Filter

The second example (in Figure 3.2) is the `particle filter` kernel from the Rodinia suite [84]. The loop nests in the kernel contain linearized (non-affine) array subscripts such as `ind[x*countOnes+y]`, and indirect array subscript (`I[ind[x*countOnes+y]]`), that may pose challenges to the compiler for analysis.

Although de-linearization techniques [89] can handle the `ind[x*countOnes+y]` case, and the fact that array `I` is read-only in the kernel can be used to handle the `I[ind[x*countOnes+y]]` case, the use of parallel loop constructs can prune conservativeness in dependence analysis, even in the absence of techniques such as delinearization. The legality of loop fusion is easily established by the fact that all variables that cross multiple loops have affine accesses with no fusion-preventing dependences and the arrays don't alias each other, as these arrays are malloc(ed) in the same kernel. The key information needed from the parallel program is that the second loop (lines 19-28 in Figure 3.2) has no loop-carried dependence. This ensures that the resulting loop after fusing all four loops can also be made parallel.

3.5 Polyhedral optimizer for Parallel Programs (PoPP)

In this section, we introduce our framework for automatically optimizing explicitly parallel programs.

Algorithm 1: Overall steps in PoPP

- 1: **Input:** Explicitly parallel program, \mathcal{I}
 - 2: $\mathcal{P} :=$ set of conservative dependences in \mathcal{I}
 - 3: $\mathcal{HB} :=$ Transitive closure of happens-before relations from parallel constructs in \mathcal{I}
 - 4: $\mathcal{P}' := \mathcal{P} \cap \mathcal{HB}$,
 - 5: Optimized schedules, $\mathcal{S} = \text{Transform}(\mathcal{I}, \mathcal{P}')$
 - 6: $\mathcal{I}' = \text{CodeGen}(\mathcal{I}, \mathcal{S}, \mathcal{P}')$
 - 7: **Output:** Optimized explicitly parallel program, \mathcal{I}'
-

Algorithm 1 shows the overall approach to conservatively handle unanalyzable accesses (step 2), extract happens-before (HB) relations from explicit parallelism (step 3), and improve the accuracy of conservative dependences (step 4). Then the resulting dependences

```

1 #define ALLOC(N) (double *) malloc(sizeof(double)*N)
2 void particleFilter(int *I, int Nparticles) {
3     ....
4     double *weights = ALLOC(Nparticles);
5     double *arrayX = ALLOC(Nparticles);
6     double *arrayY = ALLOC(Nparticles);
7     double *likelihood = ALLOC(Nparticles);
8     double *objxy = ALLOC(countOnes*2);
9     int *ind = (int*)malloc(sizeof(int) * countOnes*Nparticles);

11 #pragma omp parallel for
12 for(x = 0; x < Nparticles; x++){
13     arrayX[x] += 1 + 5*randn(seed, x);
14     arrayY[x] += -2 + 2*randn(seed, x);
15 }

17 #pragma omp parallel for private(y, indX, indY)
18 for(x = 0; x < Nparticles; x++){
19     for(y = 0; y < countOnes; y++){
20         indX = roundDouble(arrayX[x]) + objxy[y*2+1];
21         indY = roundDouble(arrayY[x]) + objxy[y*2];
22         ind[x*countOnes+y] = fabs(indX ... indY ...);
23         ...
24         likelihood[x] += ...I[ind[x*countOnes+y]]...
25     }
26     ...}

28 #pragma omp parallel for
29 for(x = 0; x < Nparticles; x++){
30     weights[x] = weights[x] * exp(likelihood[x]);

32 #pragma omp parallel for private(x) reduction(+:sumWeights)
33 for(x = 0; x < Nparticles; x++)
34     sumWeights += weights[x];
35 }
36     ....
37 }

```

Figure 3.2: Particle filter kernel from Rodinia suite

are passed to polyhedral optimizers, such as PolyAST and PLuTo, to leverage existing loop transformations (step 5). Finally, the code generator is invoked to generate the optimized parallel program (step 6).

The overall approach is summarized in Figure 3.3, which is implemented as an extension to the PolyAST optimization framework [22] implemented in the ROSE compiler [90],

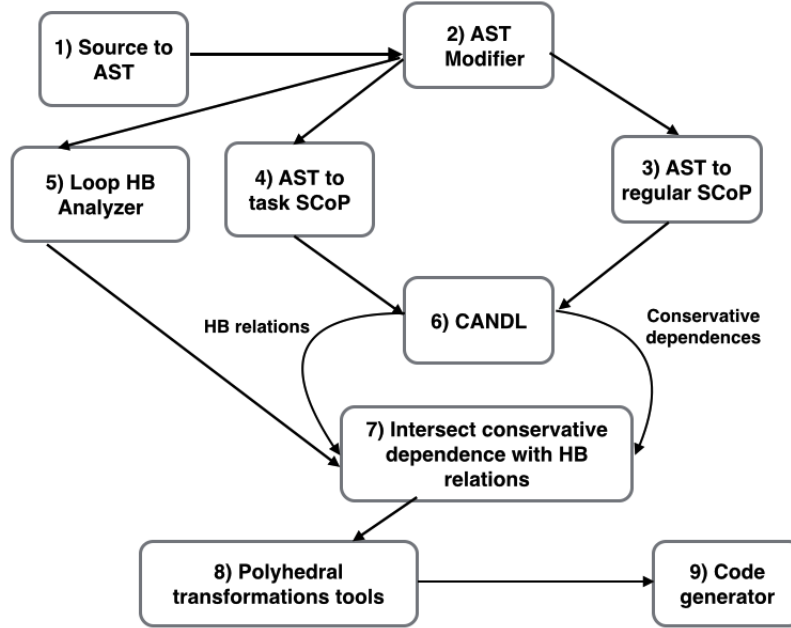


Figure 3.3: Overview of our approach

and consists of the following components: 1) Conversion from source code to AST (with support for parallel-loop and parallel-task constructs), 2) AST Modifier (handling unanalyzable accesses), 3) AST to SCoP converter for regular statements, 4) AST to task SCoP converter (preprocessing of computing HB relations based on task parallelism), 5) Loop HB analyzer to compute HB relations based on loop level parallelism, 6) Use of CANDL [91] for both conservative dependence analysis and computing task-based HB relations, 7) Intersection of conservative dependences with HB relations, 8) Communication of the resulting set of dependences to a polyhedral transformation tool, such as PLuTo [20] and PolyAST [22], and 9) Code generator to produce automatically optimized code.

3.5.1 Conservative analysis

In case of unanalyzable data accesses, compilers must follow conservative dependence analysis that overestimates dependences and may report spurious dependences. In conservative dependence analysis, the basic assumption for a compiler is that all memory accesses of an array in a statement can potentially conflict with other memory accesses of that array, or perhaps even memory accesses in other arrays (in the case of unrestricted pointer aliasing). In this work, we support such conservative assumptions by using the *dummy variable* approach described below.

Handling non-affine array subscripts. Non-affine subscripts such as linearized array subscripts and indirect array subscripts are common in regular benchmarks. These non-

affine subscripts can be handled using the *access relations* in polyhedral extraction tools by assuming that they access the entire range instead of a single element in the array. Since our polyhedral framework in the infrastructure supports access functions but not access relations, we implemented similar functionality using a *dummy variable* approach. We treat non-affine subscript as a dummy variable and create affine inequalities such that these variables access the entire range of the array dimension [26]. There exist other conservative approaches such as array region analysis [92], fuzzy array data flow analysis [93] and other variants to approximate the access relations for arrays having non-affine subscripts.

Handling function calls. Function calls in the kernel pose challenges to polyhedral frameworks for analysis and transformations. We handle library/ user-defined function calls by treating them as regular statements and conservatively assume the statements read and write any array in the SCoP. But, there exist other sophisticated approaches such as array region analysis [94] used in PIPS compiler to approximate access relations and enhance dependence analysis in case of procedure calls.

Handling non-affine conditionals. Currently, polyhedral extraction tools have limitations in representing non-affine branch conditions in a polyhedral representation (SCoP). As a workaround, we handle an `if`-statement with a non-affine conditional and its corresponding `then` and `else` branches as a compound statement that inherits all the access relations in the condition, and the `then` and `else` branches³. Note that we allow multi-write per statement in the framework. For the benchmarks studied in Section 3.6, such non-affine control flows are closed within a loop body; this approximation keeps the granularity of compound statements small enough to enable transformations and parallelization. The Polyhedral Extraction Tool (PET) [95] also provides a way to represent data dependent assignments, data dependent accesses and data dependent conditions in the access relations.

After converting the given parallel program into polyhedral representation (SCoP) with above modifications, we use an existing polyhedral dependence analyzer (CANDL [91]) with the sequential schedule ignoring parallel constructs. The resulting dependence polyhedra can be directly used as conservative dependences. Adhering to polyhedral dependence notations, we use $\mathcal{P}_d^{S_i \rightarrow S_j}$ to represent the dependence between source statement S_i and target statement S_j at depth d where depth represents the loop nest level that carries the data dependence. The conservative dependences for the Jacobi kernel (in Figure 3.5(a)) are shown in Figure 3.5(c).

³This has been manually performed for programs with non-affine conditionals in Section 3.6, but can be automated in future work.

3.5.2 Extraction of happens-before relations

Happens-before (HB) relations [82] have been introduced in describing memory models. These relations can be defined as follows in the context of dependences between statements in the program.

*Assume S_i and S_j are the statements in the program. If S_i **happens-before** S_j , then the memory effects of S_i effectively become visible before statement S_j is executed.*

Explicit parallel constructs in the program specify the logical parallelism, which in turn describes the happens-before relations on the statements in the program. Let $\mathcal{U}_d^{S_i \rightarrow S_j}$ represent a given sequential ordering between source statement S_i and target statement S_j at depth d in the program when ignoring parallel constructs. Any happens-before relation is initialized to this sequential ordering:

$$\mathcal{HB}_d^{S_i \rightarrow S_j} = \mathcal{U}_d^{S_i \rightarrow S_j} \quad (3.1)$$

According to the explicit parallel constructs, the happens-before relations will be updated. This section introduces our approach to compute such happens-before relations in the cases of loop-parallel and task-parallel constructs, where the *serial-elision* property holds.

Loop-level parallelism

In the OpenMP, loop-level parallelism is expressed through the `#pragma omp parallel for` construct. This particular construct is annotated on specific loops whose iterations can run in parallel, thereby it guarantees there are no happens-before relations among iterations of the annotated loop. Let S_i and S_j be statements enclosed in a `parallel for` loop at depth d , the corresponding happens-before relation is updated as:

$$\mathcal{HB}_d^{S_i \rightarrow S_j} = \phi \quad (3.2)$$

Note that the variables specified within `private` clause are expanded as arrays such that each parallel iteration accesses a unique element of the arrays, before the polyhedral compilation. In the post-polyhedral phase, the expanded arrays are replaced by the original variables with `private` attribute if they remain in parallel loops. This approach only applies to cases where each parallel loop is in an OpenMP parallel region by itself, and not to general OpenMP parallel regions (which are not supported by the approach in this work).

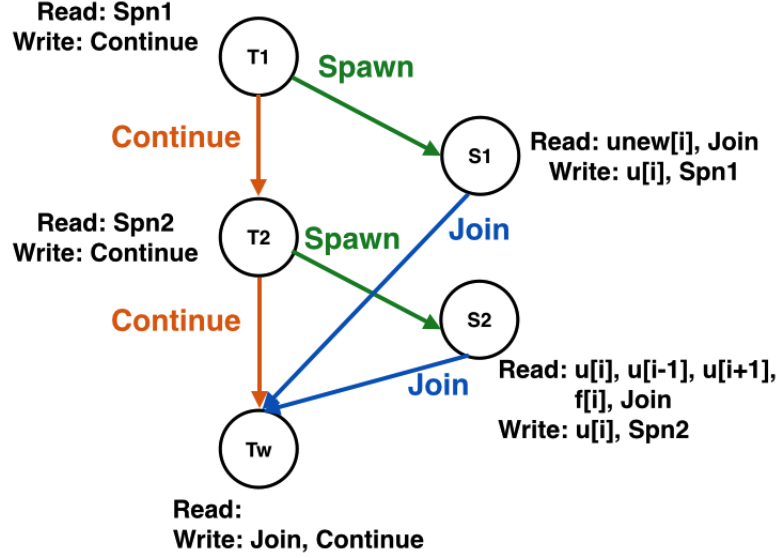


Figure 3.4: Happens-before relations for the Jacobi program in Figure 3.5(a) due to task-spawn, task-wait, and sequential ordering

Task parallelism including dependences

In OpenMP, task parallelism is specified through the `task`, `taskwait`, and `depend` constructs. As described in Section 3.3.2, these constructs specify ordering constraints 1) from parent task to child tasks via *task-spawn*, 2) from child tasks to parent task via *task-wait*, and 3) among sibling tasks via *inter-task dependence*. Computing happens-before relations in the presence of the inter-task dependences is challenging as it requires dependence analysis on the variables including arrays listed in the `depend(in/out)` clause.

In our approach, we encode these task-related constructs in the SCoP format by handling tasks as statements and `in/out` dependence type as read/write access; this is later processed by polyhedral dependence analyzers such as CANDL [91]. The resulting dependence polyhedra, $\mathcal{P}_{task_d}^{S_i \rightarrow S_j}$, are used as happens-before relations due to inter-task dependences among sibling tasks:

$$\mathcal{HB}_d^{S_i \rightarrow S_j} = \mathcal{P}_{task_d}^{S_i \rightarrow S_j} \quad (3.3)$$

The other relations among parent and child tasks are captured in the same manner by introducing special dependence variables shared by parent and children. We detail our approach in the rest of this section.

Task SCoP. Given a code region that contains `task` constructs, we define *task* SCoP that captures all required information to compute happens-before relations derived from tasks. As with regular SCoP, it includes set of statements and access relations which are modified as follows while domains and schedules are same as those in regular SCoP. **Statement.**

The statements not enclosed in a task construct are handled in the same manner as regular SCoP statements. The `#pragma omp task` and `#pragma omp taskwait` are also handled as stand-alone statements that represent task-spawn and task-wait points, respectively. Finally, the body of task construct is handled as a compound statement, say *task-body* statement.

```

1 jacobi (double *u_, double *unew_, double *f_)
2 {
3   double (*f)[nx][ny] = (double (*)[nx][ny])f_;
4   double (*u)[nx][ny] = (double (*)[nx][ny])u_;
5   double (*unew)[nx][ny] = \
6     (double (*)[nx][ny])unew_;
7
8   #pragma omp parallel
9   #pragma omp single
10  {
11    for (int it = itold + 1; it <= itnew; it++) {
12      for (int i = 0; i < nx; i++) {
13        #pragma omp task depend(out: u[i]) \
14          depend(in: unew[i]) // T1
15          for (int j = 0; j < ny; j++)
16 S1:      u[i][j] = unew[i][j];
17      }
18      for (int i = 0; i < nx; i++) {
19        #pragma omp task depend(out: unew[i]) \
20          depend(in: f[i], u[i-1], u[i], u[i+1]) // T2
21          for (int j = 0; j < ny; j++)
22 S2:      cpd(i, j, unew, u, f);
23      }
24    }
25    #pragma omp taskwait // Tw
26  }
27 /* Original schedule
28   S1: (0, it, 0, i, 0, j, 0)
29   S2: (0, it, 1, i, 0, j, 0) */

```

```

1 jacobi (double *u_, double *unew_, double *f_)
2 {
3   double (*f)[nx][ny] = (double (*)[nx][ny])f_;
4   double (*u)[nx][ny] = (double (*)[nx][ny])u_;
5   double (*unew)[nx][ny] = \
6     (double (*)[nx][ny])unew_;
7
8   #pragma omp parallel for private(c3,c5) ordered(2)
9   for (c1 = itold + 1; c1 <= itnew; c1++) {
10    for (c3 = 2 * c1; c3 <= 2 * c1 + nx; c3++) {
11      #pragma omp ordered depend(sink: c1-1, c3) \
12        depend(sink: c1, c3-1)
13      if (c3 <= 2 * c1 + nx - 1) {
14        for (c5 = 0; c5 < ny; c5++)
15 S1:      u[-2*c1+c3][c5] = unew[-2*c1+c3][c5];
16      }
17      if (c3 >= 2 * c1 + 1) {
18        for (c5 = 0; c5 < ny; c5++)
19 S2:      cpd(-2*c1+c3-1, c5, unew, u, f);
20      }
21    }
22    #pragma omp ordered depend(source)
23  }
24 }
25 }
26
27 /* Transformed schedule
28   S1: (0, it, 0, 2*it+i, 0, j, 0)
29   S2: (0, it, 0, 2*it+i+1, 1, j, 0) */

```

(a) Input program: Jacobi from KASTORS suite; cpd represents compound statement.

(b) Jacobi after skewing and doacross pipelined parallelism; tiling was omitted due to space limitation.

$\mathcal{P}_1^{S1 \rightarrow S1} : it' \geq it + 1, i' = *, i = *, j' = *, j = *$	$\mathcal{H}\mathcal{B}_1^{S1 \rightarrow S1} : it' \geq it + 1, i' = i$	$\mathcal{P}'_1^{S1 \rightarrow S1} : it' \geq it + 1, i' = i, j' = *, j = *$
$\mathcal{P}_2^{S1 \rightarrow S1} : it' = it, i' \geq i + 1, j' = *, j = *$	$\mathcal{H}\mathcal{B}_2^{S1 \rightarrow S1} : \phi$	$\mathcal{P}'_2^{S1 \rightarrow S1} : \phi$
$\mathcal{P}_3^{S1 \rightarrow S1} : it' = it, i' = i, j' \geq j + 1$	$\mathcal{H}\mathcal{B}_3^{S1 \rightarrow S1} : it' = it, i' = i, j' \geq j + 1$	$\mathcal{P}'_3^{S1 \rightarrow S1} : it' = it, i' = i, j' \geq j + 1$
$\mathcal{P}_1^{S1 \rightarrow S2} : it' \geq it, i' = *, i = *, j' = *, j = *$	$\mathcal{H}\mathcal{B}_1^{S1 \rightarrow S2} : it' \geq it, i' = i$	$\mathcal{P}'_1^{S1 \rightarrow S2} : it' \geq it, i' = i, j' = *, j = *$
	$\cup it' \geq it, i' = i + 1$	$\cup it' \geq it, i' = i + 1, j' = *, j = *$
	$\cup it' \geq it, i' = i - 1$	$\cup it' \geq it, i' = i - 1, j' = *, j = *$
$\mathcal{P}_1^{S2 \rightarrow S1} : it' \geq it + 1, i' = *, i = *, j' = *, j = *$	$\mathcal{H}\mathcal{B}_1^{S2 \rightarrow S1} : it' \geq it + 1, i' = i$	$\mathcal{P}'_1^{S2 \rightarrow S1} : it' \geq it + 1, i' = i, j' = *, j = *$
	$\cup it' \geq it + 1, i' = i + 1$	$\cup it' \geq it + 1, i' = i + 1, j' = *, j = *$
	$\cup it' \geq it + 1, i' = i - 1$	$\cup it' \geq it + 1, i' = i - 1, j' = *, j = *$
$\mathcal{P}_1^{S2 \rightarrow S2} : it' \geq it + 1, i' = *, i = *, j' = *, j = *$	$\mathcal{H}\mathcal{B}_1^{S2 \rightarrow S2} : it' \geq it + 1, i' = i$	$\mathcal{P}'_1^{S2 \rightarrow S2} : it' \geq it + 1, i' = i, j' = *, j = *$
$\mathcal{P}_2^{S2 \rightarrow S2} : it' = it, i' \geq i + 1, j' = *, j = *$	$\mathcal{H}\mathcal{B}_2^{S2 \rightarrow S2} : \phi$	$\mathcal{P}'_2^{S2 \rightarrow S2} : \phi$
$\mathcal{P}_3^{S2 \rightarrow S2} : it' = it, i' = i, j' \geq j + 1$	$\mathcal{H}\mathcal{B}_3^{S2 \rightarrow S2} : it' = it, i' = i, j' \geq j + 1$	$\mathcal{P}'_3^{S2 \rightarrow S2} : it' = it, i' = i, j' \geq j + 1$

(c) Conservative dependences, \mathcal{P}

(d) Happens-before relations, $\mathcal{H}\mathcal{B}$

(e) Accurate dependences, $\mathcal{P}' = \mathcal{P} \cap \mathcal{H}\mathcal{B}$

Figure 5: Overall explanation of our framework on Jacobi benchmark from KASTORS suite.

Figure 3.5: Overall explanation of our framework on Jacobi benchmark from KASTORS suite.

Figure 3.4 shows an example corresponding to the Jacobi kernel in Figure 3.5(a), where two task-spawn statements are represented as T1 and T2, a task-wait statement is Tw, and task-body statements are shown as S1 and S2.

Access relation. The in and out dependence types in depend clause are respectively handled as read and write accesses in the corresponding task-body statement (e.g., read: `unew[i]`, write: `u[i]` of S1 in Figure 3.4). In order to capture happens-before relations between parent and child tasks, we introduce the following special dependence variables and add to access relations.

- Spawn variable `Spni` is added as a read access in *i*-th task-spawn statement and a write access in its task-body statement; the resulting Write-After-Read (WAR) dependence captures the ordering constraint on this specific task-spawn.
- Join variable `Join` is added as a read access in task-body statements and a write access in task-wait statements so that the WAR dependences capture ordering constraints on task-wait, which waits for all child tasks.
- Continue variable `Continue` is added as a write access in all statements by parent (i.e., task-spawn, task-wait, and regular statements) so that the Write-After-Write (WAW) dependences capture the sequential ordering.

Further, nested task graphs can be easily supported with the use of multiple join/continue variables for each level of nesting. The edges in Figure 3.4 represent the happens-before relations due to task-spawn, task-wait, and sequential ordering, which are computed by CANDL dependence analyzer. As with Equation 3, the resulting dependence polyhedra are used as happens-before relations based on any task parallel constructs, after mapping task-body statements (i.e., compound statements) to regular statements. We use function inlining to handle tasks in non-recursive calls. However, handling of tasks in recursive calls is not currently supported by our approach.

3.5.3 Reflection of happens-before relations

Algorithm 2: Intersection of happens-before relations with conservative dependences.

- 1: **Input:** Conservative dependences \mathcal{P} , Happens-Before relations \mathcal{HB}
 - 2: **for** each dependence $\mathcal{P}_d^{S_i \rightarrow S_j}$ in \mathcal{P} **do**
 - 3: **for** each HB relation $\mathcal{HB}_e^{S_k \rightarrow S_l}$ in \mathcal{HB} **do**
 - 4: **if** $S_i = S_k \ \& \ S_j = S_l \ \& \ d = e$ **then**
 - 5: $\mathcal{P}'_d^{S_i \rightarrow S_j} = \mathcal{P}_d^{S_i \rightarrow S_j} \cap \mathcal{HB}_e^{S_k \rightarrow S_l};$
 - 6: **end if**
 - 7: **end for**
 - 8: Add the intersected polyhedron $\mathcal{P}'_d^{S_i \rightarrow S_j}$ to \mathcal{P}' ;
 - 9: **end for**
 - 10: **Output:** Accurate dependences after intersection \mathcal{P}'
-

After the extraction of happens-before relations from parallel constructs such as loop-level and task level constructs, it is necessary to reflect the happens-before relations onto conservative dependences as it prunes the spurious dependences from the program. Note

that HB is more conservative than \mathcal{P} in all program regions that do not contain explicit parallelism. Given conservative dependences $\mathcal{P} \ni \mathcal{P}_d^{S_i \rightarrow S_j}$ and HB relations $\mathcal{HB} \ni \mathcal{HB}_e^{S_k \rightarrow S_l}$, we define $\mathcal{P}' = \mathcal{P} \cap \mathcal{HB}$ where $\mathcal{P}_d^{S_i \rightarrow S_j} \cap \mathcal{HB}_e^{S_k \rightarrow S_l}$ is non-empty if and only if $S_i = S_k \ \& \ S_j = S_l \ \& \ d = e$. According to the definition, the happens-before relation must be transitive (like all binary relations); our approach removes dependences only for pairs of source and target instances that are not in the HB relation (including transitive dependences). Therefore, the intersection keeps any dependences between code portions that are not annotated as running in parallel.

Figure 3.5(e) shows the improved dependence information for the Jacobi kernel in Figure 3.5(a), by intersecting the conservative dependences shown in Figure 3.5(c) with happens-before relations shown in Figure 3.5(d). As shown in Figure 3.5(a), the whole for-j loops are enclosed in task constructs; the happens-before relations at depth = 3 (i.e., $\mathcal{HB}_3^{S_1 \rightarrow S_1}$ and $\mathcal{HB}_3^{S_2 \rightarrow S_2}$) are not subject to task ordering constraints and kept as the initial sequential order. Note it is also possible that some smart compilers detect parallelism in the original codes, e.g., Intel compiler could detect vector parallelism at the innermost level. Even in such cases, our approach can fully utilize explicit parallelism without missing any compiler-detected parallelism.

3.5.4 PolyAST: a loop optimizer integrating polyhedral and AST-based transformations

For the performance evaluation in Section 3.6, we used the PolyAST [22] framework to perform loop optimizations, where the dependence information provided by the proposed approach is passed as input. PolyAST employs a hybrid approach of polyhedral and AST-based compilations; it detects reduction and doacross parallelism [96] in addition to regular doall parallelism. In the code generation stage, doacross parallelism can be efficiently expressed using the proposed doacross pragmas in OpenMP 4.1 [87, 97]. These pragmas allow for fine-grained synchronization in multidimensional loop nests, using an efficient synchronization library [98].

The transformed code of Jacobi kernel (Figure 3.5(a)) based on dependence polyhedra Figure 3.5(e) is shown in Figure 3.5(b). The `ordered(2)` at line 1 specifies the nest level to place `ordered depend` directives. The `ordered depend(sink: vec)` at line 4 can be viewed as a blocking operation that waits for the completion of iteration `vec`, e.g., `(c1, c3-1)`, while the `ordered depend(source)` at line 14 can be viewed as an unblocking operation to indicate that the current iteration `(c1, c3)` has completed. Thanks to the accurate dependence information at depths 1 and 2, outermost and secondary nested loops were skewed and parallelized using doacross extensions [97, 22] while the innermost loops

were kept as the original because of the conservative dependence at depth 3. Due to space limitations, we omitted loop tiling at first and second nest levels although the permutability after skewing guarantees tiling.

3.6 Experimental Evaluation

In this section, we present the evaluation of our approach. We begin with an overview of the experimental setup and benchmark descriptions used in the evaluation. Then we discuss the experimental results and conclude with a summary.

3.6.1 Experimental setup

Table 3.1: Details of architectures used for experiments.

	Intel Xeon 5660 (Westmere)	IBM Power 8E (Power 8)
Microarch	Westmere	Power PC
Clock speed	2.80GHz	3.02GHz
Cores/socket	6	12
Total cores	12	24
L1 cache/core	32 KB	32 KB
L2 cache/core	256 KB	512 KB
L3 cache/socket	12 MB	8 MB
Compiler	gcc/g++ -4.9.2 icc/icpc -14.0	gcc/g++ -4.9.2
Compiler flags	-O3 -fast(icc)	-O3
Linux kernel	2.6.32	3.13.0

Platform: Our evaluation uses two different multi-way SMP multicore setups: an Intel Westmere and a IBM Power8 system. Table 3.1 lists their hardware specifications. On both architectures, GCC-4.9.2 is used for all benchmarks as it supports OpenMP 4.0 specifications. On the Intel Westmere, the Intel C and C++ compiler (version-14.0) is also used for evaluation of the Rodinia suite. But this compiler doesn't support OpenMP 4.0 task depend clauses and hence it is not used for evaluation of the KASTORS suite. On our IBM Power8 machine, the IBM XLC compiler is currently unavailable for the experiments. Note that our results include the -fast option for icc, but not the -Ofast option for gcc; this is not a significant issue because we do not use these results to compare icc vs. gcc performance.

Benchmarks and experimental variants: We used the KASTORS and the Rodinia suites to evaluate our approach. Benchmarks in these suites cover OpenMP loop and task

Table 3.2: Sequential execution times of KASTORS and Rodinia on Intel Westmere and IBM Power 8 systems along with problem sizes. Intel ICC-14.0 compiler doesn't support OpenMP 4.0 task depend constructs. So, no execution time is reported for KASTORS on Intel platform with ICC compiler. Transformations exposed by PoPP: Permutation (P), Fusion (F), Skewing (S), Tiling (T), Doacross pipelined parallelism (D), No further optimizations (-). Manual modifications performed before passing to PoPP: Replace complex if-statements by closures i.e., outlined functions (OF), Delinearization on task-depend variables (D), Function inlining (F), Annotated inner loop as parallel (AP), Annotated inner loop as parallel with array reductions (APR), Annotated with task-depend constructs (AT), Removal of printf statements (R), No modifications (-).

Suite	Benchmark name	Manual modifications to source	Problem Size	Sequential Exec time (Sec)			Transformations by PoPP
				Intel Westmere		IBM Power8	
				ICC	GCC	GCC	
Kastors	Jacobi	OF	Matrix size: 2K Time iterations: 200	-	4.412	4.914	F, S, T, D
	Jacobi-Blocked	D	Matrix size: 2K Time iterations: 200	-	5.838	6.241	F, S, D
	Sparse LU	D, OF	Matrix size: 100 Block size: 25	-	1.632	2.284	F, D
Rodinia	Back prop.	AP	Layer size: 5 Million	1.660	1.659	0.705	P
	CFD Solver	-	file: fvcorr.domn.097K	0.002	0.002	0.015	-
	Hotspot	AT, F	Matrix size: 8K Time iterations: 12	5.828	19.385	12.532	F, S, T, D
	Kmeans	-	Clusters: 5 Attributes: 34	2.484	4.914	7.061	-
	LUD	APR	Matrix size: 2K	7.866	8.633	30.471	P
	Needle-Wunch	AT	Matrix size: 8K	1.962	1.964	8.603	P, T, D
	Particle filter	R	Size: 10K	0.341	0.603	0.920	F
Path finder	-	Size: 100K, Time iterations: 100	0.208	0.030	0.066	-	

Figure 3.6: Evaluation of the KASTORS suite (using GCC compiler). Sequential times are reported in Table 3.2. Original benchmark speedup is compared against with optimized codes from PoPP with/ without considering happens-before (HB) relations.

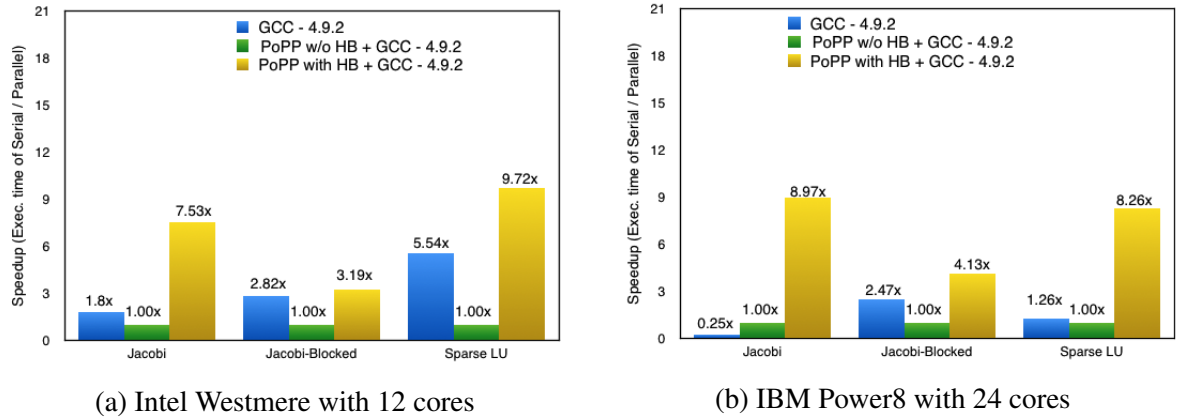
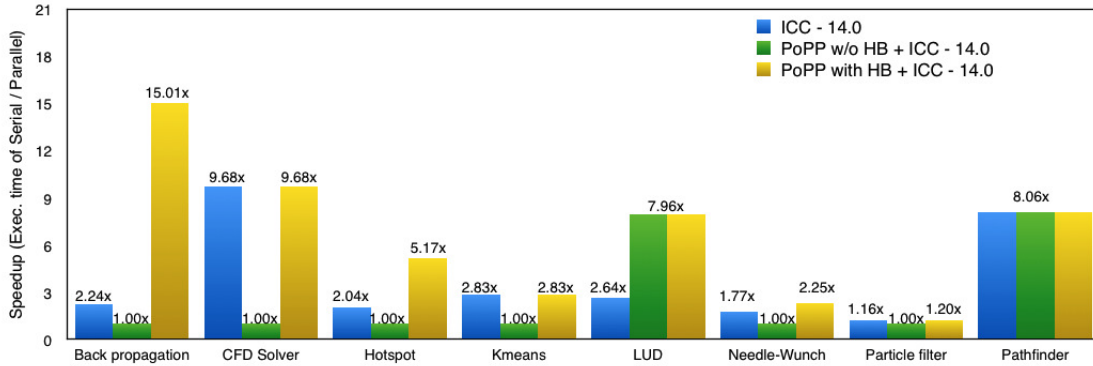


Figure 3.7: Evaluation of the Rodinia suite (using Intel compiler) on Intel Westmere with 12 cores. Sequential times are reported in Table 3.2. Original benchmark speedup is compared against with optimized codes from PoPP with/ without considering happens-before (HB) relations.



constructs. Also, these benchmarks have various data access patterns such as affine array subscripts, linearized array subscripts, indirect array subscripts, unrestricted pointer aliasing and unknown function calls. Table 3.2 summarizes problem sizes used for each benchmark. The table also includes the sequential execution times for the benchmarks while using different compilers on each platform. In all experiments, we report the mean execution time measured over 10 runs repeated in the same environment for each data point.

In the following experiments, we compare two experimental variants: OpenMP to show the original OpenMP parallel version running with all cores - i.e., 12 cores on Westmere and 24 cores on Power8 - and PoPP to show the transformed version by our framework running with all cores. The speedup of a program is defined as the execution time of the serial version of the program divided by the execution time of the parallel version of the program.

Figure 3.8: Evaluation of Rodinia suite (using GCC compiler) on Intel Westmere with 12 cores. Sequential times are reported in Table 3.2. Original benchmark speedup is compared against with optimized codes from PoPP with/ without considering happens-before (HB) relations.

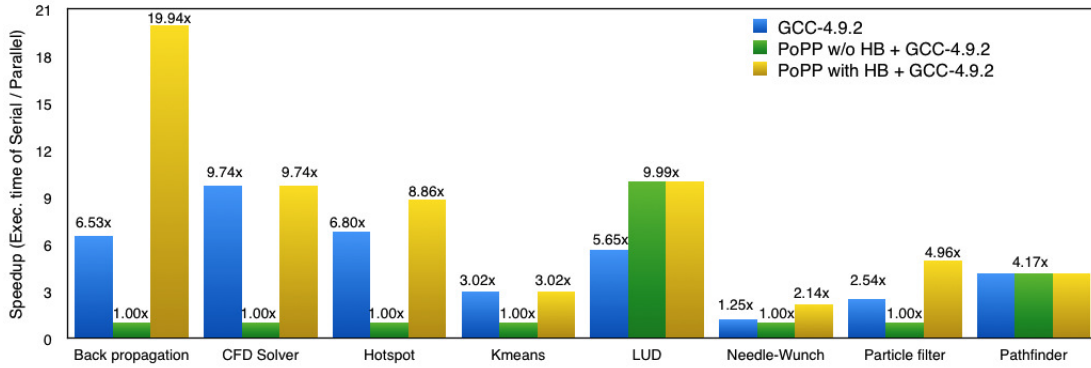
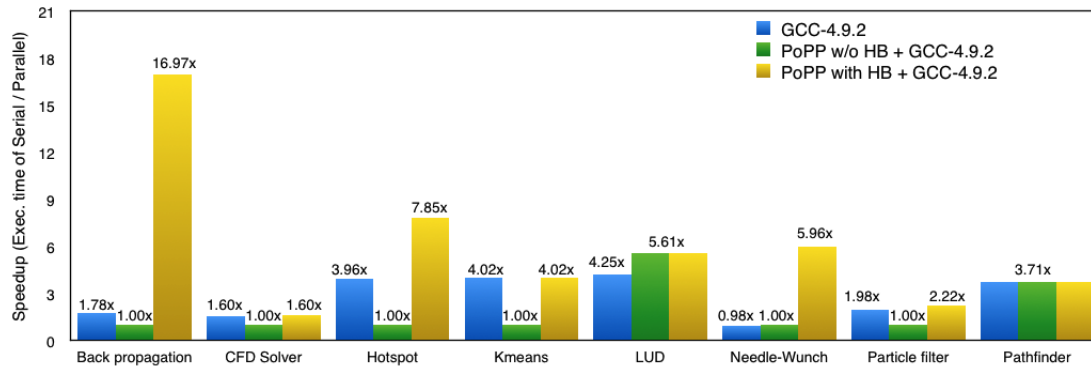


Figure 3.9: Evaluation of the Rodinia suite (using GCC compiler) on IBM Power8 with 24 cores. Sequential times are reported in Table 3.2. Original benchmark speedup is compared against with optimized codes from PoPP with/ without considering happens-before (HB) relations.



3.6.2 KASTORS Suite

KASTORS suite is designed to evaluate the efficiency of OpenMP 4.0 task dependences [83]. This suite consists of five benchmarks namely Jacobi, Jacobi-Blocked, SparseLU, Strassen and Plasma. Our implementation is currently unable to compile Strassen due to its use of recursive calls with tasks, and Plasma due to its use of C structs. As a result, we only provide results for Jacobi, Jacobi-Blocked and SparseLU from the KASTORS suite. Support for recursive task parallelism and for supporting C structs are topics for future work.

Jacobi & Jacobi-Blocked (Poisson2D): The kernel of Poisson2D is the Jacobi example discussed in Section 3.5.4 and Poisson2D - Blocked is the version where loop tiling/blocking is already applied in the OpenMP version. In both versions, the PoPP framework utilized the explicit parallelism and applied loop fusion, skewing, tiling (only to non-blocked version) and doacross parallelization. Figures 3.6(a) and 3.6(b) show that PoPP has much better performance than OpenMP for the non-blocked version because of automatic loop tiling; it also gave some improvements for the blocked version thanks to doacross parallelization.

SparseLU. This benchmark computes LU decomposition of given sparse matrix. The computation kernel is a triply nested imperfect loop nest, which contains four kinds of function calls with linearized (i.e., non-affine) array subscripts. In the OpenMP version, each function call is annotated by `task depend` constructs to implement task parallelism with inter-task dependences.

To the input kernel, we manually applied de-linearization [89] technique, which is not yet supported in the current framework. As described in Section 3.5.1, our dependence analyzer handled these function calls enclosed in non-affine `if`-statements⁴ and provided conservative dependence information. Further, the proposed approach computed exact dependence information by intersecting with the happens-before relations obtained from `task depend` constructs. The PoPP framework applied loop fusion to make a perfect loop nest and parallelized the outermost loop as doacross, as with Jacobi kernel discussed in Section 3.5.4.

Figures 3.6(a) and 3.6(b) summarize the speedup comparing with sequential execution on Westmere and Power8, which show that PoPP improved performances from 3.19 \times to 9.72 \times on Westmere and from 4.13 \times to 8.97 \times on Power8, respectively. This improvement is due to the synchronization efficiency of doacross parallelization. Although the possible dependence patterns of doacross parallelism is subset of the task constructs, the loop-based point-to-point synchronizations of doacross generally have quite small synchroniza-

⁴In preprocessing phase, `if`-statements are moved into the innermost levels so that loops are free from non-affine control flows.

tion overhead. By using our framework, programmers can specify ideal task dependences regardless of the overhead and the framework applies a sequence of transformations and converts into the efficient doacross implementations when possible.

3.6.3 Rodinia Suite

Rodinia suite is designed for heterogeneous computing and it includes kernels which target towards multi-core CPUs and GPU platforms [84]. The suite consists of 18 benchmarks and include diverse applications such as dynamic programming techniques, linear algebra kernels, graph traversals, structure grid, unstructured grid, etc. In the current evaluation, we consider eight benchmarks namely Back propagation, CFD Solver, Hotspot, Kmeans, LU decomposition, Needleman-Wunch, Particle filter, and Pathfinder. The other 10 benchmarks contain C structs, which are not yet supported in the proposed polyhedral framework. We will address these benchmarks in future work.

