UCLA
UCLA Electronic Theses and Dissertations

Title
Automatic Synthesis and Architecture Optimization of Systolic Arrays

Permalink
https://escholarship.org/uc/item/8k15d5qr

Author
Wang, Jie

Publication Date
2021

Peer reviewed|Thesis/dissertation

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

UNIVERSITY OF CALIFORNIA
Los Angeles

Automatic Synthesis and Architecture Optimization of Systolic Arrays

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

by

Jie Wang

2021

© Copyright by
Jie Wang
2021

ABSTRACT OF THE DISSERTATION

Automatic Synthesis and Architecture Optimization of Systolic Arrays

by

Jie Wang
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2021
Professor Jingsheng Jason Cong, Chair

A systolic array architecture consists of a grid of simple processing elements (PE) connected through local interconnects. With a massive number of PEs and a local interconnection, such an architecture is capable of achieving high performance and energy efficiency.

My dissertation focuses on extending the research of systolic array architecture in two fields: automatic systolic array synthesis and architecture optimization.

The first part of the dissertation focuses on the automated systolic array synthesis. Designing high-performance systolic arrays requires an understanding of both the application characteristics and hardware architecture, requiring non-trivial efforts to reap its benefits. There exists a large body of past works on developing compilation frameworks for systolic arrays. However, these works failed to reach a balance between the generality, performance, and productivity, making them hard to use in practice. Our work advances this field by leveraging two compilation technologies, the polyhedral model and high-level synthesis (HLS). We propose a new compilation framework, AutoSA, which is built on the polyhedral framework and is capable of generating high-performance systolic arrays on FPGA in HLS languages. We show that AutoSA can handle applications with complex dependence structures and generate designs with performance comparable to or better than manual designs.

AutoSA incorporates a broad set of hardware optimization techniques that open up a vast design space which is intractable to explore manually. To cope with this challenge, we propose an efficient auto-tuning framework, Odyssey, which finds optimal designs within

seconds. With both AutoSA and Odyssey, we reduce the development cycles of systolic arrays from weeks to days, which significantly boosts the productivity compared to the prior works.

In the second part of the dissertation, we present two application optimization studies that deploy systolic arrays for various applications and platforms. The first study investigates the architecture trade-offs when using systolic arrays for one important application, convolutional neural network (CNN). The results show that a single monolithic systolic array is insufficient to handle the divergent characteristics of different CNN layers. Therefore, we further explore the use of a multi-array architecture that implements several smaller systolic arrays with different configurations customized for each CNN layer. Multi-array systems help improve the throughput with a cost of longer latency. This work reveals the complexities and trade-offs when mapping a real-world application to systolic arrays.

In addition to FPGA, systolic arrays can also be mapped to GPU as an overlay above the existing GPU architecture. The second work investigates the performance trade-offs when mapping systolic arrays to GPU. We achieve a performance speedup by leveraging the shuffle instructions on Nvidia GPUs to implement the inter-PE communication compared to baselines using the shared memory.

Systolic array architecture plays an important role in the post-Moore's law era as one architecture candidate capable of delivering high performance and energy efficiency. The works presented in this dissertation provide comprehensive and efficient solutions to lowering the programming efforts and optimizing the performance of this architecture. We hope the promising results from these works will open the door to more deployment cases of systolic arrays in a broader range of applications and hardware platforms in the future.

The dissertation of Jie Wang is approved.

Jens Palsberg

Anthony John Nowatzki

Miloš D. Ercegovac

Jingsheng Jason Cong, Committee Chair

University of California, Los Angeles

2021

TABLE OF CONTENTS

1 Introduction **1**

1.1 Background 1

1.2 Overview of Dissertation 9

1.2.1 Automatic Systolic Arrays Synthesis 9

1.2.2 Architecture Optimization 12

1.3 Summary 14

2 Automatic Systolic Array Synthesis **16**

2.1 Background and Related Work 17

2.2 Contributions 19

2.3 Theoretical Background 21

2.3.1 Polyhedral Compilation Framework 21

2.3.2 Space-Time Transformation 24

2.4 First Attempt — PolySA 26

2.5 AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA 28

2.5.1 Framework Overview 28

2.5.2 Computation Management 30

2.5.3 Communication Management 36

2.5.4 Code Generation 45

2.5.5 Auto-Tuning 48

2.5.6 Evaluation Results 49

2.6 Summary 56

2.A Source Code of Evaluation Benchmarks 57

2.A.1 Matrix Multiplication . . . 57
2.A.2 Convolutional Neural Network (CNN) . . . 57
2.A.3 Matricized Tensor Times Khatri-Rao Product (MTTKRP) . . . 57
2.A.4 Chain of Tensor-Matrix Multiplications (TTMc) . . . 58
2.A.5 LU Decomposition . . . 58
2.B Systolic Arrays Generated by AutoSA for Matrix Multiplication . . . 60
2.C Systolic Arrays Generated by AutoSA for CNN . . . 61

3 Efficient Auto-Tuning Approaches for Systolic Arrays . . . 62
3.1 Background and Related Work . . . 63
3.1.1 Design Space Construction . . . 64
3.1.2 Search Methods . . . 66
3.2 Contributions . . . 68
3.3 Scope of the Odyssey Framework . . . 70
3.3.1 Scope of the Design Space . . . 70
3.3.2 Scope of the Supported Applications . . . 72
3.4 Constructing the Design Space . . . 72
3.4.1 Enhancement to AutoSA . . . 73
3.4.2 Loop Permutation Pruning . . . 74
3.4.3 Putting It All Together . . . 82
3.5 Searching the Design Space . . . 83
3.5.1 Overview of Different Search Methods . . . 83
3.5.2 Evolutionary Search . . . 86
3.5.3 Mathematical Programming . . . 89
3.6 Evaluation Results . . . 94

3.6.1 Performance Model Validation . . . 95
3.6.2 Auto-Tuning Evaluation . . . 96
3.6.3 Application 1: MM . . . 100
3.6.4 Application 2: CNN . . . 107
3.7 Lessons Learned from This Work . . . 111
3.8 Summary . . . 113
3.A Layer-Wise Systolic Array Performance Results on VGG16 . . . 114

4 Mapping Convolutional Neural Network to FPGA . . . 119
4.1 Background and Related Work . . . 120
4.2 Contributions . . . 124
4.3 Evaluation Methodology . . . 127
4.3.1 Hardware Architecture . . . 127
4.3.2 CNN Benchmarks . . . 128
4.4 Architecture 1: Single Array . . . 130
4.4.1 Architecture Overview . . . 130
4.4.2 Tuning Algorithms . . . 131
4.4.3 Evaluation Results . . . 135
4.4.4 Summary . . . 141
4.5 Architecture 2: Multi-Array with a One-Shot Schedule . . . 142
4.5.1 Architecture Overview . . . 142
4.5.2 Tuning Algorithms . . . 147
4.5.3 Evaluation Results . . . 150
4.6 Architecture 3: Multi-Array with a Periodic Schedule . . . 153
4.6.1 Architecture Overview . . . 153

4.6.2 Tuning Algorithms . . . 156
4.6.3 Evaluation Results . . . 157
4.7 Architecture 4: Multi-Array with a One-Shot Schedule and Adaptive Partitioning 159
4.7.1 Architecture Overview . . . 159
4.7.2 Tuning Algorithms . . . 160
4.7.3 Evaluation Results . . . 163
4.8 Architecture Comparison . . . 166
4.9 Summary . . . 168

5 Accelerating Genome Sequence Alignment Algorithms on GPU . . . 170
5.1 Background . . . 171
5.2 Contributions . . . 172
5.3 Understanding Shuffle Instructions . . . 173
5.3.1 Shuffle Instruction . . . 173
5.3.2 Microbenchmarks for Measuring the Shuffle Latency . . . 175
5.4 Sequence Alignment Algorithms . . . 178
5.4.1 Algorithm Overview . . . 178
5.4.2 Algorithm Details . . . 179
5.5 Implementation Details . . . 181
5.5.1 Design Methodology . . . 181
5.5.2 Design Optimization of SW . . . 183
5.5.3 Design Optimization of PairHMM . . . 185
5.5.4 Performance Model . . . 187
5.6 Evaluation Results . . . 188
5.6.1 Experiment Setup . . . 188

5.6.2 Performance Overview 189
5.6.3 Model Validation 190
5.6.4 Trade-Off Analysis 192
5.7 Related Work 194
5.8 Summary 195

6 Conclusion 196

LIST OF FIGURES

1.1 Instruction energy breakdown for an in-order processor in 45nm [Hor14]. 2
1.2 Block diagram of the Apple M1 SoC. 2
1.3 An example of a 2D systolic array for matrix multiplication. 3
1.4 Comparison between a broadcast architecture and a systolic array architecture for matrix multiplication. 5
1.5 Compilation flow of the AutoSA framework. 10
1.6 Overview of the Odyssey auto-tuning framework. 11

2.1 Schedule tree representation for the initial schedule of the example MM program. 22
2.2 Schedule tree representation for the example code in Listing 2. 23
2.3 Example of space-time transformation. 25
2.4 Compilation flow of the PolySA framework. 26
2.5 Overview of the AutoSA compilation flow. 29
2.6 Overview of computation management. 30
2.7 Example of space-time transformation. 32
2.8 Example of array partitioning. 33
2.9 Example of latency hiding. 35
2.10 Example of SIMD vectorization. 35
2.11 Insert a filter to isolate the statement instances of group g_2. 40
2.12 PE array with the I/O module for g_2 and its loop schedule. 41
2.13 Example of I/O clustering for I/O group g_2. 41
2.14 Clustering L1 I/O modules by applying the transformation $T1 := \{[c0, c1] \rightarrow [c1, c0]\}$ on the space loops. 42
2.15 A complete 2D systolic array for the MM example. 43

2.16 Detailed architecture of the I/O network for the MM example. 44
2.17 A code generation example of inserting FIFO statements into the programs. . . 46
2.18 Example of array contraction. 47
2.19 Example of host serialization. 47
2.20 Example of loop infinitization. 48
2.21 Detailed array architecture of the MM designs 0, 3, 4. 51
2.22 The dependence graph of LU decomposition and the mapped array. 54
2.23 Ablation analysis of several hardware optimization techniques. 56
2.24 Architecture of six systolic arrays with different dataflows generated by AutoSA for matrix multiplication. 60
2.25 Architecture of ten systolic arrays with different dataflows generated by AutoSA for CNN. 61

3.1 Architecture of two different systolic arrays for MM. 65
3.2 Overview of the Odyssey framework. 69
3.3 Examples of applying array partitioning with the extended schedule IR. 74
3.4 Example of systolic arrays designs generated with different loop permutation orderings. 75
3.5 Example of a different loop permutation ordering for MM. 79
3.6 Example of different loop permutation orderings for CNN. 81
3.7 Example of different mutation methods. 87
3.8 Tuning performance comparison with different hyper-parameters α. 88
3.9 Example of crossover in evolutionary search. 89
3.10 Tuning performance comparison of different objective targets. 93
3.11 Performance model validation results. 96
3.12 Comparison of the best designs identified by different tuning methods. 98

3.13 Comparison of sample efficiency of different tuning methods. 99
3.14 Best designs found in 5 seconds by Odyssey for all different designs of MM. . . . 99
3.15 Throughput comparison of different designs for MM. 100
3.16 DSP usage of different latency-optimal designs for MM. 101
3.17 Performance comparison of different energy-optimal designs for MM. 104
3.18 Energy breakdown for two designs with the loop ordering $< [i, j], k >$ for MM. . 106
3.19 Normalized throughput of different systolic arrays on CONV1, CONV2, CONV3 of VGG16. Throughput of each systolic array design is normalized against the highest number achieved on each CONV layer. 108
3.20 Architectures of two systolic arrays with dataflows $[h, i]$ and $[o, h]$ for CNN. . . . 109
3.21 Throughput comparison of different dataflows on VGG16. Throughput of each design is normalized against the highest throughput achieved on the entire network. 110
3.22 Geo. Mean of the best throughput achieved by different dataflows across all the CONV layers in VGG16. 111
3.23 Throughput and DSP usage of different designs with only divisor tiling factors for 1024×1024×1024 MM. 112
3.24 Normalized throughput of different systolic arrays on CONV1 of VGG16. Throughput of each systolic array design is normalized against the highest number achieved on each CONV layer. 114
3.25 Normalized throughput of different systolic arrays on CONV2 of VGG16. 114
3.26 Normalized throughput of different systolic arrays on CONV3 of VGG16. 115
3.27 Normalized throughput of different systolic arrays on CONV4 of VGG16. 115
3.28 Normalized throughput of different systolic arrays on CONV5 of VGG16. 115
3.29 Normalized throughput of different systolic arrays on CONV6 of VGG16. 116
3.30 Normalized throughput of different systolic arrays on CONV7 of VGG16. 116
3.31 Normalized throughput of different systolic arrays on CONV8 of VGG16. 116

3.32 Normalized throughput of different systolic arrays on CONV9 of VGG16. 117
3.33 Normalized throughput of different systolic arrays on CONV10 of VGG16. . . . 117
3.34 Normalized throughput of different systolic arrays on CONV11 of VGG16. . . . 117
3.35 Normalized throughput of different systolic arrays on CONV12 of VGG16. . . . 118
3.36 Normalized throughput of different systolic arrays on CONV13 of VGG16. . . . 118

4.1 Comparison of layer sequential and layer fusion. 122
4.2 Architecture and workload mapping of four different designs for CNN acceleration. 124
4.3 Comparison of the throughput and DSP efficiency of Arch1-4 on different CNNs. 126
4.4 Parallelism vs. data locality of all the convolutional layers in VGG16, ResNet50, and MobileNetV2. 129
4.5 Architecture, CNN workload mapping and task schedule of a single monolithic array design (Arch1). 130
4.6 Overview of the single array tuning flow. 131
4.7 Single layer throughput vs. network throughput for VGG16 on array $[o, h]$. (We have omitted layers with the same problem dimensions.) 133
4.8 Search results of the proposed tuning method for Arch1 on VGG16. 134
4.9 Normalized throughput of different systolic array designs on VGG16, ResNet50, and MobileNetV2. The throughput of each design is normalized against the highest throughput achieved by all the designs across three CNNs. 135
4.10 Comparison between the best layer-wise solutions (free) versus the single fixed array (fixed) on VGG16. 137
4.11 Comparison of normalized throughput, parallelism, and CTC of layer-optimal solutions on VGG16, ResNet50, and MobileNetV2. 138
4.12 Throughput comparison of different systolic arrays on VGG16-CONV1. 139
4.13 Latency of different hardware components in three selected systolic arrays on VGG16-CONV1. 139

4.14 Throughput comparison of different systolic arrays on ResNet50-CONV41. . . . 140
4.15 Latency of different hardware components in three systolic arrays on ResNet50-CONV41. . . . 141
4.16 Architecture, CNN workload mapping, and task schedule of the multi-array design with a one-shot schedule (Arch2). . . . 142
4.17 Architecture of the streaming buffers between two SL arrays SL_i and SL_{i+1} in Arch2. . . . 143
4.18 Task schedule of Arch2. . . . 145
4.19 Overview of the tuning flow for Arch2. . . . 147
4.20 Comparison of the search performance of the Bayesian optimizer for Arch2 on VGG16 w/ and w/o the use of initial samples. . . . 150
4.21 Achieved latency and DSP efficiency of each array in Arch2 on VGG16, ResNet50, and MobileNetV2. . . . 153
4.22 Architecture, CNN workload mapping, and task schedule of the multi-array design with a periodic schedule (Arch3). . . . 154
4.23 Task schedule of Arch3. . . . 155
4.24 Architecture, CNN workload mapping, and task schedule of multi-array design with a one-short schedule and adaptive partitioning (Arch4). . . . 161
4.25 Partitioning scheme of the optimal designs found for VGG16, ResNet50, and MobileNetV2. . . . 164
4.26 Comparison of latency, throughput, and DSP efficiency of Arch1-4 on different CNNs. . . . 165

5.1 Variants of shuffle instructions. . . . 173
5.2 An example of using shuffle instructions for reduction. . . . 174
5.3 Test results of microbenchmarks. . . . 177
5.4 Dependence graph for sequence alignment algorithms. . . . 179

5.5 PairHMM model. 180

5.6 The mapping of SW and PairHMM to systolic array architectures on GPU. . . . 182

5.7 GPU designs for sequence alignment algorithms with different types of inter-thread communication: a) design A: using shared memory, b) design B: using shuffle. . 182

5.8 Two-level tiling scheme for SW: a) coarse-grained tiling, b) fine-grained tiling. In a), cells on the boundaries that are surrounded by dashed boxes need to be stored back to the global memory. The data on the horizontal boundaries will be retrieved and consumed by the next block, and data on the vertical boundaries are used for finding the maximal value in the last column of the score matrix. . 184

5.9 Shuffle implementation for PairHMM. In this example, each thread will hold six registers for two cells in total and compute them one by one on each anti-diagonal. Inter-thread communication only happens between boundary cells. 186

5.10 Performance overview of GPU implementations: a) SW, b) PairHMM. 189

5.11 Impacts of re-batching on the performance of SW kernels: a) K1200, b) Titan X. 190

LIST OF TABLES

1.1 Normalized energy cost relative to a MAC operation extracted from a commercial 65nm process [CES16]. 6

2.1 Comparison between different systolic array compilation frameworks. 18
2.2 Dependence relations of the MM example. 37
2.3 I/O groups of the MM example. 40
2.4 Summary of the tuning options of AutoSA. 45
2.5 Benchmark description. 50
2.6 Performance comparison of different MM designs. 51
2.7 MM performance comparison with previous works. 52
2.8 LU performance comparison with LAPACK. 54
2.9 Performance results comparison on CNN, MTTKRP, and TTMc. 55

3.1 Comparison between different systolic array architecture auto-tuning frameworks. 64
3.2 Summary of search methods. 66
3.3 Scope of the AutoSA design space explored by Odyssey. 71
3.4 Design space covered by Odyssey for a 1024×1024×1024 MM. 72
3.5 Unique permutation orderings for MM and CNN. 79
3.6 Unique systolic arrays for MM and CNN. 82
3.7 Best solutions found by two mutation methods. 87
3.8 Comparison of the best solutions found by MP-based methods and Odyssey framework. 94
3.9 Parameters of validation workloads. 95
3.10 Comparison of latency and search time between different search methods (search time is represented in (mins:secs)). 98

3.11 Resource breakdown of three designs for MM with the same dataflow $[i]$. 101

3.12 Normalized energy cost relative to a MAC operation extracted from a commercial 65nm process [CES16]. 104

3.13 Performance comparison of three energy-optimal designs for MM with the same dataflow $[i]$. 105

3.14 Performance comparison of dataflows $[h, i]$ and $[o, h]$ on the CONV1 and CONV2 of VGG16. 109

4.1 Details of VGG16, ResNet50, and MobileNetV2 and the DSP efficiency of the best single systolic array for these CNNs on Xilinx Alveo U250. 121

4.2 Comparison of different hardware architecture exploration frameworks for CNN. 121

4.3 Comparison of the explored design space dimensions by different array architectures. 125

4.4 Target systolic array designs for CNN. 128

4.5 Year of release, top-1 prediction accuracy and number of parameters of VGG16, ResNet50, and MobileNetV2. 128

4.6 Comparison of best systolic array designs for three CNNs. 136

4.7 Performance details of best free and fixed architectures on CONV11 of VGG16. 137

4.8 Performance comparison of Arch1 and Arch2 on different CNNs. 151

4.9 Performance comparison between Arch1 and Arch2 on VGG16. 152

4.10 Performance comparison of Arch1 and Arch3 on different CNNs. 157

4.11 Performance breakdown of Arch3 on VGG16. 157

4.12 Layer parameters and the first-level tiling factors for loop h and w in CONV 11 and 12. 158

4.13 Performance comparison of Arch1 and Arch3 on different CNNs. 159

4.14 Performance comparison of Arch1, Arch2, Arch3, and Arch4 on different CNNs. 163

4.15 Performance comparison between Arch2 and Arch4 on ResNet50. 166

4.16 Dataflows of each array in different architectures. 167

5.1 Gap between computation and memory performance on modern GPUs. 171

5.2 Detailed information of the kernels. 191

5.3 Instruction breakdown and analysis for latency reduction. 191

ACKNOWLEDGMENTS

I have to admit that writing the acknowledgement of my dissertation is the most anticipated moment for me from the very beginning of my PhD process. Six years is not a short time in one's life, especially when it is spent on the most challenging task I have ever experienced so far in my life — pursuing a PhD degree. Therefore, starting from the very beginning of this journey, I have been waiting for this moment, wondering what achievements I could make, what memories I would keep, and what a perspective I would hold when putting an end to this emotional trip.

It may not be surprising that the first thing that comes into my mind, when looking back at the past six years, is the hardship of obtaining a PhD. Yes, doing a PhD is not easy, as well as any challenging tasks I have been faced with so far. However, it is different from all the past tasks where there is a foreseeable stop sign at the end. During the PhD journey, there does not seem to be an end or a path. Everything is hidden in the midst of darkness, waiting for me to explore.

With all this being said, there is one person to whom I would like to send my most sincere appreciation and thankfulness. I would not have made this far without him. He has been holding the torch, guiding the way, sending me encouragement, and accompanying me all the way through this trip. Yes, it is my great advisor, Prof. Jason Cong.

My acquaintance with Jason began in the summer of 2014 when I started as a summer intern at Jason's lab. During the internship, I designed an FPGA-based systolic array accelerator in RTL for matrix multiplication. This project was the seed of the entire body of works I have done later during my PhD. More importantly, it was also the first real research project I had worked on. I am thankful to Jason for offering me this opportunity, and also for the guidance he provided during and after the internship. It was this experience that helped me decide to come back to UCLA and pursue my PhD under the guidance of Prof. Cong.

There is a long list of things for me to thank Jason for. The first and foremost item on this list would be the strict training for a systematic and mathematical way of thinking when handling new research problems. We have been joking privately between students calling

Jason a mathematician instead of a computer scientist. I could not count the number of times when I heard the words like "model", "formulation", and "optimality" from Jason when presenting my research results to him. Many times after I excitedly presented some good results, I would be pushed back by Jason asking me to go back to the formulation of my optimization problem and measure how far I am from the optimal points. It could be upsetting at the early stage of my PhD, as it turned out I needed to spend more time figuring out the complete solution space and the optimality of my solutions than coming up with a new solution and collecting the results. However, as time went by, I became more and more grateful for this training which left a profound influence on my way of thinking when handling new problems. Many of my later research works and internship projects have benefited from this training.

The second thing I would like to thank Jason for is the consistent support and especially the tremendous amount of patience he has shown me. There were many times in my PhD process when I was confused or lost about the next step to take. At these times, I would send him emails or go to his office to talk to him. I should admit that most of these times I was concerned and upset about my progress and even doubted if I could make it to the end. I could not appreciate him more for all the guidance and heart-warming encouragement he gave me during all those email exchanges and talks. Especially, the encouragement of "good work takes time" will always hold a special place in my heart. It is these words that have helped me clear my mind, never give up, continue with my efforts, and reach the place I am at now.

There is a lot more to say about Jason such as all the interesting jokes he has made during the group events and the dedication to research and education. I am sure that the past six years at UCLA only marked the beginning of my relationship with Jason, that I will be a life-long student of his, and I will continue to learn from him in the future.

My next acknowledgement would be dedicated to all the smart and lovely people I have met with at UCLA. I would like to thank Peng Zhang. It is your passion about research that touched me most during my internship at UCLA and helped me decide to pursue a PhD degree. Thank you, Hui, for giving me a lot of research advice and offering so much help

during my internship at Google. Thank you, Muhuan, for all the laughter we have shared in countless adventures around LA. Thank you, Cody, for bearing my countless questions and all the advice you have given me. And thank you all, Weikang, Bing, Licheng, Zhengrong, Young, Jian, Meng, Chen, Peng Wei, for all the unforgettable memories at the beautiful campus of UCLA. Thank you, Yuze and Jason Lau, for maintaining a top-tier server experience at the research labs. Thank you, Alex, for all your kindly help to the lab members over the years. And thank you, Janice and Marci for helping edit my research papers. My special thanks should go to Xinfeng and Peipei, for accepting all my moods, being a constant support, and making me believe in what I am capable of achieving in the past six years. I would also like to thank Huizi, Peng Gu, Xiling, Ruijia, and Yuting, for being my best friends and sharing all the great life moments with me from the beginning of college to now.

Many thanks have to be given to all the professors and fellow researchers I met with through my PhD studies. I am truly impressed by the achievements of three marvelous professors on my committee. I am grateful for the thought-provoking question regarding the "real challenging task" in my oral presentation from Prof. Jens Palsberg. Your question motivated me to aim high for all my challenges in the future. I appreciated all the advice and kindly help from Prof. Miloš D. Ercegovac. You have demonstrated to me what a life-long researcher and educator looks like and I hope you enjoy every moment of retirement. I would like to give my special respect and appreciation to Prof. Tony Nowatzki, for living as a role model for what a young professor should be, for your remarkable standard and passion for research, and for the care in which you showed to your students. I would also like to thank the following professors and fellow researchers for all the fruitful discussions we had and your selfless help: Prof. Zhiru Zhang from Cornell, Prof. Miryung Kim from UCLA, Prof. Monica S. Lam from Stanford, Prof. Louis-Noël Pouchet from Colorado State University, Xuechao Wei from Peking University, Sean and Shaojie from Cornell. And I would also like to thank all my great mentors during my internships, including Andreas Nowatzyk from Google (now at VMware Research), Hongbo Rong from Intel Parallel Computing Labs, and Yida Wang from AWS.

Finally, I am saving the best part for the two most important people in my life — Hongbei

Wang and Dinghui Liu. Thank you for bringing me into this world and making me the person I can proudly say I am now. And thank you both for bearing with me in this long PhD journey through the countless phone calls over the sea across days and nights. You deserve so much credit that I would say that my PhD degree should be awarded to three names, the two of you and me. Thanks again for all your love and support over the past 28 years. Thank you.

The research studies in this dissertation are partially supported by C-FAR, one of the six SRC STARnet Centers, sponsored by MARCO and DARPA, NSF/Intel Innovation Transition Grant (CCF-1436827) awarded to the Center for Domain-Specific Computing (CDSC); the CAPA program jointly funded by the NSF (CCF-1436827) and Intel; the ICN-WEN Award jointly funded by NSF (CNS-1719403) and Intel; NSF NeuroNex Award DBI-1707408; and the members from the CDSC Industrial Partnership Program. We acknowledge the valuable support from the Xilinx Adaptive Compute Clusters (XACC) program.

VITA

2011–2015 B.S., Electronic Engineering, Tsinghua University, Beijing, China.

2012–2015 B.S., Economics, Tsinghua University, Beijing, China.

PUBLICATIONS

Jie Wang, Licheng Guo, and Jason Cong. "AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA." *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2021.

Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecener Ustun, Zhiru Zhang, and Jason Cong. "AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs." *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2021. (**Best Paper Award**)

Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, Youhui Zhang, Jason Cong, Nithin George, Jose Alvarez, Christopher Hughes, and Pradeep Dubey. "SuSy: A programming model for productive construction of high-performance systolic arrays on FPGAs." *2020 IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, 2020.

Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. "End-to-end optimization of deep learning applications." *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2020.

Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Yu Cody Hao, Yuan Zhou, Jason Cong, and Zhiru Zhang. "HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing." *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2019. (**Best Paper Award**)

Jie Wang and Jason Cong. "PolySA: Polyhedral-based systolic array auto-compilation." *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2018.

Jie Wang, Xinfeng Xie, and Jason Cong. "Communication optimization on GPU: A case study of sequence alignment algorithms." *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, 2017.

Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. "Going deeper with embedded FPGA platform for convolutional neural network." *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA)*, 2016.

CHAPTER 1

Introduction

1.1 Background

We are now inhabiting a world where massive amounts of data are produced every day. As an example, in 2019, Youtube reported that there are more than 500 hours of videos uploaded to their website per minute [Rut19]. To put this into perspective, it will take one person nearly 82 years to finish watching all the videos uploaded to Youtube in only one hour.

Processing such an unprecedented amount of data puts a significant challenge to the modern computing systems. We have been witnessing an exponential growth of CPU processor performance in the past few decades. Such a growth is mostly driven by the technological progress (e.g., feature size scaling, frequency scaling) and micro-architecture improvements. However, the end of Dennard scaling [DGR74] has put an end to this growth as the processor hits the power wall that halts the frequency scaling [EBA11]. As a result, we have entered a new era of domain-specific architectures (DSA), which are defined as a group of customized hardware architectures developed for a particular domain of applications [JYP18]. To understand this trend better, we present the following equation:

$$P = \frac{Energy}{Op} \times \frac{\#Ops}{s} \tag{1.1}$$

The total power P of a processor is computed by the product of the energy consumed per operation ($Energy/Op$) and the operations performed per second ($\#Ops/s$). As the total power is capped by the power wall, to further improve the performance ($\#Ops/s$), we need to boost the energy efficiency ($Energy/Op$) of the current hardware.

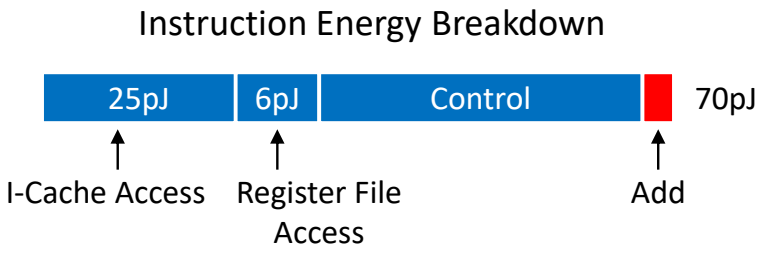


Figure 1.1: Instruction energy breakdown for an in-order processor in 45nm [Hor14].

Figure 1.1 shows the instruction energy breakdown of a simple in-order processor on the 45nm technology node [Hor14]. As illustrated in the figure, the majority of the energy is spent on the control and memory access, leading to the low energy efficiency of general processors [CGG14]. In comparison to general processors, DSAs customize the control and data paths to reduce the energy overheads on these components. As a result, DSAs usually achieve orders of magnitude higher performance and energy efficiency than the general processors.

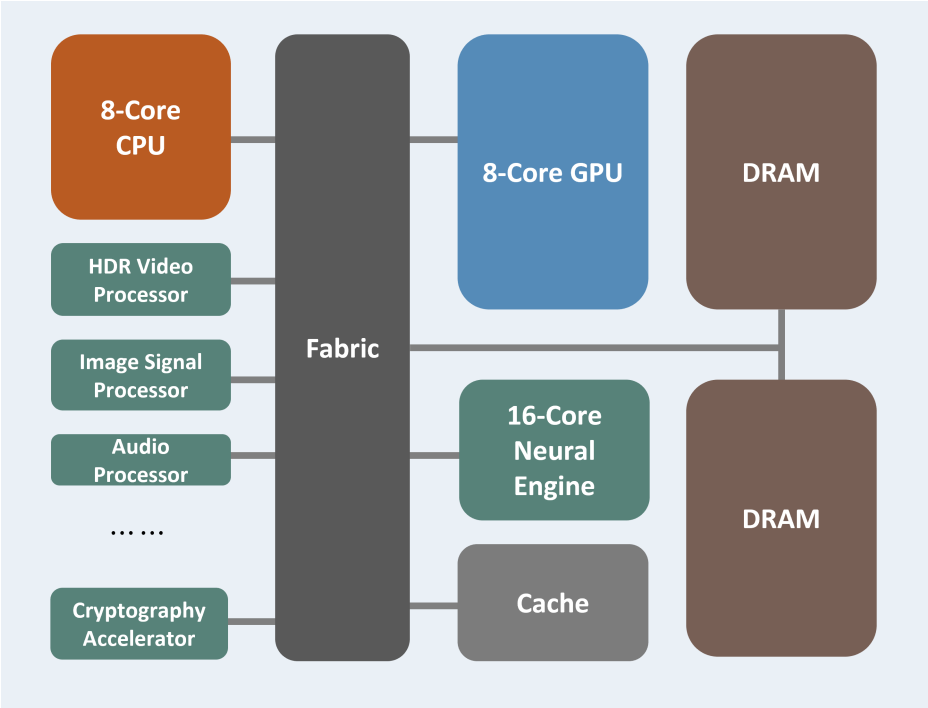


Figure 1.2: Block diagram of the Apple M1 SoC.

New DSAs are emerging everywhere: from the continuous efforts of academia [WLD20, CES16, SQA20, KSK18, SWC20, ZYW20] to the exciting products coming from indus-

try [JYP17, Ama20, Wik19, CCP16]. The latest Apple M1 chip (as seen in Figure 1.2) is one such example. In addition to the 8-core CPU, M1 integrates a great number of heterogeneous accelerators including a 8-core GPU with up to 2.6 TFLOPs of throughput, a neural processing engine that targets DNN processing with up to 11 TOPs of throughput, and a group of other DSAs dedicated for different applications such as HDR video processing, image signal processing, and cryptography.

Among the sea of DSAs, there is a special category of architectures named *systolic array* that has generated a great deal of interest. The term *systolic array* was first coined by Kung and Leiserson in 1978 [KL79]. This architecture is composed of a grid of regular and simple processing elements (PEs) that are connected via local interconnects. The name, *systolic*, is derived from the computation pattern in the systolic array that the input and output data are flowing through the network of PEs in a rhythmic fashion.

Figure 1.3 depicts a two-dimensional systolic array for the matrix multiplication problem shown below.

$$C = A \times B, \text{ where } A, B, C \in \mathbb{R}^{3\times 3} \tag{1.2}$$

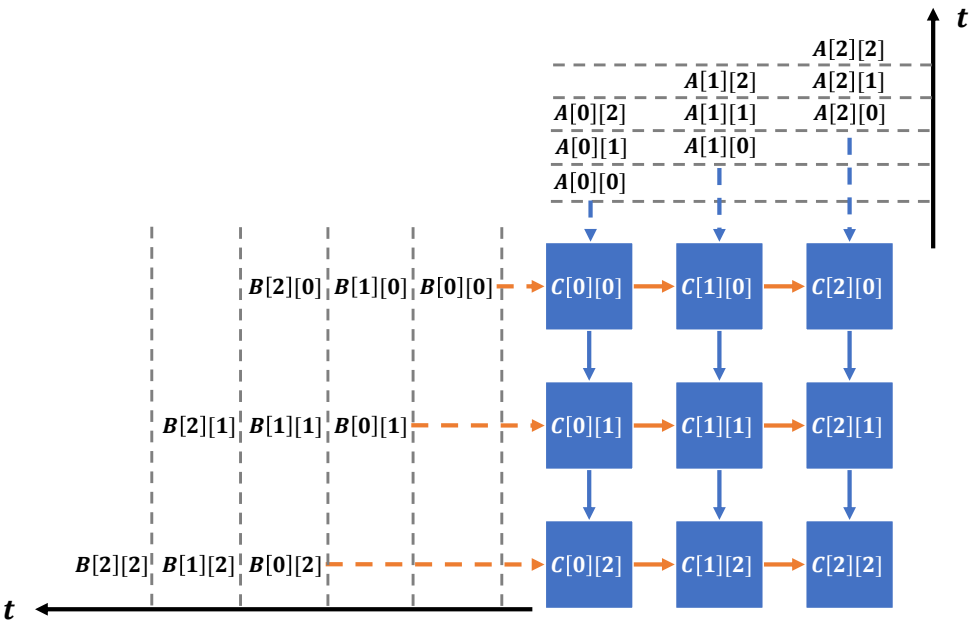


Figure 1.3: An example of a 2D systolic array for matrix multiplication.

Elements of matrix A and B are fed from two sides of the array. Each PE computes

one element of matrix C. At every cycle, each PE consumes one element from A and one element from B, performs local multiply-and-accumulate (MAC) to update the element of C, and passes the input data from A and B to neighboring PEs. After several rounds of computation, the final results of C are generated inside each PE and drained out through the local interconnects.

Systolic arrays have two major architectural benefits: 1) a simple and regular design, 2) a balance between computation and communication. As shown in Figure 1.3, systolic arrays exploit the parallelism by a grid of identical PEs with local interconnects. Such a simple architecture can be easily scaled out with minimal additional developing costs. In addition, systolic arrays take advantage of the data locality by reusing the data through local interconnects. This feature helps generate a more balanced design with the potential to achieve both high performance and energy efficiency.

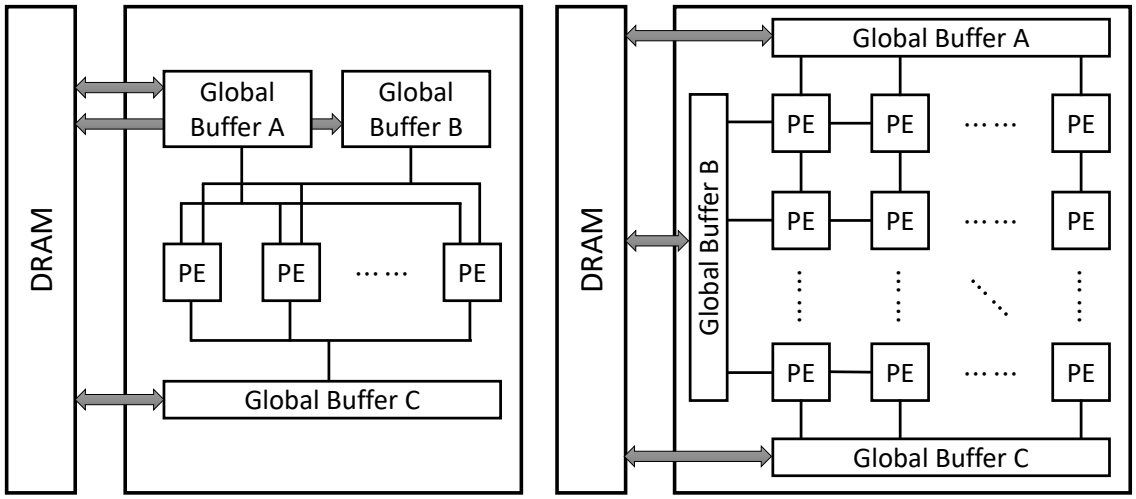
With these benefits, following the introduction of systolic arrays, there has been intensive research on this architecture in the following two decades. Many systolic array designs were proposed for different applications, such as linear algebra [GK82], signal processing [Kun88, KY85], image processing [KY82], and dynamic programming [LL85]. The list keeps growing. Apart from these accelerators designed for one specific application, there have been projects on developing programmable systolic array architectures targeting a domain of applications [RRA93, SSS97] and general applications [BCC88, PSW91].

Despite the continuous interest from academia, systolic array hasn't found its way into the real world. One of the significant milestones is the iWarp machine [PSW91], which is a general-purpose systolic array processor designed by researchers from Carnegie Mellon University and Intel in the 1990s. However, this effort was discontinued several years after the first release of the machine.

The reasons for such a dilemma are multi-fold. First, most systolic array designs in publications were still "paper designs" which only focus on the designing of the computation array while overlooking the memory system and the integration with the host processor. Systolic array designs that consider the memory system and integration mostly targeted

VLSI circuits, and those require significant development costs. The last yet most important reason is that the performance gains of such an architecture were easily overshadowed by the exponential performance improvement of general processors that took advantage of Moore's law.

However, with the new challenge in the computer architecture community nowadays and several technology advancements, we believe that now is a good time to revive the systolic array architecture. We draw this conclusion from the following three points:



(a) Broadcast architecture (b) Systolic array architecture

Figure 1.4: Comparison between a broadcast architecture and a systolic array architecture for matrix multiplication.

The need for architectures with higher energy efficiency The challenge of the power wall has created a strong need for energy-efficient architectures. Systolic array is proven to be one of the architectures that meets this requirement. It reduces the energy of control logic with a simple lock-step execution model and minimizes the energy of memory access with local communication. Figure 1.4 compares a broadcast architecture with a systolic array for matrix multiplication. In the broadcast design, data of matrix A and B are broadcast to PEs through on-chip crossbars. The results of matrix C are gathered via another crossbar to on-chip global buffers. In comparison, in the systolic array, data of matrix A, B, C are sent or collected at the array boundary, and transferred across PEs via local interconnects.

Table 1.1: Normalized energy cost relative to a MAC operation extracted from a commercial 65nm process [CES16].

	DRAM	Global Buffer (>100kB)	Array (Inter-PE) (1-2mm)	RF (0.5kB)
Normalized Energy	200×	6×	2×	1×

Table 1.1 lists the normalized energy cost of memory accesses at different memory levels against a MAC operation on a commercial 65nm technology node [CES16]. According to this table, inter-PE communication consumes 3× less energy than the global buffer access, suggesting the potential of achieving higher energy efficiency on systolic arrays compared to the broadcast designs. Based on this observation, authors in [CES16] implemented a systolic-array-like architecture named Eyeriss for deep learning applications and achieved a higher energy efficiency compared to other hardware designs. In the recent years, systolic arrays have been widely adopted for accelerating deep learning applications [CES16, JYP17, Wik19, KMZ19, Xil19b, SWC20]. Several representative examples include the tensor processing unit (TPU) from Google [JYP17], the neural network accelerator from Tesla [Wik19], and the Xilinx ML accelerator [Xil19b].

The advancement of heterogeneous platforms Heterogeneous hardware, such as the graphic processing unit (GPU) and field programmable gate array (FPGA), is playing a more and more important role in the modern computing systems. These platforms usually deliver several orders of magnitude higher performance and energy efficiency than CPU.

Both platforms can be targeted for mapping systolic arrays and achieve a much lower development cost than VLSI circuits. FPGA is a natural target for implementing systolic arrays. The island-style architecture matches the grid structure of systolic arrays perfectly. We could start by mapping one PE to FPGA and easily scale out by replicating PEs to occupy the entire chip. In addition, the local interconnects of systolic arrays benefit the routing process on FPGA, leading to a design with both high resource utilization and frequency. There exists a large body of works mapping systolic array to FPGA for a wide range of application

domains such as linear algebra [MKN18, SRB19], machine learning [WYZ17, WLL18], and genomics [RF16, IO16].

As for GPU, we could use systolic array as an overlay above the existing GPU architecture. We keep the same execution model of systolic arrays and map different components of the array to the existing hardware. For example, PEs of systolic arrays can be mapped to GPU threads. The local communication between PEs can be realized by the existing inter-thread communication mechanisms such as shared memory or global memory. On Nvidia GPUs, starting from the Kepler architecture, a new instruction called *SHFL*, or *"shuffle"* [Har14], was introduced. The shuffle instruction enables threads within the same warp to directly exchange data with each other at the register level. This instruction adds to the existing inter-thread communication mechanisms on GPU and achieves a shorter communication latency than the other mechanisms. We have included one work in this dissertation that illustrates the mapping of two sequence alignment algorithms in genomics pipeline to GPU with a systolic array overlay [WXC17]. There have been several recent works that have achieved a similar success as ours in mapping systolic arrays on GPU for other applications like stencil computation [PEW19, CWT19]. This research field is relatively new. Given the benefits that have been demonstrated by the initial studies, we are expecting more future research showing up along this direction.

The maturing of compilation tools The barriers to entry for systolic arrays are incredibly high as designers need a thorough understanding of both the application characteristics and hardware architecture. Automating such a designing process has been an important topic since the birth of systolic arrays. Many previous works have attempted to overcome this challenge [YCJ96, SM00, GQQ01, LL97, SAM02, BT03]. These works use a technique named space-time transformation [Kun88] to transform algorithms to systolic arrays and produce RTL code targeting VLSI circuits or FPGA. Regardless of the rich body of the previous works, the manual designing process remains the norm. This is due to the fact that the previous automatic systolic array synthesis works failed to achieve a balance between the three metrics, generality, performance, and productivity, all of which are critical to the designing

process. As an example, several previous works use customized inputs such as recurrence equations [YCJ96, SM00, GQQ01, BT03], which requires non-trivial efforts to implement for average programmers, thus lowering productivity. There are frameworks [SAM02, BRS07] that take C programs as inputs. However, these works are limited in other factors such as the generality of the framework in terms of the types of program they support or the performance of the generated design.

However, with the continuous advancement in the compiler community, especially two noticeable technological improvements in the recent decade, we have found new opportunities to embrace the idea of automatic systolic array synthesis again. The first improvement is the maturation of polyhedral compilation tools. The polyhedral compilation framework is a loop transformation framework that has a strong relationship with systolic array synthesis. It can be used to conduct the accurate dependence analysis and apply space-time transformation for generating systolic arrays. The polyhedral model used to be "blamed" for its limited scope of the supported programs and the immature compilation tools. However, over the years, the polyhedral model has been extended to support a broad range of applications that cover the majority of high-performance computing applications [BPC10, Ver10] and had been integrated with modern compilation infrastructure such as LLVM [GZA11], MLIR [LAB20], and Halide [VZT19]. Numerous tools based on the polyhedral framework have been developed targeting different devices like CPU [BHR08, Pou21], GPU [BRS10, VCC13, BRR19], and FPGA [PZS13, BRS07].

The second improvement comes from the high-level synthesis (HLS) techniques. HLS tools raise the abstraction level of hardware programming model from the conventional RTL to high-level programming languages like C/C++. It reduces the development cycles by a great amount and generates designs with a performance comparable to manual designs [CM08, CLN11]. Commercial tools such as Xilinx HLS [Xil19a], Intel OpenCL [Int21], and LegUp [Leg19] are being adopted by both academia and the industry at an increasing speed.

The advancement of the polyhedral compilation techniques and HLS tools paved the way for automatic systolic array synthesis. The polyhedral model offers a general and robust

compilation framework for transforming the applications written in high-level programming languages like C to systolic array designs. Such designs can be described by HLS languages and ported to different hardware in a short amount of time. We demonstrate our efforts in this dissertation with a polyhedral model-based systolic compilation framework that is capable of generating systolic arrays with a similar or better performance than manual designs and reducing the development time from weeks to days.

1.2 Overview of Dissertation

The quest for efficient architectures and the advancement of modern hardware and compilation techniques have motivated us to focus on systolic arrays as one promising solution to coping with the challenges in the post-Moore's law age. In this dissertation, we make attempts in the field of *automatic systolic array synthesis* to make this architecture accessible to average programmers and perform *architecture optimizations* of systolic arrays for different hardware and applications. We divide the works in this dissertation into two parts: automatic systolic array synthesis and architecture optimization.

1.2.1 Automatic Systolic Arrays Synthesis

We have pointed out the limitations of previous works on automatic systolic array synthesis which fail to achieve a balance between three metrics: generality, performance, and productivity.

To overcome these limitations, in Chapter 2, a new automatic systolic array compilation framework, AutoSA [WGC21], is proposed. AutoSA is an open-source compilation framework[1] for generating systolic arrays on FPGA. Figure 1.5 shows the overall compilation flow. AutoSA is built on the polyhedral framework [Ver10] which supports accuracy dependence analysis and loop transformation. It takes applications in C language and maps them to systolic arrays through a sequence of computation and communication optimizations. At present, AutoSA

[1]https://github.com/UCLA-VAST/AutoSA

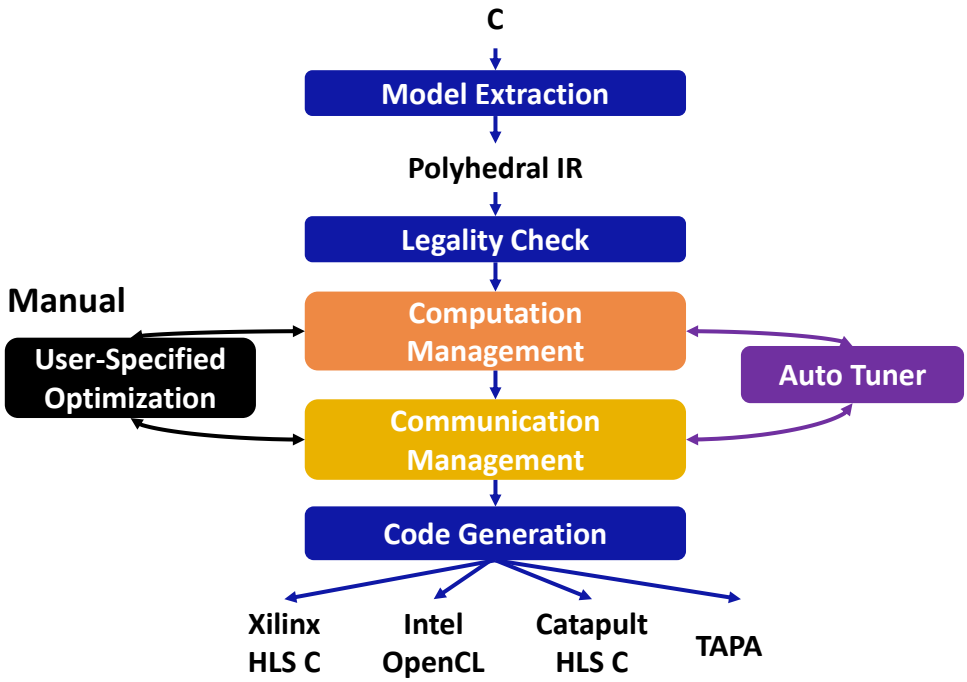


Figure 1.5: Compilation flow of the AutoSA framework.

supports four hardware back-ends, including Xilinx HLS [Xil19a], Intel OpenCL [Int21], Catapult HLS [Sie21], and TAPA [CGL21].

We have demonstrated AutoSA on a wide range of applications, on which AutoSA generates high-performance systolic arrays within a short amount of time. As an example, for matrix multiplication, AutoSA achieves 934 GFLOPs, 3.41 TOPs, and 6.95 TOPs in floating point, 16-bit and 8-bit integer data types on Xilinx Alveo U250, which is similar or better than the best performance reported by other state-of-the-art systolic array compilation frameworks [JLW20, LRZ20, SRB19].

AutoSA implements a rich set of hardware techniques which constructs a large design space that is intractable to explore manually. For example, for a $1024 \times 1024 \times 1024$ matrix multiplication, AutoSA generates a design space including $O(2^{40})$ design points which is infeasible to handle by manual exploration. To tackle this challenge, in Chapter 3, we introduce an auto-tuning framework, Odyssey[2], to explore such a design space efficiently.

[2]Odyssey is abbreviated from AUtomatic DEsign Space exploration for SYstolic arrays.

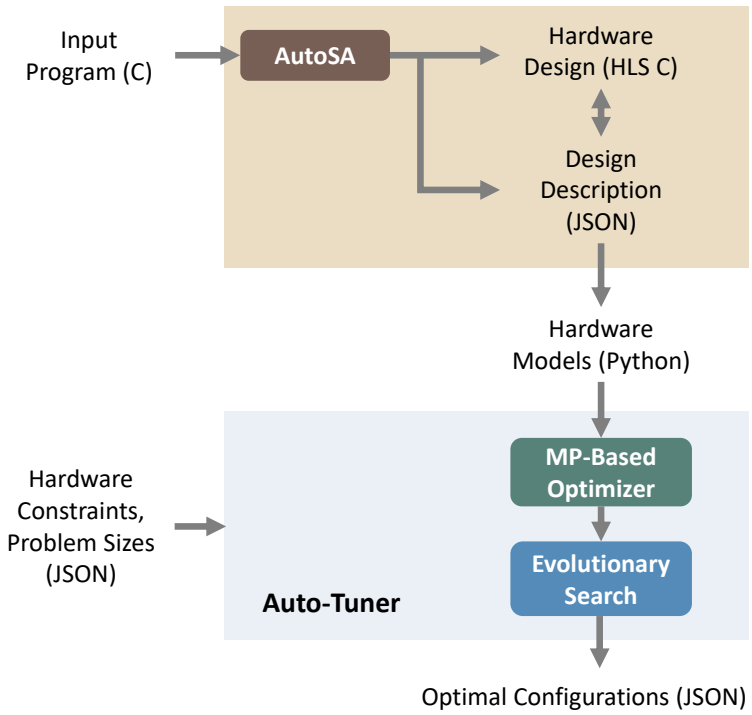


Figure 1.6: Overview of the Odyssey auto-tuning framework.

Figure 1.6 depicts the overview of Odyssey framework. Odyssey leverages AutoSA to build the design space automatically. We have extended AutoSA to generate a design description file in JSON format that covers the full details of the generated hardware. Odyssey builds Python functions for estimating the design performance including the latency and resource usage from the description file. The generated performance model is then plugged into the auto-tuner to search for the optimal design configurations. Inside the auto-tuner, Odyssey implements a two-stage flow which consists of a mathematical programming (MP)-based optimizer and an evolutionary search-based auto-tuner. The MP-based optimizer uses an optimization solver to generate a high-quality design. This design is used as the initial population of the evolutionary search. The use of initial samples from the MP optimizer helps reduce the convergence time and improves the search quality.

Odyssey can find designs with high performance in a short amount of time, For example, for the $1024 \times 1024 \times 1024$ matrix multiplication, it locates a design that achieves 90% of the optimal performance found by an exhaustive search in only 5 seconds. Odyssey is

open-sourced[3] and seamlessly integrates with AutoSA. Combining both tools together, we can generate high-performance systolic arrays for a wide range of applications, and reduce the development time from weeks to days, surpassing priors works in all three dimensions in terms of generality, performance, and productivity.

1.2.2 Architecture Optimization

In the second part of the dissertation, we demonstrate two deployment cases of systolic arrays on different applications and platforms.

The first work in Chapter 4 focuses on one important application, convolutional neural network (CNN). CNNs pose unique challenges to systolic arrays given its high computation complexity and the divergence of performance characteristics such as parallelism and data locality across CNN layers. Numerous design factors need to be taken into account when mapping CNNs to systolic arrays. For example, previous works have shown that the performance of CNN layers varies across systolic arrays with different configurations [CES16, KCS20, KLP21, HKD21, YGL20]. Therefore, mapping CNNs to a single monolithic array with a fixed architecture may result in a limited performance. Our study in this dissertation shows that such an architecture suffers from severe computation resource underutilization. The best single systolic array for VGG16 network [SZ14] we have found achieves only 73% DSP efficiency[4] on Xilinx Alveo U250. This number further degrades to 35% and 19% on two smaller networks, ResNet50 [HZR16] and MobileNetV2 [SHZ18], respectively. There exists a large amount of previous works that have proposed different architecture enhancements to tackle this challenge [CES16, WYZ17, MKN18, KCS20, YGL20, HKD21]. In our work, we consider three design factors, including the choice of dataflows, the number of systolic arrays, and the layer-wise execution model. We evaluate three different architecture candidates: a single monolithic systolic array and two multi-array architectures proposed by the previous works DNNExplorer [ZYW20] and TGPA [WLL18]. Our evaluation results show

[3]https://github.com/UCLA-VAST/AutoSA

[4]DSP efficiency is computed as the ratio of the actual throughput achieved by the design over the maximal ideal throughput achieved when fully utilizing the DSP resources of the design.

that multi-array architectures which implement several smaller arrays customized for different CNN layers improve the system throughput with a cost of longer latency. Furthermore, such a performance trade-off varies across different multi-array configurations, and such trade-offs need to carefully examined when choosing the correct configuration for the target hardware and CNN.

While the proposed multi-array architectures in the previous works demonstrated its effectiveness in improving the design performance, we found several pitfalls in the proposed architectures that prevent them from achieving the optimal performance. For example, DNNExplorer [ZYW20] used a rigid network partititioning algorithm which splits the CNN into two halves such that, before the splitting point, all the former layers are mapped to separate arrays and, after the splitting point, the latter layers are grouped and mapped to a single array. As a consequence, the last array is faced with the similar performance issues of low computation efficiency as a single monolithic array. To further improve the performance, we propose a new multi-array architecture with an adaptive workload partitioning scheme which groups CNN layers with similar characteristics together and maps them to the same array to improve the overall performance. This architecture achieves the highest throughput among all the evaluated architectures. Specifically, it achieves a 1.3×, 2.6×, and 5.8× throughput improvement over the single array baseline on VGG16, ResNet50, and MobileNetV2, respectively. The DSP efficiency on these three networks are improved from 73%, 35%, 19% on single systolic array to 90%, 91%, 74%, respectively.

This work demonstrates the complexities of mapping a complicated real-world application like a CNN to systolic arrays. While AutoSA and Odyssey offer an efficient approach to generating systolic arrays for a singe workload (one CNN layer), the disparity of performance characteristics of CNN layers brings new dimensions to the design space and requires additional architecture optimizations to achieve high performance.

The next work in Chapter 5 explores implementing systolic arrays on GPUs using two sequence alignment algorithms from the genomics pipeline as case studies. In this work, we use the systolic array architecture as an overlay upon the existing GPU architecture and map different components of the systolic array to the existing GPU architecture. The major

contribution of work lies in the exploration of efficient inter-PE communication mechanisms on GPU. There used to be limited choices for inter-thread communication on GPU where GPU threads could only exchange data between each other through global memory or shared memory. However, starting with the Kepler architecture, Nvidia introduced an instruction named SHFL, or "shuffle" [Har14], which allows threads within a warp to directly exchange data with each other.

We start with building a suite of microbenchmarks that evaluate the latency of different shuffle instructions. The evaluation results show that shuffle instructions have the least latency among all inter-thread communication mechanisms. Based on such findings, we implement the inter-PE communication using the shuffle instructions for two sequence alignment algorithms, Smith-Waterman (SW) and Pairwise-Hidden-Markov-Model (PairHMM) and compare the performance to baselines using the shared memory. Experiments results show a speedup of 1.2× and 2.1× by using shuffle instructions for SW and PairHMM over the shared memory baselines.

1.3 Summary

The works in this dissertation extend the research on systolic array architectures in two dimensions: automatic systolic array synthesis and architecture optimization. The contributions of our work are summarized as below.

Novel and efficient compilation and auto-tuning approaches to reduce the programming efforts of systolic arrays Making the best use of systolic array architectures requires a decent knowledge in both the software and hardware worlds. The new compilation and auto-tuning approaches proposed in this work help bridge the gap of the two worlds by providing efficient solutions for generating high-performance systolic array designs with minimal programming efforts. This will help make systolic arrays accessible to the average programmers and increase the level of acceptance of this architecture in the future computing systems.

Comprehensive and effective architecture optimization techniques of systolic arrays on different applications and hardware platforms The evolution of applications and hardware have brought new challenges and opportunities to systolic array architectures. We show the complexities and trade-offs when mapping real-world applications to this architecture through two case studies. Besides, we demonstrate that with proper architecture optimizations, systolic array architecture is capable of delivering high performance and high energy efficiency. We hope the studies in this dissertation will serve as a reference and open the gate to more research employing systolic arrays on various applications and hardware in the future.

The remainder of this dissertation is organized as follows. Chapter 2 presents AutoSA, an end-to-end systolic array compilation framework on FPGA. Chapter 3 introduces the auto-tuning framework, Odyssey, which performs efficient design space exploration for systolic array designs generated from AutoSA. Based on AutoSA and Odyssey, Chapter 4 investigates architecture optimization techniques when mapping CNNs to systolic arrays. Next, Chapter 5 discusses details about our efforts in mapping systolic arrays to GPU. Chapter 6 concludes the whole dissertation and discusses future works.

CHAPTER 2

Automatic Systolic Array Synthesis

While systolic array architectures have the potential to deliver tremendous performance, it is notoriously challenging to customize an efficient systolic array for a target application. Designing systolic arrays requires knowledge for both high-level characteristics of the application and low-level hardware details, thus making it a demanding and inefficient process. To relieve users from the manual iterative trial-and-error process, we present *AutoSA*, an end-to-end compilation framework for generating systolic arrays on FPGA. AutoSA is based on the polyhedral compilation framework, and further incorporates a set of optimizations on different dimensions to boost performance. We have demonstrated AutoSA on a wide range of applications, on which AutoSA generates designs with high performance within a short amount of time. This chapter discusses the details of the AutoSA compilation framework. As will be shown in this chapter, the large set of optimization techniques that AutoSA incorporates introduce a vast design space which is intractable to handle manually. We have also developed an efficient auto-tuner named Odyssey to handle this challenge. Details of the auto-tuning framework will be covered in Chapter 3.

2.1 Background and Related Work

The systolic array architecture is capable of delivering high performance for a wide range of applications, such as linear algebra [MKN18], signal processing [KY85, CY89, WD08], machine learning [WYZ17], and genomics [RF16]. In recent years, we have also seen a wide adoption of systolic array architectures in the field of deep learning [CES16, GHI19, KMZ19, CHW20, WYZ17, Wik19, JYP17, Ama20].

However, designing high-performance systolic arrays is never an easy task. It requires the expert knowledge for both the target application and the hardware. Specifically, designers need to identify the systolic array execution pattern from the application, transform the algorithm to describe a systolic array, write the hardware code for the target platform, and tune the design to achieve optimal performance. Each step will take significant efforts, raising the bar to reap the benefits of such an architecture. As an example, a technical report from Intel [Ron17] mentioned that it took months of efforts for designing systolic arrays even for industry experts.

To lower the programming efforts of systolic arrays, there is an active research domain to automate the systolic array generation process [GQQ01, SAM02, BT03, BRS07, CW18, SRB19, LRZ20, WYZ17, GHI19]. We classify these works into three categories.

Polyhedral compilers The polyhedral model is a compilation framework for loop transformation [Ver10, BPC10, LL97, LWB17, PZS13]. Most prior systolic array compilers are built upon the polyhedral framework [GQQ01, BT03, SAM02, BRS07, CW18]. The commonality of these frameworks is the use of space-time transformation [Kun88, LQR99] to convert an algorithm into a new program that describes the architecture and execution of the systolic array. The support for automatic loop transformation by the polyhedral model helps improve the productivity, however, most of these compilers fail to deliver performance on par with manual designs due to the missing of several important hardware optimization techniques that are critical to the design performance. The first two columns of Table 2.1 list two representative polyhedral frameworks. For example, MMAlpha [GQQ01] does not support

array partitioning, which is essential in handling large-scale programs given the limited hardware resource. The tool proposed by Bondhugula et al. [BRS07] only implemented array partitioning and missed other important techniques such as latency hiding and double buffering. As a result, both works generated designs with limited performance.

Table 2.1: Comparison between different systolic array compilation frameworks.

Feature	MMAlpha [GQQ01]	Bondhugula et al. [BRS07]	SuSy [LRZ20]	PolySA [CW18]	AutoSA [WGC21]
Generality					
Imperfectly Nested Loops	No	No	No	No	Yes
Multi-Statement	Yes	No	Yes	No	Yes
Performance					
Array Partitioning	No	Auto	Semi-Auto	Auto	Auto
Latency Hiding	No	No	Semi-Auto	Auto	Auto
SIMD Vectorization	No	No	Semi-Auto	Limited	Auto
Double Buffering	No	No	Semi-Auto	Auto	Auto
Data Packing	No	No	Semi-Auto	Limited	Auto
Productivity					
Input	DSL	C	C	DSL	C
Auto-Tuning	No	No	No	Yes	Yes
Space-Time Transformation	Semi-Auto	Auto	Semi-Auto	Auto	Auto

DSLs There are several recent works that implemented a domain-specific language (DSL)-based compiler for generating systolic arrays [SRB19, LRZ20, YGL20]. T2S-Tensor [SRB19] is a DSL compiler extended from Halide [RBA13] for generating systolic arrays for tensor programs. Following the similar design principle of Halide, T2S-Tensor explicitly decouples the compute and communication optimization of systolic arrays. T2S-Tensor is limited to supporting only tensor programs with a single statement in perfectly nested loops. The followup work SuSy [LRZ20] further improved T2S-Tensor by supporting programs with multiple statements in perfectly nested loops. Both T2S-Tensor and SuSy achieved high performance on FPGA. However, since Halide does not have dependence analysis and relies on conservative rules to determine the legality of program transformation, these compilers are semi-automatic and require programmers to analyze the systolic array execution pattern from the algorithms and specify the necessary program transformations in the input DSL program. For instance, SuSy [LRZ20] requires programmers to transform the program into a special form of recurrence equations [Kun88]. Data reuse should be explicitly specified by users in SuSy. As we will show later, such a job is challenging when handling complicated applications with irregular dependence structure. Furthermore, neither tools supported auto-tuning. All

of these limitations have reduced the productivity of these tools.

Other frameworks There are some other frameworks that target a specific domain of applications [MKN18, GHI19, WYZ17, FKH20, JLW20, DLH19, GKP19]. Gemmini [GHI19] is a framework for generating systolic arrays for matrix multiplication. It uses a code template that can be reconfigured to generate different arrays. Wei et al. [WYZ17] implemented a template-based code generator for CNN. These frameworks achieved high performance with many application-specific optimization techniques. However, the mapping mythology is limited to specific applications and cannot extend to general ones.

In summary, we found that previous works face the following limitations that prohibit them from being used in practice:

- Limited *generality*. Works such as [LRZ20, BRS07, SRB19] place rigid restrictions on the input programs that limit the application scope of the compiler.
- Limited *performance*. Works such as [GQQ01, BT03, SAM02, BRS07] support limited program optimizations that restrain the performance of the generated designs.
- Limited *productivity*. Works such as [LRZ20, SRB19, YGL20] require programmers to analyze the program and specify the program transformations manually. This requirement stretches the development cycles, leading to low productivity.

2.2 Contributions

To overcome the limitations of previous works, in this chapter, we propose a new compilation framework, *AutoSA*. AutoSA is a polyhedral model-based compilation framework. The top priority for us is to improve the generality to support compiling any programs to systolic arrays, as long as they can be supported by the polyhedral framework and can be legally mapped to a systolic array. AutoSA supports SCoP programs[1] with imperfectly nested

[1]SCoP, as an abbreviation for Static Control Part, is a category of programs that can be supported by the polyhedral model. Details will be introduced in Section 2.3.

loops and multiple statements. To compile such programs to systolic arrays, we propose a set of comprehensive and novel techniques for automatic translation of regular sequential programs to parallel ones that describe a complete system of systolic arrays, including both the processing elements (PEs) and the on-chip I/O network. We would like to emphasize that the compilation of even this restricted class of programs is very challenging and that no automatic and general solution exists despite decades of research. To support these transformations, previous works either rely on a semi-automatic workflow that requires users to determine the transformation manually prior to the compilation (e.g., MMAlpha [GQQ01], SuSy [LRZ20]), or restrain themselves to a narrow set of programs and techniques (e.g., Bondhugula et al. [BRS07], PolySA [CW18]). In this work, we demonstrate that AutoSA is not only able to handle simple and regular applications such as matrix multiplication and convolution, but also supports applications with complicated and irregular dependence structures like LU decomposition.

