

**UCLA**

**UCLA Electronic Theses and Dissertations**

**Title**

Customized Computing and Machine Learning

**Permalink**

<https://escholarship.org/uc/item/4kw0h4nz>

**Author**

Sohrabizadeh, Atefeh

**Publication Date**

2024

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA  
Los Angeles

Customized Computing and Machine Learning

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

by

Atefeh Sohrabizadeh

2024

© Copyright by  
Atefeh Sohrabizadeh  
2024

# ABSTRACT OF THE DISSERTATION

Customized Computing and Machine Learning

by

Atefeh Sohrabizadeh

Doctor of Philosophy in Computer Science

University of California, Los Angeles, 2024

Professor Jingsheng Jason Cong, Chair

Nowadays, abundant data across various domains necessitate high-performance computing capabilities. While we used to be able to answer this need by scaling the frequency, the breakdown of Dennard’s scaling has rendered this approach obsolete. On the other hand, Domain-specific Accelerators (DSAs) have gained a growing interest since they can offer high performance while being energy efficient. This stems from several factors, such as, 1) they support utilizing special data types and operations, 2) they offer massive parallelism, 3) one can customize the memory access, 4) customizing the control/data path helps with amortizing the overhead of fixed instructions, and 5) one has the option of co-designing the algorithm with the hardware.

Unfortunately, despite the huge speedups that DSAs can deliver compared to general-purpose processors, their programmability has not caught up. In the past few decades, High-Level Synthesis (HLS) tools were introduced to raise the abstraction level and free designers from delving into architecture details at the circuit level. While HLS can significantly reduce the efforts involved in the hardware architecture design, not every HLS code yields optimal performance, requiring designers to articulate the most suitable microarchitecture for the target application. This can affect the design turnaround times as there are more choices to explore at a higher level. Moreover, this limitation has confined the DSA community primarily to hardware designers, impeding widespread adoption. This dissertation endeavors to alleviate this problem by combining customized computing and machine learn-

ing. Consequently, this dissertation consists of two core parts: 1) customized computing tailored for machine learning applications, and 2) machine learning employed to automate the optimization process of customized computing. Our focus will be on FPGAs as their cost-effective nature and rapid prototyping capabilities make them especially suitable for our research.

The large amounts of data available in data centers have motivated researchers to develop machine learning algorithms for processing them. Given that a significant portion of data stored in these centers exists in the form of images or graphs, our attention is directed towards two prominent algorithms designed for such tasks: Convolutional Neural Network (CNN) and Graph Convolutional Network (GCN). In the first part of the dissertation, we develop architecture templates for accelerating these applications. This approach facilitates a reduction in the development cycle, allowing the instantiation of module templates with customizable parameters based on the specific target application.

In the second part of the dissertation, we move our focus to general applications and work on automating their optimization steps including design space exploration and performance/area modeling. Therefore, we structure our problem in a way that can be fed into the learning algorithms. We develop a highly efficient bottleneck optimizer to explore the search space. We also explore different learning algorithms including multi-layer perceptron, graph neural networks, attention networks, jumping knowledge networks, etc., aiming to create a performance predictor that is both highly accurate and robust. Our studies show that we can optimize the microarchitecture of general applications quickly using our automated tools. This can open new doors to those without hardware knowledge to try customized computing which in turn helps to broaden the FPGA community and further improve its technology.

The dissertation of Atefeh Sohrabizadeh is approved.

Sitao Huang

Anthony John Nowatzki

Yizhou Sun

Jingsheng Jason Cong, Committee Chair

University of California, Los Angeles

2024

*To my family.*

## TABLE OF CONTENTS

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Dissertation Overview	3
1.1.1	Customized Computing for Machine Learning Acceleration	3
1.1.2	Machine learning for Designing Customized Accelerators	6
<b>2</b>	<b>Background</b>	<b>10</b>
2.1	High-Level Synthesis (HLS)	10
2.1.1	The Merlin Compiler	12
2.2	Convolutional Neural Networks	15
2.2.1	Other Convolutional Layers	16
2.3	Graph Neural Networks	17
2.3.1	Common Graph Convolutional Layers	18
2.4	Related Works	19
2.4.1	Convolutional Neural Network Accelerators	19
2.4.2	Graph Convolutional Network Accelerators	21
2.4.3	Machine Learning for Electronic Design Automation	23
2.4.4	HLS Performance and Resource Modeling	24
2.4.5	Design Space Exploration for HLS	26
2.4.6	Code Transformation for High-Performance Computing	28
<b>I</b>	<b>Customized Computing for Machine Learning Acceleration</b>	<b>30</b>
<b>3</b>	<b>FlexCNN: End-to-end Acceleration of Convolutional Neural Networks</b>	<b>32</b>



3.1	Introduction . . . . .	33
3.2	Application Driver - OpenPose . . . . .	36
3.3	The FlexCNN Architecture . . . . .	37
3.3.1	Dynamic Tiling . . . . .	39
3.3.2	Data Layout Optimization . . . . .	41
3.4	TensorFlow Integration . . . . .	43
3.5	FlexCNN Compilation System . . . . .	45
3.6	Experimental Results . . . . .	46
3.6.1	Experiment Setup . . . . .	46
3.6.2	Hardware Optimizations . . . . .	46
3.6.3	Integration Optimizations . . . . .	48
3.6.4	Comparative Studies . . . . .	50
3.7	Conclusion . . . . .	50
<b>4</b>	<b>StreamGCN: Accelerating Graph Convolutional Networks with Streaming Processing . . . . .</b>	<b>52</b>
4.1	Introduction . . . . .	52
4.2	Background . . . . .	56
4.2.1	Graph Convolutional Network Computation Details . . . . .	56
4.3	StreamGCN Architecture . . . . .	57
4.3.1	StreamGCN Design Principles . . . . .	57
4.3.2	Baseline Architecture . . . . .	58
4.3.3	Extension 1: Multi-layer Support and Inter-layer Pipelining . . . . .	61
4.3.4	Extension 2: In Situ Sparsity Support in FT Step . . . . .	62
4.4	StreamGCN Application to Graph Matching . . . . .	65

4.4.1	SimGNN . . . . .	65
4.4.2	Att Architecture . . . . .	66
4.4.3	NTN + Fully-connected Network Architecture . . . . .	67
4.4.4	Putting It All Together . . . . .	68
4.5	Experimental Results . . . . .	69
4.5.1	Benchmark . . . . .	69
4.5.2	Experimental Setup . . . . .	69
4.5.3	Impact of GCN Architecture Optimizations . . . . .	70
4.5.4	End-to-end Acceleration of SimGNN . . . . .	71
4.6	Other Related Works . . . . .	74
4.7	Conclusion . . . . .	75
<b>II Machine Learning for Designing Customized Accelerators</b>		<b>76</b>
<b>5 AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators</b>		<b>78</b>
5.1	Introduction . . . . .	79
5.2	Problem Formulation . . . . .	84
5.3	The AutoDSE Framework . . . . .	86
5.3.1	Merlin Compiler and Design Space Definition . . . . .	86
5.3.2	Framework Overview . . . . .	88
5.4	AutoDSE Methodology . . . . .	89
5.4.1	Application-oblivious Heuristics . . . . .	89
5.4.2	AutoDSE Exploring Strategy - Bottleneck-guided Coordinate Optimizer	93
5.4.3	Efficient Design Space Representation . . . . .	98

5.4.4	Design Space Partitioning	101
5.5	Evaluation	102
5.5.1	Experimental Setup	102
5.5.2	Impact of Parameter Ordering	103
5.5.3	Evaluation of Optimization Techniques	104
5.5.4	Comparison with Other DSE Approaches	106
5.5.5	Comparison with Expert-level Manual HLS Designs	107
5.5.6	Frequency of Employed Parameters	109
5.6	Conclusion	110
<b>6</b>	<b>GNN-DSE: Automated Accelerator Optimization Aided by Graph Neural Networks</b>	<b>113</b>
6.1	Introduction	114
6.2	Problem Formulation	116
6.3	Our Proposed Methodology	117
6.3.1	Database Generation	118
6.3.2	Program Representation	119
6.3.3	Predictive Model	121
6.3.4	Design Space Exploration	128
6.4	Evaluation	129
6.4.1	Experimental Setup	129
6.4.2	Model Evaluation	130
6.4.3	Results of Design Space Exploration	134
6.4.4	Results on Unseen Kernels	135
6.5	Conclusion	136

<b>7</b>	<b>HLSyn: Benchmark for High-Level Synthesis Targeted to FPGAs</b>	<b>137</b>
<b>8</b>	<b>HARP: Robust GNN-based Representation Learning for HLS</b>	<b>143</b>
8.1	Introduction	144
8.2	HARP Methodology	147
8.2.1	Hierarchical Graph Representation	148
8.2.2	Decoupling Program and Transformation	151
8.2.3	Transfer Learning	159
8.3	Experimental Results	162
8.3.1	Experimental Setup	162
8.3.2	Model Accuracy	162
8.3.3	DSE Results	165
8.3.4	Ablation Study: Transfer Learning with Abundant Data	167
8.3.5	Ablation Study: Impact of the Classification Model	169
8.3.6	Ablation Study: Alternative Hierarchy Structure and Larger Model	169
8.4	Conclusion	171
<b>9</b>	<b>Concluding Remarks</b>	<b>173</b>
<b>A</b>	<b>Appendix</b>	<b>180</b>
A.1	Optimized HLS Code for CNN	180
A.2	Correlation Matrices of HLSyn Benchmark	183

## LIST OF FIGURES

1.1	The overview of the dissertation. . . . .	4
2.1	High-level synthesis design flow. . . . .	11
2.2	Structure of our targeted convolutional layers along with their respective weights. . . . .	16
2.3	The computation of a GCN layer on a sample node. . . . .	18
2.4	How the attention coefficient is calculated in a GAT layer. . . . .	19
P.1	Overview of Part 1 – Architecture-guided Optimization. . . . .	31
3.1	Performance comparison of hardware accelerators using uniform and dynamic tiling factors for the first 24 convolutional layers in the CNN network of Figure 3.3. . . . .	34
3.2	Runtime breakdown of an FPGA-based CNN acceleration pipeline in TensorFlow. . . . .	35
3.3	OpenPose-V2 network topology. . . . .	37
3.4	The FlexCNN architecture overview. . . . .	38
3.5	Architecture support for dynamic tiling in the Depth Conv module for a $3 \times 3$ filter with Tw (tiling factors for width) of size 6/8/10. . . . .	41
3.6	Data organization to apply the concatenation operation on the fly. Example is taken from the OpenPose network. . . . .	41
3.7	First level of pipeline when integrating the FPGA to Tensorflow. . . . .	44
3.8	The overview of the Process Graph stage in the first level of pipeline, creating the second level of pipelining. . . . .	44
3.9	FlexCNN compilation system overview. . . . .	45
3.10	Layers in Table 3.5 under the roofline model. . . . .	49
4.1	The optimized computation order to reduce II for the Feature Transformation step in GCN. . . . .	59

4.2	Baseline architecture for accelerating GCN: intra-layer pipelining between MULT module (multiplication unit for Feature Transformation step) and ACG Module (accumulation unit for Feature Transformation step + Aggregation step) . . . . .	60
4.3	High-level overview of GCN accelerator architecture in StreamGCN. . . . .	61
4.4	The benefit of streaming the node embeddings and mapping the weights to the SIMD dimension when processing the Feature Transformation step of a GCN layer. CU stands for computation unit. Colored cells in the matrix show non-zero values and white cells show zero values. . . . .	63
4.5	StreamGCN architecture support for sparse computation in the Feature Transformation step of GCN. . . . .	64
4.6	Architecture overview of the second stage of SimGNN in StreamGCN: Att module implementing the global context-aware attention layer. . . . .	67
4.7	Architecture overview of the last two stages of SimGNN in StreamGCN: NTN and FCN module implementing the neural tensor network and fully connected network. . . . .	68
4.8	Resource breakdown of SimGNN accelerator on AMD Xilinx Alveo U280 board. . . . .	72
P.2	Overview of Part 2 – General Microarchitecture Optimization. . . . .	77
5.1	The AutoDSE framework overview. . . . .	89
5.2	Performance speedup of the design found by S2FA [Yu+18] compared to the manual design over time. . . . .	91
5.3	The proposed design space representation and its impact on DSE. P1 and P2 denote the <code>pipeline</code> and <code>parallel</code> pragmas of loop <code>j</code> in Code 5.1, respectively. . . . .	99
5.4	Comparison of performance speedup over an Intel Xeon CPU core with different compilation strategies: no pragmas, pragma-augmented design generated by different approaches, and manually optimized design. . . . .	105

5.5	Performance speedup of generated designs compared to the manual designs over time using AutoDSE for four cases where the bottleneck-guided optimizer demonstrated significant impact. . . . .	106
5.6	Performance speedup and the number of reduced pragmas using AutoDSE compared to the Vision kernels of Xilinx Vitis libraries [Xilb] . . . . .	110
5.7	The number of candidate pragmas vs. the number of pragmas that were adjusted to their non-default values for the optimal designs of each target kernel. . . . .	111
6.1	High-level overview of our model-based frameworks for optimizing the design. . . . .	117
6.2	Database generator for a model-based DSE framework such as GNN-DSE. . . . .	118
6.3	Part of the LLVM IR of Code 6.1 and its graph representation illustrating the different kinds of nodes and edges in GNN-DSE’s representation. . . . .	122
6.4	Graph generator of GNN-DSE. . . . .	122
6.5	The architecture of GNN-DSE’s predictive model. . . . .	123
6.6	Node attention scores of a design of the stencil kernel determined by GNN-DSE’s model. The size of each circle corresponds to its attention score, with larger circles indicating higher attention. . . . .	125
6.7	t-SNE [MH08] visualization of the design configurations of stencil. Each point represents a different pragma combination with colors indicating its latency value. While the initial embeddings tend to cluster points irrespective of their latency values, GNN-DSE’s embeddings achieve a more distinct and informative clustering. . . . .	127
6.8	Correlation matrix for the database used in training GNN-DSE’s model with 1 (-1) showing a perfect positive (negative) correlation and 0 indicating no correlation. . . . .	132
6.9	Speedup comparison of GNN-DSE relative to the optimal design in the initial database. Following each round of DSE, the top designs are added to the database to retrain the model and improve the discovered optimal designs. . . . .	134

8.1	The comparison of design objectives when the HLS tool is changed from AMD Xilinx SDx 2018.3 to Vitis 2020.2. Each point on the graph represents a distinct design configuration, plotted against the $y = x$ line. . . . .	148
8.2	(a) Two sample code snippets; (b) The hierarchical structures of the two sample code snippets, showing only the pseudo nodes and the connections between them; (c) A sample hierarchical graph focusing on demonstrating the pseudo nodes and their connections. . . . .	149
8.3	Separating the vector representation of the program P and its transformation T. Distinct vectors are generated for each one. A further reconstruction loss with an autoencoder is used to enhance the influence of pragmas on the T vector. . .	152
8.4	t-SNE visualization of the generated embeddings that are color-coded by the kernel name. . . . .	155
8.5	t-SNE visualization of the T vector compared to input pragma options that are color-coded by the performance value (log of speedup). Warmer colors indicate higher performance (lower latency). . . . .	156
8.6	Modeling pragmas as function transformations using NPT: each pragma type is modeled as a learnable MLP which takes in the embeddings of the pseudo node of the pragma block along with the pragma option. A second level of MLP is used to merge the results. . . . .	157
8.7	Correlation matrix of the design objectives resulted from AMD Xilinx SDx 2018.3 (v18) and Vitis 2020.2 (v20). The data is taken from our HLSyn dataset (Chapter 7). . . . .	161
8.8	Impact of alternative hierarchy structure and larger model on the Mean Squared Error (MSE) on test set. In the new hierarchy graph, we define new pseudo nodes for ‘for’ loops. . . . .	171
A.1	Correlation matrix of the design objectives resulted from AMD Xilinx SDx 2018.3 (v18) and Vitis 2021.1 (v21). . . . .	182