Back propagation. This benchmark is a machine-learning algorithm that trains the weights of connecting nodes on a layered neural network. This benchmark has two functions each of which contains an OpenMP parallelized doubly nested loops⁵, which is the source of parallelism in this benchmark. However, because of the unrestricted pointer aliasing among function arguments (2D pointer-to-pointer arrays), this loop parallelism is impossible to detect without sound inter-procedural pointer analysis. Run-time checking for the absence of aliasing without inter-procedural analysis can be expensive in this benchmark even though there exist speculative approaches [88] and compiler flags to identify pointer aliases. Alternatively, our framework utilized the happens-before relations derived from parallel loop constructs; based on the improved dependence information, our framework applied loop permutation to the kernel loops so that the resulting kernels have better spatial data locality to enhance cache reuse and vectorization. As can be seen from Figures 3.7, 3.8, and 3.9, PoPP versions show much better speedup than the OpenMP versions on both systems.

Other benchmarks. As shown in the figures, we also observed performance improvements for Hotspot, LU Decomposition, Needle-Wunch, and Particle filter, by the PoPP framework. Based on the improved dependence information via happens-before relations, the PoPP applied loop fusion and/or permutation to improve spatial data locality to these five benchmarks. Note that Rodinia benchmarks aim at parallelization on accelerators such as GPUs while the optimization criteria in PolyAST framework are customized for CPU cache architectures and thereby enabled these transformations. Also, we removed some print statements as summarized in Table 3.2, which provided more opportunities for transformations. For evaluation of Hotspot and Needle-Wunch, although the OpenMP variants

⁵For evaluation, we also specified the inner loop’s parallelism.

reported in Figures 3.7, 3.8, and 3.9 are the original loop parallel versions, we converted these loop constructs into `task depend` constructs and passed to PoPP framework. Loop skewing (only to Hotspot), tiling and doacross parallelization were applied automatically on these benchmarks by PoPP to enable better data reuse and synchronization efficiency. No transformations were applied to CFD Solver, Kmeans, and Pathfinder; same performance was observed between OpenMP and PoPP. The overall experimental results show geometric mean performance improvements of 1.62x and 2.75x on the Intel Westmere and IBM Power8 platforms respectively, relative to the original OpenMP versions.

3.7 Limitations

This section describes the limitations imposed on the current framework and briefly discusses how we can address such restrictions in the future work.

The algorithmic limitations are as follows:

- Intersection of conservative dependence with happens-before relations is applicable only in the case of programs that satisfy serial-elision property. This limitation is also due to the underlying polyhedral representations that support only sequential - i.e., total ordered - schedules. In our future work, we will address the parallel constructs that don't satisfy serial-elision property, e.g., *SPMDized code with explicit OpenMP threads and barriers*, by extending data dependence definition with happens-before relations for ordering. There is certain amount of related work in this direction [99, 100].
- In this work, we consider only nested tasks all of which are included in the same lexical scope without recursive calls. On the other hand, the generic OpenMP task parallel constructs support wider range of parallelism including arbitrary patterns of dynamic task parallelism. We plan to address recursive task patterns (e.g., observed in Strassen benchmark in KASTORS suite) by a hybrid approach of polyhedral and graph-based optimizations, which also have a long history including the Sisal language [101].
- In the current framework, we implement the detected doall and doacross parallelism by the `#pragma omp for` and `#pragma omp for ordered` constructs in the code generation phase. In our future work, we will also support task parallel constructs including inter-task dependences in the codegen phase and appropriate cost models to determine which construct of task dependence or doacross is more beneficial in

the parallelization phase. In general, `doacross` construct has less synchronization overhead while task dependence is more robust overload unbalance.

The implementation limitations are as follows:

- In this work, we limit our analysis to support only `doall` (`#pragma omp for`) parallelism and task parallelism including inter-task dependence (`task depend` constructs). We will extend our framework so that other parallel constructs in OpenMP, such as *sections* that do satisfy serial-elision property, can be expressed as happens-before relations. Once they are converted into HB relations in step 3 of Algorithm 1, the remaining steps seamlessly reflect such parallelism in the final dependence information.
- We extend our approach to handle *C structs* by encoding the fields of *structure* onto separate names and extend dependence analysis accordingly, as with access relations in ISL library [102].

3.8 Related Work

There is an extensive body of literature on applying polyhedral transformations to non-affine static program regions. We focus on past contributions that are most closely related to this work. The comparison between our approach and other related work is discussed in [26].

PENCIL [103], a platform-neutral compute intermediate language, aimed at facilitating automatic parallelization and optimization on multi-threaded SIMD hardware for domain specific languages. The language allows users to supply information about dependences and memory access patterns to enable better optimizations. PENCIL provides directives such as *independent*, *reductions* to remove data dependences on the loop, but doesn't have support for task directives as in our approach. Another key difference from our approach is that we are interested in general-purpose parallel languages such as OpenMP while PENCIL is focused on supporting DSLs in which certain coding rules are enforced related to pointer aliasing, recursion, unstructured control flow. There is a similarity in the semantics of the *independent* pragma from PENCIL and the *parallel for* pragma from OpenMP, as they both indicate no dependences among loop iterations.

Pop and Cohen have presented a preliminary approach to increase optimization opportunities for parallel programs by extracting the semantics of the parallel annotations [104]. This extracted information is brought into compiler's intermediate representation and leverage existing polyhedral frameworks for optimizations. They envisaged on considering

streaming OpenMP extensions carrying explicit dependence information, to enhance the accuracy of data dependence analyses.

A number of works addressed the problem of data-flow analysis of explicitly parallel programs, including extensions of array data-flow analysis to data-parallel and/or task-parallel programs [99], and adaptation of array data-flow analysis to the X10 programs with finish/async parallelism [100]. In these approaches, the happens-before relations are first analyzed and the data-flow is computed based on the partial order imposed by happens-before relations. On the other hand, our approach first overestimates dependences based on the sequential order and intersect the happens-before relations with the conservative dependences. The main focus of our work is on transformations of explicitly parallel programs for improved performance, whereas the work in [99] and [100] is only focused on analysis.

There has also been work done in partitioned global address space languages such as Co-Array FORTRAN (CAF) and Unified Parallel C (UPC), where certain compiler optimizations have been enabled by introducing language extensions and new synchronization constructs [105]. There has been significant effort to handle certain subsets of non-affine accesses, including delinearization techniques [89] for linearized subscripts, polynomial accesses [106] in the polyhedral model for dependence analysis and loop transformations.

Recent works in [107, 108] are inspired from our work on leveraging “serial-elision” property and happens-before relations to improve dependence analysis. For instance, Nicklas et al. [108] improved loop dependence analysis for enhancing auto-vectorization capabilities of GCC 6.1 compiler by utilizing work-sharing loop constructs in OpenMP programming model. In addition, Tao et al. in their Tapir framework [107] exploited the serial-elision property to enable classical scalar optimizations - including loop-invariant-code motion, common sub-expression elimination, and tail-recursion elimination – for Cilk/OpenMP parallel programs.

3.9 Summary

This work is motivated by the observation that software with explicit parallelism is on the rise. This explicit parallelism can be used to enable larger set of polyhedral transformations by mitigating conservative dependences, compared to what might have been possible if the input program had been sequential. We introduced an approach that reduces spurious dependences from the conservative dependence analysis by intersecting them with the happens-before relations from parallel constructs. The final set of the dependences can then be passed on to a polyhedral transformation tool, such as P_{Lu}To or PolyAST, to enable transformations of explicitly parallel programs.

We evaluated our approach using OpenMP benchmark programs from the KASTORS and the Rodinia benchmark suites. The approach reduced spurious dependences from the conservative analysis of these benchmarks and the resulting dependence information broadened the range of legal transformations in the polyhedral optimization phase. Overall, our experimental results show geometric mean performance improvements of 1.62x and 2.75x on the 12-core Intel Westmere and 24-core IBM Power8 platforms respectively, relative to the original OpenMP versions. The main focus of our future work will be to address the limitations summarized in Section 3.7.

In the next chapter (Chapter 4), we focus on eliminating dependence cycles arising from memory-based dependences such as anti-, and output-dependences to enable more freedom for optimizers to perform larger set of transformations such as enabling vectorization.

CHAPTER 4

POLYSIMD: A UNIFIED APPROACH TO VARIABLE RENAMING FOR ENHANCED VECTORIZATION

4.1 Abstract

Despite the fact that compiler technologies for automatic vectorization have been under development for over four decades, there are still considerable gaps in the capabilities of modern compilers to perform automatic vectorization for SIMD units. One such gap can be found in the handling of loops with dependence cycles that involve memory-based anti (write-after-read) and output (write-after-write) dependences. Past approaches, such as variable renaming and variable expansion, break such dependence cycles by either eliminating or repositioning the problematic memory-based dependences. However, the past work suffers from three key limitations: 1) Lack of a unified framework that synergistically integrates multiple storage transformations, 2) Lack of support for bounding the additional space required to break memory-based dependences, and 3) Lack of support for integrating these storage transformations with other code transformations (e.g., statement reordering) to enable vectorization.

In this work, we address the three limitations above by integrating both Source Variable Renaming (SoVR) and Sink Variable Renaming (SiVR) transformations into a unified formulation, and by formalizing the “cycle-breaking” problem as a minimum weighted set cover optimization problem. To the best of our knowledge, our work [28] is the first to formalize an optimal solution (reflecting best execution time) for cycle breaking that simultaneously considers both SoVR and SiVR transformations, thereby enhancing vectorization and reducing storage expansion relative to performing the transformations independently. We implemented our approach in PPCG, a state-of-the-art optimization framework for loop transformations, and evaluated it on eleven kernels from the TSVC benchmark suite. Our experimental results show a geometric mean performance improvement of $4.61\times$ on an Intel Xeon Phi (KNL) machine relative to the optimized performance obtained by Intel’s ICC v17.0 product compiler. Further, our results demonstrate a geometric mean performance improvement of $1.08\times$ and $1.14\times$ on the Intel Xeon Phi (KNL) and Nvidia Tesla V100 (Volta) platforms relative to past work that only performs the SiVR transformation [29], and of $1.57\times$ and $1.22\times$ on both platforms relative to past work on using both SiVR and SoVR transformations [30].

4.2 Introduction

There is a strong resurgence of interest in vector processing due to the significant energy efficiency benefits of using SIMD parallelism within individual CPU cores as well as in streaming multiprocessors in GPUs. These benefits increase with widening SIMD vectors, reaching vector register lengths of 512 bits in the Intel Xeon Phi Knights Landing (KNL) processor, Intel Xeon Skylake processor and 2048 bits in the scalable vector extension of the Armv8 architecture [109]. Further, there is a widespread expectation that compilers will continue to play a central role in handling the complexities of dependence analysis, code transformation and code generation necessary for vectorization for CPUs. Even in cases where the programmer identifies a loop as being vectorizable, the compiler still plays a major role in transforming the code to use SIMD instructions. This is in contrast with multicore and distributed-memory parallelism (and even with GPU parallelism in many cases), where it is generally accepted that programmers manually perform the code transformations necessary to expose parallelism, with some assistance from the runtime system but little or no help from compilers. It is therefore important to continue advancing the state of the art of vectorizing compiler technologies, so as to address the growing needs for enabling modern applications to use the full capability of SIMD units.

This work focuses on advancing the state of the art with respect to handling *memory-based anti* (write-after-read) or *output* (write-after-write) dependences in vectorizing compilers. These dependences can theoretically be eliminated by allocating new storage to accommodate the value of the first write operation thereby ensuring that the following write operation need not wait for the first write to complete. However, current state-of-the-art vectorizing compilers only perform such storage transformations in limited cases, and often fail to vectorize loops containing cycles of dependences that include memory-based dependences. This is despite a vast body of past research on storage transformations, such as variable renaming [55, 110, 111, 112] and variable expansion [113], which have shown how removing storage-related dependences can make it possible to “break” dependence cycles.

We believe that the limited use of such techniques in modern compilers is due to three key limitations that currently inhibit their practical usage:

1. Lack of a unified framework that synergistically integrates multiple storage transformations,
2. Lack of support for bounding the additional space required to break memory-based dependences, and

3. Lack of support for integrating these storage transformations with other code transformations (e.g., statement reordering) to enable vectorization.

The goal of this work is to enhance the current state-of-the-art in vectorizing compilers to enable more loops to be vectorized via systematic storage transformations (variable renaming) that remove selected memory-based dependences to break their containing cycles, while optionally using a bounded amount of additional space. We view our tool, called *PolySIMD*, as an extension to vectorization technologies that can be invoked when a state-of-the-art vectorizer fails to vectorize a loop. Thus, we do not focus on replicating all state-of-the-art vectorization capabilities in *PolySIMD*. For example, we focus on enabling vectorization of innermost loops in *PolySIMD*, though many state-of-the-art compilers support outer loop vectorization as well (and we believe that our contributions can also be applied to outer loop vectorization). By default, our tool takes sequential code as input, and focuses on identifying the best use of variable renaming to maximize opportunities for vectorization. An input loop can optionally be annotated with a pragma that specifies a bound (*spacelimit*) on the maximum amount of extra storage that can be allocated to break dependences. As discussed later, the two main variable renaming transformations that we employ in our approach are *Source Variable Renaming* (SoVR) and *Sink Variable Renaming* (SiVR).

The main technical contributions of this work are as follows:

- We formalize the problem of identifying an optimized set of SoVR and SiVR variable renaming transformations to break cycles of dependences as a minimum weighted set cover optimization problem, and demonstrate that it is practical to use ILP formulations to find optimal solutions to this problem. If the user provides an optional *space-limit* parameter, our formalization ensures that the additional storage introduced by our transformations remains within the user-provided bounds.
- We created a new tool, *PolySIMD*, to implement our approach by selecting and performing an optimal set of SoVR and SiVR transformations, along with supporting statement reordering transformations. Given an input sequential loop, *PolySIMD* either generates transformed sequential CPU code that can be input into a vectorizing compiler like ICC or generates GPU code (CUDA kernels) that can be processed by a GPU compiler like NVCC. *PolySIMD* is implemented as an extension to the PPCG framework [114, 115], so as to leverage PPCG’s dependence analysis and code generation capabilities.
- We evaluated our approach on eleven kernels from the TSVC benchmark suite [116], and obtained a geometric-mean performance improvement of $4.61\times$ on an Intel Xeon

Phi (KNL) machine relative to the optimized performance obtained by Intel’s ICC v17.0 product compiler.

- We also compared our approach with the two most closely related algorithms from past work, one by Calland et al. [29] that only performed SiVR transformations, and the other by Chu et al. [30]. that proposed a (not necessarily optimal) heuristic to combine SiVR and SoVR transformations.

Relative to Calland et al’s approach, our approach delivered an overall geometric-mean performance improvement of 1.08× and 1.14× on the Intel KNL and Nvidia Volta platforms respectively, though our approach selected exactly the same (SiVR-only) transformations for six of the eleven benchmarks. Relative to Chu et al’s approach, our approach delivered an overall geometric-mean performance improvement of 1.57× and 1.22× on the Intel KNL and Nvidia Volta platforms respectively.

Original program having cycles	Applying SoVR(s2, a[i+1]) on the original program	Applying SiVR(s1, a[i]) on the original program
<pre>for i = 1 to N { a[i] = b[i]+c[i]; //s1 a[i+1] = a[i-1]+2*a[i+1]; //s2 }</pre>	<pre>for i = 1 to N { a[i] = b[i]+c[i]; //s1 float k = a[i+1]; //s21 a[i+1] = a[i-1]+2*k; //s2 }</pre>	<pre>float a_temp[N]; for i = 1 to N { a_temp[i] = b[i]+c[i]; //s11 a[i] = a_temp[i]; //s1 a[i+1] = (i > 1) ? \ //s2 (a_temp[i-1] : a[i-1])+2*a[i+1] }</pre>
<p>Dependence graph of the original program</p>	<p>Dependence graph after applying SoVR(s2, a[i+1]) on the original program</p>	<p>Dependence graph after applying SiVR(s1, a[i]) on the original program</p>

Figure 4.1: An example to illustrate SoVR and SiVR transformations.

4.3 Discussion on Variable Renaming Transformations

In this section, we discuss on two variable renaming transformations that are considered in this work, and they are *Source variable renaming* (SoVR)¹ introduced by Kuck et al. in [55] and *Sink variable renaming* (SiVR) introduced by Chu et al. in [111]. Furthermore, these two renaming transformations were formalized by Calland et al. in [29], and

¹SoVR was also referred as node splitting by Kuck et al. in [55].

referred SoVR and SiVR as T1 and T2 transformations respectively. Note that these transformations reposition memory-based dependences to break cycles but do not eliminate the dependences unlike variable expansion techniques [113] and Array SSA [112].

4.3.1 Source Variable Renaming (SoVR)

Source variable renaming transformation is introduced to handle anti-dependences in cycles of memory-based dependences, and the transformation is applied on a read access of a statement to reposition an outgoing anti-dependence edge from the read access [55]. Applying SoVR on a read access (say r) of a statement introduces a new assignment statement that copies the value of r into a temporary variable (say k), and then the original statement's read access is replaced with k . Since the transformation is renaming source (read access) of an anti-dependence, we call this transformation as a source variable renaming transformation.

Example. Applying SoVR on the read access $a[i+1]$ of the statement $s2$ in the original program (shown in Figure 4.1) introduces a new assignment statement $s21$ copying the value of $a[i+1]$ into a temporary variable k , and then the statement $s2$ refers to k in-place of $a[i+1]$. As a result, the source of the anti-dependence from the read access $a[i+1]$ is repositioned to $s21$. This reposition helps in breaking one of the cycles through $s2$, i.e., the cycle involving a flow-dependence from $a[i]$ of $s1$ to $a[i-1]$ of $s2$, and an anti-dependence from $a[i+1]$ of $s2$ to $a[i]$ of $s1$.

Usefulness. Since SoVR transformation is applied on a read access of a statement, the transformation can modify only incoming flow- and outgoing anti-dependences related to that read access. Hence, applying a SoVR transformation on a statement is useful in breaking cycles if the statement has an incoming anti- or output-dependences and an outgoing anti-dependence [29]. Also, SoVR transformation can be useful if the statement's incoming flow-dependence and outgoing anti-dependence are on different accesses.

Space requirements & Additional memory traffic. The temporary variable introduced as part of a SoVR transformation is private to a loop carrying an anti-dependence that we are interested in repositioning. Hence, SoVR requires an additional space equivalent to the length of vector registers (i.e., VLEN) of target hardware. Furthermore, the transformation additionally introduces only one scalar load and one scalar store per every iteration of the target loop.

4.3.2 Sink Variable Renaming (SiVR)

Sink variable renaming transformation is introduced to handle both anti- and output-dependences in cycles of memory-based dependences [111]. The transformation is applied on a write access of a statement to reposition an outgoing flow-dependence from the write access and also an outgoing anti-dependence from the statement. Applying SiVR on a write access (say w) of a statement s introduces a new assignment statement that evaluates the right-hand side of the statement into a temporary array (say temp), and then any references to the value of w are replaced by accessing the temp . Since SiVR transformation is applied on a write access of a statement, the transformation can modify only incoming anti- or output-dependences related to that write access. As a result, applying SiVR transformation is useful in breaking cycles if the statement has either an incoming anti- or output-dependences and either an outgoing flow- or anti-dependences [29]. Since the transformation is renaming the sink (the write access) of an incoming anti- or output-dependence, this transformation is called as sink variable renaming transformation [111].

Example. Applying SiVR on the write access $a[i]$ of the statement $s1$ in the original program (shown in Figure 4.1) introduces a new assignment statement $s11$ that evaluates the rhs of $s1$ into a temporary array a_temp , and then the transformation replaces the references to $a[i]$ (such as $a[i-1]$) with the a_temp . As a result, the source of the flow-dependence from the write access $a[i]$ is repositioned to $s11$. This repositioning helps in breaking all of the cycles present in the original program including the one that is not broken by the previous SoVR transformation, i.e., the cycle involving a flow-dependence from $a[i]$ of $s1$ to $a[i-1]$ of $s2$, and an output-dependence from $a[i+1]$ of $s2$ to $a[i]$ of $s1$.

Usefulness. Applying a SiVR transformation is useful in breaking cycles if the statement has either an incoming anti- or output-dependences, and either an outgoing flow- or anti-dependences [29].

Space requirements. The temporary array introduced as part of a SiVR transformation is not private to a loop unlike SoVR transformation, because references to the newly allocated storage can be across iterations. Hence, SiVR requires an additional space equivalent to the number of iterations of a loop. However, the additional storage can be reduced by strip mining the loop, and vectorizing only the strip [117]; whose space requirement is now proportional to the strip size, and the strip can be as minimal as vector length.

Additional memory traffic. SiVR transformation introduces pointer-based loads and stores, unlike the SoVR transformation which introduces only scalar loads and stores. The new assignment statement as part of a SiVR transformation introduces one additional pointer-based store and one pointer-based load per one iteration of the loop. Along with

a new assignment statement, each reference to the newly allocated storage introduces one additional pointer-based load, leading to overall $(1+\#\text{references})$ of pointer-based loads per one iteration of the loop. In this work, we focus on applying renaming transformations for vectorizing only inner-most loops, and this focus helps in conservatively counting the references to the newly allocated storage by traversing the loop body and ignoring conditionals.

4.3.3 Synergy between SoVR and SiVR

In general, SoVR transformation is neater in code generation and performs more efficiently than SiVR since the SoVR transformation introduces scalar loads and stores. But, SoVR transformation has limited applicability (i.e., handling only anti-dependences) in breaking cycles compared to SiVR, which has broader applicability through breaking output-dependences. Furthermore, the SiVR transformation can reposition several distinct anti-dependences if all of those anti-dependences share a common sink variable, unlike SoVR transformation which needs to be applied on each anti-dependence edge. Table 4.1 shows a comparison between SoVR and SiVR transformations related to space requirements and additional stores and loads.

Table 4.1: A comparison between SoVR and SiVR transformations related to the space requirements and additional stores, loads introduced by these transformations in one iteration of the target loop. * – Additional scalar loads/stores for SiVR transformation may go negative in case of renaming scalars.

		SoVR	SiVR
Storage	#Additional space	Vector length	Loop length
Additional loads & stores	#scalar loads	1	0*
	#scalar stores	1	0*
	#pointer-based loads	0	1+#references
	#pointer-based stores	0	1

4.4 Motivating Example

To motivate the need of a unified framework that synergistically integrates multiple variable renaming transformations, we consider a running example (shown in Figure 4.2) from [29] whose dependence graph consists of three cycles (i.e., $s1-s3-s2-s4-s1$, $s1-s3-s4-s1$, and $s1-s2-s4-s1$) which prohibit vectorization. Past work by Calland et al. [29] uses only SiVR transformations to eliminate all of the above three cycles by applying SiVR transformations on the statements $s2$ and $s3$. But these transformations require an additional space close to

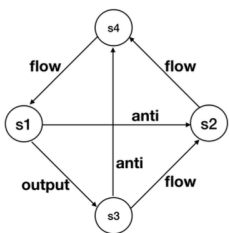
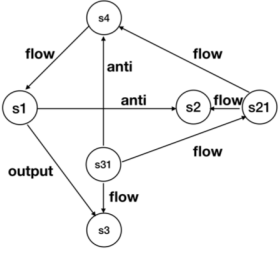
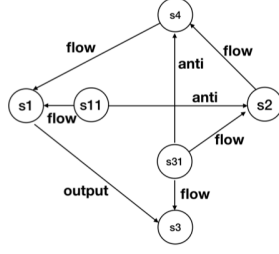
Original program from [5] having cycles	Past approach by Calland et al. [5] on the original program	Our approach on the original program
<pre>float a[N], b[N], c[N]; for i = 4 to T { a[i+5]=c[i-3]+b[2i+2]; //s1 b[2i] = a[i-1] + 1; //s2 a[i] = c[i+5] - 1; //s3 c[i] = b[2i-4]; //s4 }</pre>	<pre>float a[N], b[N], c[N]; float a_temp[T], b_temp[T]; for i = 4 to T { a[i+5] = c[i-3]+b[2i+2]; //s1 b_temp[i] = (i >= 5) ? \ a_temp[i-1]:a[i-1]+1; //s21 b[2i] = b_temp[i]; //s2 a_temp[i]=c[i+5]-1; //s31 a[i] = a_temp[i]; //s3 c[i] = (i >= 6) ? \ b_temp[i-2]:b[2i-4]; //s4 }</pre>	<pre>float a[N], b[N], c[N]; float a_temp[T]; for i = 4 to T { float k = b[2i+2]; //s11 a[i+5] = c[i-3] + k; //s1 b[2i] = (i >= 5) ? \ a_temp[i-1]:a[i-1]+1; //s2 a_temp[i]=c[i+5]-1; //s31 a[i] = a_temp[i]; //s3 c[i] = b[2i-4]; //s4 }</pre>
<p>Dependence graph of the original program</p> 	<p>Dependence graph after applying the past approach by Calland et al. [5] on the original program SiVR(s2, b[2i]), SiVR(s3, a[i])</p> 	<p>Dependence graph after applying our approach on the original program SoVR(s1, b[2i+2]), SiVR(s3, a[i])</p> 

Figure 4.2: A running example from [29] whose dependence graph consists of three cycles $c1/c2/c3$: $s1-s3-s2-s4-s1/s1-s3-s4-s1/s1-s2-s4-s1$ which prohibit vectorization. The table also lists dependence graphs and transformed codes after applying past approach [29] and our integrated approach on the original program.

2 times the number of iterations of the loop- i , i.e., a total of $(2 \times T)$, and also introduce additional 2 pointer-based stores and 4 pointer-based loads per one iteration of the loop.

However, instead of applying SiVR transformation on the statement $s2$ to break the cycle ($c3$), SoVR transformation can be applied on the $s1$ to break the same cycle ($c3$). This results in lesser additional space $(T + VLEN)$, and also introduces lesser additional 1 pointer-based store and 2 pointer-based loads per one iteration of the loop. Our approach identifies such optimal transformations from a set of valid SoVR and SiVR transformations by formalizing the “cycle-breaking” problem as a minimum weighted set cover optimization problem with a goal of reducing overhead arising from additional loads and stores introduced by these transformations. The speedup’s after applying our approach over the original program is $5.06\times$ and $4.02\times$ compared to the original program and the transformed program after applying the Calland et al. approach [29] respectively on the Intel Knights Landing processor (More details about the architectures and compiler options can be found in Table 4.3).

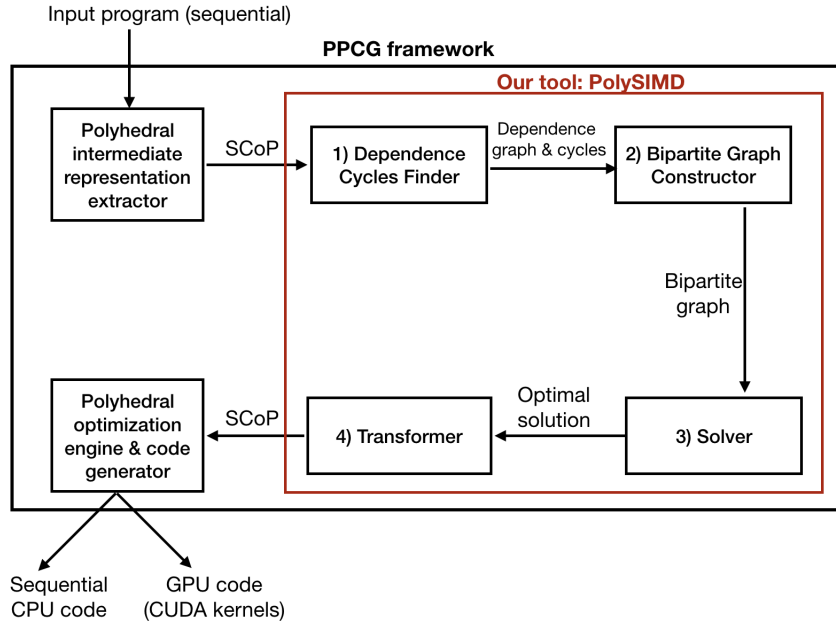


Figure 4.3: Workflow of *PolySIMD* implemented as an extension to the PPCG [114].

4.5 Our Unified Approach to Variable Renaming

In this section, we introduce our approach that synergistically integrates SoVR and SiVR transformations into a unified formulation to break cycles of dependences involving memory-based dependences, and the approach is implemented in a tool called *PolySIMD*.

The overall approach is summarized in Figure 4.3, which is implemented as an extension to the PPCG framework [114] (a state-of-the-art optimization framework for loop transformations), and consists of the following components: 1) Dependence cycles finder (Extracting flow-, anti-, and output-dependences on a target loop, then constructing a dependence graph, and then finding cycles in the graph using the *Johnson’s algorithm* [118]), 2) Bipartite graph constructor (Building a bipartite mapping from a union over useful SoVR and SiVR transformations to the breakable cycles, in such a way that there is an edge between them if the transformation can break the cycle), 3) Solver (Reducing the problem of breaking cycles as a weighted set covering optimization problem and finding an optimal solution using the ILP solver of ISL framework [119]), 4) Transformer (Applying SoVR and SiVR transformation from the optimal solution to break cycles).

4.5.1 Dependence Cycles Finder

This component takes the polyhedral intermediate representation (also referred to as SCoP) extracted from a target loop as an input. Then, the loop-carried and loop-independent flow-,

Table 4.2: Bipartite graph constructed on the dependence graph of the original program in Figure 4.2.

Transformations (T)	Cycles (C)
t1 = SoVR(s1, b[2i+2])	c3
t2 = SiVR(s2, b[2i])	c3
t3 = SiVR(s3, a[i])	c1, c2
t4 = SoVR(s3, c[i+5])	c2
t5 = SiVR(s4, c[i])	c2

anti-, and output-dependences (including both data and control dependences) of the target loop are computed using the PPCG dependence analyzer. Afterwards, these dependences are represented as a directed graph, where a node denotes a statement, and an edge denotes a dependence between two statements. Also, each edge of a directed graph is annotated with a dependence type: flow-, anti-, or output-. Now, *PolySIMD* computes all strongly connected components (SCC's) of the directed graph using the *Tarjan's algorithm* [120]. Then, all elementary cycles² for every SCC of the dependence graph are identified using the *Johnson's algorithm* [118], an efficient algorithm to enumerate all elementary cycles of a directed graph. The worst case time complexity of the algorithm is $O((n + e)(c + 1))$ where n is the number of vertices, e is the number of edges and c is the number of distinct elementary cycles in a directed graph. For example, applying *Johnson's algorithm* on a dependence graph of the running example (shown in Figure 4.2 and has only one SCC) results in three elementary cycles $c1/c2/c3$: $s1-s3-s2-s4-s1/s1-s3-s4-s1/s1-s2-s4-s1$ on the `loop-i`.

Note that SoVR and SiVR transformation cannot break a cycle if the cycle is either a pure flow- or pure output-dependence cycle [29]. Since our approach considers only SoVR and SiVR into the formulation, if *PolySIMD* encounters any dependence cycle involving pure flow- or pure output-dependences in a SCC, then the tool ignores the SCC and continues with the rest of SCC's. If each SCC have either a pure flow- or pure output-dependence cycle, then *PolySIMD* will skip rest of steps in our approach, otherwise the tool continues with next steps. Since the three cycles $c1$, $c2$, and $c3$ of the running example are neither pure-flow nor pure-output dependence cycles, our approach proceeds to the next step.

4.5.2 Bipartite Graph Constructor

This component constructs a bipartite graph between a union of useful SoVR and SiVR transformations (see Section 4.3 for usefulness criteria) and breakable cycles of the dependence graph such that there is an edge between them if applying the transformation can break the cycle. As from the usefulness criteria, Table 4.2 shows a tabular version of the bipartite graph constructed for the running example.

4.5.3 Solver

After constructing the bipartite graph, the problem of finding an optimal set of transformations for cycle breaking is reduced to a minimum weighted set cover optimization problem (C, T, W) where C refers to a collection of cycles, T refers to a set of useful SoVR and SiVR transformations, and W refers to a set of weights for each transformation. The goal of the optimization problem is to identify the minimum weighted sub-collection of T whose union covers all cycles in C , and the optimization problem is known to be NP-hard. Hence, we formulate the minimum weighted set covering problem as the following integer linear programming (ILP) in our tool-chain.

Variables:

- A variable t_i for each transformation of T

$$t_i \in \{0, 1\}, \forall t_i \in T$$

where $t_i = 1$ indicates that the transformation t_i should be applied on the original program, otherwise it should be ignored.

- A weight parameter w_i for each transformation t_i to indicate an additional execution overhead (ignoring cache effects), and is measured using the additional loads and stores introduced by the transformation per one iteration of the target loop (See Table 4.1 for more details).
- A *latencyratio* parameter to indicate the ratio of access times of main memory to registers, and this parameter is used in converting weight parameters of SiVR transformations (introduced pointer-based loads/stores) into same units as of weight parameters of SoVR transformations (introduces scalar-based loads/stores).

²An elementary cycle of a directed graph is a path in which no vertex appears twice except the first and last vertices. Since elementary cycles form a basis for enumerating all cycles in a directed graph, breaking all of them results in an acyclic graph.

Acyclicity constraint: The acyclic constraint on the dependence graph is modeled into a condition that each cycle of C should be covered by at-least one transformation of T .

$$\forall c_j \text{ in } C, \left(\sum_{\substack{\forall t_i \text{ in } T \\ \text{such that } t_i \text{ can break } c_j}} t_i \right) \geq 1$$

Objective function: Our approach targets at minimizing additional overhead introduced by the optimal set of transformations.

$$\text{Minimize} \left(\sum_{\forall t_i \text{ in } T} w_i \times t_i \right)$$

The ILP formulation for the example is as follows (Assuming *latencyratio* as 50).

$$T = \{t1, t2, t3, t4, t5\}, \quad C = \{c1, c2, c3\}, \quad t_i \in \{0, 1\}, \quad \forall t_i \in T,$$

$$w1 = w4 = 2, \quad w2 = w3 = w5 = 50 \times 3 = 150,$$

$$t3 \geq 1, \quad t3 + t4 + t5 \geq 1, \quad t1 + t2 \geq 1,$$

$$\text{Minimize} \left(2 \times (t1 + t4) + 150 \times (t2 + t3 + t5) \right)$$

The optimal solution obtained for the above formulation is ($t1=1, t2=0, t3=1, t4=0,$ and $t5=0$), i.e., applying SoVR on $s2$ and SiVR on $s3$ can break all cycles present in the running example with minimal additional overhead introduced. Note that the above solution is different to the solution ($t2 = 1, t3 = 1$) from the Calland et al's approach in [29] since our approach considers both SoVR and SiVR transformations into the formulation, unlike the Calland et al's approach which includes only SiVR transformations.

Heuristics. There can be simple heuristics such as applying SoVR transformation in the beginning to break as many cycles it can and followed by applying SiVR transformation to break rest of cycles, which can lead to the similar performance improvements compared to our approach. The solution from such heuristics may include redundant SoVR transformations, which can be observed on the running example. Applying transformation $t4$ (SoVR) on the running example (ahead of SiVR transformations) to break the cycle $c2$ is redundant because the transformation $t3$ (SiVR) will eventually break the cycle $c2$ and also can break cycle $c1$ that cannot be broken by any SoVR transformation.

There can exist other heuristics or greedy algorithms to the minimum weighted set

cover optimization problem. But, we believe that an ILP formulation formalizes the optimization problem without being tied to specific heuristics, which in turn reduces performance anomalies that can occur in optimization heuristics; Also, the compile-times for the results in this experimental evaluation are less than half a second (see Table 4.4 for more details). We also believe that our framework can be easily extended to include other heuristics or greedy algorithms to the optimization problem.

4.5.4 Transformer

This component applies the optimal set of transformations obtained from the solver onto the intermediate polyhedral representation of the target loop. It is also mentioned in [29] that the order of applying SoVR and SiVR transformations doesn't have any effect on the final program. Hence, *PolySIMD* first applies SoVR transformations from the optimal solution, and then followed by SiVR transformations from rest of the optimal solution. The generation of new assignment statements, modifying schedules of statements, and updating the references as part of the code transformations are implemented using the dependence analyzer and schedule trees of the PPCG framework.

After applying all transformations from the optimal solution, *PolySIMD* feeds the transformed intermediate polyhedral representation to the PPCG optimization engine to perform statement reordering based on the topological sorting of the transformed dependence graph. Note that all of the benchmarks in the experimental evaluation required statement reordering transformation to be performed without which the Intel's ICC v17.0 product compiler couldn't vectorize. This demonstrates the necessity of coupling storage optimizations with the loop optimization framework. Finally, *PolySIMD* leverages code generation capabilities of the PPCG framework to generate transformed sequential CPU code that can be input into a vectorizing compiler like ICC or generates GPU code (CUDA kernels) that can be processed by a GPU compiler like NVCC.

4.5.5 Bounding Additional Space

We believe that one of the major key limitations in the unavailability of variable renaming techniques (especially on arrays) in modern compilers is due to the lack of support for bounding the additional space required to break memory-based dependences. Hence, we provide a clause (i.e., *spacelimit*) to the directive “`#pragma vectorize`” that can help programmers to limit the additional space to enable enhanced vectorization of inner-most loops, and the *spacelimit* is expressed in multiples of vector registers length. The clause *spacelimit* essentially helps our approach to compute strip size that can be vectorized, and

the formula to compute the strip size (in multiples of vector length) is as follows.

$$\text{strip size} = \left\lfloor \frac{\text{spacelimit} \times VLEN - |T_{SoVR}| \times VLEN}{|T_{SiVR}| \times VLEN} \right\rfloor = \left\lfloor \frac{\text{spacelimit} - |T_{SoVR}|}{|T_{SiVR}|} \right\rfloor$$

where $|T_{SoVR}|$ and $|T_{SiVR}|$ refer to number of SoVR and SiVR transformations in the optimal solution respectively. If the strip size value is non-positive for a given *spacelimit*, then our approach ignores applying renaming transformations. Otherwise, our approach does strip mining of the target loop before applying any of the renaming transformations from the optimal solution.

4.6 Performance Evaluation

In this section, we present an evaluation of our *PolySIMD* tool relative to Intel’s ICC v17.0 product compiler and to the two algorithms presented in past work [29, 30] for performing SiVR and SoVR transformations to break cycles of a dependence graph. We begin with an overview of the experimental setup and the benchmark suite used in our evaluation, and then present experimental results for the three different comparisons.

4.6.1 Experimental Platforms

Our evaluation uses the following two SIMD architectures. 1) A many-core Intel Xeon Phi Knights Landing (KNL) processor with two 512-bit vector processing units (VPU) per core. Thus, each 512-bit VPU can perform SIMD operations on 16 single-precision floating point values, i.e., the VPU has an effective vector length of 16 (for 32-bit operands). Since we are evaluating vectorization for single-threaded benchmarks, we only use one core of the KNL processor in our evaluation, though our approach can be applied to multithreaded applications as well. 2) A Nvidia Volta accelerator (Tesla V100) with 80 symmetric multi-processors (SMs), each of which can multiplex one or more thread blocks. A thread block can contain a maximum of 1024 threads, which are decomposed into 32-thread warps for execution on the SM. Thus, each SM can be viewed as being analogous to a VPU with an effective vector length of 32 (for 32-bit operands). For consistency with our KNL results, we only generate one block of 1024 threads per benchmark, thereby only using one SM in the GPU. However, our approach can be applied to multi-SM executions as well. Table 4.3 lists the system specifications and the compiler options used in our evaluations. The comparison with ICC could only be performed on KNL, since ICC does not generate code for Nvidia GPUs. The comparison with the two algorithms from past work [29, 30] were performed on both platforms.

Table 4.3: Summary of SIMD architectures and compiler flags used in our experiments. SP refers to Single Precision floating point operands, VPU refers to a KNL Vector Processing Unit, and SM refers to a GPU Streaming Multiprocessor.