In addition, AutoSA further improves performance and productivity. AutoSA covers a superset of all the previous optimization techniques that have been applied on systolic arrays and extends with new techniques to further improve the performance. Several representative techniques that AutoSA supports can be found in Table 2.1. For example, SIMD vectorization is an important technique to increase parallelism and resource efficiency. AutoSA supports auto-detection of vectorizable loops with rigid dependence analysis and performs automatic program transformation and code generation to generate a vectorized design. In comparison, such a feature is either not supported in the previous work, or requires human assistance to analyze and transform the program [CW18, LRZ20, SRB19]. The missing of such an optimization opportunity could lead to final designs with sub-optimal performance.

AutoSA takes C code as the input. The entire compilation flow is automated with minimal human intervention. Besides, an auto-tuning module is implemented to ease the efforts of design space exploration. In the evaluation section, we show that programmers are able to generate high-performance systolic arrays with AutoSA in days.

The rest of this chapter is organized as follows. We cover the background of polyhedral compilation framework and space-time mapping in Section 2.3. Then we discuss our previous

work, PolySA in Section 2.4. PolySA marks our first attempt in automatic systolic array synthesis. While achieving competitive performance results on two applications, matrix multiplication and CNN, PolySA is limited in generality and productivity. Next, in Section 2.5, we introduce our followup work AutoSA that overcomes all these limitations of PolySA and provides a general and efficient solution to automatic systolic array synthesis.

2.3 Theoretical Background

2.3.1 Polyhedral Compilation Framework

2.3.1.1 Polyhedral Program Representation

The polyhedral model is a mathematical framework for loop nest optimization. Loop nests that satisfy the requirements of the polyhedral model are called *Static Control Part* (SCoP) [BHR08, BPC10]. A SCoP is defined as a set of statements with loop bounds and conditions as affine functions of the enclosing loop iterators and the variables that are constant during the SCoP execution.

A program in the polyhedral model is typically represented by three components: *iteration domains*, *access relations*, and a *schedule*. We use a running example of matrix multiplication (MM) to illustrate these concepts. Listing 1 shows the example code[2].

```
1 for (int i = 0; i < I; i++)
2   for (int j = 0; j < J; j++)
3     for (int k = 0; k < K; k++)
4       C[i][j] += A[i][k] * B[k][j]; // S0
```

Listing 1: Example code of matrix multiplication.

The *iteration domain* contains all the loop instances of each statement in the program. The iteration domain of the statement $S0$ in the example program has the form $\{S0[i,j,k] : 0 \leq i < I \wedge 0 \leq j < J \wedge 0 \leq k < K\}$. Throughout this dissertation, when representing

[2]The matrix initialization is omitted for brevity.

the components of the polyhedral model, we use the same format as *integer set library* (`isl`) [Ver10], which is a library for polyhedral compilation.

The *access relation* maps a statement instance to an array index. For example, the access relations for the read accesses in the statement $S0$ have the form $\{S0[i,j,k] \rightarrow A[i,k]; S0[i,j,k] \rightarrow B[k,j]; S0[i,j,k] \rightarrow C[i,j]\}$.

Finally, a *schedule* maps instance sets to multi-dimensional time. The statement instances are executed following the lexicographic order of the multi-dimensional time. As an example, the schedule of the statement $S0$ has the form $\{S0[i,j,k] \rightarrow [i,j,k]\}$. `isl` represents the schedule of a SCoP program using *schedule trees* [VGG14]. We give a brief introduction of schedule trees in the next subsection. For more details, please refer to the original paper [VGG14].

2.3.1.2 Schedule Tree

A schedule tree is a tree-formed representation of program schedules in the polyhedral framework. Figure 2.1 shows the schedule tree of the example program in Listing 1. The schedule tree starts with a domain node that defines the iteration domain of the program, followed with band nodes that encode the partial schedules at each loop dimension. The `isl` library manipulates the schedule tree of the program to perform loop transformation. When generating the final code, an AST is obtained from the schedule tree and then lowered to the target code (e.g., C).

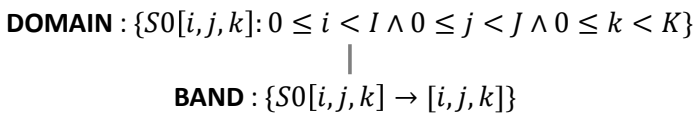


Figure 2.1: Schedule tree representation for the initial schedule of the example MM program.

A generic schedule tree is composed of the following nodes: domain nodes, context nodes, sequence nodes, set nodes, filter nodes, band nodes, extension nodes, and mark nodes.

Domain nodes define the iteration domain of the loop program.

Context nodes provide information about the program parameters. Context nodes

introduce new symbolic constants into the program. For example, in the process of systolic array generation, context nodes can be used to introduce identifiers of the hardware modules.

The program schedule is constructed by *sequence* and *set* nodes. Children of sequence nodes should be executed in order. A set node is similar to a sequence node except that its children may be executed in any order. Children of sequence and set nodes are always *filter* nodes, which restrict the schedules of their descendants to a filtered iteration domain defined by the filter nodes.

Band nodes encode the partial loop schedules in the transformed program. Loops inside each band node are fully permutable.

Extension nodes are used to introduce auxiliary statements into the program. As will be discussed later, we could use this node to introduce additional FIFO statements into the generated hardware program.

Finally, *mark* nodes annotate the nodes in the schedule tree with additional information. They are preserved and printed as comments in the generated code.

Listing 2 shows an example HLS C code of PEs inside a systolic array for matrix multiplication. The corresponding schedule tree for this program is presented in Figure 2.2. In addition to the original computation statement $S0$, we insert four user statements $U0$, $U1$, $U2$, and $U3$ for loading in and writing out the operand data from matrix A and B through FIFOs. In the schedule tree, we insert an extension node to introduce these additional statements and order them by a sequence node.

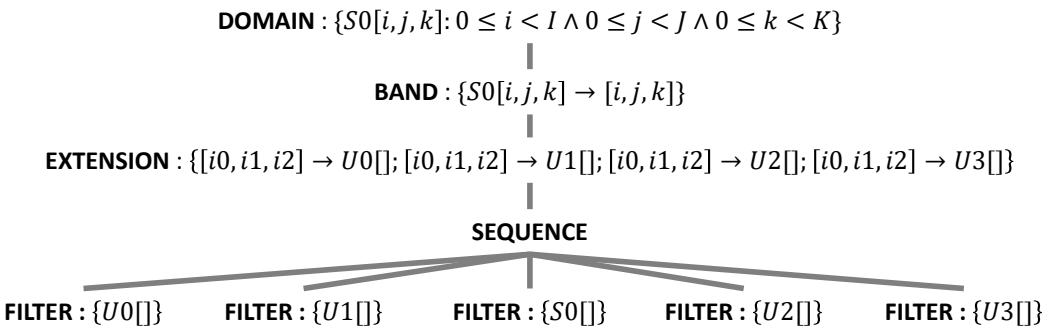


Figure 2.2: Schedule tree representation for the example code in Listing 2.

```
1  data_t regA;
2  data_t regB;
3  for (int i = 0; i < I; i++)
4    for (int j = 0; j < J; j++)
5      for (int k = 0; k < K; k++) {
6          regA = fifo_A.read(); // U0
7          regB = fifo_B.read(); // U1
8          C[i][j] += regA * regB; // S0
9          fifo_A.write(regA); // U2
10         fifo_B.write(regB); // U3
11     }
```

Listing 2: Example code of PEs inside a systolic array for matrix multiplication.

2.3.2 Space-Time Transformation

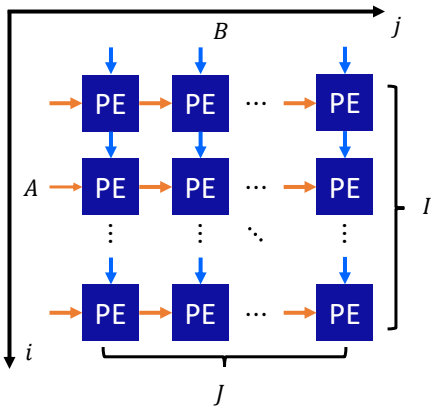
Space-time transformation [Kun88, LQR99, Mol83, RK88] is the foundation of automatic systolic array synthesis. It applies loop transformations on the target program and assigns new semantics (*space* and *time*) to the generated loops. Space loops map loop instances to different PEs that are executed concurrently, and time loops describe the computation inside each PE.

To generate a legal systolic array, the following constraints should be satisfied by the loop transformation: First, the transformation should be semantics-preserving which means it should not break any data dependence in the original program. Second, all data dependences should be uniform (with constant dependence distance). Third, the dependence distances on space loops should be no greater than one so that the data communication only happens between neighbor PEs. Note that for the first and second constraints, we consider all types of dependences (flow, anti, output and input/read dependences). We take into account the read dependences since the data transfer needs to be managed explicitly in systolic arrays including the read-only data. As for the third constraint, we only examine the flow and read dependences which are associated with the inter-PE communication. Since each PE has its own address space, anti and output dependences do not contribute to the data communication between PEs [Bon13].

For the MM example in Listing 1, we obtain one flow dependence (domain constraints

```
// space loops
for (int i = 0; i < I; i++)
  for (int j = 0; j < J; j++)
    // time loop
    for (int k = 0; k < K; k++)
S0:   C[i][j] += A[i][k] * B[k][j];
```

(a) Optimized code after space-time transformation.



(b) 2D systolic array for MM.

Figure 2.3: Example of space-time transformation.

omitted for brevity) as $D1 := \{S0[i,j,k] \rightarrow S0[i,j,k+1]\}$, and two read dependences for array references `A[i][k]` and `B[k][j]` as $D2 := \{S0[i,j,k] \rightarrow S0[i,j+1,k]\}$ and $D3 := \{S0[i,j,k] \rightarrow S0[i+1,j,k]\}$, respectively. One possible space-time transformation is $S := \{S0[i,j,k] \rightarrow [i,j,k]\}$, which is an identity mapping that keeps the original loop structure. We could calculate the dependence distances for the above-mentioned three dependences $D1$, $D2$, and $D3$ under the schedule S, which are $(0,0,1)$, $(0,1,0)$, and $(1,0,0)$. All dependences are uniform (we omit the discussion about output and anti dependences for brevity). Besides, dependence distances on all three loops are no greater than one. Therefore, all three loops are eligible to be selected as the space loops. As an example, we select the first two loops i and j as space loops and leave the loop k as the time loop. The transformed code after space-time transformation is shown in Figure 2.3a. This transformation leads to a 2D systolic array with the space dimensions of $I \times J$. Each PE in this array computes one array element `C[i][j]`. In addition, according to the two read dependences $D2$ and $D3$, data of matrices A and B are reused along the loops j and i, respectively. Therefore, horizontal and vertical interconnects between PEs are inserted to transfer the reused data. The complete array is depicted in Figure 2.3b.

2.4 First Attempt — PolySA

PolySA [CW18] is our first attempt in developing an automatic systolic synthesis tool on FPGA. Figure 2.4 presents the compilation flow of PolySA.

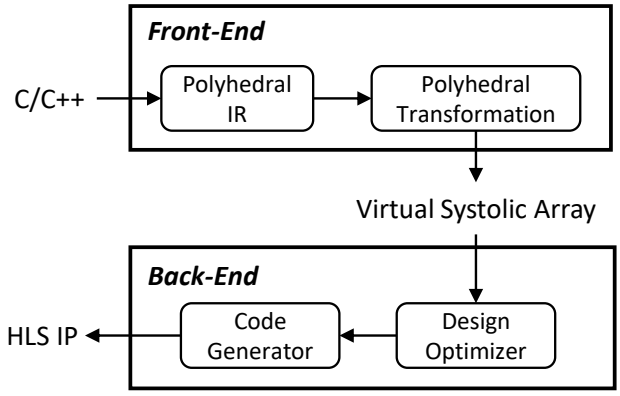


Figure 2.4: Compilation flow of the PolySA framework.

PolySA takes in algorithms in C/C++ and lowers them into polyhedral IR. Then, it applies a sequence of program transformations including the space-time mapping to generate a systolic array with its topology and execution model described in polyhedral IR, named *virtual systolic array* (VSA). Given a VSA, the design optimizer searches for the best design which minimizes certain performance metric (e.g., latency). A final design is generated in the HLS language.

We evaluated PolySA with two applications, matrix multiplication and CNN[3] and achieved performance close to manual designs. On Xilinx VU9P FPGA, the generated systolic arrays from PolySA achieved 69% and 91% of the peak performance of the state-of-the-art manual designs on MM and CNN. The larger performance gap on MM is due to the lower frequency of HLS designs. In comparison, manual designs implemented well-optimized RTL code that achieved a much higher frequency than HLS designs.

PolySA demonstrated the feasibility of using polyhedral model and HLS to generate high-performance systolic arrays on FPGA. However, when using PolySA for more applications, multiple issues were brought to our attention which are summarized as below:

[3]PolySA generates a single systolic array for each CNN layer.

Limited scope of supported applications PolySA only supports programs with a single statement in perfectly nested loops. This limitation makes it hard to apply this tool on complex applications such as LU decomposition which is involved with a more complicated loop structure.

Inflexible code generation methods PolySA implements a fixed code template considering different array topology and PE structures. This template is instantiated with different parameters to generate the final designs. This code generation method is extremely limited when targeting different back-ends like Intel OpenCL. Besides, the code template only supports programs with computation statements of simple multiply-and-add (MAC) operations. Therefore, when targeting new hardware back-ends or complicated programs with statements beyond the MAC operations, we will have to modify the code template manually, introducing non-trivial development efforts.

Inefficient compilation infrastructure and auto-tuning approaches PolySA uses Clan [BCG03] for extracting the polyhedral IR, which only supports a subset of C99 standard C programs. For example, conditional branches like `a?b:c` are not supported by Clan. In addition, PolySA is implemented using Matlab which leads to low compilation speed and restrains the possibility for open-source. Lastly, PolySA uses exhaustive search with pruning heuristics to search the design space which faces scalability issues when handling large-scale designs.

These limitations have motivated us to develop a second compilation framework, AutoSA. AutoSA overcomes all the issues listed above, providing a general, performant, and productive solution to the problem of automatic systolic array synthesis. Details of this framework will be covered in the next section.

2.5 AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA

2.5.1 Framework Overview

In this section, we clarify the application scope that AutoSA covers. Then, we give a bird's-eye view of the overall compilation flow of AutoSA.

2.5.1.1 The Scope of AutoSA

AutoSA is built on the polyhedral framework, which takes SCoP programs as the input. AutoSA assumes all the dependences of the input programs have been rendered uniform before the compilation. Prior studies on uniformization [MMR01] which converts affine dependences to uniform dependences can be integrated as a pre-processing step to relax this constraint. This work is orthogonal to our framework and will be left as future work.

2.5.1.2 Compilation Flow

Figure 2.5 presents the overall compilation flow of AutoSA.

Model extraction This step extracts the polyhedral model consisted of the iteration domains, access relations, schedule, and data dependences from the input C program. In comparison to PolySA, AutoSA uses the state-of-the-art polyhedral extraction tool (PET) [VG12] for extracting the polyhedral IR. PET is a fully-fledged polyhedral front-end which is built on LLVM Clang and provides full support for C99 standard. It is seamlessly integrated with `isl` which parses the polyhedral IR extracted by PET and applies program transformations to optimize the performance.

Legality check This step checks if the input program can be legally mapped to a systolic array. Specifically, we check the three constraints as mentioned in Section 2.3.2. If the input program fails to meet any of those constraints, AutoSA will skip the following steps and

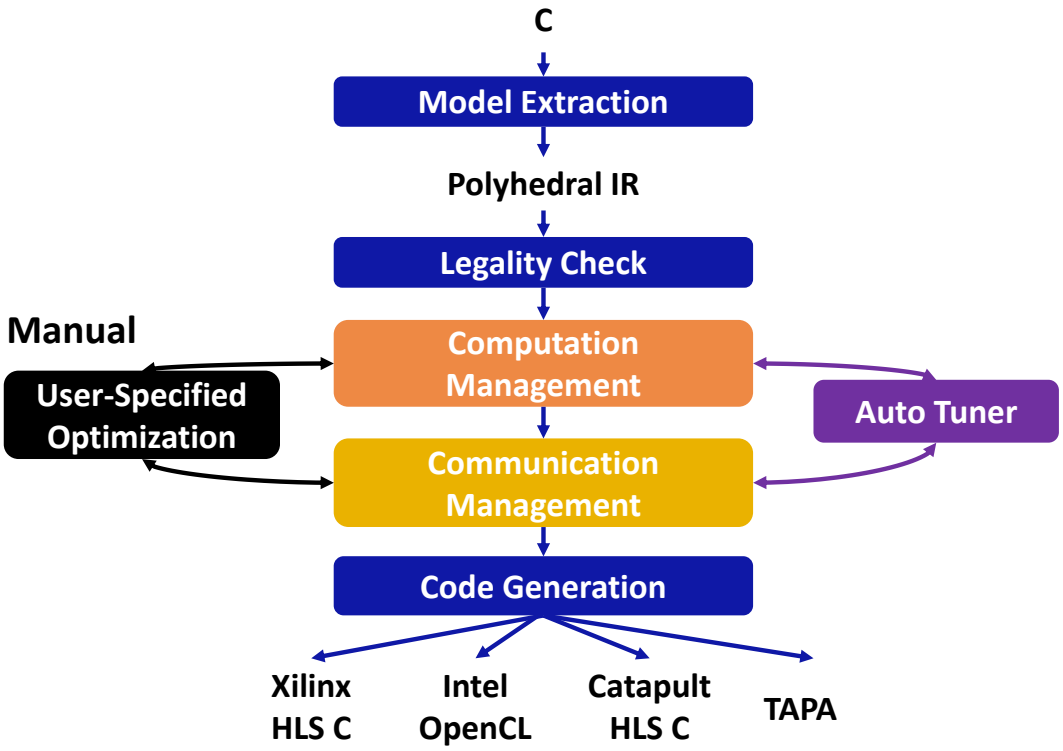


Figure 2.5: Overview of the AutoSA compilation flow.

dump out a CPU code from the current program schedule.

Computation and communication management A complete systolic array architecture consists of both the PE array and the on-chip I/O network. AutoSA separates the process of building these two components into two stages: computation and communication management. The stage of computation management constructs the PE and optimizes its micro-architecture. After that, the stage of communication management builds the I/O network for transferring data between PEs and the external memory. Details of these two stages will be covered in Section 2.5.2 and 2.5.3, respectively.

Code generation After the previous stages, AutoSA generates the AST from the optimized program. The AST is then traversed to generate the final design code for the target hardware. The AST-based code generation approach is highly flexible and requires minimal efforts to extend to new hardware back-ends. As an example, we were able to connect AutoSA to

TAPA [CGL21], an HLS compilation framework for dataflow programs, in less than two weeks with the help of a PhD student at UCLA.

Auto-tuner The stages of computation and communication management involve multiple optimization techniques, each introducing several tuning options. AutoSA implements tunable knobs for these techniques which can be set by users manually or tuned by an auto-tuner. The proposed auto-tuner implements a hybrid method including both a mathematical programming-based optimizer and evolutionary search. It is able to find high-quality designs in seconds on a single-thread CPU. Details of the auto-tuner will be covered separately in the next chapter.

2.5.2 Computation Management

The stage of computation management consists of four steps: space-time transformation, array partitioning, latency hiding, and SIMD vectorization. Figure 2.6 shows the overview of each step in the computation management. We will go though them in detail in the following sections.

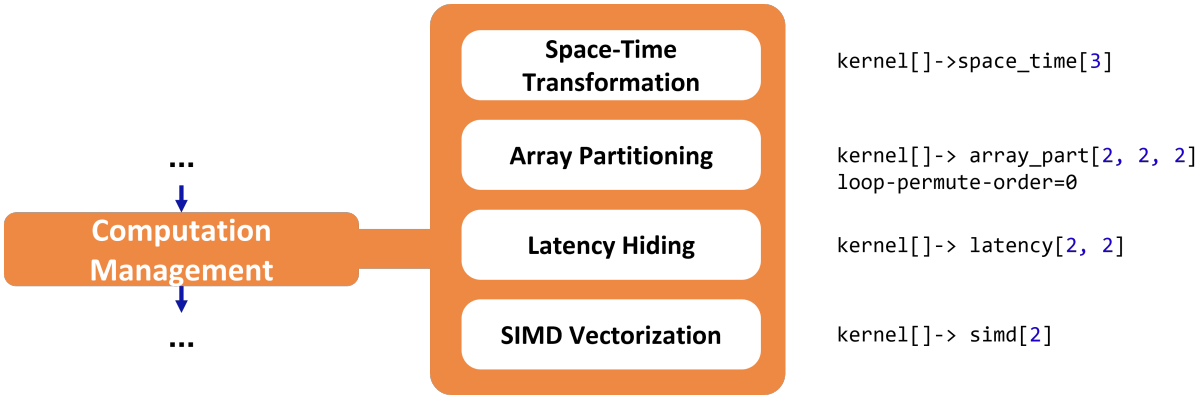


Figure 2.6: Overview of computation management.

2.5.2.1 Space-Time Transformation

This step performs the space-time transformation to map the input program to a systolic array. Algorithm 1 describes how AutoSA applies the space-time transformation. AutoSA searches

for the loops in the outermost loop band in the schedule tree with flow/read dependence distances no greater then one. Those loops are put into a pool as the candidate space loops. Next, AutoSA enumerates all space loop combinations from the candidate pool. The selected space loops are permuted outermost. All the loops below the space loops are assigned as time loops. At present, AutoSA generates 1D and 2D systolic arrays. This constraint can be relaxed to generate higher-dimensional arrays if necessary. There will be multiple systolic arrays generated from this step, each with a unique schedule. Users can choose which array to process manually, or leave it to be explored by the auto-tuner.

Algorithm 1: Space-time transformation.

Inputs : A schedule tree s of the input program
Outputs: A list of schedule trees S of the transformed programs describing systolic arrays
Initialize the space loop candidate pool $P \leftarrow \emptyset$;
Extract the outermost permutable loop band d from s;
for *each loop l in the loop band d* **do**
    if *all flow/read dependence distances on loop $l \leq 1$* **then**
        $P \leftarrow P \cup l$;
/* Generate 2D systolic array. */
for *each pair of loops (l_1, l_2) in the pool P* **do**
    Duplicate the schedule tree $s' \leftarrow s$;
    Modify s' by permuting the loops l_1, l_2 to outermost;
    $S \leftarrow S \cup s'$;
/* Generate 1D systolic array (omitted), similar to the 2D case with only one space loop selected. */

Figure 2.7 presents two examples of space-time transformation for matrix multiplication. In the left example, we assign the loop j as the space loop and the loops i and k as time loops, resulting in a 1D systolic array with the loop j mapped to the PE dimension. In the right example, loops i and j are assigned as space loops, leading to a 2D systolic array. AutoSA enumerates all the feasible space loop candidates to generate different space-time mappings. For the MM example, AutoSA generates 6 systolic arrays with different space-time mappings, including 3 1D systolic arrays with loops $[i]$, $[j]$, $[k]$ as the space loops, respectively, and 3 2D systolic arrays with the space loops pairs as $[i, j]$, $[i, k]$, and $[j, k]$. In the rest of this dissertation, we will use $[i, j]$ to denote a systolic array with loops i and j as the space loops. Users could supply the compiler with the argument `kernel[]->space_time[3]` to select the

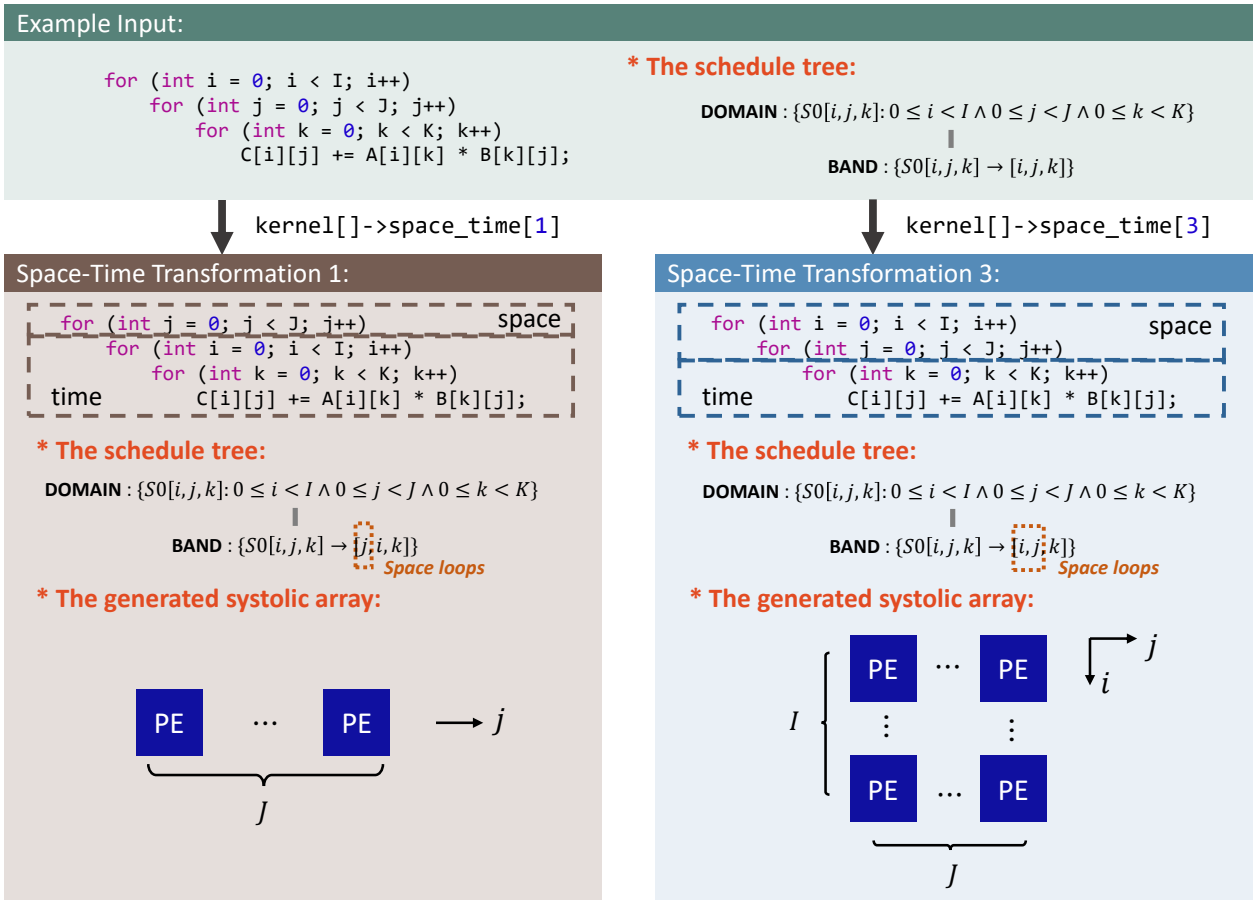


Figure 2.7: Example of space-time transformation.

4-th array (zero-indexed) as shown in Figure 2.7.

Note that previous works such as Eyeriss [CES16], Interstellar [YGL20], and Maestro [KCS20] used the term *dataflow* to identify different array topologies and execution patterns. These works covered a broader scope of architectures beyond systolic arrays. For example, Maestro considers architectures with multicast I/O network, which is not supported by systolic arrays that only implement the unicast local interconnects. Within the scope of systolic arrays, different dataflows are equivalent to different space-time mappings that determine the unique array topology and execution pattern. In the rest of the dissertation, we will use *dataflow* to refer to different space-time mappings to follow the naming convention of the previous works.

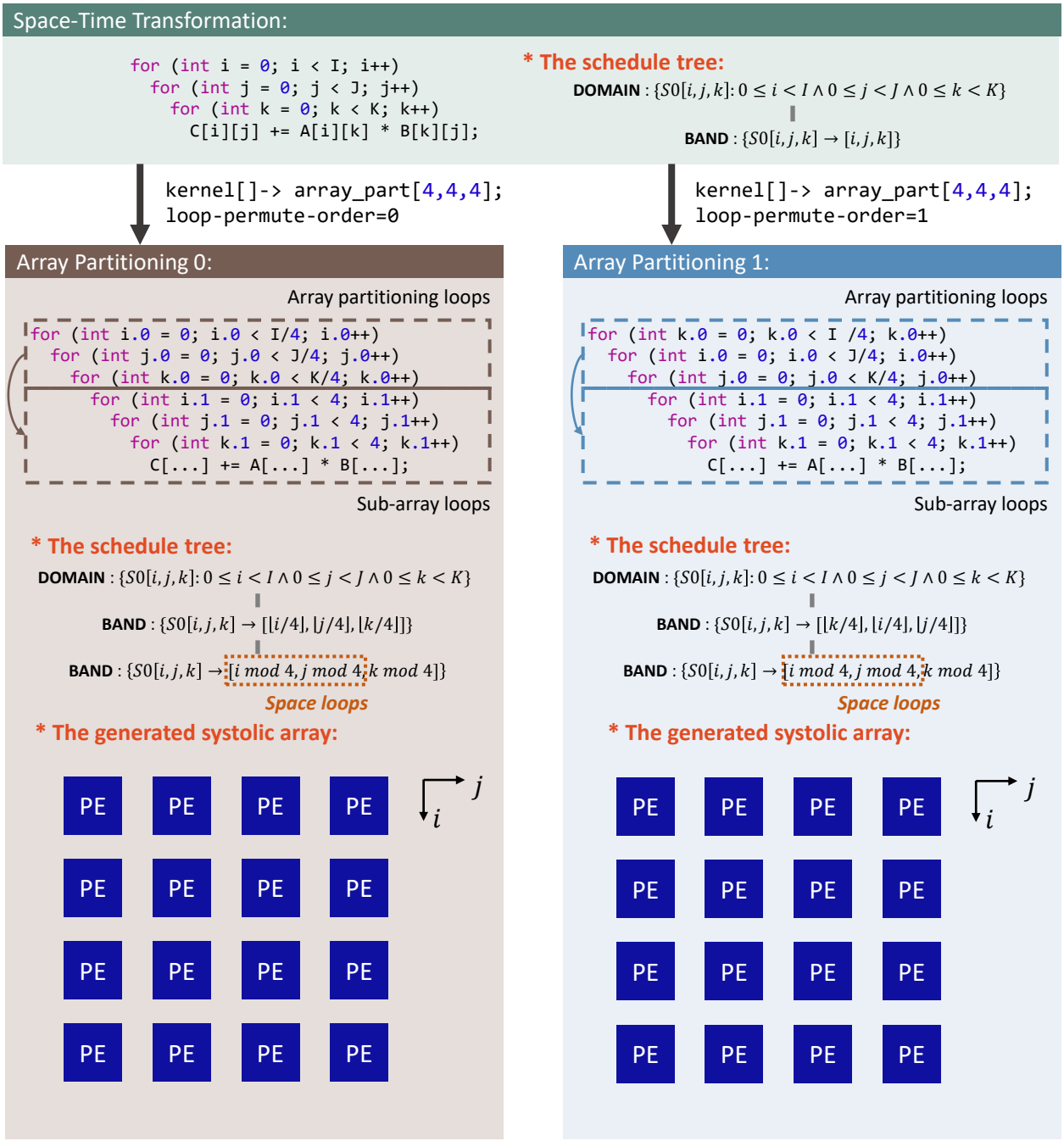


Figure 2.8: Example of array partitioning.

2.5.2.2 Array Partitioning

Given the limited on-chip resource, array partitioning is mandatory when mapping a large array to FPGA. To achieve this, AutoSA tiles the outermost permutable loop band in the schedule tree which contains the space loops. The tiling factors can be chosen by users or set by the

auto-tuner during the design space exploration. We show two examples of array partitioning in Figure 2.8. In both examples, we tile the loop band (i, j, k) with the tiling factors $(4, 4, 4)$, leading to a 4×4 systolic array on-chip. Different sub-array tiles are sequentially scheduled on this array. In AutoSA, programmers can use `kernel[]->array_part[4,4,4]` to specify the tiling factors. Another freedom when applying array partitioning is that loops in the tiled loop band can be permuted in any order. Similar as the space-time transformation, AutoSA enumerates all the possible loop permutations. Users may use the argument `--loop-permute-order=0` as shown in Figure 2.8 to select the loop permutation ordering to proceed.

At present, AutoSA uses the rectangular tiling. This will introduce redundant computation for applications with non-rectangular iteration domains like LU decomposition. We will explore tiling with other shapes such as overlapped tiling [HPS12] and diamond tiling [BPB12] in the future.

2.5.2.3 Latency Hiding

Latency hiding helps hide the pipeline stalls caused by the loop-carried dependence of the compute statements. In the MM example, the multiply-and-add (MAC) operation in the statement `C[i][j]+=A[i][k]*B[k][j]` introduces loop-carried dependence on the loop k, resulting in an initial interval (II) greater than one. To resolve this issue, AutoSA looks for parallel loops in the schedule tree, strip-mines them and permutes the point loops to the innermost. Figure 2.9 shows an example where parallel loops $i.1$ and $j.1$ are tiled by factors $(2, 2)$. The point loops $i.2$ and $j.2$ are permuted to the innermost. As there is no loop-carried dependence on the innermost loops, the pipeline could achieve 100% utilization. Similarly, programmers may supply the argument `kernel[]->latency[2,2]` to specify the tiling factors used for each loop.

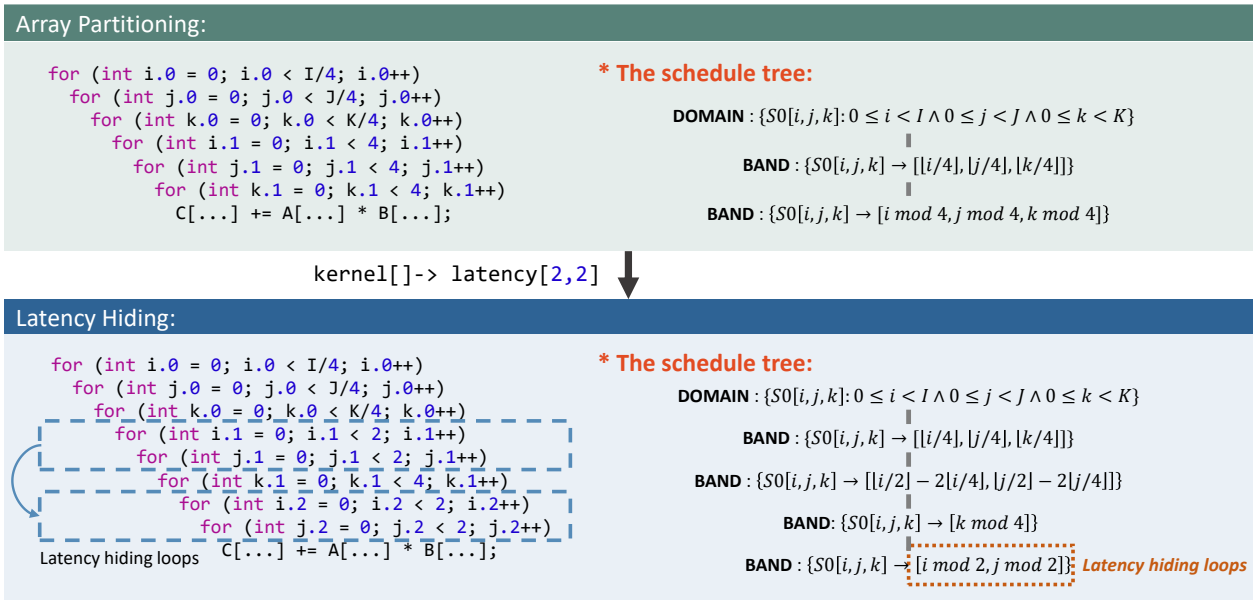


Figure 2.9: Example of latency hiding.

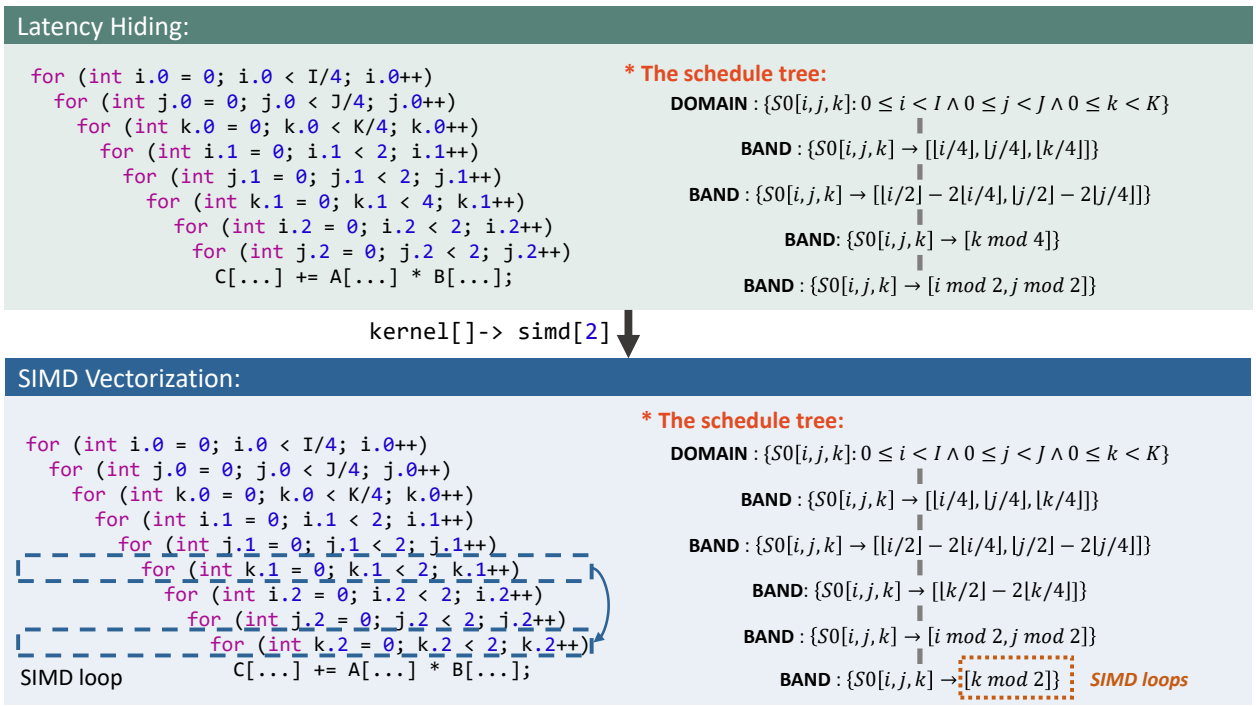


Figure 2.10: Example of SIMD vectorization.

2.5.2.4 SIMD Vectorization

SIMD vectorization duplicates the compute units inside each PE. This helps amortize the control overheads and improve the resource efficiency of the design. At present, AutoSA

detects the vectorizable loop by examining the following two criteria: 1) The loop should be a parallel loop or a reduction loop[4]. 2) All array references within the loop are stride-one or stride-zero along this loop. In the MM example, loop k is a reduction loop. Array references `C[i][j]` and `A[i][k]` are stride-zero and stride-one along loop k. The array reference `B[k][j]` requires a layout transformation to `B[j][k]` so that it becomes a stride-one access that enables the vectorization. Figure 2.10 shows the vectorized code in which we strip-mine the loop k with a factor of 2. The point loop is permuted to the innermost and marked *unroll* which will be handled by HLS tools at last. During the compilation, AutoSA examines each loop and enumerates all the possible layout transformations to expose the SIMD opportunities. Users may choose one loop to proceed or let the auto-tuner take over and make the choice.

2.5.3 Communication Management

So far we have finished the PE construction and optimization. However, the current array is still not functional as we are missing the other key component, the I/O network. The I/O network is a network on-chip that supports two types of data communication:

Inner-array communication This refers to the data communication between PEs. As an example, in Figure 2.3b, we show that for the MM example, we could transfer data elements from matrix A and B via horizontal and vertical interconnects.

Outer-array communication This refers to the data communication between PEs and the external memory (e.g., DRAM). In the MM example, the I/O network needs to fetch the data elements of matrices A and B to feed the array, drain the final results of matrix C from the array and write out to the external memory.

The stage of communication management in AutoSA analyzes the program and constructs the I/O network as mentioned above. We show that an I/O network can be built automatically via data dependence analysis in the polyhedral model. Furthermore, as the topology of

[4]The current polyhedral framework that AutoSA builds on lacks the capability to detect the reduction loop. We require users to annotate the reduction loops prior to the compilation.

the I/O network plays an important role in determining the frequency of the design, we extend the I/O network generation algorithm to build an I/O network that only involves local interconnects, hence, guaranteeing the sustained high frequency.

The following subsections explain our approaches in detail. Section 2.5.3.1 describes how we analyze the dependences in the program to extract the necessary information for constructing the I/O network. Section 2.5.3.2 builds the I/O network using the information extracted from the previous step. Section 2.5.3.3 discusses several I/O optimization techniques to further improve the I/O performance.

2.5.3.1 I/O Analysis

Data communication is associated with the data dependences. Previous works such as [Bon13, DRR13] have demonstrated how to implement the data transfer scheme for MPI programs using the polyhedral model. Our algorithms for building the I/O network of systolic arrays are inspired by that thread of work but with further extension to take into account the uniqueness of systolic arrays. To build the I/O network, AutoSA analyzes the following three types of data dependences:

- Read dependence: Used for transferring read-only data.
- Flow dependence: Used for transferring intermediate results.
- Output dependence: Used for transferring final results.

Table 2.2: Dependence relations of the MM example.

Type	Dependence Relation	Array Access
Read	$D1 := \{S0[i,j,k] \to S0[i,j+1,k]\}$	`A[i][k]`
Read	$D2 := \{S0[i,j,k] \to S0[i+1,j,k]\}$	`B[k][j]`
Flow	$D3 := \{S0[i,j,k] \to S0[i,j,k+1]\}$	`C[i][j]`
Output	$D4 := \{S0[i,j,k] \to S0[i,j,k+1]\}$	`C[i][j]`

Table 2.2 lists the dependences extracted from the MM example. I/O analysis interprets these dependences and extracts a data structure named *I/O group* that contains the necessary

information required to construct the I/O network. Algorithm 2 describes the details of this step.

Algorithm 2: I/O group construction.

Inputs : Access relations A, dependence relations D_{read}, D_{flow}, D_{output}, program schedule s

Outputs: A set of I/O groups G

Initialize the I/O group set $G \leftarrow \emptyset$;

/* Populate the I/O groups. */

for *each array access acc in* A **do**

  for *each dependence* d *in* D_{read}, D_{flow}, D_{output} **do**

    if *acc is associated with* d **then**

      Construct a new I/O group $g(acc, d)$;

      Compute the properties of the group g: I/O direction $g.dir$ and I/O type $g.type$;

      $G \leftarrow G \cup g$;

/* Merge the I/O groups. */

for *each pair of I/O groups* (g_1, g_2) *in* G **do**

  if *is_shared*(g_1, g_2) **then**

    Merge the two I/O groups $g \leftarrow merge(g_1, g_2)$;

    Update the I/O group set $G \leftarrow G \setminus (g_1 \cup g_2) \cup g$;

/* Compute the I/O group copy-in/copy-out sets. */

for *each I/O group* g *in* G **do**

  if g *is associated with read dependences* **then**

    $g.W_{in} \leftarrow \cup_{d_i \in g}(src(d_i) \cup dst(d_i))$;

  if g *is associated with flow dependences* **then**

    $g.W_{in} \leftarrow \cup_{d_i \in g} dst(d_i)$;

    $g.W_{out} \leftarrow \cup_{d_i \in g} src(d_i)$;

  if g *is associated with output dependences* **then**

    $g.W_{out} \leftarrow \cup_{d_i \in g}(dst(d_i) - src(d_i))$;

An I/O group g is defined as a tuple of $g = (A, D)$ where A is a set of array accesses that are associated with the current group g and D is the set of data dependences associated with the array accesses in A. In Algorithm 2, we first populate the initial I/O group set G. A single I/O group is constructed for each array access and its associated data dependence. For each I/O group, the following two properties are computed:

- I/O direction: This is the the dependence distance vector on the space loops.
- I/O type: An I/O group is classified as *exterior I/O* if the dependence is carried by

the space loops. Otherwise, it is classified as *interior I/O*. When data dependences are carried by the space loops, data that are associated with these dependences will be reused across PEs. This indicates that data can be transferred to the array from *exterior* boundaries. On the other hand, if dependences are not carried by the space loops, data will be used and updated inside each PE. In this case, we need to send/collect the data directly to/from each PE, i.e., the *interior* of the array.

In the MM example, for the array access `B[k][j]`, we construct an I/O group g from the array access `B[k][j]` and its associated dependence $D2$ as shown in Table 2.2. The dependence distance of $D2$ on the space loops is $(1, 0)$. Therefore, we assign the I/O direction as $g.dir = (1, 0)$ and the I/O type as $g.type = exterior$.

The last step is to compute the set of statement instances that require these data. We divide the statement instances into two sets: copy-in set W_{in} and copy-out set W_{out}. These sets contain the statement instances that require the data to be copied in or copied out across each array partition, respectively. For I/O groups with read dependences, we compute the copy-in set as the union of all source and destination domains of the dependences as all the data are required. The copy-out set is left empty. For I/O groups with flow dependences, the copy-in set consists of the destination domains of the dependences and the copy-out domain consists of the source domains. Lastly, for I/O groups with output dependences, we compute the copy-out set by subtracting the source domains from the destination domains as we are only interested in the last updated elements. The copy-in set is left empty. Table 2.3 includes the final I/O groups extracted from the MM example and their copy-in/copy-out sets. These sets will be used for I/O network construction in the next section.

2.5.3.2 I/O Construction

This step constructs the I/O modules based on the I/O grouping information extracted from the previous step. For each I/O group, AutoSA allocates a set of I/O modules for transferring the data between PEs and the external memory. Algorithm 3 describes the detailed procedure of this step.

Table 2.3: I/O groups of the MM example.

No.	A	D	W_{in}/W_{out}
g_1	`A[i][k]`	$D1$	$W_{in} := S0[i,j,k] : 0 \le i < I \wedge 0 \le j < J \wedge 0 \le k < K\}$
g_2	`B[k][j]`	$D2$	$W_{in} := S0[i,j,k] : 0 \le i < I \wedge 0 \le j < J \wedge 0 \le k < K\}$
g_3	`C[i][j]`	$D3$	$W_{in} := S0[i,j,k] : 0 \le i < I \wedge 0 \le j < J \wedge 0 < k < K\}$ $W_{out} := S0[i,j,k] : 0 \le i < I \wedge 0 \le j < J \wedge 0 \le k < K-1\}$
g_4	`C[i][j]`	$D4$	$W_{out} := S0[i,j,k=K-1] : 0 \le i < I \wedge 0 \le j < J$

Algorithm 3: I/O construction.

Inputs : Schedule s, I/O groups G, number of space loops dim
Outputs: A list of schedules for I/O modules L

$L \leftarrow \emptyset$;
/* Copy-in modules */
for *each I/O group* g *in* G **do**
  Duplicate the schedule $s' \leftarrow s$;
  Insert the domain filter $g.W_{in}$ into the schedule s';
  $io_level \leftarrow 1$;
  while $io_level \le dim$ **do**
    Perform I/O module clustering on the first $(dim - io_level + 1)$ space loops
    $s' = io_clustering(s', dim - io_level + 1, g)$;
    Add s' to L;
    $io_level \leftarrow io_level + 1$;
/* Copy-out modules (omitted for brevity) */

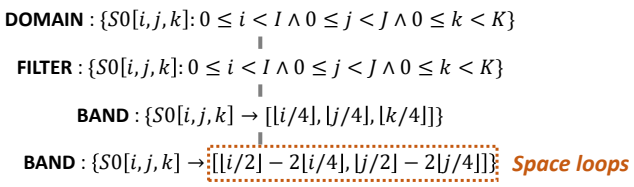


Figure 2.11: Insert a filter to isolate the statement instances of group g_2.

We start with the optimized schedule from the computation management. In the first step, we isolate the statement instances that are involved with the data communication from the current I/O group by inserting a filter node into the schedule tree with the copy-in/copy-out set. The filter node restrains the iteration domains of its children nodes by intersecting the current iteration domain with the filter set [VGG14]. As an example, Figure 2.11 shows the updated schedule with the filtered domain for the I/O group g_2 in Table 2.3 (loops inside the

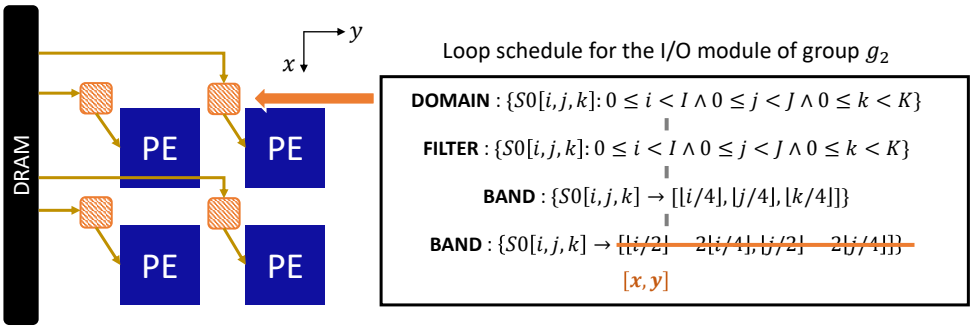
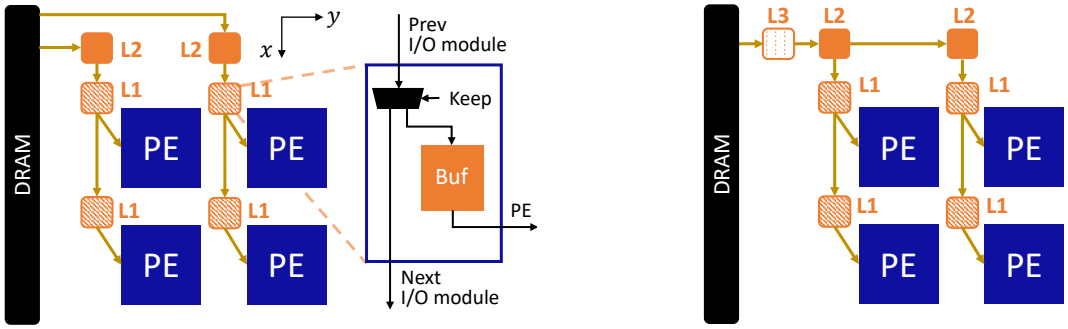


Figure 2.12: PE array with the I/O module for g_2 and its loop schedule.

space loops are omitted for brevity). At this stage, we could already generate a set of I/O modules that load the data from the external memory and send the data directly to each PE. This can be realized by equating the space loops to the PE indices x and y in the updated schedule. Figure 2.12 shows the generated I/O network and the corresponding loop schedule for each I/O module.



(a) Clustering L1 I/O modules.

(b) Clustering L2 I/O modules.

Figure 2.13: Example of I/O clustering for I/O group g_2.

However, the current architecture faces scalability issues as the data are scattered directly from the external memory which causes high fan-outs and could lead to routing failure. To resolve this issue, we choose to *localize* the I/O network by using a daisy-chain architecture that has been seen in many previous works [WYZ17, MKN18, SRB19, CWY18]. In this architecture, each I/O module fetches data from the upper-stream I/O modules. The I/O module works as a filter that keeps the data belonging to the PEs that it is connected to and passes the rest data to the down-stream I/O modules. We name the I/O modules that are directly connected to PEs as level-one (L1) I/O modules. We could first cluster the L1

I/O modules along the x-axis, as shown in Figure 2.13a. Every two L1 modules along the x-axis are connected to an upper-level (L2) I/O modules, which helps reduce the memory fan-outs from four to two. We name such a process as *I/O clustering*. I/O clustering can be applied multiple times in a hierarchical way. For example, we could apply the I/O clustering again on the L2 I/O modules, generating one L3 I/O module that connects to the DRAM, as shown in Figure 2.13b. Eventually, we reduce the memory fan-outs from four to one.

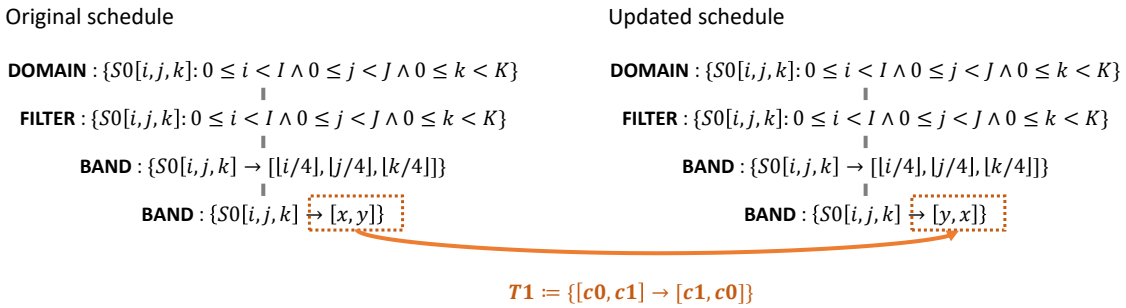
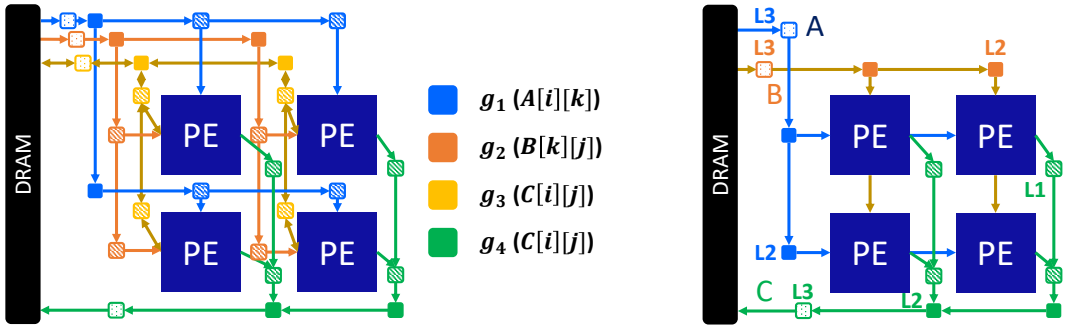


Figure 2.14: Clustering L1 I/O modules by applying the transformation $T1 := \{[c0, c1] \rightarrow [c1, c0]\}$ on the space loops.

I/O clustering is realized by applying loop transformation on the space loops. For example, when clustering the L1 I/O modules for the group g_2, since the group g_2 has the I/O type as exterior I/O and I/O direction as $(1, 0)$, indicating that data are reused along the x-axis, we will cluster the I/O modules along the $(1, 0)$ direction. This is achieved by applying a loop transformation $T1 := \{[c_0, c_1] \rightarrow [c_1, c_0]\}$ on the space loops. Figure 2.14 shows the loop transformation applied on the space loops to cluster the L1 I/O modules for group g_2. Next, when clustering the L2 I/O modules, only the first space dimension is involved. For simplicity, we apply an identity transformation $T2 := \{[c0] \rightarrow [c0]\}$ that clusters the L2 I/O modules along the y-axis. Note that when choosing the I/O clustering directions, for the L1 I/O modules of I/O groups with exterior I/O, AutoSA uses the default I/O direction of the group. As for the rest of the cases, at present, AutoSA simply clusters the I/O modules vertically or horizontally. Users can also provide their own inputs to control the I/O clustering direction manually. Figure 2.15a depicts the final array architecture after the I/O clustering for all the I/O groups in Table 2.3.

2.5.3.3 I/O Optimization

In this step, AutoSA applies multiple passes to further optimize the I/O network.



(a) Initial array. (b) Optimized array.

Figure 2.15: A complete 2D systolic array for the MM example.

I/O module embedding L1 I/O modules with exterior I/O are embedded into the PEs to save the resource.

I/O module pruning When transferring the data between different sub-array tiles, AutoSA checks if the copy-out set of the previous tile equals the copy-in set of the current tile at the PE level. If two sets are equal at the PE level, it indicates the data can be reused cross the array partitions and the data transfer from the external memory is unnecessary. For such a case, I/O modules for this I/O group are pruned away to save the off-chip communication and on-chip resource. For the MM example, I/O modules for the group g_3 will be pruned away since the data of matrix C are accumulated locally inside each PE. Figure 2.15b shows the optimized array by applying two techniques as mentioned above.

Data packing To reduce the data transfer latency between the I/O modules, AutoSA performs data packing between I/O modules. Packing more data helps reduce the data transfer latency, however, it leads to FIFOs with a larger width and higher resource usage. Users could specify the data packing factors between different I/O levels with an argument like `kernel[]->data_pack[8,32,64]`. With this argument, we instruct AutoSA to pack at most 8/32/64 bytes of data between L1/L2/L3 I/O modules.

Double buffering By default, AutoSA allocates a local buffer inside the L1 I/O modules for I/O groups with interior I/O or inside the L2 I/O modules for I/O groups with the exterior I/O. For I/O modules with local buffers, AutoSA offers options to enable the double buffering that overlaps the memory transfer with the PE computation. For example, the array A in the MM example can be double buffered with the argument `double-buffer-assign="{kernel[]->A[]}"`.

Figure 2.16 shows the detailed architecture of the I/O network with all the above-mentioned optimizations.

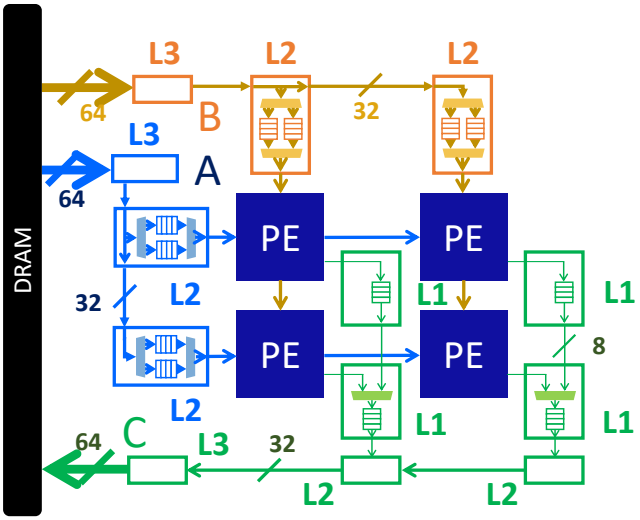


Figure 2.16: Detailed architecture of the I/O network for the MM example.

To summarize, AutoSA generates a complete systolic array with a series of hardware optimization passes. Each pass can be manually altered with additional tuning options. Table 2.4 summarizes all the available tuning options in AutoSA.

In addition, AutoSA supports the use of non-divisor tiling factors in array partitioning. Note that when a non-divisor tiling factor is used, AutoSA requires users to modify the original problem dimension manually to make it a multiple of the tiling factor. Otherwise, additional loop guards will be inserted into the HLS program, leading to unnecessary hardware overheads. Tiling factors of latency hiding and SIMD vectorization are required to be divisors as the sub-array dimension is fixed after the array partitioning and non-divisors tiling factors result in prologue/epilogue phases and expensive max/min operations in the generated program,

Table 2.4: Summary of the tuning options of AutoSA.

Compilation Pass	Tuning Options	Command Example
Space-Time Transformation	Dataflow	`kernel[]->space_time[3]`
Array Partitioning	Loop Permutation Loop Tiling	`loop-permute-order=0` `kernel[]->array_part[2,2,2]`
Latency Hiding	Loop Tiling	`kernel[]->latency[2,2]`
SIMD Vectorization	Loop Tiling	`kernel[]->simd[2]`
Data Packing	Parameters	`kernel[]->data_pack[8,32,64]`
Double Buffering	Arrays	`kernel[]->A[];kernel[]->B[]`

leading to designs with inferior performance.

2.5.4 Code Generation

2.5.4.1 Extensibility

AutoSA is built on the integer set library (`isl`) [Ver10], which is a polyhedral compilation library for manipulating the polyhedral IR for loop transformations. The `isl` library also provides complete support to code generation including generating ASTs from the polyhedral IR and printing out the final programs by traversing ASTs. During the AST generation, user-defined AST nodes can be created and printed with customized callback functions. We leverage this feature to generate platform-dependent program statements on different hardware.

Figure 2.17 shows an example of generating FIFO statements for reading and writing out the data of matrix A in hardware modules. We start by inserting two *extension* nodes into the schedule tree before and after the computation statement. An extension node is a special type of nodes in the schedule tree that can be used to insert additional statements into the program. The modified schedule tree is then lowered to an AST with the extension node translated to a user-defined node that can be printed by a callback function for printing Xilinx HLS APIs for FIFO transactions.

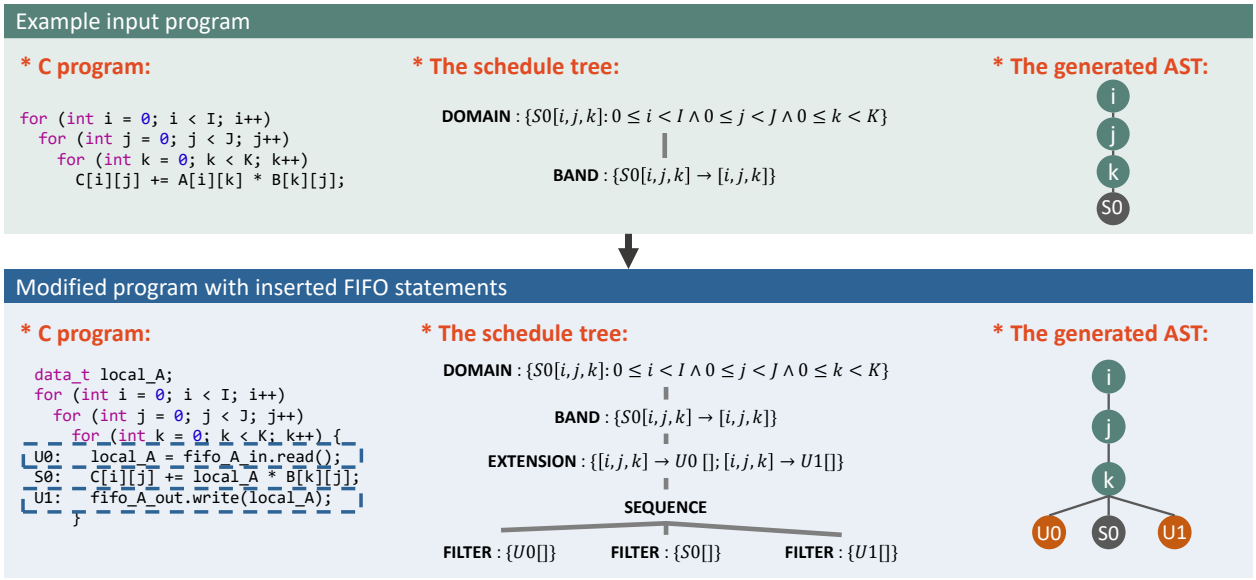


Figure 2.17: A code generation example of inserting FIFO statements into the programs.

With this infrastructure, we can easily extend AutoSA to different hardware back-ends. At present, in addition to Xilinx HLS C, AutoSA also supports three other back-ends, including Intel OpenCL [Int21], Catapult HLS C [Sie21], and TAPA [CGL21].

2.5.4.2 Hardware Efficiency

The stage of code generation also applies several passes on the transformed program to further improve the hardware efficiency. We list these passes as below.

Bitwidth reduction Many HLS compilers provide arbitrary precision datatypes to reduce the hardware overheads. For example, in Xilinx HLS C, programmers could specify a variable with the data type `ap_uint<7>` to limit the datawidth of this variable to 7 bits. Using a reduced datawidth helps save the hardware resource. AutoSA automatically applies the bitwidth reduction on all loop iterators with constant loop bounds.

Array contraction Figure 2.18 shows an example of array contraction. In the example program, we allocate a local buffer `C_buf[I][J]` on-chip. Different elements of `C_buf[i][j]` are only updated once and then drained out through FIFOs. Array contraction can be applied

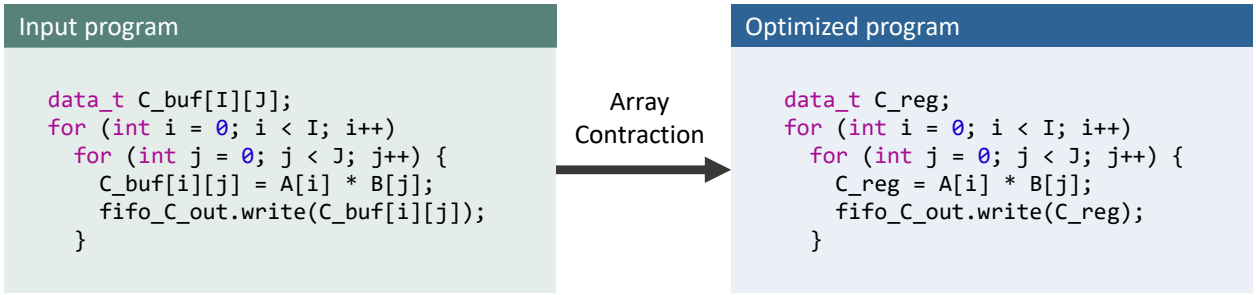


Figure 2.18: Example of array contraction.

on the array `C_buf` by storing it in a single register `C_reg` to save the memory. AutoSA performs the liveness analysis on the local arrays and applies array contraction automatically to reduce the memory usage.