A.2 Correlation matrix of the design objectives resulted from AMD Xilinx Vitis 2020.2 (v20) and Vitis 2021.1 (v21). . . . .	183
---	-----

## LIST OF TABLES

2.1	Common AMD Xilinx HLS Pragmas. . . . .	11
2.2	The Merlin pragmas with architecture structure. . . . .	13
3.1	FlexCNN design parameters with their explanations. . . . .	39
3.2	Frequency and resource utilization of FlexCNN implementing OpenPose-V2 on the AMD Xilinx VCU1525 board. . . . .	47
3.3	Performance of FlexCNN under different settings tested on OpenPose-V2. All Dynamic refers to the case where dynamic tiling and data layout reorganization are applied. All Uniform maintains a uniform tiling factor and data layout across all layers, specifically chosen for optimal performance within the uniform setting. . . . .	47
3.4	Performance impacts of dynamic tiling and data layout transformation for a given layer in OpenPose-V2 with $1 \times 1$ filter size. . . . .	48
3.5	Performance of FlexCNN across different convolutional layers on the AMD Xilinx VCU1525 board. . . . .	48
3.6	Performance impacts of the integration (FlexCNN to TensorFlow) optimizations. The first level overlaps TensorFlow’s overheads with the FPGA-related steps. The second one further overlaps FPGA’s computation with data movement steps. . . . .	49
3.7	Performance comparison of different hardware platforms running OpenPose-V2. . . . .	51
4.1	Properties of StreamGCN compared to the state-of-the-art GCN accelerators. . . . .	55
4.2	Summary of architecture parameters for the accelerator of each GCN layer in StreamGCN. . . . .	65
4.3	Properties of the FPGAs used to implement StreamGCN. . . . .	70

4.4	Impact of the GCN architecture optimizations tested on AMD Xilinx Alveo U280 board. The meaning of design parameters is summarized in Table 4.2. Baseline design shows a single set of design parameters because it uses the same hardware for all the layers. . . . .	70
4.5	Performance and resource utilization of a StreamGCN design accelerating SimGNN on different target FPGAs. . . . .	72
4.6	Performance comparison of running SimGNN on different hardware platforms. . . . .	73
5.1	Analysis of poor performance in Code 5.1. . . . .	81
5.2	Design space definition based on the Merlin pragmas. . . . .	88
5.3	Performance and area compared to the base design when the parameters of Line 16 in Code 5.1 change. TIMEOUT is set to 60 minutes. The results suggest that applying fine-grained optimization first lets the HLS tool synthesize the design easier. . . . .	98
5.4	Effect of the parameter ordering on the performance and area of the generated optimized design for Code 5.1 after running the DSE for 7 hours. . . . .	104
5.5	Performance speedup of our approach compared to S2FA [Yu+18], Lattice-traversing DSE [FAP18b], and Gaussian process-based Bayesian Optimization [Sun+21] . . . . .	107
5.6	Average (geometric mean) performance speedup of the Vitis tool, the Merlin Compiler, and AutoDSE over the manually optimized kernels from Xilinx Vitis libraries. The manual designs are the original kernels from the library and are summarized in the Manually Optimized column for Vitis. The performance of those designs is compared to when the optimization pragmas we search for (UNROLL, PIPELINE, ARRAY_PARTITION, DEPENDENCE, LOOP_FLATTEN, and INLINE) are removed and the code is passed to the three different tools. AutoDSE is able to retrieve the removed pragmas automatically. . . . .	108

6.1	The statistics for our target kernels including the number of pragmas, the size of the solution space, and the database size utilized for training the GNN-DSE model.	131
6.2	Model evaluation on the test set of our database. RMSE loss is used as the evaluation metric for the regression task. The sum of all losses is reported in the ‘All’ column. For the classification task, the accuracy and F1-score are reported.	133
6.3	GNN-DSE’s performance on unseen kernels without further adaptation. The runtime speedup numbers are with respect to AutoDSE, after running it for up to 21 hours. GNN-DSE could achieve about the same performance but in much less time.	136
7.1	Statistics of HLSyn dataset which consists of 41 unique kernels among which 21 of them are shared in all databases. We denote the normalized value of latency as <code>perf</code> as it corresponds to the base-2 logarithm of the speedup relative to a reference latency value.	139
7.2	Statistics of each kernel in the HLSyn dataset. <code>v18</code> , <code>v20</code> , and <code>v21</code> refer to AMD Xilinx SDx 2018.3, Vitis 2020.2, and Vitis 2021.1, respectively. Difference columns indicate the description of the application’s functionality, number of nodes and edges in the hierarchical graph representation, number of candidate pragmas, total number of points in the solution space, and the number of points in each version of the database.	139
8.1	Descriptions of models indicating the graph representation type and network architecture.	163
8.2	Total Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and <code>perf</code> ranking ( <code>tau</code> ) of the models. For RMSE and MAE, the lower the better. For <code>tau</code> , the higher the better. The percentage of difference is measured with respect to GNN-DSE (Chapter 6). Table 8.1 contains the definitions of the models.	163

8.3	The performance of the best design found by each DSE with respect to the best one found by AutoDSE in 25h. . . . .	167
8.4	The transfer learning results when a larger dataset is present. The performance comparison shows the best design found by each DSE with respect to the best one found by AutoDSE in 25h. . . . .	169
8.5	Impact of the classification model. The baseline is AutoDSE after running for 25h.	170
8.6	Impact of alternative hierarchy structure and larger model. In the new hierarchy graph, we define new pseudo nodes for ‘for’ loops. The baseline is AutoDSE after running for 25h. . . . .	171

## ACKNOWLEDGMENTS

Completing a Ph.D. has been one of the most challenging and transformative journeys of my life. Unlike anything else I have faced, the future often felt uncertain and intimidating. This journey has been an emotional rollercoaster, with many highs and lows. Above all, I want to sincerely thank my advisor, Professor Jason Cong. As a new Ph.D. student with no background in high-level synthesis, I greatly relied on his consistent guidance and unwavering support, particularly during moments of self-doubt. Throughout the research process, Professor Cong taught me to identify the problems that matter and address them systematically. I really appreciate how he trusted me to pursue topics I was passionate about, even when progress was slow. Whenever I felt frustrated, his wise advice and experience kept me going. His constant support gave me the confidence to grow as a researcher. Professor Cong did not just treat me as a student; he treated us all like family. He was always there to help and guide us toward our full potential. Being part of his lab was a real privilege, and I could not have asked for a better advisor. I will always cherish the valuable lessons and guidance he provided.

I would like to extend my gratitude to my committee members, Professor Yizhou Sun, Professor Tony Nowatzki, and Professor Sitao Huang, for their dedication, patience, and valuable suggestions to enhance the quality of my dissertation. I wish to thank Professor Sun for sharing her extensive knowledge of the world of deep learning and guiding me through it. Her invaluable insights and visionary approach to developing models capable of learning EDA tasks significantly influenced my research direction. I am thankful to Professor Nowatzki for the invaluable lessons I gained from his course on machines that learn hardware tasks and from our collaboration on the OverGen project. Additionally, I thank Professor Huang for spending his time discussing the opportunities in developing a design space explorer for high-level synthesis and raising the abstraction level.

Many thanks to Alexandra Luong for her fantastic work in handling the paperwork related to my graduate studies throughout my Ph.D. years. I enjoyed our frequent chats and will miss them. I would like to thank Marci Baun for editing my papers. I am grateful to all

my collaborators, from whom I learned so much. I want to especially thank Jie Wang for patiently answering my numerous questions about high-level synthesis and microarchitecture design, Cody Hao Yu for sharing his expertise in design automation and his vision for its future, and Yusheng Bai for sharing his experience in training graph neural networks. A special thanks to Peipei Zhou for providing tips on how to succeed in my Ph.D. and for being a constant source of motivation. I had three wonderful internships at Samsung Semiconductor Inc. and NVIDIA that greatly contributed to my development as a researcher. I am thankful to all the amazing people who helped me there, including Xuebin Yao, Caroline Kahn, Bryan Catanzaro, Jonathan Raiman, Robert Kirby, and Shrimai Prabhunoye.

I would also like to extend my appreciation to Vidushi Dadu for making my time in the lab enjoyable, along with my great labmates at VAST Lab: Licheng Guo, Jason Lau, Yuze Chi, Zhe Chen, Karl Marrett, Weikang Qiao, Young-kyu Choi, Linghao Song, Wan-Hsuan Lin, Stephane Pouget, Zijian (Brady) Ding, Neha Prakriya, Daniel Tan, Chengdi Cao, Jason Kimko, Suhail Basalama, Zifan He, and Jake Ke.

I also want to thank myself for never giving up.

As a final remark, I dedicate this dissertation to my parents, brothers, and husband. Their love, support, and encouragement were essential for me to reach this point. I am eternally grateful to them, and words cannot express my gratitude enough.

The research studies in this dissertation are partially supported by the ICN-WEN award jointly funded by NSF (CNS-1719403) and Intel (34627365), the NeuroNex Award funded by NSF (DBI-1707408), the CAPA award jointly funded by NSF (CCF1723773) and Intel (36888881), the RTML award funded by NSF (CCF-1937599), the NSF III-1705169 award, NSF 2211557, NSF 2119643, NSF 2303037, Samsung Semiconductor Inc., CDSC industrial partners, Okawa Foundation grant, Amazon research awards, CISCO research grant, Picsart gifts, Snapchat gifts, NASA, and SRC JUMP 2.0 Center.

## VITA

- 2013-2018 B.S., Electrical Engineering,  
Sharif University of Technology, Iran.
- 2018-2021 Master, Computer Science,  
University of California, Los Angeles, USA.

## PUBLICATIONS

Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong. “Robust GNN-based Representation Learning for HLS”. Proceedings of the 42nd IEEE/ACM International Conference on Computer-Aided Design (*ICCAD*), 2023. (**Best Paper Award candidate**)

Yunsheng Bai, Atefeh Sohrabizadeh, Zongyue Qin, Ziniu Hu, Yizhou Sun, and Jason Cong. “Towards a Comprehensive Benchmark for High-Level Synthesis Targeted to FPGAs”. Advances in Neural Information Processing Systems (*NeurIPS*), 2023.

Suhail Basalama, Atefeh Sohrabizadeh, Jie Wang, Licheng Guo, and Jason Cong. “FlexCNN: An End-to-End Framework for Composing CNN Accelerators on FPGA”. ACM Transactions on Reconfigurable Technology and Systems (*TRETS*), 2023.

Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Lucheng Zhang, Rishabh Mani, Lucheng Zhang, Jason Cong, and Tony Nowatzki, “OverGen: Improving FPGA Usability through Domain-specific Overlay Generation”. Proceedings of 55th IEEE/ACM International Symposium on Microarchitecture (*MICRO*), 2022. (**Best Paper Runner-up Award**)

Yuze Chi, Weikang Qiao, Atefeh Sohrabizadeh, Jie Wang, and Jason Cong, “Democratizing Domain-specific Computing”. Communications of the ACM (*CACM*), 2022.



Atefeh Sohrabizadeh, Yunsheng Bai, Yizhou Sun, and Jason Cong, “Automated Accelerator Optimization Aided by Graph Neural Networks”. Proceedings of the 59th ACM/IEEE Annual Design Automation Conference (*DAC*), 2022.

Yunsheng Bai, Atefeh Sohrabizadeh, Yizhou Sun, and Jason Cong, “Improving GNN-based Accelerator Design Automation with Meta Learning”. Proceedings of the 59th ACM/IEEE Annual Design Automation Conference (*DAC*), 2022. **(Invited)**

Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong, “AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators”. ACM Transactions on Design Automation of Electronic Systems (*TODAES*), 2022. **(Best Paper Award)**

Atefeh Sohrabizadeh, Yuze Chi, and Jason Cong, “StreamGCN: Accelerating Graph Convolutional Networks with Streaming Processing”. Proceedings of 2022 IEEE Custom Integrated Circuits Conference (*CICC*), 2022. **(Invited)**

Linghao Song, Yuze Chi, Atefeh Sohrabizadeh, Young-kyu Choi, Jason Lau, and Jason Cong, “Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication”. Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (*FPGA*), 2022.

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.

# CHAPTER 1

## Introduction

According to Statista, in 2024, spending on data center systems is expected to amount to 235 billion U.S. dollars [Dep24a] with the projected global data to be 147 zettabytes ( $10^{21}$  bytes) [Dep24b]. Due to this rapid growth of datasets in recent years, the demand for scalable high-performance computing continues to increase. However, the breakdown of Dennard’s scaling [Den+74] has put an end to frequency scaling and made energy efficiency an important concern in data centers. This has spawned exploration into using accelerators such as Field-Programmable Gate Arrays (FPGAs) to alleviate power consumption while achieving high performance with customized computing units and memory access. For example, one of the biggest companies manufacturing general-purpose processors, AMD, has acquired FPGA companies, Xilinx (for 50 billion U.S. dollars in 2021) [AMD22; Reu22]. Microsoft has adopted CPU-FPGA systems in its data center to help accelerate the Bing search engine [Put+14]. Amazon introduced the F1 instance [Ama24], a compute instance equipped with FPGA boards, in its commercial Elastic Compute Cloud (EC2). Samsung offers SmartSSD which brings FPGA-based acceleration near storage [Sma24b]. Similarly, Cisco has developed SmartNIC to offer an FPGA-based solution for low-latency networking [Sma24a].

Although the interest in customized computing using FPGAs is growing, they are more difficult to program compared to CPUs and GPUs. This is because the traditional Register-Transfer Level (RTL) programming model is more like circuit design rather than software implementation. To improve the programmability, High-Level Synthesis (HLS) [Con+11; Zha+08] has attracted a large amount of attention over the past decades. Currently, most FPGA vendors have their commercial HLS products e.g., AMD Xilinx Vitis [AMD24a] and

Intel High-Level Synthesis Compiler [Int24a]. With the help of HLS, one can program the FPGA more easily by controlling how the design should be synthesized from a high-level view. The main enabler of this feature is the ability to iteratively reoptimize the microarchitecture quickly just by inserting synthesis directives in the form of pragmas instead of rewriting the low-level (circuit-level) description of the design. Because of the reduced code development cycle and the shorter turnaround times, HLS has been rapidly adopted by both academia and industry [Dua+18; Lai+19; ZPM18; And+17; WGC21].