	Intel Xeon Phi	Nvidia Volta
Microarch	Knights Landing	Tesla V100
SIMD lanes	16 SP per VPU (2 VPU's per core)	32 SP per SM
Compiler	Intel ICC v17.0	Nvidia NVCC v9.1
Compiler flags	-O3 -xmic-avx512	-O3 -arch=sm_70 -cbin=icc

4.6.2 Benchmarks

We use the Test Suite for Vectorizing Compilers (TSVC) benchmark suite in our evaluation, originally developed in FORTRAN to assess the vectorization capabilities of compilers [121]. Later, the benchmark suite was translated into C with additional benchmarks to address limitations in the original suite [116], so we used this C version for our evaluations. A detailed study of these benchmarks, along with the vectorization capabilities of multiple compilers can be found in [116, 122]. Since our goal is to evaluate the effectiveness of renaming variables on breaking dependence cycles that inhibit vectorization, we restrict our attention to TSVC benchmarks that contain multi-statement dependence cycles containing at least one anti/output dependence and that cannot be broken by scalar privatization. Further, since *PolySIMD* is based on a polyhedral optimization framework, we further restricted our attention to the subset of these benchmarks that do not contain non-affine expressions that prevent polyhedral analysis³. This selection resulted in 11 benchmarks from the TSVC suite that will be the focus of our evaluation, and are summarized in Table 4.4.

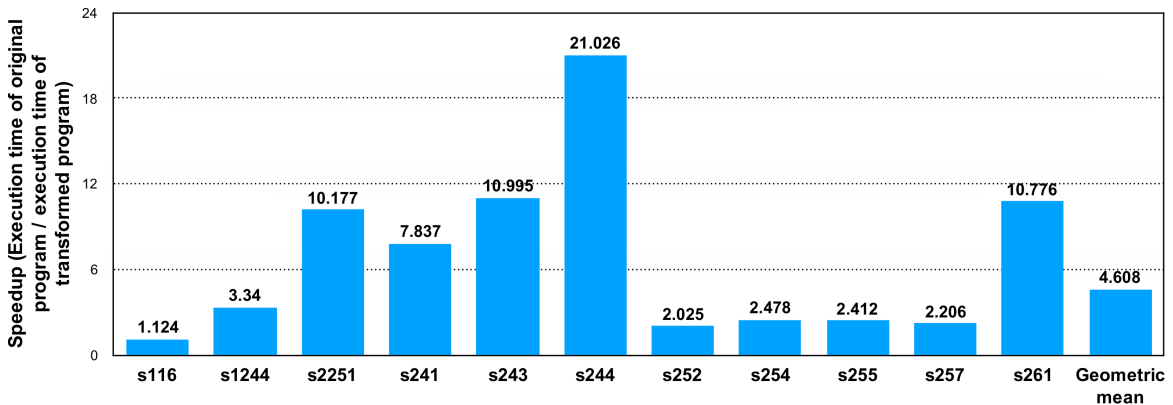


Figure 4.4: Speedups using *PolySIMD* on the eleven benchmarks from the TSVC suite, compiled using the Intel’s ICC v17.0 product compiler and running on a single core of Intel Knights Landing processor.

³This constraint arises from the implementation of our algorithm in *PolySIMD*; our algorithm can be applied in a non-polyhedral compiler setting as well.

Table 4.4: Summary of the 11 benchmarks from the TSVC suite used in our evaluation, including the number of statements, number of dependences, and number of elementary cycles per benchmark (excluding self-loop cycles). The benchmarks were executed using $N = 2^{25}$ and $T = 200$ as input parameters. Number of SiVR and SoVR transformations performed by *PolySIMD* for the 11 benchmarks, and also the overall compilation times required. Coincidentally, none of these benchmarks triggered a case in which both SiVR and SoVR transformations had to be performed.

Benchmark	#Stmts	#Deps	#Elementary cycles	Our ILP Solution		Compilation time (sec)	
				#SoVR's	#SiVR's	PolySIMD	Total
s116	5	5	1	1	0	0.08	0.10
s1244	2	2	1	1	0	0.01	0.02
s241	2	3	1	1	0	0.01	0.03
s243	3	6	2	1	0	0.02	0.04
s244	3	4	1	1	0	0.02	0.03
s2251	3	4	1	0	1	0.02	0.03
s252	3	5	2	0	2	0.02	0.04
s254	2	2	1	0	1	0.01	0.02
s255	3	6	3	0	2	0.02	0.04
s257	2	3	1	0	1	0.02	0.04
s261	4	9	3	0	2	0.02	0.04

4.6.3 Comparison with ICC

As discussed in Figure 4.3, *PolySIMD* takes a sequential program as input, and generates sequential code as output with selected variable renamings and statement reorderings that enable enhanced vectorization. Figure 4.4 shows the speedups obtained by using *PolySIMD* as a preprocessor to Intel's ICC v17.0 product compiler on the KNL platform. The speedup represents the ratio of the execution time of the original program compiled with ICC to the execution time of the transformed program compiled with ICC, using the compiler options in Table 4.3 in both cases. As can be seen in Figure 4.4, the use of *PolySIMD* as a preprocessor results in significant performance improvements for the 11 kernels. The transformations performed by *PolySIMD* are summarized in Table 4.4; the fact that no benchmark required both SiVR and SoVR transformations is a pure coincidence. We now discuss the two groups of benchmarks for which *PolySIMD* applied the SoVR and SiVR transformations respectively.

Source Variable Renaming (SoVR): The benchmarks s116, s1244, s241, s243, s244 in the first five entries of Table 4.4 contain multi-statement recurrences involving outgoing anti-dependences. Hence, *PolySIMD* applied the SoVR transformation on these benchmarks to reposition these outgoing anti-dependence edges to break the cycles, as dictated by the column titled *SoVR* under *ILP solution* of Table 4.4. There are a few interesting observations that can be made from the results in Table 4.4 for these five benchmarks:

- The SoVR transformation enabled vectorization for all five benchmarks (as con-

firmed by the compiler log output), and resulted in speedups varying from 1.12× to 21.02× on Intel KNL relative to the original program using the Intel’s ICC v17.0 product compiler.

- The s1244 benchmark involves dead-write statements (i.e., there are no reads of a write before another statement writing to the same location) whose removal eliminate dependence cycles. Currently, *PolySIMD* doesn’t check for dead-write statements unlike the Intel compiler (with O3 optimization flag enabled) which remove the dead writes to enable the vectorization. As a result, there is a lower speedup with our approach compared to the Intel compiler.
- The reason for less speedup in case of the s116 benchmark is the generation of non-unit (unaligned) strided loads and stores leading to inefficient vectorization (as confirmed by the compiler log output describing the estimated potential speedup as 1.36×).
- All these five benchmarks required statement reordering to be performed after the SoVR transformations, without which the Intel’s compiler wasn’t able to vectorize. This indicates the necessity of loop transformations framework to output the final code that can be vectorizable by the existing compilers.

Sink Variable Renaming (SiVR): The column titled *SiVR* under *ILP solution* indicates that the SiVR transformation should be performed on the remaining benchmarks (s2251, s252, s254, s255, s257, s261) in Table 4.4. These benchmarks have dependence cycles involving anti- and output-dependences, and hence our approach chose only the SiVR transformations to break these dependence cycles. As with the earlier five benchmarks, there are a few interesting observations that can be made from the results in Table 4.4 for these later three benchmarks:

- The SiVR transformation enabled vectorization for all the remaining six benchmarks, and resulted in speedups varying from 2.02× to 10.77× on the Intel KNL platform relative to the original program. The compiler log output shows that vectorization was indeed performed in all cases.
- The benchmarks s252, s254, s255, s257 have loop-carried flow-dependence and loop-independent anti-dependences on scalars, and resolving these dependences on scalars using our approach introduced higher overhead from temporary arrays pointer-based loads and stores. As a result, the performance improvements in these benchmarks are relatively low.

- As seen with earlier five benchmarks benefited by the SoVR transformation along with the statement reordering, these six benchmarks also required statement reordering to be performed after the SiVR transformations, without which the Intel’s compiler wasn’t able to vectorize.

4.6.4 Comparison with Calland et al’s approach

The heuristics proposed by Calland et al. [29] aim to find the minimum number of SiVR transformations to break all dependence cycles involving memory-based dependences. As a result, the heuristics choose only SiVR transformations for vectorizing all the eleven benchmarks. However, our approach chooses to perform SoVR transformations on five of the eleven benchmarks (s116, s1244, s241, s243, s244), since SoVR incurs less overhead than SiVR. Hence, we observe speedups (shown in Table 4.5) with our approach relative to Calland’s approach, varying from 1.07× to 1.24× on the Intel KNL platform and 1.12× to 1.57× on the NVIDIA Volta. For the remaining six benchmarks, our approach chose exactly the same set of SiVR transformations as did their approach, and hence there is no performance improvement in these cases. The overall geometric-mean speedups on all of the eleven benchmarks are 1.08× and 1.14× relative to their approach on the KNL and Volta platforms.

Table 4.5: Speedups on the Intel KNL processor and NVIDIA Volta accelerator using *PolySIMD* on seven benchmarks from the eleven benchmarks relative to past approaches, i.e., Calland et al. [29] and Chu et al. [30]. We excluded the remaining four benchmarks from the table since our results were similar to both of the past works.

Bench -mark	Intel KNL		NVIDIA Volta	
	Calland et al. approach	Chu et al. approach	Calland et al . approach	Chu et al. approach
s116	1.20x	1.03x	1.29x	1.27x
s1244	1.10x	4.03x	1.57x	1.51x
s241	1.07x	1.49x	1.31x	1.70x
s243	1.27x	1.59x	1.47x	1.61x
s244	1.24x	1.22x	1.12x	1.32x
s257	1.00x	9.74x	1.00x	1.08x
s261	1.00x	1.20x	1.00x	1.19x

4.6.5 Comparison with Chu et al’s approach

Chu et al. proposed an algorithm for resolving general multi-statement recurrences which considers both SoVR and SiVR transformation [30]. The solution obtained by their algorithm depends on a traversal of the dependence graph, and may not be optimal in gen-

eral. Further, their algorithm may include redundant SiVR transformations, which were observed when applying their algorithm to benchmarks s241, s243, s257 and 261, leading to lower performance compared to our approach. We observed performance improvements on these benchmarks with our approach (relative to Chu et al), varying from 1.20× to 9.74× on KNL and 1.08× to 1.70× on Volta. For the remaining three benchmarks s116, s1244 and s244 in Table 4.5, our approach chose the same solution as their approach, but we still obtained better performance because *PolySIMD* generates private scalars for SoVR transformations, unlike their algorithm which generates temporary arrays for the SoVR transformations. The generation of private scalars enabled our approach to achieve performance improvements speedups ranging from 1.03× to 4.03× on KNL and 1.27× to 1.51× on Volta. The overall geometric-mean speedups on all of the eleven benchmarks were 1.57× and 1.22× on the KNL and Volta platforms.

4.7 Related Work

Since there exists an extensive body of research literature in handling memory-based dependences, we focus on past contributions that are closely related to variable expansion [113], variable renaming including SoVR [55, 29], SiVR [30, 111, 123, 29] and Array SSA [112, 124].

Comparison with past approaches involving SoVR and/or SiVR transformations. Calland et al. [29] formally defined both SoVR and SiVR transformations, and also explained the impact of these transformations on a dependence graph. Also, Calland et al. proved that the problem of finding the minimum number of statements to be transformed—to break artificial dependence paths involving anti- or output-dependences—is NP-complete, and proposed some heuristics. However, the implementation and impact of these techniques on the performance of representative benchmarks were not mentioned. But, *PolySIMD* utilizes both SoVR and SiVR in a complementary manner to coordinate each other, and is built on a polyhedral framework (PPCG), and leveraged it for statement reordering to enable vectorization. Also, we did not find a framework publicly available from the past approaches. Chu et al. work in [30, 111] discussed dependence-breaking strategies in the context of recurrence relations, and developed an algorithm for the resolution of general multi-statement recurrences using the proposed strategies. But, the proposed algorithm for the resolution of cycles is not optimal and may generate solutions having redundant SoVR transformations.

Other works on storage transformations. Array SSA has been developed to convert a given program into a static single assignment form to enable automatic parallelization of

loops involving memory-based dependences [112], and also to extend classical scalar optimizations to arrays [124]. However, applying renaming on writes of every statement of a loop body is significantly expensive in terms of additional space requirements, and may not be required for enabling vectorization. Other approaches such as variable expansion [113] can be used to break specific memory-based dependences. The variable expansion may be beneficial for applying onto scalars but expanding multi-dimensional arrays inside the inner-most loop for vectorization is expensive in terms of additional space. But variable expansion can be useful in eliminating pure-output dependence cycles unlike with SoVR and SiVR, which is a part of our future work.

Bounding additional space. There has been lack of support for bounding the extra space required to break memory-based dependences in the past approaches [29, 30]. But our approach provides a *spacelimit* clause that can help programmers to specify the maximum amount of extra storage that can be allocated. An alternative approach to enable parallelization or vectorization has always been to convert the program to (dynamic) single assignment form, through array expansion, followed by affine scheduling [125] for vectorization, and then applying storage mapping optimization [126] (a generalized form of array contraction). Yet no such scheme can provide the guarantees that the affine transformations obtained on the fully expanded arrays will enable storage mapping optimization to restore a low-footprint implementation. Enforcing an a priori limit on memory usage would be even harder to achieve. Furthermore, no integrated system enabling vectorization through such a complex path of expansion and contraction has been available until now.

4.8 Summary

Despite the fact that compiler technologies for automatic vectorization have been under development for over four decades, there are still considerable gaps in the capabilities of modern compilers to perform automatic vectorization for SIMD units. This work focuses on advancing the state of the art with respect to handling *memory-based anti* (write-after-read) or *output* (write-after-write) dependences in vectorizing compilers. In this work, we integrate both Source Variable Renaming (SoVR) and Sink Variable Renaming (SiVR) transformations into a unified formulation, and formalize the “cycle-breaking” problem as a minimum weighted set cover optimization problem. Our approach also can ensure that the additional storage introduced by our transformations remains within the user-provided bounds.

We implemented our approach in PPCG, a state-of-the-art optimization framework for loop transformations, and evaluated it on eleven kernels from the TSVC benchmark suite.

Our experimental results show a geometric mean performance improvement of $4.61\times$ on an Intel Xeon Phi (KNL) machine relative to the optimized performance obtained by Intel's ICC v17.0 product compiler. Further, our results demonstrate a geometric mean performance improvement of $1.08\times$ and $1.14\times$ on the Intel Xeon Phi (KNL) and Nvidia Tesla V100 (Volta) platforms relative to past work that only performs the SiVR transformation [29], and of $1.57\times$ and $1.22\times$ on both platforms relative to past work on using both SiVR and SoVR transformations [30]. We believe that our techniques will be increasingly important in the current era of pervasive SIMD parallelism, since non-vectorized code will incur an increasing penalty in execution time on future hardware platforms.

So far, we described our enhancements in the compiler optimizations targeting advances in general-purpose architectures such as increasing numbers of light-weight cores and larger SIMD units (Chapter 3 and Chapter 4). In the next chapters, we describe our advancements in compiler optimizations for domain-specific accelerators for the domain of machine learning and graph analytics.

CHAPTER 5

MARVEL: A DATA-CENTRIC COMPILER FOR DNN OPERATORS ON SPATIAL ACCELERATORS

5.1 Abstract

The efficiency of a spatial DNN accelerator depends heavily on the compiler and its cost model ability to generate optimized mappings for various operators of DNN models on to the accelerator’s compute and memory resources. But existing cost models lack a formal boundary over the operators for precise and tractable analysis, which poses adaptability challenges for new DNN operators. To address this challenge, we leverage the recently introduced Maestro Data-Centric (MDC) notation. We develop a formal understanding of DNN operators whose mappings can be described in the MDC notation, because any mapping adhering to the notation is always analyzable by the MDC’s cost model. Furthermore, we introduce a transformation for translating mappings into the MDC notation for exploring the mapping space.

Searching for the optimal mappings reflecting best latency and energy efficiency is challenging because of the large space of mappings, and this challenge gets exacerbated with new operators and diverse accelerator configurations. To address this challenge, we propose a decoupled off-chip/on-chip approach that decomposes the mapping space into off-chip and on-chip subspaces, and first optimizes the off-chip subspace followed by the on-chip subspace. The motivation for this decomposition is to reduce the size of the search space dramatically and also to prioritize the optimization of off-chip data movement, which is 2-3 orders of magnitude more compared to the on-chip data movement. We implemented our approach in a tool called *Marvel*, and another major benefit of our approach is that it is applicable to any DNN operator conformable with the MDC notation.

Overall, our approach reduced the mapping space by an $O(10^{10})$ factor for the four major CNN models (AlexNet, VGG16, ResNet50, MobileNetV2), while generating mappings that demonstrate a geometric mean performance improvement of $10.25\times$ higher throughput and $2.01\times$ lower energy consumption compared with three state-of-the-art mapping styles from past work. We also evaluated our approach over the GEMM, LSTM, and MLP workloads and also compared with the optimizers from past work.

5.2 Introduction

Deep learning (DL) is a fundamental technology for many emerging applications such as autonomous driving [5], translation [4], and image classification [3], with accuracy close to, and even surpassing, that of humans [66, 67, 68]. Achieving low latency and energy goals with stringent computation and memory constraints of deep neural network models (DNNs) for mobile [127] and edge [128] devices has emerged as an important challenge. To cope with this challenge, specialized hardware accelerators for DNN inference are being developed and deployed [129, 47, 46, 128, 130]. Most of these accelerators are “spatial”, i.e., they are built by interconnecting hundreds to thousands of processing elements (PEs). They achieve high throughput by exploiting parallelism over the PEs and energy efficiency by maximizing data reuse within the PE array via direct data forwarding between PEs and the use of scratchpad memories [40, 45, 46, 42, 41, 43, 131, 132].

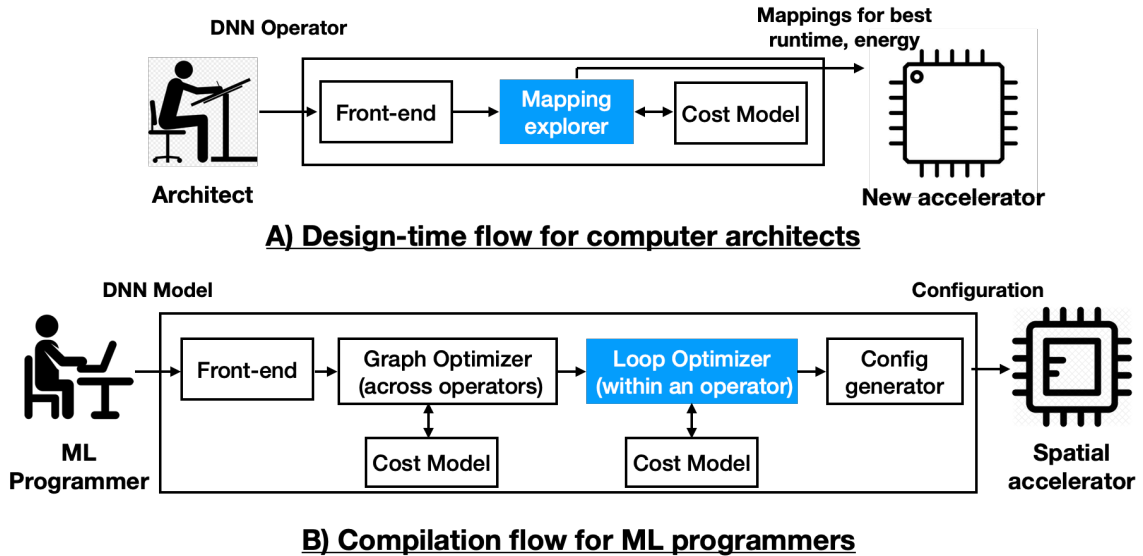


Figure 5.1: Overview of the design-time flow for computer architects developing new accelerators, and the compilation flow for ML programmers leveraging the accelerators. Scope of this work is the mapping explorer and the loop optimizer in the above diagram.

The efficiency of accelerators depends heavily on the compiler’s ability to generate optimized *mappings* for various operators of DNN models on to the accelerator’s compute and memory resources. A mapping involves parallelization, tiling, and scheduling strategies [133, 134]. Optimized compilers (or mappers) optimizing various DNN operators are necessary during compile-time for ML programmers, and design-time for computer architects to understand reuse and data movement behaviors to design a new accelerator, as shown in Figure 5.1. Thus, expressing DNN mappings and determining optimal ones is a crucial component of DNN deployment on accelerators.

Mappings are often expressed as *loop nests*, a syntax that resembles a simple imperative programming language with explicit parallelism. Many cost models such as TimeLoop [133], DMazeRunner [135], Interstellar [136] are developed over the loop nest description of mappings. The loop nests syntax is very generic and can help architects/compiler in expressing a wide range of operator mappings, but the underlying cost models may not analyze all possible mappings expressible in loop nests. Furthermore, these cost models do not have a formal boundary over DNN operators for precise and tractable analysis. Having such no formal boundaries can bring adaptability challenges to these cost models in the compiler infrastructures and also to computer architects for design-time exploration of new DNN operators onto accelerators.

In this work, we address the above challenge. We leverage the recently introduced “Maestro Data-Centric” (MDC) notation [134] for expressing mappings. MDC is promising because any mapping adhering to the notation can be analyzable using the MDC’s cost model. Moreover, the notation explicitly defines data mapping and organization, instead of inferring it from loop nests. The overall focus of this work is on (1) developing a formal understanding of DNN operators whose mappings can be described in the MDC notation, (2) introducing a transformation for translating mappings into the MDC notation for exploring the mapping space, and finally (3) proposing an efficient exploration strategy to quickly navigate the large mapping space of DNN operators. The key contributions are briefly described below.

1) Conformable DNN operators. The promising aspect of the MDC notation, i.e., analyzability, comes at the cost of its expressiveness. In this work, we introduce a formal set of rules (Section 5.4) in identifying DNN operators whose mappings can be described in the MDC notation. We call an operator satisfying the formal rules as the *conformable* operator, and Table 5.1 lists the conformability of the popular operators with the MDC notation.

2) Transformation. The MDC notation is powerful in expressing and reasoning complex mappings of DNN operators onto the diverse spatial accelerators, but explicitly writing and exploring such mappings can be error-prone and tedious. Computer architects [133] and DNN compiler frameworks [76] view the operators and their mappings majorly in the loop nest form. Hence, we introduce a *transformation* (Section 5.5) that translates a mapping specified in the loop nest form to the MDC notation and can help both the architects and compilers for mapping space exploration.

3) Mapping space exploration. The efficiency of any mapping is tightly cross-coupled with both the algorithmic aspects of DNN operators and the microarchitectural aspects of accelerators. Searching for the optimal mapping reflecting best latency and energy effi-

ciency is challenging because of a massive space of possible loop transformations on the operators. For example, there are over 10^{19} valid mappings for the CONV2D on average for mapping ResNet50 [137] and MobileNetV2 [138] on a representative DNN edge accelerator. This challenge gets exacerbated with new operators (e.g., depth-wise) and diverse hardware accelerator configurations. Much of the prior work [132, 131, 139, 140] targeted hardware with limited capabilities or fixed certain aspects of the mapping space such as choice of parallel loops and loop orders [136, 135, 132, 131, 141]. Approaches supporting broader classes of architectures and mappings suffer from a combinatoric explosion in the size of mapping space.

Our approach for the mapping problem is motivated by the observation that the off-chip data movement between DRAM and accelerator is 2-3 orders of magnitude more compared to the on-chip data movement involving the PE array and the local scratchpad buffers [40, 142]. Hence, we propose an approach (Section 5.6) referred as “decoupled off-chip/on-chip” that decomposes the mapping space into two subspaces, i.e., off-chip and on-chip subspaces, and first optimizes the off-chip subspace followed by exploring the on-chip mapping subspace constructed with the optimal mappings from the off-chip subspace. In contrast to prior work [133, 136, 135], we use different approaches and cost models for these subspaces, i.e., a classical distinct-block (DB) locality cost model [143, 56] to explore the off-chip subspace, and the MDC’s cost model [134] for the on-chip subspace.

We implemented the above approach in a tool called “Marvel”, and our approach is applicable to any operator conformable with the MDC notation. Given a conformable DNN operator, workload sizes, and a target accelerator configuration, Marvel explores the mapping space of the operator using the decoupled approach and then outputs the mappings optimized for runtime and energy. Overall, our approach reduced the mapping space by an $O(10^{10})$ factor for the four major CNN models (AlexNet, VGG16, ResNet50, MobileNetV2), while generating mappings that demonstrate a geometric mean performance improvement of $10.25\times$ higher throughput and $2.01\times$ lower energy consumption compared with three state-of-the-art mapping styles from past work. We also evaluated our approach over the GEMM, LSTM, and MLP workloads and also compared Marvel generated mappings with the optimizers from past work.

5.3 Background

In this section, we provide a brief overview of the spatial DNN accelerators and also the MDC notation to describe mappings of a DNN operator onto the accelerators.

5.3.1 Spatial DNN Accelerators

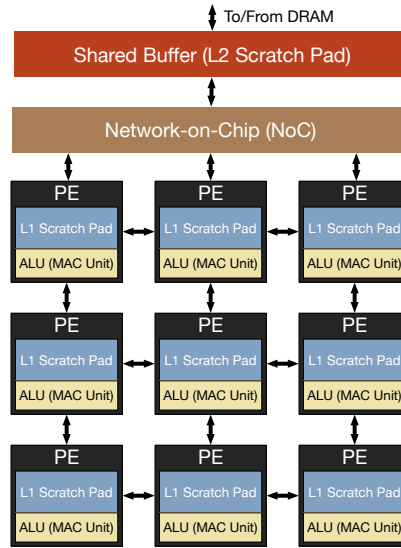


Figure 5.2: Abstract spatial accelerator model which is pervasive in many state-of-the-art accelerators [40, 46, 144, 43].

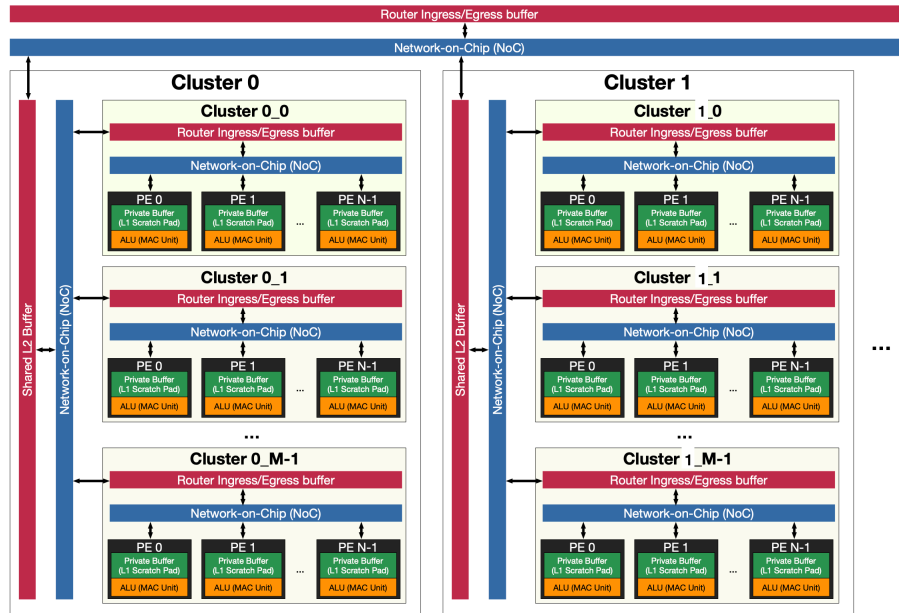


Figure 5.3: A broader overview of a spatial accelerator system having multiple accelerators in the form of clusters.

Spatial DNN accelerators based on ASICs and FPGAs have emerged to address extreme demands on performance and energy-efficiency of CNN layers [40, 45, 46, 42, 41, 43]. Such accelerators are built using an array of processing elements (PEs) to provide high parallelism and use direct communication instead of via shared memory for energy-efficiency. An abstract model of spatial accelerators is shown in Figure 5.2 (and its broader

overview with multiple accelerators in a system shown in Figure 5.3), where each PE of an accelerator consists of a single/multiple ALU(s) dedicated for multiply-accumulate operations (MACs) and a local scratchpad (L1 buffer). Also, accelerators employ various network-on-chips (NoCs) for direct communication among PEs and between PE array and L2 scratchpad buffer. The interconnection network often supports multi-casting data to multiple PEs, which can reduce the total number of data reads from L2 buffer to PEs. Unlike GPU cores, PEs can communicate with adjacent PEs (data forwarding) using a NoC, which can significantly reduce the energy consumption for expensive L2 buffer accesses. Accelerators also typically employ a large shared L2 scratchpad buffer to stage data from DRAM and also partial accumulations from PE arrays. Both L1 and L2 scratchpad buffers are software-controlled memories, i.e., programmer/compiler directly controls contents of the buffer, unlike cache memories, which implicitly manages them, and this is because the memory traffic in accelerators is known in advance. Many spatial accelerators can be further interconnected together to create a scale-out system [47].

5.3.2 MDC Notation

The Maestro Data-Centric (MDC) notation for a DNN operator mapping onto a spatial accelerator consists of two aspects, i.e., 1) Computation and tensor sizes, and 2) Data mapping directives over tensor dimensions. A sample mapping of the CONV1D operator in the MDC notation is shown in Figure 5.4(B). A major novelty of the MDC notation is that the data mappings of tensors across space (PEs) and time are explicitly specified using a set of data mapping directives, which makes the MDC’s cost-model to estimate data movement and reuse behaviors of a mapping precisely and quickly. We briefly describe the data mapping directives of the MDC notation with the mapping in Figure 5.4(B) as the example.

1) TemporalMap (size, offset) d specifies a distribution of the dimension d of a tensor across time steps in a PE, and the mapped set of dimension indices is same across PEs in a given time step. The **size** parameter refers to the number of contiguous indices mapped in the dimension d to each PE, and the **offset** parameter describes the shift in the starting indices of d across consecutive time steps in a PE. For instance, the directive **TemporalMap(2, 2) d_w** in the running example represents the distribution of first dimension (d_w) of the weight tensor with two indices mapped in each time step (i.e., $d_w=\{0,1\}$ in PE0 and PE1 at $t = 0$). Also, the offset of two denotes the increment in d_w index after each time step (i.e., $d_w=\{2,3\}$ in PE0 and PE1 at $t = 1$) till the extent of d_w dimension is explored.

2) SpatialMap (size, offset) d specifies a distribution of the dimension d of a tensor across PEs. The **size** parameter refers to the number of contiguous indices mapped in the dimension d to each PE, and the **offset** describes the shift in the starting indices of d

A) CONV1D operation

```
Tensors: O[4],W[4],I[7]
for(i0 = 0; i0 < 4; i0++)
  for(i1 = 0; i1 < 4; i1++)
    O[i0] += I[i0+i1] * W[i1]
```

B) A sample mapping in MDC representation

```
Tensors: O[4],W[4],I[7]
#do, dw, di – index variables
over tensor dimensions
```

```
Computation:
O[do] += (I[di] x W[dw])
```

```
Mapping Directives:
#di is inferred with do and dw
SpatialMap(1,1) do
TemporalMap(2,2) dw
```

C) Visualization of the data mapping on 2 PEs

	PE0	PE1
t=0	d _o = {0} d _w = {0,1} d _i = {0,1}	d _o = {1} d _w = {0,1} d _i = {1,2}
t=1	d _o = {0} d _w = {2,3} d _i = {2,3}	d _o = {1} d _w = {2,3} d _i = {3,4}
t=2	d _o = {2} d _w = {0,1} d _i = {2,3}	d _o = {3} d _w = {0,1} d _i = {3,4}
t=3	d _o = {2} d _w = {2,3} d _i = {4,5}	d _o = {3} d _w = {2,3} d _i = {5,6}

Figure 5.4: A mapping of the CONV1D in the MDC notation along with the visualization of its data mappings.

across consecutive PEs. For instance, the directive `SpatialMap(1,1) do` in the running example represents the distribution of first dimension (d_o) of the output tensor with one index mapped to each PE (i.e., $d_o=\{0\}$ in PE0 and $d_o=\{1\}$ in PE1 at $t = 0$). If the number of PEs is not sufficient to cover all indices of the dimension mapped, then the mapping is folded over time across the same set of PEs.

3) Directive order. The sequence of spatial and temporal map directives in a mapping dictates the change of data mappings to PEs across time. Similar to a loop order, all the dimension indices corresponding to a mapping directive are explored before its outer mapping directive in the sequence begins exploring its next set of indices. For instance, the sequence of directives in the running example, i.e., spatial map over d_o followed by temporal map over d_w dictates that all the dimension indices of the weight tensors need to be explored before exploring the next set of d_o indices. This order results in accumulating the partial results of an output before computing another output, popularly referred to as “output stationary” mapping [145]. However, the sequence notation has a limitation that it cannot capture scenarios where there is more than one dimension index simultaneously changing over time (except at the dimension boundaries).

4) Clusters (size) logically groups multiple PEs or nested sub-clusters with the group size as the size parameter. For example, `Cluster (2)` directive on an accelerator with

ten PEs arranges the PEs into five clusters with the cluster size as two. All the mapping directives above a cluster directive operate over the introduced logical clusters, while those below the cluster directive operate within a logical cluster. The cluster directive is extremely useful in exploiting spatial distribution of more than one tensor dimensions (e.g., row-stationary mapping [40]). Also, the directive helps in constructing hierarchical accelerators by recursive grouping.

The above aspects of the MDC notation can help in precisely specifying a wide range of mappings, including popular and sophisticated mapping styles such as row-stationary in Eyeriss [40], weight-stationary in NVDLA in [46], output-stationary in ShiDianNao [145] accelerators. However, its not clearer if all mapping behaviors of an operator can be represented in the MDC notation.

5.4 Conformable DNN Operators

In this section, we introduce formal rules in identifying conformable DNN operators whose mappings (reuse, parallelization and tiling strategies) can be described using the MDC notation. We discuss rules over the abstract loop nest notation of DNN operators without any transformations for reuse and parallelization (e.g., CONV1D in Figure 5.5).

R1: A conformable DNN operator in the abstract loop nest form must be a perfectly nested loop without any conditional statements.

The MDC notation restricts its computation to be uniform across all PEs at all time-steps. This restrict is satisfied if the computation is enclosed in a perfectly nested loop without any conditional statements. Most of the DNN operators such as CONV2D, GEMM, MLP (more in Table 5.1) can be expressed in the form of perfectly nested loops without any conditionals. But there can be implementation of certain operators such as fusion of two convolutions, where each PE requires executing the non-uniform computation. Hence, such operators are discarded and are non-conformable to the MDC notation.

R2: The perfectly nested loop must not have any dependences (flow, anti, output) except reduction dependences, and thus the loops can be freely reordered.

The MDC notation restricts the input and output tensors of an operator to be different, and this results in not having any flow- and anti-dependences between the tensors. However, the notation can support reduction operations (e.g., add, max, min), and this leads to supporting reduction dependences, i.e., flow, anti, output dependences only on the output tensor. Similar to the rule R1, most of the DNN operators mostly have only reduction dependences, except few operators such as parametric multi-step LSTMs which have flow dependences.

R3: The dimension dependence graph (DDG) of the perfectly nested loop must

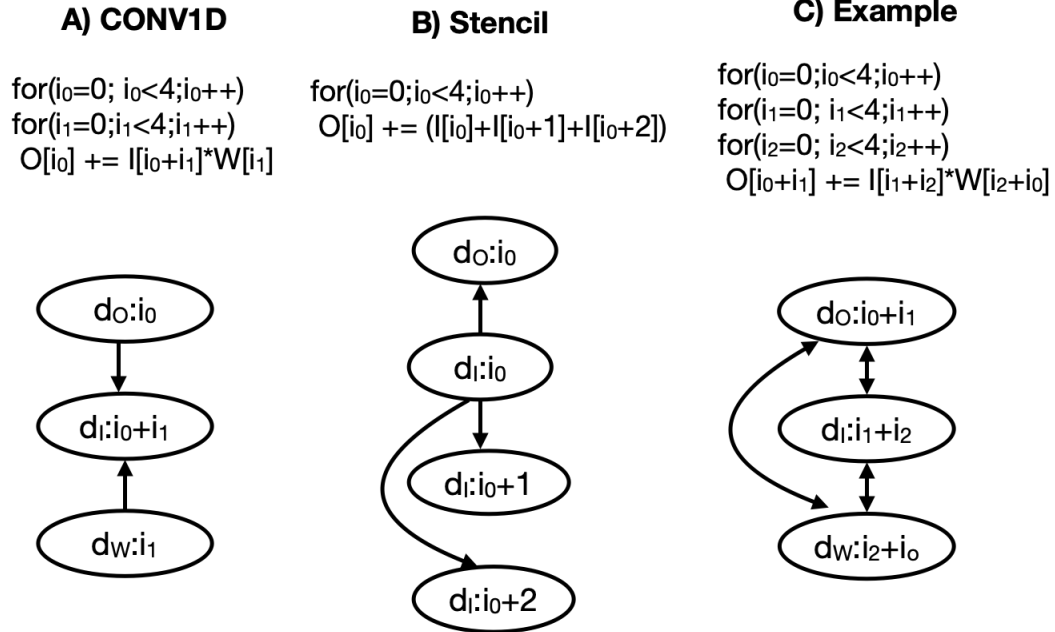


Figure 5.5: The dimension dependence graph (DDG) of simple operators such as CONV1D and stencil satisfying the rule R3, and an example violating the rule R3. $d_O/d_I/d_W$: tensor dimension variables corresponding to the output, input, and weight tensors.

have a topological ordering, and the subscripts of dependent dimension variables of the DDG graph must be in the form of linear combination of its loop iterators.

The directive order (sequence of mapping directives) of the MDC notation dictates the change of the data mappings to PEs across time. As described in the Section 5.3.2, the directive order has limitations in capturing more than one tensor dimension variable changing simultaneously over the time (except at boundaries). We introduce a directed graph called *Dimension Dependence Graph* (DDG) to find the possibility of such data movement behaviors in a DNN operator.

Each node of a DDG graph denotes a tensor dimension variable along with the array subscript referenced in that dimension. For instance, the node $(d_I : i_0 + i_1)$ in Figure 5.5(a) represents the tensor subscript $i_0 + i_1$ used in the input tensor dimension with name d_I . The edges of the DDG are constructed as follows: 1) An edge is added from a node having a SIV/MIV subscript¹ to another node having a MIV subscript if there is a common loop iterator in their subscripts. For e.g., there is a directed edge from the node $(d_O : i_0)$ to $(d_I : i_0 + i_1)$ in Figure 5.5(a) since they have a loop iterator i_0 in common. 2) All the SIV subscripts are grouped based on their loop iterators, and then edges are added from the SIV subscript of a group having the lowest constant value (randomly choose if there exists

¹Single Index Variable (SIV) subscript involves one loop iterator, whereas Multiple Index Variable (MIV) subscript involves more than one loop iterator [146].

multiple) to other SIV subscripts in the same group. For e.g., there is a directed edge from the node $(d_I : i_0)$ to all the nodes $(d_I : i_0 + 1)$, $(d_I : i_0 + 2)$, and $(d_O : i_0)$ in Figure 5.5(b). 3) If there is a loop iterator (say i) dependent on other loop iterators (say j) in its loop bounds, then construct an edge from a node with subscript having the loop iterator i to other nodes having the loop iterator j in their subscripts.

Now, the possibility of having multiple dimension variables changing simultaneously is reduced to the problem of finding a topological ordering in the DDG graph. In essence, the absence of a topological ordering indicates the presence of mutually dependent dimension variables (e.g., example in Figure 5.5(c)). In the presence of a topological ordering, the MDC notation requires the data mappings of independent dimension variables to be specified, and these variables are identified from the nodes of the DDG graph having zero in-degree. For e.g., in the case of CONV1D in Figure 5.5(a), only the data mappings of dimension variables related to output and weight tensors must be specified, and the dimension variable related to the input tensor is inferred by the underlying MDC's cost model. Hence, the subscripts of dependent dimension variables need to be linear expressions of loop iterators so as to be analyzable by the MDC's cost model. In addition, the MDC notation expects to have only one data mapping over an independent dimension variable. If there exists more than one node with zero in-degree in the DDG graph associated with the same dimension variable, then we consider that DNN operator to be non-conformable.

R4: The subscripts associated with the independent dimension variables of the DDG graph must be in the form of linear combinations of its loop iterators with the positive unit coefficients and no constants.

A mapping directive (either spatial or temporal) over a dimension variable restricts the variable to start from zero and increase with unit stride. These restrictions don't allow the dimension variable to have strided increments or negative strides. To characterize the implication of above restrictions, we assume the abstract loop nest form of the DNN operator to be normalized, i.e., its loop iterators start from zero and have unit strides. To support the restricts imposed the mapping directives, each subscript (in the normalized form) associated with an independent dimension variable must be in the form of a linear combination of the subscript's loop iterators with the positive unit coefficients and no constants. For e.g., the subscript i_0 associated with the dimension variable d_O in Figure 5.5(a) is in the linear form of its iterators (i_0) with coefficient as one and no constant.