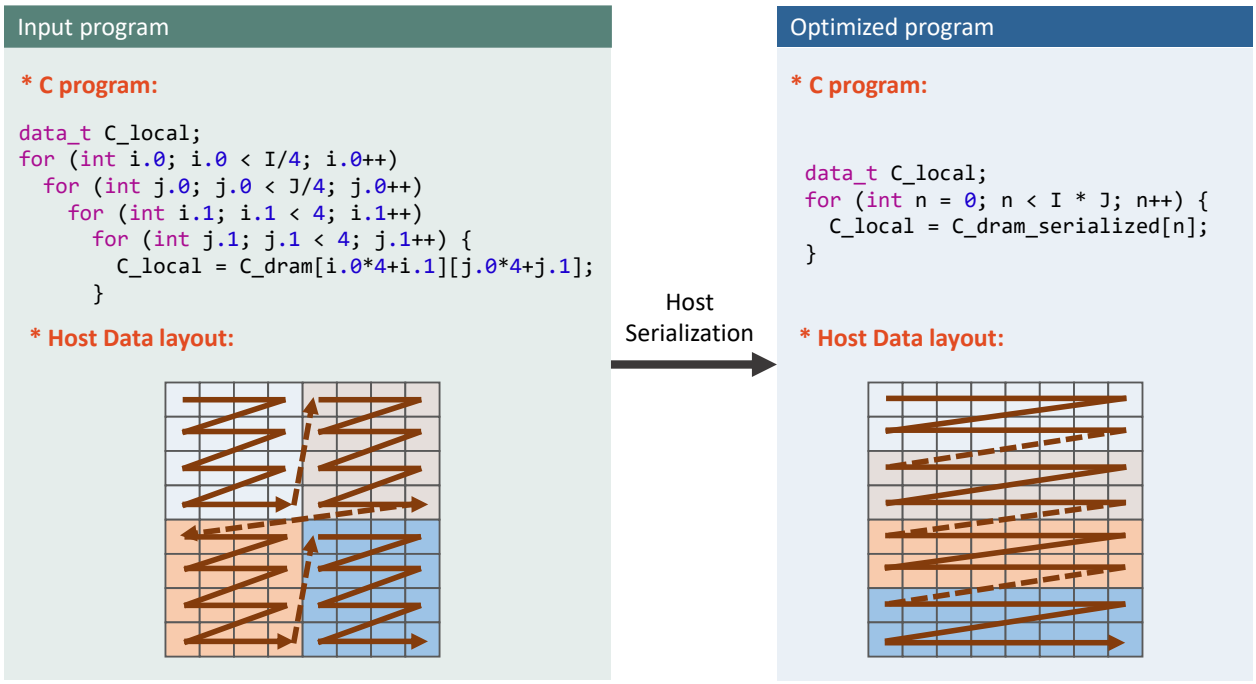


Figure 2.19: Example of host serialization.

Host data serialization For the input data from the external memory, we could reorganize their layout on the host side to increase the memory burst length. Figure 2.19 shows an example of this optimization. Without serialization, the maximal burst length when reading the array `C_dram` is 4 words. After serializing the data, the maximal burst length is increased to the total amount of elements in the array, which is $I \times J$ words in

this example. An increased burst length helps improve the effective DRAM bandwidth. For communication-bound applications, host data serialization reduces the off-chip data access latency and improves the overall design performance.

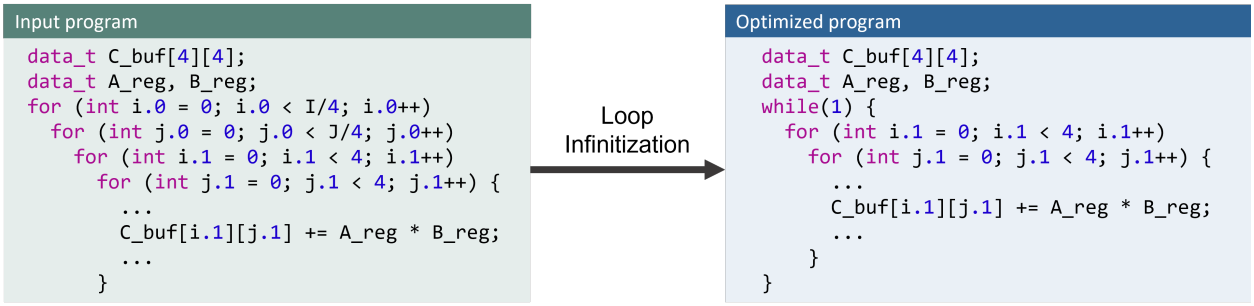


Figure 2.20: Example of loop infinitization.

Loop infinitization For outermost loops that are not used by the loop statements, we could eliminate them and replace them with a single `while (1)` loop that helps reduce the control logic overheads of `for` loops. Figure 2.20 shows one example of loop infinitization. AutoSA will check if any outermost loop is not used by the inner computation statements and apply the loop infinitization to reduce the hardware overheads.

2.5.5 Auto-Tuning

The optimizations in the stage of computation and communication management introduce many design factors that compose a large design space which is impractical to explore manually. AutoSA provides an efficient auto-tuner to find a good design with high performance. We describe the optimization problem formulation as below. Details of the auto-tuner will be discussed in the next chapter.

Given an input program P and a target FPGA device D, AutoSA searches for the design which minimizes the performance metric m without over-utilizing the on-chip resource (DSP, BRAM, URAM) and off-chip bandwidth BW. The complete optimization problem is

summarized as:

$$\begin{aligned} \underset{x \in DC(P)}{\text{minimize}} \quad & m(x) \\ \text{subject to} \quad & r_i(x) \leq R_i, \\ & i = \text{DSP, BRAM, URAM, BW} \end{aligned}$$

where $DC(P)$ includes all legal design tuning choices of the program P as summarized in Table 2.4. The function $m(x)$ estimates the performance of the target design configured by the design factors x. At present, we consider two performance metrics: latency and energy. $r_i(x)$ is the usage of the resource i for the design configured by the design factors x, and R_i is the limit of different types of resources on-chip.

2.5.6 Evaluation Results

AutoSA uses PET [VG12] for extracting the polyhedral IR from the input C programs and performs program transformations using the `isl` library [Ver10]. The core of AutoSA is implemented in C/C++ with about 40K lines of code. The auto-tuner is written in Python. We use Xilinx Vitis 2019.2 for synthesizing and implementing the FPGA designs and target Xilinx Alveo U250 board. We also adopt AutoBridge [GCW21], an automatic place-and-route tool for HLS designs, to improve the design frequency. Table 2.5 summarizes the benchmarks we used to evaluate AutoSA. Source code of all the benchmarks can be found in Appendix 2.A.

By default, we run all the experiments in floating point. In addition, we evaluate the matrix multiplication with 16-bit and 8-bit integer data types. We verify the correctness of the generated hardware designs by computing the difference between the hardware-generated results and the golden results computed by the original algorithms (listed in Appendix 2.A) on CPU. Hardware results are marked wrong if the element-wise difference is greater than the preset threshold. We set the threshold to 0.001 for all floating point experiments and 0 for integer experiments.

In the following sections, we first perform two case studies on matrix multiplication and LU decomposition. We use matrix multiplication to assess the performance of AutoSA and LU decomposition to assess the generality of AutoSA. Lastly, we present the rest of the

Table 2.5: Benchmark description.

Application	Problem Size	#Stmts	LOC
Matrix Multiplication	$[I, J, K] : [1024, 1024, 1024]$	2	6
CNN	$[I, O, H, W, P, Q] : [512, 512, 56, 56, 3, 3]$	2	9
MTTKRP [SRB19]	$[I, K, L, J] : [512, 512, 512, 512]$	2	7
TTMc [SRB19]	$[I, J, K, L, M] : [128, 128, 128, 128, 128]$	2	8
LU Decomposition	$[N] : [12/16/20/24]$	9	26

results on other benchmarks and evaluate the productivity of this tool.

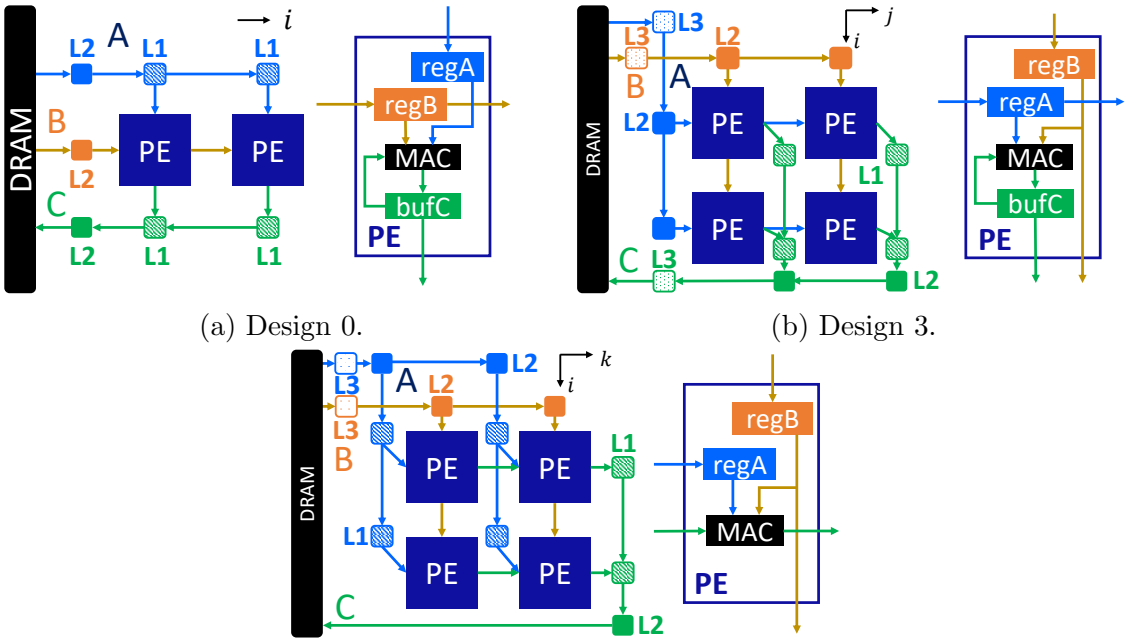
2.5.6.1 Case Study 1: Matrix Multiplication

AutoSA is able to generate six systolic arrays with different dataflows for MM. This is realized by selecting loops ❶i, ❷j, and ❸k as the space loop for 1D arrays, and loops ❹$[i, j]$, ❺$[i, k]$, and ❻$[j, k]$ as space loops for 2D arrays. Detailed architecture of all six arrays can be found in Appendix 2.B. We denote these six arrays as designs 0-5 in sequence. Among these six designs, designs 0 and 1, designs 4 and 5 are symmetric. Therefore, we choose to conduct experiments on designs 0, 3, and 4 for simplicity. Figures 2.21a, 2.21b, and 2.21c depict the architecture of these three designs.

In design 0, as shown in Figure 2.21a, loop i is assigned as the space loop. As a result, matrix A is associated with interior I/O and is fed to each PE directly. Elements of matrix B are reused along the i-axis. Each PE accumulates the elements of matrix C locally. We allocate a local buffer for matrix C (denoted by $bufC$) in the PE to store the intermediate results. After the computation is finished, the final results of matrix C are drained out and sent to the DRAM. Such an architecture can be found in previous works like [JCP05, FKH20].

Design 3, as shown in Figure 2.21b, is generated by selecting loops i and j as the space loops. Elements of matrix A and B are reused along the j-axis and i-axis, respectively. Data of matrix C are accumulated inside PEs and will be drained out after the computation is completed. Such an architecture can be found in previous works [CW18, SRB19, LRZ20, MKN18].

Design 4, as shown in Figure 2.21c, is generated by choosing loops i and k as the space



(a) Design 0.

(b) Design 3.

(c) Design 4.

Figure 2.21: Detailed array architecture of the MM designs 0, 3, 4.

loops. The key difference between design 4 and design 3 is that elements of matrix C are now accumulated along the k-axis. Therefore, local buffers (bufC) are saved through array contraction. However, data from matrix A need to be sent directly to each PE. Data of matrix B are reused along the i-axis. The Google TPU used this architecture [JYP17].

We use designs 0 and 3 to study the performance impacts of different array dimensions, and use designs 3 and 4 to study the impacts of different space-time mappings. All the experiments are done in the floating point. Table 2.6 presents the detailed results of these three designs.

Table 2.6: Performance comparison of different MM designs.

Designs	SA Sizes	MHz	GFLOPs	LUT	FF	BRAM	DSP
0	128×8	346	555	38%	31%	10%	42%
3	13×16×8	300	934	52%	42%	41%	68%
4	13×12×8	300	660	46%	37%	10%	51%

We observe several interesting trade-offs from this table:

Design 0 vs. Design 3: The 1D systolic array limits the dimensions of the design space

with one less space dimension to explore compared to 2D arrays. The best 1D systolic array we found contains 128 PEs with a SIMD factor of 8. Placing more PEs will either lead to routing failure or wasted cycles spent on the padded elements that lowers the effective GFLOPs. As a result, the 1D array (design 0) attains a lower throughput than the 2D array (design 3). However, the 1D systolic array achieves a higher frequency than the 2D arrays, which is contributed by a more regular architecture and less resource.

Design 3 vs. Design 4: We are able to place more PEs in design 3 compared to design 4. This is due to a simpler I/O network. As shown in Figure 2.21b, design 3 exploits both the data reuse of matrix A and B and therefore only generates L2 I/O modules at the PE boundary. However, design 4 (shown in Figure 2.21c) only exploits the reuse of matrix B. Data elements of matrix A need to be sent separately to each PE via L1 I/O modules. This increases the routing complexity and limits the maximal size of the array that this design could achieve.

We found that design 3 achieves a balance between the resource usage and routing complexity and therefore achieves the highest performance among these designs. The comprehensive design space that AutoSA provides and the generality of both the front-end and back-end of AutoSA enable us to explore such as a design space effectively.

Furthermore, we compare the best designs generated by AutoSA with other systolic array compilation frameworks. AutoSA supports different data types. Table 2.7 shows the best results that AutoSA achieved in the floating point, 16-bit and 8-bit integer data types, as well as numbers from the previous works.

Table 2.7: MM performance comparison with previous works.

	PolySA [CW18]	SuSy [LRZ20]	Moss et al. [MKN18]			Jia et al. [JLW20]		AutoSA		
Data Type	FP32	FP32	FP32	int16	int8	FP32	int16	FP32	int16	int8
Platform	Xilinx VU9P	Intel Arria 10	Intel Arria 10			Xilinx VU9P		Xilinx Alveo U250		
MHz	229	202	315	N/A	N/A	265	323	300	250	300
DSP	89%	93%	100%	N/A	N/A	75%	60%	68%	67%	52%
LUTs/ALMs	49%	40%	N/A	N/A	N/A	78%	5%	52%	38%	45%
TOPs	0.56	0.55	0.8	1.53	3.83	0.68	2.64	**0.93**	**3.41**	**6.95**
DSP Efficiency	98%	96%	99%			100%		**94%**		

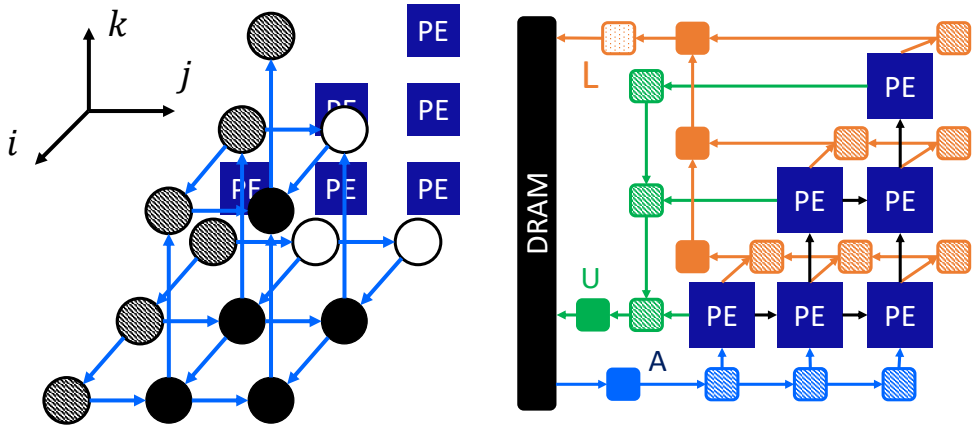
AutoSA achieves 934 GFLOPs for the floating point. For int16, as the number of DSPs for each MAC operation is reduced from 5 (in FP32) to 1, the performance is improved to 3.41 TOPs. For int8, we combine the LUTs and DSPs to implement MACs and achieve 6.95 TOPs. AutoSA achieves higher throughput compared to the previous works with more DSPs utilized or higher frequency. We also compare the DSP efficiency for FP32 designs, AutoSA achieves similar DSP efficiency compared to the previous works.

2.5.6.2 Case Study 2: LU Decomposition

LU decomposition is an important kernel used in solving the systems of linear equations. It factorizes a matrix A as the product of a lower triangular matrix L and an upper triangular matrix U ($A = L \times U$). We choose the algorithm implemented in PolyBench [YP20]. Figure 2.22a shows the dependence graph of the LU algorithm. In the dependence graph, each node represents a loop instance in the program. Nodes are connected if there is any dependence associated with the loop instance they represent. The dependence structure of LU decomposition introduces several new challenges to the systolic array compilation frameworks:

- The iteration domain is in a pyramid shape which cannot be handled by the current Halide-based frameworks that only support rectangular iteration domains.
- The statement instances perform different computations, which introduces a more complex dependence structure that is challenging to analyze manually. We mark the nodes with different computation statements in different shades in Figure 2.22a.

We use LU decomposition as a stress test to assess the generality of our framework. With the general algorithms we have proposed in the previous sections, AutoSA is able to compile such an application and generate a functionable systolic array. Figure 2.22b shows an example array generated by AutoSA by choosing the loops j and k as the space loops. This mapping leads to a 2D systolic array in a triangular shape. In this array, data of matrix A are fed only to the first row of the array. The final results of matrix L are drained out from all the PEs,



(a) Dependence graph. (b) Generated array.

Figure 2.22: The dependence graph of LU decomposition and the mapped array.

while the results of matrix U are drained only from the PEs on the diagonal of the array. Such an architecture can be found in several previous manual designs [Kun88, Raj88].

As there is no reported numbers for LU decomposition in the previous automation frameworks due to its high irregularity, we compare the performance of the generated arrays with LAPACK benchmark [Net20]. The comparison results are shown in Table 2.8. The FPGA results here are collected from RTL simulation with the assumed frequency of 250MHz. The LAPACK routine is evaluated on a server with an Intel Xeon E5-2699 v3 CPU and 189 GB of main memory. We call the functions 10,000 times and calculate the average as the final results.

Table 2.8: LU performance comparison with LAPACK.

Matrix Size	12	16	20	24	Geo. Mean
LAPACK	11.69us	41.06us	78.09us	83.38us	
AutoSA	3.42us	5.23us	7.68us	10.44us	
Speedup	3.4×	7.8×	10.2×	8.0×	**6.8×**

The systolic array achieves an average speedup of 6.8× compared to the LAPACK baseline. This result is not surprising as all the PEs are fully-pipelined and the systolic array extracts the maximal parallelism from the application.

2.5.6.3 Other Results

We have further evaluated AutoSA on three other benchmarks, including the convolutional kernel from the convolutional neural network (CNN) and two tensor contraction kernels, matricized tensor times Khatri-Rao product (MTTKRP), and tensor times matrix-chain (TTMc) that are studied by previous works [SRB19, LRZ20]. Table 2.9 presents the detailed results of these benchmarks and compares them with the results from the other systolic array compilation works.

Table 2.9: Performance results comparison on CNN, MTTKRP, and TTMc.

Benchmark		Platform	SA Sizes (Row×Col×SIMD)	Data Type	LUT	FF	BRAM	DSP	GFLOPs	MHz	DSP Efficiency	Input Code LOC
CNN	[WYZ17]	Intel Arria 10	8×19×8	FP32	59%	40%	47%	81%	602.8	253	97%	9
	AutoSA	Xilinx Alveo U250	16×14×8	FP32	58%	46%	30%	73%	**950.2**	272	97%	9
MTTKRP	[SRB19]	Intel Arria 10	8×9×16	FP32	N/A	N/A	56%	81%	700	204	99%	32
	AutoSA	Xilinx Alveo U250	16×8×8	FP32	42%	32%	26%	67%	**896.7**	296	99%	7
TTMc	[SRB19]	Intel Arria 10	8×10×16	FP32	N/A	N/A	62%	90%	738	205	94%	38
	AutoSA	Xilinx Alveo U250	16×8×8	FP32	42%	32%	21%	67%	**886.2**	290	99%	8

As seen in the table, AutoSA achieves higher throughput than the previous work on all the benchmarks with more DSPs utilized and higher frequency. With the wide coverage of different hardware optimization techniques and the help of an auto-tuner, the designs generated by AutoSA achieve an average DSP efficiency of 99%. Figure 2.23 presents an ablation study of a few optimization techniques applied by AutoSA. All the other techniques are enabled by default. We collect the design latency by incrementally applying each technique. All the performance numbers are normalized against the baseline applied with array partitioning. As can be seen from the figure, all the techniques in the computation management combined together bring an average speedup of 16.1×. The double buffering in the communication management further increases the throughput, on average, by 1.5×. The benefits of double buffering on TTMc is minimal as the application is more compute-intensive. This indicates the needs of an auto-tuner to make the right trade-offs between different design factors.

AutoSA takes a regular C program that describes the original algorithm with minimal modifications. Both of the polyhedral compilation and the auto-tuning only take a few seconds to finish. Taking into account the time of FPGA bitstream generation, which usually finishes in a few hours, AutoSA is able to generate high-performance designs in less than one

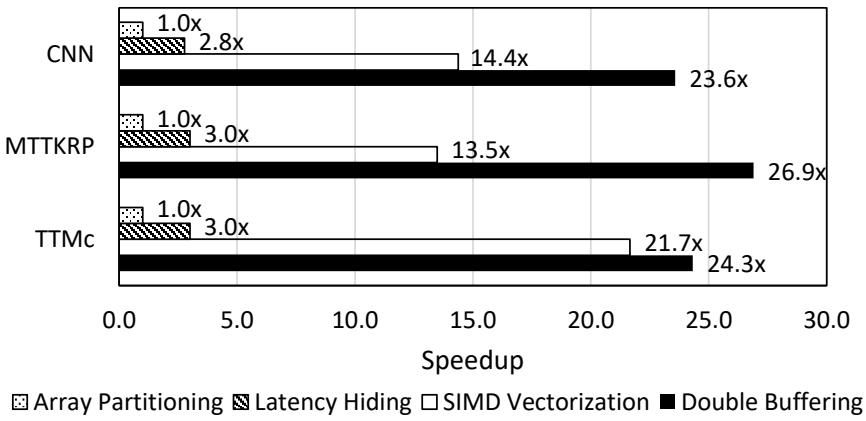


Figure 2.23: Ablation analysis of several hardware optimization techniques.

day, which significantly boosts the productivity of the designing process of systolic arrays.

2.6 Summary

This chapter presents AutoSA, an open-source compiler framework for generating high-performance systolic arrays on FPGA. We present general optimization techniques implemented in the polyhedral framework that help improve the compute and communication efficiency of systolic array designs. We evaluate AutoSA on a suite of benchmarks and achieve high performance. AutoSA strikes a balance between generality, performance, and productivity. We hope such a tool can facilitate more architecture studies and application deployment on systolic arrays in the future.

2.A Source Code of Evaluation Benchmarks

2.A.1 Matrix Multiplication

```
void MM(
  data_t A[I][K], data_t B[K][J], data_t C[I][J]
){
  for (int i = 0; i < I; i++)
    for (int j = 0; j < J; j++) {
      C[i][j] = 0;
      for (int k = 0; k < K; k++)
        C[i][j] += A[i][k] * B[k][j];
    }
}
```

Listing 3: Input code of matrix multiplication.

2.A.2 Convolutional Neural Network (CNN)

```
void CNN(
  data_t fi[I][H + P - 1][W + Q - 1], // input feature maps
  data_t w[O][I][P][Q],               // weights
  data_t fo[O][H][W],                 // output feature maps
){
  for (int o = 0; o < O; o++)
    for (int h = 0; h < H; h++)
      for (int w = 0; w < W; w++) {
        fo[o][h][w] = 0;
        for (int i = 0; i < I; i++)
          for (int p = 0; p < P; p++)
            for (int q = 0; q < Q; q++)
              fo[o][h][w] += fi[i][h + p][w + q] * w[o][i][p][q];
      }
}
```

Listing 4: Input code of CNN.

2.A.3 Matricized Tensor Times Khatri-Rao Product (MTTKRP)

```
void MTTKRP(
  data_t A[I][K][L], data_t B[K][J], data_t C[L][J], data_t D[I][J]
){
  for (int i = 0; i < I; i++)
    for (int j = 0; j < J; j++) {
      D[i][j] = 0;
      for (int k = 0; k < K; k++)
        for (int l = 0; l < L; l++)
          D[i][j] += A[i][k][l] * B[k][j] * C[l][j];
    }
}
```

Listing 5: Input code of MTTKRP.

2.A.4 Chain of Tensor-Matrix Multiplications (TTMc)

```
void TTMc(
  data_t A[I][L][M], data_t B[L][J], data_t C[M][K], data_t D[I][J][K]
){
  for (int i = 0; i < I; i++)
    for (int j = 0; j < J; j++)
      for (int k = 0; k < K; k++) {
        D[i][j][k] = 0;
        for (int l = 0; l < L; l++)
          for (int m = 0; m < M; m++)
            D[i][j][k] += A[i][l][m] * B[l][j] * C[m][k];
      }
}
```

Listing 6: Input code of TTMc.

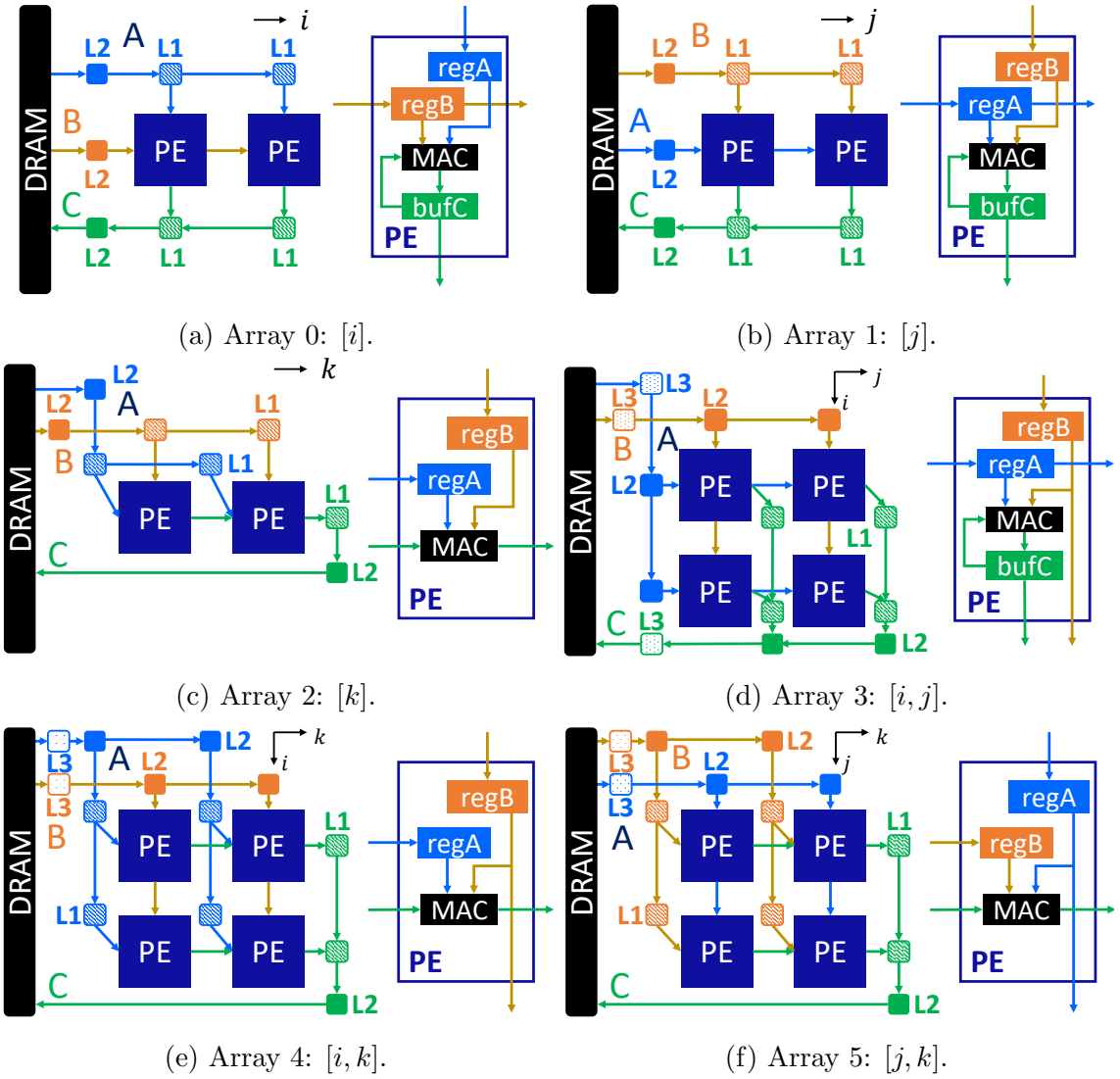
2.A.5 LU Decomposition

```
void LU(
  data_t A[N][N], data_t L[N][N], data_t U[N][N]
){
  data_t prev_V[N][N];
  data_t V[N][N];
  data_t U_tmp[N][N];
  data_t L_tmp[N][N];

  for (int k = 0; k < N; k++)
    for (int j = k; j < N; j++)
      for (int i = k; i < N; i++) {
        if (k == 0)
          prev_V[i][j] = A[i][j];
        else
          prev_V[i][j] = V[i][j];
        if (j == k) {
          U_tmp[i][j] = prev_V[i][j];
          U[j][i] = U_tmp[i][j];
        } else {
          U_tmp[i][j] = U_tmp[i][j - 1];
          if (i == k) {
            L_tmp[i][j] = prev_V[i][j] / U_tmp[i][j];
            L[i][j] = L_tmp[i][j];
          } else {
            L_tmp[i][j] = L_tmp[i - 1][j];
          }
          V[i][j] = prev_V[i][j] - L_tmp[i][j] * U_tmp[i][j];
        }
      }
}
```

Listing 7: Input code of LU decomposition.

2.B Systolic Arrays Generated by AutoSA for Matrix Multiplication



(a) Array 0: $[i]$.

(b) Array 1: $[j]$.

(c) Array 2: $[k]$.

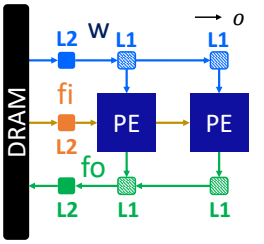
(d) Array 3: $[i, j]$.

(e) Array 4: $[i, k]$.

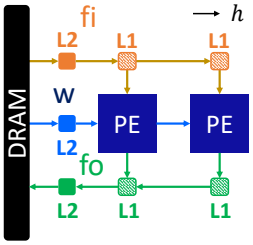
(f) Array 5: $[j, k]$.

Figure 2.24: Architecture of six systolic arrays with different dataflows generated by AutoSA for matrix multiplication.

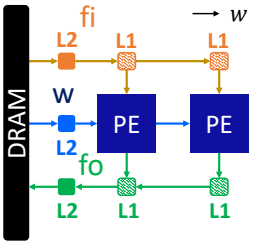
2.C Systolic Arrays Generated by AutoSA for CNN



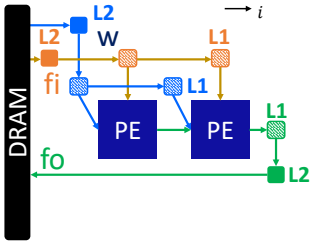
(a) Array 0: $[o]$.



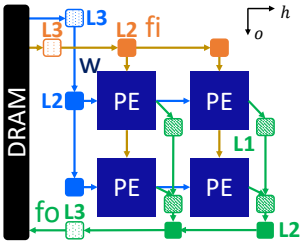
(b) Array 1: $[h]$.



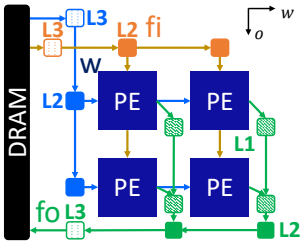
(c) Array 2: $[w]$.



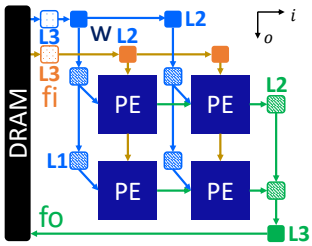
(d) Array 3: $[i]$.



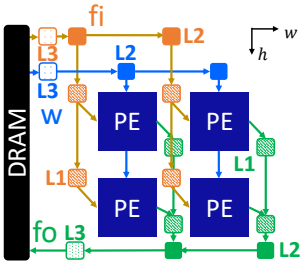
(e) Array 4: $[o, h]$.



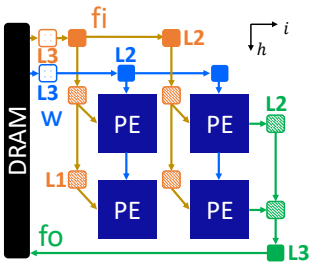
(f) Array 5: $[o, w]$.



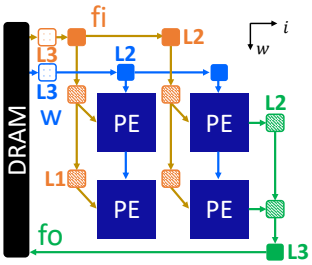
(g) Array 6: $[o, i]$.



(h) Array 7: $[h, w]$.



(i) Array 8: $[h, i]$.



(j) Array 9: $[w, i]$.

Figure 2.25: Architecture of ten systolic arrays with different dataflows generated by AutoSA for CNN.

CHAPTER 3

Efficient Auto-Tuning Approaches for Systolic Arrays

While AutoSA provides an end-to-end solution to generating a complete functionable systolic array, the job to tune the design parameters to achieve high performance is still challenging. The numerous optimization passes that AutoSA applies when generating the designs lead to a vast design space which is infeasible to explore manually. As an example, considering all the available tuning parameters in AutoSA, the design space of a $1024 \times 1024 \times 1024$ matrix multiplication contains up to $O(2^{40})$ design points. Assume that each design point can be evaluated in 1 ms, it will take up to 34.9 years to exhaustively explore the entire design space. This challenge highlights the strong needs for efficient auto-tuning approaches.

In this chapter, we discuss our efforts in developing efficient auto-tuning approaches for AutoSA. We have developed an auto-tuning framework, Odyssey, that combines both a mathematical programming-based optimizer and evolutionary search to achieve high sample efficiency. As a result, for the same $1024 \times 1024 \times 1024$ matrix multiplication, Odyssey can locate a good design that achieves 90% of the best performance found by an exhaustive search in 5 seconds. This auto-tuner is integrated seamlessly with AutoSA and open-sourced.

3.1 Background and Related Work

Performance tuning has long been an important topic given the complexity of hardware systems and applications. On conventional hardware platforms such as CPU and GPU, hardware vendors provide high-performance libraries targeting a specific set of applications, e.g., Intel MKL [WZS14] and Nvidia CuDNN [CWV14]. However, this approach is costly when being applied to new hardware systems and applications given the considerable efforts of manual tuning. This situation is exacerbated in the domain of deep learning where new applications are emerging at an unprecedented speed. This poses a significant challenge to the library-based approaches. As a result, recent deep learning compilation frameworks such as TVM [CMJ18] and Halide [RBA13] which offer efficient auto-tuning support have attracted a great deal of interest from both academia and industry. These frameworks provide expressive DSLs for constructing the design space and leverage machine-learning techniques to scan the design space efficiently. Such an approach has shown its success in terms of both portability and performance. As an example, the latest auto-tuning framework for TVM, Ansor [ZJS20], demonstrates that its evolutionary search-based auto-tuner is able to improve the execution time of deep neural networks over the library-based approaches by up to 3.8× and 1.7× on Intel CPU and Nvidia GPU, respectively.

The challenge of performance tuning is further intensified on domain-specific architectures, on which designers are exposed to more hardware details that open up a much larger design space to explore. In the previous chapter, we have introduced various hardware optimization techniques that AutoSA applies to generate a design with high performance. When considering all the available design parameters offered by the compiler, the design space bloats up to $O(2^{40})$ for a $1024 \times 1024 \times 1024$ matrix multiplication. Such a vast design space is infeasible to explore manually, underscoring the strong needs for efficient auto-tuning approaches.

A complete auto-tuning problem of systolic arrays contains two dimensions: the *construction* of the design space and the implementation of the *search method*. An incomplete design space may leave out optimal designs and lead to an inferior performance, while a more comprehensive design space brings a higher complexity and poses greater challenges to the

search methods.

Many previous works have attempted this challenging task by looking into different dimensions of the design space and proposing various auto-tuning approaches [PRS19, KJK20, HKD21, DKA19, CKR20, LGW21, YGL20, KK20]. Table 3.1 lists several representative works that have explored the design space of systolic array architectures. Note that some of the listed works in the table cover a broader set of architectures beyond systolic arrays. In this work, we only focus on the systolic array architecture. In the rest of this section, we discuss the prior works from two dimensions: the construction of the design space and search methods.

Table 3.1: Comparison between different systolic array architecture auto-tuning frameworks.

	Design Space							Search Methods
	Dataflows	Loop Transformation			Performance Models		Input	
		Permutation	Tiling	Non-Divisors	Method	Prologue/ Epilogue		
Timeloop [PRS19]	Y	Y	Y	N	Analytical Model	N	Manual	Exhaustive w/ Pruning Random Search
dMazeRunner [DKA19]	Y	Y	Y	N	Analytical Model	N	Manual	Exhaustive w/ Pruning
Interstellar [YGL20]	Y	Y	Y	N	Analytical Model	N/A	Manual	Exhaustive w/ Pruning
Marvel [CKR20]	Y	Y	Y	N	Analytical Model	Y	Manual	Mathematical Programming + Exhaustive w/ Pruning
ConfuciuX [KJK20]	Y	Fixed	Fixed	N/A	Analytical Model	Y	Manual	Reinforcement Learning + Evolutionary Search
CoSA [HKD21]	Y	Y	Y	N	Analytical Model	Y	Manual	Mathematical Programming
TENET [LGW21]	Y	Fixed	Fixed	N/A	Analytical Model	N	Manual	Exhaustive w/ Pruning
Odyssey	Y	Y	Y	Y	Program-Derived Analytical Model	Y	Auto	Mathematical Programming + Evolutionary Search

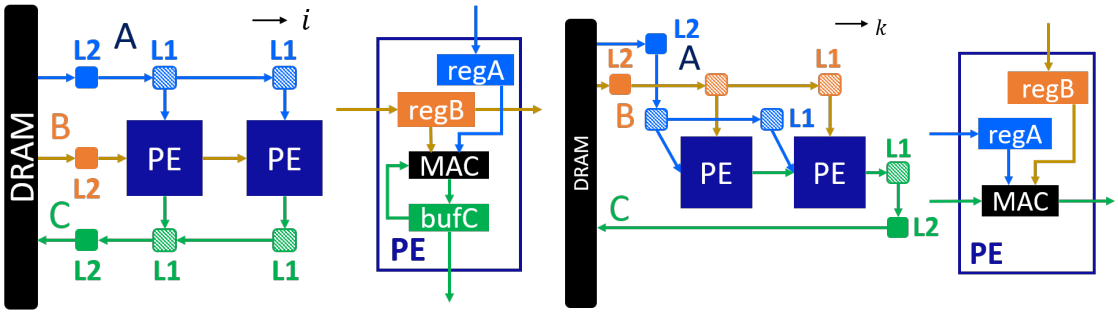
* ConfuciuX and TENET use a fixed set of loop permutation orderings and tiling factors for each dataflow. We mark them as *fixed* in the table.

3.1.1 Design Space Construction

We consider the following dimensions in the design space.

Dataflows AutoSA generates systolic arrays with various dataflows by choosing different space-time mappings. Figure 3.1 shows two systolic arrays with different dataflows for matrix multiplication. In the left array, we choose i as the space dimension. Data of matrix B are reused across PEs, and data of matrix A and C are directly transferred at the PE boundary. In comparison, the systolic array on the right uses k as the space dimension. As a result, both elements from matrix A and B are directly sent to PEs, while data of matrix C are accumulated across PEs. These two arrays exploit different types of parallelism and data

reuse from the application, which may lead to distinct performance on the same application.



(a) Dataflow $[i]$ (b) Dataflow $[k]$

Figure 3.1: Architecture of two different systolic arrays for MM.

Loop transformation AutoSA applies a sequence of loop transformations in the stage of computation management. We consider two minor dimensions here: loop permutation and loop tiling. When selecting the tiling factors for array partitioning, we consider if non-divisors are included. Many prior works chose to use the divisors as the tiling factors for simplicity [PRS19, DKA19, YGL20, CKR20, HKD21]. However, this choice could lead to significant performance loss. Our experiments reveal a 39% performance loss when only considering the divisors during the design space exploration for a $1024 \times 1024 \times 1024$ matrix multiplication.

Performance model As the real hardware evaluation is usually expensive, previous works chose to use analytical models to approximate the hardware performance. The accuracy of such models plays an important role in determining the quality of the search results. For example, when estimating the latency of systolic arrays designs, prior works such as TENET [LGW21] and dMazeRunner [DKA19] chose to compute the maximum of the compute and communication latency as an approximation to the real hardware latency. However, such a model overlooks the prologue and epilogue phases in the critical path, thus underestimating the design latency. For the same matrix multiplication problem as mentioned above, such an inaccurate model leads to inferior designs with a 9% performance gap compared to the best design found with the accurate latency model that considers the prologue/epilogue latency.

3.1.2 Search Methods

Table 3.2: Summary of search methods.

Category	Methods	Prior Works
Brute-Force	Random Search, Exhaustive Search w/ Pruning	Timeloop [PRS19], dMazeRunner [DKA19], Interstellar [YGL20]
MP-Based		CoSA [HKD21], [LXS21]
Iterative	Bayesian Optimization, Reinforcement Learning, Simulated Annealing, Evolutionary Search, etc.	ConfuciuX [KJK20], AutoTVM [CZY18], Ansor [ZJS20], [RHA17]

The vast design space makes it infeasible to explore with exhaustive search. Prior works have adopted various approaches to tackle this challenge, as summarized in Table 3.1. We classify the search methods into three categories: brute-force methods, mathematical programming (MP)-based methods, and iterative methods.

Brute-force methods Many previous works used brute-force approaches such as random search [PRS19, CZY18] and exhaustive search with pruning [WYZ17, WGC21, DKA19, YGL20, CKR20, LGW21]. These methods are sufficient for designs with a relatively small design space. However, the converge time stretches significantly for large problem sizes due to the low sample efficiency. Another commonly overlooked problem for the pruning-based exhaustive search is the choice of pruning heuristics. As an example, several prior works have chosen to use resource utilization as the pruning heuristic [CKR20, DKA19, WYZ17]. However, this heuristic needs to be adjusted dynamically to avoid over-pruning the profitable designs. For instance, on Xilinx Alveo U250 FPGA, the best systolic array design for a $64 \times 64 \times 64$ matrix multiplication (MM), only takes 14% of the total DSP resource, given the limited parallelism. When increasing the matrix dimension from 64 to 1024, the best systolic array will use 98% of the total DSP resource. The previous work dMazeRunner [DKA19] chose 60% as the utilization pruning threshold. Using this number will prune away all the feasible designs for the small MM problem as presented above. The process of choosing an ideal pruning heuristic requires non-trivial efforts as programmers usually face an unknown design space without any prior knowledge of the performance distribution of the feasible designs. Taking into account all of the challenges above, brute-force approaches are limited in practice when handling complex hardware systems and applications.

Mathematical programming-based methods MP-based methods formulate the search problem as a closed-formed mathematical optimization problem and solve it with optimization solvers [LXS21, LSV19, CKR20, HKD21]. This method can converge to a solution in seconds, however, it usually yields sub-optimal search results due to the over-simplification of the design space. The recent work CoSA [HKD21] models the design space including dataflow selections, loop permutation and tiling as an mixed integer programming (MIP) problem. However, to accommodate for the formulation requirements of MIP, it uses an inaccurate performance model with many simplifications. For example, it computes the memory usage as the product of all the tiling factors which overlooks the banking property of SRAMs. It sets the objective functions as a linear combination of many high-order factors such as resource utilization and data communication volume. The less accurate models used in such approaches will lead to inferior designs. We conducted an experiment in which we modeled the design optimization problem for AutoSA-generated designs as a non-linear optimization problem and used the off-the-shelf solver (AMPL [Gay15] with Ipopt [COI21]) to generate the design solutions. The best design obtained from this approach is 1.5× slower than the optimal design found by an exhaustive search.

Iterative methods Iterative-based tuning methods reach a sweet spot between the first two categories. It requires minimal prior knowledge of the design space, making it easier to adapt to different search problems. It achieves a higher sample efficiency with the feedback-based exploration strategy compared to the brute-force approaches. Furthermore, programmers can supply it with an accurate and detailed performance model or evaluate the design point on the hardware directly, which helps cover the optimal designs with high performance compared to the MP-based methods. Many recent works have deployed different iterative search methods for auto-tuning, such as simulated annealing and evolutionary search in [CZY18, ZJS20], Bayesian optimization in [RHA17]. and reinforcement learning in [KJK20]. Nonetheless, iterative tuning methods are usually involved with many hyper-parameters which need to be carefully adjusted to achieve the best performance. This additional complexity leads to trade-offs in choosing different tuning methods for various search problems.

As shown in Table 3.1, most previous works only covered part of the design space, and relied on brute-force methods such as exhaustive search with pruning to explore the design space. The recent work ConfuciuX [KJK20] proposed a search method based on reinforcement learning and evolutionary search to explore the design space efficiently. However, the search space is highly limited as they only considered choosing the memory size and PE numbers in a fixed parameter set.

The failure in either the comprehensiveness and accuracy of the design space modeling or the efficiency of the search methods will impact the quality of the search results, as well as the architecture conclusions derived from these results. Unfortunately, regardless of the plethora of past studies on this topic, we observe no prior work that obtained a balance between these two points. This situation has motivated us to tackle this challenge. Specifically, the goal in this project is to *develop an efficient design space exploration framework that covers a comprehensive and accurate design space for systolic arrays.*

3.2 Contributions

In this chapter, we propose a new automatic design space exploration framework for systolic arrays, Odyssey[1], which explores such an architecture efficiently. Figure 3.2 depicts the proposed tuning flow. Odyssey leverages AutoSA, an open-source systolic array compiler, to construct the design space automatically. AutoSA takes in a C program that describes the algorithm to be mapped to systolic arrays and applies a sequence of hardware optimizations based on the polyhedral framework to generate the hardware designs in Xilinx HLS C. We extend the AutoSA framework to generate a design description file in JSON that covers the full details of the generated hardware. Odyssey then builds Python functions for estimating the design performance including the latency and resource usage from the description file. After that, it plugs in the performance model into the auto-tuner to search for the optimal design. Inside the auto-tuner, Odyssey implements a two-stage flow that starts with a mathematical programming (MP)-based optimizer to generate an initial high-quality design.

[1]Odyssey is abbreviated from AUtomatic DEsign Space exploration for SYstolic arrays.

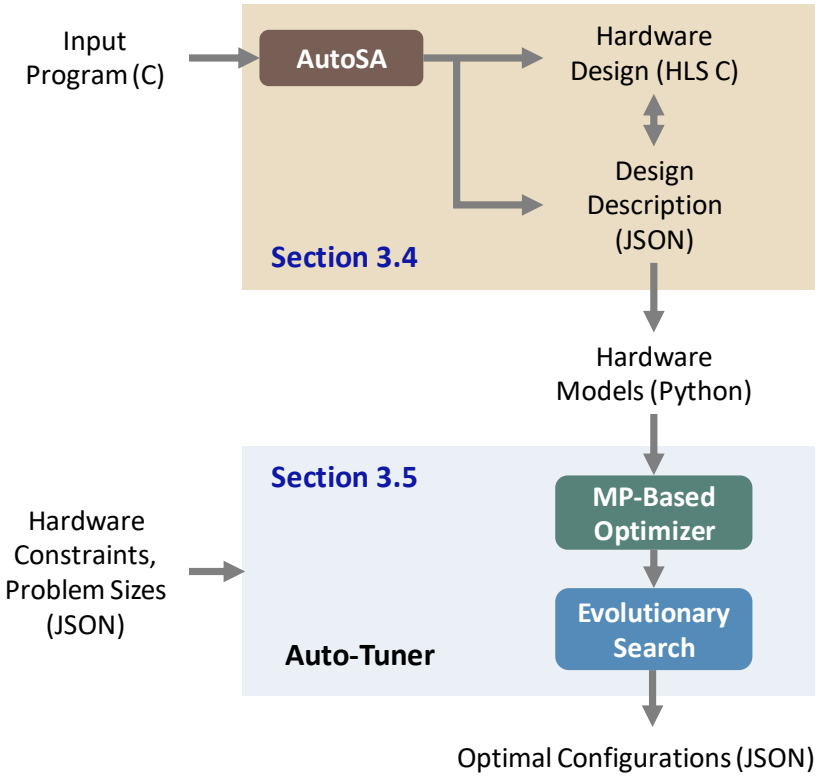


Figure 3.2: Overview of the Odyssey framework.

This sample design is then sent to the evolutionary search as the initial population that helps reduce the convergence time and improve the search quality.

Our work surpasses the previous works in multiple dimensions. First, we cover a comprehensive design space with accurate performance modeling. Previous works [CKR20, HKD21, LGW21, KCS20, YGL20] have shown that different design factors, including the choices of dataflows, loop permutation and tiling, affect the performance of the systolic arrays in an intertwined manner. All of these design factors can be explored by AutoSA and are taken into account by the Odyssey framework. In addition to the comprehensive design space, we achieve accurate performance modeling of the generated designs. Odyssey derives the performance models directly from the generated hardware, in contrast to previous works that built analytical models based on the abstract architectures. For example, the latency models are generated by traversing the ASTs of the generated HLS designs from AutoSA. With such an approach, we achieve only 1.99% of the latency estimation errors compared to the real hardware execution. Last but not least, the entire flow is fully automated. We use AutoSA to

generate all possible architecture candidates and produce the hardware models for auto-tuning without any user intervention. In comparison, several prior works [DKA19, KCS20, LGW21] require designers to explicitly describe the hardware models which takes non-trivial efforts and is error-prone.

In addition to the comprehensive and accurate design space modeling, we propose a highly efficient auto-tuning method to explore this design space. The proposed algorithm combines the MP-based and iterative methods. It uses the evolutionary search as the backbone and leverages the MP-based optimizer to generate high-quality seeds to reduce the convergence time. Our search framework can find designs with high performance in a short amount of time. For example, for a $1024 \times 1024 \times 1024$ matrix multiplication, the proposed auto-tuner locates a design that achieves 90% of the optimal performance found by an exhaustive search in less than 5 seconds.

The rest of this chapter is organized as follows. Section 3.3 covers the scope of Odyssey framework in terms of the dimensions of the design space and the supported applications. Section 3.4 explains how the design space is constructed in Odyssey. Then, Section 3.5 introduces the proposed search methods to explore such a design space. Section 3.6 validates the accuracy of the performance models generated by Odyssey and evaluates the search performance of the auto-tuner. We have also used Odyssey to investigate the architecture trade-offs of systolic arrays for two applications, matrix multiplication and CNN. Section 3.7 discusses the lessons learned from this work and Section 3.8 concludes this work.

3.3 Scope of the Odyssey Framework

3.3.1 Scope of the Design Space

As introduced in Sections 2.5.2 and 2.5.3, AutoSA separates the construction of PE arrays and the I/O networks into two stages: computation management and communication management. In each stage, AutoSA applies multiple passes of hardware optimizations to boost the compute and communication efficiency of the generated design. AutoSA implements tunable knobs to

control each step during the compilation process. Table 2.4 summarizes the available tuning options in AutoSA.

In this work, we focus on the tuning space of the computation management. We left the communication optimization to be handled by the compiler automatically. Specifically, we will apply double-buffering automatically on all the on-chip buffers inside the I/O modules and choose the highest feasible[2] data-packing factors for maximizing the on-chip data transfer throughput. In the future, we will consider extending the framework to consider the tuning options in the communication management, such as the selection of double buffering and data packing factors.

Table 3.3: Scope of the AutoSA design space explored by Odyssey.

AutoSA Compilation Pass		Space-Time Transformation	Array Partitioning	Latency Hiding	SIMD Vectorization
Dataflow		✓			
Loop Tiling	Divisor		✓	✓	✓
	Non-Divisor		✓		
Loop Permutation			✓		

Table 3.3 summarizes the design space covered by Odyssey. Odyssey explores different dataflows in the space-time transformation and the loop tiling factors in array partitioning, latency hiding, and SIMD vectorization. Odyssey also supports the exploration of non-divisor tiling factors in the step of array partitioning. If a non-divisor tiling factor is found by the auto-tuner, we ask the programmer to manually modify the original problem dimension to be a multiple of the tiling factor. Otherwise, additional loop guards will be inserted in the HLS program that leads to an inferior performance. Tiling factors of latency hiding and SIMD vectorization are required to be divisors as the sub-array dimension is fixed after the array partitioning and non-divisor tiling factors will introduce prologue/epilogue phases in the computation and costly max/min operations in the generated hardware. Lastly, Odyssey

[2]Data-packing factors should be divisors of the last dimension of the packed array and do not surpass the maximal packable elements threshold set by the compiler. By default, we set the data-packing factors upper bound in AutoSA using the argument `kernel[]->data_pack[8,64,64]`. With this argument, we instruct AutoSA to pack at most 8/64/64 bytes of data between L1/L2/L3 I/O modules.

explores loop permutation in the step of array partitioning. As will be explained later, different loop orderings in the array partitioning band may lead to different performance.

Considering all the tuning parameters covered by Odyssey, the design space of a $1024 \times 1024 \times 1024$ MM contains up to $O(2^{40})$ design points. We summarize the design choices for this example in Table 3.4. Such a vast design is intractable to handle manually, which highlights the strong needs for efficient auto-tuning approaches.

Table 3.4: Design space covered by Odyssey for a 1024×1024×1024 MM.

Optimization Pass	Design Space			
	Loop Permutation	Loop Tiling	Combinatorial Choices	#Design Points
Space-Time Transformation	N/A	N/A	6	6
Array Partitioning	6	2^{30}	N/A	6×2^{30}
Latency Hiding	N/A	$O(2^{10})$	N/A	$O(2^{10})$
SIMD Vectorization	N/A	4	N/A	4
			Total	$O(2^{40})$

* For SIMD vectorization, we limit the maximal tiling factor to 8. Tiling factors are chosen from the set $[1, 2, 4, 8]$.

3.3.2 Scope of the Supported Applications

The second restriction on the current auto-tuning framework is that we limit the input programs to ones with rectangular iteration domains. The current framework cannot generate performance models automatically for programs with non-rectangular iteration domains such as LU decomposition. For such a case, programmers may implement the design performance models in Python files manually and supply it to the auto-tuner.

3.4 Constructing the Design Space

In this section, we discuss how we construct the design space. We introduce the enhancement to AutoSA for generating the design description file for auto-tuning. Our design space considers loop permutation. However, it is worth noting that the loop permutation orderings can be pruned effectively without filtering out the optimal designs. We cover the pruning

heuristics in detail in the second part of this section.

3.4.1 Enhancement to AutoSA

In the current framework, we use AutoSA to generate a design descriptor that contains all the necessary information for estimating the design performance. This descriptor has five components as listed below:

- Abstract syntax tree (AST): ASTs of all the hardware modules, which will be used for latency estimation.
- Memory information: Properties of on-chip buffers including buffer size, data type, and memory banking, which are used for memory usage estimation.
- Compute information: Information about the PE computation logic, such as the SIMD lanes and computation statements. We maintain an internal database about the required computation resources for different types of computation statements. This information is used for computation resource estimation.
- Array Topology: Detailed information about the topology of the systolic array, such as the array dimension and the number of different hardware modules. This information, along with the memory and computation information are used for resource estimation.
- Tuning parameters: A list of all the tunable loop tiling factors.

The auto-tuner uses this description file to generate a Python file comprised of functions for estimating the design performance. During the search, the auto-tuner will sample the design space and plug in different tuning parameters into the performance models for assessing the design performance. To support this functionality, all the performance models should be symbolic expressions of the tuning parameters. However, this requirement cannot be satisfied by the original AutoSA framework which is built upon the integer set library (`isl`) [Ver10] that only supports loop tiling with constant factors.

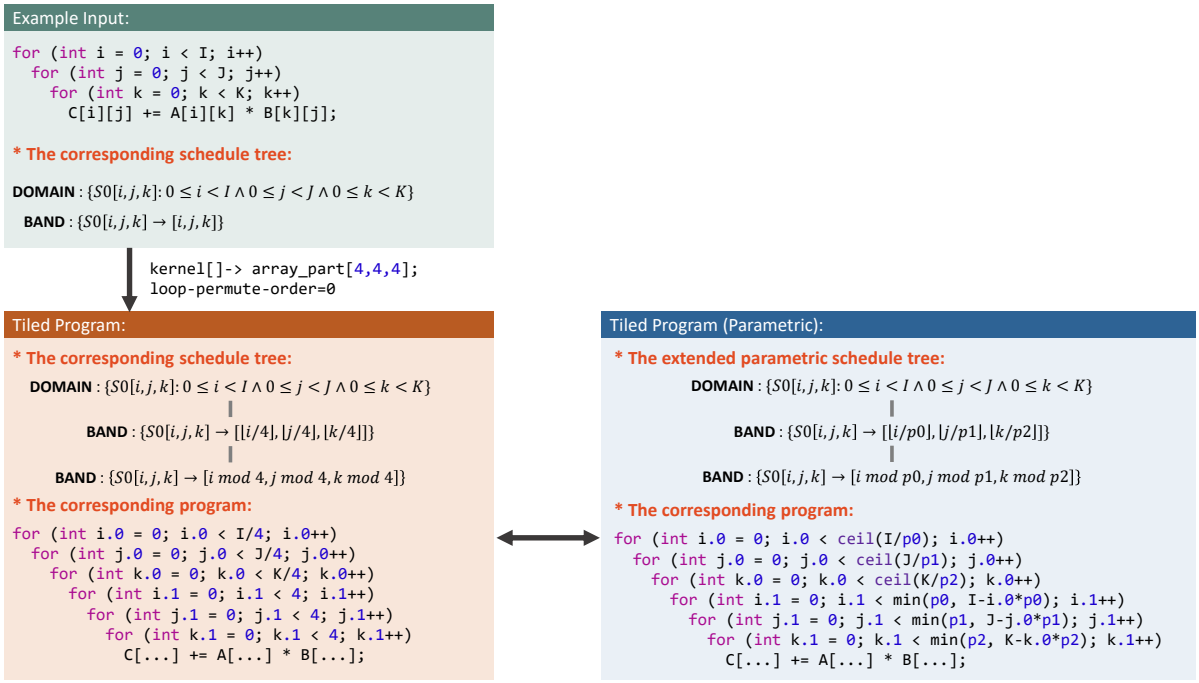


Figure 3.3: Examples of applying array partitioning with the extended schedule IR.

To generate symbolic expressions, we extend AutoSA with a new layer of scheduling IR that shadows all of the loop transformations applied by AutoSA. Figure 3.3 shows an example when applying the loop tiling during the array partitioning with the extended scheduling IR. `isl` uses a tree-based internal structure named schedule tree to represent the loop structure. The new scheduling IR keeps the exact structure of the internal schedule tree. When AutoSA applies the loop tiling with a constant factor on the original schedule tree, it replaces the constant factor with an undetermined parameter and updates the corresponding schedule tree with parametric tiling instead. In such a manner, we shadow all the loop transformations performed by AutoSA while maintaining a separate parametric schedule that can be used to generate a parametric loop program and produce symbolic expressions as required by the design description file.

3.4.2 Loop Permutation Pruning

Odyssey explores different loop permutation orderings in the step of array partitioning. Different loop orderings could lead to different systolic array architectures. Figure 3.4

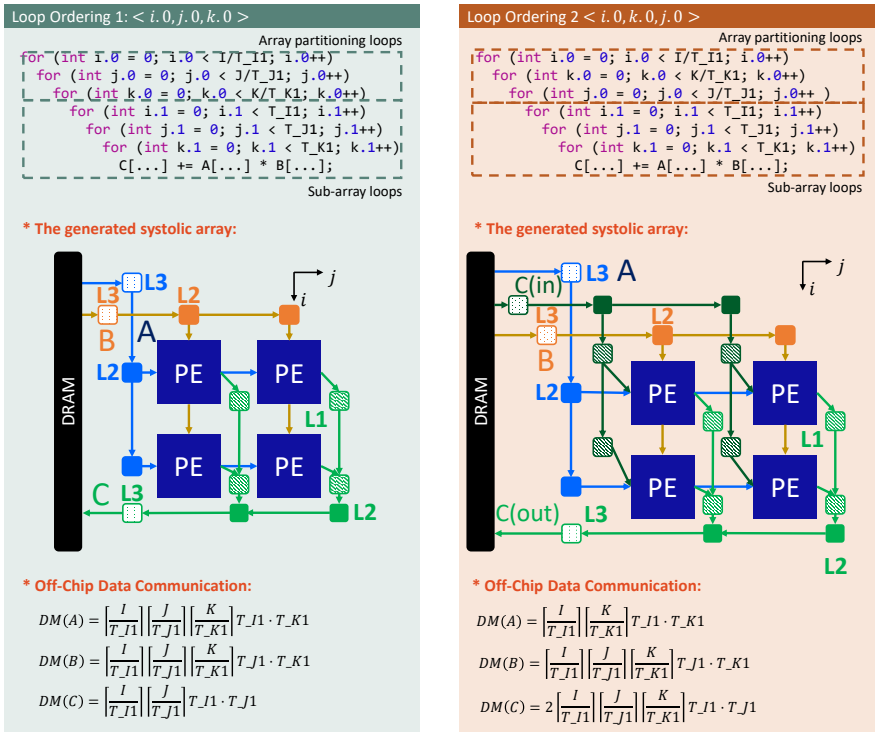


Figure 3.4: Example of systolic arrays designs generated with different loop permutation orderings.

presents two loop orderings with the systolic arrays they generate. With the first loop ordering $< i.0, j.0, k.0 >$, intermediate data of matrix C are accumulated locally on-chip before draining out. In comparison, when using the second loop ordering $< i.0, k.0, j.0 >$, intermediate data of matrix C need to be swapped off-chip at each loop $j.0$ iteration. Here, we assume that the tiled data in one array partition fully occupy the on-chip memory C.

$$T(A) + T(B) + T(C) = T_I1 \times T_K1 + T_J1 \times T_K1 + T_I1 \times T_J1 \leq C \tag{3.1}$$

At each array partition, we load a tile from matrix A with the size of $T(A) = T_I1 \times T_K1$, a tile from matrix B with the size of $T(B) = T_J1 \times T_K1$, and compute a tile of matrix C with the size of $T(C) = T_I1 \times T_J1$. These three data tiles fully occupy the on-chip memory C.

With the loop ordering $< i.0, k.0, j.0 >$, when a new tile of matrix C is computed, we will need to swap out the old C tile to spare the memory space for the new data. On the systolic

array architecture, in addition to the I/O modules to drain out the matrix C (`C(out)`), additional I/O modules (`C(in)`) are allocated to load in the intermediate results from the external memory. Compared to the first loop ordering $< i.0, j.0, k.0 >$, the second loop ordering $< i.0, k.0, j.0 >$ introduces additional hardware overheads and a higher volume of off-chip data communication that could impact the final performance negatively.

By default, AutoSA enumerates all the loop permutation orderings. For matrix multiplication (MM), there are $3! = 6$ different orderings to consider. The number grows larger for a more complicated application like convolutional neural network (CNN).

```
1 for (int o = 0; o < O; o++)
2   for (int h = 0; o < H; h++)
3     for (int w = 0; o < W; w++)
4       for (int i = 0; o < I; i++)
5         for (int p = 0; o < P; p++)
6           for (int q = 0; o < Q; q++)
7             fo[o][h][w] += fi[i][h+p][w+q] * w[o][i][p][q];
```

Listing 8: Example code of a CNN layer. The initialization of the output feature map array fo is omitted for brevity.

Listing 8 shows the example code of one CNN layer. The six-level nested loop will lead to $6! = 720$ different loop orderings. However, as pointed out by previous works [LXS21, LSV19, DKA19], among all the loop orderings, many of them are dominated by a few orderings in terms of performance, thus can be safely pruned without leaving out the optimal points. Next, we will show that with proper pruning, we could reduce the number of loop orderings to consider for both MM and CNN to only 3. We consider resource usage and latency when assessing the design performance.

Different loop orderings will impact the structure of the I/O network, resulting in different resource usage. In Figure 3.4, we showed that hoisting the loop $k.0$ from the innermost position of the array partitioning loop band introduces additional I/O modules for transferring the intermediate results of matrix C into the array, and, therefore, increases the total resource usage. The key takeaway here is that by placing the loops that carry the flow dependences innermost, intermediate data are accumulated locally on-chip without the needs to be swapped

out across different array partitions, thus avoiding the resource overhead brought by the additional I/O modules. We summarize this observation as follows.

Lemma 3.4.1. *For any array reference r in the program, let $RL(r)$ be the array partitioning loops that carry the flow dependences associated with r, placing the loops in $RL(r)$ to the innermost of the array partitioning loop band helps reduce the resource usage by eliminating the additional I/O modules for loading in the intermediate results of r.*

Latency-wise, different loop orderings will result in different volumes of off-chip data communication. We compute the volume of off-chip data movement for the two example designs with different orderings in Figure 3.4. For the first ordering $< i.0, j.0, k.0 >$, the final results of matrix C are only drained out at the last iteration of loop $k.0$, leading to a total amount of data movement as:

$$DM(C) = \lceil \frac{I}{T_I1} \rceil \lceil \frac{J}{T_J1} \rceil T_I1 \cdot T_J1 \tag{3.2}$$

As for the second ordering $< i.0, k.0, j.0 >$, the intermediate results of matrix C are swapped off-chip at each loop iteration of $j.0$. We compute the total data movement of matrix C as:

$$DM(C) = 2\lceil \frac{I}{T_I1} \rceil \lceil \frac{K}{T_K1} \rceil \lceil \frac{J}{T_J1} \rceil T_I1 \cdot T_J1 \tag{3.3}$$

The factor 2 is multiplied here to take into account both the inbound and outbound traffic of matrix C. In comparison to the first loop ordering, the second loop ordering introduces a higher volume of data movement for matrix C. For communication-bound applications, this could lead to a longer latency. In addition to the loops that carry the flow dependence, orders of loops that carry the read dependence will impact the data communication as well. We use matrix A as an example. As shown in Figure 3.4, under the loop ordering $< i.0, j.0, k.0 >$, at each array partition, we load new array tiles with a size of $T_I1 \times T_K1$ from array A, leading to a total communication volume computed as below:

$$DM(A) = \lceil \frac{I}{T_I1} \rceil \lceil \frac{J}{T_J1} \rceil \lceil \frac{K}{T_K1} \rceil T_I1 \cdot T_K1 \tag{3.4}$$

In comparison, under the loop ordering $< i.0, k.0, j.0 >$, data of matrix A are reused along the loop $j.0$. Therefore, new tiles of data are only loaded at each new loop iteration of loop $k.0$, leading to a data movement as:

$$DM(A) = \lceil \frac{I}{T_I1} \rceil \lceil \frac{K}{T_K1} \rceil T_I1 \cdot T_K1 \tag{3.5}$$

As long as the loop $j.0$ is not the innermost loop, new array tiles of matrix A always need to be loaded from the external memory at each array partition. Based on the analysis above, we draw the following conclusion.