Even though HLS is suitable for hardware experts to quickly implement an optimal design, it is not friendly for most of the general software designers who have limited FPGA domain knowledge. Since the hardware architecture inferred from a syntactic C implementation could be ambiguous, current commercial HLS tools usually generate architecture structures according to specific HLS C/C++ code patterns. As a result, even though it was shown in [Con+11] that the HLS tool is capable of generating FPGA designs with a performance as competitive as the one in RTL, not every C program gives a good performance. The designers rather must manually reconstruct the HLS C/C++ kernel with specific code patterns and hardware-specific pragmas to optimize the microarchitecture and achieve high performance [Con+18b; Con+18a].

Mastering FPGA optimization techniques is challenging and time-consuming for general software programmers, making them more likely to prefer other popular accelerators such as power-hungry GPUs over FPGAs. This creates significant barriers to the adoption of FPGAs in data centers, limits the expansion of the FPGA user community, and hinders the advancement of FPGA technology. This issue has inspired us to work on making FPGAs more accessible by automating their compilation and optimization processes, enabling programmers with reasonable expertise to develop customized accelerators with minimal effort. Additionally, an automated compilation tool can help shorten design turnaround times

In parallel, a lot of the data stored in data centers are in the form of images or graphs. To process these data, researchers have proposed to apply deep neural networks to automatically extract their features. Convolutional Neural Network (CNN) is one of the most

established algorithms in this domain with applications in computer vision such as image classification (e.g., [KSH12; He+16]), object detection (e.g. [Cao+17; WSH20]), etc. Because of their wide applications, there is a growing interest in accelerating CNN computation using FPGAs due to their high energy efficiency and performance (e.g., [Aki+18; BZH18; Gua+17; Li+16; NSW18; Sud+16; Wei+17; Zha+15; Zha+18a; Zha+18b; SFM17b]). However, the irregularity and high variation of the CNN models have made this task challenging and there is a need for an accelerator that can adapt to these variations. Inspired by the success of CNNs, Graph Convolutional Networks (GCNs) [KW16] were developed to extract structured low-dimensional features from the graphs which are mainly unstructured and have high dimensionality. Unfortunately, the differences between an image and a graph structure make the countless CNN accelerators proposed in the literature incompatible with GCNs. Hence, they require a separate accelerator.

## 1.1 Dissertation Overview

The growth of Machine Learning (ML) and the development of HLS has made us believe they can benefit from each other by combining customized computing with ML. In this dissertation, we focus on making FPGAs more accessible by improving their programmability. As such, we divide our efforts into two parts as demonstrated in Fig. 1.1. First, we implement customized accelerators for common deep learning networks: CNN and GCN. Then, we exploit ML techniques to develop frameworks that can *learn* to design accelerators and reach the expert-level quality of design for general applications.

### 1.1.1 Customized Computing for Machine Learning Acceleration

The irregularity of recent CNN models such as lower data reuse and parallelism due to the extensive network pruning and simplification creates new challenges for FPGA acceleration. More specifically, different CNN models make use of different types of convolutional layers that have disparate computation patterns. Furthermore, the same type of convolutional

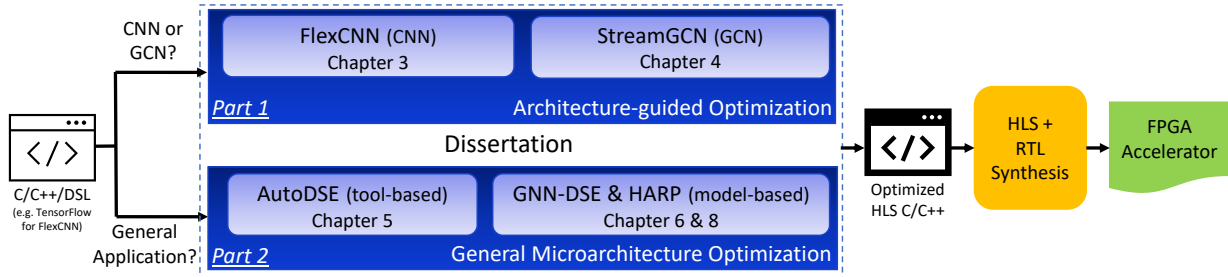


Figure 1.1: The overview of the dissertation.

layers in a network have diverse characteristics in terms of their input, output, and kernel (filter) size. Thus, the Computation to Communication (CTC) ratio varies across different layers which affects the attainable performance. To alleviate this problem, we propose a flexible and composable architecture called FlexCNN which adopts techniques including dynamic tiling and data layout to deliver high computation efficiency for different types of convolutional layers. FlexCNN has the architecture template to support the common CNN layers such as normal convolution, depthwise separable convolution, ReLU, pooling, bilinear upsampling, and any combination of these layers (for example, residual bottleneck block [San+18]). The FlexCNN compilation system starts with parsing TensorFlow’s protobuf file which contains the computation graph of the target CNN. After extracting the required information, FlexCNN performs a Design Space Exploration (DSE), which uses two analytical models that estimate the latency and resource consumption of each module, to find the best hardware configurations for the architecture based on the target network. Once the optimal hardware configuration is found, the network is run through an instruction generator to produce a (Very Long Instruction Word) VLIW-like instruction for processing each of the layers. Each instruction consists of the information to run that layer which includes the tiling factors associated with that layer, the DRAM locations for reading and writing the data, and the enable signals for each of the modules.

Without proper optimization, there could be significant overheads when integrating FPGAs into existing machine learning frameworks like TensorFlow which can lead to up to  $8.45\times$  performance degradation according to our experiments. Such a problem is mostly

overlooked by previous studies. As such, in FlexCNN, we employ a two-level pipelining to let the FPGA computation be executed simultaneously with the rest of the overheads. The first level of pipelining is for overlapping with the overheads that TensorFlow introduces and the second one is to overlap FPGA computation with the DMA data movement steps. Experimental results show that the FlexCNN architecture optimizations bring  $2.3\times$  performance improvement when tested on the OpenPose [Cao+17] network. The pipelined integration stack leads to a further  $5\times$  speedup. Hence, the SW/HW co-optimization produces a speedup of  $11.5\times$  proving the efficiency of FlexCNN. Details of the optimizations can be found in Chapter 3.

GCNs follow the same behavior as CNNs in learning. They consist of multiple layers in which the features of the nodes are propagated within them until rich information of the input graph is derived. In each layer, the GCN updates the node features by gathering the neighbors' features and passing the result through a filter. Despite this similarity to CNNs, they have different computation complexity and memory access patterns. This is because, compared to an image, the neighbors of a node in a graph may be stored in any location in memory. This will result in many irregular memory accesses to all levels of the memory hierarchy. Furthermore, as their computation can be modeled as matrix multiplication, they have much lower data reuse compared to CNNs. On the other hand, compared to the traditional graph algorithms such as Breadth First Search (BFS), Single-Source Shortest Path (SSSP), PageRank (PR), etc., each node deals with a long vector rather than a scalar. As a result, we can exploit intra-node parallelism. Furthermore, all the vectors associated with different nodes share a weight matrix which brings in more data reuse opportunities. These differences have created a new line of research to exploit the new acceleration opportunities (e.g., [Yan+20a; Gen+20; ZP20; Lia+20; ZKP21]).

The previous works in this domain mainly target large graphs. While some graph data tend to scale rapidly, many graph data are naturally limited in size, for example, chemical compounds and molecules [NCI04; SI15; Bol+08; Che+19] which have a wide application in different domains including drug development, quantum mechanics, physical chemistry,

biophysics, etc [Wu+18; Che+19]. It turns out that the size of the graphs impacts the scheduling technique we employ and brings in streaming opportunities throughout the whole network. Furthermore, the node embeddings in a GCN create a sparse matrix that requires in-situ sparsity support, but it is overlooked in the previous works. To fill in this void, we present StreamGCN (Chapter 4) as an efficient and flexible GCN accelerator for streaming small graphs - from the global memory and through the network - while exploiting all the available sparsity. The experimental results demonstrate that our optimizations result in  $2.3\times$  speedup while requiring  $1.7\times$  less computation resources (DSP).

### 1.1.2 Machine learning for Designing Customized Accelerators

Although HLS can potentially help shorten the code development cycle by raising the abstraction level, there are many design choices (e.g., specified by the `pipeline` and `unroll` pragmas) at a higher level that require more time to explore. This is because the solution space grows exponentially by the number of candidate pragmas. To make matters worse, it takes several minutes to hours to get feedback from the HLS tool on the Quality of Results (QoR). As a result, it may take several weeks to optimize the microarchitecture for each new application. As our first effort to improve the programmability of FPGAs for general software programmers, we focus on automatically augmenting the behavioral description of the program in C/C++ with the best HLS pragma configurations. To do so, we develop and implement AutoDSE as an automated framework to explore the different pragma configurations to systematically close in on high-quality design points.

Since the HLS tools are based on multiple heuristics which makes it hard to capture its behavior with an analytical model [SW19], we treat the HLS tool as a black box and invoke the tool each time we want to evaluate a design point. As mentioned above, not only are we dealing with a huge design space, but it is also very time-consuming to assess each design point since we rely on the HLS tool for it. As a result, we cannot afford to spend many iterations to improve the design’s quality and must explore the solution space intelligently. For that, AutoDSE adapts a bottleneck-guided coordinate optimizer

to systematically search for better configurations by mimicking an expert’s approach in design optimization. This means that at each iteration, AutoDSE gets to try the high-impact parameters (pragmas) by recognizing and analyzing the design’s bottleneck. We have extensively tested AutoDSE on 45 different kernels. The experimental results show that the best design found by AutoDSE can achieve, on the geometric mean,  $19.9\times$  speedup over one CPU core for MachSuite [Rea+14] and Rodinia [Che+09] benchmarks. Furthermore, it can achieve the same or better performance compared to AMD Xilinx Vitis library [Xilb] while reducing the required optimization pragmas by  $26.38\times$ , on the geometric mean. Chapter 5 explains the details of AutoDSE in more detail.

While AutoDSE has been shown to outperform its previous state-of-the-art approaches, directly using the HLS tool can significantly slow down the optimization process. As a result, we are forced to explore only a small portion of the solution space. The most effective solution to this problem is to develop a model as a surrogate of the HLS tool to be used instead of invoking the tool. Therefore, we can evaluate each design candidate in milliseconds as opposed to several minutes to hours. However, it is not a trivial task due to the different heuristics employed by the HLS tools. When developing a model, the *first challenge* we need to address is deciding how we should represent each design (C/C++ program). For this matter, we designed a heterogeneous graph representation of the program which includes both program semantics (control, data, and call flows) and pragma flows. Then, we can exploit a Graph Neural Network (GNN) to extract the required features of the graph for predicting the design’s objectives. Current HLS tools optimize the design based on specific code patterns. Although different applications have different domains, they may share the same code structures for some parts. Thus, this approach can help with identifying the different code patterns and learning their effect to be able to transfer the knowledge from one application to another. As a result, we can develop a single model and utilize it across many applications. Our results confirm that our proposed representation and model can indeed achieve this goal with high accuracy and a resulting throughput of 22 inferences/second. More details of our approach are explained in Chapter 6. As ML-based techniques require



a large dataset, we compiled the HLSyn database in Chapter 7, consisting of 41 distinct kernels with over 81,000 labeled designs.

While GNN-DSE exhibits promise, several challenges must be addressed to enhance the effectiveness of HLS performance prediction. The initial challenge arises from lengthy dependency chains in the program, where an element may depend on another one located far away. In response, we introduce HARP, utilizing a hierarchical graph representation of HLS designs to establish relationships at different levels. We augment our previous graph representation with auxiliary nodes that provide high-level hierarchical information about the design, resulting in a coarsened view of the design that facilitates coping with long-range dependencies. This representation significantly reduces the average shortest path of our benchmark by a factor of 5, enabling the GNN model to efficiently pass nodes' messages throughout the entire graph. Moreover, recognizing that design objectives are influenced by both program context and pragmas, it is advantageous to develop a model capable of learning the effect of each component separately. To address this, HARP introduces a Neural Pragma Transformer (NPT), modeling pragmas as learnable functions applied to the program representation. This architectural design aligns more naturally with the transformative nature of pragmas.

These optimizations to the representation and model architecture are not only crucial for improving predictive model accuracy but also contribute to building more robust GNN representations adaptable to environmental shifts. A significant source of this shift occurs due to the updates to the HLS tool and subsequent changes in the heuristics utilized, thereby influencing the objectives of the design. Considering the substantial cost associated with regenerating the entire database (requiring the execution of the HLS tool) and retraining the model, a preferable approach is to transfer the model to the new shift using a smaller dataset. Experimental results illustrate that HARP yields a further 34% improvement in prediction accuracy in the new environment. Consequently, it identifies designs with  $1.31\times$  lower cycle counts compared to model-based DSE, on average. When compared to model-free DSE, HARP identifies designs with  $2.13\times$  lower cycle counts on average for both the

initial environment and after transferring the learned model to the new environments. For a more in-depth exploration of HARP, detailed insights are provided in Chapter 8.

The remainder of this dissertation is structured as follows: Chapter 2 provides an overview of HLS and the Merlin Compiler, along with a description of the computation of CNNs and GNNs. This chapter also delves into related literature on accelerating CNNs and GCNs, as well as performance models and DSE techniques for Electronic Design Automation (EDA) and HLS. The first part of the dissertation starts with the introduction of FlexCNN, our CNN accelerator in Chapter 3, followed by the presentation of StreamGCN, our GCN accelerator in Chapter 4. The second part of the dissertation starts with the proposal of AutoDSE in Chapter 5 as a push-button bottleneck-based optimizer for HLS programming. We introduce GNN-DSE in Chapter 6 to demonstrate how one can develop a surrogate of the HLS tool. We then explain the details of our HLSyn database in Chapter 7. In Chapter 8, we discuss what optimizations are needed to make the model predictions more robust. The dissertation concludes with Chapter 9, presenting the final remarks and summary of the research findings.

# CHAPTER 2

## Background

### 2.1 High-Level Synthesis (HLS)

With the growing popularity of FPGAs, their design tools are improving as well. Nowadays, FPGA vendors provide fully automated design tools that translate a high-level behavioral description of the program all the way down to a bitstream that can be executed on the FPGA as shown in Fig. 2.1. The first step, done by the HLS tool, is to generate the equivalent of the input program in the RTL format. Then, the RTL code is passed through *logic synthesis* which performs architecture-independent optimizations, technology mapping, and architecture-dependent optimizations to generate a gate-level representation of the design. Finally, the resulting netlist would go through placement and routing (*physical design* step) to generate the bitstream. As pointed out by the previous works (e.g., [Con+11]), the quality of the final design is highly impacted by the input HLS code and not every HLS code gives a good performance. In this dissertation, our goal is to generate better (HLS) C/C++ codes to produce a design with a higher quality. In the first part, we do it for two specific applications, CNN and GCN; in the second part, our focus is to automatically generate better input codes for general applications.