With positive unit coefficients and no constants, the SIV subscript associated with an independent dimension variable is simply a unique loop iterator (e.g., i_0 for d_O , i_1 for d_W in Figure 5.5(a)). Furthermore, the MIV subscript associated with an independent dimension variable is also in the form of adding the subscript's loop iterators. These loop iterators

cannot be part of any subscripts associated with other dimension variables; otherwise, their in-degree wouldn't have been zero. Hence, the loop iterators corresponding to such MIV subscript can be merged into a single loop. *Overall, the subscripts associated with each of the independent dimension variables are simply unique loop iterators* (e.g., i_0 for d_o , i_1 for d_w in Figure 5.5(a)).

Finally, an operator is said to MDC conformable if it satisfies all the four rules described above. Table 5.1 lists the set of popular DNN operators and the conformability of these operators with the MDC notation. As can be seen, the MDC notation can capture most of the DNN operators except parametric LSTM's, and the mappings of these operators can be analyzable by the MDC's cost model.

Table 5.1: Conformability of the popular DNN operators onto the MDC notation (Y/N refers to YES/NO).

DNN Operator	Types	R1	R2	R3	R4	Conformable to MDC
CONV1D	Regular	Y	Y	Y	Y	Y
CONV2D	Regular	Y	Y	Y	Y	Y
	Point-wise, Depth-wise	Y	Y	Y	Y	Y
	Strided, Dilated	Y	Y	Y	Y	Y
MLP	Fully connected	Y	Y	Y	Y	Y
Pooling	Max, Avg	Y	Y	Y	Y	Y
GEMM	Regular	Y	Y	Y	Y	Y
	Triangular	Y	Y	Y	Y	Y
LSTM	Single cell	Y	Y	Y	Y	Y
	Parametric multi-cell	Y	N	Y	Y	N
Element wise	Residual	Y	Y	Y	Y	Y
	ReLU	Y	Y	Y	Y	Y
Stencils	Regular	Y	Y	Y	Y	Y

Coverage of MDC Conformable Operators. We have used Tensorflow profiler to identify the DNN primitive operators in the DNN models of MLPerf suite, VGG16, and AlexNet models. Table 5.2 lists those primitive operators and their occurrences in each DNN model. All the identified primitive operators are conformable with the MDC notation, and also, we did not have to rewrite any of those operators to make it MDC conformable.

5.5 Transformation

The MDC notation is powerful in expressing and reasoning complex mappings of DNN operators onto the diverse spatial accelerators, but explicitly writing and exploring such

Table 5.2: DNN primitive operators, occurrences, and MDC conformability in the MLPerf [147] DNN models, VGG16, and AlexNet models.

DNN Primitive Operator	MLPerf Suite					VGG16	AlexNet	MDC Confor-mable?
	Mobile-NetV1	Res-Net50	SSD-MobileNet	SSD-ResNet34	GNMT			
CONV2D	15	54	34	51	0	16	9	✓
Depth-wise CONV2D	13	0	13	0	0	0	0	✓
Bias Add	1	1	12	12	0	1	1	✓
Batch Normalization	13	20	13	15	0	0	0	✓
ReLU	27	49	35	37	0	15	8	✓
Softmax	1	0	0	1	0	1	1	✓
Avg pooling	1	0	0	0	0	0	0	✓
Max Pooling	0	1	0	1	0	5	3	✓
GEMM	0	0	0	0	9	0	0	✓

mappings can be error-prone and tedious. Computer architects [133] and DNN compiler frameworks [76] view the operators and their mappings majorly in the loop nest form [133, 132, 131]. This section introduces a transformation to translate a mapping of the conformable DNN operator in the loop-nest form into the MDC notation. In this work, we assume the target spatial accelerators having three levels of the memory hierarchy (private L1 buffer, shared L2 buffer, and DRAM). However, our transformation can be easily extendable to more levels of hierarchy.

As described in the Section 5.3.2, the MDC notation consists of two aspects, i.e., 1) Computation and tensor sizes, and 2) Data mapping directives over independent tensor dimensions. The statements enclosed in the perfectly nested loop form of the conformable DNN operator are used as the computation, and the tensor sizes are extracted from the workload configuration. The computation and tensor sizes of the MDC notation remains the same for each mapping of the operator. Then, the dimension dependence graph of the operator is constructed to identify the set of independent tensor dimension variables (having zero in-degree). If there are no such independent dimension variables, then the operator is discarded as non-conformable. The rest of the section focuses on generating data mapping directives for each mapping.

5.5.1 Data Mapping directives

According to the rule R2, the loops of a conformable DNN operator can be freely reordered, so it is safe to perform multi-level tiling to exploit temporal reuse across each level of the memory hierarchy and also to exploit parallelism of the accelerator. Each tiling, reuse

and parallelization behavior of an operator onto a spatial accelerator is referred to as a “mapping”. An example of the mapping of a CONV1D operation over a 3-level accelerator is shown in Figure 5.6 (C), and the different aspects of the mapping are described below.

1) Multi-level tiling tile sizes. A mapping includes tile sizes of all loop iterators for each level of tiling, i.e., 1) Level-1 tiling for the private L1 buffer, 2) Level-2 tiling for the parallelism, and 3) Level-3 tiling for the shared L2 buffer.

2) Inter-tile loop orders. A mapping also includes inter-tile loop orders² to describe the execution order of tiles reflecting various reuse opportunities. E.g., the level-2 inter-tile loop order reflects spatio-temporal reuse over the PE array, and the level-3 inter-tile loop order reflects temporal reuse over the on-chip L2 buffer. But, the level-1 inter-tile loop order doesn’t reflect any reuse, because these loops are annotated with parallelism. Also, the loop order among point-loops doesn’t provide any reuse opportunities because there is no more intermediate staging between the PE and its L1 buffer.

An n-level tiling will have n set of tile-loops (including parallel loops) and a set of point-loops. Each set of loops can have a different data movement (reuse) behavior based on its sizes and loop order. We introduce a term called “region” to denote a sequence of data mapping directives over independent tensor dimension variables (e.g., Region R1 in Figure 5.6(d)) without any cluster directives, and each region captures the data movement behavior present in each set of loops. Given a mapping of the operator in the form of multi-level tile sizes and inter-tile loop orders, our approach transforms the mapping into the MDC notation as per the following steps.

1) Point-loops. As described in Rule 4, each subscript associated with an independent dimension variable is simply an unique loop iterator. Our approach translates each loop of point-loops into a temporal map directive over the corresponding independent dimension variable with `size` and `offset` parameters of the directive being the point-loop size. For, e.g., the point loop t_{li} with tile size as T_{li} in Figure 5.6(c) is directly translated into `TemporalMap(T_{li}, T_{li}) do` in the region R1 shown in Figure 5.6(d). Since the loop order among the point-loops doesn’t provide any reuse benefits, the directive order in the region R1 doesn’t matter.

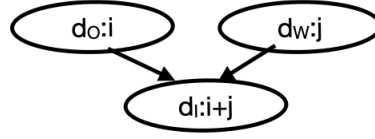
2) Parallel-loops. Since each independent dimension variable is uniquely associated with a loop iterator, parallel execution of each loop iterator introduces a different data movement behavior. Hence, for each parallel loop, we introduce a region with a spatial map over the dimension variable associated with the parallel loop, and the temporal maps for the rest of

²An n-dimensional loop nest after one level of tiling will have 2n loops. The outer n-loops are referred to as inter-tile loops and the later n-loops as intra-tile loops. The innermost n-loops after multi-level tiling are called as point-loops.

A) CONV1D operation

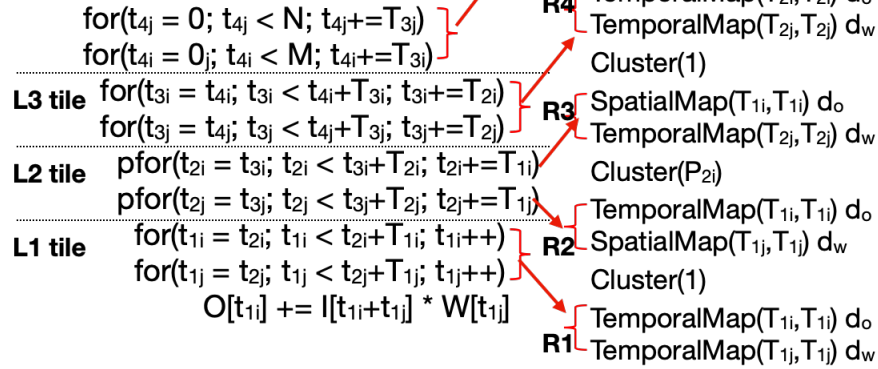
```
for(i = 0; i < M; i++)
  for(j = 0; j < N; j++)
    O[i] += I[i+j] * W[j]
```

B) DDG



C) A sample mapping in the loop-nest representation

Level-1 tile sizes: T_{1i}, T_{1j}
 Level-2 tile sizes: T_{2i}, T_{2j}
 Level-3 tile sizes: T_{3i}, T_{3j}
 Level-2 inter-tile order: t_{3i}, t_{3j}
 Level-3 inter-tile order: t_{4j}, t_{4i}



D) Mapping directives

d_o, d_w, d_i — index variables over tensor dimensions

$$\#P_{2i} = \frac{T_{2i}}{T_{1i}}, P_{2j} = \frac{T_{2j}}{T_{1j}}$$

Figure 5.6: A brief overview of the mapping expressed in the loop-nest form of CONV1D, and its translation into the MDC notation with data mapping directives.

the dimension variables in the region. For, e.g., there are two regions with name R2 and R3 for the parallel loops corresponding to t_{2j} and t_{2i} , respectively. Also, the dimension d_w associated with the iterator t_{2j} and the dimension d_o associated with the iterator t_{2i} are translated into spatial maps in R2 and R3 regions respectively. The size and offsets of each spatial map over a dimension variable is derived from the strides of the parallel loop iterators corresponding to the dimension variable. The order of directives in each region corresponding to a parallel loop doesn't matter because the number of iterations arising from the rest of the temporal maps is one. Each region corresponding to a parallel loop (except the innermost) is ended with a cluster directive with size as the number of iterations in the parallel loop. For, e.g., the region R3 is ended with a cluster directive with size as the number of iterations of the loop t_{2i} .

3) Inter-tile loops. For each set of tile-loops excluding parallel loops, our transformation generates a region by creating a temporal map directive for each loop of the set with the size and offset of the directive as the loop stride. For, e.g., the inter-tile loop t_{3j} with

stride as T_{2j} in Figure 5.6(c) is directly translated into $\text{TemporalMap}(T_{2j}, T_{2j})d_I$ in the region R4 shown in Figure 5.6(d). The order of directives in a region is governed by the loop order among the corresponding tile-loops. For, e.g., the level-3 inter-tile loop order (t_{3j}, t_{3i}) dictates the temporal map over d_W outer compared to temporal map over d_O in region R5. Furthermore, each region is separated by cluster directive with size one to support different data movement behaviors across each set of tile-loops.

5.6 Mapping Space Exploration

The mapping space of a conformable DNN operator onto an accelerator having three levels of memory hierarchy is a cross product of valid level-1 tile sizes, level-2 tile sizes (parallelism), level-2 inter-tile loop orders, level-3 tile sizes, and level-3 inter-tile loop orders. For example, there are over 10^{19} valid mappings for a single CONV2D operator on average for mapping ResNet50 and MobileNetV2 on a representative DNN edge accelerator. Because of this massive space of mappings, searching for efficient mappings is really challenging. This challenge gets exacerbated with new operators (e.g., depth-wise) and diverse hardware accelerator configurations (e.g., tree-based interconnect [144]).

We consider (optional) a limited form of data-layouts, i.e., innermost dimension re-ordering [148] for the tensors of operators on the DRAM. Overall, the mapping space of an operator is a Cartesian product of six dimensions which represent different aspects of a mapping, i.e., 1) level-1 tile sizes, 2) level-2 tile sizes (parallelism), 3) level-2 inter-tile loop orders, 4) level-3 tile sizes, 5) level-3 inter-tile loop orders, and 6) data-layout of tensors. The first three dimensions are grouped under “*on-chip mapping subspace*” since they influence parallelization and on-chip data movement, and the remaining three dimensions are grouped under “*off-chip mapping subspace*” since they influence the off-chip data movement.

Our approach towards the mapping space exploration is motivated by the observation that the off-chip data movement between DRAM and accelerator is 2-3 orders of magnitude more compared to the on-chip data movement. Hence, we propose an approach referred as “decoupled off-chip/on-chip” that decomposes the mapping space into two subspaces, i.e., off-chip and on-chip subspaces, and first optimizes the off-chip subspace followed by the on-chip subspace which is constructed with the optimal mappings from the off-chip subspace. In contrast to prior work [133, 136, 135], we use different approaches and cost models for these subspaces, i.e., a classical distinct-block (DB) locality cost model [143, 56] to explore the off-chip subspace, and the MDC’s cost model [134] for the on-chip subspace. The overall approach is implemented as a standalone tool (shown in Figure 5.7) that

takes a conformable DNN operator, workload sizes, and a target accelerator configuration, then explores the mapping space of the operator using the decoupled approach, and finally outputs the mappings optimized for runtime and energy.

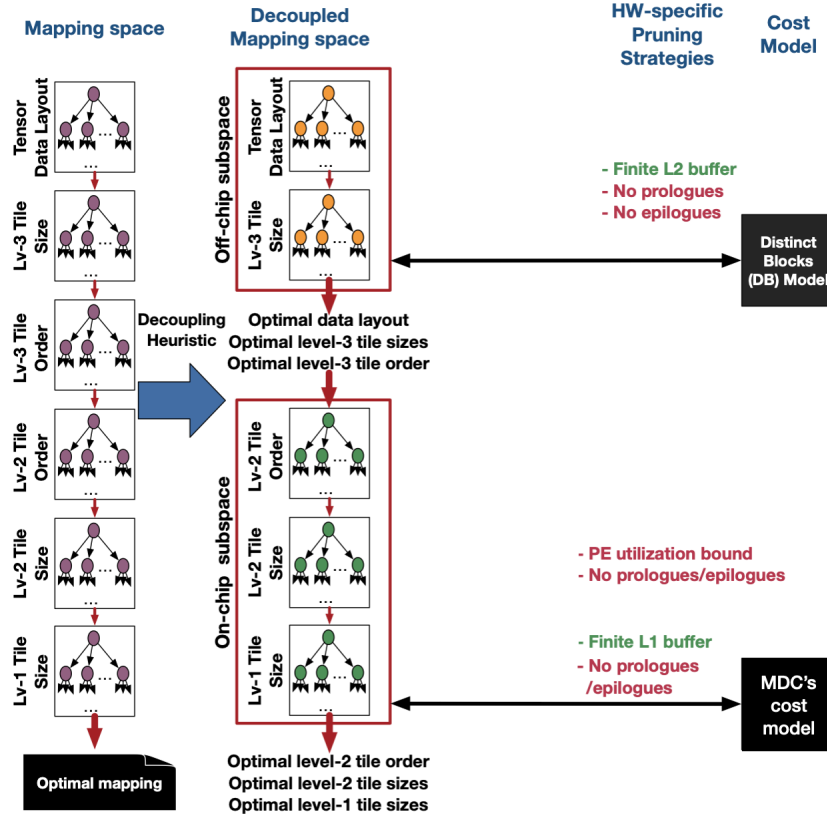


Figure 5.7: An overview of our approach along with pruning strategies for searching mapping space of convolutions. The pruning strategies in green color preserve optimal mappings, whereas the strategies in red color may prune optimal.

5.6.1 Solving off-chip mapping subspace

The goal of finding an optimal mapping in the off-chip mapping subspace is to minimize off-chip data movement between DRAM and the L2 buffer of an accelerator. In our work, we assume the L2 buffer to be a software-managed scratchpad buffer, and *reducing the off-chip data movement³ is equivalent to finding a level-3 tile that has highest arithmetic intensity*, this is because the highest arithmetic intensity results in higher reuse and less data transfer.

In our approach, we consider the classical distinct-block (DB) locality cost model [143] to measure the off-chip data movement cost, which was developed as part of the memory

³In case of non-software-managed scratchpad buffers, reducing data movement between DRAM and L2 buffer is equivalent to finding a level-3 tile whose memory footprint can fit into the L2 buffer and is maximum.

cost analysis to guide automatic selection of loop transformations and also optimal tile size selections [56, 149, 150] in IBM XL compilers. The DB model is a good choice for our approach, since the model only focuses on optimizing for off-chip data movement. Moreover, it focuses only on perfectly nested loop, and conformable DNN operators are perfectly nested loops as per the rule R1 in Section 5.4.

The distinct blocks (DB) model starts with data-layouts of multi-dimensional arrays and also the parametric tiled version of a perfectly nested loop. Then, the model symbolically estimates the off-chip data movement cost involved in a tile of computation by measuring the number of the distinct number of DRAM blocks required for all the references in the tile of computation. Assuming the array I is laid out in the row-major order, the distinct number of DRAM blocks (with block size as B and tile sizes T_X, T_Y) required for an example array reference $I[x+y][y]$ enclosed in a triply nested loop with iterators x, y, z is computed as follows:

$$DB_I(T_X, T_Y) \approx \left(\left\lceil \frac{T_X + T_Y}{b} \right\rceil \right) \times (T_Y) \times T_Z$$

In the above formulation, the innermost access of the reference is divided by the block size⁴, because the data movement with DRAM happens in multiples of block sizes. Now, the total data movement cost (DMC), a.k.a. memory cost per iteration, involved in a tile is computed as the number of distinct DRAM blocks required for all references in the tile by the total number of iterations in the tile. The optimal level-3 tile sizes and data-layouts are computed by minimizing the data movement cost function for every layout and tile sizes in the off-chip mapping subspace with the two constraints, i.e., 1) the tile size of a loop should be greater than 0 and should not exceed its corresponding loop bound, and 2) the total data required (including double buffering) for a level-3 computation tile should fit into the on-chip L2 buffer.

After computing the optimal level-3 tile sizes and data-layouts of tensors, our approach computes the partial derivatives (slopes) of the data movement cost function (based on the optimal data-layout) with respect to parametric level-3 tile sizes (similar to [56]), and evaluate the partial derivatives by substituting optimal level-3 tile sizes. The key insight is that having a higher negative value of a partial derivative along a loop indicates the lesser distinct number of elements referenced along the loop, i.e., highest reuse along the loop, and it is suggested to keep it in the innermost position to exploit maximum temporal reuse. Similarly, the rest of the loops are ordered based on their partial derivative values.

⁴Setting block size to one ignores the impact of data-layouts that we consider in our approach (innermost dimension reordering [148]).

5.6.2 Solving on-chip mapping subspace

The on-chip mapping subspace is constructed based on the optimal values of level-3 tile sizes. Then, our approach explores the constructed subspace to find optimal mappings for each of the three optimal goals, i.e., lower runtime (higher throughput), lower energy consumption, and lower energy-delay product. For each mapping of the constructed subspace, our approach transforms the mapping into its equivalent MDC notation (described in Section 5.5). Then, our approach uses the MDC’s cost model [134] to estimate various metrics such as latency and energy of each mapping in the on-chip subspace. The MDC’s cost model precisely computes performance and energy, accounting for under-utilization, edge conditions, and data reuse or movement across time (via L1/L2 buffers [40]), space (via broadcast links [144]), and space-time (via neighboring links [43, 151]) without requiring explicit RTL/cycle-level simulations or access to real hardware.

Algorithm 3: Our approach to explore on-chip mapping subspace, including pruning strategies

```
1 for every level-2 inter-tile loop order do
2   for every level-2 tile size do
3     Hardware pruning: PE utilization bound
4     Hardware pruning: No prologues/epilogues
5     for every level-1 tile size do
6       Hardware pruning: Finite L1 size buffer
7       Hardware pruning: No prologue/epilogue
8       // Translate mapping into MDC form
9       Invoke the MDC’s cost model → (runtime, energy, and other metrics)
```

Algorithm 3 shows an overview of our approach in exploring the on-chip mapping subspace along with pruning strategies. We introduce a parameter called “PE utilization bound (p)” to prune search space of level-2 tile sizes by bounding the overall PE array utilization to be at-least the parameter p . The above technique is beneficial in finding optimal on-chip mappings with the optimization goal being throughput, because the highest throughput is typically obtained at higher PE utilization rates [129]. Our approach also includes a pruning strategy to choose level-1 and level-2 tile sizes such that they don’t result in any prologues or epilogues, i.e., the tile sizes are factors of loop bounds. All of the above-mentioned pruning strategies can be enabled/disabled in Marvel by passing them as input parameters.

5.7 Evaluation

In this section, we begin with an overview of the experimental setup used in our evaluation. Then, we present the evaluation of mappings generated by Marvel for a wide variety of DNN operators (CONV2D, GEMM, MLP, and LSTM), and discuss insights from the mappings while comparing them with previous work.

Table 5.3: Accelerator setups in our evaluation.

	Accelerator platform (P1) (Eyeriss-like [40])	Accelerator platform (P2) (Edge/IoT-like [128])
#PEs	168	1024
Clock frequency	200 MHz	200 MHz
GigaOpsPerSec(GOPS)	67.2	409.6
NoC bandwidth (GB/s)	2.4	25.6
L1 buffer size	512B	512B
L2 buffer size	108KB	108KB
DRAM block size [152]	64	64

Target accelerators. Marvel is applicable to any spatial accelerator since it abstracts accelerator details as #PEs, L1/L2 buffer sizes, NoC bandwidth, reduction/multicast support and others, which can be used to model a wide variety of accelerators including Eyeriss [40], NVDLA [46], TPU [128], xDNN. Due to space limitations, we present our evaluation for only two accelerator platforms (shown in Table 5.3): An accelerator (Eyeriss-like [40]) having 168 PEs and 2.4GB/s NoC bandwidth, and another accelerator having 1024 PEs and 25.6GB/s. We inherit L1, L2 buffer, and clock frequency for both platforms from Eyeriss [40], i.e., 512B L1 buffer, 108KB L2 buffer, and 200MHz clock frequency. The bidirectional NoC used in our evaluation is a two-level hierarchical bus, which has support for multicasting similar to Eyeriss.

Experimental variants. We have implemented few of the exploration strategies of recent optimizers such as Interstellar [136] and dMazeRunner [135] in our framework. For, instance, the Interstellar optimizer focuses on parallelizing input and output channels of CONV2D operators, whereas the dMazeRunner optimizer focuses on parallelizing only output channels and a limited set of loop orders. We compare Marvel generated mappings for each workload and accelerator platform with three variants: 1) Marvel implemented Interstellar-like [136] optimizer generated mappings, 2) Marvel implemented dMazeRunner-like [135] optimizer generated mappings, and 3) Roof-line peak based on the workload arithmetic intensities and accelerator configurations.

Methodology. We have evaluated all the mappings generated by the experimental variants using the MAESTRO cost model [134]. Moreover, the analytical cost model within the MAESTRO framework is validated against the RTL implementations of Eyeriss [40] and

MAERI [144] on VGG16 and AlexNet models. We passed a pruning option to the Marvel to choose tile sizes that divide loop bounds evenly without any remainder, and this has been the consideration in the other approaches [133, 136, 135, 131, 132]. We also set the minimum PE array utilization bound as 0.1, i.e., at-least 10% of the PE array should be mapped with computation. We apply 8-bit fixed point precision for all the tensors used in our evaluation.

5.7.1 Evaluation on CONV2D

The CONV2D is a widely used DNN operator in convolution neural networks, and these operators account for more than 90% of overall computation [63, 40], dominating overall latency, and energy consumption in inferences. In our evaluation, we considered popular CNN models, such as AlexNet [64], VGG16 [65], ResNet50 [137], and MobileNetV2 [138], with a batch size of one as this captures the low latency requirement use case and also represents a more challenging setup for energy efficiency and throughput [129]. In addition, these models encompass a broad spectrum of CONV2D operators such as regular, point-wise, depth-wise, strided variants with different filter shapes.

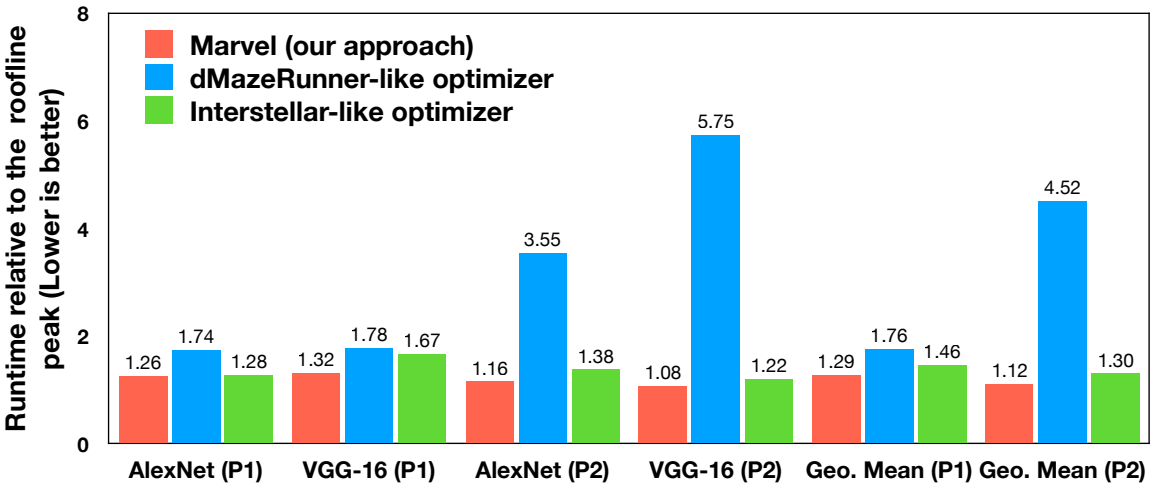


Figure 5.8: Performance comparison of Marvel generated mappings with the mappings of dMazeRunner-like optimizer [135] and Interstellar-like optimizer [136] relative to the roof-line peaks of the AlexNet and VGG-16 models on both the platforms (P1 and P2).

Comparison with the existing optimizers. Figure 5.8 presents the runtimes of optimized mappings generated by Marvel, dMazeRunner-like optimizer [135], and Interstellar-like optimizer [136] relative to the roof-line peaks of the AlexNet and VGG-16 models on both the platforms. Since each model involves multiple CONV2D operations, we have added the runtimes of the each CONV2D operator to present our evaluation at the level of

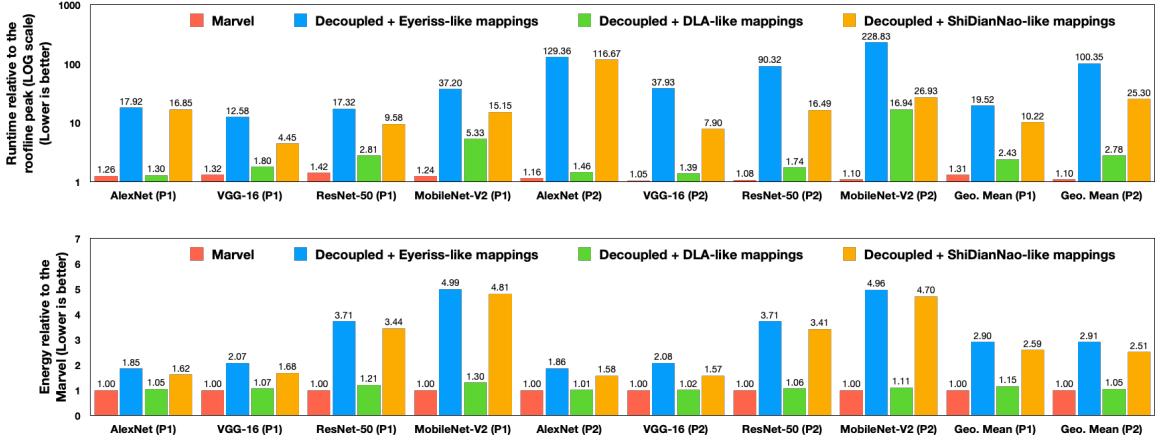


Figure 5.9: Runtime and energy comparison of Marvel generated mappings with the popular mapping styles such as row-stationary (RS) from Eyeriss [40], weight-stationary from DLA [46], output-stationary from ShiDianNao [145] for the AlexNet [64], VGG-16 [65], ResNet-50 [137], MobileNet-V2 [138] models on both the platforms (P1 and P2).

DNN models. The Interstellar-like optimizer is almost equivalent to the brute-force exploration except that it restricts exploiting parallelism along only input and output channels. As a result, the evaluation using the Interstellar-like optimizer is really time-consuming (multiple days for MobileNetV2 and ResNet50), and hence we restricted the comparison to only AlexNet and VGG16 models. As can be observed from the Figure 5.8, Marvel generated mappings are geometrically $2.35\times$ and $1.15\times$ faster compared to the mappings obtained by the dMazeRunner-like optimizer and Interstellar-like optimizer, respectively. The dMazeRunner-like optimizer focuses on exploiting parallelism along only output channels (in presence of unit batch size) to avoid inter-PE communication, and this results in under-utilization of the PE array for both models. But, the Interstellar-like optimizer is able to perform close to Marvel, because the number of input and output channels in these models are larger (except at the initial layers). But, it can underperform for DNN models such as UNet [153], where input and output channels are smaller and output width and height are larger. Furthermore, our approach is able to identify mappings in seconds to few minutes for each operator on a local machine, unlike the Interstellar-like optimizer which takes almost 1-5 hours for each operator. We don't compare the search time with the dMazeRunner-like optimizer, because we haven't implemented all the heuristic strategies, for, e.g., exploring tiling factors that highly utilize (at-least 60 %) the scratchpad buffers. Table 5.4 shows the impact of our decoupling and pruning strategies on the original search space of mappings of the four models with an average reduction of $O(10^{10})$ in the mapping space.

Comparison with the popular mapping styles. Some of the state-of-the-art mapping

Table 5.4: The statistics (min/avg/max) of the CONV2D mapping space in our evaluation and the resultant mapping subspaces after decoupling and pruning strategies.

Variants	Search space size		
	Min	Avg	Max
Original search space	2.7×10^{17}	9.4×10^{18}	1.8×10^{19}
Off-chip schedules search space after decoupling	7.3×10^8	3.6×10^{11}	1.3×10^{12}
On-chip schedules search space after decoupling	2.9×10^7	2.4×10^{10}	1.4×10^{11}
Off-chip schedules search space after decoupling + pruning	9.9×10^5	1.5×10^8	6.3×10^8
On-chip schedules search space after decoupling + pruning	3.8×10^5	5.9×10^7	2.4×10^8

styles are row-stationary (RS) from Eyeriss [40], weight-stationary from DLA [46], and output-stationary from ShiDianNao [145]. In our evaluation, we encoded the above mapping styles in the form of parallelization and loop order constraints on the on-chip mapping space of our decoupled approach. For instance, weight-stationary (DLA) mapping style includes parallelization across input and output channels with the loop iterators corresponding to the weight tensor in the innermost positions of the loop orders. As can be observed from Figure 5.9, the runtimes of Marvel generated mappings for all the models are only $1.31 \times$ and $1.10 \times$ higher relative to the roof-line peaks of all the models on both accelerator platforms P1 and P2, respectively.

The Eyeriss-like mappings [40] exploit parallelism along output width and filter width dimensions, whereas the ShiDianNao-like mappings [145] exploit along output width and height. But, the extents of these dimensions are relatively small especially in modern DNN models such as ResNet50 and MobileNetV2. Hence, these mappings are often resulted in under-utilization of the PE array leading to higher runtimes compared to the roof-line peak (e.g., $100.36 \times$ for Eyeriss-like mappings on platform P2). But these mappings exploit popular row-stationary and output-stationary behavior leading to lower energy consumption (e.g., $2.91 \times$ for Eyeriss-like mappings on platform P2) relative to the Marvel reported energy-efficient mappings.

The DLA-like mappings exploit parallelism along input and output channels, and the extent of these dimensions are sufficient enough to keep the PE array busy for most of the layers of AlexNet, VGG16, and ResNet50 models. However, the MobileNetV2 model has introduced depth-wise operators which lacks parallelism in the input channels. This resulted in less performance of the DLA-like mapping compared to the roof-line peak, and our approach exploited alternate dimensions (more than one) for the parallelism. However, the DLA-like mappings exploit weight-stationary reuse behavior, and these DNN models

have large number of weight parameters compared to other tensors. This resulted in only $1.10\times$ higher energy consumption relative to the Marvel reported energy-efficient mappings.

Table 5.5: Two layers from VGG16 and MobileNetV2 for brief discussion on our approach generated mappings; Level-3 tile sizes and degree of parallelism are part of the mappings identified by our approach on Platform P2.

Loop iterators of CONV2D	CONV1 in VGG16			Bottleneck6.3.2 in MobileNetV2		
	Regular CONV2D			Depth-wise separable		
	Loop sizes	Level-3 tile sizes	Degree of parallelism	Loop sizes	Level-3 tile sizes	Degree of parallelism
Batch(N)	1	1	1	1	1	1
Filters (K)	64	64	8	1	1	1
Input channels (C)	3	3	1	576	64	32
Output width (P)	222	111	37	5	5	5
Output height (Q)	222	6	3	5	5	5
Filter width (S)	3	3	1	3	3	1
Filter height (R)	3	3	1	3	3	1

To deeply explain the mappings generated by our approach and its difference with respect to the state-of-the-art mapping styles, we consider two convolutions, i.e., a regular CONV2D from VGG16 and a depth-wise CONV2D from MobileNetV2, whose details are shown in Table 5.5.

Impact of level-3 tile sizes. The CONV2D operator in VGG16 Layer 1 has higher output width and height (P, Q) compared to the output and input channels (K, C). However, the level-3 tile size corresponding to output height is shrunk to fit into the on-chip buffer with maximum temporal reuse. As a result, our approach exploited parallelism along output width (P) and filters (K) to utilize the PE array maximum. However, none of the state-of-the-art mapping styles and also dMazeRunner-like/Interstellar-like optimizers exploit parallelism along P and K dimensions.

Impact of modern operators. The modern DNN models such as MobileNetV2 have introduced depth-wise CONV2D operators, and these operators reduce total number of MAC operations by not performing reduction across input channels, there by sacrificing arithmetic intensity. As a result, these operators have less parallelization opportunities and are often bounded by NoC bandwidth. For example, the depth-wise CONV2D operator in Table 5.5 have the value of K set to one and also the level-3 tile size of C is shrunk to a smaller value to fit into the on-chip buffer with maximum temporal reuse. To fully leverage the PE array, our approach generated mapping exploited parallelism along three dimensions – Input channels (C), Output width (P), Output height (Q), where none of the prior state-of-the-art mapping styles and dMazeRunner-like/Interstellar-like optimizers exploited more

than two levels of parallelism. Furthermore, the performance of the generated mapping was close to the roof-line peak, which is dominated by NoC bandwidth.

An overall performance (runtime) and energy comparison of Marvel generated mappings with respect to the prior state-of-the-art mapping strategies is shown in Figure 5.9.

5.7.2 Evaluation on GEMM

In this evaluation, we have considered GEMM workloads from the recent work in [154]. An interesting aspect of these workloads is that they are irregular in their shapes making the rigid accelerators (e.g., TPUs) hard to reach their peak utilization [154]. A summary of these workloads is shown in Table 5.6, where M, N, K refers to number of rows, columns of first matrix followed by the columns of second matrix.

Table 5.6: Description of the GEMM workloads taken from the recent work in [154].

Workload	Application	Dimensions		
		M	N	K
GNMT	Machine Translation	128	2048	4096
		320	3072	4096
		1632	36548	1024
		2048	4096	32
DeepBench	General Workload	1024	16	500000
		35	8457	2560
Transformer	Language Understanding	31999	1024	84
		84	1024	84
NCF	Collaborative Filtering	2048	1	128
		256	256	2048

We translated the GEMM workloads into their equivalent CONV2D workloads for the Interstellar-like and dMazeRunner-like optimizers, because their exploration strategies are specific to the CONV2D workloads (e.g., parallelization strategies). Figure 5.10 presents the runtime of optimized mappings generated by Marvel, dMazeRunner-like optimizer [135], and Interstellar-like optimizer [136] relative to the roof-line peak of each GEMM workload. The runtimes of Marvel generated mappings are only 1.24 \times and 1.10 \times higher relative to the roof-line peaks of accelerator platforms P1 and P2 respectively, thereby demonstrating the closeness of mappings obtained using our approach to the peak. Furthermore, we observed that maximum reuse (spatial, temporal, spatio-temporal) is exploited only when all the dimensions of the GEMM operator are parallelized. Hence, Marvel generated mappings included parallelization of the three dimensions to make the PE array occupied along with exploiting maximum reuse. This is in contrast to other approaches, i.e., Interstellar-like optimizer focusing on parallelizing only (N, K) dimensions and dMazeRunner-like optimizer focusing on parallelizing only (K) dimension. As a result,

Marvel generated mappings are $6.87\times$ and $1.81\times$ faster compared to the mappings obtained by the dMazeRunner-like optimizer and Interstellar-like optimizer for all the GEMM workloads on the both accelerator platforms.

5.7.3 Evaluation on MLP and LSTM

In this evaluation, we have considered the MLP and LSTM workloads from the Interstellar work in [136], and a summary of these workloads are shown in Table 5.7.

Table 5.7: Description of the MLP and LSTM workloads taken from the Interstellar work in [136].

Network	Layer	Input channels	Output channels
MLP-M	FC1	784	1000
	FC2	1000	500
	FC3	500	250
MLP-L	FC1	784	1500
	FC2	1500	1000
	FC3	1000	500
Network	Embedding size		Batch size
LSTM-M	500		128
LSTM-L	1000		128
RHN	1500		128

We translated the MLP workloads into CONV2D workloads for the Interstellar-like and dMazeRunner-like optimizers. We also translated the LSTMs workloads into their equivalent CONV2D workloads via first converting into GEMM workloads. For instance, a LSTM workload with batch size as B and embedding size⁵ as E can be translated into a GEMM workload with M being the batch size (B), N being the embedding size (E), and K being the $2\times E$. Figure 5.10 presents the runtime of optimized mappings generated by Marvel, dMazeRunner-like optimizer [135], and Interstellar-like optimizer [136] relative to the roof-line peak of each workload in Table 5.7. Marvel generated mappings are $4.46\times$ and $1.22\times$ faster compared to the mappings obtained by the dMazeRunner-like optimizer and Interstellar-like optimizer for all the workloads on the both accelerator platforms. The benefits compared to dMazeRunner-like optimizer is higher because of its parallelization across only a single dimension (Embedding size in case of LSTM and Output channels in case of MLP) and also exploring only limited loop orders for reuse. In addition, Marvel is able to do better compared to Interstellar-like optimizer by exploring more levels of parallelism to make the PE array occupied (e.g., only $1.04\times$ higher relative to roof-line peak on platform P2).

⁵Embedding size is the size of input and hidden vectors.

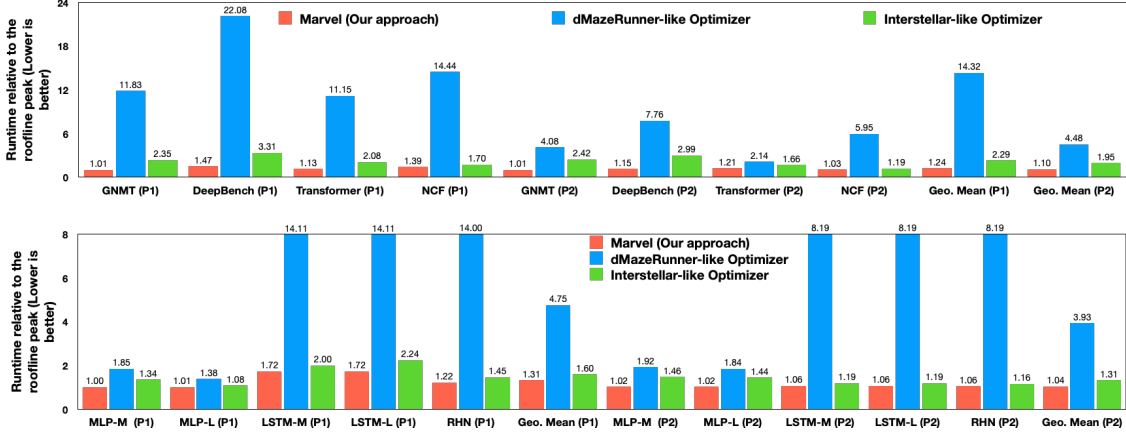


Figure 5.10: Performance comparison of Marvel generated mappings with the mappings of dMazeRunner-like optimizer [135], and Interstellar-like optimizer [136] relative to the roof-line peaks of the GEMM workloads in Table 5.6 and LSTM, MLP in Table 5.7 on both the platforms (P1 and P2).