Lemma 3.4.2. *For any array reference r in the program, let $RL(r)$ be the array partitioning loops that carry the read or flow dependences associated with r, placing the loops in $RL(r)$ to the innermost of the array partitioning loop band reduces the off-chip data communication of r, thus influencing the design latency.*

Note that while the off-chip data communication is a critical factor to the design latency, it does not necessarily lead to a worse performance. For compute-bound applications, minimizing the off-chip data communication might not affect the total design latency.

For the MM example, loop $i.0$ carries the read dependence for array B, loop $j.0$ carries the read dependence for array A, and the loop $k.0$ carries the flow dependence for array C. In total, there are three unique loop orderings for this program which lead to systolic arrays with a potentially different performance. Note that as long as the innermost loop is fixed, the ordering of other loops will not impact the performance. We show another loop ordering $< j.0, i.0, k.0 >$ in Figure 3.5. This ordering is equivalent to $< i.0, j.0, k.0 >$ in terms of the achievable performance. In the rest of this chapter, we use the annotation $< [i.0, j, 0], [k.0] >$ to identify a unique loop ordering. All the loops in the same brackets can be permuted freely with equivalent performance. We will choose one ordering randomly in practice. Table 3.5 summarizes the three different loop orderings for MM.

In the MM example, each array is only reused on one loop. Next, we illustrate a more complicated case where array references are associated with more than one reuse loops.

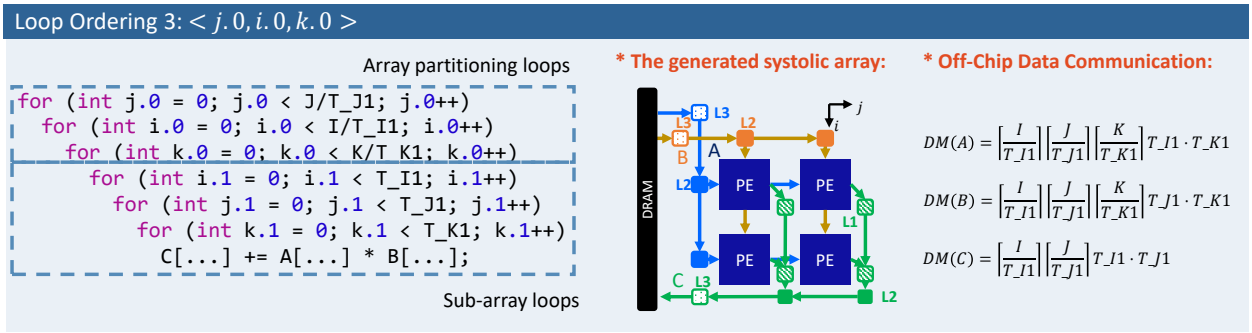


Figure 3.5: Example of a different loop permutation ordering for MM.

Table 3.5: Unique permutation orderings for MM and CNN.

Application	Loop Permutation Orderings
MM	<[i,j],k>, <[i,k], j>, <[j,k],i>
CNN	<[o,h,w],[i,p,q]>, <[o,i,p,q],[h,w]>, <[i,h,w,p,q],o>

Listing 8 shows the code of one CNN layer. In this code, the output feature map array fo is reused along three loops i, p, and q. When placing all these three loops innermost with the ordering $< o.0, h.0, w.0, i.0, p.0, q.0 >$, we are able to generate a systolic array that accumulates the intermediate results locally on-chip without being swapped out in the middle. Changing the order of these three loops will not affect the array structure or the off-chip data communication, thus not affecting the array performance. The left side of Figure 3.6 depicts the architecture of the generated array. However, if we hoist any of these three loops, as shown in the right design in Figure 3.6 with the loop ordering $< o.0, h.0, i.0, w.0, p.0, q.0 >$, intermediate results are then swapped out at the loop $w.0$, leading to a different array architecture and data communication pattern. However, since this new architecture introduces more resource overheads and an increased data communication, its performance is dominated by the left design that places all the reuse loops innermost. Therefore, this ordering can be safely pruned without leaving out any optimal designs. We summarize this observation as below.

Lemma 3.4.3. *For any array reference r in the program, let $RL(r)$ be the array partitioning loops that carry the read or flow dependences associated with r and $NRL(r)$ be the rest array partitioning loops, the loop ordering $< NRL(r), RL(r) >$ dominates any other loop ordering*

in terms of the resource usage of I/O modules and off-chip data communication associated with r.

Putting it all together, we have a complete picture of the pruning strategy for loop permutation orderings. For each array reference r in the program, we could identify the loops $RL(r)$ that carry the flow or read dependences that are associated with r. Then, we could generate all unique loop orderings that may lead to different resource usage and design latency, by placing loops $RL(r)$ to the innermost of the array partitioning loop band. Any other loop ordering that hoists any loop from $RL(r)$ is dominated by this ordering in terms of resource usage and off-chip data communication associated with r. We formulate this pruning strategy as below.

Theorem 3.4.4. *Given a program that can be mapped to systolic arrays, let $RL(r)$ be the set of array partitioning loops that carry the read/flow dependences associated with the array reference r, and $NRL(r)$ be the rest of the loops in the array partitioning loop band, the set of unique loop orderings O can be obtained as the union of loop orderings in the form of $< NRL(r), RL(r) >$ for each array reference r. All the other loop orderings are dominated by O in terms of resource usage and latency. Note that $RL(r)$ could be an empty set if there is no read/flow dependence associated with the r. For this case, the loop ordering is in the form of $< NRL(r) >$, and is added into O for consideration.*

Proof. Assume the above statement is false, i.e., there is a loop ordering o' out of the set O that achieves better performance than loop orderings in O. Then, there exists at least one loop l_{rl} in o' that carries the read/flow dependences for a certain array reference r, and is placed above a certain loop l_{nrl} that belongs to $NRL(r)$. We group all such loops l_{rl} into a set $RL'(r)$ and permute them to the innermost of the array partitioning loop band to generate a new loop ordering $\hat{o}$ that belongs to O. If r is associated with flow dependences, o' introduces additional I/O modules for loading in the intermediate results, increasing the resource usage compared to $\hat{o}$. If r is associated with read/flow dependences, o' increases off-chip data communication, and could lead to a longer latency than $\hat{o}$ if the design is bound with data communication of r. Overall, this loop ordering o' is dominated by $\hat{o}$ in

both resource usage and design latency, which contradicts to the initial assumption that o' dominates loop orderings from O in performance. □

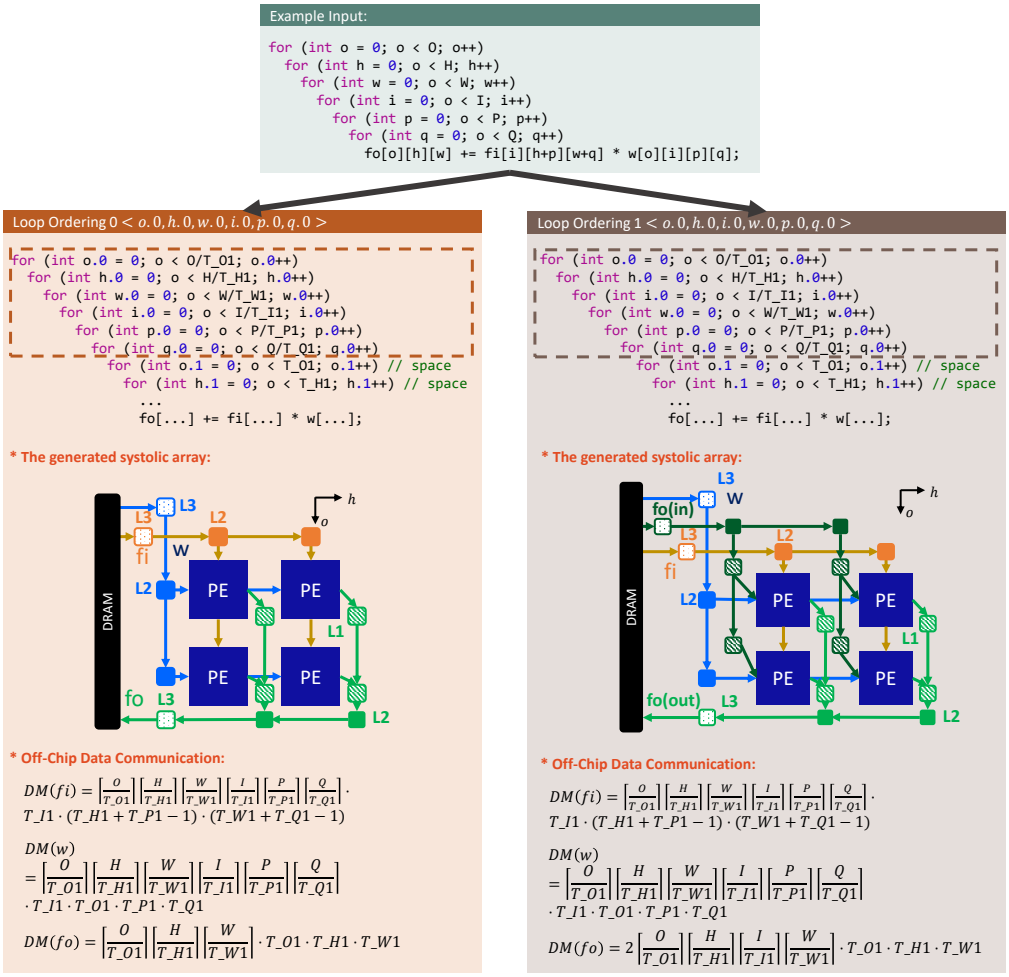


Figure 3.6: Example of different loop permutation orderings for CNN.

Algorithm 4 summarizes the pruning algorithm used by Odyssey. Given an input program, we examine all the array references in the program, identify the loops that carry the read/flow dependences, and enumerate unique loop orderings by placing these loops at the innermost position of the array partitioning loop band. The relative ordering between these loops and the ordering between the rest loops will not affect the performance. Table 3.5 lists all the unique loop permutation orderings generated for MM and CNN.

Algorithm 4: Loop permutation orderings generation.

Inputs : An input program p
Outputs : A list of unique loop permutation orderings O
Extract all the array references $Refs$ from the program p;
Extract all the read and flow dependences $Deps$ from the program p;
Extract all the loops L in the array partitioning loop band of the program p;
Initialize the loop permutation orderings $O \leftarrow \emptyset$;
for *each array reference r in* $Refs$ **do**
    Initialize the reuse loop list $RL(r) \leftarrow \emptyset$;
    for *each data dependence d in Deps* **do**
        if *r is associated with d* **then**
            for *each loop l in L* **do**
                if *d is carried on l* **then**
                    $RL(r) \leftarrow RL(r) \cup l$;
    Update the non-reuse loop list $NRL(r) \leftarrow L/RL(r)$;
    Append the loop ordering to the list $O \leftarrow O \cup < NRL(r), RL(r) >$;
Remove duplicate orderings in O;

Table 3.6: Unique systolic arrays for MM and CNN.

Application	Dataflows	Loop Permutation Orderings	#Designs
MM	[i], [j], [k], [i,j], [i,k], [j,k]	<[i,j],k>, <[j,k],i>, <[i,k], j>,	18
CNN	[o], [h], [w], [i], [o,h], [o,w], [o,i], [h,w], [h,i], [w,i]	<[o,h,w],[i,p,q]>, <[o,i,p,q],[h,w]>, <[i,h,w,p,q],o>	30

3.4.3 Putting It All Together

To summarize, given an input program, we use AutoSA to explore different dataflows and loop permutation orderings of the array partitioning loops and leave the tiling factors as tunable parameters to be explored by the auto-tuner. For each systolic array with a unique dataflow and loop permutation ordering, AutoSA generates a corresponding hardware description file that can be parsed by the auto-tuner to generate the parametric performance models for evaluation. For example, for MM, we generate 6 different dataflows with 3 different loop permutation orderings, leading to 18 different designs in total. For CNN, AutoSA generates 10 different dataflows and 3 different loop orderings. Table 3.6 summarizes all the unique designs identified by AutoSA. We will sweep these designs in sequence and use the auto-tuner to find the optimal tiling factors. All of these design factors construct a vast design space to explore. In the next section, we will discuss our auto-tuner to explore this design space

efficiently.

3.5 Searching the Design Space

In this section, we introduce the details of the proposed search method. We start with an overview of several representative search methods used by previous works in Section 3.5.1. We evaluated these methods on our search problem. Detailed evaluation results are presented in Section 3.6.2. Based on the evaluation results, we chose the evolutionary search which achieved the best performance in terms of both the quality of search results and convergence speed. Section 3.5.2 covers the details of the evolutionary search. Along with the evolutionary search, we implement a MP-based optimizer to provide high-quality seeds to evolutionary search which helps further improve the search performance. Section 3.5.3 covers the details of this optimizer.

3.5.1 Overview of Different Search Methods

Previous works on architecture exploration have adopted various search methods. We have selected a few representative approaches below and use them as baselines to evaluate the effectiveness of the proposed search method in the Odyssey framework.

- *Random search.* This method randomly samples the design space until it converges to a good result or meets the time limit. Random search is invariant to the characteristics of the design space and is effective in handling a small design space. Previous works such as Timeloop [PRS19] and AutoTVM [CZY18] have used it.

- *Exhaustive search with pruning.* Many previous works employed this method for design space exploration [PRS19, DKA19, YGL20, CKR20, LGW21]. This method is easy to implement and works well with proper pruning heuristics. However, it faces scalability issues when handling large design spaces. In addition, as discussed early in Section 3.1.2, selecting an effective pruning heuristic takes non-trivial efforts.

- *Simulated annealing.* Simulated annealing randomly samples the design space and

accepts the points that improve the objective. Besides, with a certain probability, it also accepts points that may worsen the objective as an attempt to avoid getting stuck in a local optimum. The probability to accept the worse points is determined by a parameter named *temperature* T which mimics the heating and cooling process of metals. Initially, we set T to a high value which gives a higher probability to accept worse solutions. As time goes by, the temperature gradually decreases, leading to a lower probability to accept worse solutions. Simulated annealing has been demonstrated to be effective in many auto-tuning frameworks such as AutoTVM [CZY18] and DSAGEN [WLD20].

- *Bayesian optimization.* Bayesian optimization is a statistical framework for global optimization on black-box functions. It works by building a cheap surrogate model for the unknown objective function, which is typically built using Gaussian Processes (GPs) [Ras03]. GPs are fit to previously searched design points and used to make predictions for the objective function values in the region that is not yet searched. In addition to the surrogate model, an acquisition function is updated at the same time for selecting the next query point that might improve the objective function. Bayesian optimization assumes that the evaluation cost of the real objective functions is expensive, therefore, it limits the evaluation time by using a surrogate model which can be cheaply computed and spends more time on the acquisition function for selecting the next query point to achieve high sample efficiency. Several previous works that targeted search tasks with high sample evaluation costs used this method [NSK19, RHA17]. For example, HyperMapper [NSK19] is a black-box optimization tool built on Bayesian optimization which is used for tuning the hardware designs generated by a DSL compiler.

- *OpenTuner [AKV14].* OpenTuner is an auto-tuning framework built on an ensemble of several efficient search techniques. Multiple search techniques are running at the same time. As the search progresses, techniques that produce better results are granted more samples to evaluate, while techniques with poor results are penalized with fewer samples for evaluation or disabled. OpenTuner includes a diverse set of search techniques such as several variants of evolutionary search, particle swarm optimization, and random

search. OpenTuner has been demonstrated effective in cases such as searching the GCC compilation flags and optimal schedules for Halide programs [AKV14].

- *Reinforcement learning.* Reinforcement learning (RL) is a machine learning algorithm which consists of an *agent* taking a sequence of *actions* on the *environment*. Depending on the actions that the agent makes, the environment generates either rewards or penalties to the agent. The goal to is maximize the total *rewards* in a limited number of actions. The agent implements a *policy* to determine the next action given all the past actions and feedback from the environment. RL has been used on many application domains including neural architecture search [ZL16] and hardware design space exploration [KJK20, JYS20]. For example, previous work ConfuciuX [KJK20] implemented a two-step search algorithm for tuning the dataflow architectures which employs RL as the first step to locate a good sub design space at the coarse-grained level and utilizes evolutionary search to perform a more fine-grained search later to find the best design.

- *Evolutionary search.* Evolutionary search [Hol92] is a generic meta-heuristic algorithm inspired by biological evolution, in which individuals of a population gradually improve themselves through a series of biological mechanisms such as mutation, crossover, and selection. In the context of hardware design space exploration, each design point is termed as an individual in the population. The feature vector that describes a design point is coined as the genome of the individual. Each design point is assigned a fitness score by its performance assessed by the real hardware measurement or the performance model. At each iteration, we select a group of individuals with the highest fitness scores in the population as the parents to produce the children in the next iteration (*selection*). These parents will breed new individuals with *mutation* and *crossover* operations. In the next iteration, we filter out the low-fitted individuals and repeat the same reproduction process until we find satisfactory solutions. Evolutionary search is light-weighted in implementation and invariant to the characteristics of design space. It is used in many recent auto-tuning frameworks such as OpenTuner [AKV14],

GAMMA [KK20], Ansor [ZJS20], and Tensor Comprehension [VZT19].

3.5.2 Evolutionary Search

In the Odyssey framework, we select the evolutionary search as the backbone method. Based on our evaluation, evolutionary search achieved the best performance in terms of both the quality of search results and convergence speed compared to other baseline methods. Details of the evaluation results are covered in Section 3.6.2. We present the implementation details of the evolutionary search in the Odyssey framework in this section.

Encoding scheme AutoSA generates designs with different dataflows and loop permutations, and leaves the tiling factors to the auto-tuner to explore. Figure 3.7 shows the encoding example of the MM program. The encoded genome includes the problem size and the tiling factors used for each compilation pass including the array partitioning, latency hiding, and SIMD vectorization. At present, we leave the communication management to be handled automatically by the compiler.

Mutation Next, we perform the tiling factor mutation. As depicted in Figure 3.7, we first compute the loop bounds with the current tiling factors. The mutation process operates on the values of the loop bounds. We implement two types of mutation methods: *factorization-based* mutation and *random* mutation.

1) Factorization-based mutation: We randomly choose one loop l_1, divide it by a random divisor α ($\alpha | l_1$), and multiply this value to another loop l_2 that is derived from the same loop as l_1. In the example shown in Figure 3.7, we choose the loop $k.2$ and divide it by 2, then we randomly choose between the two other tiled loops $k.0$ and $k.1$ that are derived from the same original loop as $k.2$. In this example, we choose the loop $k.1$ and multiply its current value with the factor 2. The factorization-based mutation keeps the product of the tiled loop sizes unchanged, guaranteeing the muted program is always valid.

Since the factorization-based mutation always chooses the divisors of the loop bounds,

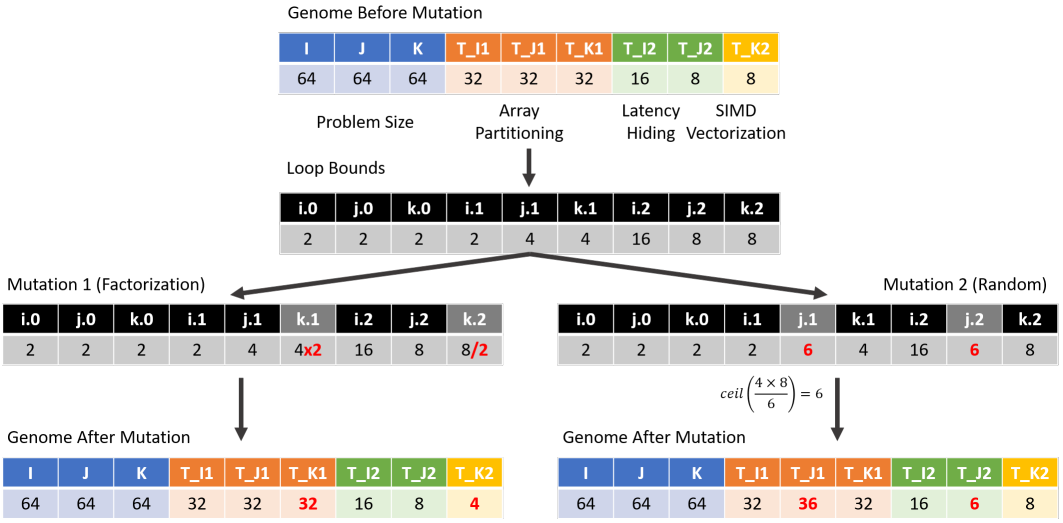


Figure 3.7: Example of different mutation methods.

it fails to cover the non-divisor tiling factors. While such a restriction is common in many previous works such as CoSA [HKD21], Timeloop [PRS19], dMazeRunner [DKA19] and Interstellar [YGL20], it could result in a reduced design space with inferior performance. Table 3.7 compares the best systolic arrays identified by two search methods for a $1024 \times 1024 \times 1024$ MM problem. The first method uses the factorization-based mutation that only considers divisor tiling factors. The second method employs the hybrid mutation method which we will introduce soon that considers non-divisor tiling factors. As shown in the table, the first design found with the factorization-based mutation has a 39% performance gap compared to the second design. The use of non-divisor tiling factors (e.g., $T_I1 = 129$, $T_J1 = 130$) helps locate a better design that fully utilizes the DSPs, while the divisor-only design only uses 60% of the DSPs due to the limited choices of tiling factors. This result reveals the necessity of taking non-divisors into consideration to achieve high performance. To accommodate for the non-divisor tiling factors, we introduce the second mutation scheme, *random mutation*.

Table 3.7: Best solutions found by two mutation methods.

Mutation Methods	Throughput	BRAM	DSP	T_I1	T_J1	T_K1	T_I2	T_J2	T_K2
Factorization	0.61×	48%	60%	64	128	128	16	4	8
Factorization + Random	1.00×	99%	100%	129	130	64	3	13	4

2) Random mutation: We randomly select one loop l_1 and mutate this loop by changing the loop bound to a random value $s \in [1, l_1]$. Next, we select another corresponding loop l_2 and change its loop bound to a new value s' computed by the following equation:

$$s' = ceil(\frac{l_1 \times l_2}{s}) \tag{3.6}$$

We show one example of random mutation in Figure 3.7. In the right example, we choose the loop $j.2$ and change its original loop size from 8 to 6. Then we choose the other loop $j.1$ to mutate. We compute the new loop bound for $j.1$ as $ceil(4 \times 8/6) = 6$. As a result, the original array partitioning tiling factor T_J1 is changed from 32 to 36, which is a non-divisor of the original problem size $J = 64$. The random mutation method adjusts two loops at the same time as an effort to keep the product of these two loops with minimal change. The use of `ceil()` function guarantees the new product is no less than the original product so that the mutated program is always legal.

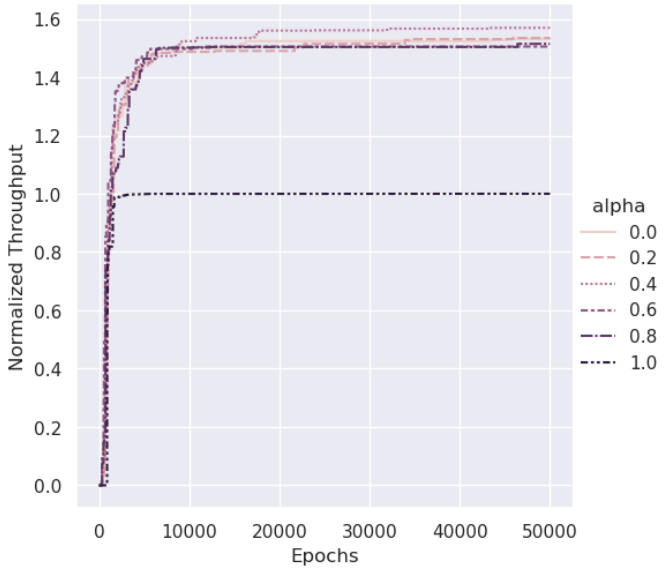


Figure 3.8: Tuning performance comparison with different hyper-parameters α.

When performing the mutation, we assign a certain probability α to execute the factorization-based mutation and assign the probability $1 - \alpha$ to the random mutation. We conduct a grid search to find the best parameter α that maximizes the search performance. Figure 3.8 presents the search results on the same MM problem with a set of different α values. We see

that the evolutionary search converges fastest and to the best results when $\alpha = 0.4$. In our tuning experiments, we set α to 0.4 by default.

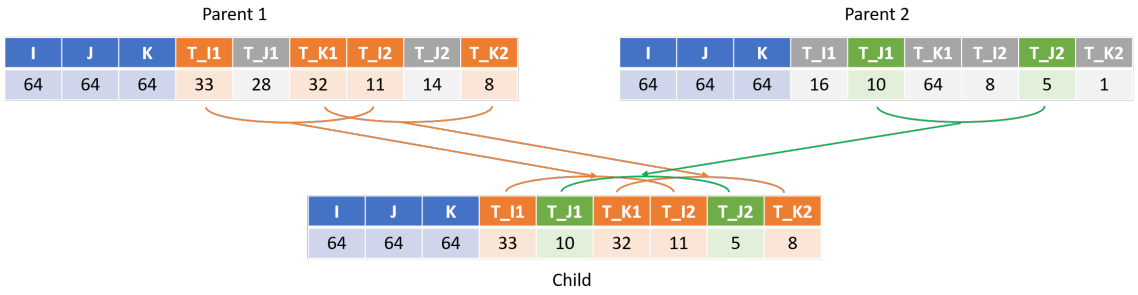


Figure 3.9: Example of crossover in evolutionary search.

Crossover A crossover operation exchanges the genomes of two individuals. To guarantee the validness of the offspring, we exchange the tiling factors associated with the same original loop together, as seen in Figure 3.9.

3.5.3 Mathematical Programming

Previous works like ConfuciuX [KJK20] have revealed that a high-quality initial population can improve the convergence of evolutionary search. Meanwhile, previous works such as [LXS21, LSV19, HKD21] have uncovered that formulating the design space with mathematical formulation and using solvers to obtain the optimal solutions can obtain satisfactory search results. Therefore, to further improve the performance of evolutionary search, we implement a mathematical programming (MP)-based optimizer to produce high-quality designs serving as the initial population to the evolutionary search. We introduce the problem formulation of the optimization problem as follows.

3.5.3.1 Constraints

Memory resource A valid hardware design should not overuse the available memory and computation resource on-chip. Specifically, for FPGA designs, we consider the BRAM and

DSP usage. We model the memory usage as:

$$U_{mem} = \sum_{m \in Modules} U_{mem}(m) \times N_m \tag{3.7}$$

For each type of hardware modules m (e.g., PE, L1 I/O module), we compute the memory usage of each module $U_{mem}(m)$, multiplied by the total number of such modules N_m. The memory usage of module m is computed as:

$$U_{mem}(m) = \sum_{a \in Arrays} \lceil \frac{DW(a) \times Bank(a)}{18} \rceil \lceil \frac{N(a)}{Bank(a) \times 1024} \rceil \tag{3.8}$$

We sum the BRAMs used by all the arrays inside the hardware module. On Xilinx FPGA, we configure the basic BRAM unit with a datawidth of 18 bits and a depth of 1024 words. We compute the datawidth of the on-chip buffer for storing the array a as the array data width $DW(a)$ multiplied by the number of memory banks $Bank(a)$. We calculate the depth of the buffer as the total number of array elements $N(a)$ divided by the number of memory banks $Bank(a)$.

Finally, the total number of the BRAM units should not surpass the maximal available memory resource on-chip $Mem_{available}$.

$$U_{mem} \leq Mem_{available} \tag{3.9}$$

Computation resource Similar to the memory resource, the total usage of computation resources (i.e., DSP) should not go beyond the limit on-chip.

$$U_{DSP} \leq DSP_{available} \tag{3.10}$$

We compute the DSP usage as:

$$U_{DSP} = \sum_{m \in Modules} U_{DSP}(m) \times N_m \tag{3.11}$$

$$U_{DSP}(m) = \sum_{op \in Compute_Ops} SIMD(op) \times U_{DSP}(op) \tag{3.12}$$

We compute the total amount of DSPs as a sum of DSPs consumed by all the hardware modules. The per module DSP usage $U_{DSP}(m)$ is computed as a sum of the products of SIMD factor $SIMD(op)$ of each computation operator *op* and the number of DSPs consumed by each single SIMD lane $U_{DSP}(op)$ over all the computation operators in the hardware module. We maintain an internal database for the per operator DSP usage $U_{DSP}(op)$.

Architecture constraints In addition to the resource constraints, there are two other additional constraints we need to consider.

Latency hiding. To achieve 100% of the pipeline utilization with latency hiding, we require the product of the loop bounds of all the latency hiding loops $T(l)$ to be no less than the critical path latency of the computation statements $L_{compute_critical}$.

$$\prod_{l \in latency_hiding_loops} T(l) \geq L_{compute_critical} \tag{3.13}$$

We maintain an internal database storing the latency of different computation statements and accumulate them along the critical path to estimate $L_{compute_critical}$.

SIMD vectorization. We require the SIMD factor to be an even number to minimize the hardware overheads.

3.5.3.2 Objective Function

Given the high complexity of the hardware designs, it is usually hard to have a close-formed performance model suitable for the solvers as the objective function. Instead, previous works chose different high-order functions that impact the performance as the objective functions,

such as the off-chip data communication [LXS21, CKR20] and resource utilization [HKD21]. The recent work CoSA [HKD21] has proposed using a linear combination of three high-order functions including the memory resource utilization, ideal compute latency, and the on-chip data traffic. However, the coefficients of these factors need to be specified by users which adds unnecessary complexity to the tuning process. We have conducted an experiment on evaluating the effectiveness of several objective functions.

Objective 1: Computation resource We use the total DSP usage U_{DSP} as the optimization target. The heuristic is that a design with higher performance should use more DSPs.

$$Obj1 : min(-U_{DSP}) \tag{3.14}$$

Objective 2: Off-chip communication For communication-bound applications, reducing the off-chip communication could improve the design performance. We aggregate the off-chip data movement of all the arrays in the program as the objective function.

$$Obj2 : min \sum_{a \in Arrays} DM(a) \tag{3.15}$$

Objective 3: Off-chip communication - computation resource This objective function takes both computation and communication into consideration. Ideally, we would like to maximize the computation resource and reduce the off-chip communication.

$$Obj3 : min(\sum_{a \in Arrays} DM(a) - U_{DSP}) \tag{3.16}$$

In our experiment, we apply the off-the-shelf solver (AMPL [Gay15] with Ipopt [COI21]) to implement the optimization problem and find the best tiling factors with different optimization targets. To stabilize the optimization process, we normalize all the metrics mentioned above. We normalize the DSP usage against the total available DSP resource on-chip. For data

movement, we compute a baseline by adding up the total amount of elements in each array. For arrays that are associated with flow dependences (e.g., the output matrix C in matrix multiplication), we multiply the total amount of elements by 2 in consideration of both inbound and outbound traffic. In all the optimization problems, we normalize the data movement $DM(a)$ against this baseline.

The best solution obtained from the solver is then fed to the evolutionary search as the initial population. Figure 3.10 shows the comparison results between these different optimization targets. Table 3.8 compares the performance of designs found by solvers.

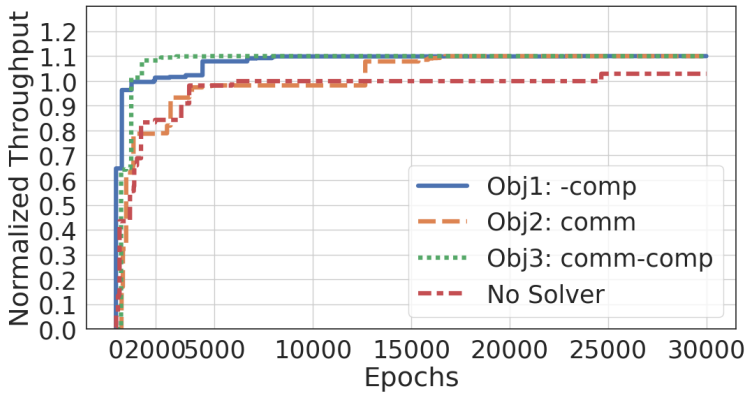


Figure 3.10: Tuning performance comparison of different objective targets.

As seen in the figure, all three objective functions help reduce the convergence time and yield better results compared to the evolutionary search-only method (annotated as *No Solver*). Specifically, Obj3 helps significantly reduce the convergence time. With Obj3, the auto-tuner locates a better design than the baseline (No Solver) within 2000 epochs. In Odyssey, we use the Obj3 as the optimization target of the solver. The implications from this experiment are multi-fold.

First, MP-based methods are insufficient to find optimal designs. As demonstrated in Table 3.8, the best design identified by the MP-based approach is 1.5× slower than the design discovered by Odyssey with a hybrid search method that combines both mathematical programming and evolutionary search.

Second, considering either computation or communication alone is insufficient for performance tuning as a high-quality design is determined by both factors in an intertwined

Table 3.8: Comparison of the best solutions found by MP-based methods and Odyssey framework.

Methods	Opt. Target	Latency	Off-Chip Data Movement	DSP
Mathematical Programming	Obj1:-comp	1.5×	1.7×	6.3×
Mathematical Programming	Obj2:comm	8.6×	1.0×	1.0×
Mathematical Programming	Obj3:comm-comp	2.5×	1.0×	4.0×
Odyssey	comm-comp	1.0×	4.9×	6.7×

manner. In Table 3.8, the best design located by our auto-tuner introduces 4.9× more off-chip data communication than the design with the least off-chip data movement discovered by the MP-based methods. This indicates that previous works such as Marvel [CKR20] and [CHW20] that chose to decouple the design space by first identifying the subspace that minimizes the off-chip data communication will leave out the optimal designs and achieve inferior performance.

Lastly, our search results echo the findings from ConfuciuX [KJK20] which demonstrated that a high-quality initial population can improve the performance of evolutionary search. Based on the experiment above, in Odyssey, we use the Obj3 as the optimization target of the solver.

3.6 Evaluation Results

In this section, we evaluate the performance of the Odyssey framework. We first validate the performance models from Odyssey against the real hardware. Then, we compare the proposed auto-tuning method with several other search methods to assess its efficiency. Lastly, we use the Odyssey framework to investigate the architecture trade-offs of systolic arrays on two applications, MM and CNN.

Table 3.9: Parameters of validation workloads.

Workload	Parameters	#Designs
MM	[I,J,K]: [64,64,64]	18
CNN	[I,O,H,W,P,Q]: [16,16,16,16,3,3]	30

3.6.1 Performance Model Validation

3.6.1.1 Approaches

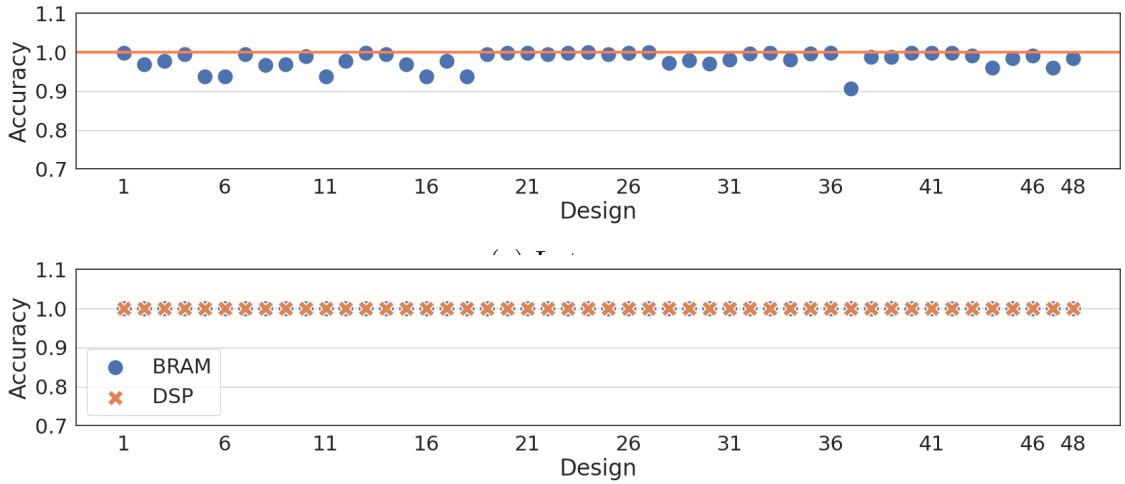
Workloads We evaluate the accuracy of the hardware models derived from Odyssey on two important workloads in HPC and deep learning, MM and CNN. Table 3.9 details the parameters for the chosen workloads. We choose a relatively small problem size in consideration of the long RTL simulation time. All the experiments in this section target Xilinx Alveo U250 FPGA.

Baselines We measure the design latency by RTL simulation as the baseline for latency models. We use the reported resource numbers from the HLS synthesis reports as the baseline for resource models.

Designs We validate all 18 and 30 different designs generated for MM and CNN by AutoSA, as listed in Table 3.6. To select the tiling factors, we randomly sample the design space. We compare the performance numbers estimated by the models derived from Odyssey against the numbers measured by RTL simulation and HLS synthesis.

3.6.1.2 Validation Results

Figure 3.11 compares the model-predicted performance numbers against the measured numbers for all the designs. The performance models generated by Odyssey are highly accurate, with estimation error rates of 1.99%, 0%, and 0% for latency, BRAM, and DSP, respectively.



(b) BRAM and DSP

Figure 3.11: Performance model validation results.

3.6.2 Auto-Tuning Evaluation

3.6.2.1 Approaches

Workloads We evaluate the performance of the proposed auto-tuning method using a $1024 \times 1024 \times 1024$ matrix multiplication.

Baselines We employ the following methods as baselines.

- *Random search.* We randomly sample the design space and update the best solutions. We also maintain the history of searched solutions to avoid duplicate sampling.
- *Exhaustive search with pruning.* We extend the random search by pruning the design samples based on the DSP utilization. Designs with a DSP usage below the threshold will be pruned without further evaluation. This helps save the evaluation time on pruned samples. We set the DSP pruning threshold at 25%, as the smallest design among the 18 designs only uses 30% of DSPs.
- *Simulated annealing.* We use the Python package [Sci21] as the baseline. We set the temperature T at 200 based on a grid search which indicates this number helps maximize

the search performance. We implement a customized step-taking function to explore new points utilizing the proposed hybrid mutation method for the evolutionary search.

- *Bayesian optimization.* In our experiment, we employ the Python package [Nog21] as the baseline. We keep the default settings of the Python package.

- *OpenTuner [AKV14].* We use the latest release of of OpenTuner from its Github repository [AKV21].

- *Reinforcement learning.* We use the open-source implementation from ConfuciuX [KJK20] as the RL baseline [KJK21]. ConfuciuX applied a 3-layer multi-layer perceptron (MLP) neural network for the policy network.

Designs We compare our tuning methods against the baselines on all 18 different designs generated for MM by AutoSA. AutoSA generates 6 different dataflows and 3 unique loop permutation orderings for MM. We annotate those designs using the format $[0, 0]$. The first number is the index of dataflows in Table 3.6. The second number is the index of loop permutation ordering listed in Table 3.6. For example, the design $[0, 0]$ refers to the design with the dataflow $[i]$ and the loop ordering $< [i, j], k >$.

Method For each systolic array design, we run the search method for 5 minutes and repeat it 3 times. The final results are averaged from the 3 runs. All the search methods are executed with a single CPU thread. RL baseline uses Pytorch which implements multi-threading during the training of MLP. All experiments are executed on a workstation with Intel Xeon E5-2680 v4 CPU.

3.6.2.2 Results

Search results quality Table 3.10 summarizes the best performance and convergence time[3] of all the search methods on the 18 systolic array designs. The last column of the table

[3]Convergence time is measured as the last time when the search results are updated.

Table 3.10: Comparison of latency and search time between different search methods (search time is represented in (mins:secs)).

		Random Search		Exhaustive w/ Pruning		Simulated Annealing		Bayesian Optimization		OpenTuner		RL		Odyssey		Optimal
ID	Designs	Latency	Time	Latency	Time	Latency	Time	Latency	Time	Latency	Time	Latency	Time	Latency	Time	Latency
0	[0,0]	1,197,046	4:05	965,001	4:08	804,426	1:50	7,753,699	3:33	1,066,686	0:42	1,490,757	0:39	**804,426**	0:54	804,426
1	[0,1]	1,707,095	3:27	1,597,922	1:00	1,333,317	4:27	15,762,801	0:18	1,344,688	1:32	1,833,651	3:18	**1,325,729**	0:04	1,325,049
2	[0,2]	1,639,325	1:05	1,867,522	3:15	**1,323,336**	0:09	12,838,521	0:18	1,329,249	1:18	1,606,922	3:02	1,323,580	0:01	1,322,712
3	[1,0]	888,202	0:08	1,224,476	0:21	**819,046**	1:01	19,412,852	2:38	824,932	1:27	1,235,176	1:59	823,825	0:19	819,046
4	[1,1]	2,846,557	1:30	3,216,149	1:22	**1,745,029**	0:04	35,023,219	1:34	2,135,296	1:41	3,484,467	4:11	1,745,833	0:02	1,745,029
5	[1,2]	3,197,700	4:38	3,522,388	0:16	3,189,578	0:47	35,023,219	1:32	2,320,752	1:42	3,949,897	3:35	**2,064,033**	0:36	2,029,771
6	[2,0]	2,126,674	4:26	2,205,084	2:53	2,108,802	0:14	3,923,625	1:19	2,105,474	1:35	2,134,458	2:25	**2,105,474**	0:01	2,105,474
7	[2,1]	2,119,212	3:09	2,154,008	0:36	2,109,007	0:01	4,020,501	3:07	**2,105,474**	1:01	2,109,826	4:39	2,107,394	0:01	2,105,474
8	[2,2]	2,131,838	2:11	2,114,834	4:02	2,111,234	0:05	5,432,099	0:22	**2,105,474**	1:15	2,139,578	4:42	2,110,722	0:01	2,105,474
9	[3,0]	757,148	3:19	756,241	0:23	663,706	2:30	13,293,868	1:40	684,545	1:20	1,127,288	0:37	**660,353**	0:09	660,353
10	[3,1]	1,132,476	0:24	1,240,020	2:47	849,804	1:34	13,315,008	1:40	879,713	1:27	2,643,599	0:06	**828,046**	1:37	828,046
11	[3,2]	1,136,000	0:01	1,386,002	1:51	1,007,223	1:50	13,315,008	1:40	1,109,312	1:04	2,798,976	4:17	**963,600**	2:27	963,600
12	[4,0]	938,374	2:01	1,207,362	2:53	824,420	1:37	3,930,315	1:22	774,156	1:18	1,211,979	3:56	**728,241**	0:04	728,241
13	[4,1]	821,946	0:27	1,284,980	4:07	750,513	3:03	4,194,911	0:23	1,070,048	1:16	957,565	2:04	**725,697**	2:46	725,697
14	[4,2]	840,157	3:22	1,569,976	2:06	858,585	1:56	3,930,315	1:21	968,809	1:23	1,115,874	2:28	**738,781**	1:06	729,472
15	[5,0]	912,444	0:03	1,230,436	2:20	771,295	4:09	3,930,314	1:22	789,690	0:53	1,177,852	0:01	**722,353**	0:01	683,521
16	[5,1]	1,032,695	1:59	1,292,530	1:19	858,637	1:31	3,930,314	1:24	792,237	1:04	892,932	0:24	**744,993**	0:33	729,472
17	[5,2]	822,396	0:10	1,236,155	0:42	759,169	0:47	3,930,314	1:22	1,063,150	1:05	1,182,393	4:29	**728,241**	0:00	728,241

is the optimal performance discovered by a modified exhaustive search[4]. The best result found for each design is marked in green color. Figure 3.12 compares the normalized throughput against the optimal results found by exhaustive search from all the tuning methods. Odyssey found design configurations with best performance on 13 designs out of the total 18 designs. For the remaining 5 designs, the performance gap is within 1% of the best performance identified by other baselines (OpenTuner and simulated annealing). Overall, Odyssey locates designs that achieve more than 95% of the optimal performance.

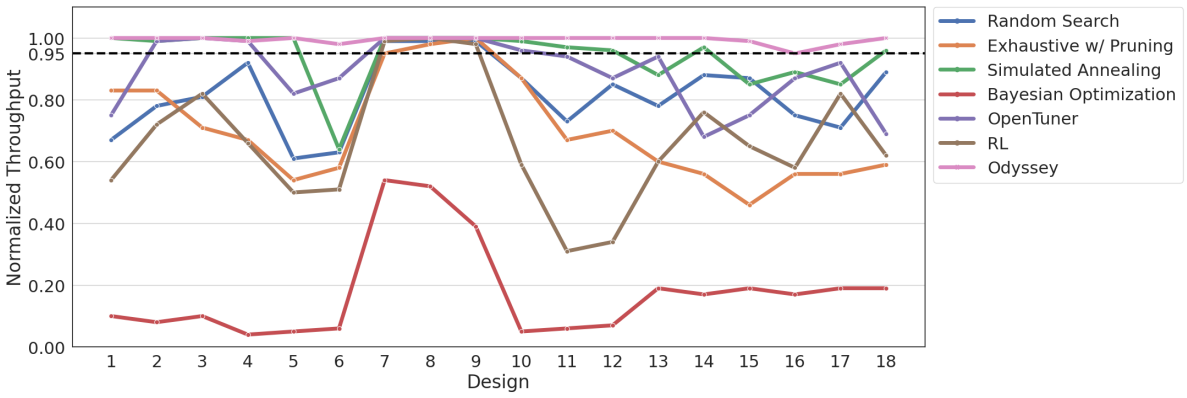


Figure 3.12: Comparison of the best designs identified by different tuning methods.

[4]As exhaustive search takes too long and is infeasible for this task, we apply the DSP pruning and adjust the DSP pruning threshold dynamically for each design based on the best design found by other methods. The final results are collected by running an exhaustive search with pruning until it completes.

Sample efficiency In addition to the high-quality search results, Odyssey achieves high sample efficiency. Figure 3.13 compares the convergence traces of all the tuning methods on design $[3, 0]$, which achieves the best performance among all the designs. As shown by the figure, Odyssey converges to a good design achieving 93% of the optimal performance after evaluating 3000 design samples. Simulated annealing achieves the second best performance which finds a design achieving 66% of the optimal performance.

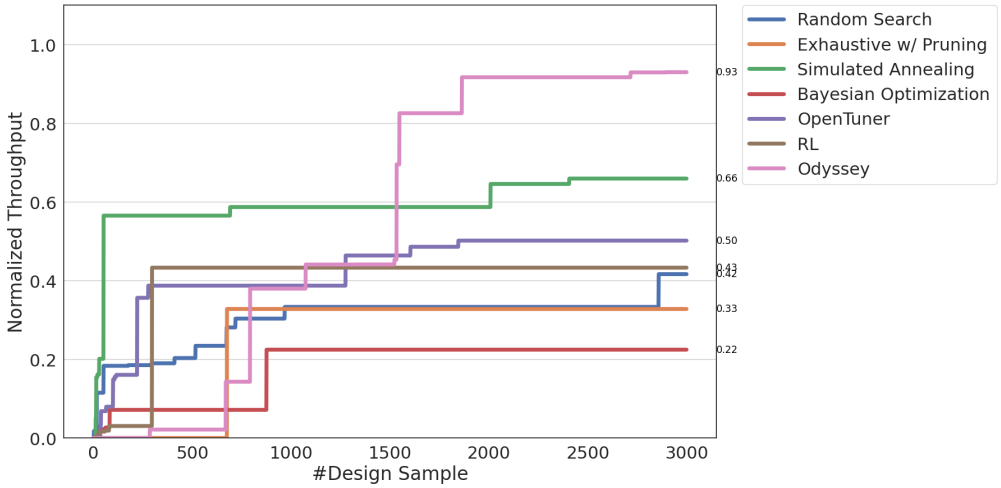


Figure 3.13: Comparison of sample efficiency of different tuning methods.

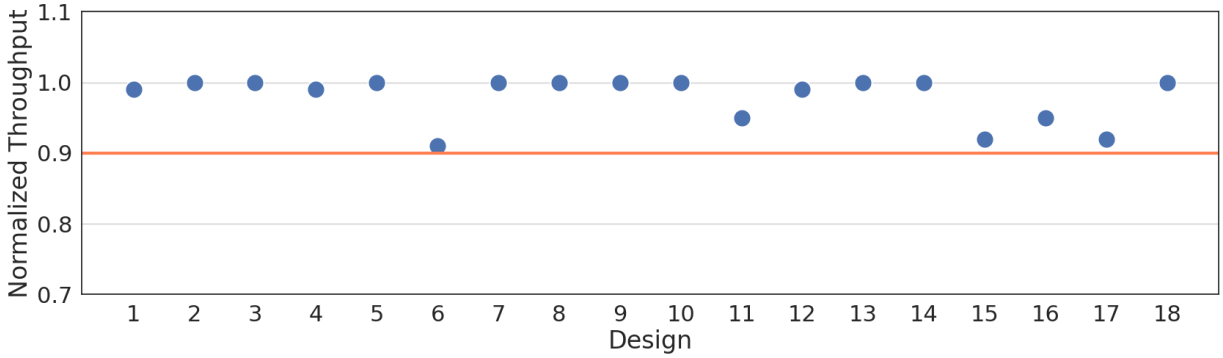


Figure 3.14: Best designs found in 5 seconds by Odyssey for all different designs of MM.

Search time Lastly, Figure 3.14 compares the best designs located by Odyssey within 5 seconds. Odyssey identifies design configurations achieving over 90% of the optimal performance on all the systolic array designs within 5 seconds.

3.6.3 Application 1: MM

In this section, we perform a detailed architecture comparison of different systolic arrays for matrix multiplication using the proposed search method.

3.6.3.1 Metric 1: Latency/Throughput

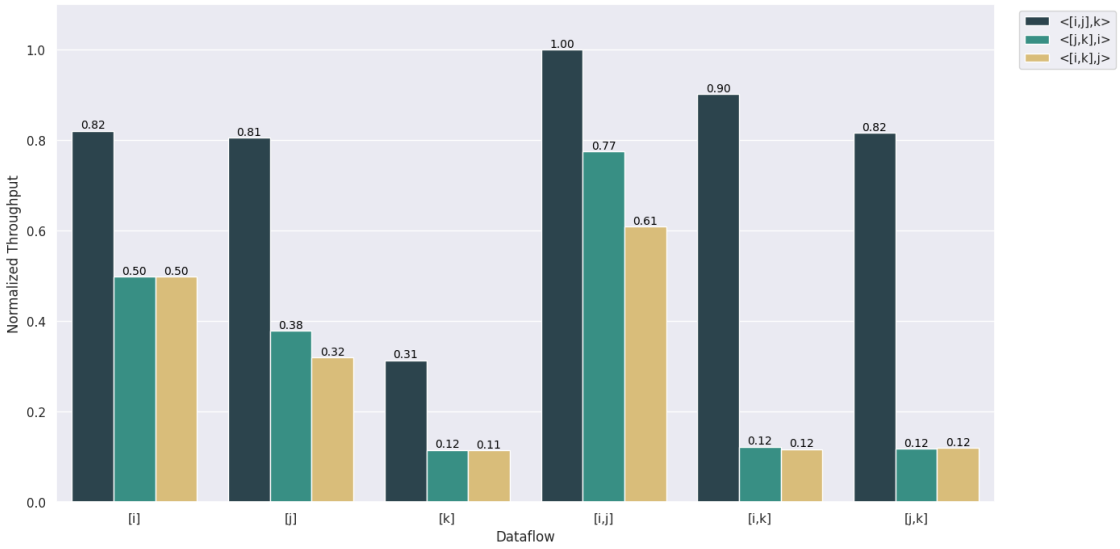


Figure 3.15: Throughput comparison of different designs for MM.

Figure 3.15 shows the normalized throughput of the best configuration discovered for each systolic array. We make the following observations from this figure.

Loop permutation The loop permutation ordering $< [i, j], k >$ dominates the performance across different dataflows. The major reason for such a performance gap is the extra memory usage introduced by the two other loop orderings. As discussed earlier in Section 3.4.2, the loop orderings $< [j, k], i >$ and $< [i, k], j >$ introduce extra I/O modules for loading in the intermediate results, leading to a higher memory usage than the loop ordering $< [i, j], k >$. Table 3.11 shows the resource usage breakdown for three designs with the same dataflow $[i]$ and different loop permutation orderings. As seen in the table, designs with loop orderings $< [j, k], i >$ and $< [i, k], j >$ allocate more BRAMs to I/O modules for matrix C, which limits the size of array ($\#PE$), leading to a worse performance than the ordering $< [i, j], k >$. Note

Table 3.11: Resource breakdown of three designs for MM with the same dataflow $[i]$.

Loop Orderings	Latency	#PE	BRAM Usage				
			Total	I/O Module (A)	I/O Module (B)	I/O Module (C)	PE
<[i,j],k>	1.00×	1.66×	3,621	76%	1%	19%	5%
<[j,k],i>	1.65×	1.00×	3,519	47%	3%	47%	3%
<[i,k],j>	1.64×	1.05×	3,594	48%	1%	48%	3%

that all three designs use a small amount of memory in I/O modules for matrix B as the data are reused along the spatial dimensions, thus requiring the minimal storage for on-chip data.

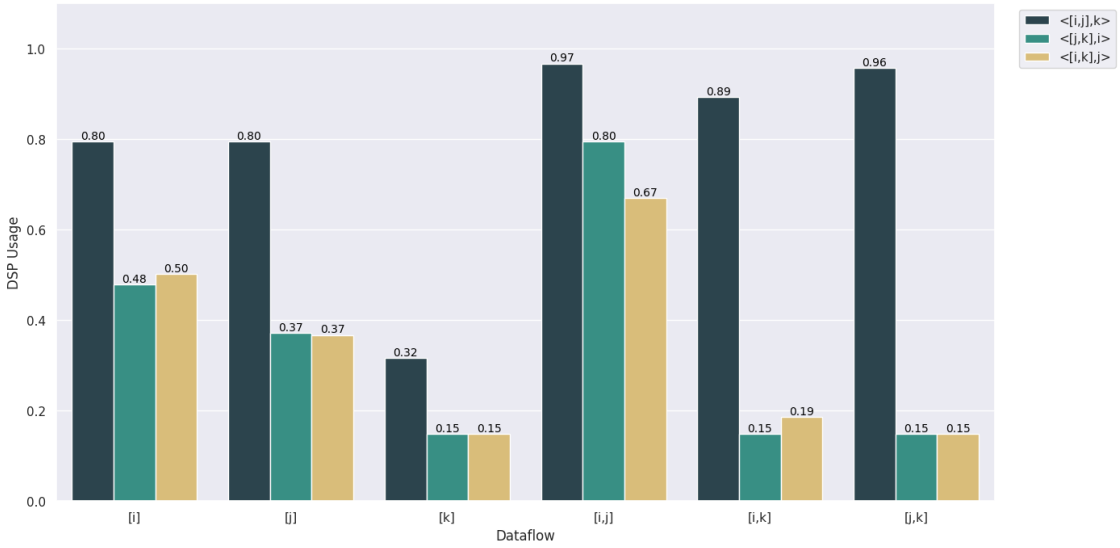


Figure 3.16: DSP usage of different latency-optimal designs for MM.

Dataflow Among designs with the same loop ordering, the dataflow $[i, j]$ achieves the best performance because the dataflow $[i, j]$ exploits the most degrees of parallelism from the application. For the matrix multiplication, we vectorize the loop k for all the designs which serve as the first degree of parallelism. The space dimensions of the systolic array exploits the rest of the degrees of parallelism. 1D systolic arrays can exploit at most one more degree of parallelism. In comparison, the dataflow $[i, j]$ exploits two degrees of parallelism (i and j), adding up to three degrees of parallelism in total. The two other 2D systolic arrays include k in the space dimensions such that they could extract at most one more degree of parallelism similar to the 1D arrays. More dimensions of parallelism add more degrees of freedom to the design space, covering more profitable design points. As demonstrated in Figure 3.16,

the dataflow $[i, j]$ utilizes the most DSPs, achieving the highest throughput among all the designs.

3.6.3.2 Metric 2: Energy

The Odyssey framework can be easily extended to support different performance metrics. Here, we show one example of energy optimization. We model the energy consumed by each design following the same strategy used in Eyeriss [CES16]. The energy of one design E is modeled as:

$$E = E(mem_{off_chip}) + E(mem_{on_chip}) + E(compute) + E(inter_array) \tag{3.17}$$

The total energy cost of one design is computed as the sum of four components: the off-chip memory access energy $E(mem_{off_chip})$, the on-chip memory access energy $E(mem_{on_chip})$, the computation energy $E(compute)$, and the inter-array data communication energy $E(inter_array)$.

Off-chip memory access energy $E(mem_{off_chip})$ is estimated as the product of total amount of off-chip memory accesses for each array reference ref ($\#Acc(ref)$) and the unit off-chip memory access cost e_{off_chip}:

$$E(mem_{off_chip}) = \sum_{ref \in ArrayRefs} \#Acc(ref) \times e_{off_chip} \tag{3.18}$$

Similarly, on-chip memory access energy $E(mem_{on_chip})$ is estimated as the product of the total amount of on-chip memory accesses at each memory level l (e.g., registers (Reg), BRAM) for each array reference ref ($\#Acc(ref, l)$) and the unit on-chip memory access cost at each memory level l ($e_{on_chip}(l)$):

$$E(mem_{on_chip}) = \sum_{ref \in ArrayRefs} \Big(\sum_{l \in \{Reg, BRAM\}} \#Acc(ref, l) \times e_{on_chip}(l) \Big) \tag{3.19}$$

The computation energy $E(compute)$ is estimated as the total number of compute operations $\#Ops$ multiplied by the unit energy cost per operation e_{comp}:

$$E(compute) = \#Ops \times e_{comp} \tag{3.20}$$

The last component of Equation 3.17, the inter-array energy $E(inter_array)$, measures the energy spent on transferring data through the on-chip interconnects. For each array reference, we count the amount of hardware modules (hops) that the array reference ref has passed through ($\#Hop(ref)$), multiply it by the per hop data transfer energy e_{hop} and the total number of array accesses $\#Acc(ref)$. The final inter-array energy is estimated by summing the inter-array data transfer energy for each array reference.

$$E(inter_array) = \sum_{ref \in ArrayRefs} \#Hop(ref) \times e_{hop} \times \#Acc(ref) \tag{3.21}$$

In the systolic array architecture, each array element is first loaded from the off-chip memory, transferred through multiple levels of I/O modules, and reused across PEs. We add up the total number of hardware modules along this path for each array reference. For example, Figure 3.1 shows two systolic arrays with dataflow $[i]$ and $[k]$ for MM. For the design on the left, each element of array A is first loaded from the off-chip memory by the L2 I/O module, and then passed through L1 I/O modules, finally sent to PEs. We count the number of hops for the data that reach the last PE as 4, including one L2 I/O module, two L1 I/O modules, and one PE.

When estimating the energy cost, we use the numbers from Eyeriss [CES16], which are extracted from a commercial 65nm process. Table 3.12 illustrates the normalized energy cost relative to a MAC operation[5]. Note that the energy model can be easily replaced with different models targeting different hardware and technology nodes. The comparison here focuses on the relative comparison between different designs rather than the absolute values.

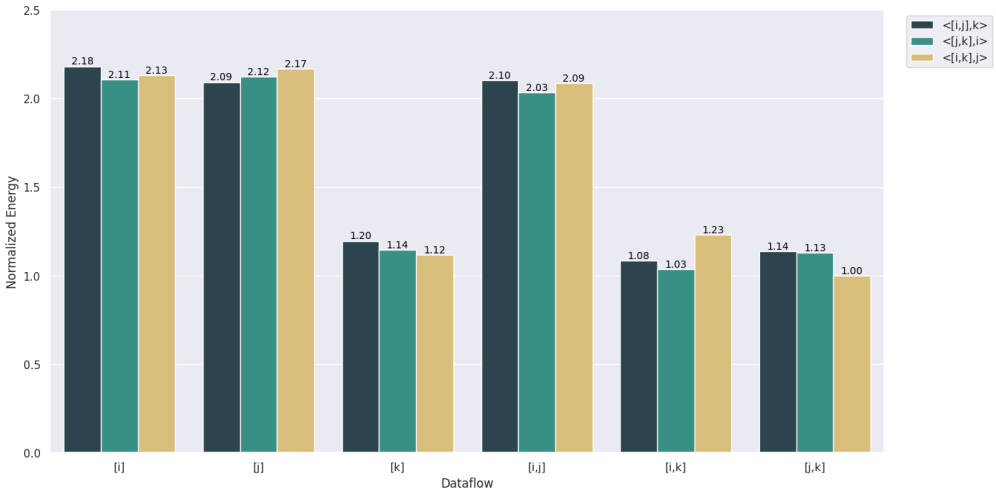
We replace the performance evaluation functions in Odyssey with the energy models explained above. All the components in the energy models can be computed automatically

[5]We estimate the unit memory access energy cost for BRAMs using the global buffer access cost from Eyeriss [CES16]. Both are implemented by SRAMs.

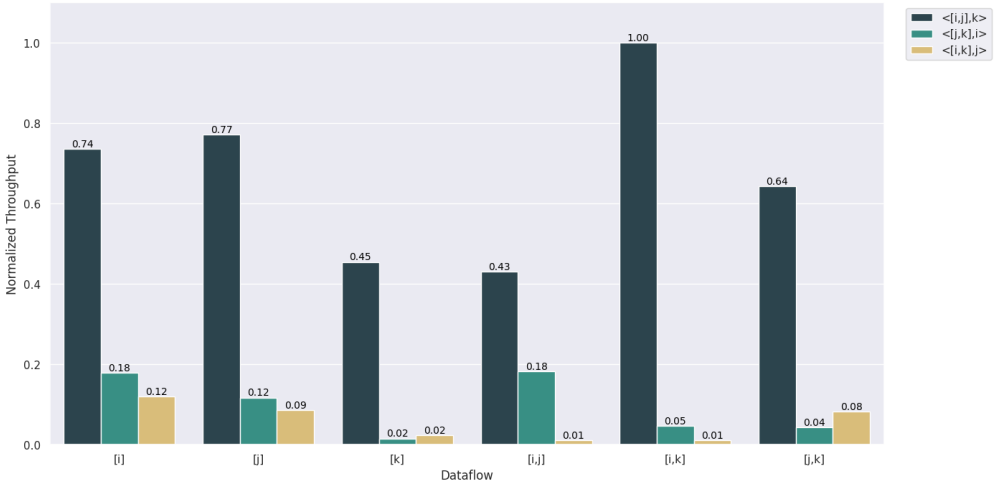
Table 3.12: Normalized energy cost relative to a MAC operation extracted from a commercial 65nm process [CES16].

Item	e_{off_chip}	e_{BRAM}	e_{Reg}	e_{comp}	e_{hop}
Normalized Energy	200	6	1	1	2

with the help of the design description file generated by AutoSA. Figure 3.17 presents the search results on all 18 different designs for MM. We draw the following conclusions from this figure.



(a) Energy



(b) Throughput

Figure 3.17: Performance comparison of different energy-optimal designs for MM.

Loop permutation For the designs with the same dataflow, we observe all the permutation orderings achieve the similar minimal energy. However, as seen in Figure 3.17b, for each dataflow with a similar energy consumption, the loop ordering $< [i, j], k >$ achieves the highest throughput. Table 3.13 compares the first three designs with the dataflow $[i]$ and different loop orderings. As shown in the table, the loop ordering $< [i, j], k >$ saves more BRAMs than the two other designs with the elimination of the I/O modules for loading in the intermediate results. Therefore, the design achieves a higher DSP usage and throughput with a similar energy cost.

Table 3.13: Performance comparison of three energy-optimal designs for MM with the same dataflow $[i]$.

Loop Orderings	Latency	#PE	DSP Usage	BRAM Usage				
				Total	I/O Module (A)	I/O Module (B)	I/O Module (C)	PE
$< [i, j], k >$	1.00×	1.61×	4,120	2,282	72%	1%	18%	9%
$< [j, k], i >$	4.12×	1.00×	2,560	3,600	28%	1%	57%	14%
$< [i, k], j >$	6.13×	1.08×	2,760	3,758	29%	1%	59%	11%

Dataflow We see that dataflows $[k]$, $[i, k]$, and $[i, k]$ achieve lower energy than the other dataflows. The major reason for such an energy reduction is the savings of on-chip buffers within PEs. For dataflows $[k]$, $[i, k]$, and $[i, k]$, as data of matrix C are accumulated across PEs along the k axis, AutoSA applies array contraction and only allocates a single register inside PEs. We give such an example in Figure 3.1b. In contrast, for the rest of the dataflows, we allocate local buffers implemented by BRAMs inside PEs for storing the intermediate results. Figure 3.18 presents the energy cost breakdown for two designs with dataflows $[i]$ and $[k]$ and the loop ordering $< [i, j], k >$. As seen from the figure, the BRAM access energy inside PEs takes up 56% of the total design energy for dataflow $[i]$. With the elimination of local buffers inside PEs, the second design $[k]$ achieves a 1.8× energy reduction compared to the first one.