The HLS tools make use of two types of pragmas. Table 2.1 lists the common pragmas that can be used with the AMD Xilinx HLS tool. The first type is the non-optimization pragmas (e.g., how the I/O ports will be interfaced with the global memory) which are relatively easy to learn and apply. The second type includes the optimization pragmas, such as *pipeline* and *unroll* pragmas that define the microarchitecture of the resulting accelerator. As a result, they can influence the achievable performance. In fact, the same programs

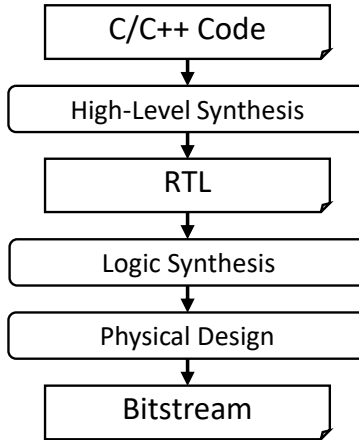


Figure 2.1: High-level synthesis design flow.

with different pragma combinations can have several orders of magnitude differences in their final performance. Unfortunately, these pragmas are usually much more challenging for an average software programmer with no knowledge of hardware to learn and master. Therefore, to expand the FPGA’s community, it is important to relieve the users from the burden of this optimization step by eliminating the need to apply optimization pragmas manually and instead automating their exploration process.

Table 2.1: Common AMD Xilinx HLS Pragmas.

Non-optimization Pragmas	Optimization Pragmas
INTERFACE	STREAM
ALIAS	DATAFLOW
LATENCY	LOOP_MERGE
LOOP_TRIPCOUNT	LOOP_FLATTEN
TOP	UNROLL
RESET	DEPENDENCE
	PIPELINE
	ARRAY_PARTITION
	ARRAY_RESHAPE
	INLINE

### 2.1.1 The Merlin Compiler

The Merlin Compiler<sup>1</sup> [Con+16b; Con+16a; Fal] was developed to raise the abstraction level in FPGA programming by introducing a reduced set of high-level optimization directives and automatically generating the respective HLS code along with the required HLS pragmas to enable the designated optimizations. It uses a simple programming model similar to OpenMP [DM98], which is commonly used for multi-core CPU programming. Like in OpenMP, it defines a small set of compiler directives in the form of pragmas for optimizing the design.

Table 2.2 lists the Merlin pragmas with architecture structures. Based on these user-specified pragmas, the Merlin Compiler performs source-to-source code transformation and automatically generates the related HLS pragmas to apply the corresponding architecture optimizations. For example, array partitioning is required to increase the number of memory access concurrency when parallel accesses to a buffer are needed. Based on the parallel factor and the buffer's access type, the Merlin Compiler decides the best way for partitioning the buffers; thus, it does not require the `ARRAY_PARTITION` pragma. The `fg` option in the fine-grained pipeline mode refers to the code transformation that tries to apply fine-grained pipelining to a loop nest by fully unrolling all its sub-loops; whereas, the `cg` option in the coarse-grained pipelining transforms the code to enable double buffering. The `parallel` and `tiling` pragmas allow us to adjust the duplication factor of the processing elements (in the case of `cg` parallelization) or the arithmetic operations (in the case of `fg` parallelization) as well as the amount of cached data, respectively. Even though the Merlin Compiler only takes three pragmas, it generates several HLS pragmas based on them, including `PIPELINE`, `UNROLL`, `ARRAY_PARTITION`, `INLINE`, `DEPENDENCE`, and `LOOP_FLATTEN`. Furthermore, based on these pragmas, it automatically employs code transformations to implement memory coalescing, apply memory burst, and cache the required data for enabling architectural optimizations.

---

<sup>1</sup>Open-sourced at <https://github.com/Xilinx/merlin-compiler>.

Table 2.2: The Merlin pragmas with architecture structure.

Keyword	Available Options	Architecture Structure
parallel	factor=<int>	CG & FG parallelism
pipeline	mode=<cg fg no>	CG or FG or no pipeline
tiling	factor=<int>	Loop Tiling

CG: Coarse-grained; FG: Fine-grained

Code 2.1: Input code snippet in Merlin C.

---

```

1 void AddMatrix(const float a[N][M], const float b[N][M], float c[N][M]) {
2 #pragma ACCEL pipeline
3     for (int i = 0; i < N / 64; ++i){
4         for (int ii = 0; ii < 64; ++ii){
5             for (int j = 0; j < M; ++j){
6                 c[i * 64 + ii][j] = a[i * 64 + ii][j] + b[i * 64 + ii][j];
7             }}
8 }

```

---

To illustrate the key advantages of using the Merlin Compiler, we analyze the transformations applied to Code 2.1. The transformed version is shown in Code 2.2. The generated code has more than 100 lines and utilizes 18 optimization pragmas alongside 7 interface pragmas (`#pragma HLS INTERFACE`). However, for brevity, we only display the most important sections of the code. The Merlin Compiler optimizes the code by employing memory coalescing to fully utilize the available 512 bits of DRAM bandwidth. Additionally, it loads/stores bursts of data and creates on-chip buffers to cache them. Furthermore, to enable coarse-grained pipelining as we requested in Line 2 of Code 2.1, the code is divided into three stages: load, compute, and store, allowing for overlapping of computation and communication. As we will discuss in Chapter 5, the Merlin Compiler is highly effective at applying code transformations involving repeated architectural optimizations that benefit a wide range of applications. Consequently, we can optimize a forward path of a CNN code with only 4 Merlin pragmas, as shown in Code 5.2, and achieve a speedup of over 7000 $\times$ . By contrast, attempting the same improvements directly in Vitis HLS C would require extensive code

Code 2.2: Generated HLS code for Code 2.1 by the Merlin Compiler.

---

```
1 // Skip headers for brevity
2
3 void load(merlin_uint_512 *a, ...) {
4 #pragma HLS INLINE OFF
5 memcpy_wide_bus_read_float_2d_4096_512(
6   a_buf, 0, 0, (merlin_uint_512 *)a,
7   (i*64*M*4), sizeof(float) * ((unsigned long )(64*M)), 64*M);
8 ...
9 }
10
11 void compute(float* a_buf, float* b_buf, float* c_buf){
12 #pragma HLS INLINE OFF
13   for (int ii = 0; ii < 64; ++ii) {
14     for (int j = 0; j < M; ++j) {
15 #pragma HLS dependence variable=c_buf array inter false
16 #pragma HLS pipeline
17       c_buf[ii][j] = a_buf[ii][j] + b_buf[ii][j];
18     }}
19 }
20
21 void store(float *c_buf, ...) {
22 #pragma HLS INLINE OFF
23 memcpy_wide_bus_write_float_2d_4096_512(
24   (merlin_uint_512 *)c, c_buf, 0, 0,
25   (i*64*M*4), sizeof(float) * ((unsigned long )(64*M)), 64*M);
26 }
27
28 void AddMatrix(merlin_uint_512 *a, merlin_uint_512 *b, merlin_uint_512 *c){
29 // Skip interface pragmas
30 // Skip local buffer generations and array_partition pragmas
31 for (int i = 0; i < N / 64 + 2; ++i) {
32   if (i % 3 == 0) {
33     load(a, a_buf_0, ...);
34     compute(a_buf_1, ...);
35     store(c_buf_2, ...);}
36   else if (i % 3 == 1)
37     ...
38   // Skip the rest for brevity
39 }
40 }
```

---

modifications, resulting in over 150 lines of code and the use of 28 optimization pragmas (refer to Appendix A.1).

Therefore, to reduce the size of the solution space and leverage these code transformations, we chose to utilize the Merlin Compiler as the backend of our tools in the second part of the dissertation. This means that the solution space consists of three types of pragmas: `pipeline`, `parallel`, and `tiling`. The Merlin Compiler requires fewer pragmas, as it conducts source-level code reconstruction and generates the necessary HLS pragmas. This characteristic results in a more concise design space, making it more suitable for developing a DSE, as demonstrated in references [Con+18a; Yu+18].

## 2.2 Convolutional Neural Networks

A *Convolutional Neural Network* (CNN) is a widely used deep learning model with applications in computer vision [Fan+20], speech processing [PMC19], and more. The success of CNNs comes from their ability to automatically identify important input features without human intervention [Gu+18]. A CNN includes multiple layers such as convolutional layers, pooling layers, and activation layers like ReLU [NH10], with the convolutional layers being the primary components for extracting the features of the input. The convolutional layers take a 3-dimensional input of size  $N \times H \times W$  as the input with  $N$ ,  $H$ , and  $W$  representing the number of feature maps, height, and width of the input. They combine and filter the  $N$  input feature maps to generate  $M$  output feature maps. A standard convolutional layer employs weight filters of size  $M \times N \times K \times K$  as shown in Fig. 2.2(a). Each input feature map is filtered by a separate weight filter of size  $K \times K$ . Their result would be added together to produce one output feature map and this process is repeated until  $M$  output feature maps are generated. Therefore, the total computation cost for this layer is  $M \times N \times H \times W \times K \times K$ .



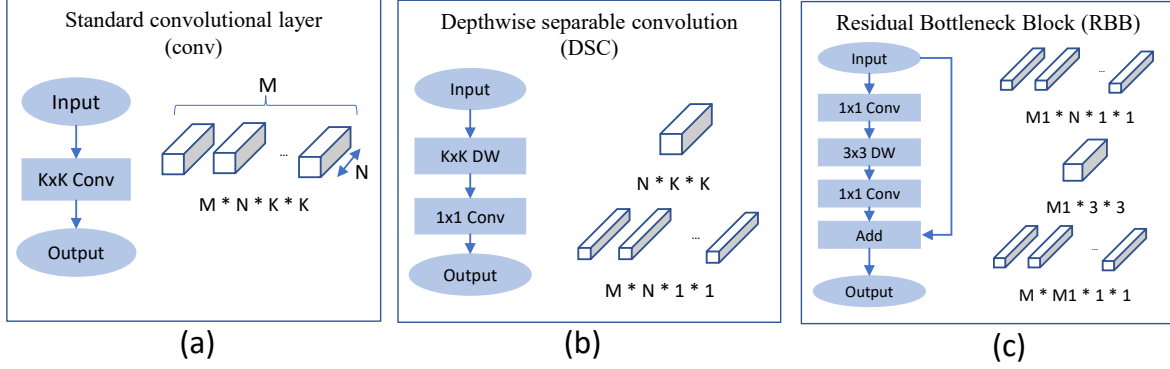


Figure 2.2: Structure of our targeted convolutional layers along with their respective weights.

## 2.2.1 Other Convolutional Layers

### 2.2.1.1 Depthwise Separable Convolution

Sifre et al. [SM14] introduced *Depth-wise Separable Convolutions* (DSC) to decrease the computational workload of a convolutional layer. In a standard convolutional layer (conv), the feature maps are filtered and combined in one step. The DSC, however, splits this step into two phases as shown in Fig. 2.2(b). The first phase, depthwise convolution (DW), does lightweight filtering, and the second phase, pointwise convolution (PW), merges the resulting filtered feature maps through  $1 \times 1$  weight filters (linear combinations of the filtered feature maps). As a result, a DSC uses  $N \times K \times K$  filters (weights) for DW and  $M \times N \times 1 \times 1$  filters for PW. By applying this change, the amount of computation is reduced by a factor of  $\frac{1}{M} + \frac{1}{K^2}$  [How+17]:

$$\frac{(N \times H \times W \times K \times K) + (M \times N \times H \times W)}{M \times N \times H \times W \times K \times K} = \frac{1}{M} + \frac{1}{K^2} \quad (2.1)$$

### 2.2.1.2 Residual Bottleneck Block

*Residual Bottleneck Blocks* (RBB) [San+18; He+16] can also be used to strike a balance between the computational efficiency and expressive power of the CNN. Fig. 2.2(c) depicts the architecture of this block. It consists of a  $1 \times 1$  conv followed by a  $3 \times 3$  DW and then another  $1 \times 1$  conv, each of which is followed by ReLU and a batch normalization

layer. The  $1 \times 1$  convolutions are used for dimension reduction or restoration. The nature of this block allows us to adjust the number of feature maps by either increasing (as in MobileNetV2 [San+18]) or decreasing (as in ResNet [He+16]) the number of intermediate channels.

## 2.3 Graph Neural Networks

In recent years, there has been a growing volume of graph data, prompting researchers to develop learning models specifically tailored for the efficient processing of such data. Among various graph machine learning methods, *Graph Neural Networks* (GNN) [Wu+20] are gaining popularity due to their effectiveness across diverse applications, e.g., social network analysis [TLH19], biomedical tasks [Yue+20], traffic forecasting [JL21], etc.

The core idea of a GNN model is to extract the graph information by learning the features (embeddings) of each node in the graph by aggregating information from its neighboring nodes, commonly referred to as “message passing”. A GNN model, like a CNN, passes the node embeddings through a series of layers until it derives a rich encoding of the graph. The computation of one layer of a typical GNN, in its general form, can be formulated as follows:

$$\vec{h}'_i = \sigma \left( \text{FT} \left( \text{AGG} \left( \{ \vec{h}_j \mid j \in \mathcal{N}(i) \} \right) \right) \right) \quad (2.2)$$

where  $\vec{h}_i \in \mathbb{R}^F$  and  $\vec{h}'_i \in \mathbb{R}^{F'}$  denote the initial and updated embeddings of node  $i$  in spaces with dimensions  $F$  and  $F'$ , respectively. The AGG function is the aggregation function, responsible for collecting and combining the embeddings of neighboring nodes ( $\mathcal{N}(i)$ ). The function FT performs feature transformation on the aggregated results for each node, while the activation function  $\sigma$  introduces non-linearity into the model.

### 2.3.1 Common Graph Convolutional Layers

#### 2.3.1.1 Graph Convolutional Network

*Graph Convolutional Network* (GCN) [KW16] is a popular form of a GNN that adopts a simple aggregation function to perform a weighted summation of the embeddings of neighboring nodes using the degree of the node,  $d_i$ , as shown in Fig. 2.3:

$$\vec{h}'_i = \sigma \left( \mathbf{W} \sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{d_j d_i}} \vec{h}_j \right) \quad (2.3)$$

where  $\mathbf{W}$  is a trainable weight matrix for the FT step to act as a filter. Fig. 2.3 illustrates this operation for a node. i.e., Node 1. It applies a weighted summation on the embeddings of its neighbors (and itself) based on the edge weights, which are determined by the degree of both the source and destination nodes. Then, the result is multiplied by the weight matrix ( $\mathbf{W}$ ) to calculate the output node embedding.

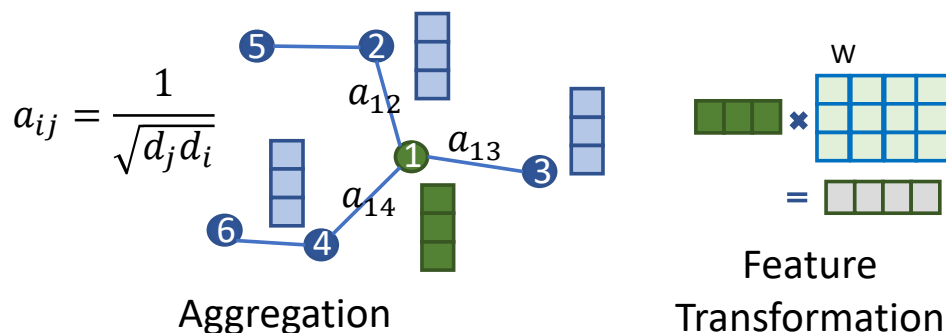


Figure 2.3: The computation of a GCN layer on a sample node.