5.8 Related Work

A major difference between the compilers for spatial accelerators and CPUs/GPUs is the need for an “accurate” cost model for finding optimal mappings. This is because spatial accelerators’ performance is sensitive to the mapping parameters, for, e.g., a small change in the tile size or degree of parallelism would drastically change the latency or energy efficiency numbers. Even though high-level frameworks such as TVM [76], TC [155], PlaidML [156], Stripe [157], Polyhedral (Tiramisu [158]), and MLIR [159] have richer expressibility than our MDC notation, none of these frameworks have accurate cost models targeting flexible spatial accelerators. Also, it’s not clearer if accurate cost models can exist for any operator expressed in their notations. An overview of the comparison of our MDC notation with prior notations in terms of expressiveness, mapping notation, the presence of accurate cost models for spatial accelerators is shown in Table 5.8. Frameworks such as TVM, TC, PlaidML, Stripe, ISAMIR, Tiramisu, MLIR can represent interleaving of operators resulting in imperfectly nested loops, but there doesn’t exist any accurate cost models in these frameworks to explore interleaving on spatial accelerators.

In addition to Marvel supporting all the primitive operators supported by other frameworks Figure 5.11, Marvel can also support operators having non-rectilinear iteration spaces (e.g., Symmetric GEMM [160]), that none of the other frameworks in Figure 5.11 support. Furthermore, a strong guarantee of our approach is that any operator conformable to the MDC notation can leverage our compiler along with the underlying accurate cost-model. This is in contrast to other frameworks for spatial accelerators such as TimeLoop [133] and Interstellar [136], where there are no such guarantees.

Table 5.8: Comparison of our MDC notation with prior compilers in terms of expressiveness, mapping notation, and the presence of accurate cost models.

Notation	Operator expressiveness			Mapping representation	Accurate cost models for spatial accelerators
	Loop nest structure	Array subscripts	Iteration domain		
MDC	Perfect	Affine	Affine	Data-centric	YES
TVM, TC, PlaidML, Stripe, ISAMIR	Perfect/Imperfect	Affine	Rectangular	Loop-centric	NO (*only for limited scenarios [136])
Polyhedral (e.g., Tiramisu)	Perfect/Imperfect	Affine	Affine	Loop-centric	NO
Generic loop nests (e.g., MLIR)	Perfect/Imperfect	Any	Any	Loop-centric	NO (*only for limited scenarios [133])

Now, we discuss prior work only on compilers/mappers (shown in Figure 5.11) for finding efficient mappings of DNN operators on to the spatial accelerators. Prior work [139, 40] focused on developing mappers specific to their architectures, for, e.g., mRNA mapper [139] for the MAERI accelerator [144], Auto-TVM [76] for the GEMM core of the VTA architecture [161] limiting their applicability to generic spatial accelerators. Prior work such as Zhang et al. [132], Ma et al. [131] focused on spatial accelerators without L1 buffers inside a PE, again limiting their mapping space formulation. Furthermore, they don’t employ accurate cost models and focus only on optimizing for runtime.

In addition, other prior works such as Interstellar [136], dMazeRunner [135] fixed certain aspects of mapping space such as choice of parallel loops, loop orders, and these choices may not reflect the efficient mappings for a wide variety of DNN operators. To the best of our knowledge, TimeLoop [133] is the only framework that considers all aspects of a mapping for a fully flexible spatial accelerator. However, it employs either an exhaustive linear search or a random sampling-based heuristic to explore the search space. In contrast to all of the above works, our approach considers all the aspects of mapping space and uses the decoupled strategy to efficiently navigate the mapping space.

A key novelty of our work is the formalization of MDC conformable operators using the four rules defined in Section 5.4, and with the conformability, our approach always generates a correct set of MDC directives corresponding to a loop nest mapping of the operator. The prior work introducing MDC directives [134] doesn’t have any formalization and also any correctness checker over the usage of MDC directives. Further, the prior-work is limited to hardware DSE and doesn’t have any mapping explorer, unlike our approach.

Compiler/ Mapper	Target architecture	Target goal	Accurate cost models	Operators supported/ evaluated	Level-1 Tiling	Level-2 tiling			Level-3 tiling		Approach
					Tile sizes	Parallel loops	Degree of parallelism	Inter-tile order	Tile sizes	Inter-tile order	
mRNA	MAERI	Runtime, Energy	YES	CONV2D	NA	YES	YES	YES	NO	NO	Bruteforce
Auto-TVM	VTA	Runtime	NO	CNNs	NA	YES	YES	YES	YES	YES	Annealing
Zhang et al.	Spatial	Runtime	NO	CONV2D	NA	FIXED	YES	FIXED	YES	YES	Bruteforce
Ma et al.	Spatial	Runtime	NO	CONV2D	NA	FIXED	YES	FIXED	YES	YES	Bruteforce
dMaze Runner	Spatial	Runtime, Energy	YES	CONV2D	YES	FIXED	YES	FIXED	YES	FIXED	Bruteforce
Interstellar	Spatial	Runtime, Energy	YES	CONV2D, LSTM, MLP	YES	FIXED	YES	YES	YES	YES	Bruteforce
TimeLoop	Spatial	Runtime, Energy	YES	DeepBench, CNNs	YES	YES	YES	YES	YES	YES	Brute-force, random sampling
Marvel	Spatial	Runtime, Energy	YES	Any MDC Conformable	YES	YES	YES	YES	YES	YES	Decoupled

Figure 5.11: Comparison of Marvel with prior compiler approaches for spatial accelerators (mRNA [139], Zhang et al. [132], Ma et al. [131], Auto-TVM [76], dMazeRunner [135], Interstellar [136], TimeLoop [133]) for the mapping space exploration of DNN operators. Our approach (Marvel) supports any operator conformable with the MDC notation.

5.9 Summary

In this chapter, we provide a formal understanding of DNN operators whose mappings on flexible templated spatial accelerators can be described in the MDC notation by introducing a set of rules over the abstract loop nest form of the operators. Furthermore, we introduce a transformation for translating mappings into the MDC notation for exploring the mapping space. Then, we also proposed a decoupled off-chip/on-chip approach that decomposes the mapping space into off-chip and on-chip subspaces, and first optimizes the off-chip subspace followed by the on-chip subspace. We implemented our decoupled approach in a tool called *Marvel*, and a major benefit of our approach is that it is applicable to any DNN operator conformable with the MDC notation for any templated flexible spatial accelerator. Our approach reduced the search space of CONV2D operators from four major DNN models from 9.4×10^{18} to $1.5 \times 10^8 + 5.9 \times 10^8 \approx 2.1 \times 10^8$, which is a reduction factor of ten billion (Table 5.4), while generating mappings that demonstrate a geometric mean performance improvement of $10.25\times$ higher throughput and $2.01\times$ lower energy consumption compared with three state-of-the-art mapping styles from past work.

In the next chapter, we focus on a specific instantiation of spatial architectures for machine learning applications, i.e., Xilinx AI Engine, and we describe our advances required in compiler analysis, transformations, and code generation required to generate high-performant code for tensor convolutions on the AI Engine.

CHAPTER 6

VYASA: A HIGH-PERFORMANCE VECTORIZING COMPILER FOR TENSOR CONVOLUTIONS ON THE XILINX AI ENGINE

6.1 Abstract

Xilinx’s AI Engine is a recent industry example of energy-efficient vector processing that includes novel support for 2D SIMD datapaths and shuffle interconnection network. The current approach to programming the AI Engine relies on a C/C++ API for vector intrinsics. While an advance over assembly-level programming, it requires the programmer to specify a number of low-level operations based on detailed knowledge of the hardware. To address these challenges, we introduce *Vyasa*, a new programming system that extends the Halide DSL compiler to automatically generate code for the AI Engine. We evaluated *Vyasa* on 36 CONV2D and 6 CONV3D workloads, and achieved geometric means of 7.6 and 23.3 MACs/cycle for 32-bit and 16-bit operands (which represent 95.9% and 72.8% of the peak performance respectively). For 4 of these workloads for which expert-written codes were available to us, *Vyasa* demonstrated a geometric mean performance improvement of 1.10 \times with 50 \times smaller code relative to the expert-written codes.

6.2 Introduction

It is widely recognized that a major disruption is under way in computer hardware as processors strive to extend, and go beyond, the end-game of Moore’s Law. Unlike previous generations of hardware evolution, these “extreme heterogeneity” systems will have a profound impact on future software. As part of these trends, there is a strong resurgence of interest in improving vector processing (SIMD) units due to the significant energy efficiency benefits of using SIMD parallelism. These benefits increase with widening SIMD vectors, reaching vector register lengths of 2048 bits in the scalable vector extension of the Armv8 architecture [48]. Furthermore, there is an emphasis on specializing SIMD units to further improve energy efficiency benefits for specific domains such as Machine learning, Computer Vision, and 5G Wireless. An important specialization, which is referred to as “2D vector SIMD datapath” [31, 32, 33], is the ability of each vector lane to execute more than one scalar operation and to chain the results from one operation to another. Another specialization includes the removal of expensive data permutation units (e.g., shuffle units) [34, 35] and instead introduce sophisticated, programmable intercon-

nection networks (a.k.a shuffle networks) between the SIMD datapath and vector register file to support the required data permutation patterns [36, 33].

A recent industry example with these specializations is the Xilinx Versal AI Engine [49], a high-performance VLIW SIMD core which can deliver performance comparable to traditional FPGA solutions for Computer Vision, Deep Learning, and 5G wireless domains, but with 50% less power consumption and up to eight times more compute capacity per silicon area [49]. AI Engine cores are tightly integrated with programmable logic in Xilinx Versal ACAP devices to form a seamless heterogeneous compute platform [162, 7] applicable to a wide variety of HPC applications. Furthermore, the Versal AI Engine series VC1902 has a total of 400 AI Engines that together delivers a peak performance of 6.4 TOPS, 25.6 TOPS and 102.4 TOPS for 32-bit, 16-bit, and 8-bit operands, respectively [7].

Tensor convolution is a widely used mathematical operation in these domains, and it is becoming increasingly important with the rise of its use in image processing workflows [70, 71, 72] and with the proliferation of deep learning models [3, 66, 67, 68] in data centers, edge, and mobile devices. There has been a lot of prior work in optimizing tensor convolutions for a variety of target hardware devices such as CPUs [70, 71, 163], GPUs [70, 164, 163], FPGAs [132, 131, 139, 165, 163], and Dataflow accelerators [139, 166, 167, 43]. However, even for well understood applications like convolution, generating the best code for new high-performance processor architectures from high-level descriptions can be challenging. This work demonstrates the ability to automatically optimize tensor convolutions for the AI Engine and to obtain close to the peak performance for various workloads while using a high-level programming model, rather than low-level C/C++ intrinsics.

Challenges. Achieving peak performance on the AI Engine requires leveraging several architectural features to maximize vector datapath occupancy during program execution. Unlike standard SIMD architectures which operate on 1D vectors, the AI Engine architecture includes 2D vector operations for some datatypes which conceptually implement the fusion of several 1D vector operations. Also, unlike other architectures, the AI Engine doesn't implement direct support for unaligned loads, scalar broadcasts, and data manipulation operations. Instead, the AI Engine architecture includes a novel shuffle network which selects desired elements of a vector register for a vector operation instead of explicitly shuffling and storing them into another vector register. In order to effectively leverage these features, the layout of data in memory must match the capabilities of the shuffle network.

Existing AI Engine compilers do not perform auto-vectorization, leaving it to expert programmers to explicitly write high-performance vector code using architectural intrinsic functions. Optimizing programs in this way can be time-consuming even for experts. At

the same time, there are a wide variety of tensor convolution operators in common use, for instance, deep neural networks may contain regular 2D convolutions, depth-wise convolutions, and point-wise convolutions. Even within the same network, the shape of tensor data can vary radically between the early and late layers in DNN models. We find that no single optimization strategy is an optimal choice for all these scenarios. Reducing the need for manual optimization and quickly adapting to new tensor operations through automatic optimization avoids these problems.

With all these challenges, the overall goal of our work is *to automate the generation of high-performance vector code for tensor convolutions based on their variations and shapes, while exploiting the unique capabilities of the Xilinx AI Engine without requiring manual effort in development and tuning*. Achieving this goal requires significant loop-level reuse analysis, code transformation, and data-layout transformation, along with optimized low-level code generation taking into account the shuffle network and memory optimizations such as vector register reuse (including partial reuse) [168, 169].

The main technical contributions of this work are briefly described below:

- We introduce a new domain-specific intermediate representation called *Triplet* to symbolically capture the loop body of a tensor convolution, and to simplify analyses and transformations required to generate high-performance code for the AI Engine.
- We propose a novel multi-step compiler approach which includes analyses and transformations to 1) exploit the 2D SIMD datapath by identifying multiple 1D logical vector operations that can be legally fused, 2) realize unaligned loads, scalar broadcasts, data manipulation using the shuffle network, 3) improve memory utilization by performing vector register reuse and also loop optimizations, and 4) generate code that is more amenable to enabling VLIW instruction scheduling for the AI Engine.
- We created a new tool, *Vyasa*¹, to implement our multi-step compiler approach. *Vyasa* is built on the Halide framework [70] and includes extensions needed for the AI Engine that are not supported by Halide. Given a tensor convolution specification in the Halide language and workload sizes, *Vyasa* generates high-performance C-code with vector intrinsics for the AI Engine.
- We evaluated *Vyasa* on 36 CONV2D and 6 CONV3D workloads using a cycle-accurate simulator². Our results show geometric means of 7.6 and 23.3 MACs/-

¹*Vyasa* means “compiler” in the Sanskrit language, and also refers to the sage who first compiled the Mahabharata.

²Since the AI Engine architecture was developed for real-time processing applications which require deterministic performance, the simulator results are reliably correlated with actual performance of the AI

cycle for 32-bit and 16-bit operands (which represent 95.9% and 72.8% of the peak performance respectively). For four of these workloads for which expert-written implementations were available to us, Vyasa achieved a geometric mean performance improvement of 1.10× from Halide code that is around 50× smaller than the expert-written C/C++ code.

6.3 Background

In this section, we start with a brief overview of tensor convolutions, and then we briefly summarize the key architectural features of the Xilinx Versal AI Engine.

6.3.1 Tensor Convolutions

A convolution is a mathematical operation which computes the amount of overlap of a function g as it is shifted over another function f , and it is symbolically represented as $f \circ g$. In this section, we restrict our attention to describing CONV2D, a popular convolution operator widely used in Deep learning [64, 65, 3, 66, 67, 68] and Computer Vision [69, 70, 71, 72]. In these domains, the function f and g are referred to as the “input” tensor (a.k.a image/activations) and “weight” tensor (a.k.a filters/kernels), respectively. The CONV2D deals with three four-dimensional tensors, i.e., Output (O), Weight (W), and Input (I), whose dimensions are described below.

Tensor	Dim1	Dim2	Dim3	Dim4
Output (O)	Width (X)	Height (Y)	Channels (K)	Batch (N)
Weight (W)	Width (R)	Height (S)	Channels (C)	Batch (K)
Input (I)	Width (X')	Height (Y')	Channels (C)	Batch (N)

The mathematical expression of the CONV2D operations is shown below, where f refers to stride factor.

$$O(x, y, k, n) = \sum_c^C \sum_s^S \sum_r^R W(r, s, c, k) \times I(x \times f + r, y \times f + s, c, n)$$

The convolutions used in Computer Vision are special cases of the CONV2D operator, where each tensor has only the first two dimensions (width and height) and stride factor set to one. However, there exist a wide variety of filter sizes (ranging from 2 to 11) used in Engine hardware.

many different image processing operators, such as Gaussian smoothing and edge detection [69].

A wide variety of other specialized variations of the CONV2D operator are used in Convolutional Neural Networks such as point-wise, depth-wise separable, and spatially separable convolutions. These variations can be viewed as constraints on the regular CONV2D operator, and are shown below.

Operator	Constraints on CONV2D
Point-wise (PW)	Filter width = Filter height = 1
Fully-connected (FC)	Filter width = Input width Filter height = Input height
Spatially separable (SS)	Filter width = 1 or Filter height = 1
Depth-wise separable (DS)	Input channels = Filter channels = 1

Even though we briefly described the CONV2D operator and its variations, our approach is applicable to other convolution operators such as CONV1D and CONV3D.

6.3.2 Xilinx AI Engine

Driven by the performance and energy efficiency requirements of many computing applications, Xilinx introduced Versal Advanced Compute Acceleration Platform (ACAP) [162, 7], a fully software-programmable, heterogeneous compute platform. The Versal platform consists of three types of programmable processors – Scalar Engines (CPUs), Adaptable Engines (Programmable Logic), and an array of Intelligent Engines (AI Engines) [7]. In this work, we focus on AI Engines, which are specialized SIMD and VLIW high-performance processors for compute-intensive applications such as computer vision, machine learning workloads, and 5G wireless. AI Engines are highly energy efficient compared to FPGAs and can deliver up to 8X silicon compute density at 50% the power consumption of traditional FPGA solutions [49].

An AI Engine includes a 2D SIMD datapath for fixed-point vector operations (our focus), a 1D SIMD datapath for floating-point vector operations, and a scalar unit for scalar operations. Each AI Engine also has access to 128KB scratchpad (a.k.a data/local) memory, a 16KB program memory, and a 256B vector register file (a total of 16 registers with each size being 128 bits). These high-performance AI Engines are programmed using the C/C++ programming language with optional pragmas. A simplified overview of the key architectural features of the AI Engine core is shown in Figure 6.1, and these features are briefly described below.

1) Two-dimensional SIMD Datapath. The fixed-point vector unit of the AI Engine is a two-dimensional SIMD datapath, and vector operations on the 2D SIMD datapath are

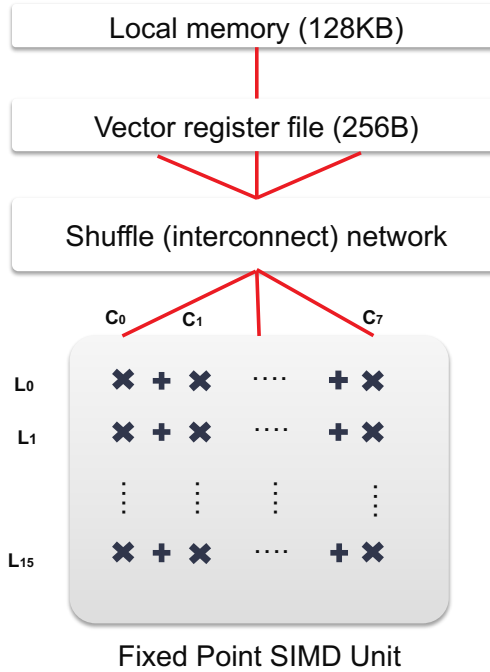


Figure 6.1: A pictorial overview of the key architectural features of the Xilinx AI Engine, i.e., 2D vector SIMD datapath and shuffle network.

described using *lanes/rows* and *columns*. The number of lanes corresponds to the number of output values generated from the vector operation. The number of columns is the number of operations that are done per output lane, with each of the results being reduced together. This technique of executing back to back dependent scalar operations along a vector lane is popularly known as operation chaining [31] and can improve energy efficiency by not writing intermediate values back to the register file. Furthermore, the number of columns is dependent on the operand precision. Operations on 32-bit types are organized as 8 lanes with 1 column, without internal reduction. Operations on 16-bit types are organized as either 16 lanes with 2 columns or 8 lanes with 4 columns. Operations on 8-bit types are organized as 16 lanes with 8 columns. As a result, the 2D datapath can perform either 8 MACs on 32-bit inputs, 32 MACs on 16-bit input, or 128 MACs on 8-bit input per cycle.

2) Shuffle network. A key novelty of the AI Engine architecture is its *shuffle network*, a flexible interconnection network between the 2D SIMD datapath and vector register file to allow flexible data selection from the input vector registers for the multipliers of each lane and column of the SIMD datapath. The ability to configure the shuffle network for each vector operation is exposed to programmers via the arguments of the vector intrinsic functions. Unlike the data manipulation units in traditional SIMD units, the data selection using the shuffle network over a vector register can only be used during a vector operation. The granularity of data selection using the shuffle network on the vector registers is 32b, and

so the network allows full flexibility for making data selection, replication, and permutation on vectors of 32b data types. However, for data types of smaller sizes such as 16b and 8b data types, the shuffle network imposes further constraints on data selection.

Vector loads and stores in the AI Engine must be aligned to 128-bit data memory boundaries. The AI Engine does not implement unaligned loads or scalar broadcasts. Instead, these operations are typically realized/implemented using a combination of aligned loads and configuration of the shuffle network.

3) VLIW capabilities. The AI Engine has support for very long instruction word (VLIW) that can provide up to 6-way instruction parallelism to hide long instruction latencies. The VLIW instruction includes two scalar operations, two vector load operations, one vector store operation, and one fixed/floating-point vector operation. The AI Engine compilers have support for automatic software pipelining [50] of innermost loops to exploit instruction-level parallelism.

6.4 Our Approach

In this section, we introduce our approach to generating high-performance vector code for a given high-level specification of tensor convolution and its workload sizes that fit into a single AI Engine’s data memory. These vector codes are intended to execute on a single AI Engine and will be integrated by a high-level compiler to run larger tensor convolutions across multiple AI Engines. Our approach is summarized in Figure 6.2 and is implemented in a tool called Vyasa. The tool is developed as an extension to the Halide framework [70]. Our approach begins with an auto-tuner taking the specification of a tensor convolution in the Halide language and also the corresponding workload sizes. Then, the auto-tuner iterates through each possible schedule in the space of loop transformations and data-layouts, and invokes our multi-step compiler approach to generate high-performance vector c-code corresponding to the schedule. Then, our approach evaluates the generated code using a cycle-accurate simulation of the AI Engine, and chooses the best one among all schedules to finally emit as the performant output code.

Our multi-step compiler approach starts from the specification of a tensor convolution, a schedule from the auto-tuner, and workload sizes. It consists of the following steps:

1. Transforming the loop body of the tensor convolution operation in the Halide IR (after lowering) into our symbolic *triplet* representation for convenience in doing analyses, transformations, and code generation,
2. Performing ‘lazy stores’ optimization by accumulating all partial (intermediate) results of an output before generating a store to reduce the memory traffic,

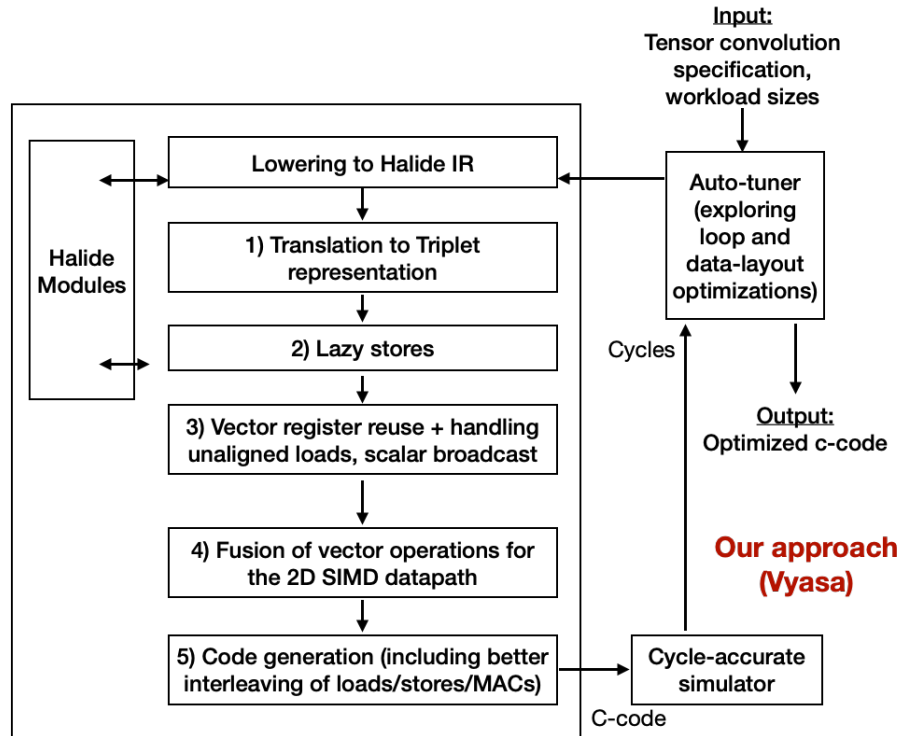


Figure 6.2: Workflow of our approach (Vyasa) which is implemented as an extension to the Halide framework [70].

3. Exploiting vector register reuse, and realizing unaligned loads and scalar broadcast operations using the shuffle (interconnection) network,
4. Identifying suitable 1D logical vector operations (multiplications) that contribute to same output through accumulation/reduction and fusing them into operations matching the 2D SIMD datapath,
5. Interleaving load and store operations with vector operations to make it easy for the AI Engine compilers to perform VLIW instruction scheduling,
6. Generating C-code with vector intrinsics.

6.4.1 Translating into Triplet Representation

In general, tensor convolutions are specified/implemented as multi-dimensional perfectly nested loops, where each statement of the loop body has two aspects – 1) A group of multiply-and-accumulate (MAC) operations over input and weight tensors, and 2) An update (reduction) operation to the output tensor. Since each statement in the convolution loop body performs a reduction operation and the reduction is commutative, the order of each statements doesn't impact its correctness. Hence, a representation holding information for

```

1 Buffer<int16> I(W,H); Buffer<int16> W(4,3);
2 Var x, y; RDom r(4, 3); Func 0; //output

4 //(a) Description of the convolution computation
5 O(x,y) += W(r.x, r.y) * I(x+r.x, y+r.y);

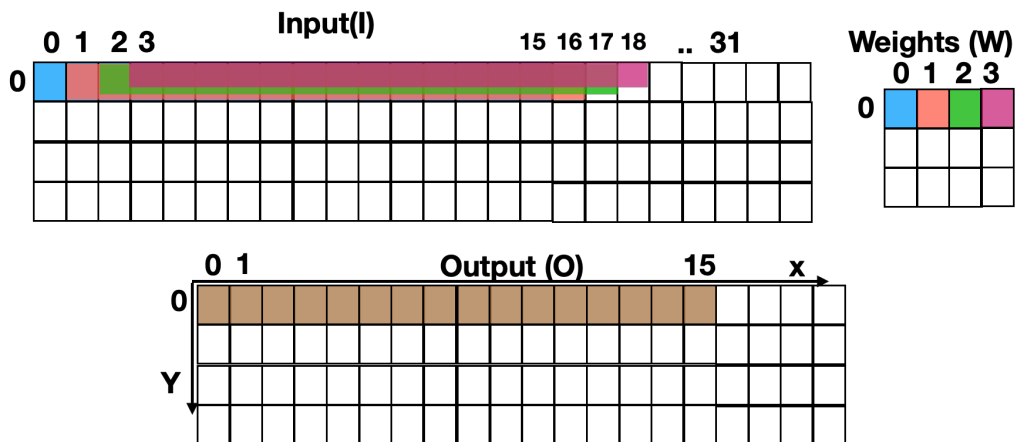
7 //(b) A sample schedule: Unrolling reduction loops
8 //Vectorizing loop corresponding to image width
9 O.update().unroll(r.x, 4).unroll(r.y,3)
10     .vectorize(x, 16);

12 //(c) Intermediate code after lowering
13 for y:
14   for x: (vectorized)
15     O(x:x+15,y) += W(0,0) * I(x:x+15,y);
16     O(x:x+15,y) += W(1,0) * I(x+1:x+16,y);
17     O(x:x+15,y) += W(2,0) * I(x+2:x+17,y);
18     O(x:x+15,y) += W(3,0) * I(x+3:x+18,y);
19     .....

```

Figure 6.3: Algorithmic description of the convolution of a 4x3 filter over an input 2D image in the Halide language [70]. A(a:b,c) is a short hand vector notation for denoting a contiguous slice from A(a,c) to A(b,c) in one direction.

each statement about the two major aspects described above is sufficient to capture the body precisely. We call this representation a “triplet” since it holds information about the access patterns of the two operands of each multiplication and the update operand of each statement symbolically.



$$O(0:15, 0) = I(0:15,0) * W(0,0) + I(1:16,0) * W(1,0) + I(2:17,0) * W(2,0) + I(3:18,0) * W(3,0) + \dots$$

Figure 6.4: A pictorial overview of the convolution of 4x3 filter based on the schedule described in Figure 6.3(b) at the loop iterations $x = 0$ and $y = 0$.

We consider the convolution of a filter with size 4×3 on an input with size $W \times H$ as a running example (shown in Figure 6.3(a)) to illustrate each step of our compiler approach. A sample schedule for the above convolution is shown in Figure 6.3(b), which refers to unrolling loops corresponding to filter dimensions $(r.x, r.y)$ and vectorizing the loop- x with vector length as 16. A pictorial overview of the computation at the loop iterations $x = 0, y = 0$ is shown in Figure 6.4.

After lowering the convolution specification using the schedule into the Halide IR, the first step in our approach is to translate the convolution loop body into our triplet representation. For instance, the triplet representation of the loop body in Figure 6.3(c) is shown in Table 6.1, where each row in the table symbolically captures the access patterns of multiplication operands and update operands of a statement in the loop body.

Table 6.1: Triplet representation of the loop body in Figure 6.3(c)

Update Operation Operand	MAC Operations	
	Operand1	Operand2
$O(x:x+15, y)$	$W(0, 0)$	$I(x:x+15, y)$
$O(x:x+15, y)$	$W(1, 0)$	$I(x+1:x+16, y)$
$O(x:x+15, y)$	$W(2, 0)$	$I(x+2:x+17, y)$
$O(x:x+15, y)$	$W(3, 0)$	$I(x+3:x+18, y)$
..

6.4.2 Lazy Stores Optimization

An approach to code generation based on the triplet representation involves generating a vector store for each row of the representation. But this code generation can result in immediately writing multiplication results to the data memory causing more traffic. We introduce “lazy stores” optimization to delay writing the multiplication results of an output until there are no operations that can contribute to output. The optimization works by grouping all the rows of the triplet representation contributing to the same output. The benefits of the optimization can be observed in the presence of multiple statements in the loop body contributing to the same output. An example of such behavior is seen in Table 6.1, where all the statements contribute to the same output $(O(x:x+15, y))$, and all these statements can be grouped into a single group (Table 6.2). Now, the code generation involves generating a single vector store for each group, instead of generating for each row of the triplet representation.

Table 6.2: Triplet representation after the lazy stores optimization

Update Operation Operand	MAC Operations	
	Operand1	Operand2
O(x:x+15, y)	$\bar{W}(0, 0)$	I(x:x+15, y)
	$\bar{W}(1, 0)$	I(x+1:x+16, y)
	$\bar{W}(2, 0)$	I(x+2:x+17, y)
	$\bar{W}(3, 0)$	I(x+3:x+18, y)

6.4.3 Exploiting Vector Register Reuse & Realizing Unaligned Loads and Scalar Broadcast

Our approach leverages the AI Engine architecture’s unique shuffle network to realize unaligned vector loads and scalar broadcast operations which are common in vectorization of tensor convolutions. Our approach further uses the network to exploit the vector register reuse opportunities.

1) Realizing unaligned vector loads. Simple vectorization of tensor convolutions often result in unaligned vector loads. For example, if the vector load (I(x:x+15, y)) in Figure 6.3 is aligned to the boundary, then the subsequent vector loads such as I(x+1:x+16, y) are unaligned. Prior work on vectorization for SIMD architectures having no unaligned load/store support address this by generating two adjacent aligned loads covering the required load and using data manipulation/shuffle (register-to-register) instructions to realize an unaligned vector load [169]. Since the AI Engine architecture doesn’t support unaligned loads or shuffle instructions, an alternative solution is necessary. Our approach leverages the AI Engine architecture support for grouping vector registers into a larger vector register. Then, our approach constructs a larger aligned vector load which subsumes the required unaligned load and selects the data corresponding to the original unaligned vector load using the shuffle network. For instance, the unaligned vector load I(x+1:x+16, y) can be realized through a larger aligned vector load I(x:x+31, y) and appropriate data selection parameters during vector operations on the load.

2) Exploiting vector register reuse. Tensor convolutions often exhibit significant data reuse between vector loads. For instance, the two vector loads I(x:x+15, y) and I(x+1:x+16, y) have 15 data elements in common. Exploiting vector register reuse by reusing those common elements instead of fetching again from the data memory is important to reduce memory traffic and achieve better performance. Our approach groups individual vector loads having such reuse and constructs a larger aligned vector load that subsumes the individual vector loads having reuse. During the vector operations, the individual vector loads are realized through appropriate data selection on the larger vector using the shuffle network.

Our approach implements the above idea by constructing a *reuse graph*, an undirected

graph where each node denotes a vector load in the triplet representation and an edge is constructed between two nodes if they have at least one common element between them, i.e., presence of a reuse. In the current approach, we don't associate any weights to the edges of the reuse graph, and we briefly commented on the benefits of adding weights later in the section. The reuse graph corresponding to the vector loads of the tensor I in Table 6.2 is shown in Figure 6.5, for instance, nodes $I(x:x+15, y)$ and $I(x+1:x+16, y)$ corresponds to two vector loads and the edge between them denotes the presence of common elements/reuse.

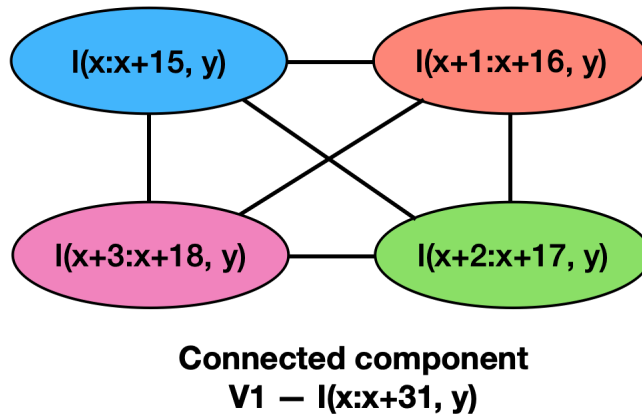


Figure 6.5: Reuse graph corresponding to the vector loads of the tensor I in Table 6.2, and its connected components to construct larger vector loads.

After constructing the reuse graph, our approach identifies connected components in the reuse graph, where each component represents a larger vector load that subsumes the individual vector loads in that component. For instance, the connected component $I(x:x+31, y)$ in Figure 6.5 represents a larger vector load subsuming the vector loads $I(x:x+15, y)$, $I(x+1:x+16, y)$, $I(x+2:x+17, y)$, and $I(x+3:x+18, y)$. Since our approach hasn't yet fused the logical 1D vector operations to exploit all columns of the 2D SIMD datapath, computing the data selection parameters is deferred to a later step (Section 6.4.4). After replacing each individual vector load with its corresponding larger load, the running example results in having only three larger vector loads instead of twelve individual vector loads for the tensor I.

Our approach currently reports a compilation error if the size of a connected component is larger than the maximum logical vector register size, because our approach maps a connected component to a larger aligned vector load subsuming individual vector loads. However, our approach can be enhanced by partitioning the connected component by making sure that the individual vector loads are only part of a single partition of the connected component. The edges of the reuse graph can be annotated with the common elements as the weights to further enhance the partitioning approach.

3) Realizing scalar broadcasts. Similar to unaligned vector loads, vectorization of tensor convolutions involve scalar operands and require the support for scalar to vector broadcast operation, for, e.g., the scalar operand $W(0, 0)$ in Table 6.2. A naive approach to realize the broadcast operation of a scalar operand is by loading an aligned vector covering the operand and then using the shuffle network to select the the operand for all the lanes. A downside of the above approach is that it may result in loading an entire vector while using only one value fetched from memory.

The scalar operands in the tensor convolutions typically exhibit significant spatial locality, e.g., the scalar operands such as $W(0, 0)$ and $W(1, 0)$ in Table 6.2 are contiguous in the data memory. Similar to our approach in exploiting vector register reuse, we construct another reuse graph to identify scalar operands that are adjacent in data memory and can be subsumed as part of a single vector load. For instance, the operands $W(0, 0)$, $W(1, 0)$, $(2, 0)$, $W(3, 0)$ can be realized over a vector load (say V2) of $W(0:7, 0)$.

We represent the data selection of a set of values from a vector register using the shuffle network during a vector operation as $SELECT(V, \{s_{ij}\})$ where V represents the vector register and s_{ij} denotes the index of the required element in the register V for the i^{th} lane and j^{th} column multiplier in the 2D SIMD datapath. Our approach defers the computation of data selection parameters to the next step. The triplet representation after realizing the unaligned loads, scalar broadcasts, and exploiting vector register reuse is shown in Table 6.3.

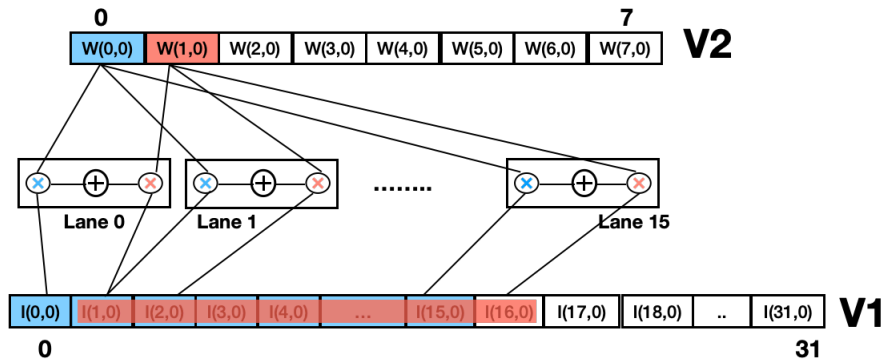
Table 6.3: Triplet representation after addressing unaligned loads, scalar broadcast, and exploiting vector register reuse.

Update Operation Operand	MAC Operations	
	Operand1	Operand2
$0(x:x+15, y)$	$SELECT(V2, \{ \})$	$SELECT(V1, \{ \})$
	$SELECT(V2, \{ \})$	$SELECT(V1, \{ \})$
	$SELECT(V2, \{ \})$	$SELECT(V1, \{ \})$
	$SELECT(V2, \{ \})$	$SELECT(V1, \{ \})$

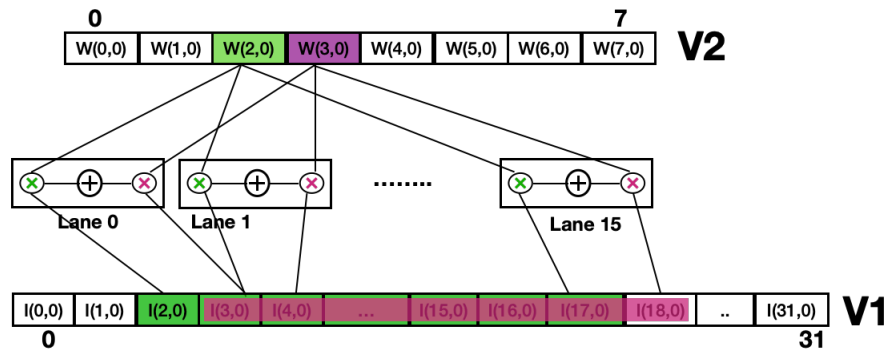
6.4.4 2D Vector SIMD Datapath

A key distinguishing feature of the AI Engine relative to the traditional SIMD units is the presence of a two-dimensional SIMD datapath which performs reduction across all columns of a SIMD lane. A single 1D logical vector operation can occupy a single column of 2D datapath, but the vector operations on the 2D SIMD datapath require using all the columns of the datapath and don't allow partial utilization. Hence, our approach identifies and logically groups (fusing) all suitable 1D logical vector operations that con-

tribute to the same output through accumulation/reduction and use the same set of vector register operands. The identification is done by searching in the triplet representation for operations having the same update operand and the same set of vector registers as multiplication operands. Finally, our approach partitions the logical groups based on the number of columns available for the given operand type and also constraints imposed by the shuffle network on the data selection over vector register operands. If the data selection required for the operands of fused vector operations is incompatible with the constraints of the shuffle network, then our approach generates a compilation error and prunes that candidate code variant.



$$\mathbf{a) } \mathbf{O(0:15, 0) = W(0, 0) * I(0:15, 0) + W(1, 0) * I(1:16, 0)}$$



$$\mathbf{b) } \mathbf{O(0:15, 0) += W(2, 0) * I(2:17, 0) + W(3, 0) * I(3:18, 0)}$$

Figure 6.6: An overview of the two fused vector operations (a and b) over the vector registers V1, V2 for input and weights, respectively of the running example shown in Table 6.2 at $x=0$ and $y=0$. The shuffle network of the AI Engine helps each multiplier of the 16 lanes and 2 columns of the 2D SIMD datapath to choose required elements from the vector registers.