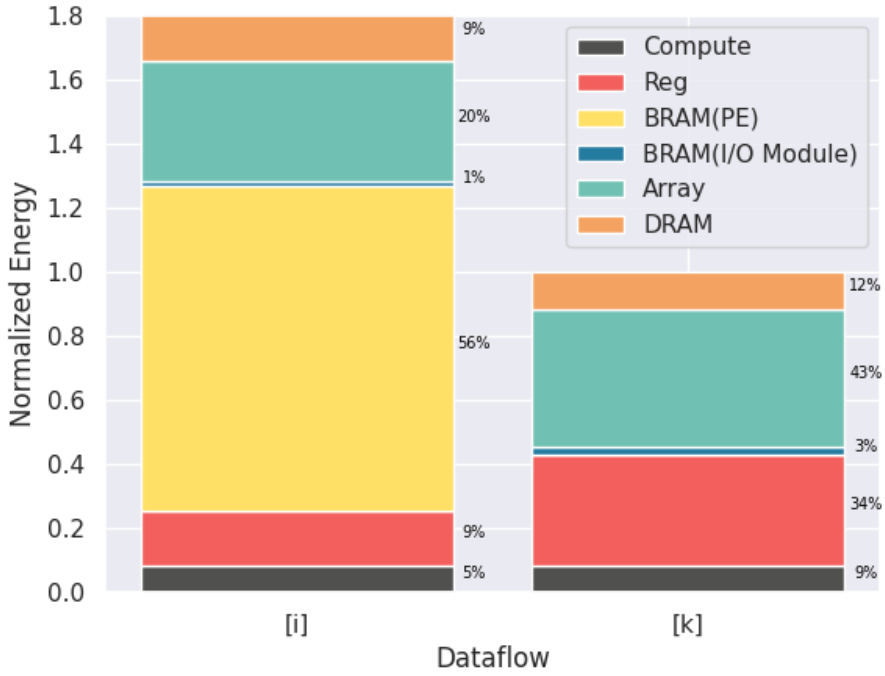


Figure 3.18: Energy breakdown for two designs with the loop ordering $< [i, j], k >$ for MM.

3.6.3.3 Summary

In this subsection, we performed a detailed architecture analysis of different systolic arrays for MM in terms of two performance metrics, latency and energy.

Regardless of the metric of latency or energy, we observe that it is always better to choose the loop ordering $< [i, j], k >$. When targeting latency-optimal designs, the dataflow $[i, j]$ achieves the maximal performance. When energy is the major concern of the system, it is better to choose from the three dataflows $[k]$, $[i, k]$, and $[j, k]$. The analysis above demonstrates the complex trade-offs when choosing the proper architecture under different performance requirements. The Odyssey framework eases the efforts of performing such a design space exploration.

3.6.4 Application 2: CNN

In the previous section, we perform a detailed performance analysis of different systolic array designs on MM. We show the performance trade-offs of different dataflows and loop orderings when targeting two different performance metrics, latency and energy. In this section, we apply the same methodology and compare the performance of different systolic arrays for CNN.

As shown in Table 3.6, Odyssey generates 30 different designs for CNN. We evaluate these designs on the VGG16 network [SZ14] and use latency as the optimization target. VGG16 network includes 13 convolutional (CONV) layers and 3 fully-connected layers. We only consider the mapping of CONV layers in this work. We annotate the 13 CONV layers as CONV[1-13] following their order in the network. Figure 3.19 presents the detailed evaluation results of the first 3 CONV layers of VGG16. Results for the rest of the layers can be found in Appendix 3.A.

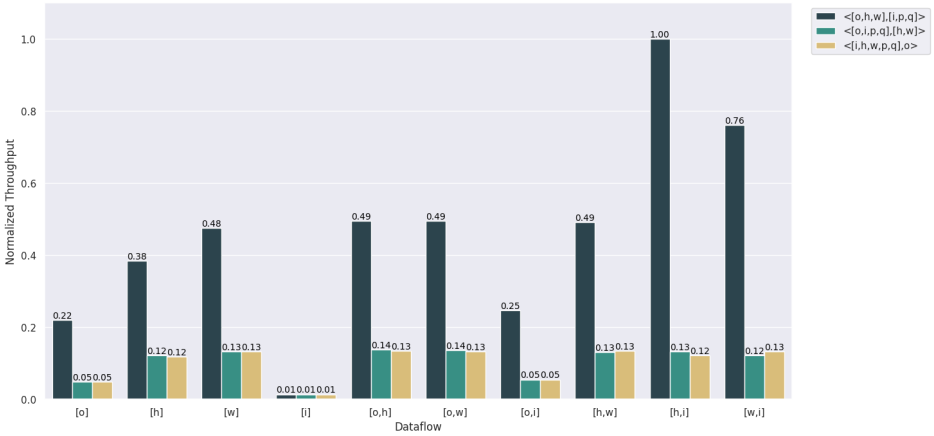
We make the following conclusions from this figure.

Loop permutation Consistent with the observation from the MM case study, designs with the loop ordering $< [o, h, w], [i, p, q] >$, which eliminates the additional I/O modules for transferring the intermediate results of output feature maps, achieve the best performance across all the dataflows.

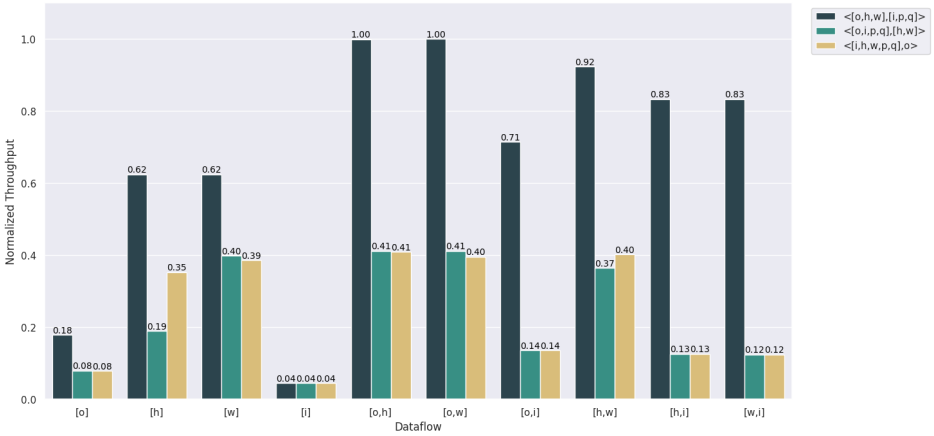
Dataflow Performance of different dataflows varies across CONV layers. On CONV1, we observe that dataflow $[h, i]$ achieves the best performance. However, on CONV2, dataflows $[o, h]$ and $[o, w]$ dominate the performance. Below, we examine the best designs on these two CONV layers in detail.

Table 3.14 summarizes the performance details of two designs with dataflows $[h, i]$ and $[o, h]$ on first two CONV layers. Figure 3.20 depicts the detailed architectures of these two designs.

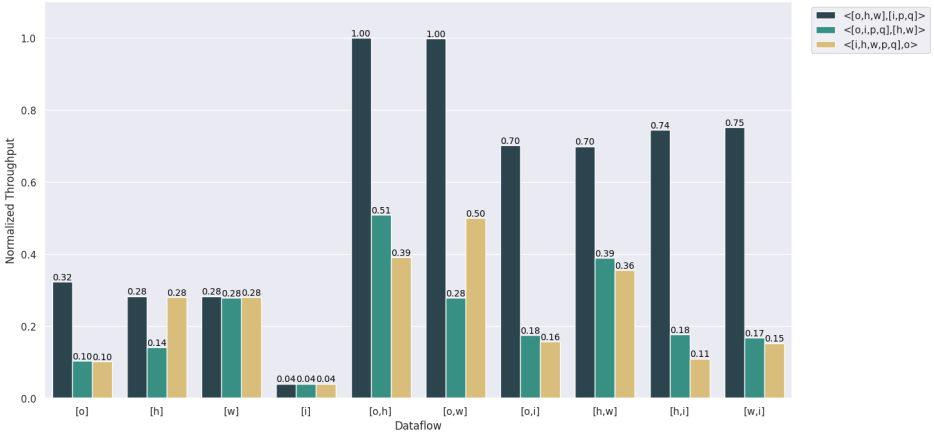
On CONV1, the design latency of dataflow $[o, h]$ is bound by the data transfer latency



(a) CONV1



(b) CONV2



(c) CONV3

Figure 3.19: Normalized throughput of different systolic arrays on CONV1, CONV2, CONV3 of VGG16. Throughput of each systolic array design is normalized against the highest number achieved on each CONV layer.

Table 3.14: Performance comparison of dataflows $[h, i]$ and $[o, h]$ on the CONV1 and CONV2 of VGG16.

CONV Layer	Problem Sizes	Dataflow	SIMD	Latency	PE Latency	fo I/O Latency	DSP	DSP Eff.	T_I1	Bottleneck
1	[I,O,H,W,P,Q]:	[h.i]	[i]	2.86×	2.85×	1.00×(L2)	1,120	67%	4	Problem size
	[3,64,224,224,3,3]	[o,h]	[i]	3.99×	2.85×	3.99×(L1)	1,120	48%	4	I/O network
2	[I,O,H,W,P,Q]:	[h.i]	[i]	7.16×	7.13×	2.14×(L2)	7,680	84%	64	Parallelism
	[64,64,224,224,3,3]	[o,h]	[i]	5.88×	5.85×	4.17×(L1)	8,320	94%	64	

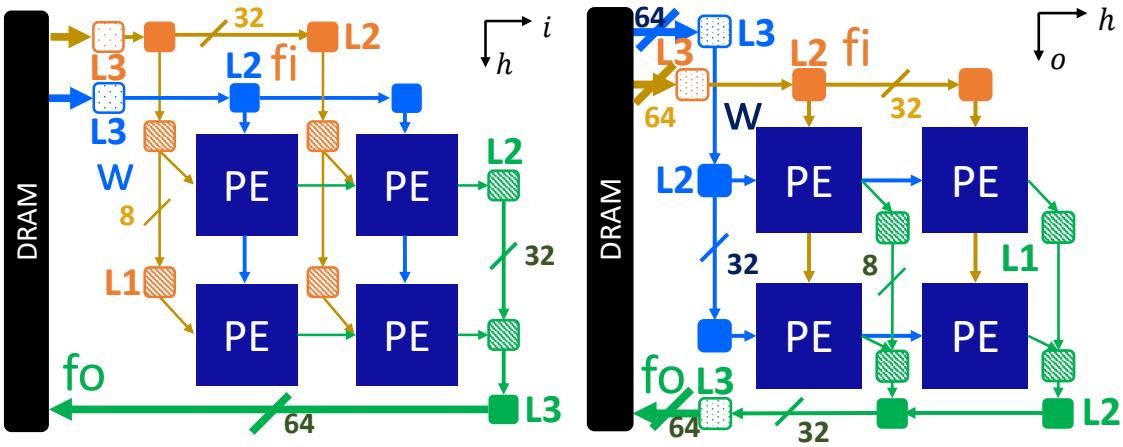


Figure 3.20: Architectures of two systolic arrays with dataflows $[h, i]$ and $[o, h]$ for CNN.

in the L1 I/O modules for draining out the output feature maps fo. In comparison, on dataflow $[h, i]$, data of fo are accumulated across PEs and drained out by L2 I/O modules. By default, we instruct AutoSA to pack at most 8 and 64 bytes of data between L1 and L2 I/O modules, respectively. As more data are packed between the L2 I/O modules, the I/O network latency is no longer the performance bottleneck for dataflow $[h, i]$. Instead, the loss of the performance comes from the zero padding on the input channel dimension i. The best first-level tiling factor T_I1 found for both dataflows is 4. As a consequence, the input dimension i is padded from 3 to 4 for both dataflows, introducing 25% computation overheads.

On CONV2, the I/O network is no longer the performance bottleneck for the dataflow $[o, h]$. The input dimension i increases from 3 to 64 on this layer. As a result, there is more computation inside PEs on accumulating the intermediate results along the i dimension. We see from Table 3.14 that the PE latency now dominates the design, making this design compute-bound. On this layer, dataflow $[h, i]$ achieves a slightly lower performance than the

$[o, h]$ due to the fewer degrees of parallelism to explore. Both dataflows vectorize the loop i. The dataflow $[o, h]$ further explores two more levels of parallelism by mapping the loops o and h to space dimensions. In comparison, the dataflow $[h, i]$ only exploits one more level of parallelism on the loop h. With more degrees of freedom to exploit in the design space, we are able to find a design which utilizes more DSPs on-chip and achieves higher performance.

Finally, Figure 3.21 presents the best throughput achieved by systolic arrays with different dataflows on VGG16. We fix the loop ordering to $< [o, h, w], [i, p, q] >$ for all dataflows. Figure 3.21 computes the geometric mean of the best throughput of each dataflow over the entire network.

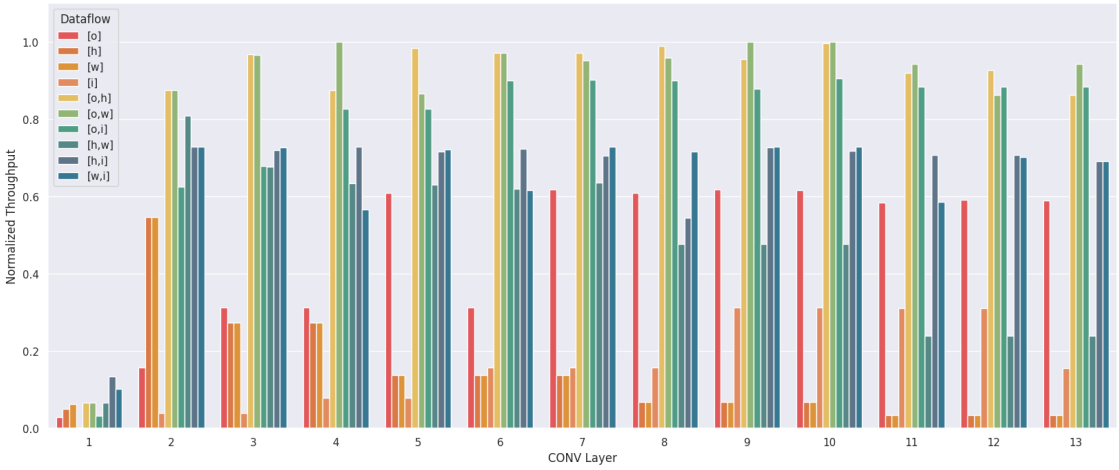


Figure 3.21: Throughput comparison of different dataflows on VGG16. Throughput of each design is normalized against the highest throughput achieved on the entire network.

Overall, we observe that 2D systolic arrays achieve higher performance than the 1D arrays with more degrees of parallelism to exploit. Dataflows $[o, h]$ and $[o, w]$ achieve the best performance among all the dataflows, followed by $[o, i]$, $[h, i]$, and $[w, i]$.

The best average throughput of a single dataflow on VGG16 is 77% of the peak layer-wise throughput attained on the entire network. This is due to the divergent problem sizes of each CONV layer that limit the performance of systolic arrays. For example, we just showed in Table 3.14 that the performance of dataflow $[o, h]$ is limited by the small value of i on CONV1. As a result, this dataflow only reaches 7% of the peak throughput on this layer.

The limited average throughput indicates the low computation efficiency when mapping

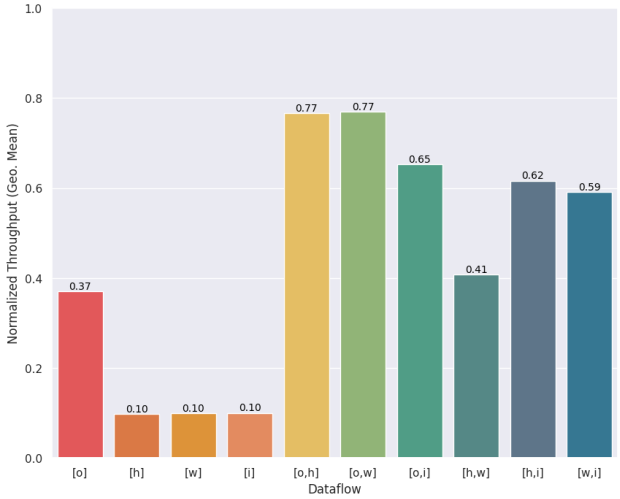


Figure 3.22: Geo. Mean of the best throughput achieved by different dataflows across all the CONV layers in VGG16.

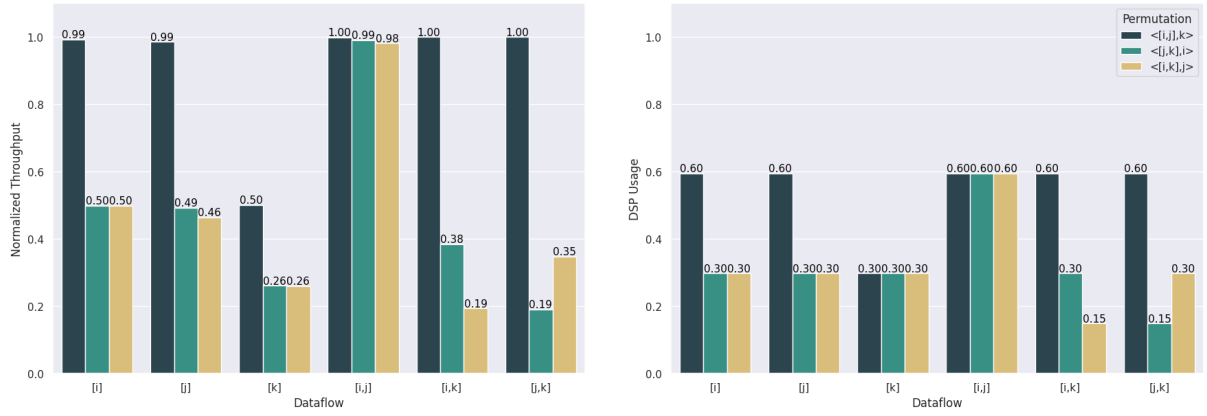
CNNs to a single systolic array. Ideally, we would expect the systolic array to realize 100% of the peak throughput across the entire network such that all the DSP resources are fully utilized. An average throughput lower than 100% indicates that only part of the DSPs are utilized in certain layers. This issue is not unique to VGG16 and can be found on other CNNs. We will discuss this issue in detail and present our solutions in the next chapter.

3.7 Lessons Learned from This Work

This section summarizes four major lessons we learned from this work.

Lesson 1: Incomplete and inaccurate design space modeling could lead to inferior search results and skewed architecture conclusions. We have demonstrated that considering only divisor tiling factors or overlooking the prologue and epilogue phases could lead to inferior designs. This could further hurt the validity of the architecture conclusions derived from these results. For example, Figure 3.23 illustrates the search results for MM with only divisor tiling factors. With the limited design space, the best systolic array design only uses 60% of the DSP. Furthermore, architects may draw skewed conclusions from these results such as all three 2D array dataflows achieve the equivalent performance, which contradicts to

the findings as seen in Figure 3.15.



(a) Throughput (b) DSP

Figure 3.23: Throughput and DSP usage of different designs with only divisor tiling factors for 1024×1024×1024 MM.

Lesson 2: A high-performance design needs to balance computation and communication. We show that the latency-optimal design does not necessarily minimize the off-chip data communication and both computation and communication need to be considered. Several previous works [CHW20, CKR20] that used the off-chip communication to prune the design space may leave out the optimal designs, leading to inferior performance.

Lesson 3: Iterative search methods can be enhanced with mathematical programming solvers. Iterative methods achieve high sample efficiency and are portable to different search problems. It can be further enhanced with mathematical programming solvers. We have demonstrated that a MP-based optimizer which uses a simplified objective function that considers data communication and computation resource can effectively boost the convergence of evolutionary search. Although this work only focuses on systolic arrays, the methodology is general, and designers can apply it to performance tuning tasks for other applications and hardware architectures.

Lesson 4: Compiler-assisted design space construction improves productivity and performance. To the best of our knowledge, Odyssey is the first work that constructs

the design space of systolic arrays with an open-source compiler that generates real hardware designs. This approach guarantees the validity and accuracy of search results and improves the productivity. In the recent years, we have seen an emergence of domain-specific language compilers for DSAs [KFP18, HBS20, GHI19, THZ20]. We believe the methodology in this work could provide valuable insights to further enhance the performance tuning of these works and enable a wider adoption of DSAs in the post-Moore's law era.

3.8 Summary

This chapter presents Odyssey, an efficient design space exploration framework for AutoSA. Odyssey covers a comprehensive and accurate design space of systolic arrays, and incorporates a hybrid search method consisted of the mathematical programming-based optimizer and evolutionary search. This search method is highly efficient. It could find designs reaching 90% of the optimal performance in 5 seconds. Lastly, it can automatically generate the performance models of systolic arrays which minimizes the manual efforts and improves the overall productivity.

We have also conducted a detailed architecture analysis of systolic arrays on two applications, MM and CNN. The same methodology can be applied to other applications as well. We observe that dataflows, loop permutation and tiling impact the performance of systolic arrays in an intertwined manner. The studies on CNN reveal the issue of low computation efficiency of a single systolic array design. We will discuss our solutions to this problem in the next chapter.

3.A Layer-Wise Systolic Array Performance Results on VGG16

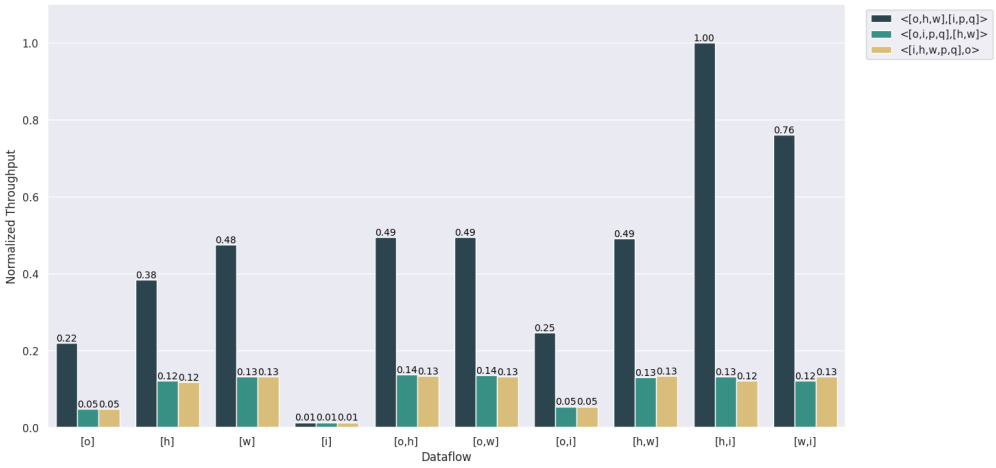


Figure 3.24: Normalized throughput of different systolic arrays on CONV1 of VGG16. Throughput of each systolic array design is normalized against the highest number achieved on each CONV layer.

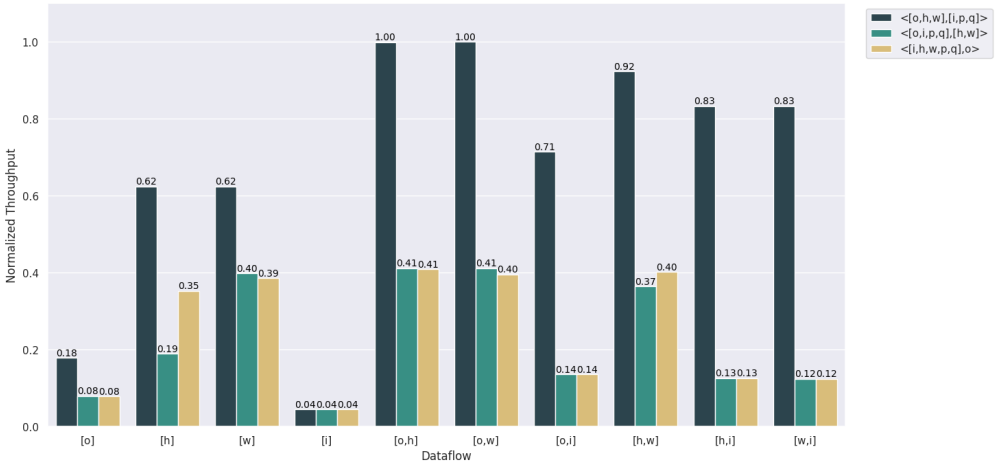


Figure 3.25: Normalized throughput of different systolic arrays on CONV2 of VGG16.

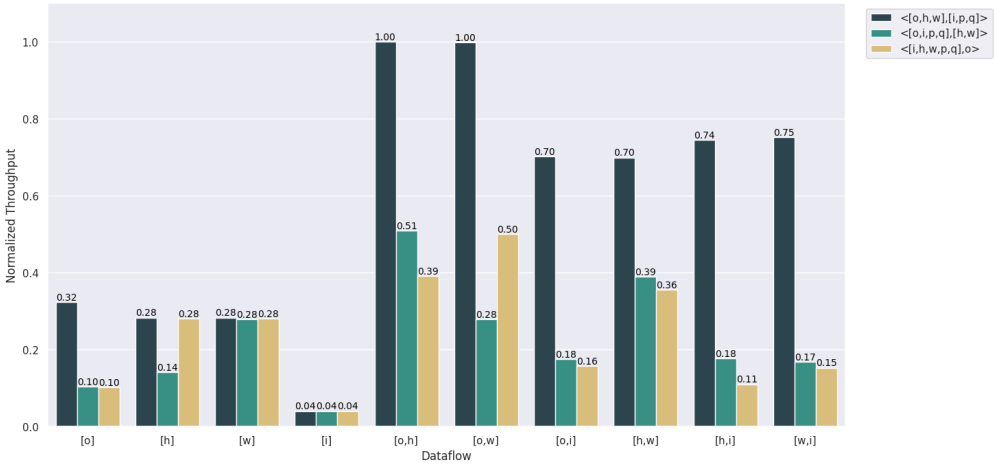


Figure 3.26: Normalized throughput of different systolic arrays on CONV3 of VGG16.

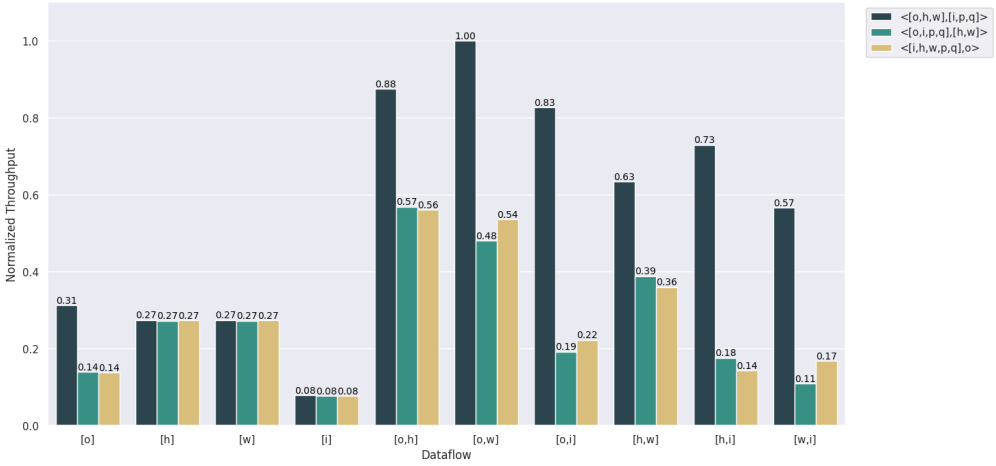


Figure 3.27: Normalized throughput of different systolic arrays on CONV4 of VGG16.

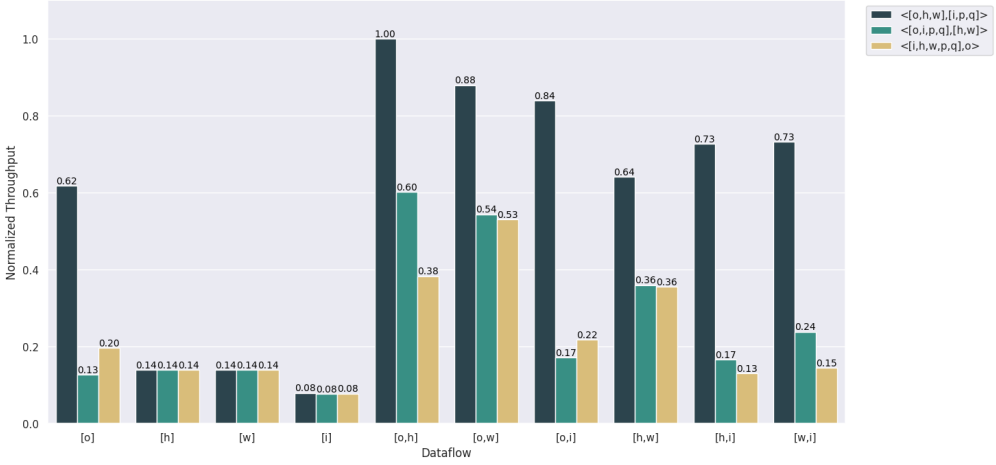


Figure 3.28: Normalized throughput of different systolic arrays on CONV5 of VGG16.

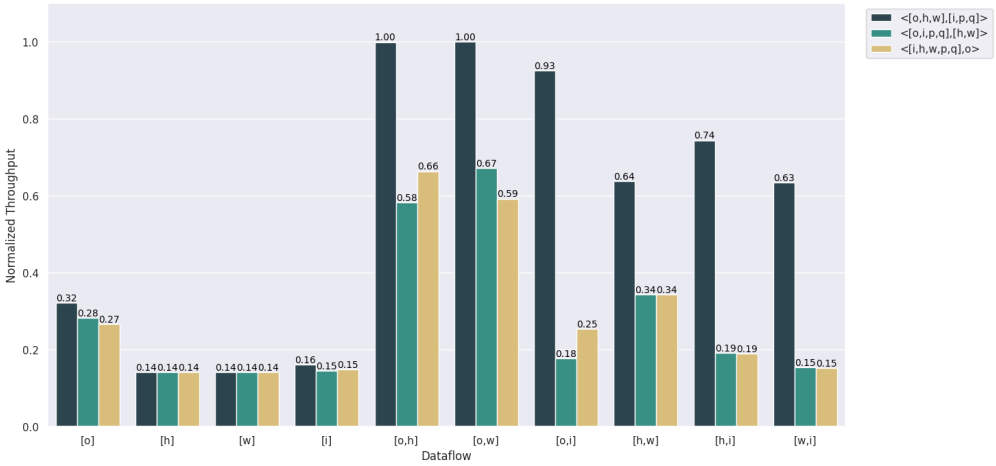


Figure 3.29: Normalized throughput of different systolic arrays on CONV6 of VGG16.

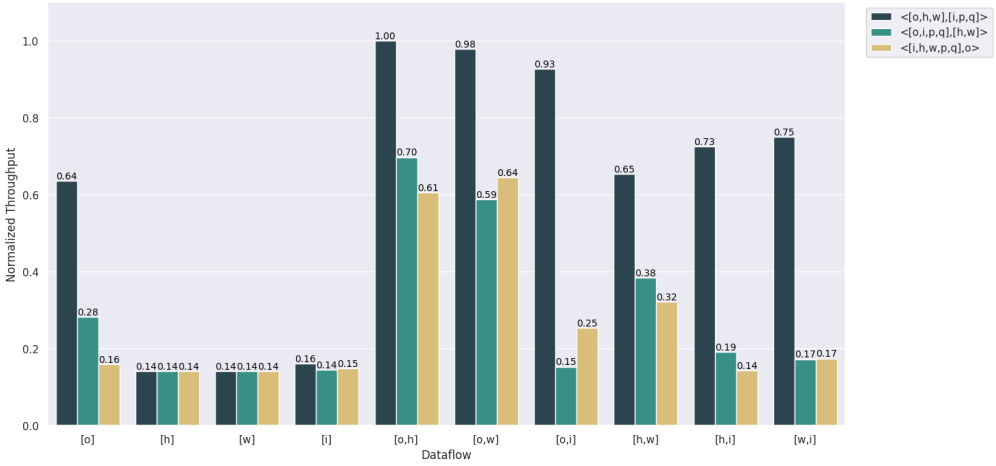


Figure 3.30: Normalized throughput of different systolic arrays on CONV7 of VGG16.

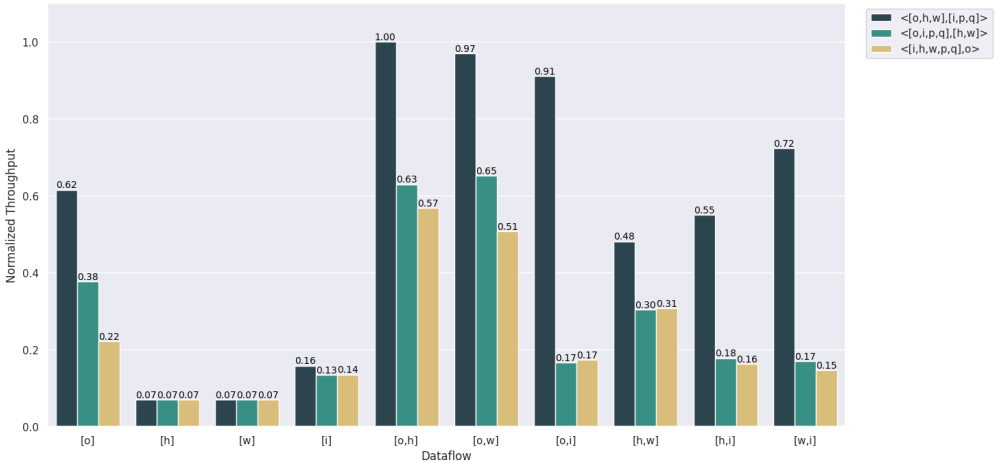


Figure 3.31: Normalized throughput of different systolic arrays on CONV8 of VGG16.

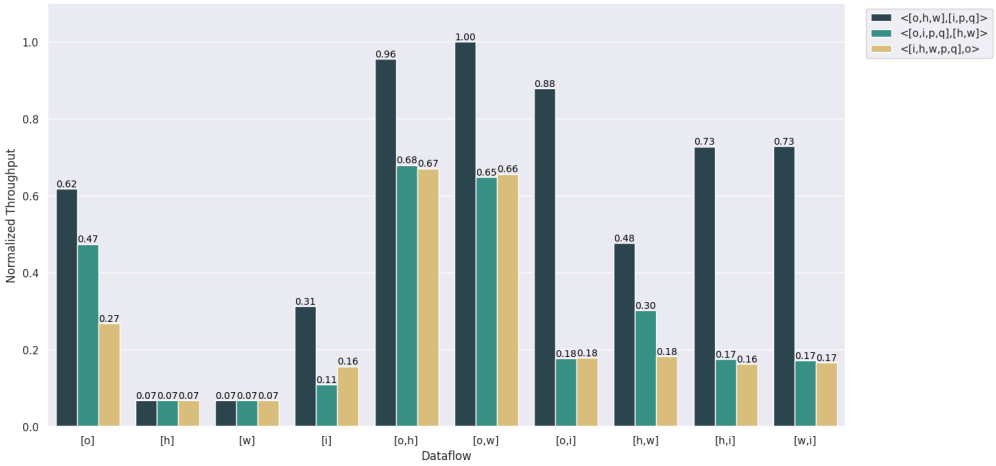


Figure 3.32: Normalized throughput of different systolic arrays on CONV9 of VGG16.

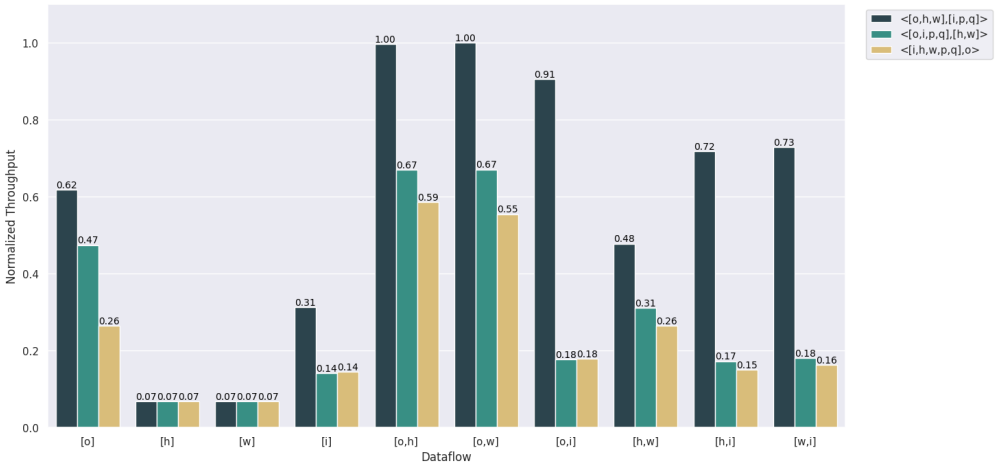


Figure 3.33: Normalized throughput of different systolic arrays on CONV10 of VGG16.

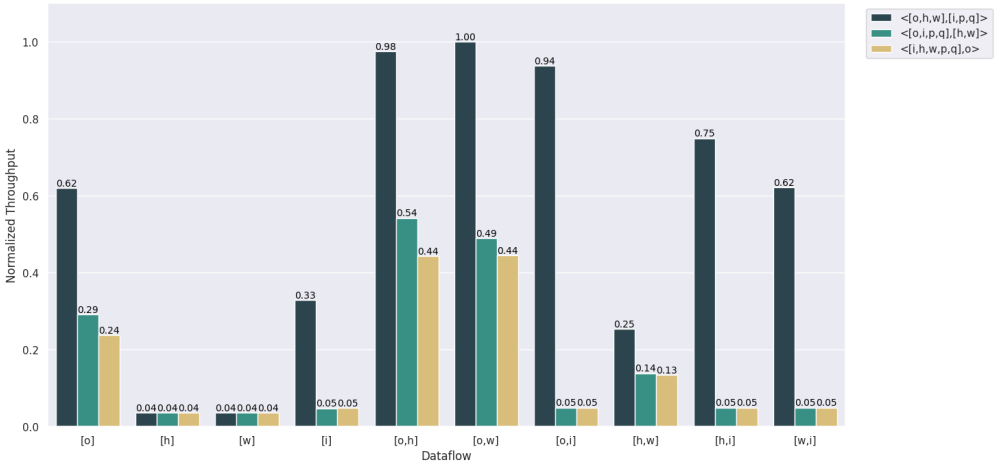


Figure 3.34: Normalized throughput of different systolic arrays on CONV11 of VGG16.

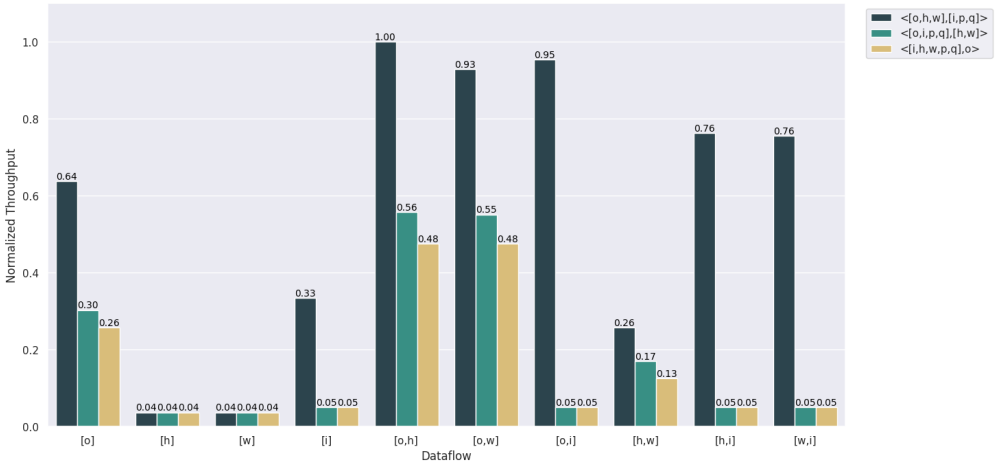


Figure 3.35: Normalized throughput of different systolic arrays on CONV12 of VGG16.

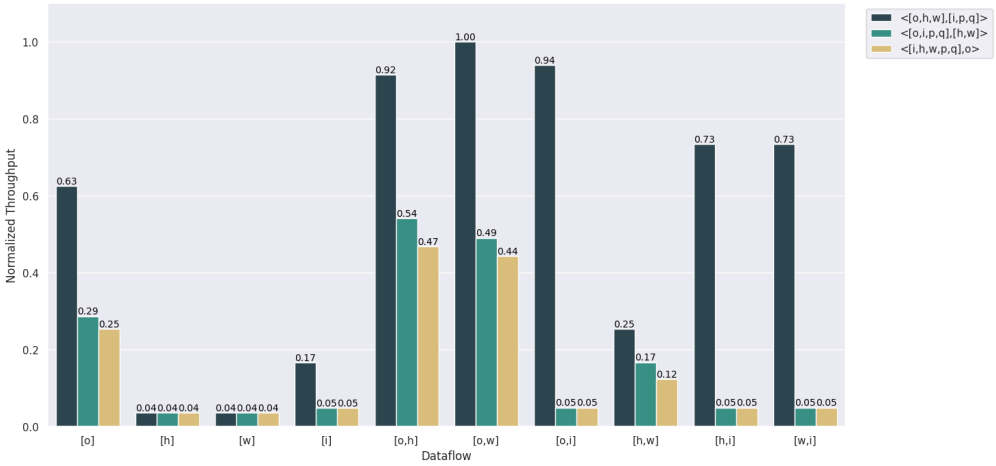


Figure 3.36: Normalized throughput of different systolic arrays on CONV13 of VGG16.

CHAPTER 4

Mapping Convolutional Neural Network to FPGA

In the previous chapter, we presented an efficient auto-tuning framework, Odyssey, for exploring systolic array architectures. Odyssey identifies the best single systolic array architecture for each workload. However, when mapping a complicated application like the convolutional neural network (CNN) to systolic arrays, a new challenge rises. CNNs usually consist of multiple convolutional (CONV) layers. We have demonstrated in the last chapter that CONV layers have divergent characteristics in terms of parallelism and data locality, thus requiring different array configurations to maximize the performance on each layer. Due to this, mapping CNNs to a single monolithic array results in a limited performance. For example, on Xilinx Alveo U250 FPGA, the best single systolic array architecture we found achieved only a 73% DSP efficiency on VGG16 [SZ14]. This issue is further exacerbated on recent CNNs with less parallelism and a lower level of data locality. On ResNet50 [HZR16] and MobileNetV2 [SHZ18], the single array architecture only managed to achieve a 35% and a 19% DSP efficiency. The low computation efficiency of single systolic array architecture reveals the need to explore new architecture optimizations to further improve the performance. In this chapter, we will explore different architecture optimizations to overcome this challenge.

4.1 Background and Related Work

Many previous works for CNN acceleration [SWC20, HKD21, Ama20, JYP17, WYZ17] have used a single monolithic systolic array. For example, the first generation of Google TPU [JYP17] implemented a 128×128 systolic array as the main compute unit. The later generations of TPUs [Clo21] further increased the number of systolic arrays on-chip. However, since each array still operates independently, they are all placed under the same umbrella as single monolithic arrays.

The dominance of such an architecture in the previous works comes from the simpleness of this design which requires minimal development efforts. However, this architecture faces severe performance issues because of the characteristic disparity of CNN layers. A CNN usually consists of a sequence of convolutional (CNN) layers with different problem sizes. These layers exhibit different levels of parallelism and data locality, thus requiring different systolic array configurations to maximize the performance. In this chapter, we have conducted a comprehensive study evaluating the performance of a single systolic array. Our evaluation results reveal that a single monolithic systolic array achieves 73%, 35%, and 19% DSP efficiency on three representative CNNs, VGG16 [SZ14], ResNet50 [HZR16], and MobileNetV2 [SHZ18], respectively. This observation has multiple profound implications. First, it indicates that simply scaling up the array sizes brings minimal benefits to the overall performance given the low computation efficiency on these networks. Second, with the recent trend of CNNs, we foresee that such a performance issue will become worse in the future. Recent CNNs reduce the network size that brings less parallelism and data locality while achieving a similar or better prediction accuracy. For example, ResNet50, first proposed in 2015, is 5.4× smaller in network size compared to VGG16, which was proposed in 2014. Table 4.1 summarizes the year of release, top-1 prediction accuracy, and the total number of parameters of three CNN networks.

The limitation of single systolic arrays drives the need for new architecture optimizations to improve the performance. Numerous previous works have made attempts in different dimensions. Table 4.2 summarizes a few representative previous works with details. In

Table 4.1: Details of VGG16, ResNet50, and MobileNetV2 and the DSP efficiency of the best single systolic array for these CNNs on Xilinx Alveo U250.

Networks	Year	Top-1 Accuracy	Parameters	DSP Efficiency
VGG16	2014	0.713	138.4M	73%
ResNet50	2015	0.749	25.6M	35%
MobileNetV2	2018	0.713	3.5M	19%

this table, we extend the scope of hardware architecture beyond systolic arrays, which also includes the conventional vector processor [ZWZ18, SFM17, ACF16] and other customized accelerators [KSK18, KLP21, KCS20]. We classify these works by the following three dimensions.

Table 4.2: Comparison of different hardware architecture exploration frameworks for CNN.

Works	Systolic Array Included?	Dataflows Explored?	Layer Execution Model	#Accelerators	Is Hardware Open-Sourced?
TGPA [WLL18]	Y	N	Sequential/Fusion	Multi	N
DNNBuilder [ZWZ18]	N	N	Fusion	Multi	Y
DNNExplorer [ZYW20]	Y	Y	Sequential/Fusion	Multi	N
FlexCNN [SWC20]	Y	N	Sequential	Single	Y
MAERI [KSK18]	N	Y	Sequential	Single	Y
TENET [LGW21]	Y	Y	Sequential	Single	N
dMazeRunner [DKA19]	Y	Y	Sequential	Single	N
CoSA [HKD21]	Y	Y	Sequential	Single	N
Maestro [KCS20]	Y	Y	Sequential	Single	N
Interstellar [YGL20]	Y	Y	Sequential	Single	N
ConfuciuX [KJK20]	Y	Y	Sequential/Fusion	Single/Multi	N
Herald [KLP21]	Y	Y	Sequential	Multi	N
Shen et al. [SFM17]	N	N	Sequential	Multi	N
Alwani et al. [ACF16]	N	N	Sequential/Fusion	Multi	N
Ours	Y	Y	Sequential/Fusion	Single/Multi	Y

Single dataflow vs. multi-dataflow We have shown in the last chapter that the CNN layer performance varies across systolic arrays with different dataflows. This phenomenon is also observed by many previous works [KLP21, KCS20, CKR20, YGL20]. Therefore, it is profitable to support multiple dataflows in the hardware architecture and choose the best dataflow for each CONV layer. The previous work MAERI [KSK18] proposed a reconfigurable architecture that can be configured dynamically during the runtime to support different dataflows. However, the reconfiguration logic introduces additional area and energy overheads.

To overcome this problem, the later work Herald [KLP21] proposed to implement multiple arrays with different dataflows and offload layers from a set of CNNs to different arrays to maximize the overall performance. Since the CONV layers from different CNNs are interleaved during the execution, the latency of each single CNN task is prolonged which may limit the usage of this architecture for latency-sensitive applications.

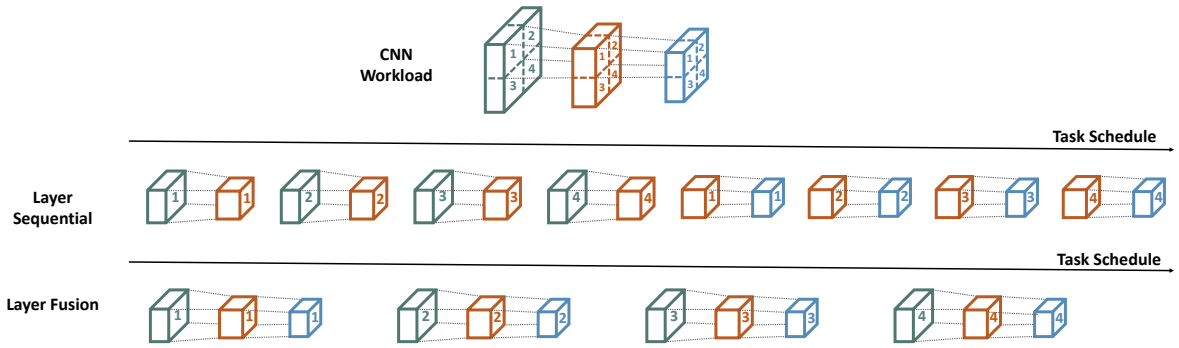


Figure 4.1: Comparison of layer sequential and layer fusion.

Layer sequential vs. layer fusion CNNs usually consist of a sequence of different layers. When executing the CNN on the accelerator, one common approach is to launch these layers in sequence [SWC20, HKD21]. The accelerator completes the entire layer first and then moves to the next layer. We name such an execution model *layer sequential*. Meanwhile, several previous works [ACF16, WLL18, ZWZ18, ZYW20] proposed to fuse a group of layers together. The computation order is reorganized at the unit of image tiles. This execution model is named *layer fusion*. Figure 4.1 compares the execution model of layer fusion against layer sequential. As shown in the figure, in the layer fusion model, as long as the first CONV layer completes the computation of one tile of the output feature maps, the outputs are immediately consumed to initiate the computation of the following layer. The intermediate results of each layer are usually stored on-chip to avoid the off-chip communication. The benefits of layer fusion are two-fold. For compute-bound layers, layer fusion saves the off-chip data communication, which helps improve the energy efficiency. For communication-bound layers, it helps reduce the latency by avoiding swapping out the intermediate results off-chip. However, layer fusion also introduces additional area overheads with the on-chip buffers which

needs to be carefully examined before making the final design choice.

Single array vs. multi-array A single monolithic array usually faces issues of low computation efficiency considering the performance disparity of CNN layers. Therefore, several previous works proposed to implement multiple arrays on-chip with each array customized for certain layers to maximize the overall performance. DNNBuider [ZWZ18] mapped each CNN layer to a single accelerator and connected them through on-chip buffers. However, as the number of CNN layers grows, the benefits of this architecture are diminished by the increasing control and logic overheads of single arrays. The later work DNNExplorer [ZYW20] proposed only to map the first few layers to separate accelerators and group the rest of the layers to a single array. Other works like TGPA [WLL18] chose to implement a few systolic arrays on-chip and assign the CNN layers periodically to these arrays. The recent work Herald [KLP21] proposed to place multiple accelerators on-chip and offload the layers from a set of different CNNs to the best suitable arrays. As discussed earlier, this architecture could significantly stretch the processing latency of each single CNN task and is less favored when the single image inference latency is the priority of the system.

Previous works have made various attempts in improving the architecture efficiency of CNN processing from different dimensions. While they all demonstrated certain performance improvement, due to the lack of open-source implementations and the inconsistent hardware baseline, it remains unclear about the relative effectiveness of these approaches.

Second, as illustrated in Table 4.2, many previous work only considered part of the design space which may limit the performance of the proposed architecture. For example, while TGPA [WLL18] and DNNExplorer [ZYW20] explored the design choices of multiple arrays and the layer fusion execution model, they only used one fixed dataflow for all their accelerators. As we will show later, this will limit the performance results of their architectures, which could further obfuscate the architecture conclusions derived from these results.

The work in this chapter shows our efforts in overcoming the two issues mentioned above. We discuss our contributions in detail in the next subsection.

4.2 Contributions

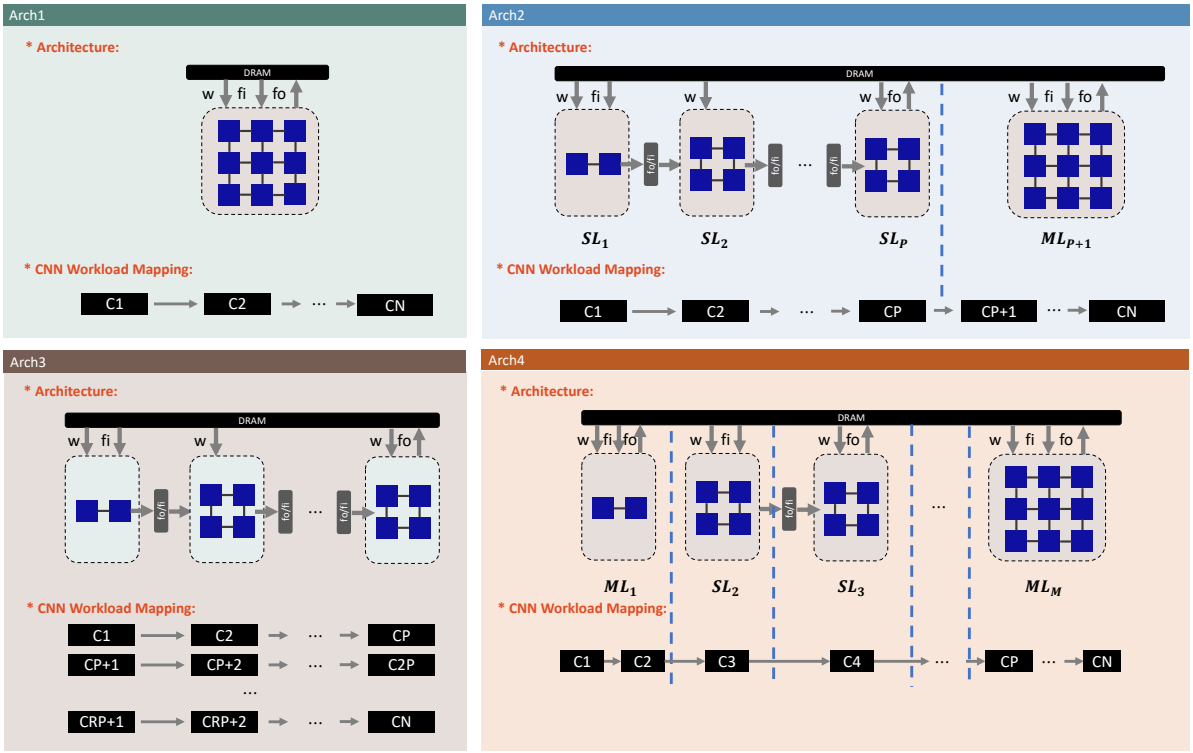


Figure 4.2: Architecture and workload mapping of four different designs for CNN acceleration.

In this chapter, we use our open-source systolic array compiler, AutoSA [WGC21], to build a consistent and reproducible hardware baseline for architecture comparison. We select three different architecture configurations as shown in Figure 4.2 that cover all the design dimensions as mentioned in the previous section.

The first architecture (Arch1) is the conventional single monolithic systolic array which we see in many previous works [SWC20, HKD21, Ama20, JYP17, WYZ17].

The second architecture (Arch2) incorporates a multi-array system. It was proposed by the previous work DNNExplorer [ZYW20]. A CNN is split into two parts. All the layers before the splitting point are mapped to separate arrays, and the layers after the splitting point are mapped to a single array. DNNExplorer only considered a single dataflow $[i, o]$ on all the arrays. In our evaluation, we extend the scope to all the dataflows generated by AutoSA. Arrays before the splitting point adopt the layer-fusion execution model. We insert on-chip streaming buffers in-between these arrays such that the second array could start as

soon as the first tile of data is ready in the streaming buffers. The single array after the splitting point uses the layer sequential model.

The third architecture (Arch3) was proposed by the previous work TGPA [WLL18]. In comparison to Arch2, which implements a one-shot schedule that maps all CONV layers to different arrays at one time, Arch3 follows a periodic schedule by periodically mapping a few adjacent CONV layers to multiple arrays and executing the entire CNN in multiple rounds. All the arrays in each round follow the layer-fusion model, and CONV layers mapped to each arrays following the layer sequential model across rounds. Similarly, we evaluate all dataflows generated by AutoSA for each array.

These three architectures cover all the design space dimensions we have discussed earlier and serve as good examples for understanding the performance impacts of different design factors. Table 4.3 summarizes the covered design dimensions by these three architectures.

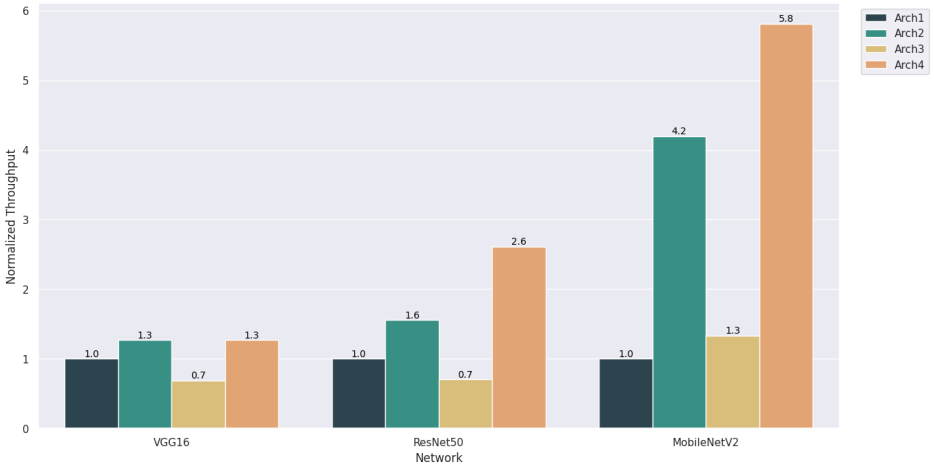
Table 4.3: Comparison of the explored design space dimensions by different array architectures.

Architecture		Arch1	Arch2	Arch3
Datatflows		✓	✓	✓
Layer Execution Model	Sequential	✓	✓	✓
	Fusion	✗	✓	✓
#Arrays		Single	Multi	Multi

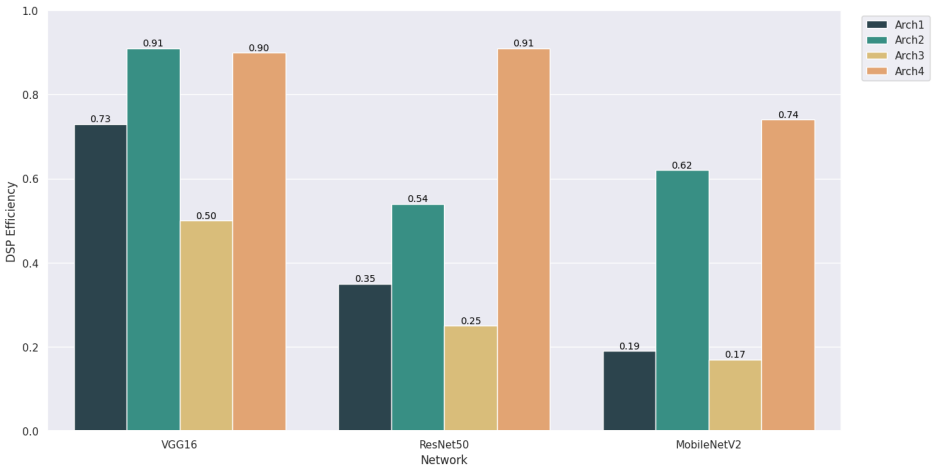
Figure 4.3 presents the performance results we collected for these three architectures on three representative CNNs, VGG16, ResNet50, and MobileNetV2. We have seen a improvement in both throughput and computation efficiency from Arch2 versus Arch1. However, Arch 3 fails to demonstrate an improvement on the evaluated CNNs. Detailed reasons will be discussed later.

Furthermore, although Arch2 helps improve the design performance effectively for VGG16 by improving the DSP efficiency from 73% to 91%. The improvements on two other CNNs are less effective. This is due to the rigid partitioning scheme of this architecture which maps all the later layers into a single systolic array that faces the same limitation of Arch1.

To further improve the performance, we propose a new architecture (Arch4) which adopts an adaptive partitioning scheme that groups CONV layers with similar performance



(a) Throughput



(b) DSP Efficiency

Figure 4.3: Comparison of the throughput and DSP efficiency of Arch1-4 on different CNNs.

characteristics together and maps to a single array. This architecture surpasses all the three architectures above in both the throughput and DSP efficiency, as illustrated in Figure 4.3.

The studies in this chapter demonstrate the complexities of mapping a complicated application like CNN to systolic arrays. While AutoSA and Odyssey provide efficient solutions to generating a single array, to achieve an optimal performance on a CNN, more hardware optimizations are required to meet the final performance target. The rest of this chapter will cover the detailed evaluation approaches and results on each architecture. Our contributions for this work can be summarized as below.

- *Approaches* We evaluated three systolic array architectures for CNN acceleration from

the previous works. We used an open-source hardware generator, AutoSA, to build a transparent and consistent hardware baseline for all three architectures and performed a systematic and quantitative comparison of these designs.

- *Architecture* The evaluation results on the three architectures reveal different architecture trade-offs when choosing these designs. In addition, we proposed a new multi-array architecture that exploits an adaptive partitioning scheme of CNNs which surpasses all the three baseline architectures in both the throughput and DSP efficiency.

The rest of this chapter is organized as follows. We introduce the evaluation methodology of all the hardware architectures in Section 4.3. Then, we cover the detailed architectural designs, performance models, tuning methods, and the evaluations results for three architectures from the previous works in Sections 4.4, 4.5, and 4.6, respectively. After that, we present our new architecture in Section 4.7. Section 4.8 compares all the evaluated architectures and Section 4.9 summarizes the entire chapter.

4.3 Evaluation Methodology

In this section, we give an overview of the evaluation methodology used in this chapter. We cover the architectures studied in this work and the CNNs for evaluation.

4.3.1 Hardware Architecture

We evaluate four architectures in this chapter, which are named *Arch1*, *Arch2*, *Arch3*, and *Arch4* in the rest of the chapter. Arch 1 implements a single monolithic systolic array. Different convolutional (CONV) layers are executed on this array in sequence. The other three architectures implement a multi-array architecture which consists of several smaller systolic arrays with different configurations. Details of each architecture will be covered in the following sections.

For all the experiments in this chapter, we target Xilinx Alveo U250 FPGA. For the systolic array implementation, we consider all 10 different systolic arrays with unique dataflows

```
for (int o = 0; o < O; o++)
  for (int h = 0; o < H; h++)
    for (int w = 0; o < W; w++)
      for (int i = 0; o < I; i++)
        for (int p = 0; o < P; p++)
          for (int q = 0; o < Q; q++)
            fo[o][h][w] += fi[i][h+p][w+q] * w[o][i][p][q];
```

Listing 9: Example code of a CNN layer. The initialization of the output feature map array fo is omitted for brevity.

Table 4.4: Target systolic array designs for CNN.

Application	Dataflows	Loop Permutation Orderings	#Designs
CNN	[o], [h], [w], [i], [o,h], [o,w], [o,i], [h,w], [h,i], [w,i]	<[o,h,w],[i,p,q]>	10

as listed in Table 4.4. Parameters in this table can be found in Listing 9 which shows an example code of one CNN layer. We set the loop ordering in the array partitioning to $< [o, h, w], [i, p, q] >$ for all these designs. As discussed in Section 3.6.4, this loop ordering dominates the rest of the candidates in terms of the design latency.

4.3.2 CNN Benchmarks

Table 4.5: Year of release, top-1 prediction accuracy and number of parameters of VGG16, ResNet50, and MobileNetV2.

Networks	Year	Top-1 Accuracy	Parameters
VGG16	2014	0.713	138.4M
ResNet50	2015	0.749	25.6M
MobileNetV2	2018	0.713	3.5M

We evaluate our architecture with three CNNs, VGG16 [SZ14], ResNet50 [HZR16], and MobileNetV2 [SHZ18]. Table 4.5 summarizes the basics of these three CNNs. VGG16 was first proposed in 2014 for the image classification contest ImageNet [DDS09]. Later in 2015, ResNet50 was introduced which surpassed VGG16 in top-1 prediction accuracy[1]. The

[1]Top-1 accuracy measures the proportion of image classification tasks for which the predicted label matches the target label.

parameter of one CNN is computed by counting the total amount of elements of the weight matrix w in the entire network. As seen from Table 4.5, ResNet50 achieves a higher prediction accuracy with a reduced network size compared to VGG16. The third network, MobileNetV2, has an even smaller network size than ResNet50. MobileNetV2 targets hardware devices with limited computation resources like mobile phones. It achieves a low computation complexity with a sacrifice on the accuracy.

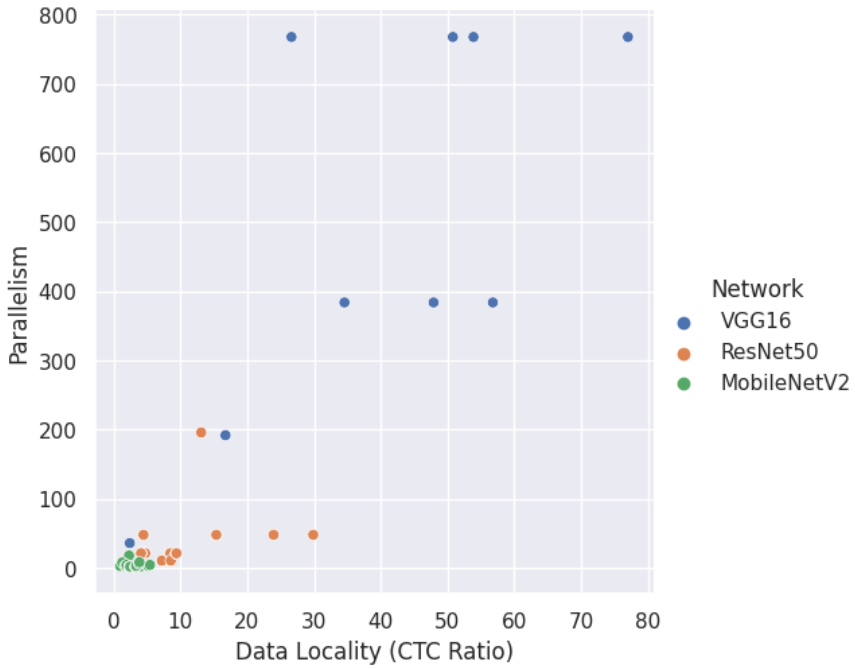


Figure 4.4: Parallelism vs. data locality of all the convolutional layers in VGG16, ResNet50, and MobileNetV2.

In addition to the above-mentioned two metrics, prediction accuracy and network size, these three CNNs also exhibit a disparity in parallelism and data locality. Figure 4.4 compares the parallelism and data locality of all the CONV layers in each network. We measure the *parallelism* as the product of the parallel and reduction loops in the CONV layer ($o \times h \times w \times i \times p \times q$). *Data locality* is estimated using the compute-to-communication (CTC) ratio, which is computed by dividing the total amount of computation operations by the total amount of data transfer in each layer. In this figure, VGG16 is the most compute-intensive network among the three networks. It consists of more CONV layers with a higher level of parallelism and data locality. In comparison, with the reduced network size, CONV layers in

ResNet50 exhibit a lower level of parallelism and data locality than VGG16. This situation is further intensified on MobileNetV2.