#### 2.3.1.2 Graph Attention Network

One problem with the GCNs is that the aggregation of features of the nodes is based on a fixed set of weights determined by the degree of the nodes. Therefore, the model has no way of prioritizing any of the neighbors to learn better embeddings. To solve this problem, another class of GNN models, *Graph Attention Networks* (GAT) [Vel+17], were introduced

to learn the *importance* of the different neighbors of a node so that they can contribute in updating the node embeddings based on their *attention*. The computation of a GAT layer can be summarized as below:

$$\vec{h}'_i = \sigma \left( \sum_{j \in \mathcal{N}(i) \cup \{i\}} \alpha_{i,j} \mathbf{W} \vec{h}_j \right) \quad (2.4)$$

$\alpha_{i,j}$ s are the attention coefficients computed by multi-head dot-product attention. The computation for each head is as follows:

$$\begin{aligned} \text{LeakyReLU}(y) &= \max(\beta y, y), 0 < \beta < 1 \\ s_{i,j} &= \text{LeakyReLU}(\vec{a}^\top [\mathbf{W} \vec{h}_i \parallel \mathbf{W} \vec{h}_j]) \\ \alpha_{i,j} &= \frac{\exp(s_{i,j})}{\sum_{k \in \mathcal{N}(i) \cup \{i\}} \exp(s_{i,k})} \end{aligned} \quad (2.5)$$

where  $\parallel$  denotes the concatenation operation and  $\mathbf{a}$  is a learnable vector controlling the attention that node  $i$  receives from node  $j$ . Note that the FT step here is the same as in GCN and only the AGG step is changed. Fig. 2.4 shows this computation on a toy graph.

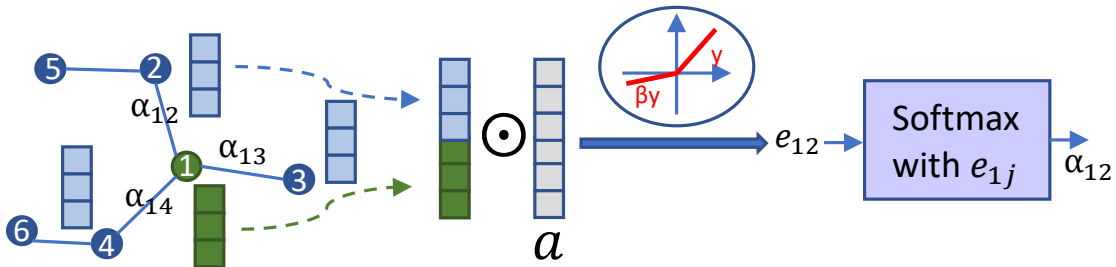


Figure 2.4: How the attention coefficient is calculated in a GAT layer.

## 2.4 Related Works

### 2.4.1 Convolutional Neural Network Accelerators

Due to the popularity of CNNs, countless previous works have focused on developing an accelerator for it. Zhang et al. [Zha+15] explored the design space of a CNN accelerator

and observed that the loop tiling factors can affect the Computation to Communication (CTC) ratio of the convolutional layers. As such, they proposed an analytical design scheme using the roofline model to pick the best tiling factors for a given CNN model based on its computing throughput and the required memory bandwidth. This work is later extended in [Zha+18a] to propose a uniformed accelerator to support both standard convolutional layers and fully connected networks. Eyeriss [CES16] proposed a novel dataflow architecture that enables the reuse of all types of data among PEs. This helped Eyeriss outperform the previously proposed dataflow schemes. Yang et al. [Yan+18] further explored the different dataflow techniques while changing the loop tiling scheme. They highlighted the importance of the tiling factors by showing that with proper loop tiling, many different dataflows can achieve similar and close-to-optimal energy efficiency.

In addition, Wei et al. [Wei+17] developed an automated compilation framework to generate systolic arrays for convolutional layers. However, it uses a uniform tiling factor for the whole design. This can lead to a sub-optimal solution as different layers have different characteristics in terms of the input, output, and filter size which affect the CTC ratio. That work is later on expanded in Wei et al. [Wei+18], which highlights the need for different accelerator configurations for different layers. The authors propose to place three different accelerators on-chip. Nonetheless, each of the accelerators still uses uniform tiling factors. DNNBuilder [Zha+18b] places one accelerator for each layer on-chip. While this method is useful for small networks, it limits the scalability of the design as the on-chip resources are limited. For this reason, it can only support CNNs that have less than 15 convolutional layers as stated in [WM19]. Furthermore, this work only enables one tiling factor, the width of the image, to be dynamic, which shrinks the design space with the possibility of losing the optimal design point. Wang et al. [CW18] developed PolySA as an automated framework that can generate systolic arrays with various topologies and different tiling factors using the Polyhedral transformations [Ben+10]. This can greatly help us in conducting a systematic approach to study the impact of different architecture choices such as systolic array topology, tiling factors, and buffer sizes. PolySA was later extended in AutoSA [WGC21] to increase

the generality of the work by including systolic array generation for both imperfect nested loops and multiple statements.

While the aforementioned works can generate high-performance accelerators for networks with standard convolutional layers, none of them support DSCs and RBBs. These layers pose new challenges such as poor data reuse and low degree of parallelism. The issue arises because, even though the inputs to these layers have the same size, their computational complexities vary. Consequently, this impacts the CTC ratio which requires further optimizations to efficiently support them without reconfiguring the hardware. Bai et al. [BZH18] implemented the MobileNetV2 [San+18], which includes all our target layers. However, the architecture only supports a limited convolution size ( $3 \times 3$ ) and achieves a low frequency of 133 MHz.

Moreover, the CNN computation is usually part of a larger pipeline which is the slowest part before acceleration. However, our experiments show that after acceleration and when the FPGA is integrated into a machine learning framework, its computation can take less than 12% of the end-to-end latency, thereby reducing the significance of the accelerator’s advantage. Unfortunately, this problem is overlooked by the previous works. While several works [Gua+17; NSW18] have focused on accelerator *generation* from TensorFlow-described networks, they did not address the challenges of *integrating* an accelerator into TensorFlow.

#### 2.4.2 Graph Convolutional Network Accelerators

In the last few years, there has been a growing interest in developing an accelerator for GCNs [ZP20; Yan+20a; Gen+20; ZKP21; Lia+20; Zha+23a]. HyGCN [Yan+20a] proposes a hybrid architecture with customized engines for each step of GCN: *Aggregation* and *Feature Transformation* (FT). To reduce the computation complexity, it samples the neighbors of a node and applies the Aggregation stage only on that subset. It further uses parallel SIMD cores to exploit feature-level parallelism along with edge-level (pipeline) parallelism for the Aggregation step. For each partition of the graph, it applies *window sliding* and *window shrinking* to reduce the zero elements of the adjacency matrix that need to be processed,

which in turn helps with decreasing the number of vertices that need to be loaded. For the FT step, it uses a systolic array to implement matrix (for the weight matrix) vector (for each node embedding) multiplication. It utilizes inter-layer pipelining for computations within a single layer, while the various layers of the GCN are processed sequentially and reuse the same engines.

GraphACT [ZP20] proposes an architecture to accelerate the training of GCN. In each training epoch, it samples a subgraph (small graph) from the full graph which can fit into FPGA’s on-chip memory. Like HyGCN, it adopts a systolic array for the FT step and processes the GCN layers sequentially on fixed hardware. For the Aggregation step, it exploits feature-level parallelization and proposes a *redundancy reduction* technique to decrease the number of operations by precomputing the repeated aggregations. Redundancy reduction is possible only when the adjacency matrix is *binary*. However, not all GCNs can benefit from this feature since they typically work with the *normalized* adjacency matrix meaning that they need *weighted* additions in this step. GraphACT is later extended in Zhang et al. [ZZP20] to support large graphs for accelerating the inference stage of GCN. It proposes a partitioning technique by reordering the nodes to increase the data locality in processing each of the partitions. BoostGCN [ZKP21] further extends Zhang et al. [ZZP20] by introducing a new partitioning method that considers both the feature dimension and node partitioning. It also enables efficient sparse computation during the FT step when sparsity exceeds 90% by converting the sparse matrix (node embedding) to the COO format. While sparsity support here is beneficial, transforming the matrix can be a potential overhead.

All the aforementioned works develop fixed hardware for all the layers of GCN and process them sequentially. This is an undesirable feature particularly when we target small graphs. This is because when dealing with small graphs, we are working on small matrices which makes their communication to and from DRAM a serious burden. Furthermore, our experiments show that the node embeddings for our target graphs have about 50% sparsity which makes the sparsity support in BoostGCN inapplicable. Nevertheless, we have seen that by pruning the zeros here, we can achieve considerable speedups. In fact, in Chapter 4,

we first develop a baseline architecture that has the same design principles as these works. Particularly, we reuse the same architecture for all the GCN layers, exploit only the sparsity of the Aggregation step, treat the FT step as regular matrix multiplication, and employ a 2D computation unit for it. Our experimental results show that not only should we execute the GCN layers in a pipelined fashion, but we should also exploit the sparsity of the node embeddings to enhance both area and performance. In fact, these optimizations bring in  $2.27\times$  speedup in the performance and an overall improvement of  $3.88\times$  in both performance and computation units' area.

AWB-GCN [Gen+20] proposes an architecture that supports inter-layer pipelining and considerations for the sparsity of the node embeddings for accelerating GCN. However, it partitions the computation by parallelizing the nodes which complicates the design of the task distributor since the node embeddings are sparse, and special consideration is needed to prevent PEs from doing unnecessary operations on the zero elements. On the other hand, feature-level parallelization is a more natural workload distribution here since once we get a non-zero embedding, we can fill the SIMD PE completely. As AWB-GCN is developed for large graphs, it adapts the inner-product matrix multiplication. For small graphs, this results in many data dependencies since there are not enough nodes to fill in the pipeline before the same node needs to be evaluated. Therefore, as we will explain further in Chapter 4, the outer-product matrix multiplication is preferred here.

### 2.4.3 Machine Learning for Electronic Design Automation

Since most problems in Electronic Design Automation (EDA) are classified as NP-complete, machine learning algorithms are gaining popularity in this domain due to their ability to efficiently solve them and produce high-quality solutions [Hua+21]. Additionally, these algorithms can aid in reducing manual effort and introducing greater automation into the design process. Machine learning and deep learning models have demonstrated remarkable success in various phases of the EDA flow, such as high-level synthesis [LC13; WXH22; Ust+20], logic synthesis [Net+19; YXD18], placement and routing in physical design [Ala+20; Xie+18;



Mir+21; LPL20; Kir+19; Kou+22], and design verification [Wan+18]. Huang et al. [Hua+21] identify four primary tasks in this field: (1) decision-making in conventional approaches, where an ML model substitutes for brute-force search or empirical configuration selection; (2) performance prediction, in which a model is employed to rapidly estimate Quality of Results (QoR); (3) black-box optimization, where a surrogate model is constructed to explore the solution space more efficiently for optimal design; and (4) automated design, where both the predictor and policy are learned and continually adjusted online to significantly reduce human effort in complex design tasks. In chapters 6 and 8 of this dissertation, we aim to enhance performance prediction to facilitate HLS black-box design optimization.

When a larger dataset is available, deep learning algorithms have demonstrated significant performance improvements in EDA. GNNs are one of the most widely used algorithms for this purpose, as graphs provide an intuitive way to model programs, Boolean functions, netlists, and layouts commonly used in many EDA problems [Kha+20; Ma+20b; Ren+22]. This is also true for the HLS problem, where analytical models cannot achieve acceptable accuracy [SW19], but learning algorithms have demonstrated superior performance. However, applying learning algorithms to the HLS problem, which constitutes an early stage of design optimization, can pose considerable challenges due to the extensive and intricate optimization procedures that a design must undergo before reaching its final microarchitecture.

#### 2.4.4 HLS Performance and Resource Modeling

The most time-consuming step in exploring the solution space of HLS design to identify Pareto-optimal points involves running the HLS tool to evaluate each point’s quality. This is because each synthesis process can take anywhere from minutes to hours, and it becomes even more time-consuming if post-implementation design metrics are required. To tackle this challenge, several studies aim to create a predictive model that can estimate the quality of an HLS design, serving as a surrogate for the HLS tool during the exploration process.

One set of studies in this domain focuses on constructing an analytical model to assess the quality of an HLS design by estimating its performance and resource utilization. The

authors in [WLZ17; Zha+17; Zho+16] build the dependence graph of the target application and utilize traditional graph analysis techniques along with analytical predictive models to search for the best design. Although this approach can quickly search through the design space, using an analytical model is inaccurate and it is difficult to maintain the model and port it to other HLS tools. This is because the HLS tools are constantly evolving, and their optimization heuristics are changing either between the tools of the different vendors or the same vendors [SW19]. Zhong et al. [Zho+14] develop a simple analytical model for performance and area estimation. However, they assume that the performance/area changes monotonically by modifying an individual design parameter, which is not a valid assumption as pointed out by Nigam et al. [Nig+20]. To increase the accuracy of the estimation model, a number of other studies restrict the target application to those that have a well-defined accelerator microarchitecture template [Chi+18; CW18; Con+18a; Reg+19; Zac+19] (like our modeling for FlexCNN architecture in Chapter 3), a specific application [Xu+20; Zhe+20], or a particular computation pattern [CC18; Koe+16; Pra+16]; hence, they lose generality.

To the same end, there are other studies that take a data-driven approach and build the predictive model using supervised learning algorithms. They train a model by iteratively synthesizing a set of sample designs and updating the model until it gets to the desired accuracy. Later on, they use the trained model for estimating the quality of the design instead of invoking the HLS tool. To learn the behavior of the HLS tool, these works adopt supervised learning algorithms such as random forest, decision tree, and linear regression to better capture the behavior of the HLS tools [Koe+16; LC13; LLS19; SW12a; Xyd+14; Zho+17]. While this technique increases the accuracy of the model, it is still time-consuming to port the model to another HLS tool in a different vendor or version. Often by changing the HLS tool or the target FPGA, new samples should be collected, which can be an expensive step. After that, for each of them, a new model should be trained to include the new dataset. Furthermore, these related works build a separate learning model per application and the results from one are not transferred to another one. Kwon et al. [KC20] proposed a model using a Multi-Layer Perceptron (MLP) network that can be used across multiple

applications. However, as the input to the model, they only use pragma configurations. As we shall show in Chapter 6, not taking the program semantics into account harms the accuracy significantly.

Some recent studies suggest employing GNNs to predict the quality of a design, given their demonstrated ability to substantially enhance accuracy [WXH22; Ust+20; Wu+22]. Moreover, using GNNs can help unify the model for several applications, as opposed to developing a separate model for each application. Ustun et al. [Ust+20] represent the HLS design (without pragmas) as a Data Flow Graph (DFG) and build a GNN-based model to predict the mapping of arithmetic operations to the DSPs and LUTs, which can improve the accuracy of delay prediction. Similarly, IronMan [WXH22] converts the program (without optimization pragmas) to DFG and predicts the critical path under different resource allocations (DSP or LUT) to the computation nodes using GCNs. Wu et al. [Wu+22] also work with HLS designs without pragmas and construct a hierarchical GNN that first performs node-level classification to predict the resource type (DSP, LUT, or FF) for implementing the node and then uses this information to estimate the critical path as the graph-level prediction. Although optimization pragmas are the primary source for improving the resulting microarchitecture [Chi+22], none of these works include the pragmas in their input representation so their models cannot be used for finding the best design configuration. In this dissertation, we aim to pinpoint the challenges associated with developing such a model and propose solutions to address them. Our goal is to resolve these shortcomings by first employing a graph representation that captures both the program semantics and the pragmas. Then, we develop a framework that is capable of learning from a set of applications, building a *single* predictive model for all of them, and quickly adapting its knowledge to new applications or new versions of the HLS tool.