There are four valid fusible logical 1D operations for each row of the filter in Table 6.2, our approach groups them into two fused vector operations whose overview is described in Figure 6.6. Furthermore, the triplet representation after fusing the logical 1D vector operations is shown in Table 6.4.

Table 6.4: Triplet representation after fusing the logical 1D vector multiplications and finding the data selection parameters

Update operation operand	MAC Operations (after fusing)	
	Operand1	Operand2
O(x:x+15,y)	SELECT(V2, {j})	SELECT(V1, {i+j})
	SELECT(V2, {j+2})	SELECT(V1, {i+j+2})

6.4.5 Code Generation

Our approach extends the code generation capabilities in the Halide [70] by implementing a code generator for the triplet representation to generate explicitly vectorized code using AI Engine intrinsic functions. A naive approach to code generation can be implemented by first emitting all vector loads, followed by all vector MAC operation, and then finally all vector stores. However, this naive approach results in variables (loads) having large live ranges, possibly leading to register spills and preventing software pipelining. Furthermore, optimization of memory accesses can be challenging for the downstream compilers only given the generated intrinsic code. Hence, our approach reorders memory accesses and interleaves them with vector MAC operations during the code generation process to reduce the live range of each variable. This process is relatively easy given the information about memory access patterns in Halide and helps the downstream compilers to improve packing of stores, loads, and vector MACs into VLIW instructions. A snippet of the final code generated by our approach with interleaving of loads, vector operations, and stores over the running example is shown in Figure 6.7.

6.4.6 Auto-tuner

Steps 1-5 in our multi-step compiler approach generates the vectorized code for a given specification of tensor convolution, a schedule from the auto-tuner, and workload sizes. The auto-tuning capabilities of the Halide framework support only multi-staged pipelines [170, 171], but our focus is only on a single stage for the convolution. Hence, we implemented custom auto-tuner in our approach exploring all possible schedules to find the best schedule for a given convolution specification and the workload sizes. The search space of schedules includes loop nest and data-layout optimizations.

Search space. The space of loop transformations includes loop interchange, loop unroll and jamming, and the choice of loop for vectorization. The space of data-layout optimizations includes dimension permutation and data tiling.

Exploration. Our approach applies the following pruning strategies: 1) unrolling of reduction loops to avoid memory traffic in writing and reading intermediate (partial) results, and

```

1 //Generated code
2 for(int y=0; y < Y; y++)
3   for(int x=0; x < X; x+=16) {
4     V1 = VLOAD(I, x:x+31, y);
5     V2 = VLOAD(W, 0:7);
6     V3 = VMUL(V2, SELECT(V2, {j}),
7              V1, SELECT(V1, {i+j}));
8     V3 = VMAC(V3, V2, SELECT(V2, {j+2}),
9              V1, SELECT(V1, {i+j+2}));
10    V4 = VLOAD(I, x:x+31, y+1);
11    V3 = VMAC(V3, V2, SELECT(V4, {j+4}),
12            V1, SELECT(V1, {i+j}));
13    V3 = VMAC(V3, V2, SELECT(V4, {j+6}),
14            V1, SELECT(V1, {i+j+2}));
15    ....
16    VSTORE(V2, 0, x:x+15, y);
17  }

```

Figure 6.7: A snippet of the generated 16-bit vector code for the running example in Figure 6.3. VLOAD/VMUL/VMAC/VSTORE refers to vector load, vector multiplication, vector multiply-and-accumulate, and vector store. SELECT symbolically represents the data selection over a vector register for the i^{th} row and j^{th} column of 2D datapath multipliers.

2) applying bounds on the unroll and jam factors to avoid code size explosion (AI Engine has only 16KB program memory) and also to avoid longer compilation times. Our auto-tuner evaluates each point in the pruned search space by generating the vectorized C-code, compiling with the AI Engine compiler, and executing it using a cycle-accurate architecture simulator. With performance as the primary optimization goal, our approach obtained a geometric mean performance improvement of $1.10\times$ fewer cycles than the expert-written and tuned codes available for four workloads, showing that automatic exploration can find useful design points which are not obvious to humans.

The auto-tuner of our framework can be enhanced with extensions to our Marvel approach (Chapter 5) and MAESTRO cost model [134] to identify the performant mappings in the mapping space and use our multi-step compiler approach only for performant mappings to reduce overall auto-tuning time.

6.5 Experiments

We evaluated our approach over a total of 36 workloads involving a wide variety of operators and variations of CONV2D and CONV3D over two operand precisions (32-bit and 16-bit) on a single AI Engine. Each workload represents a unique combination of a convolution operation, tensor shapes, and operand precision. The configuration is shown in Table

6.5 and includes a 128KB local memory pre-loaded with all the data required for the evaluation of each workload. The configuration also includes a vector register file of size 256B (a total of 16 registers with each size as 128 bits) in between the SIMD datapath and the local memory. We used the AI Engine’s cycle-accurate simulator to evaluate the functionality and performance of our generated codes. We define the performance (MACs/Cycle) of an implementation of a tensor convolution as the total number of MAC operations in the convolution divided by the total number of execution cycles taken by the implementation.

Table 6.5: The AI Engine configuration used in our evaluation.

Parameter	32-bit	16-bit
2D SIMD data path	8 x 1	16 x 2
Peak compute	8 MACs/cycle	32 MACs/cycle
Scratchpad memory	128 KB @ 96B/cycle	
Scratchpad memory ports	32B 2 read and 1 write	
Vector register file	256 B	

6.5.1 CONV2D in Computer Vision

In the following experiments, we compare two experimental variants: 1) Code written by an expert (for 3×3 and 5×5 filters) available as part of the Xilinx’s AI Engine compiler infrastructure, 2) Code generated by our approach leveraging the auto-tuner. Both codes are designed to produce a 256×16 tile of a larger image. We observe from Figure 6.8 that our approach achieved a geometric mean performance improvement of 1.10× from the Halide codes compared with the available expert-written codes.

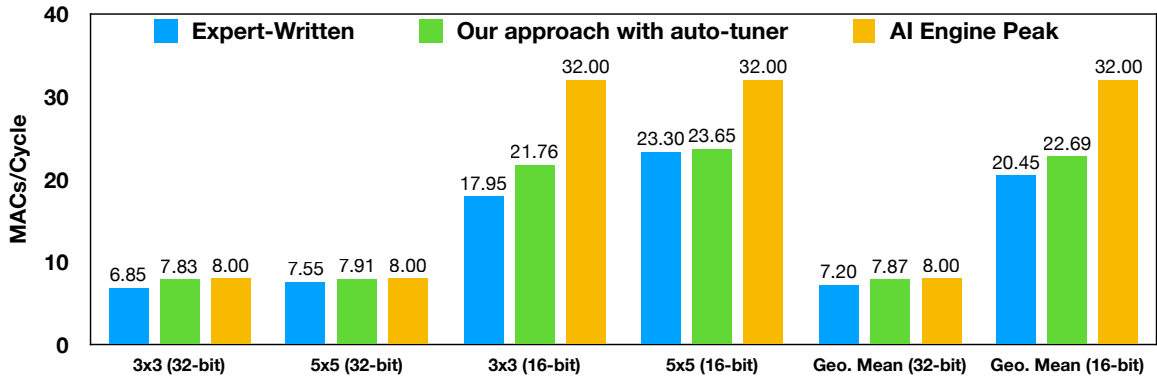


Figure 6.8: Comparison of our approach with auto-tuner against the available expert-written codes for CONV2D operation with 3×3 and 5×5 filters.

The auto-tuner of our approach was able to find better schedules than used in the expert-written codes (roof-line graphs for the workloads is shown in Figure 6.9), including non-unit unroll and jam factors along the image height (loop-y) dimension for better reuse.

These non-unit factors also enabled more opportunities in the loop body for the downstream compilers to perform better software pipelining. Furthermore, since workload sizes are also expressed in the Halide codes, our approach annotated the loops of generated codes with pragmas about the loop sizes to help the downstream compilers, especially helping the automatic software pipelining to accurately estimate the pre-ambule and post-ambule set up overheads and generate better VLIW code. Such overheads can be significant, particularly for tiled inner loops executed many times.

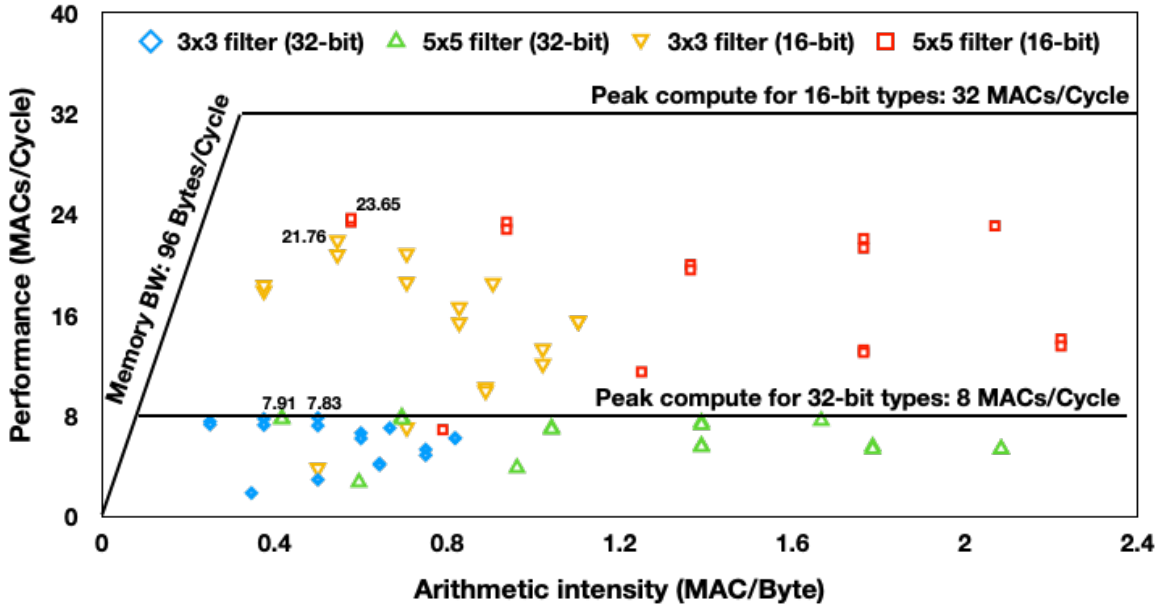


Figure 6.9: Roof-line graphs of four workloads considered in Figure 6.8, where each data point is a schedule explored by the auto-tuner.

In case of the 3x3 and 5x5 filters with 16-bit operands, the total number of fusible logical 1D vector multiplications corresponding to each row of the filters is an odd number. Hence, our approach padded the filters with an additional column to generate even number of fusible 1D operations and map onto the two columns present in the 2D SIMD datapath for 16-bit types. But, expert-written codes fused logical 1D vector multiplications corresponding to different rows of the filters, thereby avoiding the padding. This was accomplished by carefully merging the required input image data from different rows into a single vector register. Our approach currently doesn't exploit this optimization strategy and would require additional analysis to enable it. However, we see that the code generated using our approach is still able to perform better than the expert-written code by leveraging loop unroll and jam transformations.

In addition to the 3x3 and 5x5 filters, we have evaluated other filter sizes commonly used in Computer Vision applications. Table 6.6 presents those workload sizes, total MAC

Table 6.6: CONV2D workloads of Computer Vision used in our evaluation and optimal schedules from auto-tuner

Output (O) size	Weight (W) size	Input (I) size	#MACs	Optimal schedule from auto-tuner				
				Unroll and Jam factors				Loop order
				32-bit		16-bit		
				x	y	x	y	
256 x 16	2 x 2	264 x 17	16384	1	4	1	8	xy
	3 x 3	264 x 18	36864	1	4	1	2	xy
	4 x 4	264 x 19	65536	1	2	1	1	xy
	5 x 5	264 x 20	102400	1	2	1	1	xy
	6 x 6	264 x 21	147456	1	1	1	1	xy
	7 x 7	264 x 22	200704	1	1	1	1	xy
	8 x 8	264 x 23	262144	1	4	1	1	xy
	9 x 9	264 x 24	331776	1	4	1	1	xy
	10 x 10	264 x 25	409600	1	4	1	4	xy
	11 x 11	264 x 26	495616	1	4	1	4	xy

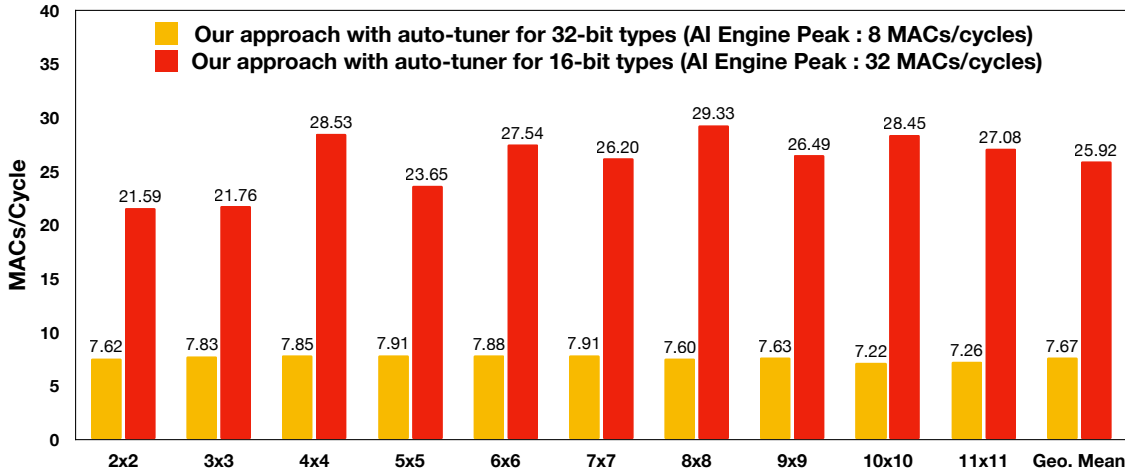


Figure 6.10: Performance of our approach generated codes for CONV2D workloads of Computer Vision over filter sizes from 2 to 11.

operations involved in each workload, and optimal schedules reported by the auto-tuner. We padded each non-even sized 16-bit filter with an additional column for evaluation, but we used the MACs obtained by the filter without padding while computing the performance (MACs/cycle). As can be observed from Figure 6.10, our approach achieved a geometric mean performance of 7.67 and 25.92 MACs/cycle for 32-bit and 16-bit types respectively for the workloads in Table 6.6. The auto-tuner chose the loop-x for vectorization for all the workloads, because it has more reuse opportunities and has larger number of iterations compared to the loop-y. The optimal unroll and jam factors are not the same for all the

workloads and also vary for different precisions of the same filter size. Even though increasing unroll and jam factors improve the reuse opportunities, but it often resulted in register spills after a threshold and also interfered with software pipelining of inner loops. Furthermore, larger unroll and jam factors along the loop-x resulted in larger connected components of the reuse graph and required larger vector register than the maximum possible (e.g., 1024b for 32-bit operands) in the hardware.

6.5.2 CONV2D in Deep Learning

Table 6.7: CONV2D workloads of deep learning used in our evaluation (variable names described in Section 6.3) and optimal schedules.

CONV type	Output (O) size (XxYxK)	Filter (F) size (RxSxCxK)	Input (I) size (X'xY'xC)	#MACs	Precision	Optimal schedules from the auto-tuner								
						Data layouts			Vector loop	SW loop	Unroll and Jam factors			Loop order
						O	W	I			x	y	k	
(REG)		3x3x8x16	144x4x8	294912	32-bit	XYK	(K/8)(C/8)SR(8)(8)	(C/8)Y'X'(8)	k	x	1	2	1	kyx
					16-bit	KYX	K(C/2)SR(2)	(C/2)Y'X'(2)	x	x	1	1	1	yxk
		5x5x8x16	144x6x8	819200	32-bit	KYX	KCSR	CY'X'	x	x	1	1	1	kyx
					16-bit	KYX	K(C/2)SR(2)	(C/2)Y'X'(2)	x	x	1	2	1	kyx
		7x7x8x16	144x8x8	1605632	32-bit	KYX	KCSR	CY'X'	x	x	1	2	1	kyx
					16-bit	KYX	K(C/2)SR(2)	(C/2)Y'X'(2)	x	x	1	2	1	kyx
(PW)	128x2x16	1x1x8x16	144x2x8	32768	32-bit	XYK	(K/8)(C/8)SR(8)(8)	(C/8)Y'X'(8)	k	x	1	2	1	kyx
					16-bit	YXK	(K/16)SR(C/2)(16)(2)	Y'X'C	k	k	1	2	1	xyk
(SS)		1x3x8x16	144x4x8	98304	32-bit	XYK	(K/8)(C/8)SR(8)(8)	(C/8)Y'X'(8)	k	p	1	2	1	kyx
					16-bit	KYX	K(C/2)SR(2)	(C/2)Y'X'(2)	x	x	1	2	1	kyx
		3x1x8x16	144x2x8	98304	32-bit	XYK	(K/8)(C/8)SR(8)(8)	(C/8)Y'X'(8)	k	x	1	1	1	kyx
					16-bit	YXK	(K/16)SR(C/2)(16)(2)	Y'X'C	k	k	1	2	1	xyk
(DS)		3x3x16x16	144x4x16	36864	32-bit	KYX	KCSR	CY'X'	x	x	1	2	1	kyx
					16-bit	KYX	KCSR	CY'X'	x	x	1	2	1	kyx
(FC)	4096x1x1	1x1x8x4096	16x1x8	32768	32-bit	XYK	(K/8)(C/8)SR(8)(8)	(C/8)Y'X'(8)	k	k	1	1	1	kyx
					16-bit	YXK	(K/16)SR(C/2)(16)(2)	Y'X'C	k	k	1	1	1	xyk

We considered a wide variety of CONV2D operations in the deep learning domain such as regular (REG) CONV2D over various filter sizes, point-wise (PW), spatially separable (SS), depth-wise separable (DS), and fully-connected (FC) operations. Table 6.7 presents those workload sizes (with unit batch size, i.e., $N = 1$), total MAC operations involved in each workload, and optimal schedules reported by the auto-tuner. Since the memory footprint of typical CONV2D operations don't fit into the local memory, we chose the similar output and input tensor memory footprint used in Table 6.6. As can be observed from Figure 6.11, our approach achieved a geometric mean performance of 7.67 and 22.53 MACs/cycle for 32-bit and 16-bit types respectively for the workloads in Table 6.7.

The auto-tuner chose either loop-x or loop-k for vectorizing the workloads, because the number of iterations of remaining loops are smaller than the vector length. The auto-tuner identified the vectorization along the loop-x to be beneficial for the REG-5x5, REG-7x7 workloads, because there exist more opportunities for vector register reuse (convolutional reuse) along the loop-x with the larger kernels sizes. But, for the workloads such as PW (REG-1x1), FC that have either less or no convolutional reuse along the loop-x, the

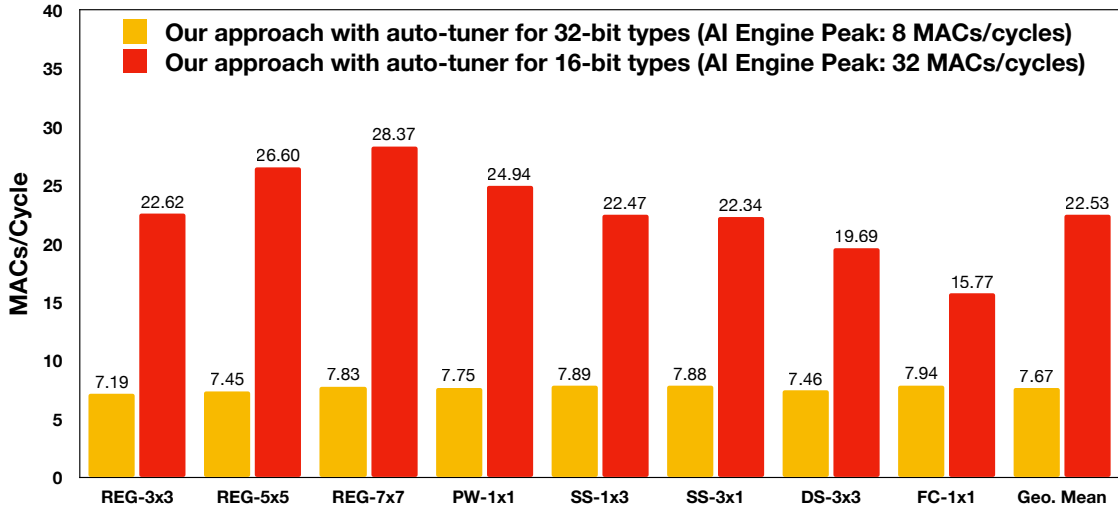


Figure 6.11: Performance of our approach generated codes for CONV2D workloads (shown in Table 6.7) of Deep Learning.

vectorization was performed on the loop-k.

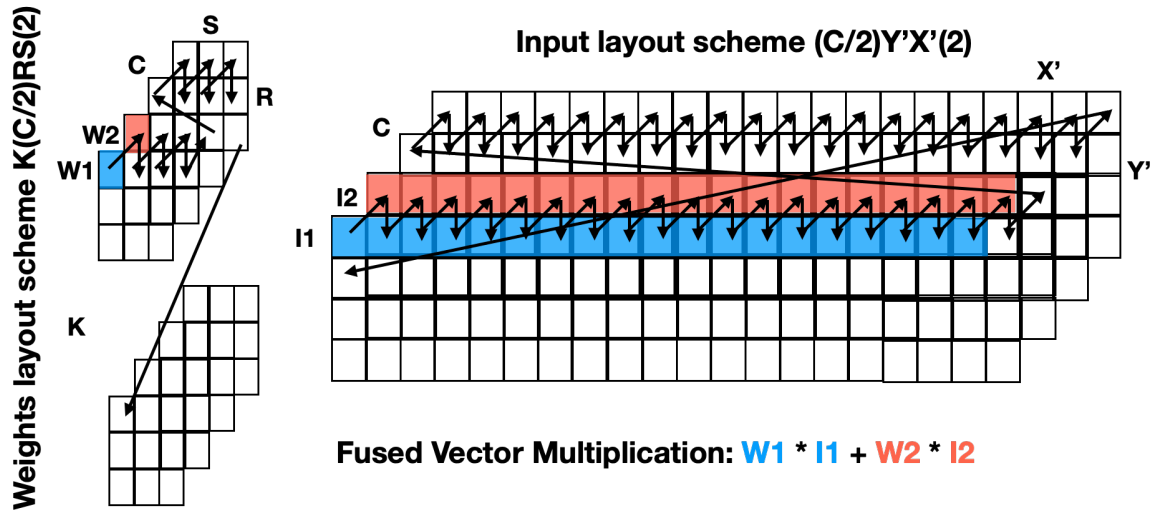


Figure 6.12: Data-layouts of input and weight tensors of the 16-bit REG-3x3 workload (Table 6.7), to enable the fusion of 1D logical vector multiplications along the channels, thereby avoiding the padding required for weights.

In these workloads, there exists an even number of fusible logical 1D vector multiplications corresponding to the filter channels, hence our approach didn't require any padding to the filter tensors (unlike in Table 6.6), except for the depth-wise CONV2D workload which has only one channel. However, the data-layouts of these workload tensors need to be modified to support the fusion of 1D logical vector multiplications along the channels. An example data-layout for the input and weights of the 16-bit REG-3x3 workload for the fusion along channels is shown in Figure 6.12, where the data-layout scheme for the input

tensor $(C/2)Y'X'(2)$ refers to first laying out a block of two channels followed by width, height, and remaining channels.

Along with the advantages of avoiding padding, data-layouts can be used for exploring better schedules as well. Such data-layout schemes over the workload tensors should respect two constraints: 1) The required number of data elements of each operand of the fused vector multiplication should fit into the maximum vector register size (e.g., 32 unique 16-bit input data elements for the vector multiplication in Figure 6.12 can fit into a 1024b vector register which is the maximum), and 2) The required data selection parameters over the vector register should respect the shuffle network constraints. Our auto-tuner was able to automatically explore a variety of such valid data-layout schemes in our evaluation. Although the resulting data-layouts can be implemented by the architecture, they can be rather complex and non-intuitive (e.g., $(K/16)SR(C/2)(16)(2)$ in Table 6.7). Manually identifying such a data layout and writing the corresponding intrinsic-based code is extremely challenging and error-prone, even for experts, thereby demonstrating the benefits of our automatic approach.

In Table 6.7, we see that the arithmetic intensity of the FC workload for the 16-bit is to the left-side of the inflection point of the roof-line graph of the AI engine, indicating memory-bound execution. This is expected, since the FC workload has little opportunity for data reuse within a single convolution operation. The workload peak performance based on its arithmetic intensity is 21.22 MACs/cycle, and our approach achieved 15.77 MACs/cycle or 75% of the workload peak.

6.5.3 CONV3D

In this evaluation, we focused on the simpler CONV3D workloads to further demonstrate the applicability of our approach. The output sizes in these workloads are the same as in the CONV2D workloads in Table 6.7, i.e., $168 \times 2 \times 16$, and the weight tensor sizes are $3 \times 3 \times 3$, $5 \times 5 \times 5$, and $7 \times 7 \times 7$ which are popular in the 3D CNN models [172, 173]. Since the number of fusible 1D logical vector multiplications corresponding to any dimension of the weight tensor in these workloads are odd, we have padded the weights with an additional column for each row. With this padding, our approach achieved a geometric mean performance of 7.55 and 21.60 MACs/cycle for 32-bit and 16-bit types, respectively shown in Figure 6.13.

Overall, our evaluation over all the workloads shows geometric means of 7.6 and 23.3 MACs/cycle for 32-bit and 16-bit operands (which represent 95.9% and 72.8% of the peak performance respectively). This difference in efficiency is not surprising, since it is more challenging to utilize two columns in the SIMD data path in the case of 16-bit operands, compared to a single column in the case of 32-bit operands. However, the absolute perfor-

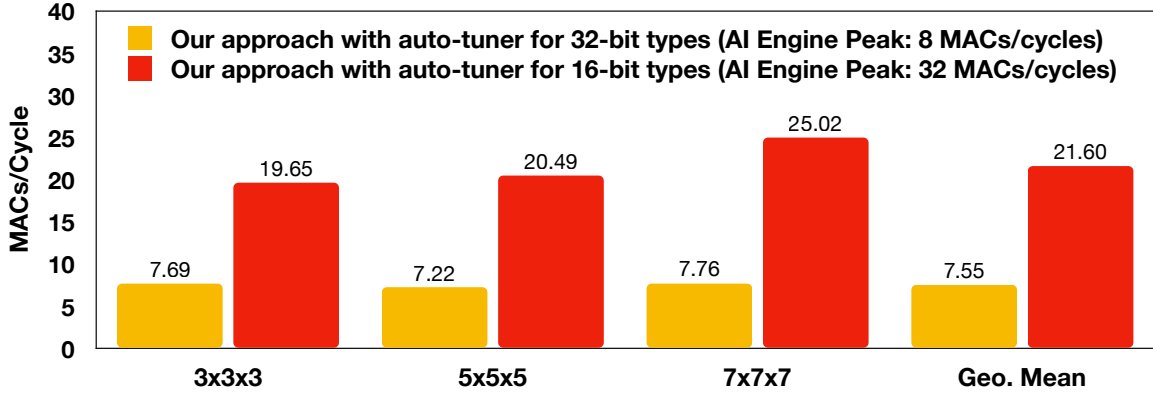


Figure 6.13: Performance of our approach generated codes for CONV3D workloads with weight sizes as 3x3x3, 5x5x5, 7x7x7.

mance in the 16-bit case is still significantly higher than the 32-bit case, despite a lower efficiency.

6.6 Related Work

High-level and domain-specific compiler frameworks [70, 174, 158, 175, 176, 177, 178, 179] have been shown to improve the productivity of application programmers, while generating high-performance code for a variety of architectures including CPUs, GPUs, FPGAs, Spatial accelerators, and distributed systems. Notably, the Halide framework [70] for image processing pipelines has gained popular attention in the academic and industrial world. Recently, Vocke et al. [180] extended the Halide framework to support specialized Digital Signal Processors (DSPs), mainly focusing on SIMD instruction sets and heterogeneous scratchpad memories of the Intel Imaging Processing Units (IPUs). Furthermore, Halide has the support for the Hexagon Vector eXtensions (HVX) on the Qualcomm Hexagon DSP processors. However, none of the above prior work focused on targeting the 2D SIMD datapaths and the shuffle interconnection networks, which are unique to the AI Engine.

To the best of our knowledge, the only prior work on auto-vectorizing for a 2D SIMD datapath is the work by Dasika et al. [32], where the authors have proposed a greedy compiler approach implemented as an extension to Trimaran [181] compiler, to identify a sequence of back to back vector operations for execution on their PEPSC’s architecture chained FPUs. But our approach identifies a group of such back to back dependent (i.e., fusible 1D logical) vector operations by searching in the triplet representation, a simplified and symbolic view of the convolution loop body.

Exploiting vector register reuse (including partial reuse) on SIMD units is a vital op-

timization to achieve high-performance, and prior work exploited the reuse by shuffling the vector registers using the data manipulation/shuffle units [168, 182, 183]. But our approach constructs a larger vector load covering the loads having reuse and uses the AI Engine’s unique shuffle network to select the desired elements. Furthermore, our approach uses the shuffle network to address the unaligned vector loads and scalar broadcasts without requiring any additional hardware support.

The vector codes generated by our approach are viewed as high-performance primitives that are intended to execute on a single AI Engine. These primitives are composed and integrated by a high-level compiler to run larger tensor convolutions across multiple AI Engines. Some of the prior works that have followed the similar strategy of automating the library/primitive development for the performance-critical kernels are SPIRAL [184] for the domain of linear transforms, ATLAS [185] for the basic linear algebra subroutines (BLAS), and FFTW [186] for the discrete Fourier transforms.

6.7 Summary

In this work, we introduced Vyasa, a high-level programming system built on the Halide framework, to generate high-performance vector codes for the tensor convolutions onto the Xilinx Versal AI Engine. Our proposed multi-step compiler approach leverages the AI Engine’s unique capabilities of the 2D SIMD datapath and the shuffle interconnection networks to achieve close to the peak performance for various workloads. Manually identifying best schedules and writing the corresponding intrinsic-based code is extremely challenging and error-prone, even for experts, thereby demonstrating the benefits of our automatic approach. Our results show geometric means of 7.6 and 23.3 MACs/cycle for 32-bit and 16-bit operands (which represent 95.9% and 72.8% of the peak performance respectively). For four of these workloads for which expert-written implementations were available to us, Vyasa achieved a geometric mean performance improvement of 1.10× from Halide code that is around 50× smaller than the expert-written C/C++ code.

Chapter 5 and Chapter 6 focused on our advances in compiler optimizations for accelerating machine learning applications on the spatial accelerators. In the next chapter, we focus on accelerating graph algorithms on a domain-specific processor, i.e., EMU, a thread-migratory and near-memory architecture introduced to optimize weak-locality optimizations.

CHAPTER 7

COMPILER OPTIMIZATIONS FOR GRAPH ANALYTICS ON A THREAD MIGRATORY ARCHITECTURE (EMU)

7.1 Abstract

Unlike dense linear algebra applications, graph applications typically suffer from poor performance because of 1) inefficient utilization of memory systems through random memory accesses to graph data, and 2) overhead of executing atomic operations. Hence, there is a rapid growth in improving both software and hardware platforms to address the above challenges. One such improvement in the hardware platform is a realization of the Emu system, a thread migratory and near-memory processor. In the Emu system, a thread responsible for computation on a datum is automatically migrated over to a node where the data resides without any intervention from the programmer. The idea of thread migrations is very well suited to graph applications as memory accesses of the applications are irregular. However, thread migrations can hurt the performance of graph applications if overhead from the migrations dominates benefits achieved through the migrations.

In this preliminary study [37], we explore two high-level compiler optimizations, i.e., loop fusion and edge flipping, and one low-level compiler transformation leveraging hardware support for remote atomic updates to address overheads arising from thread migration, creation, synchronization, and atomic operations. We performed a preliminary evaluation of these compiler transformations by manually applying them on three graph applications over a set of RMAT graphs from Graph500 –Conductance, Bellman-Ford’s algorithm for the single-source shortest path problem, and Triangle Counting. Our evaluation targeted a single node of the Emu hardware prototype, and has shown an overall geometric mean reduction of 22.08% in thread migrations.

7.2 Introduction

Though graph applications are increasing in importance with the advent of “big data”, achieving high performance with graph algorithms is non-trivial and requires careful attention from programmers [1]. Two significant bottlenecks to achieving higher performance on existing CPU and GPU-based architectures are 1) inefficient utilization of memory systems through random memory accesses to graph data, and 2) overhead of executing atomic operations. Since graph applications are typically cache-unfriendly and are not well sup-

ported by existing traditional architectures, there is growing attention being paid by the architecture community to innovate suitable architectures for such applications. One such innovation is the Emu system, a highly scalable near memory system with support for migrating threads without programmer intervention [15]. The system is designed to improve the performance of data-intensive applications exhibiting weak locality, i.e., from irregular and cache-unfriendly memory access which are often found in graph analytics [51] and sparse matrix algebra operations [52].

Emu architecture. An Emu system consists of multiple Emu nodes interconnected by a fast-rapid IO network, and each node (shown in Figure 7.1) contains *nodelets*, stationary cores and migration engines. Each *nodelet* consists of a Narrow Channel DRAM (NC-DRAM) memory unit and multiple Gossamer cores, and the co-location of the memory unit with the cores makes the overall Emu system a near-memory system. Even though each nodelet has a different physical co-located memory unit, the Emu system provides a logical view of the entire memory via the partitioned global address space (PGAS) model with memory contributed by each nodelet.

Each gossamer core of a nodelet is a general-purpose, simple pipelined processor with no support for data caches and branch prediction units, and is also capable of supporting 64 concurrent threads using fine-grain multi-threading. A key aspect of the Emu system is thread migration by hardware, i.e., a thread is migrated on a remote memory read by removing thread context from the nodelet and transmitting the thread context to a remote nodelet without programmer intervention. As a result, each nodelet requires multiple queues such as service, migration and run queues to process threads spawned locally (using *spawn* instruction) and also migrated threads.

Software support. The Emu system supports the Cilk parallel programming model for thread spawning and synchronization using `cilk_spawn`, `cilk_sync` and `cilk_for` constructs [54]. Since the Emu hardware automatically takes care of thread migration and management; hence the Cilk run-time is discarded in the toolchain. Also, it is important to note that appending a `cilk_spawn` keyword before a function invocation to launch a new task is directly translated to the `spawn` instruction of the Emu ISA during the compilation. The Emu system also provides libraries for data allocation and distribution over multiple nodelets, and intrinsic functions for atomic operations and migrating thread control functions. Also, there has been significant progress made in supporting standard C libraries on the Emu system.

Even though the Emu system is designed to improve the performance of data-sensitive workloads exhibiting weak-locality, the thread migrations across nodelets can hamper the

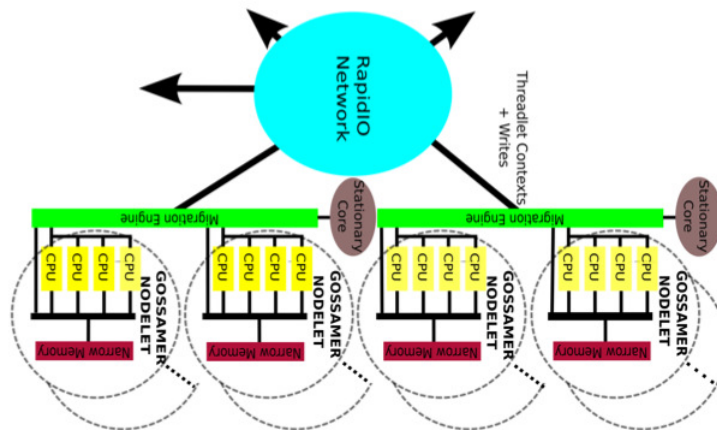


Figure 7.1: Overview of a single Emu node (figure source: [53]), where a dotted circle represents a nodelet. Note that, the co-location of the narrow channel memory unit (NCDRAM) with gossamer cores makes the overall Emu system a near memory system.

performance if overhead from the thread migration dominates the benefits achieved through the migration. In the next section, we study both high-level and low-level compiler transformations which can be applied to original graph applications to mitigate the overheads as mentioned earlier.

7.3 Compiler Transformations

In this section, we discuss two high-level compiler transformations (*Node fusion* and *Edge flipping*)¹, and one low-level compiler transformation leveraging the *remote atomic update* feature of the hardware, to mitigate the impact of overheads in the performance of graph applications on the Emu system.

7.3.1 Node/Loop Fusion

Programmers write graph applications with multiple parallel loops over nodes of a graph either to 1) compute various properties of a node (e.g., in *Conductance* [187, 188]), or 2) query on computed properties of nodes (e.g., in *Average teenage followers* [189]). In such scenarios, multiple such parallel loops can be grouped into a single parallel loop, and compute multiple properties in the same loop or query immediately after computing

¹Note that these high-level transformations – node fusion and edge flipping – have already been explored in past work on optimizing graph algorithms on the x86 architectures [79], and we are evaluating them in the context of the EMU system in this work.

the properties. This grouping can result in reducing thread migrations occurring in later loops, and also overheads arising from thread creation and synchronization. The grouping of multiple such parallel loops is akin to loop fusion, a classical compiler transformation for improving locality; but we use the transformation to reduce unnecessary migrations (for more details, see Section 7.4.2).

7.3.2 Edge Flipping

Edge flipping is another compiler transformation discussed in [79] to flip a loop over incoming edges of a node with outgoing edges of the node. However, we generalize the edge flipping transformation to allow flips between both incoming and outgoing edges. To allow this bi-directional flipping, the transformation assumes an input graph to be bi-directional, i.e., each node in the graph stores a list of incoming edges along with outgoing edges.

Vertex centric graph algorithms such as Page rank, Bellman-Ford algorithm for single-source shortest path, Page coloring offer opportunities to explore the edge flipping transformation since these algorithms either explore incoming edges of a node to avoid synchronization (pull-based approach), or explore outgoing edges to reduce random memory accesses (push-based approach), or explore a combination [190, 80]. We discuss the above push-pull dichotomy in Section 7.3.2, using the Bellman-Ford's algorithm as a representative of vertex-centric graph algorithms.

7.3.3 Use of Remote Updates

Remote updates, one of the architectural features of the Emu, are stores and atomic operations to a remote memory location that don't require returning a value to the thread, and these operations do not result in thread migrations [15]; instead they send an information packet to the remote nodelet containing the data and the operation to be performed. These remote updates also can be viewed as very efficient special-purpose migrating threads, and they don't return a result unlike regular atomic operations, but they return an acknowledgment that the memory unit of the remote nodelet has accepted the operation. We leverage this feature as a low-level compiler transformation replacing regular atomic operations that don't require returning a value by the corresponding remote updates. The benefits of this transformation can be immediately seen in vertex-centric algorithms (Section 7.4.3) and also in the triangular counting algorithm (Section 7.4.4).

7.4 Experiments

In this section, we present the benefits of applying the compiler transformations on graph algorithms. We begin with an overview of the experimental setup and the graph algorithms used in the evaluation, and then we present our discussion on preliminary results for each algorithm.

7.4.1 Experimental Setup

Our evaluation uses dedicated access to a single node of the Emu system, i.e., the Emu Chick prototype² which uses an Arria 10 FPGA to implement Gossamer cores, migration engines, and stationary cores of each nodelet. Table 7.1 lists the hardware specifications of a single node of the Emu Chick.

Table 7.1: Specifications of a single node of the Emu system.

	Emu system
Microarch	Emu1 Chick
Clock speed	150 MHz
#Nodelets	8
#Cores/Nodelet	1
#Threads/Core	64
Memorysize/Nodelet	8 GB
NCDRAM speed	1600MHz
Compiler toolchain	emusim.HW.x (18.08.1)

In the following experiments, we compare two experimental variants: 1) Original version of a graph algorithm running with all cores of a single node and 2) Transformed version after manually applying compiler transformations on the graph algorithm. In all experiments, we measure only the execution time of the kernel and report the geometric mean execution time measured over 50 runs repeated in the same environment for each data point. The speedup is defined as the execution time of the original version of a graph algorithm divided by the execution time of the transformed version of the program running with all cores of a single node of the Emu system in both cases, i.e., eight cores.