The disparity of parallelism and data locality across three networks will pose different challenges to the hardware systems. We use these three CNNs to perform a comprehensive analysis of the performance trade-offs on different systolic array architectures.

4.4 Architecture 1: Single Array

4.4.1 Architecture Overview

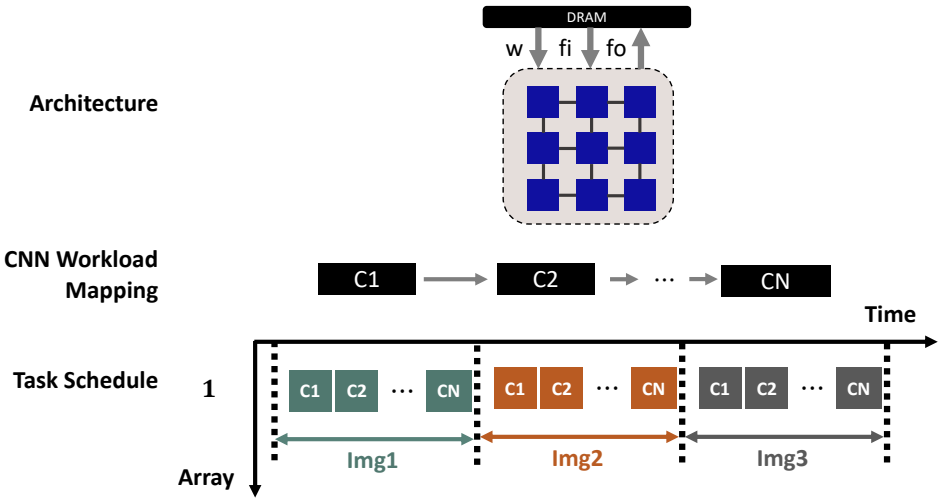


Figure 4.5: Architecture, CNN workload mapping and task schedule of a single monolithic array design (Arch1).

Figure 4.5 presents the architecture, CNN workload mapping and task schedule of this design. We implement a fixed single monolithic array. All the CNN layers are executed sequentially on this array. In the rest of this chapter, we name this array architecture *Arch1*. The major challenge here is to find a systolic array configuration that maximizes the end-to-end performance.

Many previous works [CKR20, KCS20, YGL20, HKD21] that used this architecture start with a pre-set array configuration. Configurations such as the array topology and the on-chip resource allocation are set to fixed values. Then, they explored the mapping of each CNN

layer onto this fixed architecture without violating the hardware constraints. Other works such as [SWC20, ZLS15] employed brute-force methods like exhaustive search to find the optimal array configuration, or selected one layer-optimal array configuration as the final array architecture [KJK20]. Brute-force methods face scalability issues with complex CNNs, and simply selecting one layer-optimal solution may lead to inferior designs. In this section, we present our auto-tuning method that we build upon the Odyssey framework introduced in the last chapter. We evaluate the performance of the single monolithic array architecture on three different CNNs, VGG16, ResNet50, and MobileNetV2.

4.4.2 Tuning Algorithms

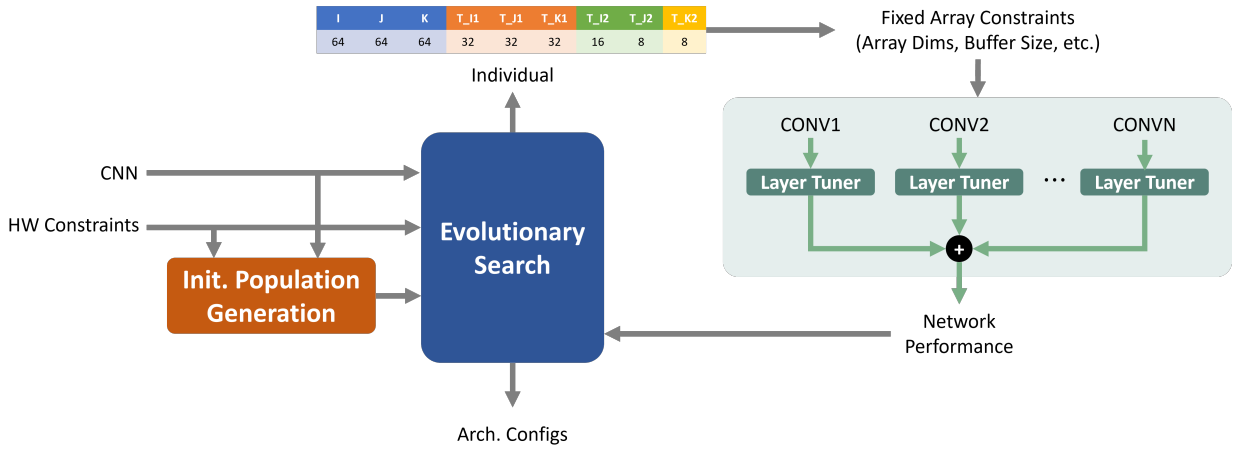


Figure 4.6: Overview of the single array tuning flow.

Figure 4.6 illustrates the proposed auto-tuning framework for this architecture. The backbone of this framework is the same evolutionary search algorithm proposed in the last chapter. Each individual in the evolutionary search is identified by the tiling factors used in the AutoSA compilation flow. These tiling factors define a complete fixed systolic array architecture, including the array dimensions, buffer size inside each hardware module, array interconnects, etc. The fitness score of each individual is the end-to-end CNN performance on this fixed systolic array. To obtain the fitness score, we run the Odyssey framework (named *layer tuner*) for each CNN layer to search for the layer-optimal tiling factors given the fixed array constraints. Specifically, when using the layer tuner, if the sampled tiling

factors lead to an array with dimensions greater than any of the dimensions of the fixed array, or overusing the maximal available compute/memory resource in each hardware module, the layer tuner will mark this configuration illegal and filter it out during the search, such that all the solutions found for each layer obey the fixed array constraints.

To speed up the search process, we implement a thread pool to execute all the layer tuners in parallel. There is one more issue left for this flow: the selection of the initial population of the evolutionary search. We have shown in the last chapter that a high-quality initial population helps improve the convergence of evolutionary search. Similarly, for this auto-tuner, we aim to find a good starting point to speed up the search process.

We discovered that the layer-optimal solutions identified by the layer tuner can serve as good seeds to the evolutionary search. Figure 4.7 presents one experiment illustrating this heuristic. In this experiment, we search the optimal systolic array configuration for each CONV layer of VGG16 separately assuming that each CONV layer can be mapped to a different single systolic array which fully utilizes the entire chip. Then, for each layer-wise optimal solution, we use it to generate a fixed single array configuration and evaluate the network performance on this architecture. Figure 4.7 compares the layer-wise throughput of each solution against the network-wise throughput. We use the systolic array with the dataflow $[o, h]$ for this experiment.

As shown from Figure 4.7, single-layer solutions with a higher layer-wise throughput tend to achieve a higher network-wise performance at the same time. Intuitively, a high-throughput layer-wise solution utilizes more on-chip resource, tending to achieve a higher throughput over the entire network.

Based on the above observation, before the evolutionary search, we introduce an additional phase to find a high-quality initial population. We choose to run the layer tuner for all the CONV layers separately assuming that each CONV layer can be mapped to a single systolic array which fully utilizes the entire hardware. These layer-optimal solutions are used as the initial population to the evolutionary search. If the total number of solutions is greater than the population size p of the evolutionary search, we rank the layer-optimal solutions by its

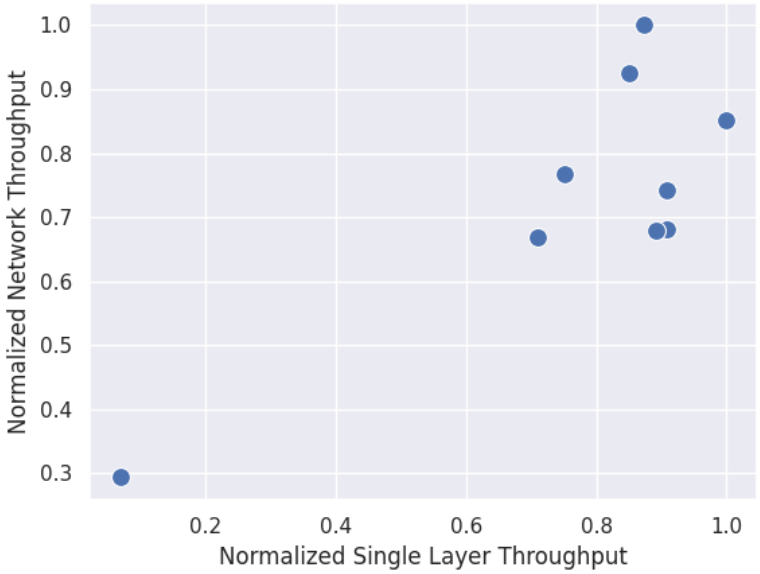


Figure 4.7: Single layer throughput vs. network throughput for VGG16 on array $[o, h]$. (We have omitted layers with the same problem dimensions.)

achieved layer-wise throughput and select the top-p solutions as the initial population.

Figure 4.8 presents the search results of the proposed search method for all 10 systolic arrays in Table 4.4 on VGG16. We limit the search time to 3 minutes for each design and have highlighted the first few epochs evaluating the initial population in the figure. In addition, we compute the ideal latency of the network as a sum of the best layer-optimal latency found in the initialization step. This number gives us an upper bound of the best performance a single monolithic array could achieve. The ideal performance is marked as *ideal max throughput* in the figure.

With the help of this heuristic, we reach up to 97% of the ideal performance among the 10 designs. On average, we find designs achieving 80% of the ideal performance using the initial population. The later iterations of evolutionary search help further improve the final performance by 1.03× on average.

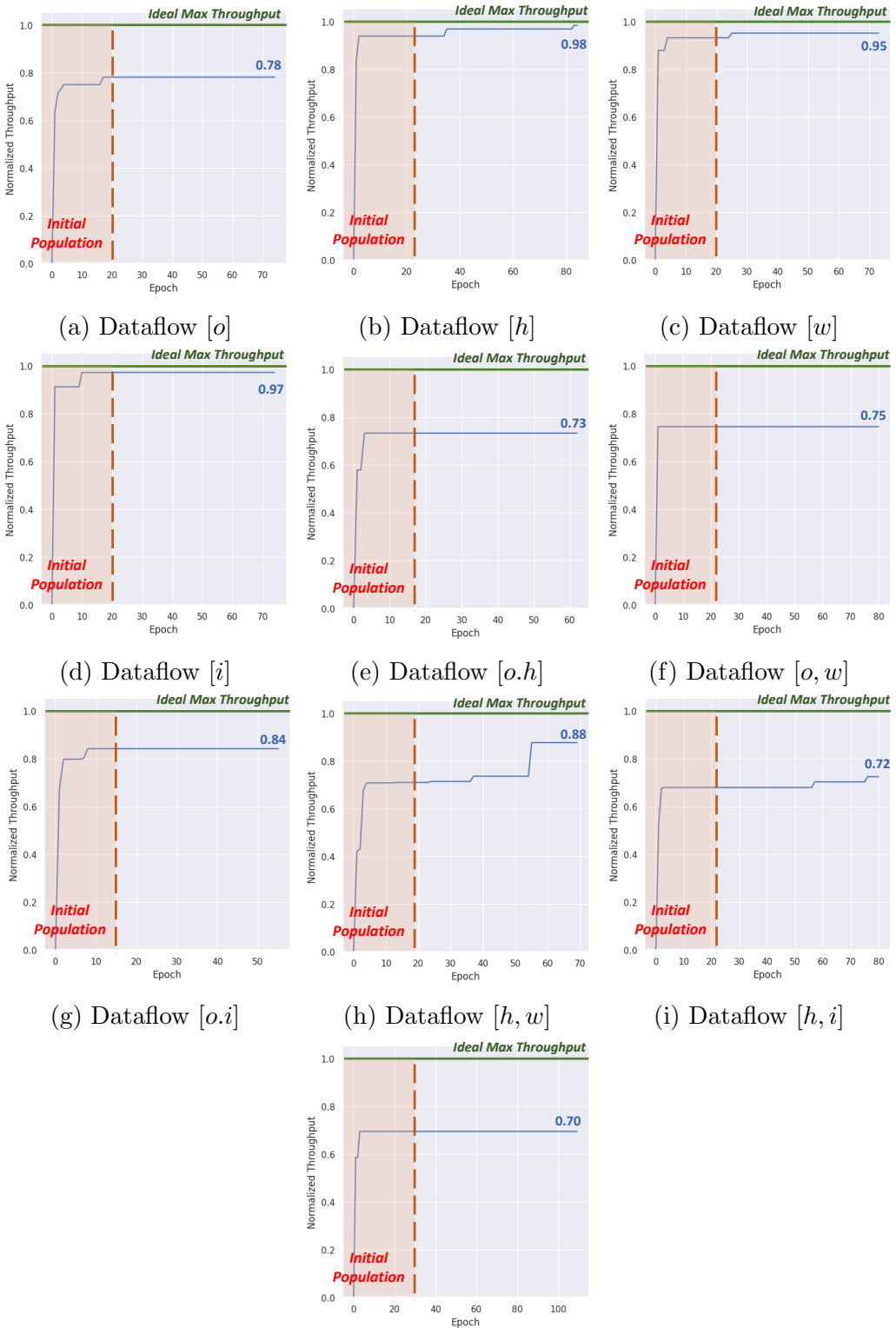


Figure 4.8: Search results of the proposed tuning method for Arch1 on VGG16.

4.4.3 Evaluation Results

4.4.3.1 Overall Results

In this section, we perform a detailed architecture analysis using the auto-tuner proposed in the previous section. We evaluate the latency of the best architecture found by this auto-tuner on three networks, VGG16, ResNet50, and MobileNetV2. The tuning time is set to 3 minutes for each search task. Figure 4.9 compares the performance between different systolic arrays on three CNNs. Table 4.6 presents the detailed information of the best array found for each network.

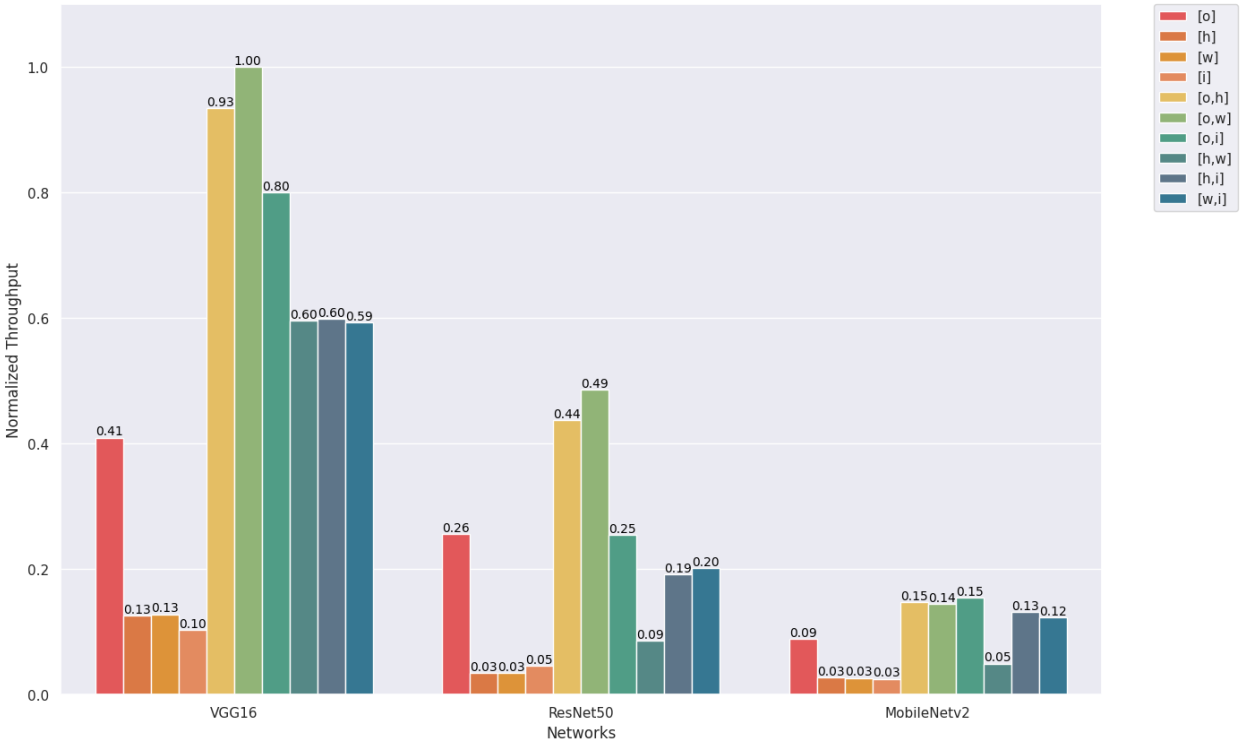


Figure 4.9: Normalized throughput of different systolic array designs on VGG16, ResNet50, and MobileNetV2. The throughput of each design is normalized against the highest throughput achieved by all the designs across three CNNs.

There are two major observations from the above results:

Performance disparity across dataflows We observe a large performance disparity between different dataflows. Two dataflows ($[o, h]$, $[o, w]$) achieve the highest throughput on

Table 4.6: Comparison of best systolic array designs for three CNNs.

Networks	Dataflow	Latency	GFLOPs	DSP	DSP Eff.	BRAM	BW(GB/s)
VGG16	[o,w]	10.0×	731.3	98%	73%	51%	19.17
ResNet50	[o,w]	4.7×	391.0	98%	35%	57%	35.08
MobileNetV2	[o,i]	1.0×	128.7	59%	19%	94%	22.43

VGG16 and ResNet50, followed by the dataflow $[o, i]$. On MobileNetV2, the dataflow $[o, i]$ attains the highest throughput, followed by $[o, h]$ and $[o, w]$. Three 1D systolic arrays ($[h]$, $[w]$, and $[i]$) produce an inferior performance compared to 2D systolic arrays. As discussed in Section 3.6.4, different dataflows exploit different levels of parallelism and data locality of the application, thus leading to such a performance disparity across CNNs. This result shows the necessity of auto-tuning to detect the suitable design for each CNN.

Low computation efficiency We notice a low computation efficiency on all three networks. For example, the best single systolic array on VGG16 achieves only a 73% DSP efficiency. This number further degrades to 35% and 19% on ResNet50 and MobileNetV2. In the next section, we perform an in-depth analysis to understand the reasons for this issue.

4.4.3.2 Understanding the Reasons of Low Computation Efficiency

Reason 1: The inflexibility of the single fixed array architecture to adapt to different CNN layers The first reason is the inflexibility of a single fixed systolic array to adapt to different CNN layers. Figure 4.10 compares the best layer-wise solutions (*free*) versus the single fixed systolic array (*fixed*) on VGG16. In the rest of this section, we name the array architecture generated by layer-optimal solutions as the *free* architecture since it omits the fixed array constraints across layers. The number above each bar in Figure 4.10 is the normalized throughput against the best throughput achieved by all the solutions; the number below it is the index of the dataflows following the order in Table 4.4. Layer-wise optimal solutions increase the throughput by 1.3× compared to the single fixed array on average. Particularly, we observe a large performance gap on the last 3 CONV layers (CONV11-13). Table 4.7 shows the detailed design information for the free and fixed

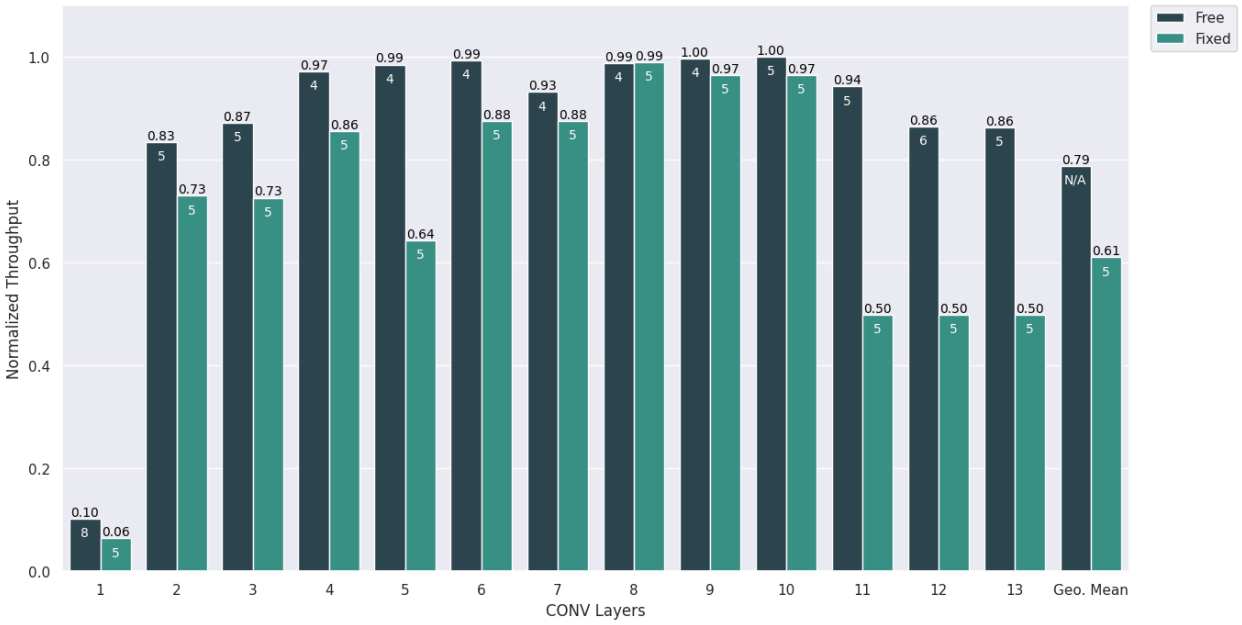


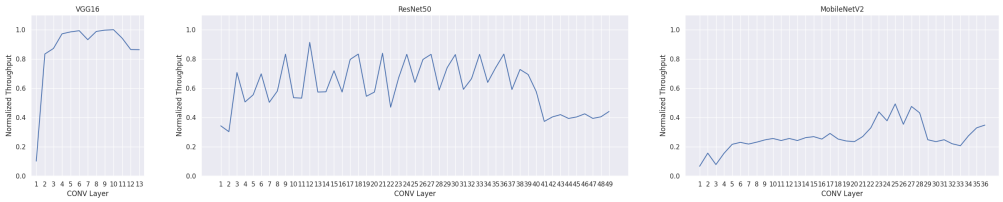
Figure 4.10: Comparison between the best layer-wise solutions (free) versus the single fixed array (fixed) on VGG16.

architectures on CONV11. Starting from CONV11, the height h and width w of feature maps are reduced to 14. The best first-level tiling factor for w (T_W1) on the free architecture is 14, which fully exploits the parallelism of this dimension without any redundant computation. However, the fixed architecture fails to adapt to this change. The tiling factor T_W1 on the fixed architecture is 28, which is selected to maximize the performance on the previous layers. This tiling factor introduces a 2× redundant computation on CONV11, leading to only a 48% DSP efficiency on this layer.

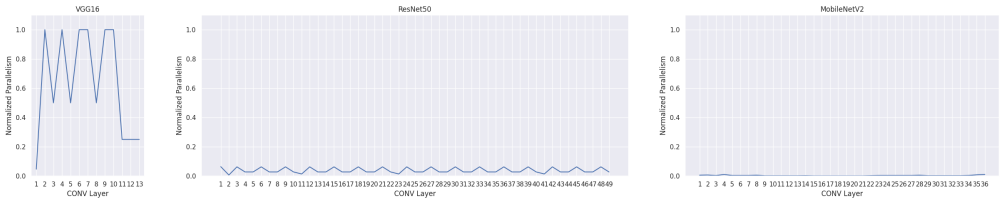
Table 4.7: Performance details of best free and fixed architectures on CONV11 of VGG16.

CONV Layer	Problem Sizes	Config.	Dataflow	Latency	DSP	DSP Eff.	BRAM	T_W1	T_O1
11	[I,O,H,W,P,Q]:	Free	[o,w]	1.0×	98%	91%	57%	14	180
	[512,512,14,14,3,3]	Fixed	[o,w]	1.9×	98%	48%	51%	28	90

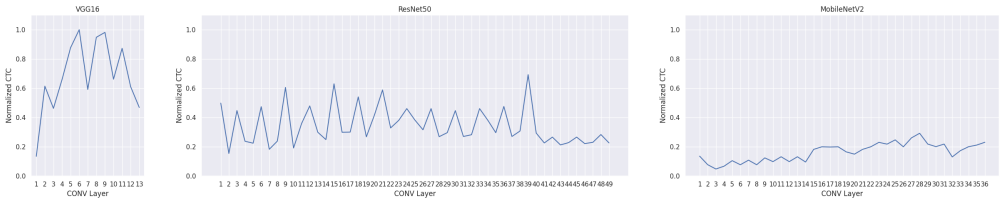
Reason 2: CONV layers with low parallelism and data locality The second reason is the low level of parallelism and data locality of certain CONV layers in these CNNs. Figure 4.11 presents the throughput, parallelism, and CTC of layer-optimal solutions in all three CNNs. All the numbers are normalized against the highest values across three CNNs.



(a) Throughput



(b) Parallelism



(c) CTC

Figure 4.11: Comparison of normalized throughput, parallelism, and CTC of layer-optimal solutions on VGG16, ResNet50, and MobileNetV2.

Layer-optimal solutions remove the restrictions of the fixed array architecture, and thus could adapt to each CONV layer more efficiently. However, we could still observe a large number of CONV layers with low throughput. The proportion of the low-throughput layers further increases on ResNet50 and MobileNetV2 compared to VGG16. This is due to the low level of parallelism and data locality of these CNNs. We could see from Figure 4.11 that CONV layers in ResNet50 and MobileNetV2 consist of less parallelism and a lower level of data locality than VGG16. This issue is more severe on MobileNetV2 compared to ResNet50. We will pick two layers from these networks for an in-depth analysis.

VGG16-CONV1: Figure 4.12 compares the throughput of different systolic arrays on the first CONV layer of VGG16 (VGG16-CONV1). All of the systolic arrays achieve low performance on this layer due to the low parallelism and data locality.

We select three dataflows $[h, i]$, $[o, h]$, and $[h, w]$ for a closer look. These three dataflows achieve the highest throughput among all the dataflows on this CONV layer. We omit

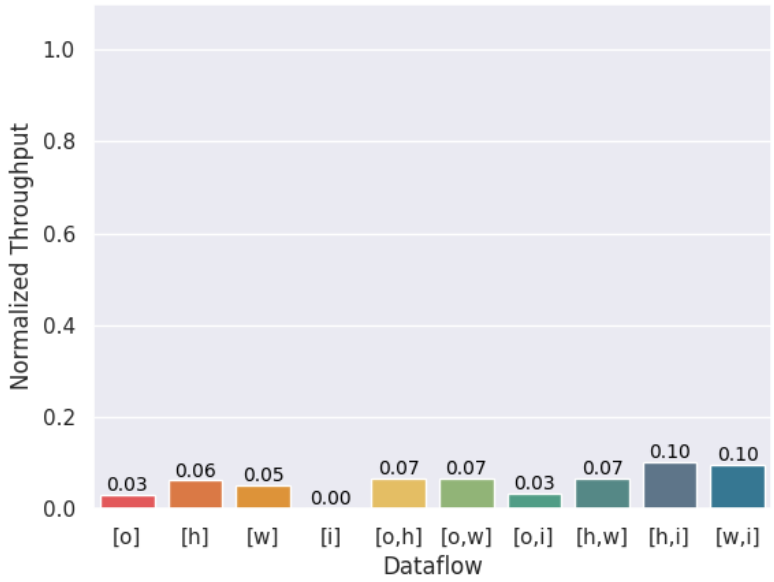


Figure 4.12: Throughput comparison of different systolic arrays on VGG16-CONV1.

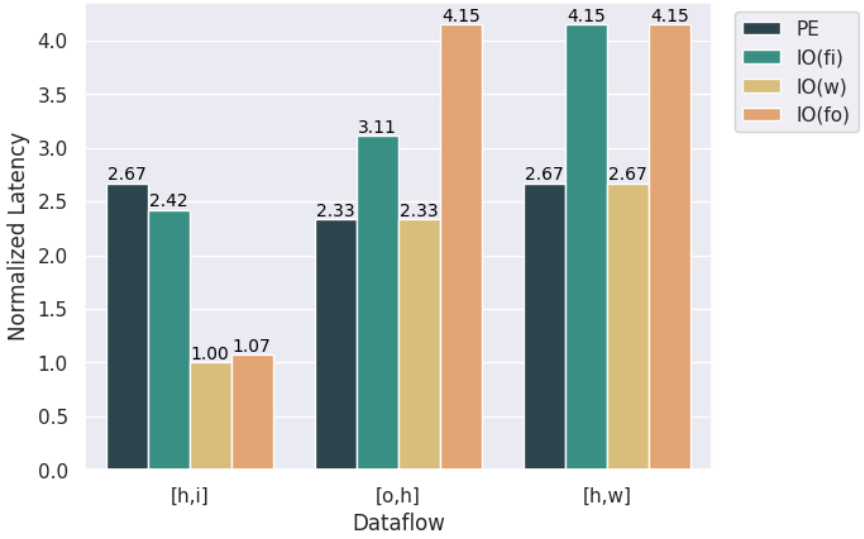


Figure 4.13: Latency of different hardware components in three selected systolic arrays on VGG16-CONV1.

the symmetric dataflows $[w, i]$ and $[o, w]$. Figure 4.13 compares the latency of different hardware modules inside the systolic array. Specially, we consider four hardware modules, the PEs for computation, the I/O modules for transferring the data of input feature maps fi, kernel weights w, and output feature maps fo. We name them *IO(fi)*, *IO(w)*, and *IO(fo)*,

respectively. As can be seen from this figure, the first dataflow $[h, i]$ is compute-bound. PE latency dominates latency of all the hardware components. However, this dataflow maps loop i to space dimensions which is only 3 in the first layer. The limited loop dimension (low parallelism) limits the performance of this design. For the other two systolic arrays, the I/O network dominates the overall latency, making them communication-bound. This is due to the low data locality of CONV1. As seen from Figure 4.11c, the first CONV layer of VGG16 has the least CTC ratio among all the layers, making it the most communication-bound layer across the network. Overall, we can see that the low parallelism and data locality lead to the low throughput of all the systolic arrays on this layer.

ResNet50-CONV41: In contrast to VGG16 where later CONV layers achieve a consistently high throughput, we observe a large performance disparity on ResNet50. Especially, the single-layer throughput further drops after CONV41. We take this layer (ResNet50-CONV41) for a closer look. Figure 4.14 compares the performance of all the systolic arrays on this layer. Similarly, we analyze the performance bottleneck on the three dataflows in Figure 4.15.

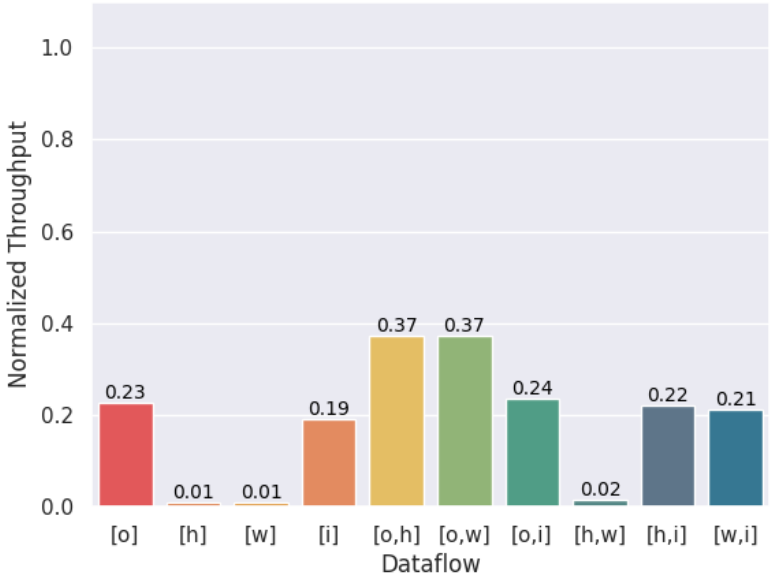


Figure 4.14: Throughput comparison of different systolic arrays on ResNet50-CONV41.

As shown in Figure 4.15, the dataflow $[o, h]$ is compute-bound. The size of h on CONV41 is reduced to only 7, which limits the maximal parallelism this design could exploit. The

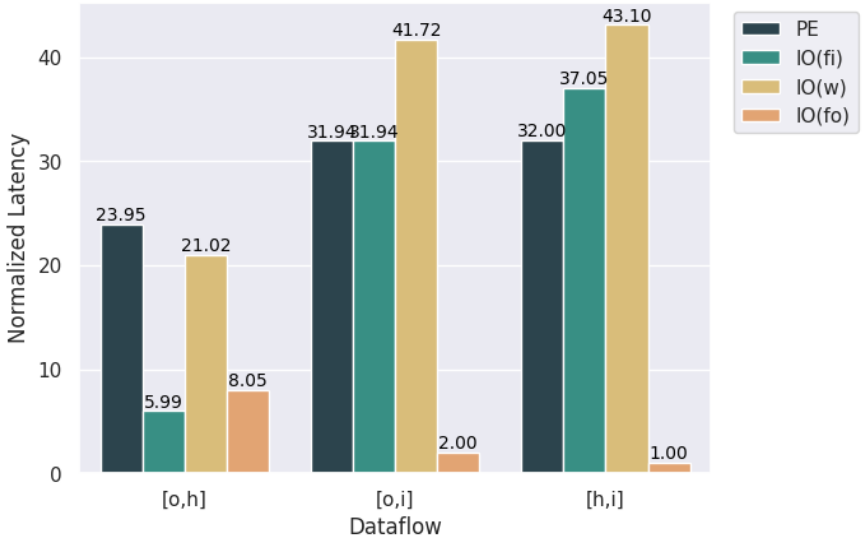


Figure 4.15: Latency of different hardware components in three systolic arrays on ResNet50-CONV41.

other two dataflows are limited by the I/O modules. However, in contrast to VGG16-CONV1 where designs are limited by the data transfer of output feature maps fo, designs on ResNet50-Conv41 are limited by the data transfer of the kernel weights w.

4.4.4 Summary

In this subsection, we performed a detailed architecture analysis on the performance of a single systolic array (Arch1) on three different CNNs. The evaluation results show that Arch1 faces a low computation efficiency on all three CNNs. This issue is more severe on ResNet50 and MobileNet50 in which most CONV layers are bound with limited parallelism and data locality.

We concluded two major reasons for this issue. First, we show that a single systolic array fails to adapt to different CONV layers with divergent performance characteristics. Second, the performance of a single systolic array is limited by the low level of data parallelism and the locality of certain CONV layers in these networks.

Based on the analysis above, we see the multi-array systolic array as a promising solution to overcome this challenge. First, with more than one systolic array implemented on-chip, a

multi-array architecture could customize each array to adapt to different layers. Second, we assign fewer resources to each systolic array in the multi-array system compared to a single array. This helps improve the computation efficiency without sacrificing the performance on CONV layers with limited parallelism. Besides, we could allocate on-chip buffers to store the intermediate results between different arrays which helps improve the performance of the communication-bound arrays. In the rest of this chapter, we will first study two different multi-array systems proposed by previous works. In addition to these two designs, we propose a more efficient multi-array architecture that achieves better performance than the previous work.

4.5 Architecture 2: Multi-Array with a One-Shot Schedule

4.5.1 Architecture Overview

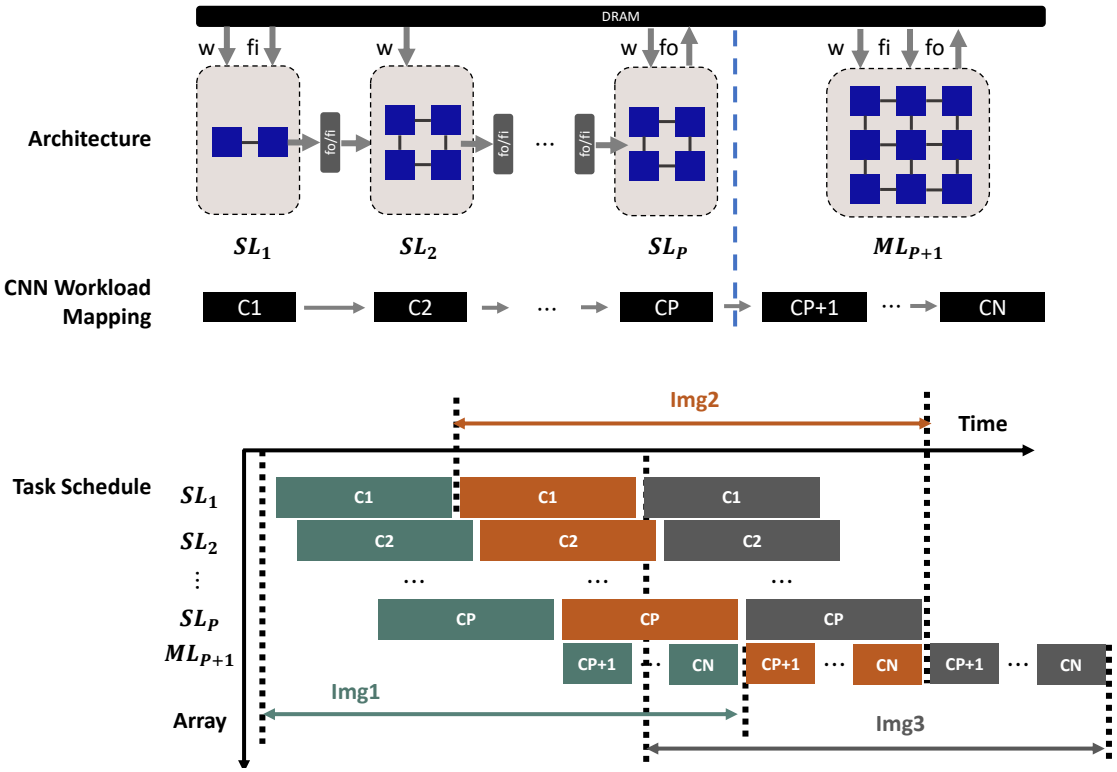


Figure 4.16: Architecture, CNN workload mapping, and task schedule of the multi-array design with a one-shot schedule (Arch2).

The first multi-array architecture we will investigate was proposed by the previous work DNNExplorer [ZYW20]. Figure 4.16 shows the architecture, workload mapping and task schedule of this design. We split the CNN network at layer P. Each CNN layer is mapped to a separate systolic array before the splitting point. We name these arrays as *single-layer (SL) arrays*. The other layers are grouped together and mapped to a single monolithic array, named the *multi-layer (ML) array*. Between the SL arrays, we insert the on-chip streaming buffers for storing the feature maps such that the second array could start the computation as soon as the first tile of data is generated from the previous array and loaded into the streaming buffers.

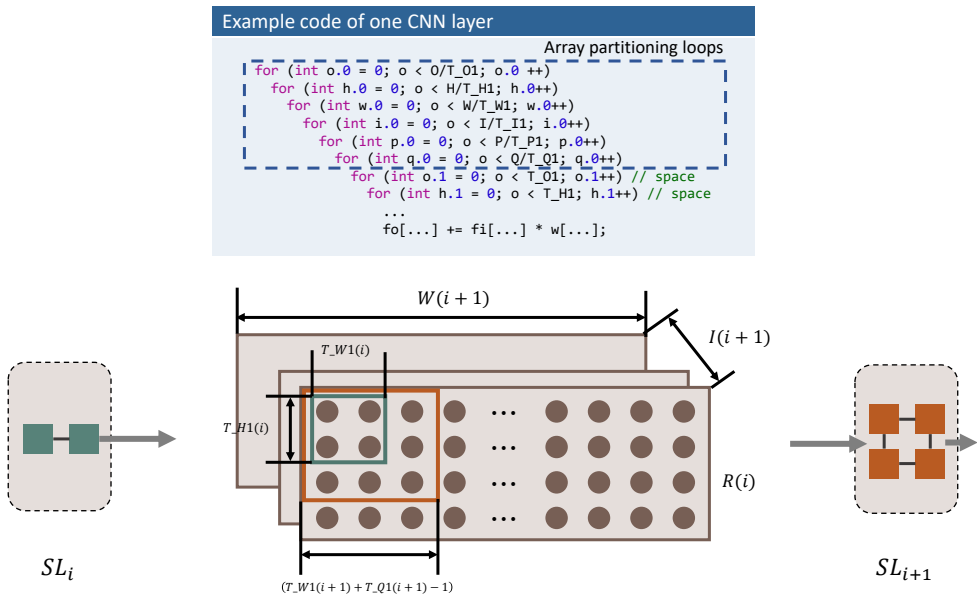


Figure 4.17: Architecture of the streaming buffers between two SL arrays SL_i and SL_{i+1} in Arch2.

Figure 4.17 shows the architecture of the streaming buffers between two SL arrays SL_i and SL_{i+1}. We use the same architecture proposed in TGPA [WLL18]. In this figure, we annotate the array partitioning tiling factors for loop h in the array SL_i as $T_H1(i)$. All the other tiling factors follow the same naming convention. As seen in the figure, the streaming buffer is composed of $I(i+1)$ line buffers. At each array partition, the first array SL_i produces a tile with the size of $T_H1(i) \times T_W1(i) \times T_O1(i)$. To start the first iteration on the second array SL_{i+1}, we require a tile of data with the size of

$(T_H1(i+1)+T_P1(i+1)-1)\times(T_W1(i+1)+T_Q1(i+1)-1)\times T_I1(i+1)$. As soon as such a data tile is ready in the streaming buffers, we can start the computation on SL_{i+1}. To support such a computation, we allocate the rows $R(i)$ and columns $C(i)$ of the line buffers following the equations below.

$$\begin{aligned} R(i) \quad &= max(T_H1(i+1)+T_P1(i+1)-1, \alpha \times T_H1(i)) \\ \text{s.t.} \quad &\alpha \times T_H1(i) \geq T_H1(i+1)+T_P1(i+1)-1 \\ &\alpha \in \mathbb{Z} \end{aligned} \tag{4.1}$$

$$C(i) = W(i+1) \tag{4.2}$$

The Odyssey framework provides the hardware performance models for each single systolic array. We extend these models to take into account the newly added hardware components. Details of the performance models of this architecture are discussed in the next two subsections.

4.5.1.1 Resource Model

We consider three types of resources in this architecture: BRAM, URAM, and DSP. Xilinx FPGAs provide two types of memory resources: BRAM and URAM. The basic unit of BRAM is configured with a datawidth of 18 bits and a depth of 1024 words. The basic unit of URAM is configured with a datawidth of 72 bits and a depth of 4096 words. All the on-chip buffers inside the systolic arrays are mapped to BRAMs, and the streaming buffers are mapped to URAMs. We estimate the three types of resources following the equations below.

$$U_{DSP} = \sum_{i \in \text{Arrays}} U_{DSP}(i) \tag{4.3}$$

$$U_{BRAM} = \sum_{i \in \text{Arrays}} U_{BRAM}(i) \tag{4.4}$$

$$U_{URAM} = \sum_{i \in \text{Streaming Buffers}} U_{URAM}(i) \tag{4.5}$$

The BRAM and DSP usage for each array are estimated by the resource models generated from the Odyssey framework. We estimate the URAM usage of the streaming buffer i between the array SL_i and SL_{i+1} as below.

$$\begin{aligned} U_{URAM}(i) &= \lceil \frac{DW(i) \times Bank(i)}{72} \rceil \lceil \frac{N(i)}{Bank(i) \times 4096} \rceil \\ N(i) &= R(i) \times C(i) \times I(i+1) \\ Bank(i) &= LCM(DP(SL_i), DP(SL_{i+1})) \end{aligned} \tag{4.6}$$

The rows and columns of the streaming buffers are computed by the Equations 4.1 and 4.2. The banks of the streaming buffers are computed as the least common multiple (LCM) of the data packing factors of the last-level I/O modules for transferring the feature maps inside two arrays SL_i and SL_{i+1}.

4.5.1.2 Latency and Throughput Model

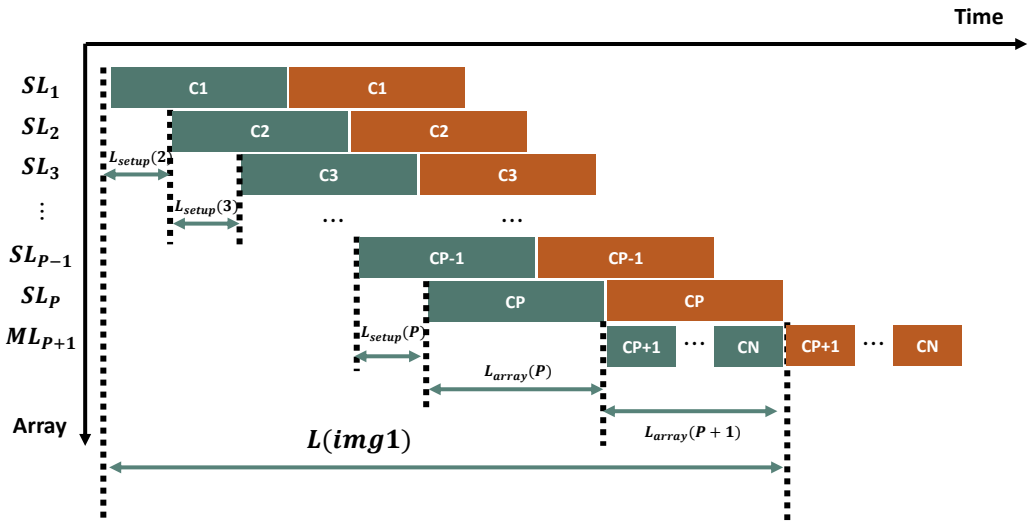


Figure 4.18: Task schedule of Arch2.

Figure 4.18 depicts the task schedule of this architecture. The latency of processing one image is composed of three parts: 1) the initial setup latency $L_{setup}(i)$ between the SL arrays

for filling up the streaming buffers, 2) the maximal array latency of all the SL arrays, and 3) the latency of the last ML array ML_{P+1}. The total latency is estimated as below:

$$\begin{aligned} L &= \sum_{2 \le i \le P} L_{setup}(i) + \max_{1 \le i \le P} L_{array}(i) + L_{array}(P+1) \\ L_{setup}(i) &= \frac{L_{array}(i-1)}{H(i-1)/R(i-1)} \end{aligned} \tag{4.7}$$

$R(i-1)$ denotes the rows of the streaming buffer between the SL array SL_{i-1} and SL_i, which can be computed by Equation 4.1.

Since we can compute the second image as long as each array finishes its computation on the first image, we estimate the throughput T of the entire design using the following equation:

$$T = \frac{1}{\max_{1 \le i \le P+1} L_{array}(i)} \tag{4.8}$$

4.5.1.3 Bandwidth Model

Odyssey provides a bandwidth model for each single systolic array. We estimate the bandwidth of each array as the sum of total off-chip data movement DW over the computation latency. By extending this model, we can estimate the bandwidth of Arch2.

$$BW = \frac{\sum_{i \in Arrays} DM(i)}{\max_{1 \le i \le P+1} L_{array}(i)} \tag{4.9}$$

We aggregate the off-chip data movement of all the systolic arrays and divide it by the maximal latency of all the arrays. When estimating the off-chip data movement of each systolic array, we exclude the data transferred through the on-chip streaming buffers.

4.5.2 Tuning Algorithms

In this section, we discuss the tuning algorithms for this architecture. There are two major challenges in this search task: 1) finding the ideal splitting point P, 2) allocating the resource for each array. The previous work DNNExplorer [ZYW20] built a two-level tuning framework for the search task. At the first level, they used the particle swarm optimization to sweep through different splitting points. For each splitting point, they implemented a greedy algorithm for resource allocation. The resource allocator starts with an initial resource allocation with heuristics, which will be iteratively adjusted by identifying the bottleneck design with the longest latency and sparing more resource for this design. We implement a similar two-level tuning framework for this architecture. Figure 4.19 illustrates the proposed tuning flow.

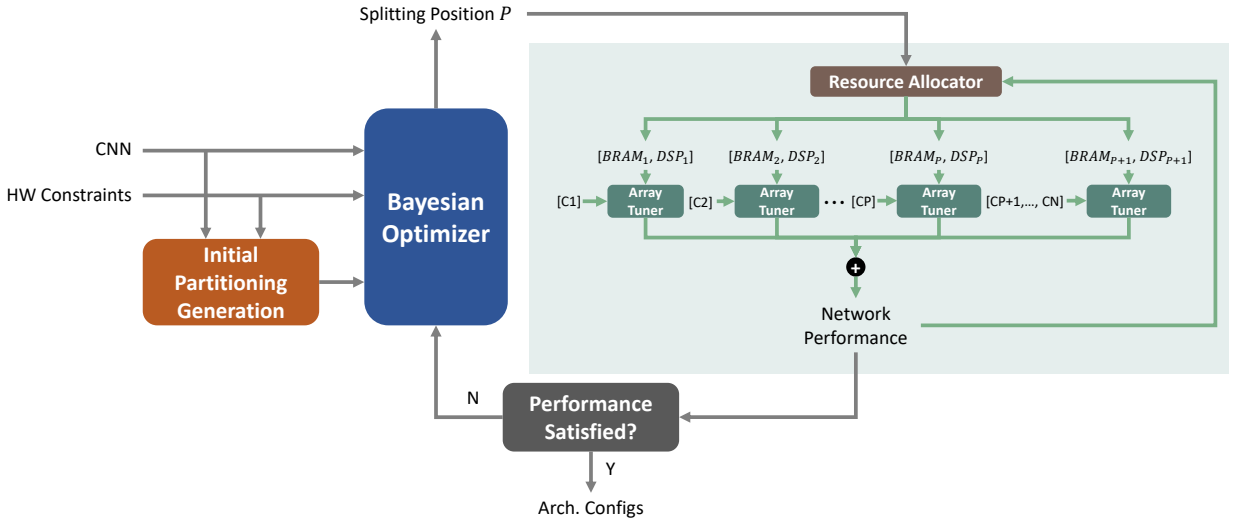


Figure 4.19: Overview of the tuning flow for Arch2.

We solve the first problem by using an ML-based auto-tuner. In this flow, we use the Bayesian optimization [Nog21] to identify the ideal splitting position P. The Bayesian optimizer is suitable for tuning problems with high evaluation costs and has shown its effectiveness in the previous design space exploration works [RHA17, NSK19].

A Bayesian optimizer can be configured to start the evaluation on a few pre-set design points that may deliver high performance. We implement a heuristic-based algorithm to rank different partitioning schemes (splitting points P) and select the top-3 partitioning schemes

as the initial design samples for evaluation by the Bayesian optimizer. The proposed ranking algorithm is described in Algorithm 5.

Algorithm 5: Initial partitioning scheme generation for Arch2.

Inputs : A CNN γ with N CONV layers, hardware constraints cst
Outputs: A list of initial partitioning schemes PS^*
Search the best single array architecture $Arch1$ for γ;
Initialize the partitioning scheme set $PS \leftarrow \emptyset$;
for *each splitting position P from 1 to N* **do**
    Estimate the design throughput T with the splitting point P:
    $T = throughput_est(\gamma, P, Arch1)$;
    Append the tuple (P, T) to PS: $PS \leftarrow PS \cup (P, T)$;
Select the top-3 partitioning schemes with the highest throughput from PS as PS^*;

Algorithm 6: Estimate the design throughput for Arch2.

Inputs : A CNN γ with N CONV layers, a splitting point P, a single array architecture $Arch1$
Outputs: An estimated throughput T of the generated multi-array architecture
Initialize the array latency list $L_{array} \leftarrow \emptyset$;
Obtain the maximal layer-wise DSP efficiency α from $Arch1$;
`/* Estimate the latency of single-workload (SL) arrays. */`
for *each SL array i from 1 to P* **do**
    Estimate the DSP efficiency of SL_i as α: $SL_i.dsp_eff = \alpha$;
    Estimate the throughput of array SL_i as: $SL_i.T = \frac{SL_i.ops}{Arch1.total_ops} \times SL_i.dsp_eff$;
    Estimate the latency of array SL_i : $SL_i.L = SL_i.ops/SL_i.T$;
    Append the array latency to L_{array}: $L_{array} \leftarrow L_{array} \cup SL_i.L$;
`/* Estimate the latency of the multi-workload (ML) array. */`
Estimate the DSP efficiency of ML_{P+1} as the average layer-wise DSP efficiency on Arch1:
$ML_{P+1}.dsp_eff = avg_{P+1 \leq i \leq N}(Arch1.layer[i].dsp_eff)$;
Estimate the throughput of array ML_{P+1}: $ML_{P+1}.T = \frac{MP_{P+1}.ops}{Arch1.total_ops} \times MP_{P+1}.dsp_eff$;
Estimate the latency of array ML_{P+1}: $ML_{P+1}.L = ML_{P+1}.ops/ML_{P+1}.T$;
Append the array latency to L_{array}: $L_{array} \leftarrow L_{array} \cup ML_{P+1}.L$;
Estimate the final design throughput as: $T = \frac{1}{\max(L_{array})}$;

In this algorithm, we first search a single systolic array (Arch1) for the target CNN. Then, we estimate the ideal throughput for the multi-array system with different splitting positions P using the single systolic array solution identified in the previous step. Algorithm 6 explains the algorithm for estimating the ideal throughput. There are two heuristics used here. First, we assume that each SL array can achieve the maximal layer-wise DSP efficiency. Second, we assume that in the ideal case, the throughput of each array is proportional to the total

computation operations of CONV layers mapped to this array such that each array could achieve a similar latency that maximizes the overall design throughput. This heuristic is explained by the equation below:

$$\frac{SL_1.ops}{SL_1.T} = \frac{SL_2.ops}{SL_2.T} = \cdots = \frac{SL_P.ops}{SL_P.T} = \frac{ML_{P+1}.ops}{ML_{P+1}.T} \tag{4.10}$$

We adjust the DSP efficiency of the ML array using the average DSP efficiency from the layer-wise solutions.

The experiment results show that this additional stage for generating initial partitioning candidates is highly effective in helping the Bayesian optimizer to quickly converge to good results. Figure 4.20 compares the convergence speed of two search methods with and without the initial design samples on VGG16. Each epoch is defined as one iteration in the Bayesian optimization examining one partitioning scheme. Without the initial samples, the Bayesian optimizer will randomly pick up one splitting point P to start. As seen from the figure, the Bayesian Optimizer converges in only one epoch with the provided initial samples, whereas it takes 3 epochs to converge with random initialization.

After fixing the splitting point P, the next step is to find a good resource allocation for each array to maximize the performance. We implement a resource allocator using a greedy algorithm as shown in Algorithm 7. At the first iteration, we allocate the DSPs to each array using the equation below.

$$\frac{SL_1.ops}{SL_1.DSP} = \frac{SL_2.ops}{SL_2.DSP} = \cdots \frac{SL_P.ops}{SL_P.DSP} = \frac{ML_{P+1}.ops}{ML_{P+1}.DSP} \tag{4.11}$$

This equation assumes that the array throughput is a monotonically increasing function of the amount of DSPs allocated to this array. To maximize the overall design throughput, all the array latency should be equal. Therefore, we allocate DSPs to each array in proportional to the total computation operations of the CONV layers mapped to each array. We partition the BRAMs to each array evenly.

After the initial allocation, we detect the bottleneck array with the longest latency and

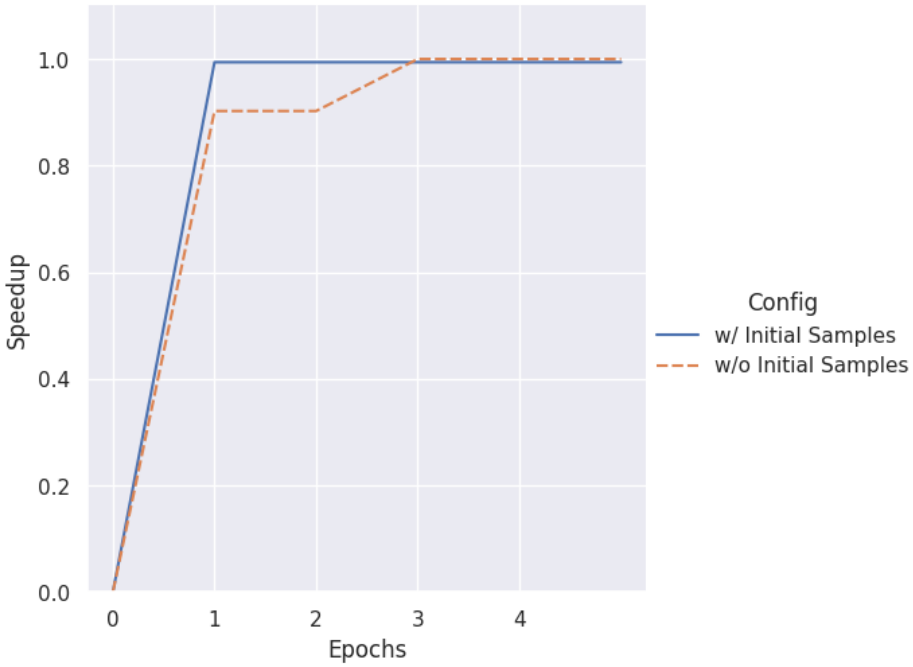


Figure 4.20: Comparison of the search performance of the Bayesian optimizer for Arch2 on VGG16 w/ and w/o the use of initial samples.

attempt to increase the resource allocation to this array until it is no longer the bottleneck. We accomplish this with two steps. In the first step, we try to allocate all the free resource to the bottleneck array. If the new array found with the updated resource constraints is still the bottleneck, we proceed to the second step. In the second step, we select a few of the fastest arrays and reduce the allocated resource to these designs by a threshold δ. The spared resource is added to the bottleneck array. The second step is repeated until the target design is no longer the bottleneck.

The proposed tuning flow can find a good design in less than 30 minutes for all three CNNs. In the next section, we present the detailed search results.

4.5.3 Evaluation Results

Table 4.8 compares the best designs found for the three networks against the single systolic array (Arch1). From the table, we see a throughput improvement on all three networks with a cost of increased latency. Arch2 achieves 1.3×, 1.6×, and 4.2× throughput improvement

Algorithm 7: Resource allocation.

Inputs : A list of arrays A, a CNN γ, hardware constraints cst
Outputs: A resource allocation $Alloc$
for *each array a in A* **do**
    Initialize the DSP allocation $Alloc[a].DSP = cst.DSP \times \frac{a.ops}{\gamma.total_ops}$;
    Initialize the BRAM allocation $Alloc[a].BRAM = cst.BRAM/\alpha$;
/* Function NeedFurtherAdjustment() is set to FALSE when: */
/* 1) The current design reaches the throughput target. */
/* 2) The search process times out. */
while *NeedFurtherAdjustment(Alloc)* **do**
    for *each array a in A* **do**
        Search the best single array architecture $Arch1(a)$ with the constraints $Alloc[a]$;
    Find the array a' with the longest latency;
    STEP 1: Allocate all the free resource to a';
    $Alloc[a'].DSP \mathrel{+}= cst.DSP - \sum_{a \in A} Alloc[a].DSP$;
    $Alloc[a'].BRAM \mathrel{+}= cst.BRAM - \sum_{a \in A} Alloc[a].BRAM$;
    Search the best single array configuration $Arch1(a')$ with the new constraints $Alloc[a']$;
    if *Array a' is still the bottleneck* **then**
        STEP 2: Find the top-k fastest arrays A';
        for *each array a in A'* **do**
            Reduce the resource allocated to a by δ;
            $Alloc[a].DSP = (1 - \delta) \times Alloc[a].DSP$;
            $Alloc[a].BRAM = (1 - \delta) \times Alloc[a].BRAM$;
            goto STEP 1;

Table 4.8: Performance comparison of Arch1 and Arch2 on different CNNs.

CNN	Architecture	Latency	Ratio	Throughput (GFLOPs)	Ratio	DSP	DSP Eff.	BRAM	URAM	BW(GB/s)	#Array
VGG16	Arch1	12,591,857	1.0×	731.3	1.0×	98%	73%	51%	0%	19.17	1
	Arch2	32,588,429	2.6×	929.8	1.3×	99%	91%	94%	31%	18.42	6
ResNet50	Arch1	5,908,327	1.0×	391.0	1.0×	98%	35%	57%	0%	35.08	1
	Arch2	37,736,962	6.4×	608.1	1.6×	99%	54%	97%	49%	22.04	23
MobileNetV2	Arch1	1,248,526	1.0×	128.7	1.0×	59%	19%	94%	0%	22.43	1
	Arch2	3,317,136	2.6×	540.1	4.2×	74%	62%	100%	30%	19.90	21

compared to Arch1 on VGG16, ResNet50, and MobileNetV2, respectively. Table 4.9 presents a layer-wise performance breakdown of Arch1 and Arch2 on VGG16.

We can see from this this table that the multi-array architecture adapts to each layer more effectively than the single monolithic array. Arch2 achieves 91% of the total DSP efficiency compared to 73% by Arch1, leading to 1.3× throughput improvment. In the meantime, Arch2 introduces a 2.6× longer latency than the Arch1. This is due to two reasons. First, due to less resource allocated to each array, the single array latency in Arch2 is longer than

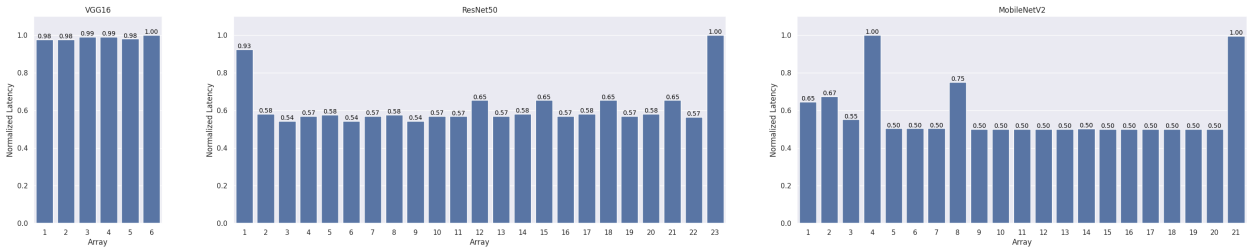
Table 4.9: Performance comparison between Arch1 and Arch2 on VGG16.

Arch1						Arch2					
Layer	Dataflow	Latency	DSP	DSP Eff.	BRAM	Layer	Dataflow	Latency	DSP	DSP Eff.	BRAM
1	[o,w]	829,291	98%	6%	51%	1	[h,i]	9,678,050	1%	100%	10%
2	[o,w]	1,553,475	98%	71%	51%	2	[o,w]	9,679,753	11%	100%	12%
3	[o,w]	780,913	98%	70%	51%	3	[h,i]	9,806,706	6%	98%	11%
4	[o,w]	1,324,569	98%	83%	51%	4	[o,h]	9,808,209	11%	98%	11%
5	[o,w]	881,429	98%	62%	51%	5	[o,h]	9,710,763	6%	99%	10%
6	[o,w]	1,295,687	98%	85%	51%	6-13	[o,w]	9,902,960	65%	88%	40%
7	[o,w]	1,295,687	98%	85%	51%						
8	[o,w]	573,175	98%	96%	51%						
9	[o,w]	1,175,233	98%	94%	51%						
10	[o,w]	1,175,233	98%	94%	51%						
11	[o,w]	569,055	98%	48%	51%						
12	[o,w]	569,055	98%	48%	51%						
13	[o,w]	569,055	98%	48%	51%						
Total	[o,w]	12,591,857	98%	73%	51%			32,588,429	99%	91%	94%

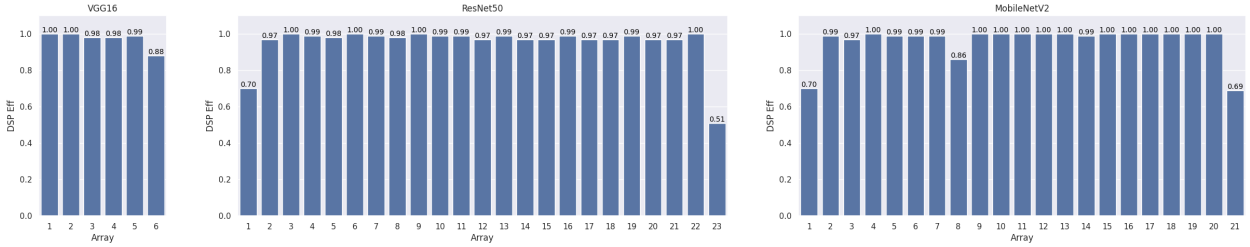
the latency of Arch1. Second, as introduced in Section 4.5.1.2, there are additional setup overheads between adjacent arrays which extend the overall design latency. These two factors lead to a longer single image processing latency than the single array architecture.

The analysis reveals the trade-offs when choosing between these two architectures. If the designer prioritizes the throughput when there is a stream of images to be processed, the multi-array architecture (Arch2) is a better choice. On the contrary, if the single image processing latency is the focus of the system, a single monolithic array (Arch1) is still a better choice regardless of the low computation efficiency.

Figure 4.21 shows the achieved latency and DSP efficiency of each array in Arch2 on all three networks. We see that the multi-array system achieves a balanced latency distribution and high DSP efficiency across three networks, leading to a throughput improvement over the single array baseline. Such an architecture is more effective on networks with less parallelism and data locality like ResNet50 and MobileNetV2.



(a) Latency



(b) DSP efficiency

Figure 4.21: Achieved latency and DSP efficiency of each array in Arch2 on VGG16, ResNet50, and MobileNetV2.