#### 2.4.5 Design Space Exploration for HLS

As outlined in [SW19], prior research either directly employs the HLS tool [Yu+18] or constructs a model to emulate the HLS tool [Zha+17; MS14] when assessing a design point.

Since it takes several minutes to hours to get feedback from the HLS tool, relying on it for evaluating a possible solution can increase the DSE time significantly. Utilizing a predictive model as the surrogate of the HLS tool can speed up the process, which has made the respective previous works conduct an exhaustive search through all the candidate designs. Nevertheless, the exhaustive search is not a scalable solution for larger design spaces. For instance, the `gemver` program from the Polybench benchmark [YP] has over 100 billion design points, making an exhaustive search unfeasible even when evaluating candidates in milliseconds. Additionally, developing an accurate model for HLS, as discussed in Section 2.4.4, poses a significant challenge.

To avoid dealing with the uncertainty of the HLS tools, another set of studies treats the HLS tool as a black box. Instead of learning a predictive model, they invoke the HLS tool every time to evaluate the quality of the design. To guide the search, they either exploit general application-oblivious heuristics (e.g., simulated annealing [MS14] and genetic algorithm [Sch17]) or they develop custom heuristics [FAP18a; FAP18b; SW12b]. For instance, S2FA [Yu+18] employs multi-armed bandit [Fia+10] to combine a set of heuristic algorithms including uniform greedy mutation, differential evolution genetic algorithm, particle swarm optimization, and simulated annealing. However, as we will discuss in Chapter 5, general hyper-heuristic approaches are unreliable for finding the high QoR design configurations. The authors in [FAP18a; FAP18b] claim that Pareto-optimal design points cluster together. They exploit an initial sampling to build the first approximation of the Pareto frontier and require local searches to explore other candidates. However, the cost of initial sampling is not scalable when the design space is tremendously large (e.g., the scale of  $10^5$  to  $10^{30}$ ). Even if it only samples 1% of the design space (the lowest sampling rate they use), it results in evaluating  $10^3$  to  $10^{28}$  design points which is not feasible. Sun et al. [Sun+21] adapt a (Gaussian process) GP-based Bayesian Optimization (BO) algorithm to explore the solution space. At each iteration, it improves a surrogate model to mimic the HLS tool, by sampling the design space. Again, as the search space grows, it will require more samples to build a good surrogate model which can limit the scalability. Moreover, the computation of a

GP-based BO can be seen to be cubic in the total number of samples (in addition to the time to evaluate the sampled point using the HLS tool), as it wants to calculate the inversion of a dense covariance matrix at each step [Sno+15] which can further limit the scalability of the approach.

Learning algorithms have also been employed to expedite the HLS DSE process in discovering Pareto-optimal points [WXH22; WS20]. This research approach essentially utilizes a data-driven method for the search. For example, IronMan uses a Reinforcement Learning (RL) agent trained to identify optimal resource allocations between DSP and LUT based on user-specified constraints, like minimizing resource consumption or optimizing the critical path. Despite the considerable power of an RL agent, a potential obstacle in its training lies in the need for a substantial dataset and/or a highly accurate predictive model [Mir+21]. In Chapter 5, we propose a dedicated heuristic for swiftly converging to high-quality points in a few iterations, offering an alternative to these existing approaches.

#### 2.4.6 Code Transformation for High-Performance Computing

The organization and structure of the code play a critical role in shaping the final microarchitecture. This is because it impacts the reuse ratio, data dependency, parallelization opportunities, efficient utilization of DRAM bandwidth, latency hiding, etc. HeteroRefactor [Lau+20] and HeteroGen [Zha+22] apply pattern-oriented program edits to convert C codes to synthesizable and performant HLS programs. These repairs involve replacing `malloc` with array-based memory accesses, removing pointers, transforming recursive functions, and optimizing integer and floating-point bitwidths to reduce resource consumption. The generated code can then be further optimized with pragmas to enhance performance. While various works have identified opportunities for optimizing HLS through code transformations [Con+18b; Fin+20], the automation of all these transformations is yet to be addressed and remains largely reliant on manual intervention from programmers. The Merlin Compiler [Con+16b; Con+16a] serves as a successful example of a source-to-source code transformer that automatically applies various code transformations, leading to higher-

quality microarchitecture, as discussed in Section 2.1.1. However, it does not address all necessary code transformations, such as loop transformations, which can affect compute order, parallelization opportunities, etc.

Recent works have focused on optimizing machine learning compilers by integrating diverse code transformations into their search space and selecting the optimal transformations through search or polyhedral optimization [Che+18; Fen+23; Vas+18]. For instance, TVM [Che+18] incorporates various code transformations such as operation fusion, data layout transformation, and latency hiding within its search space and employs a machine learning-based cost model for evaluating each choice. Similarly, TensorIR [Fen+23] utilizes an evolutionary search process for tensor optimization through loop transformations, guided by a learning-based cost model. Developing such models is more straightforward as they target specialized applications rather than general ones and can assume a fixed hardware platform when applying code transformations. This makes it possible to search for changes in computation blocks, tiling, fusion, compute order, etc. As mentioned before, in HLS programming, code transformations can additionally alter the underlying microarchitecture as well. This adds complexity to the development of an accurate cost model, as it demands a larger, more time-intensive database since it has to go through all stages of hardware generation, which can take several hours to complete. Additionally, more complex learning models are needed. We believe that our model-based performance predictors can serve as an initial step toward addressing this challenge and may pave the way for future progress in this field.

Part I

# Customized Computing for Machine Learning Acceleration

In this part of the dissertation, as described in Section 1, we use architecture-guided optimizations to speed up the execution of CNNs and GCNs. Fig. P.1 provides an overview of the work presented here. In Chapter 3, we introduce FlexCNN, a flexible and composable architecture for processing CNNs. Then, in Chapter 4, we shift focus to GCNs and develop StreamGCN, specialized for streaming the processing of many small graphs. Additionally, in Chapter 3, we show how FlexCNN can be developed into an end-to-end framework that automatically generates the most efficient accelerator for a CNN described in TensorFlow (FlexCNN is extended in [Bas+23] to accept ONNX input and support additional convolutional types). Moreover, we create a library that integrates FlexCNN into TensorFlow, allowing the offloading of CNN processing to FPGA. Although the end-to-end development process is demonstrated with FlexCNN, the same approach can be applied to StreamGCN.

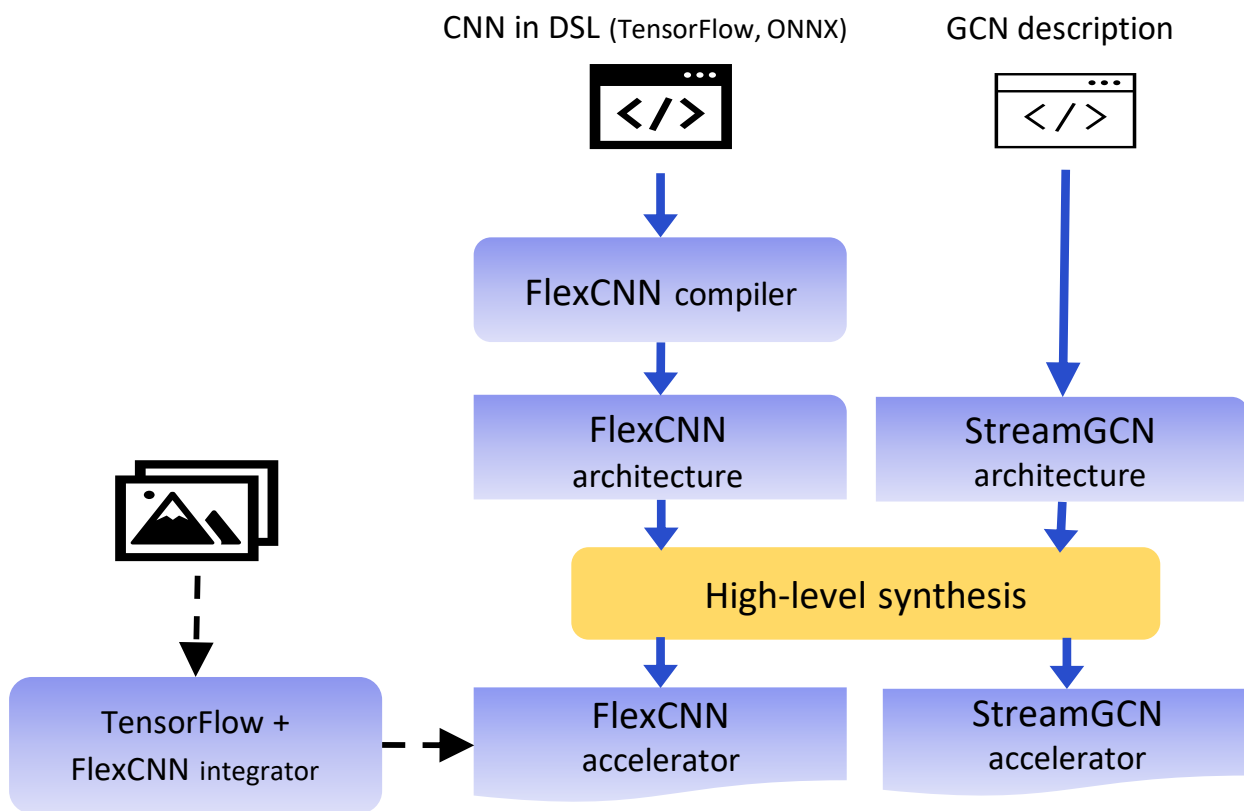


Figure P.1: Overview of Part 1 – Architecture-guided Optimization.



## CHAPTER 3

# FlexCNN: End-to-end Acceleration of Convolutional Neural Networks

The reduced data reuse and parallelism resulting from extensive network pruning and simplification in recent Convolutional Neural Network (CNN) models pose novel challenges for FPGA acceleration. Additionally, the integration of FPGAs into existing machine learning frameworks like TensorFlow can incur substantial overheads without proper optimizations—an aspect often overlooked in previous studies. Our research reveals that a simplistic FPGA integration into TensorFlow may lead to performance degradation of up to  $8.45\times$ . To tackle these challenges, we propose several software/hardware (SW/HW) co-design approaches for end-to-end optimization of CNNs. Introducing a flexible and composable architecture named *FlexCNN*, our solution achieves high computation efficiency for various convolutional layer types through dynamic tiling and data layout optimization. FlexCNN is further integrated into the TensorFlow framework via a fully-pipelined SW/HW integration flow, mitigating the overheads associated with TensorFlow-FPGA communication and other non-CNN processing stages. Using OpenPose, a widely-used CNN-based application for human pose recognition, as a case study, our experimental results demonstrate a  $2.3\times$  performance improvement with FlexCNN architecture optimizations. The pipelined integration stack further contributes to a  $5\times$  speedup. In total, the SW/HW co-optimization yields an impressive  $11.5\times$  speedup, resulting in an end-to-end performance of 23.8FPS for OpenPose with floating-point precision<sup>1</sup>.

---

<sup>1</sup>FlexCNN codes are available at <https://github.com/UCLA-VAST/FlexCNN>

### 3.1 Introduction

CNNs are widely used in many Machine Learning (ML) applications and have evolved quickly over the years. Consequently, the interest in using FPGAs to accelerate CNN computation is on the rise, driven by their high energy efficiency and performance (e.g., [Aki+18; BZH18; Gua+17; Li+16; NSW18; Sud+16; Wei+17; Zha+15; Zha+18a; Zha+18b; SFM17b]). However, the recent advancement in CNN models and FPGA-based CNN acceleration has brought several new challenges:

**Challenge 1: Performance disparity of different CNN layers:** Real-world deep learning applications may employ complex network architectures. In addition, many state-of-the-art efficient networks such as MobileNetV1 [How+17] incorporate Depth-wise Separable Convolutions (DSC) (reviewed in Chapter 2.2.1.1) to decrease the computation cost. Additionally, MobileNetV2 [San+18] utilizes inverted Residual Bottleneck Blocks (RBB) (explored in Chapter 2.2.1.2) to further reduce the computation complexity. These layers reduce the computation cost but keep the same feature map size; this can make the layer more communication-bound and reduce the computation efficiency. Apart from this irregularity, different layers of a CNN have different characteristics in terms of their input and output number of channels, feature map size, and filter size. All these differences result in various Computation-to-Communication (CTC) ratios from layer to layer. Therefore, it is important to handle these layers differently given the performance disparity across these layers. We found that tiling factors can play an important role in the performance. Zhang et al. [Zha+15] showed that the CTC ratio of a single convolutional layer varies with different tiling factors. Yang et al. [Yan+18] highlighted the importance of choosing proper tiling factors for data reuse in the near and faster memory (on-chip storage for FPGAs) for the overall latency and energy efficiency. These studies lead us to consider using different tiling factors across the network. Figure 3.1 depicts the influence of different tiling factors on the attainable performance for each layer in a CNN network example shown in Figure 3.3 for the first 24 convolutional layers (each RBB is divided into two layers). We compare the

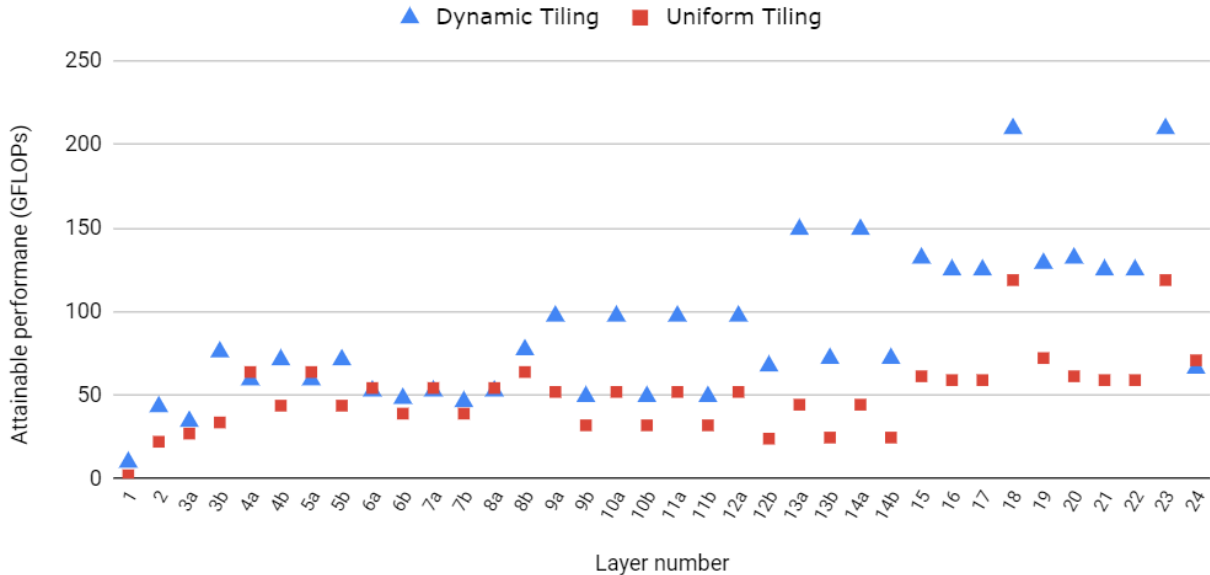


Figure 3.1: Performance comparison of hardware accelerators using uniform and dynamic tiling factors for the first 24 convolutional layers in the CNN network of Figure 3.3.

performance of using a single set of tiling factors (uniform tiling) to using different tiling factors for each layer (dynamic tiling). For the uniform tiling, we chose the tiling factor that reduces the latency of the entire network. For the dynamic tiling, we focused on each layer and selected the best tiling factor accordingly. Experimental results show that dynamic tiling can speed up the performance of the whole network by  $1.7\times$ .