We also use an in-house simulation environment of the Emu prototype, whose specifications match with the hardware details mentioned in Table 7.1, to measure statistics of programs such as thread migrations, threads created and terminated. We are not currently aware of any methods for extracting these statistics from the hardware. We define the

²Several aspects of the system are scaled down in the prototype Emu system, e.g., number of gossamer cores of a nodelet

percentage reduction in thread migrations³ as follows:

$$\begin{aligned} & \% \text{reduction in migrations} \\ & = \left(1 - \left(\frac{\# \text{migrations in the transformed version}}{\# \text{migrations in the original version}}\right)\right) \times 100 \end{aligned}$$

Finally, we evaluate the benefits of compiler transformations by measuring both improvements in execution time on the Emu hardware and reduction in thread migrations on the Emu simulator.

Graph applications: For our evaluation, we consider three graph algorithms, i.e., 1) Conductance algorithm, 2) Bellman-Ford’s algorithm for Single-source shortest path (SSSP) problem, and 3) Triangle counting algorithm. Both original and transformed versions of above algorithms are implemented using the Meatbee framework [191], an in-house experimental streaming graph engine used to develop graph algorithms for the Emu system. The Meatbee framework, inspired by the STINGER framework [192], uses a striped array of pointers to distribute the vertex array across all nodelets in the system, and also implements the adjacency list as a hash table with a small number of buckets.

Table 7.2: Experimental evaluation of three graph algorithms (Conductance, SSSP-BF and Triangle counting) on the RMAT graphs from scales 6 to 14 specified by Graph500. Transformations applied on the algorithms: Conductance/SSSP-BF/Triangle counting: (Node fusion)/(Edge flipping and Remote updates)/ (Remote updates). The evaluation is done a single node of the Emu system described in Table 7.1. Note that we had intermittent termination issues while running SSSP-BF from scale 13-14 on the Emu node, and hence we omitted its results.

Scale	#vertices	#edges	Thread migrations in the original program			Execution time of the original program (ms), geometric mean of 50 runs		
			Conductance	SSSP-BF	Triangle counting	Conductance	SSSP-BF	Triangle counting
6	64	1K	6938	10915	26407	4.45	26.32	53.63
7	128	2K	13812	22851	84168	7.51	393.04	163.36
8	256	4K	28221	48354	252440	13.89	1634.64	547.84
9	512	8K	59068	104653	809423	32.13	2887.61	1694.09
10	1K	16K	122088	220204	2475350	64.59	4589.42	3942.55
11	2K	32K	253364	474118	7381977	134.43	10225.10	12649.30
12	4K	64K	522530	1136600	21777902	844.38	32140.30	36199.60
13	8K	128K	1065640	2332741	64063958	1841.53	-	185864.00
14	16K	256K	2171311	4569519	180988114	7876.99	-	721578.00

Input data-sets: We use RMAT graphs (edges of these graphs are generated randomly with a power-law distribution), scale⁴ from 6 to 14 as specified by Graph500 [2]. Note

³Note that the thread migration counts are for the entire program, and we are not currently aware of any existing approaches on how to obtain migration counts for a specific region of code.

⁴A scale of n for an input graph refers to having 2ⁿ vertices.

that all the above graphs specified by Graph500 are generated using the utilities present in the STINGER framework. Table 7.2 presents details of the RMAT graphs used in our evaluation, and total thread migrations and execution times of the original graph algorithms on the Emu system.

7.4.2 Conductance algorithm

The conductance algorithm is a graph metric application to evaluate a graph partition by counting the number of edges between nodes in a given partition and nodes in other graph partitions [187, 188]. The algorithm is frequently used to detect community structures in social graphs. An implementation of the conductance algorithm is shown in Algorithm 4. The implementation⁵ at a high-level consists of three parallel loops iterating over vertices of a graph to compute different properties (such as `din`, `dout`, `dcross`) of a given partition (specified as `id` in the algorithm). Finally, these properties are used to compute conductance value of the partition of the graph.

Algorithm 4: An implementation of the Conductance algorithm [187, 188].

```

1 def CONDUCTANCE(V, id);
2 for each v ∈ V do in parallel with reduction
3   if v.partition_id == id then
4     ; // Thread migration for v.partition_id value
4     din+ = v.degree ;
5 for each v ∈ V do in parallel with reduction
6   if v.partition_id != id then
7     dout+ = v.degree
8 for each v ∈ V do in parallel with reduction
9   if v.partition_id == id then
10    for each nbr ∈ v.nbrs do
11      if nbr.partition_id != id then
12        dcross+ = 1
13 return dcross/((din < dout)?din : dout)

```

As can be seen from the implementation, the EMU hardware spawns a thread for every vertex (`v`) of the graph from the first parallel loop (lines 2-4), and migrates to a nodelet where the vertex property `partition_id` is stored after encountering the property (`v.partition_id`) at line 3. Since the degree property of the vertex (`v`) is also stored

⁵The implementation is from a naive translation from existing graph analytics domain-specific compilers for non-EMU platforms.

on the same nodelet as of the other property⁶, the thread doesn't migrate on encountering the property, `v.degree`, at line 4. After reduction of the `din` variable, the hardware performs a global synchronization of all spawned threads because of an implicit barrier after the parallel loop. After the synchronization, the hardware again spawns a thread for every vertex from the second parallel loop (lines 5-7), and migrates after encountering the same property (`v.partition_id` at line 6). The same behavior is repeated in the third parallel loop as well (lines 8-12). The repeated migrations to the same nodelet from multiple parallel loops, which arise from accessing the same property or a different property which is stored on the same nodelet, can be reduced by fusing all the three parallel loops into a single loop. Also, the fusion of multiple parallel loops can reduce the overhead of multiple thread creations and synchronization. As can be seen from Figure 7.2, we have observed a geometric mean reduction of 6.06% in the total number of thread migrations after fusing three loops. As a result, we also found a geometric mean speedup of 1.95x in the execution time of the computation over the scale 6-14 of RMAT graphs specified by Graph500. This performance improvement demonstrates the need for fusing parallel loops over nodes of a graph to compute values/properties together to reduce thread migrations in applications such as Conductance.

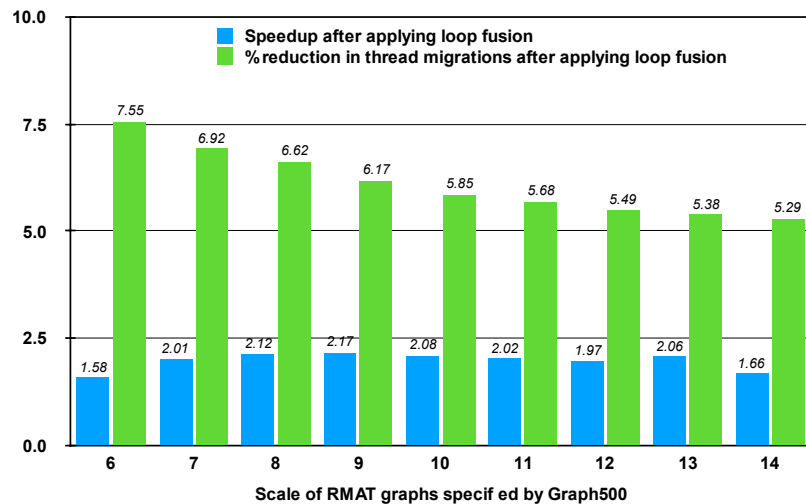


Figure 7.2: Speedup over the original conductance algorithm on a single Emu node (8 nodelets) and % reductions in thread migrations after applying loop fusion.

7.4.3 Single Source Shortest Path using Bellman-Ford's Algorithm (SSSP-BF)

Bellman-Ford's algorithm is used to compute shortest paths from a single source vertex to all the other vertices in a weighted directed graph. An implementation of the algorithm is

⁶The properties of vertices (such as `partition_id`, `degree`) are allocated similar to the vertex allocation, i.e., uniformly across all nodelets.

shown in Algorithm 5. We added a minor step (at lines 15, 18, 23-25) in the body of the t -loop to the implementation for termination if subsequent iterations of the t -loop will not make any more changes, i.e., the distance computed (`temp_distance`) for each vertex in the current iteration is the same as the distance in the previous iteration (`distance`).

Algorithm 5: An implementation of the Bellman-Ford's algorithm (SSSP-BF).

```

1 def SSSP_BFS( $V, id$ );
2  $distance(id) \leftarrow 0$ ;
3  $distance(v) \leftarrow MAX$ , for  $\forall v \in (V - \{id\})$ ;
4  $temp\_distance(v) \leftarrow 0$ , for  $\forall v \in V$ ;
5 for  $t \leftarrow 0$  to  $|V| - 1$  do
6   for each  $v \in V$  do in parallel
7     for each  $u \in incoming\_neighbors(v)$  do
8        $temp = distance(u) + weight(u, v)$ ;
9       ; // Migration for distance(u) value
10      if  $distance(v) > temp$  then
11         $temp\_distance(v) = temp$ ;
12      end
13    end
14  endfor
15  ;
16   $modified \leftarrow false$ ;
17  for each  $v \in V$  do in parallel
18    if  $distance(v) \neq temp\_distance(v)$  then
19       $modified \leftarrow true$ ;
20       $distance(v) = temp\_distance(v)$ 
21    end
22  endfor
23  ;
24  if  $modified == false$  then
25    break;
26  end
27 return  $distance$ ;

```

As can be seen from the implementation, the EMU hardware spawns a thread for every vertex (v) of the graph from the parallel loop (lines 6-13) nested inside the t -loop. The thread responsible for a particular vertex (v) in a given iteration (t) migrates to an incoming neighbor vertex (u) on encountering the accesses `distance(u)` and `weight(u, v)` (line 8). After adding the values, the thread migrates back to the original node for writing after encountering the access `temp_distance(u)` (line 9). The same migration behavior is repeated for every incoming neighbor vertex, and finally the local value based on the best distance from incoming neighbors is computed. This approach is commonly known

as a pull-based approach since the vertex pulls information from incoming neighbors to update its local value. However, the back and forth migrations for every neighbor vertex via incoming edges can be avoided by doing the edge flipping transformation (discussed in Section 7.3.2), i.e., the loop iterating over incoming edges is flipped into a loop over outgoing edges. The transformations leads to a push-based approach for the SSSP algorithm, in which a vertex pushes its contribution ($\text{distance}(u) + \text{weight}(u, v)$) to its neighbors accessible via outgoing edges and doesn't require migrating to the neighbors, as in the pull-based approach. Since multiple vertices can have a common neighbor, the contribution is done atomically, i.e., by using `atomic_min` in our implementation.

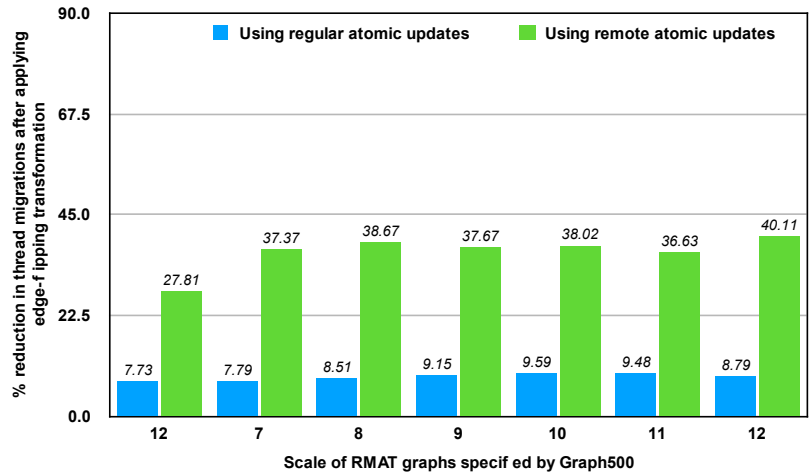


Figure 7.3: % reductions in thread migrations of SSSP-BF algorithm after applying edge flipping with regular atomic updates and with remote atomic updates on a single node (8 nodelets) of Emu Chick.

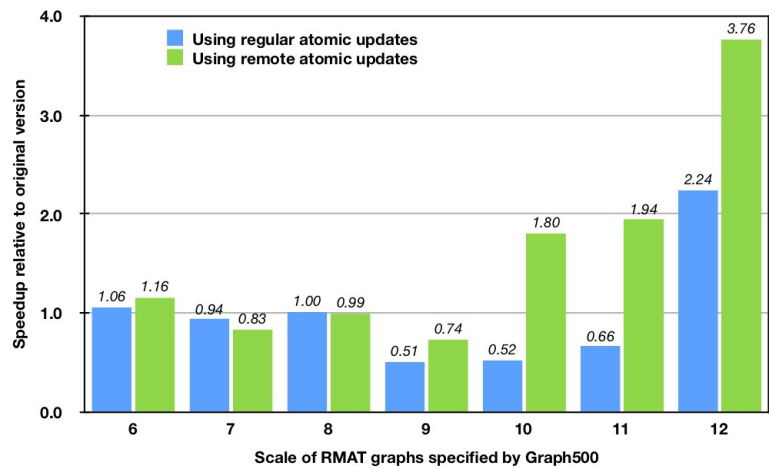


Figure 7.4: Speedup of SSSP-BF algorithm on a single Emu node (8 nodelets) after applying edge flipping with regular atomic updates and with remote updates.

As a result of applying edge-flipping transformation, we have observed a geometric

mean reduction of 8.69% in the total number of thread migrations (shown in Figure 7.3). However, the push-based approach with regular atomic updates didn't perform well compared with the pull-based approach from the scale of 7 to 9 (shown in Figure 7.4), because of irregularity in the input graphs and imbalance in the number of incoming and outgoing edges. As a result, the cost of migrating back and forth in the pull-based approach was not expensive compared to doing more atomic updates in the push-based approach for the above data points. This observation is in accordance with the push-pull dichotomy discussed in [190, 80].

Furthermore, the push-based approach can be strengthened by replacing regular atomic updates with remote atomic updates since a node which is pushing its contribution (i.e., its distance) to neighbors via outgoing edges doesn't need a return value. By doing so, we have observed a geometric mean reduction of 30.28% in thread migrations (shown in Figure 7.3) compared to the push-based approach with regular atomic updates. Also, there is an overall geometric mean improvement of 1.57x in execution time relative to the push-based approach with regular atomic updates (shown in Figure 7.4). The above performance improvement demonstrates the need for using remote atomic updates for scalable performance, and also exploring hybrid approaches involving both push and pull strategies based on input graph data.

7.4.4 Triangle Counting Algorithm

Triangle counting is another graph metric algorithm which computes the number of triangles in a given undirected graph, and also computes the number of triangles that each node belongs to [193]. The algorithm is frequently used in complex network analysis, random graph models, and also real-world applications such as spam detection. An implementation of the Triangle counting is shown in Algorithm 6, and it works by iterating over each vertex(v), picking two distinct neighbors (u, w), and check if there exists an edge between them to be part of a triangle. Also, the implementation avoids duplicate counting by delegating the counting of a triangle to the vertex with lower id.

In the above implementation of the triangle counting algorithm, whenever a triangle is identified (line 7), the implementation atomically increments the overall triangles count and triangle counts of the three vertices of the triangle. As part of the atomic update operation, the thread performs a migration to the nodelet having the address. However, the thread incrementing the triangle counts doesn't need the return value of the increment for further computation; hence, the regular atomic updates can be replaced by remote atomic updates to reduce thread migrations. After replacing with remote updates, we have observed a geometric mean reduction of 54.55% in the total number of thread migrations (shown in Fig-

Algorithm 6: An implementation of the Triangle counting algorithm [193].

```

1  $tc(v) \leftarrow 0$ , for  $\forall v \in (V)$ ;
2 for each  $v \in V$  do in parallel
3   for each  $u \in v.nbrs$  do
4     if  $nbr1 > v$  then
5       for each  $w \in v.nbrs$  do
6         if  $w > u$  then
7           if  $edge\_exists(u, w)$  then
8              $tc\_count ++$ ; //Atomic ;
9              $tc(v) ++$ ; //Atomic ;
10             $tc(u) ++$ ; //Atomic ;
11             $tc(w) ++$ ; //Atomic ;
           ; // Above regular atomics can be replaced by the
           remote updates.

```

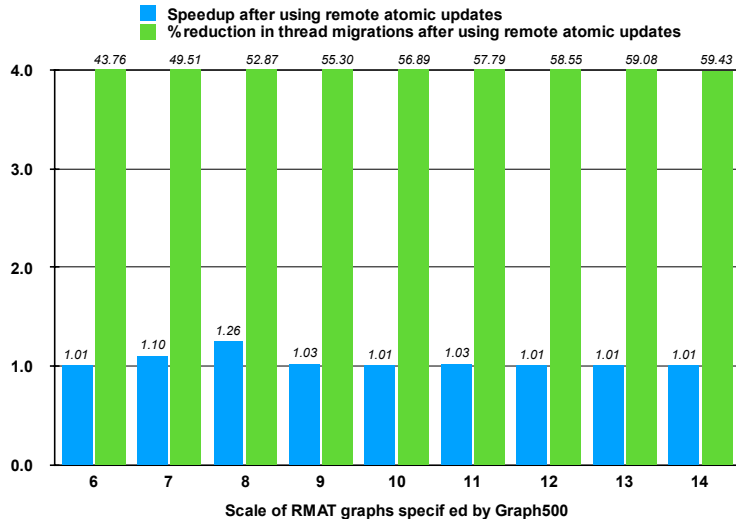


Figure 7.5: Speedup over the original triangle counting implementation on a single Emu node (8 nodelets) and % reductions in thread migrations after using remote atomic updates.

ure 7.5). As a result, we also found a geometric mean speedup of $1.05x^7$ in the execution time of the kernel over the scale 6-14 of RMAT graphs specified by Graph500.

7.5 Related Work

There is an extensive body of literature in optimizing graph applications for a variety of traditional architectures [194, 195, 196], accelerators [197, 198], and processing in mem-

⁷Note that the computational complexity of the triangle counting algorithm is significant, i.e., $O(m^{\frac{3}{2}})$ where m is number of edges, and 5% improvement is equivalent to few thousands of msecs as reported in Table 7.2.

ory (PIM) architectures [199, 200]. Also, there has been significant research done on optimizing task-parallel programs to reduce the overheads arising from task creation, synchronization [201, 202, 203] and migrations [204]. In this section, we discuss past work closely related to optimizing irregular applications for the Emu system and also past work on compiler optimizations in mitigating task (thread) creation, synchronization and migration overhead.

Emu related past work. Kogge et al. in [52] discussed migrating thread paradigm of the Emu system as an excellent match for systems with significant near-memory processing, and evaluated its advantage over a sparse matrix application (SpMV) and a streaming graph analytic benchmark (Firehose). Hein et al. [205] characterized the Emu chick hardware prototype (same as what we used in our evaluation) using custom kernels and discussed memory allocation, thread migrations strategies for SpMV kernels. In this work, we study high-level, and low-level compiler transformations that can benefit existing graph algorithms by leveraging the intricacies discussed in [15, 52, 205, 206, 207].

Programming models support and compiler optimizations for reducing thread creation, synchronization and migration overheads. Task-parallel programs often result in considerable overheads in task creation and synchronization, and hence approaches in [201, 202, 203] presented compiler frameworks to transform the input program to reduce the overheads using optimizations such as task fusion, task chunking, synchronization (`finish` construct) elimination. Our study on the loop fusion transformation to reduce thread creation and synchronization overheads on the Emu system is inspired by the above compiler frameworks and also from the Green-Marl DSL compiler [79].

7.6 Summary

Graph applications are increasing in popularity with the advent of “big data”, but achieving higher performance is not trivial. The major bottlenecks in graph applications are 1) inefficient utilization of memory subsystems through random memory accesses to the graph data, and 2) overhead of executing atomic operations. Since these graph applications are cache-unfriendly and are not well handled by existing traditional architectures, there is growing attention in the architecture community to innovate suitable architectures for such applications.

One of the innovative architectures to handle graph applications is a thread migratory architecture (Emu system), where a thread responsible for computation on a data is migrated over to a nodelet where the data resides. However, there are significant challenges which need to be addressed to gain the potential of Emu system, and they are reducing

thread migration, creation, synchronization, and atomic operation overheads. In this study, we explored two high-level compiler optimizations, i.e., loop fusion and edge flipping, and one low-level compiler transformation leveraging remote atomic updates to address the above challenges. We performed a preliminary evaluation of these compiler transformations by manually applying them on three graph applications over a set of RMAT graphs from Graph500 –Conductance, Bellman-Ford’s algorithm for the single-source shortest path problem, and Triangle Counting. Our evaluation targeted a single node of the Emu hardware prototype, and has shown an overall geometric mean reduction of 22.08% in thread migrations. This preliminary study clear motivates us in exploring the implementation of automatic compiler transformations to alleviate the overheads arising from running graph applications on the Emu system.

CHAPTER 8

CONCLUSIONS AND FUTURE DIRECTIONS

This dissertation on advancing compiler optimization is motivated by two observations – 1) computer hardware going through a significant disruption as we approach the end of Moore’s Law, and 2) demand for performance is broadening across multiple application domains. There exist numerous benefits of using the optimizing compilers relative to Ninja programmers or library-based approaches. However, the compilers still require advancements in program analysis, transformations, and code generation to enable a wide range of application domains to better exploit the advancements in both the general-purpose and domain-specific parallel architectures as part of the hardware disruption. Now, we briefly summarize our advancements and future directions in each of our compiler advancements.

1) Advancing dependence analysis using explicit parallelism for enhanced parallelization on general-purpose multi-core processors: This work is motivated by the observation that software with explicit parallelism is on the rise. This explicit parallelism could be used to enable a broader set of polyhedral transformations by mitigating conservative dependences, compared to what might have been possible if the input program had been sequential. We introduced an approach that reduces spurious dependences from the conservative dependence analysis by intersecting them with the happens-before relations from parallel constructs. The final set of the dependences can then be passed on to a polyhedral transformation tool, such as P_{Lu}To or PolyAST, to enable transformations of explicitly parallel programs. We evaluated our approach using OpenMP benchmark programs from the KASTORS and the Rodinia benchmark suites. The approach reduced spurious dependences from the conservative analysis of these benchmarks, and the resulting dependence information broadened the range of legal transformations in the polyhedral optimization phase. Overall, our experimental results show geometric mean performance improvements of 1.62x and 2.75x on the 12-core Intel Westmere and 24-core IBM Power8 platforms respectively, relative to the original OpenMP versions.

Future directions: A promising direction for future work is to extend the approach to programs with recursive task parallelism, as well as programs that do not satisfy serial elision property such as SPMDized code with explicit OpenMP threads and barriers. Analyzing more constructs in the input program such as C/C++ structs/classes and STL data structures in the input programs and generating task-parallel constructs in the code

generation phase are also subjects for future work.

Applicability to other architectures: Our approach of analyzing explicitly-parallel programs and enhancing dependence analysis can also be used for optimizing parallel programs (e.g., OpenMP/CUDA) on GPU architectures by plugging in optimizer and code-generator for GPUs, for, e.g., PPCG [114] or PolyAST [208].

2) Unification of multiple storage transformations with loop optimizations for enhanced vectorization on general-purpose vector processors (SIMD/GPUs): Despite the fact that compiler technologies for automatic vectorization have been under development for over four decades, there are still considerable gaps in modern compilers’ capabilities to perform automatic vectorization for SIMD units. This work focuses on advancing the state of the art with respect to handling *memory-based anti* (write-after-read) or *output* (write-after-write) dependences in vectorizing compilers. In this work, we integrate both Source Variable Renaming (SoVR) and Sink Variable Renaming (SiVR) transformations into a unified formulation, and formalize the “cycle-breaking” problem as a minimum weighted set cover optimization problem. Our approach also can ensure that the additional storage introduced by our transformations remains within the user-provided bounds. We implemented our approach in PPCG, a state-of-the-art optimization framework for loop transformations, and evaluated it on eleven kernels from the TSVC benchmark suite. Our experimental results show a geometric mean performance improvement of $4.61\times$ on an Intel Xeon Phi (KNL) machine relative to the optimized performance obtained by Intel’s ICC v17.0 product compiler. Further, our results demonstrate a geometric mean performance improvement of $1.08\times$ and $1.14\times$ on the Intel Xeon Phi (KNL) and Nvidia Tesla V100 (Volta) platforms relative to past work that only performs the SiVR transformation [29], and of $1.57\times$ and $1.22\times$ on both platforms relative to past work on using both SiVR and SoVR transformations [30]. We believe that our techniques will be increasingly important in the current era of pervasive SIMD parallelism since the non-vectorized code will incur an increasing penalty in runtime on future hardware platforms.

Future directions: A promising direction for future work is to further enhance the unified formulation by including variable expansion [113] and forward propagation techniques [55] to break pure-output and to handle pure-flow dependence cycles respectively. Also, we plan to extend our approach and implementation to handle non-affine regions of codes and support the vectorization of outer loops. Furthermore, we plan to investigate enabling loop transformations (such as tiling in case of cycles on tiles) using variable renaming transformations.

Applicability to other architectures: Our approach of unification of multiple storage

transformations can also be used to break dependence cycles in case of loop tiling and also to break selective storage dependences to enable more freedom for the loop transformations on both multi-core CPUs and GPU architectures. Breaking all storage dependences can be viewed as converting into Array SSA representation.

3) Data-centric compiler for deep learning operators onto domain-specific DNN spatial accelerators In this work, we provided a precise understanding of DNN operators whose mappings can be described in the MDC notation by introducing a set of rules over the operators’ abstract loop nest form. Furthermore, we introduced a transformation for translating mappings into the MDC notation for exploring the mapping space. Then, we also proposed a decoupled off-chip/on-chip approach that decomposes the mapping space into off-chip and on-chip subspaces, and first optimizes the off-chip subspace followed by the on-chip subspace. We implemented our decoupled approach in a tool called *Marvel*, and a significant benefit of our approach is that it applies to any DNN operator conformable with the MDC notation. Our approach reduced the search space of CONV2D operators from four major DNN models from 9.4×10^{18} to $1.5 \times 10^8 + 5.9 \times 10^8 \approx 2.1 \times 10^8$, which is a reduction factor of ten billion (Table 5.4), while generating mappings that demonstrate a geometric mean performance improvement of $10.25\times$ higher throughput and $2.01\times$ lower energy consumption compared with three state-of-the-art mapping styles from past work.

Future directions: In the future, we envision 1) Marvel integration with the MLIR compiler infrastructure for wide usability, 2) extending the MDC notation and its cost model to support non-conformable operators, and also 3) using for a wide range of applications, including the neuro-architecture search.

Furthermore, a promising direction for future research, is considering multiple accelerators (potentially heterogeneous) on a chip to schedule DNN models. Also, another direction is to advance compiler research in finding optimal mappings for sparse accelerators, given the prevalence of sparse DNN models arising from model pruning.

Applicability to other architectures: Our decoupled approach to reduce the overall space of mappings can also be used for exploring efficient mappings for other architecture such as multi-core CPUs and GPUs having multiple levels of the memory hierarchy.

4) High-performance vectorizing compiler for tensor convolutions on the Xilinx AI Engine (domain-specific 2D SIMD processor): In this work, we introduced Vyasa, a high-level programming system built on the Halide framework, to generate high-performance vector codes for the tensor convolutions onto the Xilinx Versal AI Engine. Our proposed multi-step compiler approach leverages the AI Engine’s unique capabilities of the 2D SIMD

datapath and the shuffle interconnection networks to achieve close to the peak performance for various workloads. Manually identifying best schedules and writing the corresponding intrinsic-based code is extremely challenging and error-prone, even for experts, thereby demonstrating the benefits of our automatic approach. Our results show a geometric mean of 7.6 and 23.3 MACs/cycle for 32-bit and 16-bit operands (which represent 95.9% and 72.8% of the peak performance respectively). For four of these workloads for which expert-written implementations were available to us, Vyasa achieved a geometric mean performance improvement of 1.10 \times from Halide code that is around 50 \times smaller than the expert-written C/C++ code.

Future directions: An exciting future direction is to extend our system to other computationally expensive linear algebra kernels. We also plan to integrate the generated high-performance codes into a high-level compiler to run larger convolutions across multiple AI Engines.

Applicability to other architectures: Our approach of fusing multiple 1D logical vector operations for 2D SIMD unit, exploring various loop transformations and data-layouts can also be applied with extensions to other domain-specific architectures such as 2D systolic arrays (e.g., TPUs [128]) and NVDLA accelerator [46] for generating optimized kernels for tensor convolutions.

5) Thread-migration aware compiler optimizations for graph analytics on a thread-migratory domain-specific hardware (EMU): Graph applications are increasing in popularity with the advent of “big data”, but achieving higher performance is not trivial. Unlike dense linear algebra applications, graph applications typically suffer from poor performance because of 1) inefficient utilization of memory systems through random memory accesses to graph data, and 2) overhead of executing atomic operations. Hence, there is a rapid growth in improving software and hardware platforms to address the above challenges. One such improvement in the hardware is the Emu system’s realization, a thread migratory and near-memory processor introduced to address application domains having weak-locality. In the Emu system, a thread responsible for computation on a datum is automatically migrated over to a node where the data resides without any intervention from the programmer. The idea of thread migrations is very well suited to graph applications as memory accesses of the applications are irregular. However, thread migrations can hurt graph applications’ performance if overhead from the migrations dominates the benefits achieved through migrations. In this work, we explored two high-level compiler optimizations, i.e., loop fusion and edge flipping, and one low-level compiler transformation leveraging remote atomic updates to address the above challenges. We performed a preliminary

evaluation of these compiler transformations by manually applying them on three graph applications over a set of RMAT graphs from Graph500 –Conductance, Bellman-Ford’s algorithm for the single-source shortest path problem, and Triangle Counting. Our evaluation targeted a single node of the Emu hardware prototype and has shown an overall geometric mean reduction of 22.08% in thread migrations. This preliminary study motivates us to explore automatic compiler transformations to alleviate the overheads arising from running graph applications on the Emu system.

Future directions: A promising direction for future work is to integrate these compiler transformations into a domain-specific language and compiler for graph algorithms such as Green-Marl [79], graphit [80] to generate efficient code for Emu system with a simple graph algorithm specification as input.

REFERENCES

- [1] A. Lenharth, D. Nguyen, and K. Pingali, “Parallel Graph Analytics,” *Commun. ACM*, vol. 59, no. 5, pp. 78–87, Apr. 2016.
- [2] D. A. Bader, J. Berry, S. Kahan, R. Murphy, E. J. Riedy, and J. Willcock, “Graph500 Benchmark 1 (search) Version 1.2,” Graph500 Steering Committee, Tech. Rep., Sep. 2011.
- [3] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, *et al.*, “Imagenet large scale visual recognition challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.
- [4] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, *et al.*, “Google’s neural machine translation system: Bridging the gap between human and machine translation,” *arXiv preprint arXiv:1609.08144*, 2016.
- [5] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, *et al.*, “End to end learning for self-driving cars,” *arXiv preprint arXiv:1604.07316*, 2016.
- [6] N. P. Jouppi, C. Young, N. Patil, and D. Patterson, “A Domain-Specific Architecture for Deep Neural Networks,” *Commun. ACM*, vol. 61, no. 9, pp. 50–59, Aug. 2018.
- [7] *Versal: The first adaptive compute acceleration platform (acap)*, v1.0.1, Xilinx, Sep. 2019.
- [8] P. Bose, “Power wall,” in *Encyclopedia of Parallel Computing*, D. Padua, Ed. Boston, MA: Springer US, 2011, pp. 1593–1608, ISBN: 978-0-387-09766-4.
- [9] R. R. Schaller, “Moore’s law: Past, present, and future,” *IEEE Spectr.*, vol. 34, no. 6, pp. 52–59, Jun. 1997.
- [10] D. M. Tullsen, S. J. Eggers, and H. M. Levy, “Simultaneous multithreading: Maximizing on-chip parallelism,” in *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, ser. ISCA ’98, Barcelona, Spain: ACM, 1998, pp. 533–544, ISBN: 1-58113-058-9.
- [11] P. Kongetira, K. Aingaran, and K. Olukotun, “Niagara: A 32-way multithreaded sparc processor,” *IEEE Micro*, vol. 25, no. 2, pp. 21–29, 2005.

- [12] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, “Conservation cores: Reducing the energy of mature computations,” *SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 205–218, Mar. 2010.
- [13] H. Esmaeilzadeh, E. Blem, R. St. Amant, K. Sankaralingam, and D. Burger, “Dark silicon and the end of multicore scaling,” in *Proceedings of the 38th Annual International Symposium on Computer Architecture*, ser. ISCA ’11, San Jose, California, USA: ACM, 2011, pp. 365–376, ISBN: 978-1-4503-0472-6.
- [14] *Memory intensive computing, the 3rd wall, and the need for innovation in architecture*, https://memsys.io/wp-content/uploads/2017/12/The_Wall.pdf, accessed 2019.
- [15] T. Dysart, P. Kogge, M. Deneroff, E. Bovell, P. Briggs, J. Brockman, K. Jacobsen, Y. Juan, S. Kuntz, R. Lethin, J. McMahon, C. Pawar, M. Perrigo, S. Rucker, J. Ruttenberg, M. Ruttenberg, and S. Stein, “Highly Scalable Near Memory Processing with Migrating Threads on the Emu System Architecture,” in *2016 6th Workshop on Irregular Applications: Architecture and Algorithms (IA3)*, 2016, pp. 2–9.
- [16] *Micron hybrid memory cube*, <http://www.micron.com/products/hybrid-memory-cube>, accessed 2015.
- [17] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, “ExTensor: An Accelerator for Sparse Tensor Algebra,” in *Proceedings of the 52Nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO ’52, Columbus, OH, USA: ACM, 2019, pp. 319–333, ISBN: 978-1-4503-6938-1.
- [18] D. F. Bacon, S. L. Graham, and O. J. Sharp, “Compiler transformations for high-performance computing,” *ACM Comput. Surv.*, vol. 26, no. 4, pp. 345–420, Dec. 1994.
- [19] V. Sarkar, “Parallel Functional Languages and Compilers,” in B. K. Szymanski, Ed., New York, NY, USA: ACM, 1991, ch. PTRAN—the IBM Parallel Translation System, pp. 309–391, ISBN: 0-201-52243-8.
- [20] U. Bondhugula, M. Baskaran, S. Krishnamoorthy, J. Ramanujam, A. Rountev, and P. Sadayappan, “Automatic transformations for communication-minimized parallelization and locality optimization in the polyhedral model,” in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, ser. CC’08/ETAPS’08, Budapest, Hungary: Springer-Verlag, 2008, pp. 132–146, ISBN: 3-540-78790-9, 978-3-540-78790-7.
- [21] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, “A practical automatic polyhedral parallelizer and locality optimizer,” in *Proceedings of the 29th*

- ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '08, Tucson, AZ, USA: ACM, 2008, pp. 101–113, ISBN: 978-1-59593-860-2.
- [22] J. Shirako, L.-N. Pouchet, and V. Sarkar, “Oil and Water Can Mix: An Integration of Polyhedral and AST-based Transformations,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '14, New Orleans, Louisiana: IEEE Press, 2014, pp. 287–298, ISBN: 978-1-4799-5500-8.
- [23] M. Kong, R. Veras, K. Stock, F. Franchetti, L.-N. Pouchet, and P. Sadayappan, “When polyhedral transformations meet simd code generation,” in *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '13, Seattle, Washington, USA: ACM, 2013, pp. 127–138, ISBN: 978-1-4503-2014-6.
- [24] D. A. Padua and M. J. Wolfe, “Advanced Compiler Optimizations for Supercomputers,” *Commun. ACM*, vol. 29, no. 12, pp. 1184–1201, Dec. 1986.
- [25] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, “Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions,” *CoRR*, vol. abs/1802.04730, 2018. arXiv: 1802.04730.
- [26] P. Chatarasi, J. Shirako, and V. Sarkar, “Polyhedral transformations of explicitly parallel programs,” in *5th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Amsterdam, Netherlands, Jan. 2015.
- [27] P. Chatarasi, J. Shirako, and V. Sarkar, “Polyhedral Optimizations of Explicitly Parallel Programs,” in *2015 International Conference on Parallel Architecture and Compilation (PACT)*, 2015, pp. 213–226.
- [28] P. Chatarasi, J. Shirako, A. Cohen, and V. Sarkar, “A Unified Approach to Variable Renaming for Enhanced Vectorization,” in *Languages and Compilers for Parallel Computing*, M. Hall and H. Sundar, Eds., Cham: Springer International Publishing, 2019, pp. 1–20, ISBN: 978-3-030-34627-0.
- [29] P. Calland, A. Darte, Y. Robert, and F. Vivien, “On the Removal of Anti- and Output-Dependences,” *International Journal of Parallel Programming*, vol. 26, no. 2, pp. 285–312, 1998.
- [30] C.-P. Chu and D. L. Carver, “An analysis of recurrence relations in Fortran Do-loops for vector processing,” in *[1991] Proceedings. The Fifth International Parallel Processing Symposium*, 1991, pp. 619–625.

- [31] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, “SODA: A High-Performance DSP Architecture for Software-Defined Radio,” *IEEE Micro*, vol. 27, no. 1, pp. 114–123, 2007.
- [32] G. Dasika, A. Sethia, T. Mudge, and S. Mahlke, “PEPSC: A Power-Efficient Processor for Scientific Computing,” in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 101–110.
- [33] M. Woh, S. Seo, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner, “AnySP: Anytime Anywhere Anyway Signal Processing,” *SIGARCH Comput. Archit. News*, vol. 37, no. 3, 128–139, Jun. 2009.
- [34] D. S. McFarlin, V. Arbatov, F. Franchetti, and M. Püschel, “Automatic SIMD Vectorization of Fast Fourier Transforms for the Larrabee and AVX Instruction Sets,” in *Proceedings of the International Conference on Supercomputing*, ser. ICS ’11, Tucson, Arizona, USA: Association for Computing Machinery, 2011, 265–274, ISBN: 9781450301022.
- [35] F. Franchetti and M. Püschel, “Generating SIMD Vectorized Permutations,” in *Proceedings of the Joint European Conferences on Theory and Practice of Software 17th International Conference on Compiler Construction*, ser. CC’08/ETAPS’08, Budapest, Hungary: Springer-Verlag, 2008, 116–131, ISBN: 3540787909.
- [36] S. Seo, M. Woh, S. Mahlke, T. Mudge, S. Vijay, and C. Chakrabarti, “Customizing Wide-SIMD Architectures for H.264,” in *Proceedings of the 9th International Conference on Systems, Architectures, Modeling and Simulation*, ser. SAMOS’09, Samos, Greece: IEEE Press, 2009, 172–179, ISBN: 9781424445028.
- [37] P. Chatarasi and V. Sarkar, “A Preliminary Study of Compiler Transformations for Graph Applications on the Emu System,” in *Proceedings of the Workshop on Memory Centric High Performance Computing*, ser. MCHPC’18, Dallas, TX, USA: ACM, 2018, pp. 37–44, ISBN: 978-1-4503-6113-2.
- [38] *Virtual machine framework for parallel hardware and software design*, http://sips.inesc-id.pt/nfvr/msc_theses/msc10g/.
- [39] *An overview of simd architectures*, <https://en.wikipedia.org/wiki/SIMD>.
- [40] Y.-H. Chen, J. Emer, and V. Sze, “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks,” in *International Symposium on Computer Architecture (ISCA)*, 2016.
- [41] H. Sharma, J. Park, D. Mahajan, E. Amaro, J. K. Kim, C. Shao, A. Mishra, and H. Esmaeilzadeh, “From high-level deep neural models to FPGAs,” in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 2016.