4.6 Architecture 3: Multi-Array with a Periodic Schedule

4.6.1 Architecture Overview

In this section, we evaluate a different multi-array architecture proposed by the previous work TGPA [WLL18]. Figure 4.22 shows the detailed architecture, workload mapping, and task schedule of this design. CNN layers are scheduled periodically onto P arrays in multiple iterations. In the rest of this section, we name one iteration in which all the P arrays execute one convolutional layer on each array as a *round*. At each round, the arrays execute a group of P consecutive layers. We insert streaming buffers similar to the previous architecture such that the second array can start computation as soon as the first tile of data is ready in the buffer. We place a synchronization barrier at the end of each round. This is necessary since the first array in the next round is dependent on the outputs produced by the last array in the previous round. The architecture of streaming buffers is the same as the one shown in Section 4.5.1. Next, we explain the details about the performance models.

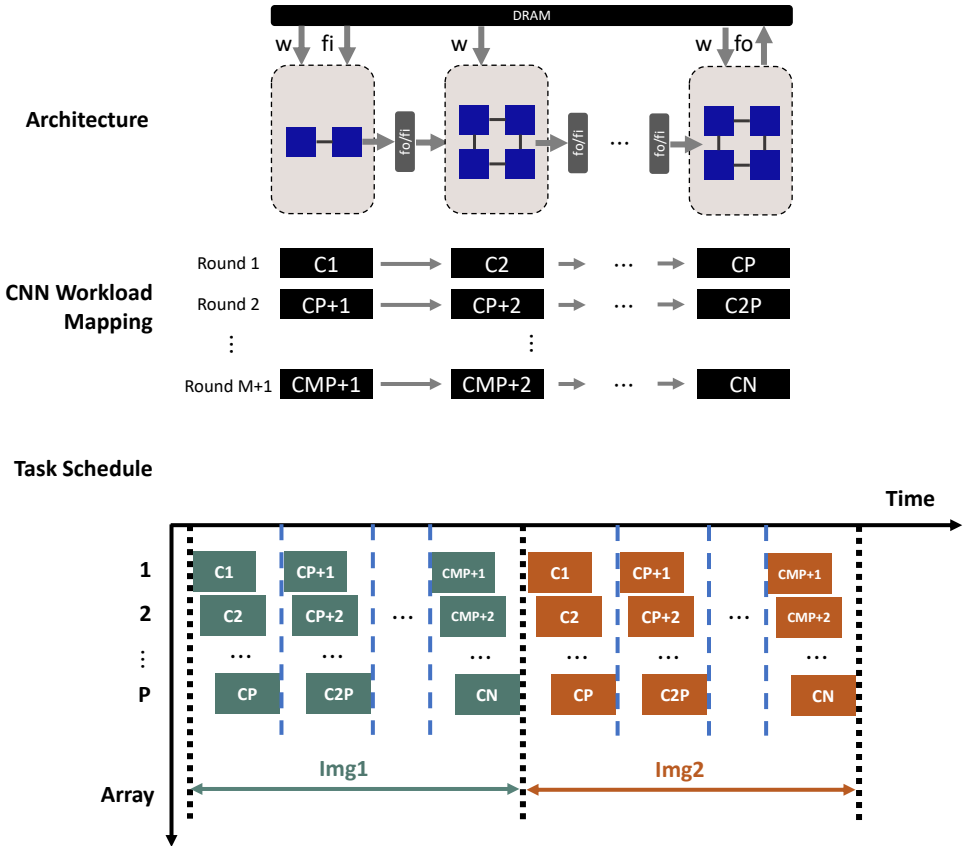


Figure 4.22: Architecture, CNN workload mapping, and task schedule of the multi-array design with a periodic schedule (Arch3).

4.6.1.1 Resource Model

The resource model is similar to the previous design. We map the streaming buffers to URAMs, and aggregate the BRAMs and DSPs of all the arrays using the models from the Odyssey framework.

4.6.1.2 Latency and Throughput Model

Figure 4.23 depicts the detailed task schedule of this architecture. The latency of processing one image is computed as a sum over the latency of each scheduling round $L_{round}(r)$. The latency of each scheduling round r is composed of two parts: 1) the initial setup latency $L_{setup}(r, i)$ between the adjacent arrays, and 2) the maximal array latency of all the arrays in the current round. Equations below show the latency model for this design.

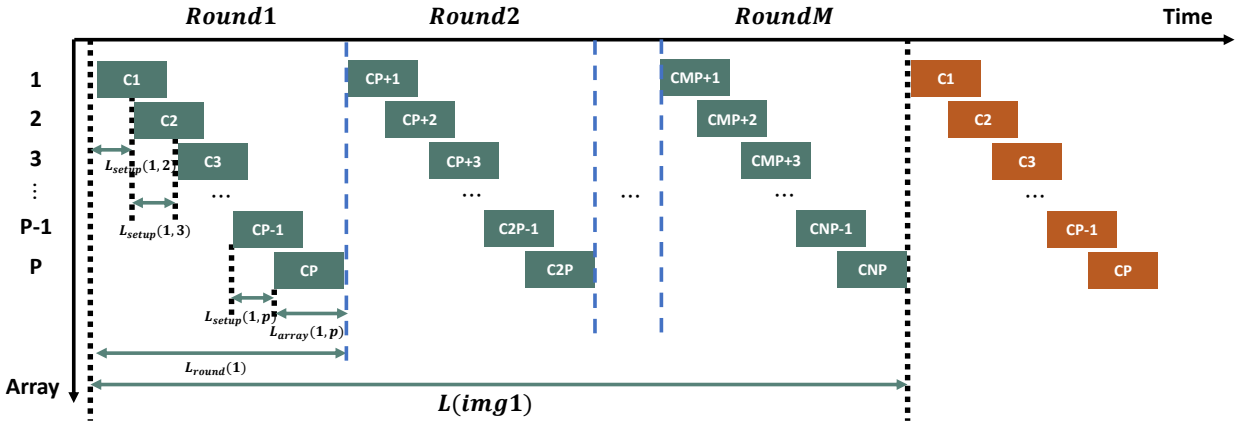


Figure 4.23: Task schedule of Arch3.

$$
\begin{aligned}
L &= \sum_{1 \le r \le ceil(N/P)} L_{round}(r) \\
L_{round}(r) &= \sum_{2 \le i \le P} L_{setup}(r,i) + \max_{1 \le i \le P} L_{array}(r,i) \\
L_{setup}(r,i) &= \frac{L_{array}(r,i-1)}{H(r,i-1)/R(r,i-1)}
\end{aligned}
\tag{4.12}
$$

$H(r, i-1)$ denotes the height of the output feature map in the CONV layer mapped to the array $i-1$ at the round r. $R(r, i-1)$ denotes the rows of the streaming buffer between the array $i-1$ and i at the round r, which can be computed by Equation 4.1.

As the second image can only start after the entire computation of the first image is completed, we estimate the throughput T as $T = 1/L$.

4.6.1.3 Bandwidth Model

The bandwidth of this architecture is estimated as the maximal bandwidth across all the rounds. At each round, we use the same bandwidth model for Arch2. We aggregate the total off-chip data movement of all the arrays in this round, and divide it by the round latency. Data that are transferred through the on-chip streaming buffers will be excluded from the total off-chip data movement.

$$
\begin{aligned}
BW &= \max_{1 \le r \le ceil(N/P)} BW(r) \\
BW(r) &= \frac{\sum_{i \in Arrays} DM(r,i)}{L_{round}(r)}
\end{aligned}
\tag{4.13}
$$

4.6.2 Tuning Algorithms

The tuning algorithm for this architecture is similar to the algorithm proposed for the last design. There are two major choices for this architecture: the number of systolic arrays P and the resource allocation. We use the same Bayesian optimizer to explore a different number of arrays P, and the same resource allocator to iteratively find the best resource partition that balances the latency of different arrays.

Algorithm 8: Initial partitioning scheme generation for Arch3.

Inputs : A CNN γ with N CONV layers, hardware constraints *cst*
Outputs: A list of initial partitioning schemes PS^*
Search the best single array architecture $Arch1$ for γ;
Initialize the partitioning scheme set $PS \leftarrow \emptyset$;
for *number of arrays P from 2 to N* **do**
    Estimate the design throughput T with the #array P:
     $T = throughput_est(\gamma, P, Arch1)$;
    Append the tuple (P,T) to PS: $PS \leftarrow PS \cup (P,T)$;
Select the top-3 partitioning schemes with the highest throughput from PS as PS^*;

Algorithm 9: Estimate the design throughput for Arch3.

Inputs : A CNN γ with N CONV layers, #array P, a single array configuration $Arch1$
Outputs: An estimated throughput T of the generated multi-array architecture
Initialize the array latency list $L_{array} \leftarrow \emptyset$;
for *array i from 1 to P* **do**
    Obtain the list of CONV layers mapped to array Arr_i: $W(i)$;
    Estimate the DSP efficiency of Arr_i as the average layer-wise DSP efficiency on Arch1:
     $Arr_i.dsp_eff = avg_{l \in W(i)}(Arch1.layer[l].dsp_eff)$;
    Estimate the throughput of array Arr_i: $Arr_i.T = \frac{Arr_i.ops}{Arch1.total_ops} \times Arr_i.dsp_eff$;
    for *each CONV layer l in W(i)* **do**
        Estimate the layer latency on array Arr_i as:
         $L_{array}(l,i) = Arch1.layer[l].ops / Arr_i.T$;
Estimate the final design latency L using Equation 4.12 with L_{array};
Estimate the final design throughput as: $T = 1/L$;

For the Bayesian optimizer, we also provide a few profitable design points for initialization.

The algorithm of this step is explained in Algorithm 8. We start with finding the single array architecture (Arch1) for the target CNN. Then, for each possible number of arrays P, the proposed algorithm estimates the throughput for each array using the layer-wise solutions from Arch1. Algorithm 9 explains how we estimate the design throughput. We assume the ideal throughput of each array is proportional to the total computation operations of CONV layers mapped to this array. The ideal throughput is then adjusted by the average DSP efficiency achieved on the single array architecture Arch1.

4.6.3 Evaluation Results

Table 4.10: Performance comparison of Arch1 and Arch3 on different CNNs.

CNN	Architecture	Latency	Ratio	Throughput (GFLOPs)	Ratio	DSP	DSP Eff.	BRAM	URAM	BW(GB/s)	#Array
VGG16	Arch1	12,877,185	1.0×	731.3	1.0×	98%	73%	51%	0%	19.17	1
	Arch3	18,242,463	1.4×	504.8	0.7×	98%	50%	55%	7%	8.40	2
ResNet50	Arch1	5,908,327	1.0×	391.0	1.0×	98%	35%	57%	0%	35.08	1
	Arch3	8,415,616	1.4×	274.5	0.7×	98%	25%	85%	6%	11.47	2
MobileNetV2	Arch1	1,264,248	1.0×	128.7	1.0×	59%	19%	94%	0%	22.43	1
	Arch3	947,733	0.7×	171.7	1.3×	88%	17%	91%	1%	14.09	2

Table 4.10 compares the best Arch2 designs found for the three CNNs against the single systolic array (Arch1). From the table, we see that Arch3 achieves a worse performance in both latency and throughput than Arch1 on VGG16 and ResNet50, and achieves a performance improvement on MobileNetV2. To understand this result, we conduct a detailed performance breakdown of Arch3 on VGG16 in Table 4.11.

Table 4.11: Performance breakdown of Arch3 on VGG16.

	Array 1			Array 2			System		
	DSP	DSP Eff.	BRAM	DSP	DSP Eff.	BRAM	DSP	DSP Eff.	BRAM
	42%	88%	21%	56%	77%	34%	98%	50%	55%
Round	Layer	Latency	DSP Eff.	Layer	Latency	DSP Eff.	Round Latency	Setup Latency	Setup Overheads
1	1	817,766	15%	2	2,584,681	75%	2,599,283	14,602	1%
2	3	1,294,438	98%	4	2,583,873	75%	2,676,332	92,459	3%
3	5	1,359,337	93%	6	2,071,825	93%	2,751,493	679,668	25%
4	7	2,846,565	89%	8	1,051,473	92%	3,795,420	948,855	25%
5	9	2,586,057	98%	10	2,427,441	79%	3,879,085	1,293,028	33%
6	11	648,322	98%	12	1,244,202	39%	1,892,525	648,322	34%
7	13	648,322	98%						

We uncover two issues in this table. First, the second array (Array 2) in Arch3 faces the similar challenge with the single systolic array in that it lacks the flexibility to adapt to

different CONV layers, leading to a relatively low DSP efficiency (77%).

Second, as the two arrays need to be synchronized at each round in this architecture, the setup latency between the adjacent arrays needs to be taken into account, which introduces non-negligible overheads to the total latency. We can see from this table that the setup latency overheads increase in the later rounds. At Round 6, the setup latency consumes 34% of the total round latency. Table 4.12 shows the layer parameters of CONV 11 and 12 in the Round 6, and the tiling factors chosen for each array.

Table 4.12: Layer parameters and the first-level tiling factors for loop h and w in CONV 11 and 12.

Arrays	CONV Layer	I	O	H/W	P/Q	T_H1	T_W1
1	11	512	512	14	3	7	14
2	12	512	512	14	3	7	14

Due to the small feature map size W, the best tiling factors of w found for these two layers T_W1 equal to the input feature map size W. The chosen tiling factor for h is $T_H1 = 7$. However, to initiate the computation of Array 2, a data tile with a size of $T_H1 + P - 1 = 9$ is required. Due to this limitation, the second array has to wait until the first array finishes the entire computation. Therefore, the setup latency for this round takes up to $648322/189252 = 34\%$ of the total round latency.

Setup overheads can be amortized with a larger batch size. So far, we only consider the batch size as one such that one image is processed each time. When there are multiple images to be processed in one batch, in every round, each array could spend more time on the computation and only wait for the initial data for the first image. Table 4.13 adds the performance results for Arch3 with a batch size of 16.

As seen from this table, a large batch size helps amortizes the setup overheads. In addition to MobileNetV2, we see a throughput improvement on ResNet50 with the large batch size. However, using a large batch size introduces an even longer processing latency for each image. For example, on MobileNetV2, while we achieve a 2.4× throughput improvement over the single array baseline, the single image latency increases by 6.6×. As a conclusion, Arch3 is

Table 4.13: Performance comparison of Arch1 and Arch3 on different CNNs.

CNN	Architecture	Latency	Ratio	Throughput (GFLOPs)	Ratio	DSP	DSP Eff.	BRAM	URAM	BW(GB/s)	#Array	Batch Size
VGG16	Arch1	12,877,185	1.0×	731.3	1.0×	98%	73%	51%	0%	19.17	1	1
	Arch3	18,242,463	1.4×	504.8	0.7×	98%	50%	55%	7%	8.4	2	1
	Arch3	214,377,339	17.0×	687.2	0.9×	97%	69%	47%	6%	12.01	2	16
ResNet50	Arch1	5,908,327	1.0×	391.0	1.0×	98%	35%	57%	0%	35.08	1	1
	Arch3	8,415,616	1.4×	274.5	0.7×	98%	25%	85%	6%	11.47	2	1
	Arch3	78,496,508	13.3×	470.9	1.2×	99%	42%	97%	18%	24.03	7	16
MobileNetV2	Arch1	1,264,248	1.0×	128.7	1.0×	59%	19%	94%	0%	22.43	1	1
	Arch3	947,733	0.7×	171.7	1.3×	88%	17%	91%	1%	14.09	2	1
	Arch3	8,372,038	6.6×	310.9	2.4×	96%	28%	97%	3%	19.59	4	16

only beneficial when throughput is a priority of the system and using a large batch size is permitted.

4.7 Architecture 4: Multi-Array with a One-Shot Schedule and Adaptive Partitioning

4.7.1 Architecture Overview

The multi-array architecture with a one-shot schedule (Arch2) has been demonstrated to be effective in helping improve the design throughput. However, the proposed architecture by DNNExplorer [ZYW20] is limited because it only maps the first few layers into separate arrays. This approach is sufficient for networks like VGG16 which is rich in parallelism and data locality on most CONV layers. However, it is insufficient for networks such as ResNet50 and MobileNetV2 which contain more computation and communication-bound CONV layers. As a result, we see in Figure 4.21 that the last systolic array on ResNet50 and MobileNetV2 only manages 51% and 69% DSP efficiency and remains the performance bottleneck among all the arrays.

A natural way to solve this problem is to map the last few layers to separate systolic arrays as well. In this section, we present an improved architecture to Arch2 that performs an adaptive partitioning of CNNs that takes into account the performance characteristics of CONV layers. We name this architecture *Arch4* in the following text.

Figure 4.24 shows the architecture, workload mapping, and task schedule of Arch4. In this architecture, we insert multiple splitting points to the network. All the layers in-between

the splitting points are mapped to a single systolic array. The goal is to group layers with similar characteristics together and maximize the overall throughput.

There are two major differences in the Arch4 architecture and the previous Arch2. First, we are not constrained by the fixed partitioning scheme which only maps the first few layers to separate arrays and groups the rest into a single array. The Arch4 architecture allows for grouping together any number of layers. As shown in Figure 4.24, we choose to map the first two CONV layers to a single array ML_1, and the later two layers to two separate arrays SL_2 and SL_3. This flexible partitioning scheme adapts to the performance disparity of CNN layers more effectively.

Second, in Arch2, only if two adjacent arrays are single-workload (SL) arrays, an on-chip streaming buffer is inserted to save the off-chip communication. In this new Arch4 architecture, we consider allocating a global buffer for each multi-workload (ML) array to store the intermediate results as well. As shown in Figure 4.24, we introduce global buffers for ML_1 and ML_M. In this architecture, the global buffers and on-chip streaming buffers are all mapped to URAMs.

4.7.2 Tuning Algorithms

We use a similar tuning framework as Arch2. The difference is that we will generate all the profitable partitioning schemes in advance and only explore these schemes using the Bayesian optimizer. Algorithm 10 explains our algorithm for generating all the partitioning scheme candidates.

We start by finding the best single fixed systolic array solution $Arch1$ for the target CNN with the hardware constraints cst. Then, we choose a set of initial splitting points P_0 using the following method. We maintain a sliding window W that stores the layer-wise DSP efficiency on $Arch1$ for a few adjacent layers. If the DSP efficiency of the next layer is greater than the maximal value of W by α, or lower than the minimal value of W by β, we choose to insert a splitting point here since the layer exhibits a much different characteristic than the previous layers.

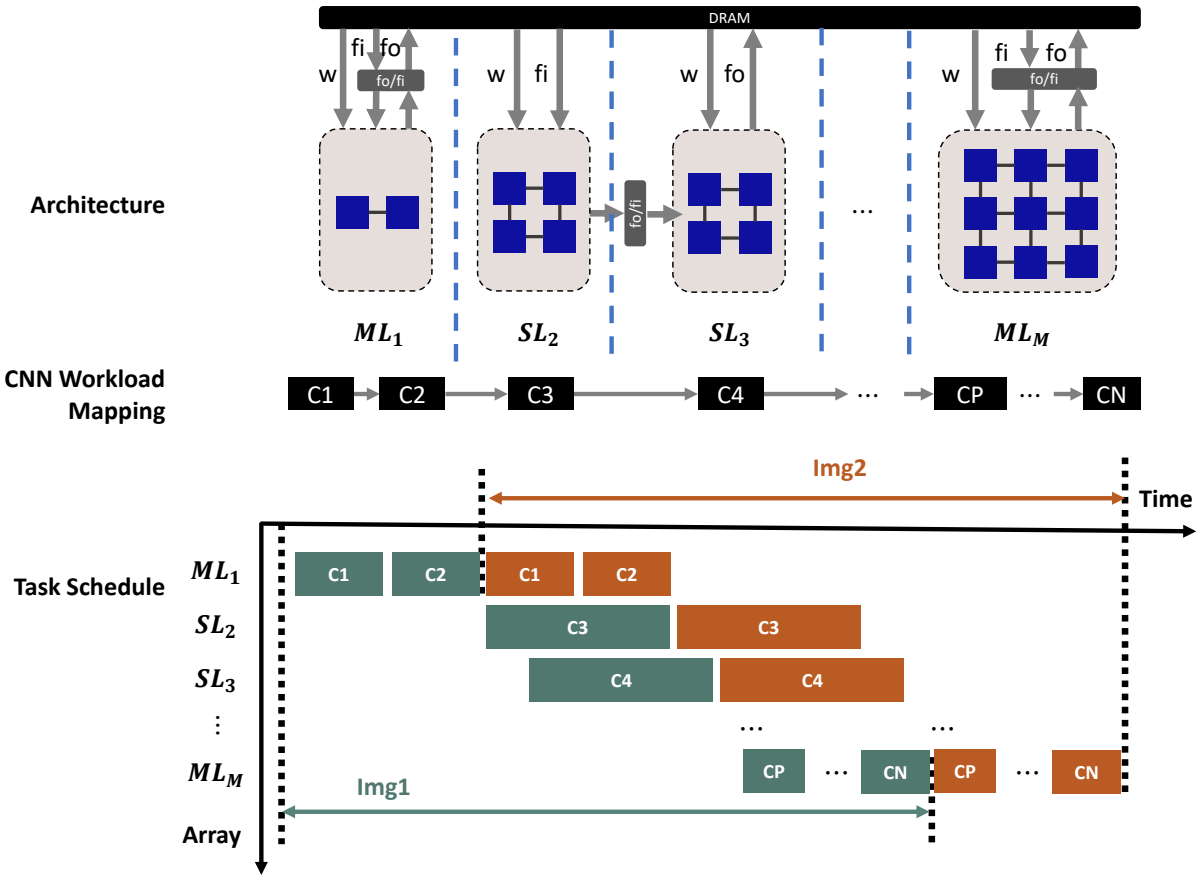


Figure 4.24: Architecture, CNN workload mapping, and task schedule of multi-array design with a one-short schedule and adaptive partitioning (Arch4).

After collecting the initial splitting points, we gradually split or merge more layers to generate different partitioning schemes that cover a different number of arrays. When splitting the layers, we select an array to split if it has the highest standard deviation of the layer-wise DSP efficiency among all the arrays. This implies the CONV layer performance is highly divergent on this array and splitting this array to multiple smaller ones could improve the performance. We enumerate all different layer positions and select one layer l that leads to the minimal standard deviation of the DSP efficiency on the rest layers. Then we split the target array into three separate arrays: the first one contains all the CONV layers before l, the second has the single-workload array with the layer l, and the third one consists of the rest of the layers after l.

On the other hand, when merging the layers, we choose to combine arrays with a similar performance, which we identify by computing the average DSP efficiency on each array and

Algorithm 10: Partitioning scheme generation for Arch4.

Inputs : A CNN γ with N CONV layers, hardware constraints cst
Outputs: A list of partitioning schemes PS, a list of initial partitioning schemes PS^*
Search the best single array architecture $Arch1$ for γ;
Initialize the initial splitting position set $P_0 \leftarrow \emptyset$;
Initialize the DSP efficiency window $W \leftarrow \emptyset$;
Append the first two CONV layers into the window: $W \leftarrow [Arch1.layer[0].dsp_eff, Arch1.layer[1].dsp_eff]$;
for *each CONV layer l from 2 to $N-1$* **do**
  /* Split the layers when the DSP efficiency on the current layer l is much larger or smaller than the layers in W. */
  if $Arch1.layer[l].dsp_eff > \alpha \times max(W)$ *or* $Arch1.layer[l].dsp_eff < \beta \times min(W)$ **then**
    Split the layers at the current position: $P_0 \leftarrow P_0 \cup l$;
    Reset the window W: $W \leftarrow [Arch1.layer[l].dsp_eff]$;
  else
    Append the current layer into the window: $W \leftarrow W \cup Arch1.layer[l].dsp_eff$;
for *the number of arrays n from 2 to N* **do**
  Duplicate the splitting position set P_0: $P \leftarrow P_0$;
  if $len(P) + 1 > n$ **then**
    /* Iteratively merge two adjacent arrays with a close average DSP efficiency. */
    while $len(P) + 1 > n$ **do**
      Generate the layer partitioning scheme ps from the splitting points P;
      Initialize the array average DSP efficiency list $D \leftarrow \emptyset$;
      for *each array workload partition ps_i in ps* **do**
        Compute the average DSP efficiency of all the layers in ps_i:
        $D \leftarrow D \cup (\sum_{l \in ps_i} Arch1.layer[l].dsp_eff)/len(ps_i)$;
      Initialize the DSP efficiency ratio list $DC \leftarrow \emptyset$;
      for *each array m from 1 to $len(P)$* **do**
        $DC \leftarrow DC \cup D[m]/D[m-1]$;
      Select the array m^* with the least relative change in DC;
      Merge array m^* with array $m^* - 1$ by removing the splitting point between them: $del\, P[m^* - 1]$;
  else
    /* Iteratively split one array with the largest standard deviation of the layer-wise DSP efficiency. */
    while $len(P) + 1 < n$ **do**
      Generate the layer partitioning scheme ps from the splitting points P;
      Initialize the array DSP efficiency standard deviation list $D \leftarrow \emptyset$;
      for *each array workload partition ps_i in ps* **do**
        Compute the standard deviation of the DSP efficiency of all the layers in ps_i:
        $D \leftarrow D \cup stdev([Arch1.layer[l].dsp_eff \text{ for } l \text{ in } ps_i])$;
      Select the array m^* with the largest stdev in D;
      Initialize the split array std list $D_{tmp} \leftarrow \emptyset$;
      for *each layer l in the partitioned layers for array m^* ($ps[m^*]$)* **do**
        Compute the standard deviation of the DSP efficiency d of the rest layers when excluding the layer l: $D_{tmp} \leftarrow D_{tmp} \cup stdev([Arch1.layer[i].dsp_eff \text{ for } l \text{ in } ps[m^*]/l])$;
      Split at the layer l with the least stdev value in D_{tmp} by adding two splitting points at l and $l+1$ into P: $P \leftarrow P \cup l \cup l+1$;
  Estimate the design throughput T with the splitting points P: $T = throughput_est(\gamma, P, Arch1)$;
  Append the tuple (P, T) to PS: $PS \leftarrow PS \cup (P, T)$;
Select the top-3 partitioning schemes with the highest throughput from PS as PS^*;

select two adjacent arrays with minimal differences. This guarantees that the merged arrays compute layers similar to each other. Note that the splitting and merging can be theoretically replaced by a more efficient dynamic programming algorithm. We leave the exploration of using dynamic programming as the future work.

After generating all the partitioning candidates for the different number of arrays. We use the same algorithm proposed in Algorithm 6 to estimate the ideal throughput for each partitioning scheme and select the top-3 schemes with the highest throughput as the initial points for probing by the Bayesian optimizer.

4.7.3 Evaluation Results

Finally, Table 4.14 presents the detailed performance results of Arch4 and compares Arch4 to all three architectures discussed in the previous sections. Figure 4.26 compares the latency, throughput, and DSP efficiency of all four architectures on three CNNs for when the batch size is one. Figure 4.25 presents the partitioning scheme of the best designs found for three CNNs.

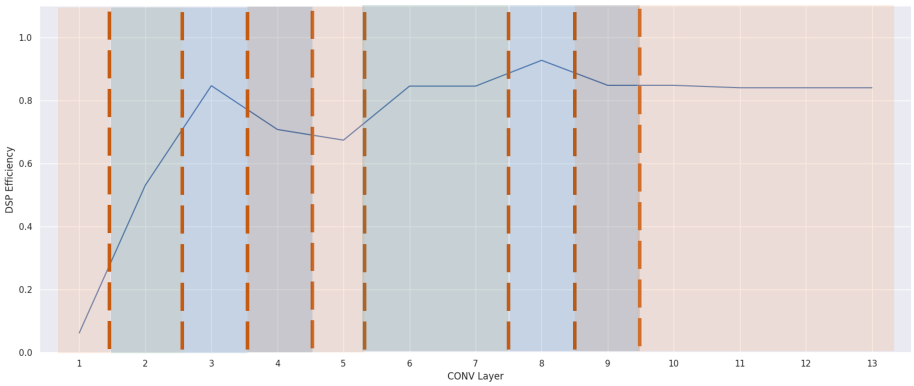
Table 4.14: Performance comparison of Arch1, Arch2, Arch3, and Arch4 on different CNNs.

CNN	Architecture	Latency	Ratio	Throughput (GFLOPs)	Ratio	DSP	DSP Eff.	BRAM	URAM	BW(GB/s)	#Array	Batch Size
VGG16	Arch1	12,877,185	1.0×	731.3	1.0×	98%	73%	51%	0%	19.17	1	1
	Arch2	32,588,429	2.6×	929.8	1.3×	99%	91%	94%	31%	18.42	6	1
	Arch3	18,242,463	1.4×	504.8	0.7×	98%	50%	55%	7%	8.4	2	1
	Arch3	214,377,339	17.0×	687.2	0.9×	97%	69%	47%	6%	12.01	2	16
	Arch4	56,046,853	4.5×	926.1	1.3×	100%	90%	75%	24%	16.04	9	1
ResNet50	Arch1	5,908,327	1.0×	391.0	1.0×	98%	35%	57%	0%	35.08	1	1
	Arch2	37,736,962	6.4×	608.1	1.6×	99%	54%	97%	49%	22.04	23	1
	Arch3	8,415,616	1.4×	274.5	0.7×	98%	25%	85%	6%	11.47	2	1
	Arch3	78,496,508	13.3×	470.9	1.2×	99%	42%	97%	18%	24.03	7	16
	Arch4	15,566,522	2.6×	1021.0	2.6×	99%	91%	89%	52%	22.22	7	1
MobileNetV2	Arch1	1,264,248	1.0×	128.7	1.0×	59%	19%	94%	0%	22.43	1	1
	Arch2	3,317,136	2.6×	540.1	4.2×	74%	62%	100%	30%	19.90	21	1
	Arch3	947,733	0.7×	171.7	1.3×	88%	17%	91%	1%	14.09	2	1
	Arch3	8,372,038	6.6×	310.9	2.4×	96%	28%	97%	3%	19.59	4	16
	Arch4	2,793,253	2.2×	747.9	5.8×	86%	74%	99%	71%	25.36	15	1

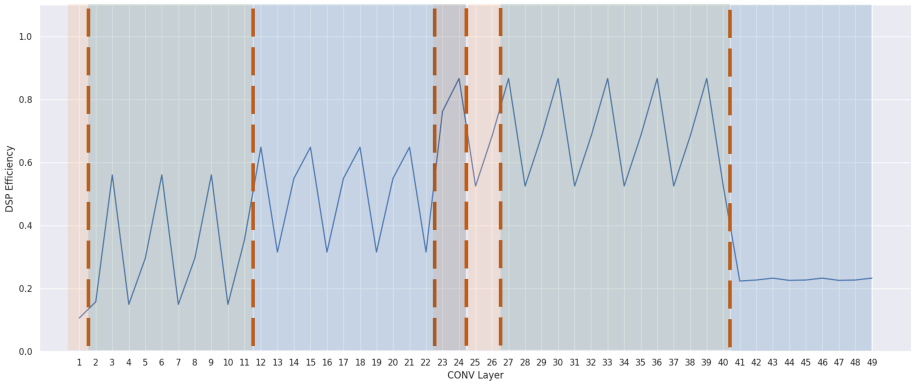
Arch4 achieves a similar throughput as Arch2 on VGG16, and surpasses all the other architectures on ResNet50 and MobileNetV2. On ResNet50, it improves the throughput over Arch1 and Arch2 by 2.6× and 1.7×, respectively. On MobileNetV2, it surpasses the throughput of Arch1 and Arch2 by 5.8× and 1.4×, respectively. Besides, Arch4 achieves a 90%, 91%, and 74% DSP efficiency on VGG16, ResNet50, MobileNetV2, respectively.

Such a performance improvement comes from the ability to adapt to the layer performance disparities with the adaptive partitioning. As an example, Table 4.15 shows a detailed performance breakdown of each array in Arch2 and Arch4 for ResNet50.

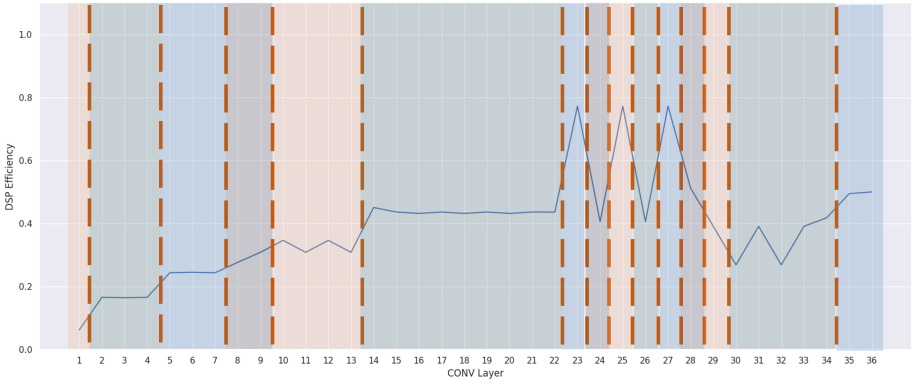
As seen from the table, Arch2 fails to efficiently handle the last few layers (CONV23-49).



(a) VGG16

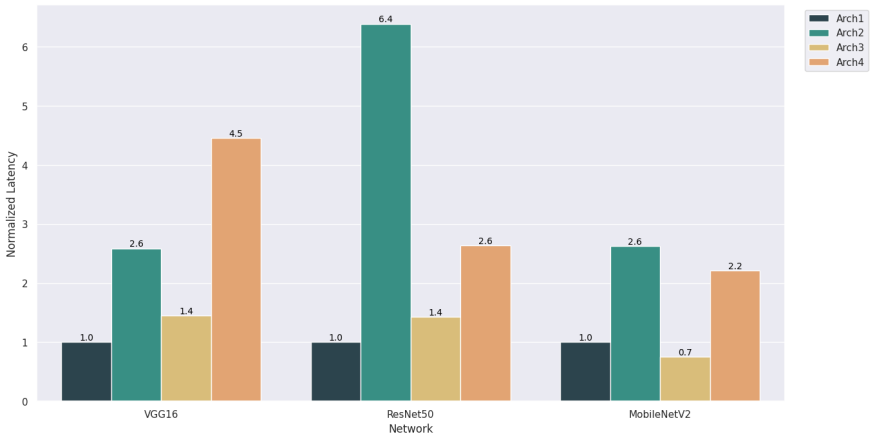


(b) ResNet50

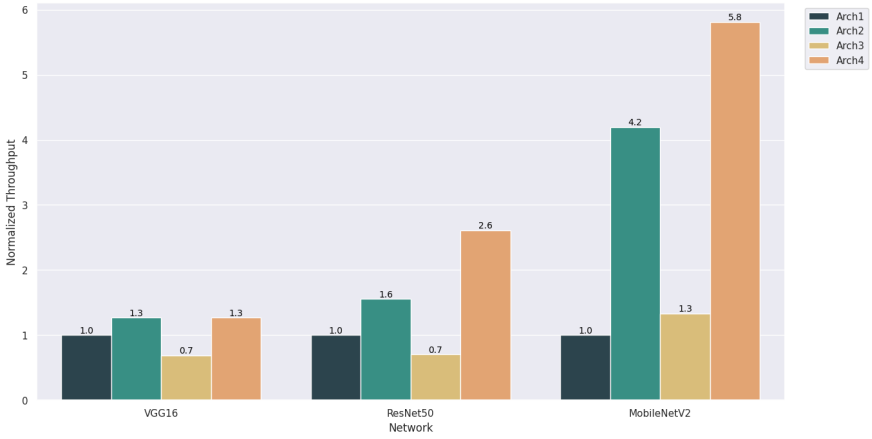


(c) MobileNetV2

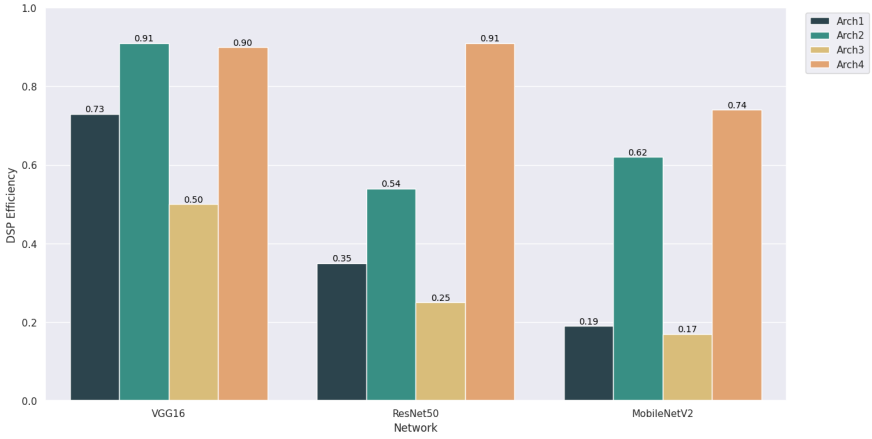
Figure 4.25: Partitioning scheme of the optimal designs found for VGG16, ResNet50, and MobileNetV2.



(a) Latency



(b) Throughput



(c) DSP Efficiency

Figure 4.26: Comparison of latency, throughput, and DSP efficiency of Arch1-4 on different CNNs.

Table 4.15: Performance comparison between Arch2 and Arch4 on ResNet50.

Arch2						Arch4					
Layers	Dataflow	Latency	DSP	DSP Eff.	BRAM	Layers	Dataflow	Latency	DSP	DSP Eff.	BRAM
1	[o,w]	3,515,043	3%	70%	3%	1	[o,w]	2,110,417	3%	100%	5%
2	[o,i]	2,208,101	0%	97%	3%	2-11	[o,w]	2,142,715	20%	89%	19%
3	[o,h]	2,067,013	3%	100%	3%	12-22	[o,w]	2,262,678	23%	93%	12%
4	[o,w]	2,161,327	1%	99%	2%	23-24	[o,i]	2,208,834	4%	100%	6%
5	[h,i]	2,193,411	1%	98%	2%	25-26	[o,h]	2,152,764	3%	99%	6%
6	[o,h]	2,067,013	3%	100%	3%	27-40	[o,h]	2,178,784	29%	95%	21%
7	[o,w]	2,161,327	1%	99%	2%	41-49	[o,w]	2,256,531	18%	92%	20%
8	[h,i]	2,193,411	1%	98%	2%						
9	[o,h]	2,067,013	3%	100%	3%						
10	[o,w]	2,161,327	1%	99%	2%						
11	[h,i]	2,160,328	1%	99%	1%						
12	[o,h]	2,485,349	3%	97%	3%						
13	[o,h]	2,161,503	1%	99%	2%						
14	[o,h]	2,211,161	1%	97%	2%						
15	[o,h]	2,485,349	3%	97%	3%						
16	[o,h]	2,161,503	1%	99%	2%						
17	[o,h]	2,211,161	1%	97%	2%						
18	[o,h]	2,485,349	3%	97%	3%						
19	[o,h]	2,161,503	1%	99%	2%						
20	[o,h]	2,211,161	1%	97%	2%						
21	[o,h]	2,485,349	3%	97%	3%						
22	[o,h]	2,150,064	1%	100%	2%						
23-49	[o,w]	3,798,949	58%	51%	44%						
Total		37,736,962	99%	54%	97%			15,566,522	99%	91%	89%

The last systolic array on these layers manages only a 51% DSP efficiency. In comparison, Arch4 chooses to partition these layers further into four arrays. As a result, these four arrays attain a DSP efficiency of 100%, 99%, 95%, and 92%, respectively. The final design achieves an overall DSP efficiency of 91%, leading to a 1.7× throughput improvement over Arch2.

4.8 Architecture Comparison

So far we have presented the detailed results of all four architectures investigated by this work. We draw the following conclusions from these results.

Dataflow Our evaluation results demonstrate that the exploration of different dataflows is crucial to the final performance. Table 4.16 shows the detailed information of dataflows chosen for each architecture. For the ease of presentation, we assign each dataflow an index following the order in Table 4.4. As seen from the table, arrays in the multi-array architecture

choose different dataflows to maximize their performance. This indicates the need to consider different dataflows in the design space when mapping CNNs to systolic arrays.

Table 4.16: Dataflows of each array in different architectures.

CNN	Architecture	#Array	Dataflows	Batch Size
VGG16	Arch1	1	5	1
	Arch2	6	858445	1
	Arch3	2	55	1
	Arch3	2	44	16
	Arch4	9	844445645	1
ResNet50	Arch1	1	5	1
	Arch2	23	56458458458444444444445	1
	Arch3	2	55	1
	Arch3	7	4444544	16
	Arch4	7	5556445	1
MobileNetV2	Arch1	1	6	1
	Arch2	21	444845448686856666646	1
	Arch3	2	56	1
	Arch3	4	5656	16
	Arch4	15	484454666666455	1

Single array vs. multi-array Single monolithic arrays fail to cope with the performance disparity of CONV layers and face a low computation efficiency on CNNs. This issue is further exacerbated on recent networks with less parallelism and data locality like ResNet50 and MobileNetV2. We have demonstrated the effectiveness of using a multi-array architecture to solve this problem. The proposed multi-array architecture with a one-shot schedule and adaptive partitioning handles the performance disparity of CONV layers effectively and produces the best performance. It improves the throughput by 1.3×, 2.6×, and 5.8× over single systolic arrays on VGG16, ResNet50, MobileNetV2, respectively. Furthermore, it increases the DSP efficiency on these three networks from 73%, 35%, 19% on single systolic array to 90%, 91%, 74% on VGG16, ResNet50, MobileNetV2, respectively. However, we also see a loss in latency for multi-array architectures. The proposed multi-array architecture takes a 4.5×, 2.6×, and 2.2× longer latency than a single systolic array. This is mainly due

to the setup overheads between adjacent arrays. This result reveals the trade-offs a designer needs to make when choosing the proper architecture. If throughput is the priority of the system, a multi-array architecture is a suitable choice. On the other hand, if latency is still the major concern, a single array is the best choice regardless of the low throughput.

Comparison of different multi-array architectures Among the three multi-array architectures we evaluated, the one-shot schedule outperforms the periodic schedule. The large setup and synchronization overheads in periodic-schedule architectures make it hard to improve the performance on the most evaluated CNNs. Among the two one-shot schedule architectures, we showed that our proposed architecture with a flexible partitioning scheme achieves a better performance on CNNs with more divergent performance characteristics.

4.9 Summary

In this chapter, we perform a comprehensive analysis of different systolic array architectures for CNNs. We use a consistent hardware baseline derived from AutoSA and conduct a quantitative analysis of performance trade-offs of different architectures.

The evaluation results show that the divergent performance characteristics of CNN layers limit the throughput and computation efficiency of a single monolithic array. Multi-array systems help overcome this problem because they have a more flexible architecture that allows the designer to customize different arrays to adapt more efficiently to different layers. We have also proposed a new multi-array architecture which partitions the network in an adaptive manner that achieves the best throughput and computation efficiency among all the designs. However, it is worth noting that all the multi-array architectures introduce a longer single image processing latency than the single array. In the case where the single image processing latency is the major concern of the system, it is still beneficial to adopt the single array architecture.

This chapter reveals the challenges of mapping real-world complex applications like CNNs to systolic arrays. While AutoSA and Odyssey offer efficient solutions to generating

a high-performance systolic array for a single convolutional layer, the divergent problem sizes of different CONV layers add additional dimensions to the design space which requires non-trivial efforts to optimize the final performance.

Multi-array architectures serve an interesting example to demonstrate the unique advantages of FPGA over other hardware platforms. With the reconfigurability of FPGA, we can design optimal multi-array architectures for different CNNs and map them to FPGA to maximize the performance. This feature is hard to implement on ASICs which usually implement a single monolithic array that achieves lower computation efficiency than FPGA designs.

CHAPTER 5

Accelerating Genome Sequence Alignment Algorithms on GPU

In this chapter, we explore mapping systolic arrays to GPU. The GPU is not a configurable device like an FPGA. It is impossible to map an arbitrary systolic array directly to GPU. However, we could use the systolic array as an overlay above the existing GPU architecture. Specifically, we keep the execution model of the systolic array and map PEs and interconnects to different hardware components of GPU. GPU designs can benefit from such a mapping as systolic arrays exploit the maximal parallelism from the application and minimize the data communication. The key question here is: *how to effectively map such an architecture to GPU?*

This chapter performs a case study on two genome sequence alignment algorithms from the genomics pipeline. We design a linear systolic array for the target application. Each PE in the systolic array is mapped to a GPU thread. PEs use a special hardware intrinsic *shuffle* for inter-PE communication. Shuffle instructions were first introduced on Nvidia Kepler GPU to support threads in the same warp to directly exchange data with each other. It offers an efficient inter-thread communication mechanism with much less latency compared to other schemes such as shared memory and global memory. Our work demonstrates that using shuffle instructions to realize the inter-PE communication in systolic arrays helps improve the design performance. Recent works [PEW19, CWT19] have demonstrated a similar success of implementing systolic array-like architectures on GPU with shuffle instructions. We believe that such a success will motivate more future works on mapping systolic arrays to GPU and create new opportunities for performance optimization.

5.1 Background

The graphics processing unit (GPU) is capable of delivering orders of magnitude higher performance than the conventional CPU. It has become an integral part of today's computing systems.

Table 5.1: Gap between computation and memory performance on modern GPUs.

	Nvidia K1200	Nvidia Titan X
GFLOPs	1,057	6,611
Shared Memory BW(GB/s)	550	3,302
Global Memory BW(GB/s)	80	336

Communication is an important factor for both the performance and energy efficiency of GPU designs. Table 5.1 shows the computation throughput, shared memory and global memory bandwidth on two Nvidia GPUs. As seen from the table, there is a big gap between the computation and memory performance, making it usually infeasible to fully exploit the rich computation resource on GPU. As a consequence, there is an active research community about communication optimization on GPU [Hwu14, XF10, JPJ11].

Communication among threads (*inter-thread communication*) takes up a large portion of the total communication traffic on GPU. There are two conventional methods for GPU threads to communicate with each other: *shared memory* and *global memory*. Threads within the same thread block can communicate through shared memory, which is a scratchpad memory that offers high bandwidth. Threads in different thread blocks can communicate via global memory. For both methods, we need to set explicit synchronization barriers to prevent potential data hazards.

Starting with the Nvidia Kepler architecture, threads within the same warp can communicate with each other using a new instruction called SHFL, or *"shuffle"* [Nvi16]. By using shuffle instructions, a designer can save the shared memory, which may help improve the GPU occupancy, and, because threads within the same warp implement implicit synchronization, it also eliminates the synchronization overheads.

This new instruction provides a unique opportunity for performance optimization on applications with intensive inter-thread communication. However, shuffle has its limitations. Only threads within the same warp can use it. And it will increase the register usage which may become the new bottleneck of occupancy. Such trade-offs need to be carefully examined. Previous works [HKX13, BRS15, JG15] employing shuffle instructions to obtain performance gains did not investigate such trade-offs and there is a lack of quantitative and systematic approaches for analyzing the impacts of shuffle instructions on kernel performance. This motivates us to conduct a detailed analysis of shuffle instructions.

5.2 Contributions

In this work, we select two DNA sequence alignment algorithms (Smith-Waterman (SW) and Pairwise-Hidden-Markov-Model (PairHMM)) from a widely used genomics application (GATK) [Ins16a] as case studies to evaluate the effectiveness of shuffle instructions on performance optimization. For each algorithm, we implement two designs using either shared memory or shuffle for inter-thread communication. Furthermore, we develop a suite of microbenchmarks to measure the latency of shuffle instructions. We build a performance model to analyze the kernel performance based on the measured latency from these microbenchmarks. With the help of the performance model and microbenmarks, we conduct a quantitative analysis of the trade-offs of using shuffle instructions for communication optimization.

We summarize the contributions of this work as follows:

- We conduct a quantitative and systematic analysis of the impact of shuffle instructions on communication optimization. We develop a suite of microbenchmarks for measuring the latency of shuffle instructions and a performance model for analyzing the performance of two sequence alignment algorithms. This model helps validate the results from microbenchmarks and estimate the performance gains of using shuffle instructions.

- We use shuffle instructions to optimize the inter-thread communication for two sequence alignment algorithms, SW and PairHMM. Compared to designs using shared memory,

we achieve speedups of 1.2× and 2.1× for SW and PairHMM, respectively.

- We conclude detailed trade-offs of using shuffle instructions from the case studies of two sequence alignment algorithms. This work provides valuable insights to CUDA programmers when using shuffle instructions for communication optimization in a wider class of applications.

5.3 Understanding Shuffle Instructions

In this section we first introduce the basics of shuffle instructions. Then we present the microbenchmarks for measuring the latency of shuffle and several related GPU instructions.

5.3.1 Shuffle Instruction

Shuffle allows threads within a warp to directly exchange data in registers. It can only be used for threads within the same warp. No explicit synchronization is needed as threads in the same warp implement the implicit synchronization.

There are four variants of shuffle instructions, as depicted in Figure 5.1. *__shfl* can directly copy data from any indexed lane. *__shfl_up* and *__shfl_down* copy data from a lane with either a lower or a higher ID relative to the caller lane. The last instruction *__shfl_xor* copies data from a lane based on bit-wise XOR of the caller's own lane ID.

Figure 5.1: Variants of shuffle instructions.

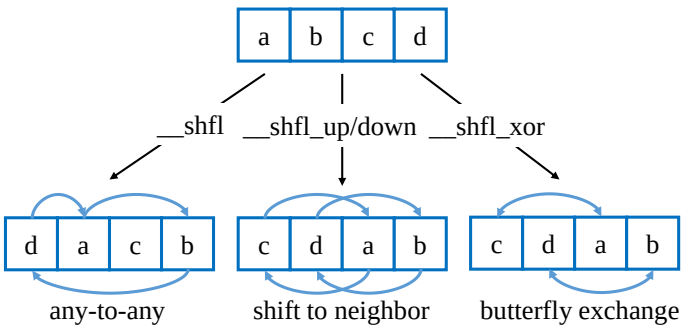
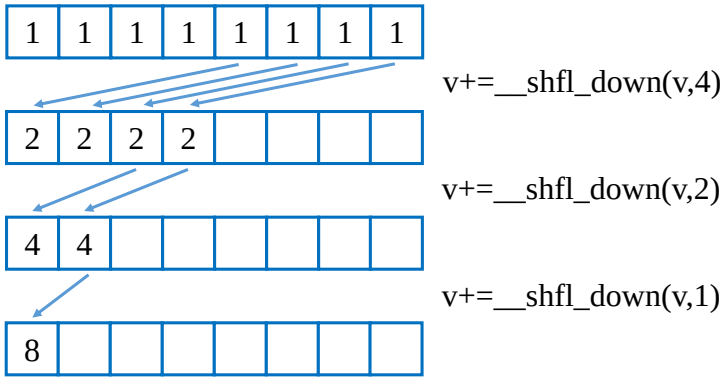


Figure 5.2 depicts an example using shuffle for reduction. In this example, we reduce the data directly between threads at the register level. Without shuffle, we need to store the

Figure 5.2: An example of using shuffle instructions for reduction.



intermediate data at each reduction stage back to either shared or global memory.

At the high level, the benefits of using shuffle are summarized as below [Har14]:

- Save shared memory. Shuffle operates at the register level that helps save the shared memory.
- Reduce instruction count. For read-after-write access (e.g., Figure 5.2), when using shared memory, three instructions (write, synchronize, and read) are needed. Shuffle can finish the same work with only one single instruction that helps reduce the total instruction count.
- Eliminate synchronization overheads. Explicit synchronization is not required since threads within the same warp are implicitly synchronized due to the single-thread-multiple-thread (SIMT) execution model of GPU.

Except the third point, the impacts on performance from the two other points are not clear. Although using shuffle can save shared memory, the register usage will increase, which may become the new limiter of GPU occupancy and affect the final performance. As for the instruction reduction, latency of shuffle instructions is not publicly disclosed by Nvidia, which makes it difficult to estimate the actual latency reduction from the reduced instructions. To better understand these two trade-offs, we develop a suite of microbenchmarks to measure the shuffle latency and perform a a quantitative and systematic analysis of the performance trade-offs using two sequence alignment algorithms as case studies.

5.3.2 Microbenchmarks for Measuring the Shuffle Latency

Shuffle instructions were first introduced on Nvidia Kepler architecture. However, latency of these instructions is not disclosed. Such information is critical to performance estimation. Therefore, we have developed a suite of microbenchmarks to estimate the latency of shuffle and several other related GPU instructions.

This microbenchmark covers all variants of shuffle instructions shown in Figure 5.1. To provide a performance comparison, the microbenchmark also includes the shared memory access and synchronization using *_syncthreads()*.

Listing 10 shows the major code used in the microbenchmark. For the kernel *shuffle()*, each thread in the warp loads one item from the global memory and updates it utilizing the data from other threads for several iterations. Considering the flow dependence of variable a across different iterations, we can calculate the elapsed time of the kernel with the following equation:

$$t_{shuffle} = \#iteration \times (latency_{shuffle} + \alpha) + \beta \tag{5.1}$$

The factor α refers to the sum of latency of all the remaining instructions (e.g., multiplication). The factor β covers overheads outside the loop.

The kernel *register()* employs only register access. Similarly, we calculate the elapsed time for this kernel with the following equation:

$$t_{reg} = \#iteration \times (latency_{reg} + \alpha) + \beta \tag{5.2}$$

After executing multiple runs with different numbers of iterations, we can use a linear regression model to obtain the slope factors $k_{shuffle} = latency_{shuffle} + \alpha$ and $k_{reg} = latency_{reg} + \alpha$. We then derive the latency of the *shfl()* instruction as $latency_{reg} + k_{shuffle} - k_{reg}$.

To avoid the interference of warp scheduling among different thread blocks, we launch only one block with 32 threads. We apply this configuration to all the shuffle instruction tests in this benchmark.

```
__global__ void reg(float* in, float* out) {
  float a = in[threadIdx.x];
  for (int i=0; i<ITERATIONS; i++)
    a *= a;
  out[threadIdx.x] = a;
}
__global__ void shuffle(float* in, float* out) {
  float a = in[threadIdx.x];
  for (int i=0; i<ITERATIONS; i++)
    a *= __shfl(a, src_thread);
  out[threadIdx.x] = a;
}
__global__ void sharedMem(float* in, float* out) {
  __shared__ float buf[32];
  for (int i=0; i<32; i++)
    buf[i] = in[i];
  int ind = buf[0];
  float a = 1.0;
  for (int i=0; i<ITERATIONS; i++) {
    ind = buf[ind];
    a *= ind;
  }
  out[0] = a;
}
__global__ void sharedMemSync(float* in, float* out) {
  __shared__ float buf[32];
  for (int i=0; i<32; i++)
    buf[i] = in[i];
  int ind = buf[0];
  float a = 1.0;
  for (int i=0; i<ITERATIONS; i++) {
    ind = buf[ind];
    a *= ind;
    __syncthreads();
  }
  out[0] = a;
}
```

Listing 10: CUDA code for measuring the instruction latency.

The kernel *sharedMem()* measures the shared memory access latency using the pointer chasing. The goal of this benchmark is to evaluate the latency instead of throughput; therefore,

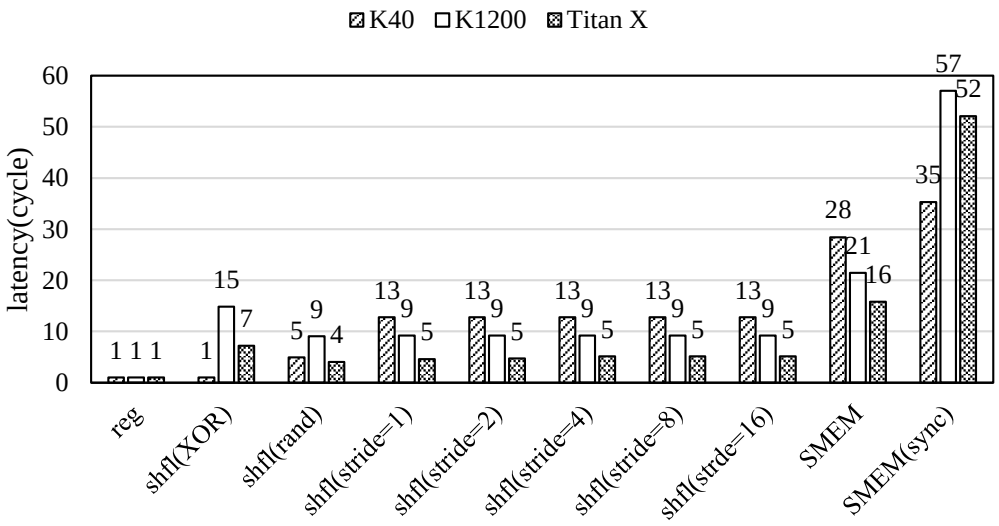
we launch only one thread block with one thread. We use the kernel *sharedMemSync()* to evaluate the latency of *__syncthreads()*. We add *__syncthreads()* after each iteration in the code. With the same methodology, we can derive the latency of shared memory access and synchronization as below:

$$latency_{sharedMem} = latency_{reg} + k_{sharedMem} - k_{reg} \tag{5.3}$$

$$latency_{sync} = latency_{reg} + k_{sync} - k_{reg} - latency_{sharedMem} \tag{5.4}$$

On GPU, the register access takes one cycle to finish, therefore, $latency_{reg} = 1$ in all the equations. We execute all the tests with ten runs and different values of *ITERATIONS*. Figure 5.3 shows the results. We test *__shfl_up/down* with different strides and *__shfl* with randomly generated lane IDs.

Figure 5.3: Test results of microbenchmarks.



As we can see, on average, the latency of shuffle is in-between that of shared memory access and register access. Besides, the latency of different types of shuffle instructions varies. For example, on K1200, *shfl_xor* requires a longer latency than any other shuffle instructions. Such results indicate that the underlying hardware architectures might be different for different shuffle instructions.

The latency pattern of shuffle is consistent on the GPUs with the same architecture (K1200 and Titan X, with Maxwell architecture). However, we notice that such latency varies across different architectures. As we can see, on K40 (Kepler architecture), *shfl_xor* is the instruction with the lowest latency among all shuffle instructions, while it is the instruction with the longest latency on Maxwell architecture. It shows that the underlying architecture for shuffle is also modified across different GPU generations.

The results from this microbenchmark help clear up previous confusion regarding shuffle instructions. Clearly, the shuffle instruction is not as fast as direct register access, but it is still faster than shared memory access. The latency also varies across different types of instructions and GPU architectures.

5.4 Sequence Alignment Algorithms

In this section we will first introduce the basics of sequence alignment algorithms, and then describe the details of the algorithms selected in this work, SW and PairHMM.

5.4.1 Algorithm Overview

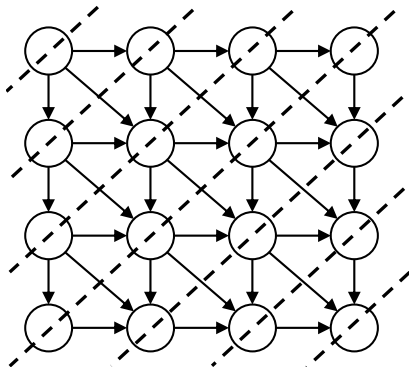
Sequence alignment algorithms, such as Smith-Waterman, Needleman-Wunsch, and PairHMM, have been employed in many application domains such as bioinformatics, finance, language processing, etc. The principle of these algorithms is to traverse possible alignments between two sequences and select the alignment with the best score according to certain criteria using a dynamic programming approach. This is done by updating a matrix with the size of $M \times N$, in which M, N are the lengths of two sequences. The computation complexity of these algorithms is $O(MN)$. Each entry in the matrix depends on its neighbors from certain directions. Figure 5.4 shows the common dependence graph of these algorithms. Each entry depends on its left, upper, and upper-left neighbors.

To accelerate the application, programmers can explore two kinds of parallelisms: 1) inter-task parallelism and 2) intra-task parallelism. Inter-task parallelism refers to the

parallelism among different alignment tasks which are independent of each other while intra-task parallelism refers to the parallelism among different cells on the same anti-diagonal. Figure 5.4 illustrates one example of intra-task parallelism. Many previous works [LN08, MV08, IO16, Int16] used either one or both of these two kinds of parallelisms to accelerate the computation.

Exploiting intra-task parallelism in these algorithms will introduce intensive inter-thread communication. Therefore, we choose these algorithms as our application drivers to justify the effectiveness of shuffle instructions.

Figure 5.4: Dependence graph for sequence alignment algorithms.



5.4.2 Algorithm Details

The SW and PairHMM algorithms in this chapter are extracted from the HaplotypeCaller [Ins16b] of GATK. These two algorithms are used to align DNA sequences for discovering variants in human genes.

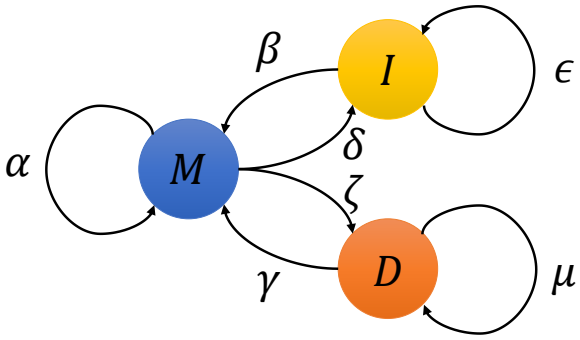
SW [SW81] is a well-known algorithm for sequence alignment. It implements dynamic programming to identify the optimal local alignment between two sequences. Given two

sequences s_1 and s_2 of length M and N, it computes the score matrix H as follows:

$$H_{i,j} = \max \begin{cases} 0 \\ H_{i-1,j-1} + s(a_i, b_j) \\ \max_{k \geq 1}\{H_{i-k,j} + W_k\} \\ \max_{l \geq 1}\{H_{i,j-l} + W_l\} \end{cases} \tag{5.5}$$

where $1 \leq i \leq M$ and $1 \leq j \leq N$. a_i and b_j are i-th and j-th characters in sequence s_1 and s_2, respectively. $s(a, b)$ is a similarity function for two characters, and W_k and W_l are the gap-scoring scheme. For the latter two cases in Equation 5.5, we maintain two buffers holding the local maximum along each direction so that each time only the left and up neighbors need to be accessed. Meanwhile, a back-tracing matrix *btrack* recording the paths chosen for each cell is updated. After the computation, we will locate the cell with the maximal value in the last row and column of the score matrix H, and retrieve the optimal alignment of two sequences back from this position using the matrix *btrack*. Note that the conventional SW will find the maximum throughout the whole score matrix, and GATK has modified the algorithm to adapt to the needs of HaplotypeCaller.

Figure 5.5: PairHMM model.



PairHMM [DEK98] is a variant of SW. However, there are several fundamental differences between the two algorithms. PairHMM uses the hidden Markov model to align two sequences and will generate a probability score, measuring the similarity of two sequences. Figure 5.5 shows the HMM used for this algorithm. There are three states in total: *match*, *insertion*,

and *deletion*. $\alpha, \beta, \gamma, \delta, \epsilon, \zeta, \mu$ are transition probabilities among different states.

Different from SW, in PairHMM, we will compute three score matrices ($match(M)$, $insertion(I)$, and $deletion(D)$) using the following equations:

$$\begin{cases} M_{i,j} = P_{i,j}(\alpha M_{i-1,j-1} + \beta I_{i-1,j-1} + \gamma D_{i-1,j-1}) \\ I_{i,j} = \delta M_{i-1,j} + \epsilon I_{i-1,j} \\ D_{i,j} = \zeta M_{i,j-1} + \mu D_{i,j-1} \end{cases} \tag{5.6}$$

where $P_{i,j}$ is the prior probability of emitting two characters (a_i, b_j) in the two sequences s_1 and s_2. The sum of all the cells in the last row of matrix I and D is the probability that measures the similarity of two sequences.

In conclusion, compared to SW, there are three matrices (M,I, and D) to update instead of only one score matrix H. Also, there is no back-tracing phase in PairHMM, and the output of the PairHMM will be one single number measuring the similarities between two sequences.

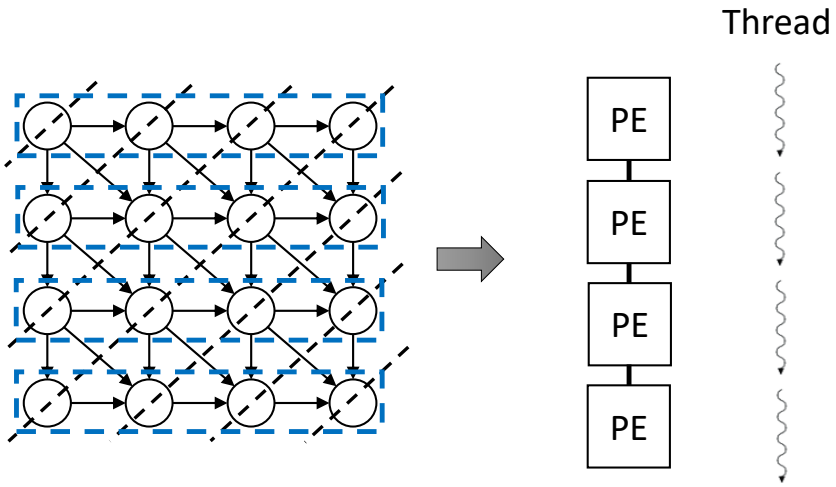
5.5 Implementation Details

In this section, we will first discuss the general methodology of accelerating these two sequence alignment algorithms. Then, we describe further optimization techniques for both kernels. In the end, we present the performance model for analyzing the performance of our designs.

5.5.1 Design Methodology

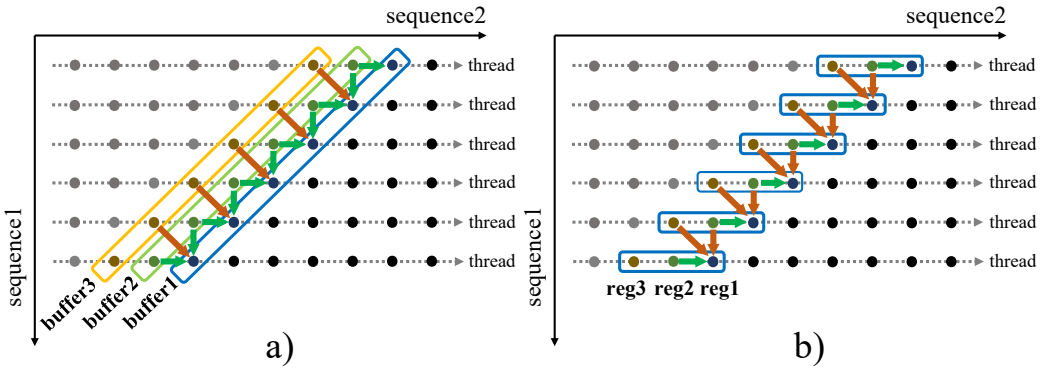
We can map both applications with the dependence pattern as shown in Figure 5.4 to systolic array architecture efficiently to extract the maximal parallelism. Figure 5.6 depicts this mapping. In the dependence graph, we can compute all the computation nodes along the same anti-diagonal in parallel and map such nodes to different PEs. Originally, each node depends on its left, upper, and upper-left neighbors. After the mapping, the data from the left nodes will be kept local to the PE, and data from upper and upper-left nodes need to be transferred from the neighbor PEs.