**Challenge 2: Integration overheads of using FPGA in ML frameworks:** When processing a CNN application in modern ML frameworks such as TensorFlow [Aba+16], the complete stack consists of reading the input, computing the CNN, processing the result, and displaying and writing the result. Previous works have only focused on optimizing the CNN kernel on FPGA (e.g., [Aki+18; BZH18; Gua+17; Li+16; Sud+16; Wei+17; Zha+15; Zha+18b]). This is because the CNN computation is the most time-consuming step of the whole stack. Hence the rest of the overheads are ignored. While several works [Gua+17; NSW18] have focused on accelerator *generation* from TensorFlow-described networks, they did not address the challenges of *integrating* an accelerator into TensorFlow. By integrating our accelerator with TensorFlow, we can directly run networks from TensorFlow on an

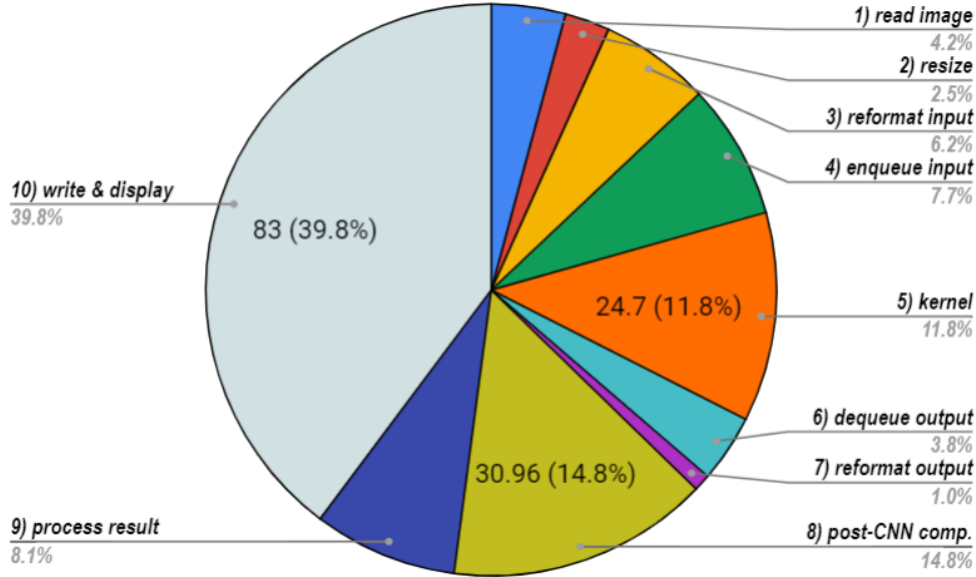


Figure 3.2: Runtime breakdown of an FPGA-based CNN acceleration pipeline in TensorFlow.

FPGA. Integrating FPGA into TensorFlow introduces a new set of overheads: communication between TensorFlow and FPGA and the communication between the host and the FPGA kernel itself. Figure 3.2 shows the breakdown of the end-to-end runtime for processing a  $384 \times 384$  RGB image using the network in Figure 3.3. These steps are listed and described in Section 3.4. The CNN processing time with our accelerator (referred to as the kernel) accounts for only 11.8% of the overall runtime. This underscores the need for a comprehensive end-to-end SW/HW co-optimization. Our experiments show that such optimization can further increase the end-to-end performance of this network from 4.8FPS to 23.8FPS, leading to a  $5\times$  speedup.

To solve the challenges above, in this chapter, we propose an FPGA-based CNN accelerator named FlexCNN. It employs dynamic tiling and data layout transformation to adapt to the performance disparity of different CNN layers. The accelerator is further integrated into the TensorFlow framework. To mitigate the large integration overheads, we propose a two-level pipeline to overlap the overheads with the computation.

We use OpenPose [Cao+17] as a driving application to test FlexCNN’s performance. OpenPose is designed to jointly learn body part locations and their associations. Its high

computation cost and irregular network architecture make it a challenging network to implement on FPGA. To our knowledge, there is only one prior work [Aki+18] that has implemented OpenPose on FPGA. Its CNN kernel processes an image in 42.6ms. Whereas, FlexCNN can process an image in 24.7ms, leading to  $1.7\times$  speedup.

In summary, the key contributions of this chapter are:

- A flexible and composable accelerator architecture, named FlexCNN, supporting dynamic tiling and data layout transformation to improve computation efficiency for running CNNs.
- A TensorFlow to FPGA runtime environment for running CNN on FPGA from TensorFlow and an optimized software/hardware pipeline to mitigate the integration overheads.
- A fully automated compilation system for the FlexCNN architecture.
- The fastest FPGA accelerator to run OpenPose. FlexCNN yields a  $2.3\times$  speedup from supporting dynamic tiling and optimized data layout. Besides, our framework achieves  $5\times$  speedup from software/hardware pipelining, resulting in a final performance of 23.8FPS. In addition, FlexCNN is  $3.8\times$  more energy efficient than GPU.

## 3.2 Application Driver - OpenPose

OpenPose [Cao+17] is the winner of the COCO 2016 Keypoints Challenge that can detect 2D poses of multiple people in an image. OpenPose network first extracts the features of the input image using the first 10 layers of VGG-19 [SZ14]. This is the backbone of the network. These feature maps are the inputs to a two-branch network. The first branch detects confidence maps, representing body part locations, and the second branch detects part affinity fields, a set of 2D vectors showing the location and orientation of the limbs. The results of these two branches are concatenated with the feature maps from the backbone network and form the input for the next stage. After several iterations (stages), these

branches would refine and produce the final predictions.

This network is interesting to us since it is a deep network and has an irregular architecture compared to modern CNN-based deep learning applications. Instead of just a linear forward path where each layer consumes the result of its previous layer, it has concatenation layers that need extra data movement. Moreover, to decrease the computational complexity of the network, we adopt a customized version of OpenPose [Kim18]. This variant substitutes the backbone with a modified version of MobileNetV2 [San+18], incorporating RBB, and utilizes DSC for the rest of the network, following the trend in the ML community. These variations further add to the complexity of the network which makes it suitable for evaluating our accelerator. Figure 3.3 depicts the network topology of this version, we call this network OpenPose-V2. Due to the space limitation, we only show the convolutional layers. Each convolution is followed by ReLU and batch normalization layers.

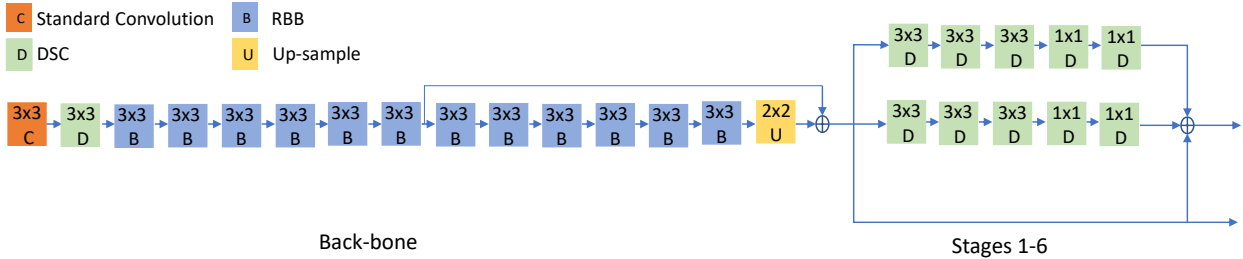


Figure 3.3: OpenPose-V2 network topology.

### 3.3 The FlexCNN Architecture

The basic layers in different CNNs are standard convolution, depth-wise convolution, ReLU, bias/batch normalization, downsampling/pooling, upsampling, and add. The rest of the building blocks are usually a combination of these layers. Thus, the FlexCNN architecture has these components as building blocks and uses them to compose different parts of the network. This strategy can improve the hardware utilization on FPGA.

Figure 3.4 depicts the detailed architecture of FlexCNN. It implements the dataflow architecture to process the network by fusing the operation of different layers of the network

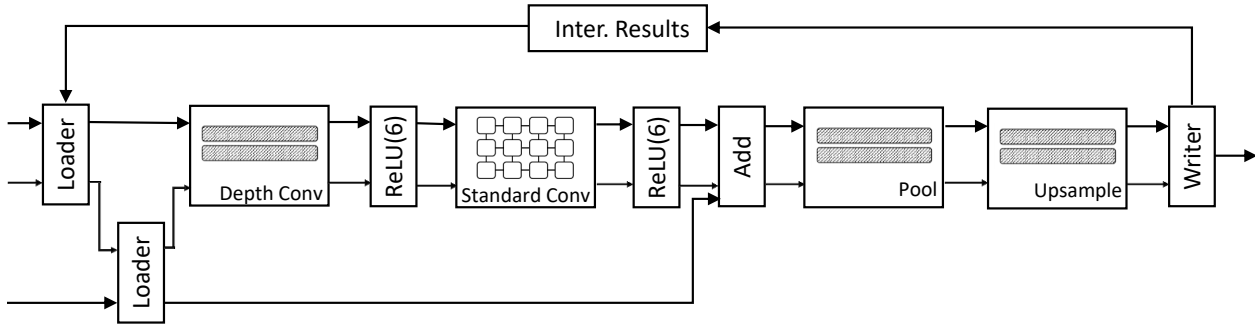


Figure 3.4: The FlexCNN architecture overview.

in one pass of the architecture. Hence, we can reduce the communication time between them by avoiding redundant off-chip memory transactions. Each pass of the architecture can load up to two sets of input feature maps, those from the current pass and a previous pass. Loading the feature maps from a previous pass is required for supporting convolutional layers like RBB where the results of the current and previous convolutional layers need to be added together.

Since the convolutional operations are the most computation-intensive layers, we pack and fuse as many operations as possible in one pass with them. As such, the loaded feature maps will pass through modules including the depth-wise convolution module (*Depth Conv*), ReLU(6)<sup>2</sup> module, standard convolution module (*Standard Conv*), ReLU(6) module, add module, max-pooling module (*Pool*), and bi-linear upsampling module (*Upsample*). The final results will be written out to DRAM via *Writer*. The operations in the batch normalization layer and bias layer are fused into ReLU(6) modules. Each of the convolutional operations may be followed by any of the ReLU(6) or normalization layers. Hence, we put the ReLU(6) module after both the Depth Conv and Standard Conv modules. In fact, this is an example of how we can compose the architecture from the building blocks designed in FlexCNN.

For Standard Conv, the systolic array architecture is used. It is generated using the systolic array compiler in Wang et al. [CW18]. It can compute a standard convolutional layer with any given filter size. We implement line-buffer-based streaming architectures

<sup>2</sup>ReLU6 outputs the minimum of the value 6 and a normal ReLU.

Table 3.1: FlexCNN design parameters with their explanations.

Design Parameters	Explanation
$Th(k), Tw(k), Tn(k), Tm(k)$	Tiling factors for $H, W, N,$ and $M$ for layer $k$
$SIMD$	SIMD lanes for all modules
$SA\_ROW, SA\_COL$	Rows and columns of the systolic array kernel

H: height, W: width, N: input channels, M: output channels

for Depth Conv, ReLU(6), Add, Pool, and Upsample modules using a similar stencil-based architecture as in [Chi+18]. All these modules are parameterized by the factors listed in Table 3.1, which will be explored by the Design Space Exploration (DSE) engine covered in section 3.3.1.1, for achieving the optimal performance.

Note that all the modules can be bypassed if they are not being used for a specific pass of the architecture. We apply double buffering in both the Loader modules and the Writer module. Furthermore, if the outputs of the whole pass can fit into the on-chip buffer, the data will be pushed into on-chip buffers and directly fetched by the Loader to further save the off-chip communication time.

### 3.3.1 Dynamic Tiling

Tiling is applied when processing the network to improve the data locality and minimize communication. Table 3.1 summarizes the tiling factors employed in FlexCNN. When the tiling factors are not sub-multiples of the tiled dimensions, redundant computation is introduced which degrades the performance of the design. As explained in Section 3.1, in a normal CNN network, the types and configurations of different layers vary from each other. Therefore, the optimal tiling factors will be different from each other as well. We have observed that using uniform tiling factors for the whole network will lead to up to  $1.7\times$  performance slowdown compared to the ideal case of using different tiling factors across layers. Therefore, in this work, we apply dynamic tiling by reconfiguring the tiling factors of the accelerators on-the-fly for different layers to maximize the performance. This will bring some hardware overheads to support the dynamic tiling. However, such overheads are negligible compared



to the performance improvement. Section 3.6 evaluates the impacts of this technique in detail.

Previous works such as [Wei+18; Zha+18b; SFM17b] have also emphasized the need for different tiling factors across layers. Our architecture distinguishes from the previous work by changing all the tiling factors across each layer dynamically, whereas previous work only adjusted part of the tiling factors or used several accelerators, each with distinct uniform tiling factors on-chip. Eq. 3.1 shows the restriction on the tiling factors.

$$\begin{aligned}
 Tw(k) &= c_1 \times SA\_COL \\
 Tm(k) &= c_2 \times SA\_ROW \\
 Tn(k) &= c_3 \times SIMD \\
 Tm(k) &= Tn(k + 1)
 \end{aligned}
 \tag{3.1}$$

In FlexCNN, the width and output channels of the feature maps are mapped to columns and rows of the systolic array respectively. As a result, for each layer,  $Tw(k)$  and  $Tm(k)$  should be multiples of their respective systolic array dimension. The reduction of multiple input channels is computed in parallel inside each PE of the systolic array, which is defined as the SIMD lane. This implies that  $Tn(k)$  should be a multiple of SIMD lane.  $Th(k)$  can be any arbitrary value. As mentioned before, the computation in the depth conv module can be seen as a stencil kernel. Figure 3.5 depicts the  $3 \times 3$  stencil window connected by line buffers. We realize dynamic tiling by connecting consecutive rows of the line buffer via a MUX, enabling data feeding from different locations.

### 3.3.1.1 Design Space Exploration

Given the network, the accelerator architecture, and the FPGA’s resource information, we will perform DSE to select the optimal design parameters that maximize the performance on the target FPGA. Table 3.1 lists the design parameters to be determined.