- [42] A. Parashar *et al.*, “Scnn: An accelerator for compressed-sparse convolutional neural networks,” in *International Symposium on Computer Architecture (ISCA)*, 2017, pp. 27–40.
- [43] N. P. Jouppi, C. Young, N. Patil, D. Patterson, G. Agrawal, R. Bajwa, S. Bates, S. Bhatia, N. Boden, A. Borchers, *et al.*, “In-datacenter performance analysis of a tensor processing unit,” in *International Symposium on Computer Architecture (ISCA)*, IEEE, 2017, pp. 1–12.
- [44] V Aklaghi, A. Yazdanbakhsh, K. Samadi, H Esmailzadeh, and R. Gupta, “Snapea: Predictive early activation for reducing computation in deep convolutional neural networks,” in *International Symposium on Computer Architecture (ISCA)*, 2018.
- [45] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, “Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning,” in *International conference on Architectural support for programming languages and operating systems (ASPLOS)*, 2014.
- [46] NVIDIA, *NVIDIA Deep Learning Accelerator (NVDLA)*, <https://nvlsla.org>, 2018.
- [47] E. S. Chung, J. Fowers, K. Ovtcharov, M. Papamichael, A. M. Caulfield, T. Madsengill, M. Liu, D. Lo, S. Alkalay, M. Haselman, M. Abeydeera, L. Adams, H. Angepat, C. Boehn, D. Chiou, O. Firestein, A. Forin, K. S. Gatlin, M. Ghandi, S. Heil, K. Holohan, A. E. Hussein, T. Juhász, K. Kagi, R. Kovvuri, S. Lanka, F. van Megen, D. Mukhortov, P. Patel, B. Perez, A. Rapsang, S. K. Reinhardt, B. Rouhani, A. Sapek, R. Seera, S. Shekar, B. Sridharan, G. Weisz, L. Woods, P. Y. Xiao, D. Zhang, R. Zhao, and D. Burger, “Serving DNNs in Real Time at Datacenter Scale with Project Brainwave,” *IEEE Micro*, vol. 38, no. 2, pp. 8–20, 2018.
- [48] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, and *et al.*, “The ARM Scalable Vector Extension,” *IEEE Micro*, vol. 37, no. 2, 26–39, Mar. 2017.
- [49] *Xilinx ai engines and their applications*, v1.0.2, Xilinx, Oct. 2018.
- [50] M. Lam, “Software Pipelining: An Effective Scheduling Technique for VLIW Machines,” in *Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, ser. PLDI ’88, Atlanta, Georgia, USA: Association for Computing Machinery, 1988, 318–328, ISBN: 0897912691.
- [51] P. M. Kogge, “Graph Analytics: Complexity, Scalability, and Architectures,” in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 1039–1047.

- [52] P. M. Kogge and S. K. Kuntz, “A Case for Migrating Execution for Irregular Applications,” in *Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms*, ser. IA3’17, Denver, CO, USA: ACM, 2017, 6:1–6:8, ISBN: 978-1-4503-5136-2.
- [53] EmuTechnology, *Emu system level architecture*, 2017 (accessed December 12, 2017).
- [54] M. Frigo, C. E. Leiserson, and K. H. Randall, “The Implementation of the Cilk-5 Multithreaded Language,” in *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, ser. PLDI ’98, Montreal, Quebec, Canada: ACM, 1998, pp. 212–223, ISBN: 0-89791-987-4.
- [55] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe, “Dependence Graphs and Compiler Optimizations,” in *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL ’81, Williamsburg, Virginia: ACM, 1981, pp. 207–218, ISBN: 0-89791-029-X.
- [56] V. Sarkar, “Automatic Selection of High-order Transformations in the IBM XL FORTRAN Compilers,” *IBM J. Res. Dev.*, vol. 41, no. 3, pp. 233–264, May 1997.
- [57] *OpenMP Specifications*, <http://openmp.org/wp/openmp-specifications>.
- [58] B. Chamberlain, D. Callahan, and H. Zima, “Parallel programmability and the chapel language,” *International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.
- [59] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou, “Cilk: An efficient multithreaded runtime system,” in *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’95, Santa Barbara, California, USA: ACM, 1995, pp. 207–216, ISBN: 0-89791-700-6.
- [60] P. Charles, C. Grothoff, V. Saraswat, C. Donawa, A. Kielstra, K. Ebcioglu, C. von Praun, and V. Sarkar, “X10: An object-oriented approach to non-uniform cluster computing,” *SIGPLAN Not.*, vol. 40, no. 10, pp. 519–538, Oct. 2005.
- [61] J. Kim, A. Sukumaran-Rajam, V. Thumma, S. Krishnamoorthy, A. Panyala, L. Pouchet, A. Rountev, and P. Sadayappan, “A Code Generator for High-Performance Tensor Contractions on GPUs,” in *IEEE/ACM International Symposium on Code Generation and Optimization, CGO 2019, Washington, DC, USA, February 16-20, 2019*, M. T. Kandemir, A. Jimborean, and T. Moseley, Eds., IEEE, 2019, pp. 85–95, ISBN: 978-1-7281-1436-1.
- [62] R. Li, A. Sukumaran-Rajam, R. Veras, T. M. Low, F. Rastello, A. Rountev, and P. Sadayappan, “Analytical Cache Modeling and Tilesize Optimization for Ten-

- tor Contractions,” in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '19, Denver, Colorado: ACM, 2019, 74:1–74:13, ISBN: 978-1-4503-6229-0.
- [63] J. Cong and B. Xiao, “Minimizing computation in convolutional neural networks,” in *International conference on artificial neural networks (ICANN)*, Springer, 2014, pp. 281–290.
- [64] A. Krizhevsky *et al.*, “Imagenet classification with deep convolutional neural networks,” in *NIPS*, 2012.
- [65] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” in *International Conference on Learning Representations (ICLR)*, 2015.
- [66] A. Karpathy and L. Fei-Fei, “Deep visual-semantic alignments for generating image descriptions,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015.
- [67] A. Toshev and C. Szegedy, “Deeppose: Human pose estimation via deep neural networks,” in *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.
- [68] C. Farabet, C. Couprie, L. Najman, and Y. LeCun, “Learning hierarchical features for scene labeling,” *PAMI*, vol. 35, no. 8, pp. 1915–1929, 2013.
- [69] L. Shapiro and G. Stockman, *Computer Vision*. Upper Saddle River, NJ: Prentice-Hall, 2001.
- [70] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, “Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines,” *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.
- [71] R. T. Mullapudi, V. Vasista, and U. Bondhugula, “PolyMage: Automatic Optimization for Image Processing Pipelines,” in *ASPLOS*, ACM, 2015, pp. 429–443, ISBN: 978-1-4503-2835-7.
- [72] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, “Programming heterogeneous systems from an image processing DSL,” *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, pp. 1–25, 2017.
- [73] S. Hochreiter and J. Schmidhuber, “Long Short-Term Memory,” *Neural Comput.*, vol. 9, no. 8, 1735–1780, Nov. 1997.

- [74] *Intel DNNL: Intel deep neural network library*, <http://intel.github.io/mkl-dnn/>.
- [75] *NVIDIA cuDNN: Nvidia cuda deep neural network library*, <https://developer.nvidia.com/cudnn>.
- [76] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, “TVM: An Automated End-to-end Optimizing Compiler for Deep Learning,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’18, Carlsbad, CA, USA: USENIX Association, 2018, pp. 579–594, ISBN: 978-1-931971-47-8.
- [77] *GraphAnalytics*, <https://developer.nvidia.com/discover/graph-analytics>.
- [78] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, “Pregel: A System for Large-scale Graph Processing,” in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD ’10, Indianapolis, Indiana, USA: ACM, 2010, pp. 135–146, ISBN: 978-1-4503-0032-2.
- [79] S. Hong, H. Chafi, E. Sedlar, and K. Olukotun, “Green-Marl: A DSL for Easy and Efficient Graph Analysis,” in *Proceedings of the Seventeenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XVII, London, England, UK: ACM, 2012, pp. 349–362, ISBN: 978-1-4503-0759-8.
- [80] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. P. Amarasinghe, “GraphIt - A High-Performance DSL for Graph Analytics,” *CoRR*, vol. abs/1805.00923, 2018. arXiv: 1805.00923.
- [81] S. Borkar and A. A. Chien, “The future of microprocessors,” *Commun. ACM*, vol. 54, no. 5, pp. 67–77, May 2011.
- [82] L. Lamport, “Time, clocks, and the ordering of events in a distributed system,” *Commun. ACM*, vol. 21, no. 7, pp. 558–565, Jul. 1978.
- [83] P. Virouleau, P. Brunet, F. Broquedis, N. Furmento, S. Thibault, O. Aumage, and T. Gautier, “Evaluation of OpenMP Dependent Tasks with the KASTORS Benchmark Suite,” in *Using and Improving OpenMP for Devices, Tasks, and More - 10th International Workshop on OpenMP, IWOMP 2014, Salvador, Brazil, September 28-30, 2014. Proceedings*, 2014, pp. 16–29.
- [84] S. Che, M. Boyer, J. Meng, D. Tarjan, J. W. Sheaffer, S.-H. Lee, and K. Skadron, “Rodinia: A benchmark suite for heterogeneous computing,” in *Proceedings of the 2009 IEEE International Symposium on Workload Characterization (IISWC)*,

- ser. IISWC '09, Washington, DC, USA: IEEE Computer Society, 2009, pp. 44–54, ISBN: 978-1-4244-5156-2.
- [85] P. Feautrier and C. Lengauer, “Polyhedron model,” in *Encyclopedia of Parallel Computing*, D. A. Padua, Ed., Springer, 2011, pp. 1581–1592.
- [86] D. G. Wonnacott, “Constraint-based array dependence analysis,” UMI Order No. GAX96-22167, PhD thesis, College Park, MD, USA, 1995.
- [87] *OpenMP Technical Report 3 on OpenMP 4.0 enhancements*, <http://openmp.org/TR3.pdf>.
- [88] J. Doerfert, C. Hammacher, K. Streit, and S. Hack, “SPolly: Speculative Optimizations in the Polyhedral Model,” in *Proc. 3rd International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Berlin, Germany, Jan. 2013, pp. 55–61.
- [89] T. Grosser, J. Ramanujam, L.-N. Pouchet, P. Sadayappan, and S. Pop, “Optimistic delinearization of parametrically sized arrays,” in *Proceedings of the 29th ACM on International Conference on Supercomputing*, ser. ICS '15, Newport Beach, California, USA: ACM, 2015, pp. 351–360, ISBN: 978-1-4503-3559-1.
- [90] C. Liao, D. J. Quinlan, T. Panas, and B. R. de Supinski, “A rose-based openmp 3.0 research compiler supporting multiple runtime libraries,” in *Beyond Loop Level Parallelism in OpenMP: Accelerators, Tasking and More, 6th International Workshop on OpenMP, IWOMP 2010, Tsukuba, Japan, June 14-16, 2010, Proceedings*, M. Sato, T. Hanawa, M. S. Müller, B. M. Chapman, and B. R. de Supinski, Eds., ser. Lecture Notes in Computer Science, vol. 6132, Springer, 2010, pp. 15–28, ISBN: 978-3-642-13216-2.
- [91] *CANDL: Data dependence analysis tool in the polyhedral model*, <http://icps.u-strasbg.fr/bastoul/development/candl>.
- [92] B. Creusillet and F. Irigoin, “Exact versus approximate array region analyses,” in *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*, ser. LCPC '96, London, UK, UK: Springer-Verlag, 1997, pp. 86–100, ISBN: 3-540-63091-0.
- [93] D. Barthou *et al.*, “Fuzzy Array Dataflow Analysis,” *J. Parallel Distrib. Comput.*, vol. 40, no. 2, pp. 210–226, 1997.
- [94] B. Creusillet and F. Irigoin, “Interprocedural Array Region Analyses,” *Int. J. Parallel Program.*, vol. 24, no. 6, pp. 513–546, Dec. 1996.
- [95] S. Verdoolaege and T. Grosser, “Polyhedral Extraction Tool,” in *Second Int. Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, Paris, France, Jan. 2012.

- [96] R. Cytron, “Doacross: Beyond Vectorization for Multiprocessors,” in *ICPP’86*, 1986, pp. 836–844.
- [97] J. Shirako, P. Unnikrishnan, S. Chatterjee, K. Li, and V. Sarkar, “Expressing DOACROSS Loop Dependencies in OpenMP,” in *9th International Workshop on OpenMP (IWOMP)*, 2011.
- [98] P. Unnikrishnan, J. Shirako, K. Barton, S. Chatterjee, R. Silvera, and V. Sarkar, “A practical approach to DOACROSS parallelization,” in *Euro-Par*, 2012, pp. 219–231.
- [99] J.-F. Collard and M. Griebl, “Array Dataflow Analysis for Explicitly Parallel Programs,” in *Proceedings of the Second International Euro-Par Conference on Parallel Processing*, ser. Euro-Par ’96, 1996.
- [100] T. Yuki, P. Feautrier, S. Rajopadhye, and V. Saraswat, “Array Dataflow Analysis for Polyhedral X10 Programs,” in *Proceedings of the 18th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’07, 2013.
- [101] V. Sarkar, *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Cambridge, MA, USA: MIT Press, 1989, ISBN: 0262691302.
- [102] *Integer set library*, <http://isl.gforge.inria.fr>.
- [103] R. Baghdadi, A. Cohen, S. Guelton, S. Verdoolaege, J. Inoue, T. Grosser, G. Kouveli, A. Kravets, A. Lokhmatov, C. Nugteren, F. Waters, and A. F. Donaldson, “PENCIL: towards a platform-neutral compute intermediate language for dsls,” *CoRR*, vol. abs/1302.5586, 2013.
- [104] A. Pop and A. Cohen, “Preserving high-level semantics of parallel programming annotations through the compilation flow of optimizing compilers,” in *Proceedings of the 15th Workshop on Compilers for Parallel Computers (CPC’10)*, 2010.
- [105] C. Coarfa, Y. Dotsenko, J. Mellor-Crummey, F. Cantonnet, T. El-Ghazawi, A. Mohanti, Y. Yao, and D. Chavarría-Miranda, “An evaluation of global address space languages: Co-array fortran and unified parallel c,” in *Proceedings of the Tenth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPOPP ’05, Chicago, IL, USA: ACM, 2005, pp. 36–47, ISBN: 1-59593-080-9.
- [106] V. Maslov and W. Pugh, “Simplifying Polynomial Constraints Over Integers to Make Dependence Analysis More Precise,” In CONPAR 94 - VAPP VI, Int. Conf. on Parallel and Vector Processing, Tech. Rep., 1994.
- [107] T. B. Schardl, W. S. Moses, and C. E. Leiserson, “Tapir: Embedding fork-join parallelism into llvm’s intermediate representation,” in *Proceedings of the 22nd*

ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ser. PPOPP '17, Austin, Texas, USA: Association for Computing Machinery, 2017, 249–265, ISBN: 9781450344937.

- [108] N. B. Jensen and S. Karlsson, “Improving Loop Dependence Analysis,” *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, Aug. 2017.
- [109] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, “The ARM Scalable Vector Extension,” *IEEE Micro*, vol. 37, no. 2, pp. 26–39, Mar. 2017.
- [110] K. Kennedy and J. R. Allen, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2002, ISBN: 1-55860-286-0.
- [111] C.-P. Chu, “A Theoretical Approach Involving Recurrence Resolution, Dependence Cycle Statement Ordering and Subroutine Transformation for the Exploitation of Parallelism in Sequential Code,” UMI Order No. GAX92-07498, PhD thesis, Louisiana State University, Baton Rouge, LA, USA, 1992.
- [112] K. Knobe and V. Sarkar, “Array SSA Form and Its Use in Parallelization,” in *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, ser. POPL '98, San Diego, California, USA: ACM, 1998, pp. 107–120, ISBN: 0-89791-979-3.
- [113] P. Feautrier, “Array Expansion,” in *Proceedings of the 2Nd International Conference on Supercomputing*, ser. ICS '88, St. Malo, France: ACM, 1988, pp. 429–441, ISBN: 0-89791-272-1.
- [114] S. Verdoolaege, J. Carlos Juega, A. Cohen, J. Ignacio Gómez, C. Tenllado, and F. Catthoor, “Polyhedral Parallel Code Generation for CUDA,” *ACM Trans. Archit. Code Optim.*, vol. 9, no. 4, 54:1–54:23, Jan. 2013.
- [115] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema, J. Absar, S. v. Haastregt, A. Kravets, A. Lokhmotov, R. David, and E. Hajiyev, “PENCIL: A Platform-Neutral Compute Intermediate Language for Accelerator Programming,” in *Proceedings of the 2015 International Conference on Parallel Architecture and Compilation (PACT)*, ser. PACT '15, Washington, DC, USA: IEEE Computer Society, 2015, pp. 138–149, ISBN: 978-1-4673-9524-3.
- [116] S. Maleki, Y. Gao, M. J. Garzarán, T. Wong, and D. A. Padua, “An Evaluation of Vectorizing Compilers,” in *Proceedings of the 2011 International Conference on*

- Parallel Architectures and Compilation Techniques*, ser. PACT '11, Washington, DC, USA: IEEE Computer Society, 2011, pp. 372–382, ISBN: 978-0-7695-4566-0.
- [117] M. Weiss, “Strip Mining on SIMD Architectures,” in *Proceedings of the 5th International Conference on Supercomputing*, ser. ICS '91, Cologne, West Germany: ACM, 1991, pp. 234–243, ISBN: 0-89791-434-1.
- [118] D. B. Johnson, “Finding All the Elementary Circuits of a Directed Graph,” *SIAM Journal on Computing*, vol. 4, no. 1, pp. 77–84, 1975.
- [119] S. Verdoolaege, “Isl: An Integer Set Library for the Polyhedral Model,” in *Proceedings of the Third International Congress Conference on Mathematical Software*, ser. ICMS'10, Kobe, Japan: Springer-Verlag, 2010, pp. 299–302, ISBN: 3-642-15581-2, 978-3-642-15581-9.
- [120] J. Hopcroft and R. Tarjan, “Algorithm 447: Efficient Algorithms for Graph Manipulation,” *Commun. ACM*, vol. 16, no. 6, pp. 372–378, Jun. 1973.
- [121] D. Callahan, J. Dongarra, and D. Levine, “Vectorizing Compilers: A Test Suite and Results,” in *Proceedings of the 1988 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '88, Orlando, Florida, USA: IEEE Computer Society Press, 1988, pp. 98–105, ISBN: 0-8186-0882-X.
- [122] G. C. Evans, S. Abraham, B. Kuhn, and D. A. Padua, “Vector Seeker: A Tool for Finding Vector Potential,” in *Proceedings of the 2014 Workshop on Programming Models for SIMD/Vector Processing*, ser. WPMVP '14, Orlando, Florida, USA: ACM, 2014, pp. 41–48, ISBN: 978-1-4503-2653-7.
- [123] W.-L. Chang, C.-P. Chu, and M. S.-H. Ho, “Exploitation of parallelism to nested loops with dependence cycles,” *Journal of Systems Architecture*, vol. 50, no. 12, pp. 729–742, 2004.
- [124] S. Rus, G. He, C. Alias, and L. Rauchwerger, “Region Array SSA,” in *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques*, ser. PACT '06, Seattle, Washington, USA: ACM, 2006, pp. 43–52, ISBN: 1-59593-264-X.
- [125] U. Bondhugula, A. Acharya, and A. Cohen, “The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests,” *ACM Trans. Program. Lang. Syst.*, vol. 38, no. 3, 12:1–12:32, Apr. 2016.
- [126] S. G. Bhaskaracharya, U. Bondhugula, and A. Cohen, “SMO: An Integrated Approach to Intra-array and Inter-array Storage Optimization,” in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming*

Languages, ser. POPL '16, St. Petersburg, FL, USA: ACM, 2016, pp. 526–538, ISBN: 978-1-4503-3549-2.

- [127] *The future is here - iphone x (neural engine)*, <https://www.apple.com/newsroom/2017/09/the-future-is-here-iphone-x/>, 2017.
- [128] *Edge tpu: Google's purpose-built asic designed to run inference at the edge*. <https://cloud.google.com/edge-tpu/>, 2019.
- [129] Y.-H. Chen, T.-J. Yang, J. Emer, and V. Sze, “Eyeriss v2: A flexible accelerator for emerging deep neural networks on mobile devices,” *IEEE Journal on Emerging and Selected Topics in Circuits and Systems*, 2019.
- [130] *Accelerating dnns with xilinx alveo accelerator cards*, <https://www.xilinx.com/support/documentation/accel-dnns.pdf>.
- [131] Y. Ma, Y. Cao, S. Vrudhula, and J.-s. Seo, “Optimizing loop operation and dataflow in FPGA acceleration of deep convolutional neural networks,” in *International Symposium on Field-Programmable Gate Arrays (FPGA)*, ACM, 2017, pp. 45–54.
- [132] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, “Optimizing FPGA-based accelerator design for deep convolutional neural networks,” in *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2015, pp. 161–170.
- [133] A. Parashar, P. Raina, Y. S. Shao, Y.-H. Chen, V. A. Ying, A. Mukkara, R. Venkatesan, B. Khailany, S. W. Keckler, and J. Emer, “Timeloop: A Systematic Approach to DNN Accelerator Evaluation,” 2019.
- [134] H. Kwon, M. Pellauer, and T. Krishna, “An analytic model for cost-benefit analysis of dataflows in dnn accelerators,” *arXiv preprint arXiv:1805.02566*, 2018.
- [135] S. Dave, Y. Kim, S. Avancha, K. Lee, and A. Shrivastava, “DMazeRunner: Executing Perfectly Nested Loops on Dataflow Accelerators,” *ACM Trans. Embed. Comput. Syst.*, vol. 18, no. 5s, Oct. 2019.
- [136] X. Yang, M. Gao, Q. Liu, J. Setter, J. Pu, A. Nayak, S. Bell, K. Cao, H. Ha, P. Raina, C. Kozyrakis, and M. Horowitz, “Interstellar: Using Halide’s Scheduling Language to Analyze DNN Accelerators,” in *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '20, Lausanne, Switzerland: Association for Computing Machinery, 2020, 369–383, ISBN: 9781450371025.

- [137] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.
- [138] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, “Mobilenetv2: Inverted residuals and linear bottlenecks,” in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2018, pp. 4510–4520.
- [139] Z. Zhao, H. Kwon, S. Kuhar, W. Sheng, Z. Mao, and T. Krishna, “mRNA: Enabling Efficient Mapping Space Exploration on a Reconfigurable Neural Accelerator,” in *Proceedings of 2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, IEEE, 2019.
- [140] X. Yang, J. Pu, B. B. Rister, N. Bhagdikar, S. Richardson, S. Kvatinsky, J. Ragan-Kelley, A. Pedram, and M. Horowitz, “A Systematic Approach to Blocking Convolutional Neural Networks,” *CoRR*, vol. abs/1606.04209, 2016. arXiv: 1606.04209.
- [141] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, “Design space exploration of FPGA-based deep convolutional neural networks,” in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*, IEEE, 2016, pp. 575–580.
- [142] V. Sze, Y. Chen, T. Yang, and J. S. Emer, “Efficient Processing of Deep Neural Networks: A Tutorial and Survey,” *CoRR*, vol. abs/1703.09039, 2017. arXiv: 1703.09039.
- [143] J. Ferrante, V. Sarkar, and W. Thrash, “On estimating and enhancing cache effectiveness,” in *International Workshop on Languages and Compilers for Parallel Computing*, Springer, 1991, pp. 328–343.
- [144] H. Kwon, A. Samajdar, and T. Krishna, “Maeri: Enabling flexible dataflow mapping over dnn accelerators via reconfigurable interconnects,” in *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, ACM, 2018, pp. 461–475.
- [145] Z. Du, R. Fasthuber, T. Chen, P. Ienne, L. Li, T. Luo, X. Feng, Y. Chen, and O. Temam, “Shidiannao: Shifting vision processing closer to the sensor,” in *International Symposium on Computer Architecture (ISCA)*, 2015.
- [146] R. Allen and K. Kennedy, *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann, 2001, ISBN: 1-55860-286-0.
- [147] V. J. Reddi, C. Cheng, D. Kanter, P. Mattson, G. Schmuelling, C.-J. Wu, B. Anderson, M. Breughe, M. Charlebois, W. Chou, R. Chukka, C. Coleman, S. Davis, P. Deng, G. Diamos, J. Duke, D. Fick, J. S. Gardner, I. Hubara, S. Idgunji, T. B. Jablin, J. Jiao, T. S. John, P. Kanwar, D. Lee, J. Liao, A. Lokhmotov, F. Massa, P.

- Meng, P. Micikevicius, C. Osborne, G. Pekhimenko, A. T. R. Rajan, D. Sequeira, A. Sirasao, F. Sun, H. Tang, M. Thomson, F. Wei, E. Wu, L. Xu, K. Yamada, B. Yu, G. Yuan, A. Zhong, P. Zhang, and Y. Zhou, *MLPerf Inference Benchmark*, 2019. arXiv: 1911.02549 [cs.LG].
- [148] C. Li, Y. Yang, M. Feng, S. Chakradhar, and H. Zhou, “Optimizing memory efficiency for deep convolutional neural networks on GPUs,” in *SC’16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, IEEE, 2016, pp. 633–644.
- [149] V. Sarkar and N. Megiddo, “An Analytical Model for Loop Tiling and Its Solution,” in *Proceedings of the 2000 IEEE International Symposium on Performance Analysis of Systems and Software*, ser. ISPASS ’00, Washington, DC, USA: IEEE Computer Society, 2000, pp. 146–153, ISBN: 0-7803-6418-X.
- [150] J. Shirako, K. Sharma, N. Fauzia, L.-N. Pouchet, J. Ramanujam, P. Sadayappan, and V. Sarkar, “Analytical Bounds for Optimal Tile Size Selection,” in *Proceedings of the 21st International Conference on Compiler Construction*, ser. CC’12, Tallinn, Estonia: Springer-Verlag, 2012, pp. 101–121, ISBN: 978-3-642-28651-3.
- [151] Y.-H. Chen, T. Krishna, J. S. Emer, and V. Sze, “Eyeriss: An energy-efficient reconfigurable accelerator for deep convolutional neural networks,” *IEEE Journal of Solid-State Circuits*, vol. 52, no. 1, pp. 127–138, 2017.
- [152] Jeduc, *DDR4 SDRAM STANDARD*, <https://www.jedec.org/standards-documents/docs/jesd79-4a>, 2017.
- [153] O. Ronneberger, P. Fischer, and T. Brox, “U-net: Convolutional networks for biomedical image segmentation,” in *Medical Image Computing and Computer-Assisted Intervention (MICCAI)*, ser. LNCS, (available on arXiv:1505.04597 [cs.CV]), vol. 9351, Springer, 2015, pp. 234–241.
- [154] E. Qin, A. Samajdar, H. Kwon, V. Nadella, S. Srinivasan, D. Das, B. Kaul, and T. Krishna, “Sigma: A sparse and irregular gemm accelerator with flexible interconnects for dnn training.”
- [155] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, “Tensor Comprehensions: Framework-Agnostic High-Performance Machine Learning Abstractions,” *arXiv preprint arXiv:1802.04730*, 2018.
- [156] PlaidML, *PlaidML*, <https://github.com/plaidml/plaidml>, 2020.
- [157] T. Zerrell and J. Bruestle, *Stripe: Tensor compilation via the nested polyhedral model*, 2019. arXiv: 1903.06498 [cs.DC].

- [158] R. Baghdadi, J. Ray, M. B. Romdhane, E. Del Sozzo, A. Akkas, Y. Zhang, P. Suri-ana, S. Kamil, and S. Amarasinghe, “Tiramisu: A Polyhedral Compiler for Expressing Fast and Portable Code,” in *Proceedings of the 2019 IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO 2019, Washington, DC, USA: IEEE Press, 2019, 193–205, ISBN: 9781728114361.
- [159] C. Lattner, M. Amini, U. Bondhugula, A. Cohen, A. Davis, J. Pienaar, R. Riddle, T. Shpeisman, N. Vasilache, and O. Zinenko, *Mlir: A compiler infrastructure for the end of moore’s law*, 2020. arXiv: 2002.11054 [cs.PL].
- [160] X. S. Hu, S. Zagoruyko, and N. Komodakis, *Exploring weight symmetry in deep neural networks*, 2018. arXiv: 1812.11027 [cs.LG].
- [161] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Q. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, “A Hardware-Software Blueprint for Flexible Deep Learning Specialization,” *IEEE Micro*, vol. 39, no. 5, pp. 8–16, 2019.
- [162] B. Gaide, D. Gaitonde, C. Ravishankar, and T. Bauer, “Xilinx adaptive compute acceleration platform: Versal™ architecture,” in *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2019, pp. 84–93.
- [163] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’18, Carlsbad, CA, USA: USENIX Association, 2018, 579–594, ISBN: 9781931971478.
- [164] S. Chetlur, C. Woolley, P. Vandermersch, J. Cohen, J. Tran, B. Catanzaro, and E. Shelhamer, “cuDNN: Efficient Primitives for Deep Learning,” *CoRR*, vol. abs/1410.0759, 2014.
- [165] C. Zhang, G. Sun, Z. Fang, P. Zhou, P. Pan, and J. Cong, “Caffeine: Towards uniformed representation and acceleration for deep convolutional neural networks,” *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 2018.
- [166] P. Chatarasi, H. Kwon, N. Raina, S. Malik, V. Haridas, T. Krishna, and V. Sarkar, “MARVEL: A Decoupled Model-driven Approach for Efficiently Mapping Convolutions on Spatial DNN Accelerators,” *CoRR*, vol. abs/2002.07752, 2020. arXiv: 2002.07752.
- [167] S. Cyphers, A. K. Bansal, A. Bhiwandiwalla, J. Bobba, M. Brookhart, A. Chakraborty, W. Constable, C. Convey, L. Cook, O. Kanawi, R. Kimball, J. Knight, N. Korovaiko, V. Kumar, Y. Lao, C. R. Lishka, J. Menon, J. Myers, S. A. Narayana, A.

- Procter, and T. J. Webb, *Intel nGraph: An Intermediate Representation, Compiler, and Executor for Deep Learning*, 2018. arXiv: 1801.08058 [cs.DC].
- [168] K. A. Stock, “Vectorization and Register Reuse in High Performance Computing,” PhD thesis, The Ohio State University, 2014.
- [169] N. Sedaghati, R. Thomas, L. Pouchet, R. Teodorescu, and P. Sadayappan, “StVEC: A Vector Instruction Extension for High Performance Stencil Computation,” in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 276–287.
- [170] R. T. Mullanpudi, A. Adams, D. Sharlet, J. Ragan-Kelley, and K. Fatahalian, “Automatically Scheduling Halide Image Processing Pipelines,” *ACM Trans. Graph.*, vol. 35, no. 4, Jul. 2016.
- [171] A. Adams, K. Ma, L. Anderson, R. Baghdadi, T.-M. Li, M. Gharbi, B. Steiner, S. Johnson, K. Fatahalian, F. Durand, and J. Ragan-Kelley, “Learning to Optimize Halide with Tree Search and Random Programs,” *ACM Trans. Graph.*, vol. 38, no. 4, Jul. 2019.
- [172] E. Ahmed, A. Saint, A. E. R. Shabayek, K. Cherenkova, R. Das, G. Gusev, D. Aouada, and B. Ottersten, *A survey on Deep Learning Advances on Different 3D Data Representations*, 2018. arXiv: 1808.01462 [cs.CV].
- [173] R. Hou, C. Chen, and M. Shah, *An End-to-end 3D Convolutional Neural Network for Action Detection and Segmentation in Videos*, 2017. arXiv: 1712.01111 [cs.CV].
- [174] R. T. Mullanpudi, V. Vasista, and U. Bondhugula, “PolyMage: Automatic Optimization for Image Processing Pipelines,” in *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’15, Istanbul, Turkey: Association for Computing Machinery, 2015, 429–443, ISBN: 9781450328357.
- [175] N. Vasilache, O. Zinenko, T. Theodoridis, P. Goyal, Z. DeVito, W. S. Moses, S. Verdoolaege, A. Adams, and A. Cohen, “The Next 700 Accelerated Layers: From Mathematical Expressions of Network Computation Graphs to Accelerated GPU Kernels, Automatically,” *TACO*, vol. 16, no. 4, 38:1–38:26, 2020.
- [176] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, “TVM: An Automated End-to-End Optimizing Compiler for Deep Learning,” in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI’18, Carlsbad, CA, USA: USENIX Association, 2018, 579–594, ISBN: 9781931971478.

- [177] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula, “A DSL Compiler for Accelerating Image Processing Pipelines on FPGAs,” in *Proceedings of the 2016 International Conference on Parallel Architectures and Compilation*, ser. PACT ’16, Haifa, Israel: Association for Computing Machinery, 2016, 327–338, ISBN: 9781450341219.
- [178] V. Elango, N. Rubin, M. Ravishankar, H. Sandanagobalane, and V. Grover, “Diesel: DSL for Linear Algebra and Neural Net Computations on GPUs,” in *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, ser. MAPL 2018, Philadelphia, PA, USA: Association for Computing Machinery, 2018, 42–51, ISBN: 9781450358347.
- [179] N. K. Srivastava, H. Rong, P. Barua, G. Feng, H. Cao, Z. Zhang, D. H. Albonesi, V. Sarkar, W. Chen, P. Petersen, G. Lowney, A. Herr, C. J. Hughes, T. G. Mattson, and P. Dubey, “T2S-Tensor: Productively Generating High-Performance Spatial Hardware for Dense Tensor Computations,” in *27th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines, FCCM 2019, San Diego, CA, USA, April 28 - May 1, 2019*, 2019, pp. 181–189.
- [180] S. Vocke, H. Corporaal, R. Jordans, R. Corvino, and R. Nas, “Extending Halide to Improve Software Development for Imaging DSPs,” *ACM Trans. Archit. Code Optim.*, vol. 14, no. 3, Aug. 2017.
- [181] Trimaran., *An infrastructure for research in ILP*,) [http://www.trimaran.org/..](http://www.trimaran.org/)
- [182] L. Wang, Z. ChunYan, Y. Huang, and J. Xu, “Partial Elements Reuse of Vector Register in SIMD Mathematical Functions,” *International Journal of Advancements in Computing Technology*, vol. 4, pp. 327–335, Jan. 2012.
- [183] X. Jinchun, G. Shaozhong, and W. Lei, “Optimization Technology in SIMD Mathematical Functions Based on Vector Register Reuse,” in *2012 IEEE 14th International Conference on High Performance Computing and Communication 2012 IEEE 9th International Conference on Embedded Software and Systems*, 2012, pp. 1102–1107.
- [184] F. Franchetti, Y. Voronenko, P. A. Milder, S. Chellappa, M. R. Telgarsky, Hao Shen, P. D’Alberto, F. de Mesmay, J. C. Hoe, J. M. F. Moura, and M. Püschel, “Domain-specific library generation for parallel software and hardware platforms,” in *2008 IEEE International Symposium on Parallel and Distributed Processing*, 2008, pp. 1–5.
- [185] R. C. Whaley, A. Petitet, and J. J. Dongarra, “Automated empirical optimizations of software and the ATLAS project,” *PARALLEL COMPUTING*, vol. 27, 2001.

- [186] M. Frigo and S. G. Johnson, “The Design and Implementation of FFTW3,” *Proceedings of the IEEE*, vol. 93, no. 2, pp. 216–231, 2005.
- [187] B. Bollobas, *Modern Graph Theory*. Springer, 1998.
- [188] J. Leskovec and R. Sosič, “SNAP: A General-Purpose Network Analysis and Graph-Mining Library,” *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 1, 1:1–1:20, Jul. 2016.
- [189] S. Hong, S. Salihoglu, J. Widom, and K. Olukotun, “Simplifying Scalable Graph Processing with a Domain-Specific Language,” in *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, ser. CGO ’14, Orlando, FL, USA: ACM, 2014, 208:208–208:218, ISBN: 978-1-4503-2670-4.
- [190] M. Besta, M. Podstawski, L. Groner, E. Solomonik, and T. Hoefler, “To Push or To Pull: On Reducing Communication and Synchronization in Graph Computations,” in *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, ser. HPDC ’17, Washington, DC, USA: ACM, 2017, pp. 93–104, ISBN: 978-1-4503-4699-3.
- [191] E. Hein, *Meatbee, An Experimental Streaming Graph Engine*. <https://github.com/ehein6/meatbee>, 2017.
- [192] D. Ediger, R. McColl, J. Riedy, and D. A. Bader, “STINGER: High performance data structure for streaming graphs,” in *2012 IEEE Conference on High Performance Extreme Computing*, 2012, pp. 1–5.
- [193] T. Schank, “Algorithmic Aspects of Triangle-Based Network Analysis,” PhD thesis, Universität Karlsruhe, 2007.
- [194] D. A. Bader and K. Madduri, “Design and Implementation of the HPCS Graph Analysis Benchmark on Symmetric Multiprocessors,” in *Proceedings of the 12th International Conference on High Performance Computing*, ser. HiPC’05, Goa, India: Springer-Verlag, 2005, pp. 465–476, ISBN: 3-540-30936-5, 978-3-540-30936-9.
- [195] A. Yoo, E. Chow, K. Henderson, W. McLendon, B. Hendrickson, and U. Catalyurek, “A Scalable Distributed Parallel Breadth-First Search Algorithm on BlueGene/L,” in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing*, ser. SC ’05, Washington, DC, USA: IEEE Computer Society, 2005, pp. 25–, ISBN: 1-59593-061-2.
- [196] D. A. Bader and K. Madduri, “Designing multithreaded algorithms for breadth-first search and st-connectivity on the cray mta-2,” in *2006 International Conference on Parallel Processing (ICPP’06)*, 2006, pp. 523–530.

- [197] S. Hong, T. Oguntebi, and K. Olukotun, “Efficient Parallel Graph Exploration on Multi-Core CPU and GPU,” in *2011 International Conference on Parallel Architectures and Compilation Techniques*, 2011, pp. 78–88.
- [198] P. Harish and P. J. Narayanan, “Accelerating Large Graph Algorithms on the GPU Using CUDA,” in *Proceedings of the 14th International Conference on High Performance Computing*, ser. HiPC’07, Goa, India: Springer-Verlag, 2007, pp. 197–208, ISBN: 3-540-77219-7, 978-3-540-77219-4.
- [199] L. Nai, R. Hadidi, J. Sim, H. Kim, P. Kumar, and H. Kim, “GraphPIM: Enabling Instruction-Level PIM Offloading in Graph Computing Frameworks,” in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2017, pp. 457–468.
- [200] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, “A scalable processing-in-memory accelerator for parallel graph processing,” in *2015 ACM/IEEE 42nd Annual International Symposium on Computer Architecture (ISCA)*, 2015, pp. 105–117.
- [201] V. K. Nandivada, J. Shirako, J. Zhao, and V. Sarkar, “A Transformation Framework for Optimizing Task-Parallel Programs,” *ACM Trans. Program. Lang. Syst.*, vol. 35, no. 1, 3:1–3:48, Apr. 2013.
- [202] J. Shirako, J. M. Zhao, V. K. Nandivada, and V. N. Sarkar, “Chunking Parallel Loops in the Presence of Synchronization,” in *Proceedings of the 23rd International Conference on Supercomputing*, ser. ICS ’09, Yorktown Heights, NY, USA: ACM, 2009, pp. 181–192, ISBN: 978-1-60558-498-0.
- [203] J. Zhao, J. Shirako, V. K. Nandivada, and V. Sarkar, “Reducing Task Creation and Termination Overhead in Explicitly Parallel Programs,” in *Proceedings of the 19th International Conference on PACT*, Vienna, Austria: ACM, 2010, pp. 169–180, ISBN: 978-1-4503-0178-7.
- [204] Y. Yan, J. Zhao, Y. Guo, and V. Sarkar, “Hierarchical Place Trees: A Portable Abstraction for Task Parallelism and Data Movement,” in *Proceedings of the 22Nd International Conference on Languages and Compilers for Parallel Computing*, ser. LCPC’09, Newark, DE: Springer-Verlag, 2010, pp. 172–187, ISBN: 3-642-13373-8, 978-3-642-13373-2.
- [205] E. Hein, T. Conte, J. Young, S. Eswar, J. Li, P. Lavin, R. Vuduc, and J. Riedy, “An Initial Characterization of the Emu Chick,” in *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2018, pp. 579–588.
- [206] S. L. Mehmet E. Belviranli and J. S. Vetter, “Designing algorithms for the emu migrating-threads-based algorithms,” 2018.

- [207] J. Young, E. Hein, S. Eswar, P. Lavin, J. Li, J. Riedy, R. Vuduc, and T. Conte, *A Microbenchmark Characterization of the Emu Chick*, 2018. arXiv: 1809.07696 [cs.DC].

- [208] J. Shirako, A. Hayashi, and V. Sarkar, “Optimized Two-Level Parallelization for GPU Accelerators Using the Polyhedral Model,” in *Proceedings of the 26th International Conference on Compiler Construction*, ser. CC 2017, Austin, TX, USA: Association for Computing Machinery, 2017, 22–33, ISBN: 9781450352338.