Figure 5.6: The mapping of SW and PairHMM to systolic array architectures on GPU.



To implement such a design on GPU, we map each PE to one GPU thread. We utilize shared memory and shuffle as two alternatives for inter-PE communication. Figure 5.7 shows the designs using shared memory or shuffle (denoted by design A and B). Listing 11 exhibits the major CUDA code for two designs.

Figure 5.7: GPU designs for sequence alignment algorithms with different types of inter-thread communication: a) design A: using shared memory, b) design B: using shuffle.



In design A, we employ shared memory to store the data on the anti-diagonals. Data on the same anti-diagonal are stored in a line buffer. This enables coalesced shared memory access from different threads on the same anti-diagonal. Three line buffers are sufficient for these algorithms, as seen in Figure 5.7a). After the computation on each anti-diagonal is finished, we will rotate three line buffers and synchronize all the threads before the next iteration.

```
__shared__ data_t buf1[BUF_SIZE];
__shared__ data_t buf2[BUF_SIZE];
__shared__ data_t buf3[BUF_SIZE];
for (int diag = 0; diag < DIAG_NUM; diag++) {
        // LOAD
        data_t left = buf2[threadIdx.x];
        data_t up = buf2[threadIdx.x-1];
        data_t leftup = buf3[threadIdx.x-1];
        // COMPUTE
        data_t cur = compute(left, up, leftup);
        buf1[threadIdx.x] = cur;  // WRITE
        rotate(buf1, buf2, buf3);  // ROTATE
        __syncthreads();  // SYNC
}
```

```
data_t reg1, reg2, reg3;
for (int diag = 0; diag < DIAG_NUM; diag+) {
        // LOAD
        data_t left = reg2;
        data_t up = __shfl(reg2, threadIdx.x-1);
        data_t leftup = __shfl(reg3, threadIdx.x-1);
        // COMPUTE
        data_t cur = compute(left, up, leftup);
        // WRITE
        reg1 = cur;
        // ROTATE
        reg3 = reg2; reg2 = reg1;
}
```

Listing 11: CUDA code for implementations of using shared memory (design A) and shuffle (design B).

In design B, data in the anti-diagonals are stored in local registers, and the three line buffers using shared memory in design A are freed up. Each thread will hold three registers, as denoted by *reg1*, *reg2*, *reg3*. These registers store the three cells calculated or to be calculated by the current thread. As illustrated in Figure 5.7b), in order to calculate a cell, it will load its left neighbor from *reg2* locally, and its upper or upper-left neighbors from *reg2* or *reg3* of the adjacent thread using shuffle instructions. There is no explicit synchronization at the end of each iteration, because the threads within the warps are implicitly synchronized.

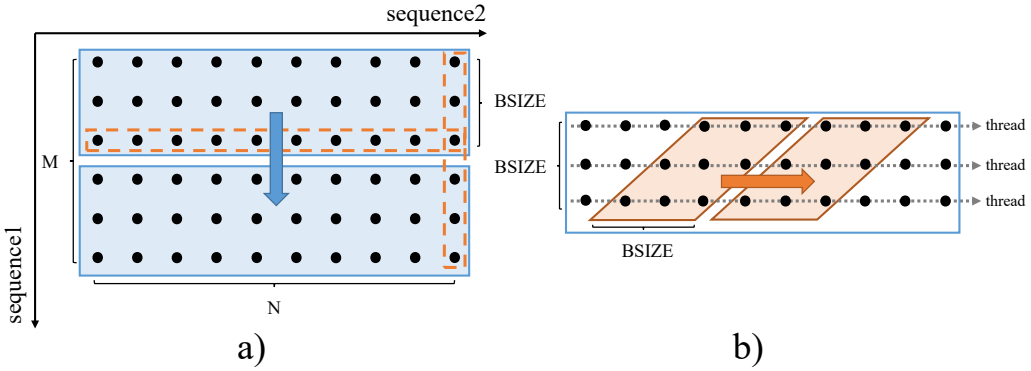
The characteristics of the algorithms and the instructions (shared access vs. shuffle) will bring unique design challenges for each kernel. In the following sections, we will describe the techniques we employ to tackle those challenges and help further improve the performance.

5.5.2 Design Optimization of SW

5.5.2.1 Shared Memory

The major challenge for SW is that we need to store the whole back-tracing matrix *btrack* for retrieving the optimal alignment later. Storing the whole *btrack* matrix in the shared memory is impossible due to the limited shared memory resource. Therefore, we apply a two-level tiling as depicted in Figure 5.8. We first tile the matrix at the row dimension. Then, we tile the whole matrix into blocks of size $BSIZE \times N$. We will store the cells lying

Figure 5.8: Two-level tiling scheme for SW: a) coarse-grained tiling, b) fine-grained tiling. In a), cells on the boundaries that are surrounded by dashed boxes need to be stored back to the global memory. The data on the horizontal boundaries will be retrieved and consumed by the next block, and data on the vertical boundaries are used for finding the maximal value in the last column of the score matrix.



on the boundaries of the block back to the global memory, as the next block will utilize the horizontal boundary, and we will use the vertical boundary for searching for the maximal score later. Inside each block, we further tile it into smaller tiles in the shape of parallelograms. Therefore, each tile covers *BSIZE* anti-diagonals. We choose the shape of parallelograms instead of normal rectangles due to the fact that using rectangular tiles will result in low warp efficiency considering most threads will be wasted at the upper-left and lower right corners of the tiles.

In the end, we will have three line buffers of length *BSIZE*, and a matrix of size $BSIZE \times BSIZE$ to store the data in *btrack*. We will assign *BSIZE* threads for the task. Each thread will work on one complete row in the matrix, and data along the row can be reused locally inside each thread. Figure 5.8 depicts the execution order of tiles.

5.5.2.2 Shuffle

We apply the two-level tiling scheme to shuffle and store the data in the anti-diagonals in local registers. This frees up the previous three line buffers in the shared memory as seen in Figure 5.7b).

5.5.3 Design Optimization of PairHMM

As discussed in Section 5.4, there are several differences between PairHMM and SW, which may affect the design choices when implementing the kernels. We have made several architecture modifications to adapt to these new features.

5.5.3.1 Shared Memory

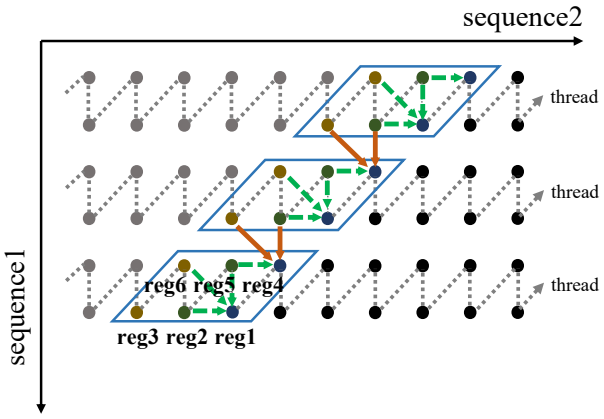
The implementation for PairHMM is nearly the same as seen in Figure 5.7a). The last thread working on the last row will have an additional job of accumulating the results of cells in matrix *match* and *insertion*. Tiling is no longer needed here, because the shared memory is mostly used by line buffers to hold the intermediate results in anti-diagonals, which can fit on-chip in our experiments. However, based on the observations that sequence lengths vary a lot in our datasets, setting the same line buffer length for all tasks is not efficient. We optimize the performance by duplicating the kernels with several copies, each with different line buffer size. Tasks with different sequence lengths will fall into different kernels at the launch time to efficiently use the shared memory.

Similar to SW, each thread will work on one complete row in the matrix, which offers the opportunity of data reuse along the horizontal axis. PairHMM benefits more from this feature, as there are more metadata (e.g., transition probabilities) of sequences used for calculation, and we can save the movement of these data by data reuse.

5.5.3.2 Shuffle

The implementation that uses the shuffle instruction faces the limitation that this instruction can be only used within one warp. Therefore, if there is more than one warp working on the anti-diagonal, threads in different warps need to communicate with each other using either shared memory or global memory. This will result in branch divergence. Besides, the access of shared/global memory will unfortunately cancel the benefits of using shuffle instructions because every shuffle instruction within the warp is accompanied by one shared/global

Figure 5.9: Shuffle implementation for PairHMM. In this example, each thread will hold six registers for two cells in total and compute them one by one on each anti-diagonal. Inter-thread communication only happens between boundary cells.



memory access across the warps. Based on these considerations, we make the compromise to use 32 threads (one warp) for calculating the whole sequence alignment task. This solution still delivers remarkable performance, as shown in Section 5.6.

Similar to what we have done for SW, we create three registers (*reg1*, *reg2*, *reg3*) for each thread. Threads fetch data from neighbors via shuffle instructions. However, using only 32 threads will bring problems for tasks whose sequence lengths are longer than 32. We solve this by assigning multiple cells along the anti-diagonal for each thread, and each thread will compute these cells one after another. We cannot use a fixed number of cells for each thread because this will cause inefficiency across tasks with different sequence lengths. Based on the similar heuristic adopted in the shared memory implementation, we will create subfunctions with different numbers of cells to calculate for each thread and assign the alignment tasks with different subfunctions during the runtime.

Figure 5.9 depicts one example when each thread needs to compute two cells on the anti-diagonal. This scheme improves performance. As we can see, inter-thread communication only takes place between boundary cells across different threads. For the rest of the cells, communication can finish by using direct register access, with the lowest access latency among all the data access methods on GPU.

5.5.4 Performance Model

In this section we present the performance model which provides a quantitative perspective to analyze the performance of different kernels for accelerating sequence alignment algorithms.

CUPs (cell update per second) is the widely used metric for measuring the performance of sequence alignment algorithms. It measures the number of cells in the matrix that can be computed per second. We adopt this metric as the measurement of kernel performance in this work. Note that for PairHMM, we count three updates in three matrices (M, I and D) as one cell update for simplicity.

The performance model is shown as below:

$$performance(CUPs) = \frac{parallelism \times frequency}{latency} \tag{5.7}$$

parallelism is defined as the number of cells updated in parallel. If we assign each thread to update one cell in the matrix, this factor equals the active threads lying in all the SMs on GPU. We can calculate this factor using the equation below:

$$\begin{aligned} parallelism = \#SM \times min\{&\frac{\#reg/SM}{\#reg/thread}, \\ &\frac{\#sharedMem/SM}{\#sharedMem/threadBlock} \times \frac{thread}{threadBlock}\} \end{aligned} \tag{5.8}$$

where $\#reg/SM$, $\#sharedMem/SM$, and $\#SM$ are platform-dependent information, and $\#reg/thread$, $\#sharedMem/threadBlock$ and $thread/threadBlock$ are kernel characteristics, which can be derived from kernels with the help of the Nvidia nvcc compiler by setting specific compilation flags. Note that this factor *parallelism* is in proportional to *occupancy* as well.

We gather the factor $frequency$ from hardware specification. *latency* refers to the average time interval to finish the computation of one cell. More specifically, when every cell on the anti-diagonal is assigned to one thread, it refers to the latency to finish the entire anti-diagonal. The latency is dominated by the critical path in each iteration which includes: 1) *load* data

from neighbor cells, 2) *compute* the current cell, and 3) *write* back the current cell.

This performance model is intuitively simple and can be quite handy when estimating and analyzing the performance of kernels. In our work, this model serves two purposes. First, it helps validate our microbenchmarks. After gathering the performance of kernels and kernel characteristics, we can calculate the *latency* and compare it to the estimated *latency* using results from our microbenchmarks. Second, it can be used to estimate the performance using different communication methods. With estimated *parallelism* employing Equation 5.8 and *latency* from our microbenchmarks, designers can easily make decisions about whether to use shuffle for a new kernel.

5.6 Evaluation Results

In this section we first introduce the experiment setup and present the overall performance of our designs on different platforms. Then we use the performance of the designs to validate the microbenchmarks with the help of the performance model. We discuss the trade-offs of using shuffle instructions in the end.

5.6.1 Experiment Setup

We evaluate the kernel performance on two Nvidia GPUs—Quadro K1200 and Titan X. Both K1200 and Titan use the Maxwell architecture, while K1200 is a low-end GPU with high energy efficiency, and Titan X is a high-end GPU with high computation capability.

The algorithms of SW and PairHMM are extracted from the HaplotypeCaller of GATK 3.6. We use the genome sample of a human with breast cancer (HCC1954) as the inputs to GATK 3.6 and dump out the internal inputs to SW and PairHMM as the evaluation datasets for two kernels.

All the GPU kernels are written in CUDA 7.5. The Nvidia nvcc compiler is used to compile the code and get kernel characteristics (e.g., register and shared memory usage). For the convenience of illustration, in the following sections, we will denote the SW implementations

using shared memory or shuffle as SW1 and SW2, and the PairHMM implementations as PH1 and PH2.

5.6.2 Performance Overview

The DNA sequence is broken into regions to be analyzed in order in HaplotypeCaller. For each region, HaplotypeCaller will have two intermediate stages, generating several pairs of sequences to align, using SW and PairHMM, respectively. We call each sequence pair a *task*. We dump out the tasks for SW and group them together as batches for each region. The input datasets for PairHMM are generated in the same fashion. For each kernel, the number of tasks in each batch varies depending on the DNA clips being analyzed. In our datasets, the average number of tasks per batch for SW is four, whereas the number is 189 for PairHMM. The insufficient tasks for SW will limit the performance, as discussed later.

We measure the GCUPs performance for each batch and take the average. For GPU implementations, each task is assigned to one thread block, and all the tasks within the same batch are launched together as a compute kernel. We set *BSIZE* as 32 for both SW1 and SW2, which offer the best performance from our experiments. We set 128 threads/threadBlock for PH1 because the maximal sequence length is less than 128, and 32 threads/threadBlock for PH2. Note that the GPU performance reported below includes the data transfer time between the host and device.

Figure 5.10: Performance overview of GPU implementations: a) SW, b) PairHMM.

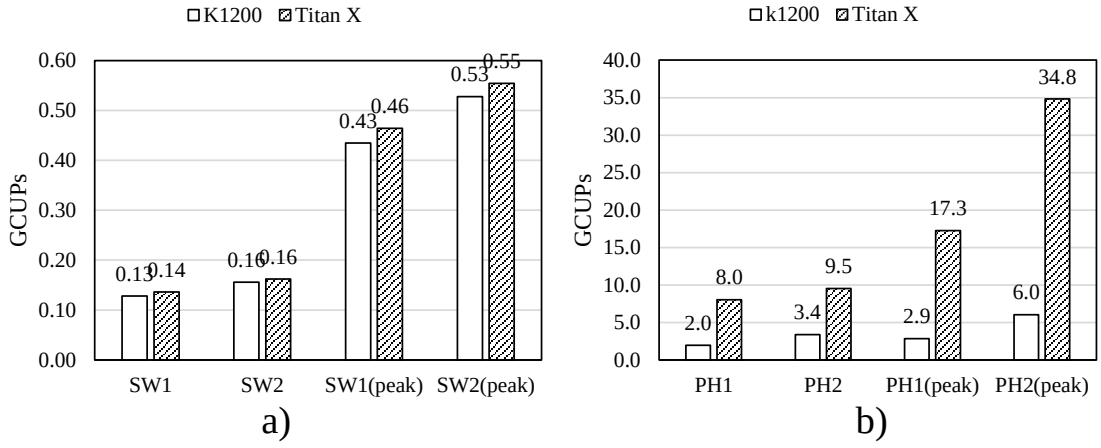
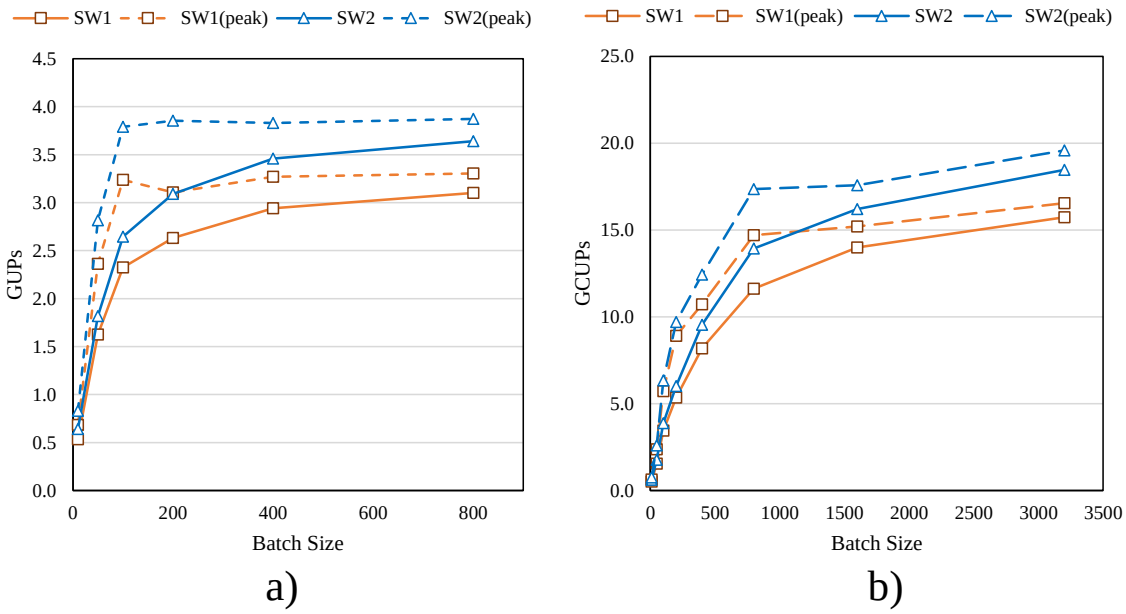


Figure 5.10 shows the average and maximal performance of all the kernels. Note that the performance for SW is relatively low, because the GPU performance is severely restrained by the batch size, as mentioned above. Therefore, we break the boundary of regions and re-batch the tasks in different regions together to evaluate SW kernels. Results are shown in Figure 5.11.

Figure 5.11: Impacts of re-batching on the performance of SW kernels: a) K1200, b) Titan X.



As we can see, our SW kernels deliver significant performance. On Titan X, design SW2 achieves the peak performance of 19.6 GCUPs and an average performance of 18.5 GCUPs when re-batching 3,200 tasks together.

As for PairHMM, on Titan X, design PH2 attains the peak performance of 34.8 GUCPs and an average performance of 6.0 GCUPs. For all the implementations, using shuffle instructions delivers a better performance than those utilizing the shared memory.

5.6.3 Model Validation

In this section, we will use our performance model to validate the results from microbenchmarks. We select the biggest batch among the original datasets (w/o re-batching) as the inputs and K1200 as the target platform. We conduct several repeated runs for each kernel

and take the average as the kernel performance. This number does not include the data transfer because our focus is on the computation part of the kernel.

Table 5.2 presents the kernel performance and statistics. Using our performance model, we can compute the average latency of each iteration for four kernels.

Table 5.2: Detailed information of the kernels.

	SW1	SW2	PH1	PH2
GCUPs	3.6	4.3	3.2	6.8
occupancy(%)	32.2	49.7	56.2	29.1
#reg/thread	54	56	56	94
#sharedMem/threadBlock(B)	1,868	1,472	6,764	2,644
latency(cycle)	1,203	1,014	1,528	379
reduction(cycle)		189		1,149

Table 5.3: Instruction breakdown and analysis for latency reduction.

operation	instruction	#instruction			
		SW1	SW2	PH1	PH2
LOAD	SMEM/(shfl,reg)	3	(2,1)	32	(6,25)
WRITE	SMEM/reg	1	1	12	12
ROTATE	SMEM/reg	2	2	24	24
SYNC		1	0	1	0
	latency	183	22	1,485	115
	est. reduction	161(-14.8%)		1,370(19.2%)	

For SW, using shuffle can cut down the iteration latency by 189 cycles. The reduction comes from: 1) data access latency, and 2) synchronization. Table 5.3 details the breakdown of the latency.

In the shared memory implementation shown in Listing 11a), there are three shared memory loads and one write in each iteration. This implementation requires two more accesses in order to rotate the three line buffers and one $__syncthreads()$ at the end of each iteration. Based on our microbenchmarks shared access takes around 21 cycles and $__sycnthreads()$

takes 57 cycles. We can estimate the latency as $3 \times 21 + 1 \times 21 + 2 \times 21 + 1 \times 57 = 183$ cycles. In SW2, three loads from neighbor cells are replaced by two shuffles and one direct register access, as shown in Listing 11b). All the other instructions can be replaced by direct register accesses; the synchronization is eliminated. The estimated latency will be $2 \times 9 + 4 \times 1 = 22$ cycles. Therefore, the estimated latency reduction is $183 - 22 = 161$ cycles. The relative error of our analysis is -14.8%.

For PairHMM, in each iteration, there are three matrices to update (eight loads in total: three in *match*, three in *insertion*, and two in *deletion*). Additionally, we use 128 threads (4 warps) to update one anti-diagonal, which will issue $8 \times 4 = 32$ shared memory instructions each time. As for design PH2, each thread will compute 4 cells in total. Inter-thread communication happens between boundary cells, which brings two shuffle instructions and one register access for each matrix (only two shuffle for *deletion*). For all the inside remaining cells, direct register access suffices. There will be six shuffle instructions and 25 register accesses in total. The analysis for other operations are similar to SW. Based on our estimation, using shuffle instruction helps reduce latency by 1370 cycles. The relative error is 19.2%.

The relative error of the prediction for both designs is low, which validates the results of the microbenchmarks. This analysis shows a normal flow for CUDA programmers to estimate the performance gains when using shuffle instructions. We can first estimate the new register and shared memory usage when using shuffle instructions and compute the factor *parallelism*. Then, we can calculate the *latency* based on the computation breakdown and latency of instructions from microbenchmarks. Finally, we can use our model to compute the performance of shuffle designs.

5.6.4 Trade-Off Analysis

Using shuffle instructions helps improve performance over designs using shared memory. As shown in Table 5.2, using shuffle instructions provides performance gains of 1.2× and 2.1× for SW and PairHMM, respectively.

For SW, using shuffle instructions helps save shared memory, and therefore increases

the occupancy (*parallelism*). Meanwhile, it reduces *latency*. Both factors contribute to the performance improvement of SW2 over SW1.

As for PairHMM, we observe a drop of occupancy from PH1 to PH2. The reason is that we assign more cells for each thread to calculate, thus significantly increasing the register usage for each thread. The register usage becomes the limiter of occupancy and drags down the occupancy from 56.2% to 29.1%. Meanwhile, inter-thread communication in PairHMM is more intensive than SW, as we will access more data (three matrices vs. one matrix) with more threads (128 threads/threadBlock vs. 32 threads/threadBlock). The reduction of *latency* from using shuffle instructions is larger than SW, which offsets the decrease of *parallelism* and improves *performance* eventually.

Based on the analysis above, we conclude the trade-offs of using shuffle instructions as follows:

- Using register shuffle can free up shared memory. However, the impact on occupancy varies among different applications. It is possible that the increased number of registers may become the new limiter of occupancy, which will hurt the *parallelism* we can obtain.

- In terms of performance, both *parallelism* and *latency* matter. For applications with intensive inter-thread parallelism, the reduction of the *latency* using shuffle instructions plays an important role in the overall *performance*, which could even offset the negative impacts of using more registers and bring performance gains in the end.

With the help of the performance model and the microbenchmarks for shuffle instructions, CUDA programmers can easily handle such trade-offs and make design decisions in advance. Note that the root cause of the first point lies in the limitation of shuffle instructions since they can only be used for threads within the same warp. Therefore, for applications like PairHMM, we need to place more cells to calculate for each thread, with more registers to use per thread. This will significantly increase the register usage and affect the occupancy eventually.

5.7 Related Work

The shuffle instruction offers a new alternative for inter-thread communication. Previous works [HKX13, BRS15, JG15] leveraged shuffle instructions as a replacement for shared memory operations and obtained performance gains from such optimization. However, there is no detailed analysis about the root causes for such benefits. Our work extends the shuffle instructions to a new application domain, sequence alignment, and is the first work to conduct a systematic and quantitative analysis of shuffle instructions.

In this work, we pick SW and PairHMM as two application drivers which present dependence patterns of near-neighbor communication. SW is a well-known sequence alignment algorithm, and there are many GPU implementations for different variants of SW. Manavski and Valle [MV08] utilized the inter-task parallelism by assigning each GPU thread one entire alignment task. The sequences need to be sorted in advance so that the task for threads within the same block can be as similar as possible. Lu and Ni [LN08] proposed a combined solution which couples CPU and GPU together to accelerate the SW protein search. They used the same programming model as Manavski and Valle [MV08] and further utilized the SIMD instructions to improve parallelism. The SW application we adopt in this chapter is different from these works in terms of both the algorithm and datasets.

PairHMM is a variant of SW which integrates HMM into the algorithm. There have been several previous works [IO16, RSA15] on CPU and FPGA that employed the intra-task parallelism along the anti-diagonals. Intel released Genomics Kernel Library (GKL) [Int16] which uses AVX intrinsics to accelerate the algorithm. Ito and Ohara [IO16] proposed a systolic array architecture on FPGA that uses the IBM CAPI interface and delivers the performance of 1.7 GCUPs on the same genome sample that we use in this work. Our implementation in this work outperforms all the previous works on PairHMM.

5.8 Summary

Data movement is one of the critical limiting factors of performance and energy efficiency. In this work, we look into the communication optimization methodology for applications with intensive inter-thread communication. The shuffle instruction offers new alternatives in addition to conventional methods for using shared memory and global memory and brings trade-offs to be considered at the same time. We conduct a quantitative analysis on shuffle instructions, using two sequence alignment algorithms (SW and PairHMM) as case studies.

To measure the latency of shuffle and several other instructions, we develop a suite of microbenchmarks. We find that the latency of shuffle is in-between that of register and shared memory access, and it varies across different types of shuffle instructions and architectures (Kepler vs. Maxwell).

We use a systolic array-like architecture to implement two algorithms on GPU. Each PE in the array is mapped to a GPU thread. We implement inter-PE communication using either shared memory of shuffle instructions. Using shuffle instructions instead of shared memory has brought significant performance gains of 1.2× and 2.1× for SW and PairHMM, respectively. This demonstrates the optimization opportunities for deploying shuffle instructions for applications with intensive inter-thread communication.

We propose a performance model that takes kernel characteristics and the instruction latency into consideration and helps analyze and estimate the design performance for such algorithms. With the help of this performance model and microbenchmarks, we conduct a detailed analysis on the performance impacts of shuffle instructions.

This work provides valuable insights for CUDA programmers for making trade-offs when using shuffle instructions in a wider class of applications.

CHAPTER 6

Conclusion

The domain-specific architecture brings new opportunities to overcome the challenges of the power and memory wall in the post-Moore's law age. However, it also brings new challenges to software programmers and hardware designers in terms of programming support and architecture design. This dissertation focuses on one such architecture, systolic array, and provides solutions to programming support and architecture optimizations on this architecture.

Designing a systolic array requires the expert knowledge of both the application characteristics and hardware architectures. A designer needs to analyze the dependence structure of the program, transform it to the systolic array execution model, write the hardware code, and optimize the final performance. Each step might take significant efforts, making the entire designing process a challenging task. Despite of a plethora of previous works on automatic systolic array synthesis, we observe that the manual designing process remains the norm due to the low level of generality, performance, and productivity of these systolic array synthesis tools. To overcome these issues, we have developed an open-source systolic array compiler, AutoSA, covered in Section 2.5. AutoSA leverages the polyhedral compilation techniques to perform program transformation and code generation, and targets high-level synthesis languages like Xilinx HLS C as outputs. It incorporates a comprehensive set of hardware optimization techniques to support general applications and generates designs with high performance. We show that AutoSA is capable of handling applications with complex dependence structure, like LU decomposition, and generates designs with comparable or better performance than the previous works. AutoSA brings a significant technical advancement in the field of automatic systolic array synthesis, making such an architecture accessible to average programmers.

The rich set of optimization techniques AutoSA incorporates brings a vast design space that is infeasible to explore manually. To further improve the productivity, in Chapter 3, we proposed an auto-tuning framework, Odyssey, for AutoSA. Odyssey implements a hybrid search method which includes both a mathematical programming (MP)-based optimizer and an evolutionary search-based search engine. The MP-based optimizer uses high-order optimization targets, including the off-chip data communication and computation resource to generate high-quality design seeds that are used as the first generation of the evolutionary search, helping to further improve the search quality and convergence speed of evolutionary search. Inside the evolutionary search, Odyssey implements a hybrid mutation method that takes both divisor and non-divisor tiling factors into consideration. The inclusion of non-divisors guarantees the completeness of the design space without leaving out any optimal design points. Lastly, Odyssey is fully automated and seamlessly integrated with AutoSA. AutoSA can automatically generate the hardware performance models and plug them into the auto-tuners in the Odyssey flow. Combining all these methods together, Odyssey achieves high search quality and convergence speed. We have shown that Odyssey is able to locate designs reaching 90% of the optimal performance found by exhaustive search in 5 seconds with a single CPU thread.

With AutoSA and Odyssey, designers could now easily generate a high-performance systolic array within days for a certain workload. However, for real-world complex applications, this only serves as the first step in achieving the system-wise high performance. Chapter 4 looks into the challenges of mapping one important application, convolutional neural network (CNN), to systolic arrays. We show that a single systolic array faces low throughput and computation efficiency on CNNs due to the divergent performance characteristics in terms of the parallelism and data locality of different CNN layers. This issue is further exacerbated on recent CNNs which achieves high prediction accuracy with a much smaller network size. For example, the best single array architecture on Xilinx Alveo U250 attains only a 73% DSP efficiency on VGG16. This number decreases to 35% and 19% on two recent CNNs, ResNet50 and MobileNetV2.

To overcome this challenge, we looked into the multi-array architecture that implements

a set of smaller systolic arrays on-chip, each customized to adapt to different CNN layers. There are many previous works on this direction, proposing different multi-array architectures for CNNs. We selected two representative architectures from two previous works DNNExplorer [ZYW20] and TGPA [WLL18] and conducted a quantitative comparison using the systolic arrays generated from AutoSA, which serve as a consistent and open-source hardware baseline. Our studies show that the multi-array architecture is effective in improving the system throughput and DSP efficiency at a cost of increased latency. Furthermore, the DNNExplorer architecture achieves a better performance which partitions the network into halves, mapping the layers in the first half to different systolic arrays and grouping the rest to a single array. In the experiments, we found the rigid partitioning scheme limits the performance of the DNNExplorer architecture. Specifically, CNNs like ResNet50 consist of CNN layers with divergent performance characteristics throughout the network. Mapping the later layers in the network to a single array faces the same performance issues as a single monolithic array. To cope with this challenge, we proposed a new architecture that partitions the network in an adaptive manner that groups layers with a similar performance together to maximize the performance. This architecture achieves the highest throughput and computation efficiency among all the architectures evaluated.

The work in Chapter 4 has several profound implications. First, it shows the complexities when mapping a complicated application to systolic arrays. In addition to the generation of single systolic arrays, which AutoSA and Odyssey efficiently handle, to achieve high performance on the application level, there are numerous new issues to take into account such as the layer diversity in CNNs. The methodologies and results in Chapter 4 can serve as a good reference to future works when mapping different applications to systolic arrays. Second, the promising results of multi-array architectures demonstrate the unique advantages of FPGAs over other hardware platforms like ASICs. Specifically, we could customize the hardware design for each CNN and FPGA board to maximize the overall performance, whereas ASICs are usually bound with a fixed architecture that fails to adapt to different applications, leading to an inferior resource utilization.

The last work in Chapter 5 provides an interesting example of mapping systolic arrays

to GPU. While a GPU is not a reconfigurable architecture like FPGA, on which we could implement arbitrary systolic arrays, we could still use the systolic array as an overlay over the existing GPU hardware by reusing the same execution model and mapping different components of the arrays to the GPU hardware. The major contribution of this work is the study of using shuffle instructions on GPU to implement the inter-PE communication. Shuffle instructions were first introduced on Nvidia Kepler architecture which allow GPU threads within a single warp to exchange data with each other at the register level. Therefore, shuffle instructions provide a new inter-thread communicating mechanism with lower latency than shared memory and global memory, offering new performance optimization opportunities to GPU designs. We use two sequence alignment algorithms from the genomics pipeline as the case studies. We map 1D systolic arrays to GPUs for these two applications, using shuffle instructions for efficient inter-PE communication. As a result, we improved the throughput over the baseline implementations using the shared memory by 1.2× and 2.1×. In addition to the hardware implementations, we also developed a suite of microbenchmarks for profiling the latency of shuffle instructions and proposed a performance model to estimate the performance of these designs for evaluating the trade-offs of using shuffle instructions versus other mechanisms like shared memory. We believe that the success of this work will motivate more future examples in deploying systolic array-like architectures to GPU and create new opportunities for performance optimization.

Lastly, there are many interesting directions to further extend the works in this dissertation and we list them here as the future work.

On the side of programming support, the multi-level IR compiler framework (MLIR) [LAB20] offers a promising solution to standardizing the development of hardware accelerators and further reduce the development costs. Furthermore, MLIR incorporates an affine dialect that brings the polyhedral compilation to the first-class citizen in its compiler infrastructure. There are also several exciting projects from the hardware communities such as the CIRCT project (Circuit IR Compilers and Tools) [LLV21] that aims to implement an open-source compilation flow for hardware designs based on the MLIR framework. We see the MLIR infrastructure as a promising target to migrate all the compilation techniques from AutoSA.

Many compilation techniques developed for AutoSA are not unique to systolic arrays. For example, the dependence-based data transfer analysis can be used for generating the on-chip I/O network for other accelerators as long as the input algorithms obey the affine constraints. Therefore, an MLIR-based systolic array compilation flow not only helps to extend AutoSA to more hardware targets like ASICs, but also benefits the development of other hardware accelerators with the general compilation techniques it incorporates.

On the side of architecture optimizations, in this dissertation, we have demonstrated the effectiveness of a multi-array architecture over the single fixed array in improving the computation efficiency and throughput. While it is hard to implement the multi-array architecture on ASICs given its high non-recurring engineering costs, there are some other interesting architecture candidates that might benefit from the multi-array designs. For example, coarse-grained reconfigurable architecture (CGRA) is an interesting hardware target to consider. CGRA includes flexible hardware interconnects between different PEs and can be easily reconfigured to apply different dataflows. Theoretically, we could partition different PEs in CGRA to implement each array in the multi-array architecture and leverage the flexible on-chip interconnects to implement different systolic array dataflows. This mapping scheme will help further improve the throughput and computation efficiency on CGRA than mapping with the single array model using the techniques discussed in Chapter 4.

This dissertation covers a broad set of in-depth solutions to the tasks of automatic synthesis and architecture optimization of the systolic array architecture. Works in this dissertation advanced the field in providing a higher level of programming support to this important architecture and introducing a new set of architecture optimizations to further boost the performance. Besides, many of the techniques proposed in this dissertation are general and can be applied to other architectures and hardware platforms. We hope these works will open the gate to more deployment cases of domain-specific architectures like systolic arrays in a wider range of applications, helping to cope with the computation challenges in this new era.

Bibliography

[ACF16] Manoj Alwani, Han Chen, Michael Ferdman, and Peter Milder. "Fused-layer CNN accelerators." In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12. IEEE, 2016.

[AKV14] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. "Opentuner: An extensible framework for program autotuning." In *Proceedings of the 23rd international conference on Parallel architectures and compilation*, pp. 303–316, 2014.

[AKV21] Jason Ansel, Shoaib Kamil, Kalyan Veeramachaneni, Jonathan Ragan-Kelley, Jeffrey Bosboom, Una-May O'Reilly, and Saman Amarasinghe. "OpenTuner." `https://github.com/jansel/opentuner`, 2021.

[Ama20] Amazon. "AWS Inferentia." `https://aws.amazon.com/machine-learning/inferentia`, 2020.

[BCC88] Shekhar Borkar, Robert Cohn, George Cox, Sha Gleason, and Thomas Gross. "Warp: An integrated solution of high-speed parallel computing." In *Proceedings of the 1988 ACM/IEEE conference on Supercomputing*, pp. 330–339. IEEE Computer Society Press, 1988.

[BCG03] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma, and Olivier Temam. "Putting polyhedral loop transformations to work." In *LCPC*, pp. 209–225, College Station, Texas, October 2003.

[BHR08] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. "A practical automatic polyhedral parallelizer and locality optimizer." In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 101–113, 2008.

[Bon13] Uday Bondhugula. "Compiling affine loop nests for distributed-memory parallel architectures." In *SC'13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12. IEEE, 2013.

[BPB12] Vinayaka Bandishti, Irshad Pananilath, and Uday Bondhugula. "Tiling stencil computations to maximize parallelism." In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–11. IEEE, 2012.

[BPC10] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen, and Cédric Bastoul. "The polyhedral model is more widely applicable than you think." In *International Conference on Compiler Construction*, pp. 283–303. Springer, 2010.

[BRR19] Riyadh Baghdadi, Jessica Ray, Malek Ben Romdhane, Emanuele Del Sozzo, Abdurrahman Akkas, Yunming Zhang, Patricia Suriana, Shoaib Kamil, and Saman Amarasinghe. "Tiramisu: A polyhedral compiler for expressing fast and portable code." In *2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 193–205. IEEE, 2019.

[BRS07] Uday Bondhugula, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. "Automatic mapping of nested loops to FPGAs." In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming*, pp. 101–111, 2007.

[BRS10] Muthu Manikandan Baskaran, Jj Ramanujam, and P Sadayappan. "Automatic C-to-CUDA code generation for affine programs." In *International Conference on Compiler Construction*, pp. 244–263. Springer, 2010.

[BRS15] Darius Bakunas-Milanowski, Vernon Rego, Janche Sang, and Chansu Yu. "A fast parallel selection algorithm on GPUs." In *2015 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 609–614. IEEE, 2015.

[BT03] Marcus Bednara and Jürgen Teich. "Automatic synthesis of FPGA processor arrays from loop algorithms." *The Journal of Supercomputing*, **26**(2):149–165, 2003.

[CCP16] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. "A cloud-scale acceleration architecture." In *The 49th Annual IEEE/ACM International Symposium on Microarchitecture*, p. 7. IEEE Press, 2016.

[CES16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. "Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks." *ACM SIGARCH Computer Architecture News*, **44**(3):367–379, 2016.

[CGG14] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Karthik Gururaj, and Glenn Reinman. "Accelerator-rich architectures: Opportunities and progresses." In *Proceedings of the 51st Annual Design Automation Conference*, DAC '14, pp. 180:1–180:6, New York, NY, USA, 2014. ACM.

[CGL21] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. "Extending high-level synthesis for task-parallel programs." In *2021 IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 204–213. IEEE, 2021.

[CHW20] Xiaoming Chen, Yinhe Han, and Yu Wang. "Communication lower bound in convolution accelerators." In *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 529–541. IEEE, 2020.

[CKR20] Prasanth Chatarasi, Hyoukjun Kwon, Natesh Raina, Saurabh Malik, Vaisakh Haridas, Angshuman Parashar, Michael Pellauer, Tushar Krishna, and Vivek Sarkar. "Marvel: A data-centric compiler for DNN operators on spatial accelerators." *arXiv preprint arXiv:2002.07752*, 2020.

[CLN11] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. "High-level synthesis for FPGAs: From prototyping to deployment."

IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, **30**(4):473–491, 2011.

[Clo21] Google Cloud. "Google demonstrates leading performance in latest MLPerf Benchmarks." `https://cloud.google.com/blog/products/ai-machine-learning/google-wins-mlperf-benchmarks-with-tpu-v4`, 2021.

[CM08] Philippe Coussy and Adam Morawiec. *High-level synthesis: From algorithm to digital circuit*. Springer Science & Business Media, 2008.

[CMJ18] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. "TVM: An automated end-to-end optimizing compiler for deep learning." In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pp. 578–594, 2018.

[COI21] COIN-OR. "Interior Point Optimizer." `https://coin-or.github.io/Ipopt/`, 2021.

[CW18] Jason Cong and Jie Wang. "PolySA: Polyhedral-based systolic array auto-compilation." In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8. IEEE, 2018.

[CWT19] Peng Chen, Mohamed Wahib, Shinichiro Takizawa, Ryousei Takano, and Satoshi Matsuoka. "A versatile software systolic execution model for GPU memory-bound kernels." *arXiv preprint arXiv:1907.06154*, 2019.

[CWV14] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. "cuDNN: Efficient primitives for deep learning." *arXiv preprint arXiv:1410.0759*, 2014.

[CWY18] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. "Latte: Locality aware transformation for high-level synthesis." In *2018 IEEE 26th Annual International*

Symposium on Field-Programmable Custom Computing Machines (FCCM), pp. 125–128. IEEE, 2018.

[CY89] C-Y Chang and Kung Yao. "Systolic array processing of the Viterbi algorithm." *IEEE transactions on information theory*, **35**(1):76–86, 1989.

[CZY18] Tianqi Chen, Lianmin Zheng, Eddie Yan, Ziheng Jiang, Thierry Moreau, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. "Learning to optimize tensor programs." *arXiv preprint arXiv:1805.08166*, 2018.

[DDS09] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. "Imagenet: A large-scale hierarchical image database." In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, pp. 248–255. IEEE, 2009.

[DEK98] Richard Durbin, Sean R. Eddy, Anders Krogh, and Graeme Mitchison. *Biological sequence analysis: probabilistic models of proteins and nucleic acids*. Cambridge University Press, 1998.

[DGR74] Robert H. Dennard, Fritz H. Gaensslen, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions." *IEEE Journal of Solid-State Circuits*, **9**(5):256–268, 1974.

[DKA19] Shail Dave, Youngbin Kim, Sasikanth Avancha, Kyoungwoo Lee, and Aviral Shrivastava. "Dmazerunner: Executing perfectly nested loops on dataflow accelerators." *ACM Transactions on Embedded Computing Systems (TECS)*, **18**(5s):1–27, 2019.

[DLH19] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. "FBLAS: Streaming linear algebra on FPGA." *arXiv preprint arXiv:1907.07929*, 2019.

[DRR13] Roshan Dathathri, Chandan Reddy, Thejas Ramashekar, and Uday Bondhugula. "Generating efficient data movement code for heterogeneous architectures with distributed-memory." In *Proceedings of the 22nd international conference on Parallel architectures and compilation techniques*, pp. 375–386. IEEE, 2013.

[EBA11] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. "Dark silicon and the end of multicore scaling." In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pp. 365–376. IEEE, 2011.

[FKH20] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. "Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis." In *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 244–254, 2020.

[Gay15] David M Gay. "The AMPL modeling language: An aid to formulating and solving optimization problems." In *Numerical Analysis and Optimization*, pp. 95–116. Springer, 2015.

[GCW21] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. "AutoBridge: Coupling coarse-grained floorplanning and pipelining for high-frequency HLS design on multi-die FPGAs." In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2021.

[GHI19] Hasan Genc, Ameer Haj-Ali, Vighnesh Iyer, Alon Amid, Howard Mao, John Wright, Colin Schmidt, Jerry Zhao, Albert Ou, Max Banister, et al. "Gemmini: An agile systolic array generator enabling systematic evaluations of deep-learning architectures." *arXiv preprint arXiv:1911.09925*, 2019.

[GK82] W Morven Gentleman and HT Kung. "Matrix triangularization by systolic arrays." In *Real-Time Signal Processing IV*, volume 298, pp. 19–26. International Society for Optics and Photonics, 1982.

[GKP19] Paolo Gorlani, Tobias Kenter, and Christian Plessl. "OpenCL implementation of Cannon's matrix multiplication algorithm on Intel Stratix 10 FPGAs." In *2019 International Conference on Field-Programmable Technology (ICFPT)*, pp. 99–107. IEEE, 2019.

[GQQ01] Anne-Claire Guillou, Fabien Quilleré, Patrice Quinton, S. Rajopadhye, and Tanguy Risset. "Hardware design methodology with the Alpha language." *FDL'01*, 2001.

[GZA11] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger, and Louis-Noël Pouchet. "Polly-polyhedral optimization in LLVM." In *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, volume 2011, p. 1, 2011.

[Har14] Mark Harris. "CUDA Pro Tip: Do The Kepler Shuffle." `https://devblogs.nvidia.com/parallelforall/cuda-pro-tip-kepler-shuffle/`, 2014.

[HBS20] Tae Jun Ham, David Bruns-Smith, Brendan Sweeney, Yejin Lee, Seong Hoon Seo, U Gyeong Song, Young H Oh, Krste Asanovic, Jae W Lee, and Lisa Wu Wills. "Genesis: A hardware acceleration framework for genomic data analysis." In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 254–267. IEEE, 2020.

[HKD21] Qijing Huang, Minwoo Kang, Grace Dinh, Thomas Norell, Aravind Kalaiah, James Demmel, John Wawrzynek, and Yakun Sophia Shao. "CoSA: Scheduling by constrained optimization for spatial accelerators." *arXiv preprint arXiv:2105.01898*, 2021.

[HKX13] Yoshiaki Hanada, Shinya Kitaoka, and Ye Xinhua. "Optimizing particle simulation for Kepler GPU." *Procedia Engineering*, **61**:376 – 380, 2013.

[Hol92] John H. Holland. "Genetic algorithms." *Scientific American*, **267**(1):66–73, 1992.

[Hor14] Mark Horowitz. "1.1 Computing's energy problem (and what we can do about it)." In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pp. 10–14. IEEE, 2014.

[HPS12] Justin Holewinski, Louis-Noël Pouchet, and Ponnuswamy Sadayappan. "High-performance code generation for stencil computations on GPU architectures." In

Proceedings of the 26th ACM international conference on Supercomputing, pp. 311–320, 2012.

[Hwu14] Wen-mei Hwu. "What is ahead for parallel computing." *Journal of Parallel and Distributed Computing*, **74**(7):2574–2581, 2014.

[HZR16] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. "Deep residual learning for image recognition." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 770–778, 2016.

[Ins16a] Broad Institute. "Genome Analysis Toolkit." `https://software.broadinstitute.org/gatk/`, 2016.

[Ins16b] Broad Institute. "HaplotypeCaller." `https://software.broadinstitute.org/gatk/gatkdocs/org_broadinstitute_gatk_tools_walkers_haplotypecaller_HaplotypeCaller.php`, 2016.

[Int16] Intel. "Genomics Kernel Library (GKL)." `https://github.com/Intel-HLS/GKL`, 2016.

[Int21] Intel. "Intel FPGA SDK for OpenCL Software Technology." `https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html`, 2021.

[IO16] Megumi Ito and Moriyoshi Ohara. "A power-efficient FPGA accelerator systolic array with cache-coherent interface for pair-HMM algorithm." In *2016 IEEE Symposium in Low-Power and High-Speed Chips (COOL CHIPS XIX)*, pp. 1–3. IEEE, 2016.

[JCP05] Ju-Wook Jang, Seonil B Choi, and Viktor K. Prasanna. "Energy-and time-efficient matrix multiplication on FPGAs." *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, **13**(11):1305–1319, 2005.

[JG15] Hanyu Jiang and Narayan Ganesan. "Fine-grained acceleration of HMMER 3.0 via architecture-aware optimization on massively parallel processors." In *Parallel and*

Distributed Processing Symposium Workshop (IPDPSW), 2015 IEEE International, pp. 375–383. IEEE, 2015.

[JLW20] Liancheng Jia, Liqiang Lu, Xuechao Wei, and Yun Liang. "Generating systolic array accelerators with reusable blocks." *IEEE Micro*, **40**(4):85–92, 2020.

[JPJ11] Thomas B. Jablin, Prakash Prabhu, James A. Jablin, Nick P. Johnson, Stephen R. Beard, and David I. August. "Automatic CPU-GPU communication management and optimization." In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '11, pp. 142–151, New York, NY, USA, 2011. ACM.

[JYP17] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. "In-datacenter performance analysis of a tensor processing unit." In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, pp. 1–12, 2017.

[JYP18] Norman P. Jouppi, Cliff Young, Nishant Patil, and David Patterson. "A domain-specific architecture for deep neural networks." `https://cacm.acm.org/magazines/2018/9/230571-a-domain-specific-architecture-for-deep-neural-networks/fulltext`, 2018.

[JYS20] Weiwen Jiang, Lei Yang, Edwin Hsing-Mean Sha, Qingfeng Zhuge, Shouzhen Gu, Sakyasingha Dasgupta, Yiyu Shi, and Jingtong Hu. "Hardware/software co-exploration of neural architectures." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **39**(12):4805–4815, 2020.

[KCS20] Hyoukjun Kwon, Prasanth Chatarasi, Vivek Sarkar, Tushar Krishna, Michael Pellauer, and Angshuman Parashar. "Maestro: A data-centric approach to understand reuse, performance, and hardware cost of DNN mappings." *IEEE micro*, **40**(3):20–29, 2020.

[KFP18] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, et al. "Spatial: A language and compiler for application accelerators." In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 296–311, 2018.

[KJK20] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. "ConfuciuX: Autonomous hardware resource assignment for DNN accelerators using reinforcement learning." In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 622–636. IEEE, 2020.

[KJK21] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. "ConfuciuX." `https://github.com/maestro-project/confuciux`, 2021.

[KK20] Sheng-Chun Kao and Tushar Krishna. "GAMMA: Automating the HW mapping of DNN models on accelerators via genetic algorithm." In *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9. IEEE, 2020.

[KL79] H. T. Kung and Charles E Leiserson. "Systolic arrays (for VLSI)." In *Sparse Matrix Proceedings 1978*, pp. 256–282. Society for Industrial and Applied Mathematics, 1979.

[KLP21] Hyoukjun Kwon, Liangzhen Lai, Michael Pellauer, Tushar Krishna, Yu-Hsin Chen, and Vikas Chandra. "Heterogeneous dataflow accelerators for multi-DNN workloads." In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 71–83. IEEE, 2021.

[KMZ19] H. T. Kung, Bradley McDanel, and Sai Qian Zhang. "Packing sparse convolutional neural networks for efficient systolic array implementations: Column combining under joint optimization." In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 821–834, 2019.

[KSK18] Hyoukjun Kwon, Ananda Samajdar, and Tushar Krishna. "Maeri: Enabling flexible dataflow mapping over DNN accelerators via reconfigurable interconnects." *ACM SIGPLAN Notices*, **53**(2):461–475, 2018.

[Kun88] Sun Yuan Kung. "VLSI array processors." *Englewood Cliffs*, 1988.

[KY82] Ashok V. Kulkarni and David W. L. Yen. "Systolic processing and an implementation for signal and image processing." *IEEE Transactions on Computers*, 1982.

[KY85] S Kalson and Kung Yao. "A systolic array for linearly constrained least squares filtering." In *ICASSP'85. IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 10, pp. 977–980. IEEE, 1985.

[LAB20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. "MLIR: A compiler infrastructure for the end of Moore's law." *arXiv preprint arXiv:2002.11054*, 2020.

[Leg19] LegUp. "High-Level Synthesis with LegUp." `http://legup.eecg.utoronto.ca/`, August 2019.

[LGW21] Liqiang Lu, Naiqing Guan, Yuyue Wang, Liancheng Jia, Zizhang Luo, Jieming Yin, Jason Cong, and Yun Liang. "TENET: A framework for modeling tensor dataflow based on relation-centric notation." *arXiv preprint arXiv:2105.01892*, 2021.

[LL85] Richard J. Lipton and Daniel Lopresti. "A systolic array for rapid string comparison." In *Proceedings of the Chapel Hill Conference on VLSI*, pp. 363–376. Chapel Hill, NC, 1985.

[LL97] Amy W Lim and Monica S Lam. "Maximizing parallelism and minimizing synchronization with affine transforms." In *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pp. 201–214, 1997.

[LLV21] LLVM. "Circuit IR Compilers and Tools." `https://github.com/llvm/circt`, 2021.

[LN08] Guoqing Lu and Jun Ni. "Highlighting computations in bioscience and bioinformatics: review of the Symposium of Computations in Bioinformatics and Bioscience (SCBB07)." *BMC Bioinformatics*, **9**(6):S1, 2008.

[LQR99] Dominique Lavenier, Patrice Quinton, and Sanjay Rajopadhye. "Advanced systolic design." *Digital Signal Processing for Multimedia Systems*, pp. 657–692, 1999.

[LRZ20] Yi-Hsiang Lai, Hongbo Rong, Size Zheng, Weihao Zhang, Xiuping Cui, Yunshan Jia, Jie Wang, Brendan Sullivan, Zhiru Zhang, Yun Liang, Jason Cong, Nithin George, Jose Alvarez, Christopher Hughes, and Pradeep Dubey. "SuSy: A programming model for productive construction of high-performance systolic arrays on FPGAs." In *2020 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2020.

[LSV19] Rui Li, Aravind Sukumaran-Rajam, Richard Veras, Tze Meng Low, Fabrice Rastello, Atanas Rountev, and Ponnuswamy Sadayappan. "Analytical cache modeling and tilesize optimization for tensor contractions." In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–13, 2019.

[LWB17] Junyi Liu, John Wickerson, Samuel Bayliss, and George A. Constantinides. "Polyhedral-based dynamic loop pipelining for high-level synthesis." *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, **37**(9):1802–1815, 2017.

[LXS21] Rui Li, Yufan Xu, Aravind Sukumaran-Rajam, Atanas Rountev, and P. Sadayappan. "Analytical characterization and design space exploration for optimization of CNNs." In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 928–942, 2021.

[MKN18] Duncan J. M. Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip HW Leong. "A customizable matrix multiplication framework for the Intel HARPv2 Xeon+ FPGA platform: A deep learning case study." In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 107–116, 2018.

[MMR01] Manju Manjunathaiah, Graham M. Megson, S. Rajopadhye, and Tanguy Risset. "Uniformization of affine dependence programs for parallel embedded system design." In *International Conference on Parallel Processing, 2001.*, pp. 205–213. IEEE, 2001.

[Mol83] Dan I. Moldovan. "On the design of algorithms for VLSI systolic arrays." *Proceedings of the IEEE*, **71**(1):113–120, 1983.

[MV08] Svetlin A. Manavski and Giorgio Valle. "CUDA compatible GPU cards as efficient hardware accelerators for Smith-Waterman sequence alignment." *BMC Bioinformatics*, **9**(2):S10, 2008.

[Net20] Netlib. "LAPACK — Linear Algebra Package." `http://www.netlib.org/lapack/`, 2020.

[Nog21] Fernando Nogueira. "Bayesian Optimization: Open source constrained global optimization tool for Python." `https://github.com/fmfn/BayesianOptimization`, 2021.

[NSK19] Luigi Nardi, Artur Souza, David Koeplinger, and Kunle Olukotun. "HyperMapper: A practical design space exploration framework." In *2019 IEEE 27th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 425–426. IEEE, 2019.

[Nvi16] Nvidia. "CUDA C Programming Guide." `http://docs.nvidia.com/cuda/cuda-c-programming-guide/#axzz4NVazmUbb`, 2016.

[PEW19] Phitchaya Mangpo Phothilimthana, Archibald Samuel Elliott, An Wang, Abhinav Jangda, Bastian Hagedorn, Henrik Barthels, Samuel J. Kaufman, Vinod Grover, Emina Torlak, and Rastislav Bodik. "Swizzle inventor: Data movement synthesis for GPU kernels." In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 65–78. ACM, 2019.

[Pou21] Louis-Noel Pouchet. "PoCC: the Polyhedral Compiler Collection." `http://web.cs.ucla.edu/~pouchet/software/pocc/`, 2021.

[PRS19] Angshuman Parashar, Priyanka Raina, Yakun Sophia Shao, Yu-Hsin Chen, Victor A. Ying, Anurag Mukkara, Rangharajan Venkatesan, Brucek Khailany, Stephen W Keckler, and Joel Emer. "Timeloop: A systematic approach to DNN accelerator evaluation." In *2019 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 304–315. IEEE, 2019.

[PSW91] Craig Peterson, James Sutton, and Paul Wiley. "iWarp: A 100-MOPS, LIW microprocessor for multicomputers." *IEEE Micro*, **11**(3):26–29, 1991.

[PZS13] Louis-Noel Pouchet, Peng Zhang, Ponnuswamy Sadayappan, and Jason Cong. "Polyhedral-based data reuse optimization for configurable computing." In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, pp. 29–38, 2013.

[Raj88] S. V. Rajopadhye. "Systolic arrays for LU decomposition." In *1988., IEEE International Symposium on Circuits and Systems*, pp. 2513–2516. IEEE, 1988.

[Ras03] Carl Edward Rasmussen. "Gaussian processes in machine learning." In *Summer School on Machine Learning*, pp. 63–71. Springer, 2003.

[RBA13] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. "Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines." *Acm Sigplan Notices*, **48**(6):519–530, 2013.

[RF16] Chris Rauer and N. Finamore. "Accelerating genomics research with openCL and FPGAs." *Altera, Now Part of Intel, Tech. Rep*, 2016.

[RHA17] Brandon Reagen, José Miguel Hernández-Lobato, Robert Adolf, Michael Gelbart, Paul Whatmough, Gu-Yeon Wei, and David Brooks. "A case for efficient accelerator design space exploration via Bayesian optimization." In *2017 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 1–6. IEEE, 2017.

[RK88] Sailesh K. Rao and Thomas Kailath. "Regular iterative algorithms and their implementation on processor arrays." *Proceedings of the IEEE*, **76**(3):259–269, 1988.

[Ron17] Hongbo Rong. "Programmatic control of a compiler for generating high-performance spatial hardware." *arXiv preprint arXiv:1711.07606*, 2017.

[RRA93] Ulrich Ramacher, Wolfgang Raab, J. Anlauf, Ulrich Hachmann, Jörg Beichter, Nico Brüls, Matthias Wesseling, Elisabeth Sicheneder, Reinhard Männer, Joachim Gläß, et al. "Multiprocessor and memory architecture of the neurocomputer SYNAPSE-1." *International Journal of Neural Systems*, **4**(04):333–336, 1993.

[RSA15] Shanshan Ren, V. M. Sima, and Z. Al-Ars. "FPGA acceleration of the pair-HMMs forward algorithm for DNA sequence analysis." In *Bioinformatics and Biomedicine (BIBM), 2015 IEEE International Conference on*, pp. 1465–1470, Nov 2015.

[Rut19] Mitja Rutnik. "YouTube in numbers: Monthly views, most popular video, and more fun stats." `https://www.androidauthority.com/youtube-stats-1016070/`, August 2019.

[SAM02] Robert Schreiber, Shail Aditya, Scott Mahlke, Vinod Kathail, B Ramakrishna Rau, Darren Cronquist, and Mukund Sivaraman. "PICO-NPA: High-level synthesis of non-programmable hardware accelerators." *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, **31**(2):127–142, 2002.

[Sci21] Scipy. "Basin-hopping package." `https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.basinhopping.html`, 2021.

[SFM17] Yongming Shen, Michael Ferdman, and Peter Milder. "Maximizing CNN accelerator efficiency through resource partitioning." In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 535–547. IEEE, 2017.

[SHZ18] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. "Mobilenetv2: Inverted residuals and linear bottlenecks." In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 4510–4520, 2018.

[Sie21] Siemens. "Catapult High-Level Synthesis." `https://eda.sw.siemens.com/en-US/ic/ic-design/high-level-synthesis-and-verification-platform/`, 2021.

[SM00] Andrew Stone and Elias S. Manolakos. "DG2VHDL: A tool to facilitate the high level synthesis of parallel processing array architectures." *Journal of VLSI signal processing systems for signal, image and video technology*, Feb 2000.

[SQA20] Nikola Samardzic, Weikang Qiao, Vaibhav Aggarwal, Mau-Chung Frank Chang, and Jason Cong. "Bonsai: High-performance adaptive merge tree sorting." In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 282–294. IEEE, 2020.

[SRB19] Nitish Srivastava, Hongbo Rong, Prithayan Barua, Guanyu Feng, Huanqi Cao, Zhiru Zhang, David Albonesi, Vivek Sarkar, Wenguang Chen, Paul Petersen, et al. "T2S-Tensor: Productively generating high-performance spatial hardware for dense tensor computations." In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pp. 181–189. IEEE, 2019.

[SSS97] Bertil Schmidt, Manfred Schimmler, and Heiko Schröder. "Morphological hough transform on the instruction systolic array." In *European Conference on Parallel Processing*, pp. 798–806. Springer, 1997.

[SW81] Temple F. Smith and Michael S. Waterman. "Identification of common molecular subsequences." *Journal of Molecular Biology*, **147**(1):195–197, 1981.

[SWC20] Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. "End-to-end optimization of deep learning applications." In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 133–139, 2020.

[SZ14] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition." *arXiv preprint arXiv:1409.1556*, 2014.

[THZ20] James Thomas, Pat Hanrahan, and Matei Zaharia. "Fleet: A framework for massively parallel streaming on FPGAs." In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 639–651, 2020.

[VCC13] Sven Verdoolaege, Juan Carlos Juega, Albert Cohen, Jose Ignacio Gomez, Christian Tenllado, and Francky Catthoor. "Polyhedral parallel code generation for CUDA." *ACM Transactions on Architecture and Code Optimization (TACO)*, **9**(4):54, 2013.

[Ver10] Sven Verdoolaege. "isl: An integer set library for the polyhedral model." In *International Congress on Mathematical Software*, pp. 299–302. Springer, 2010.

[VG12] Sven Verdoolaege and Tobias Grosser. "Polyhedral extraction tool." In *Second International Workshop on Polyhedral Compilation Techniques (IMPACT'12)*, pp. 1–16, 2012.

[VGG14] Sven Verdoolaege, Serge Guelton, Tobias Grosser, and Albert Cohen. "Schedule trees." In *International Workshop on Polyhedral Compilation Techniques*, 2014.

[VZT19] Nicolas Vasilache, Oleksandr Zinenko, Theodoros Theodoridis, Priya Goyal, Zachary Devito, William S. Moses, Sven Verdoolaege, Andrew Adams, and Albert

Cohen. "The next 700 accelerated layers: From mathematical expressions of network computation graphs to accelerated GPU kernels, automatically." *ACM Transactions on Architecture and Code Optimization (TACO)*, **16**(4):1–26, 2019.

[WD08] Jingming Wang and Babak Daneshrad. "A universal systolic array for linear MIMO detections." In *2008 IEEE Wireless Communications and Networking Conference*, pp. 147–152. IEEE, 2008.

[WGC21] Jie Wang, Licheng Guo, and Jason Cong. "AutoSA: A polyhedral compiler for high-performance systolic arrays on FPGA." In *Proceedings of the 2021 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, 2021.

[Wik19] WikiChip. "FSD Chip - Tesla." `https://en.wikichip.org/wiki/tesla_(car_company)/fsd_chip`, December 2019.

[WLD20] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. "DSAGEN: Synthesizing programmable spatial accelerators." In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pp. 268–281. IEEE, 2020.

[WLL18] Xuechao Wei, Yun Liang, Xiuhong Li, Cody Hao Yu, Peng Zhang, and Jason Cong. "TGPA: Tile-grained pipeline architecture for low latency CNN inference." In *Proceedings of the International Conference on Computer-Aided Design*, pp. 1–8, 2018.

[WXC17] Jie Wang, Xinfeng Xie, and Jason Cong. "Communication optimization on GPU: A case study of sequence alignment algorithms." In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pp. 72–81. IEEE, 2017.

[WYZ17] Xuechao Wei, Cody Hao Yu, Peng Zhang, Youxiang Chen, Yuxin Wang, Han Hu, Yun Liang, and Jason Cong. "Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs." In *Proceedings of the 54th Annual Design Automation Conference 2017*, pp. 1–6, 2017.

[WZS14] Endong Wang, Qing Zhang, Bo Shen, Guangyong Zhang, Xiaowei Lu, Qing Wu, and Yajuan Wang. "Intel math kernel library." In *High-Performance Computing on the Intel Xeon Phi*, pp. 167–188. Springer, 2014.

[XF10] S. Xiao and W. C. Feng. "Inter-block GPU communication via fast barrier synchronization." In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pp. 1–12, April 2010.

[Xil19a] Xilinx. "Vivado High-Level Synthesis." `https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html`, August 2019.

[Xil19b] Xilinx. "Xilinx Machine Learning Suite." `https://www.xilinx.com/products/acceleration-solutions/xilinx-machine-learning-suite.html`, August 2019.

[YCJ96] Jinn-Wang Yeh, Wen-Jiunn Cheng, and Chein-Wei Jen. "VASS—A VLSI array system synthesizer." *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, May 1996.

[YGL20] Xuan Yang, Mingyu Gao, Qiaoyi Liu, Jeff Setter, Jing Pu, Ankita Nayak, Steven Bell, Kaidi Cao, Heonjae Ha, Priyanka Raina, et al. "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*, pp. 369–383, 2020.

[YP20] Tomofumi Yuki and Louis-Noël Pouchet. "PolyBench/C 4.2." `https://sourceforge.net/projects/polybench/`, 2020.

[ZJS20] Lianmin Zheng, Chengfan Jia, Minmin Sun, Zhao Wu, Cody Hao Yu, Ameer Haj-Ali, Yida Wang, Jun Yang, Danyang Zhuo, Koushik Sen, et al. "Ansor: Generating high-performance tensor programs for deep learning." In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pp. 863–879, 2020.

[ZL16] Barret Zoph and Quoc V. Le. "Neural architecture search with reinforcement learning." *arXiv preprint arXiv:1611.01578*, 2016.

[ZLS15] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason 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*, pp. 161–170, 2015.

[ZWZ18] Xiaofan Zhang, Junsong Wang, Chao Zhu, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. "DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs." In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 1–8. IEEE, 2018.

[ZYW20] Xiaofan Zhang, Hanchen Ye, Junsong Wang, Yonghua Lin, Jinjun Xiong, Wen-mei Hwu, and Deming Chen. "DNNExplorer: A framework for modeling and exploring a novel paradigm of FPGA-based DNN accelerator." In *Proceedings of the 39th International Conference on Computer-Aided Design*, pp. 1–9, 2020.