Two analytical models  $resource\_est()$  and  $latency\_est()$  are built for estimating the resource usage and latency of designs. Currently, the resource model estimates Block RAM

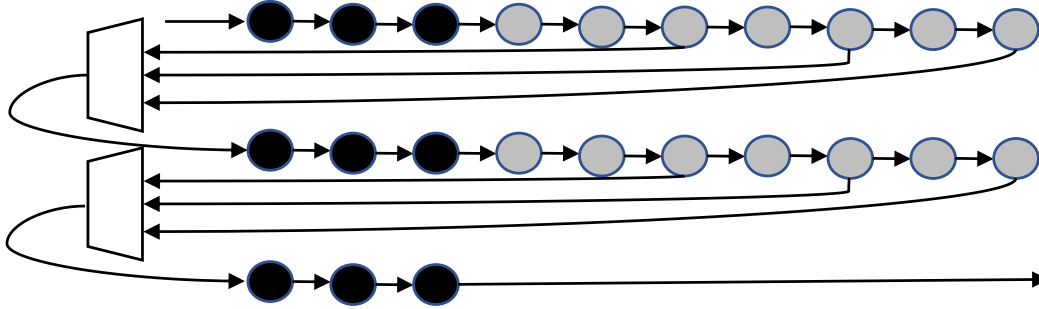


Figure 3.5: Architecture support for dynamic tiling in the Depth Conv module for a  $3 \times 3$  filter with Tw (tiling factors for width) of size 6/8/10.

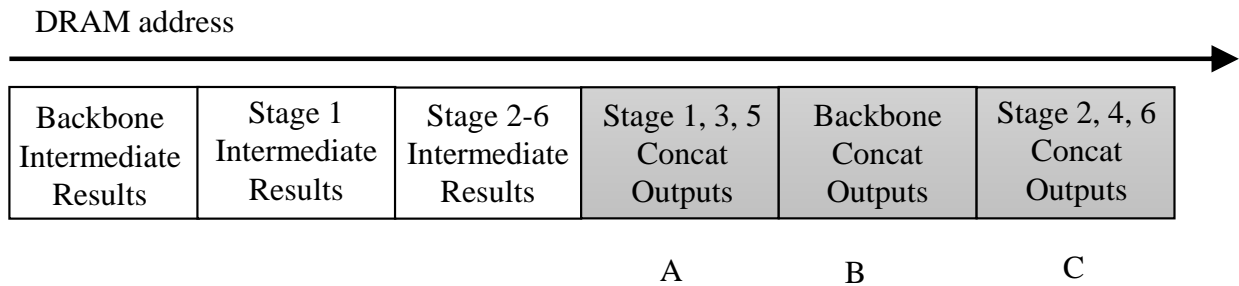


Figure 3.6: Data organization to apply the concatenation operation on the fly. Example is taken from the OpenPose network.

(BRAM) and DSP usage which are usually the bottleneck of designs. The DSE process will sweep through the design space with all feasible combinations of design parameters. For each design parameter list, the resource usage is examined first. Designs that over-utilize the resources will be pruned away. Then, we follow a greedy algorithm to select the optimal tiling factors that minimize the latency layer by layer. The DSE process finishes within minutes on a standard workstation. Moreover, the latency model can predict the runtime of the network with a 6% error rate.

### 3.3.2 Data Layout Optimization

Data layout optimizations are applied to reduce the number of accesses to DRAM (off-chip memory) and increase the effective DRAM bandwidth. The first optimization is on the

concatenation layers. A CNN network may contain blocks that concatenate the results of several layers. For example, as shown in Figure 3.3, after each stage in the OpenPose-V2 network, results from two branches will be concatenated with the outputs from the backbone network. This then serves as the inputs for the following stages. We preprocess and adjust the location that each output should be stored in to deal with the concatenation on-the-fly. Figure 3.6 presents the optimized data organization of the network.

The outputs of the backbone (region  $B$ ) and each stage (regions  $A$ ,  $C$ ) are placed close to each other as shown in Figure 3.6. To be more specific, the outputs of Stage 1 will be written to region  $A$ . The regions  $A$  and  $B$  will serve as the inputs of Stage 2. In Stage 2, the outputs will be written to region  $C$ . The regions  $B$  and  $C$  will then serve as the inputs of Stage 3. The outputs of each stage are written to regions  $A$  and  $C$  in a round-robin fashion. With this layout, the outputs of stage branches are concatenated on-the-fly, eliminating unnecessary off-chip DRAM movements.

To further improve the effective DRAM bandwidth, we change the data layout of the feature maps from  $N(k) \times H(k) \times \frac{W(k)}{Tw(k)} \times Tw(k)$  to  $\frac{N(k)}{Tn(k)} \times H(k) \times \frac{W(k)}{Tw(k)} \times Tn(k) \times Tw(k)$ . This allows us to increase the burst length from  $Tw(k)$  to  $Tn(k) \times Tw(k)$ . A DSC layer can easily become communication-bound because of its low CTC ratio since it mostly uses  $1 \times 1$  convolution filters. In this case, when the filter size of the next layer is  $1 \times 1$ , since there is no overlapped region between different tiles, we further change the data layout to  $\frac{N(k)}{Tn(k)} \times \frac{H(k)}{Th(k)} \times \frac{W(k)}{Tw(k)} \times Tn(k) \times Th(k) \times Tw(k)$ . It further increases the burst length for these layers to  $Tn(k) \times Th(k) \times Tw(k)$ . For other filter sizes, padding is applied because a tile of  $Tn(k) \times Th(k) \times Tw(k)$  does not have all the data needed for the computation. We need to have  $(p - 1)$  and  $((p - 1) \times Th(k) + (p - 1)^2)$  extra DRAM accesses with burst length of  $Tn(k) \times Tw(k)$  and  $Tn(k)$  respectively to fetch all the data ( $p$  denoting the filter size). This increases the number of DRAM accesses with a burst length of  $Tn(k)$ , which further increases the communication time, making this data layout inefficient.

### 3.4 TensorFlow Integration

We chose TensorFlow as our ML framework since it is being widely used for inference in the ML community (e.g., [Mat+17; Ign+18]). To invoke the FPGA from TensorFlow, we redefine the nodes in the original computation graph of the TensorFlow implementation. All computation nodes of CNN are merged into one node that is executed by the FPGA. The rest of the graph is still processed on the CPU.

When FPGA is connected to TensorFlow, the whole integration stack consists of the following steps: 1) reading the inputs of CNN, 2) preprocessing the input which includes stages like image resizing, 3) reorganizing the initial data layouts in CPU memory to match the format required by the accelerator, 4) transferring data from CPU to FPGA device memory, 5) computation on FPGA, 6) fetching the results back via PCIe, 7) reformatting and passing the results to TensorFlow, 8) non-CNN computation stages on CPU, 9) processing the results (e.g., estimating the human poses based on the attained results and drawing them for the OpenPose network), and 10) writing out and displaying the results.

Figure 3.2 shows the breakdown of these stages in the OpenPose application for a  $384 \times 384$  RGB input. Among the whole pipeline, which takes 208.8ms, the FPGA computation (after acceleration) in Step 5 only requires 11.8% of the total time. The results show that the integration overheads have led to  $8.45 \times$  performance slowdown. To reduce these overheads we have applied an optimized SW/HW pipelining.

A two-level pipelining is applied on the whole integration stack that enables the simultaneous processing of the aforementioned steps. The first level overlaps TensorFlow’s overheads (steps 1, 2, 9, 10) with the rest of the steps. The second one overlaps FPGA’s computation (Step 5) with data movement steps (steps 3, 4, 6, 7).

Figure 3.7 illustrates the first level of this pipeline, which is applied at the TensorFlow level. The numbers in the figure show the related step numbers. Steps 1, 2, 9, and 10 and the rest of the steps are assigned to different processes connected by queue. Therefore, steps 1, 2, 9, and 10 are overlapped with FPGA-related steps. The overall performance is determined

by the stage with the longest latency.

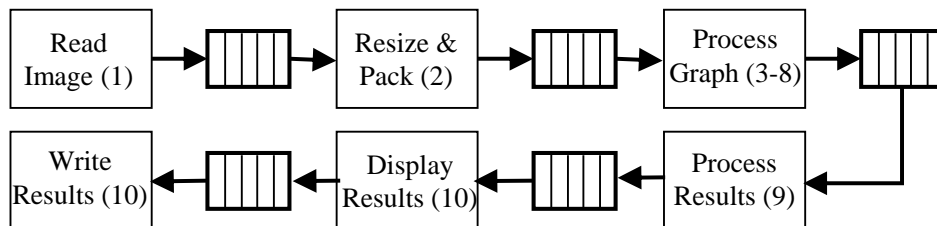


Figure 3.7: First level of pipeline when integrating the FPGA to TensorFlow.

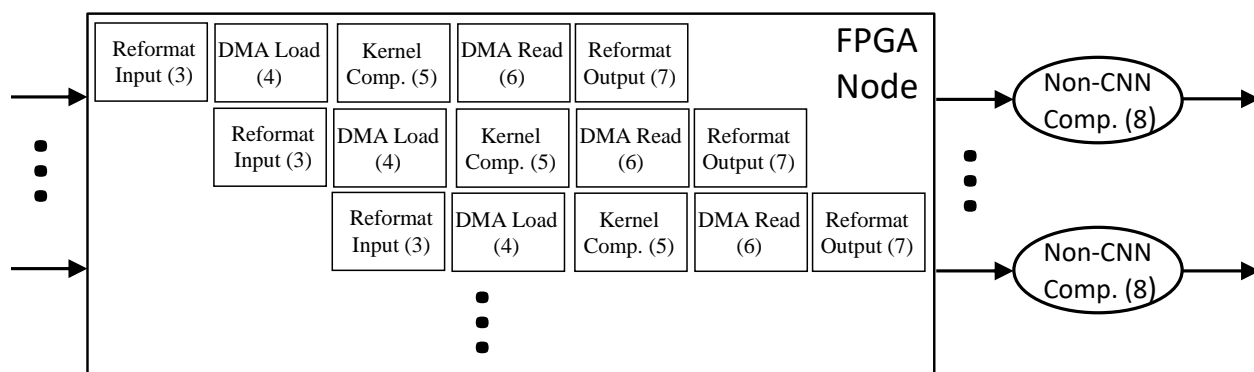


Figure 3.8: The overview of the Process Graph stage in the first level of pipeline, creating the second level of pipelining.

To further improve the performance, we fully pipeline the communication and computation of FPGA, which consists of steps 3 to 7. This creates the second level of the pipeline. To allow pipelining, a batch of images is sent to FPGA. For a certain batch size, the additional latency incurred by batch processing is dissolved when the first level of the pipeline is applied. After the FPGA finishes processing the batch, the results are passed back to TensorFlow, and the non-CNN computations are done in parallel for all the images. Figure 3.8 depicts the redefined graph that we use to achieve such a pipeline. With this optimization, the data movement steps are overlapped with kernel computation, and the latency for non-CNN computation (Step 8) is amortized for the whole batch. Note that such deep SW/HW pipelining techniques were also used in [Che+16; CWY18] for integrating FPGA accelerators to Spark-based applications.

### 3.5 FlexCNN Compilation System

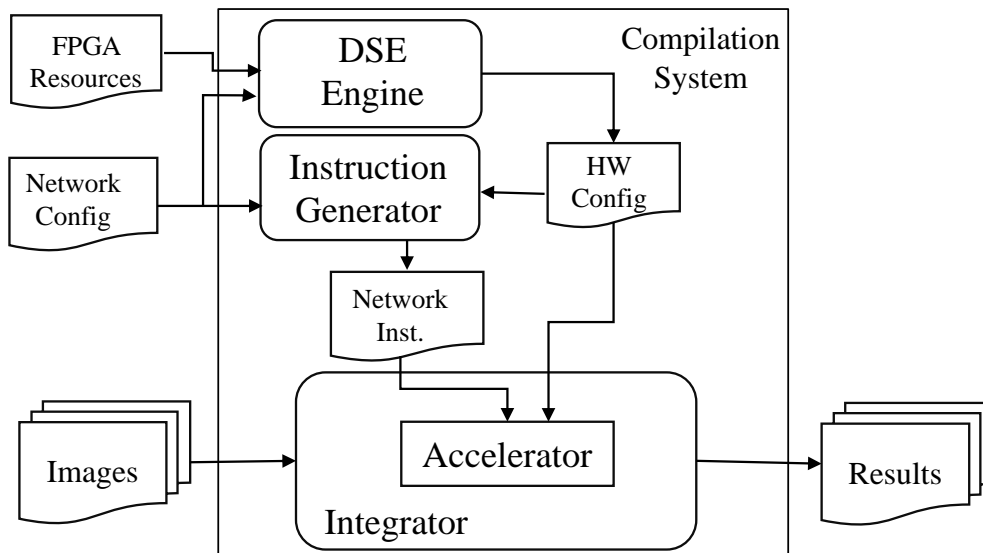


Figure 3.9: FlexCNN compilation system overview.

Figure 3.9 depicts the overview of our compilation system, which is composed of three components: design space exploration engine, instruction generator, and integrator.

- **Design space exploration engine.** The CNN network description file parsed from TensorFlow and the available target FPGA resources are fed into the DSE engine to search for the optimal hardware configuration parameters. Once found, these parameters are then used for generating the accelerator. The DSE process was explained in Section 3.3.1.1
- **Instruction generator.** The instruction generator takes the CNN network description file and the accelerator hardware parameter file as the inputs and creates one instruction for each CNN layer. The non-convolutional layers (e.g., bias, ReLU(6)) are fused with adjacent convolutional layers to improve the computation efficiency. The instruction generator produces one VLIW-like instruction for each of these fused layers. This instruction contains the enable signal for each of the modules, the layer configurations, and tiling factors. An RBB (Section 2.2.1.2) contains two standard

convolutional layers in one block. As we only have one standard convolution module in the accelerator, we would divide this block into two passes. The first pass performs the first convolution and the next one computes the rest of the block.

- **Integrator.** The integrator takes in the FPGA accelerator and integrates it into the TensorFlow framework, performing the end-to-end processing task. The optimizations on the integrator were covered in Section 3.4.

## 3.6 Experimental Results

### 3.6.1 Experiment Setup

The FlexCNN architecture is described using Vivado HLS C++ [Xila]. The target platform is AMD Xilinx Virtex Ultrascale+ VCU1525. The design is synthesized and implemented using AMD Xilinx SDAccel 2018.3. We use OpenPose-V2 network explained in Section 3.2 to test our work. The accelerator can get any input size. For this section, we are reporting the results of when it takes RGB images of  $384 \times 384$  with floating-point precision as inputs.

### 3.6.2 Hardware Optimizations

The target FPGA platform comes with four DDR banks. In our implementation, we use two DDR banks, assigning feature maps and weights (including bias) to two separate DDR banks. All the architecture choices are parameterizable and can be adjusted based on the target FPGA. As explained in Section 3.3.1.1, these parameters are tuned with our DSE engine which uses analytical models to estimate latency along with BRAM and DSP utilization. Our experiments with the OpenPose-V2 application on AMD Xilinx VCU1525 show that our models have less than 6%, 11%, and 7% error in predicting latency, BRAM, and DSP, respectively. We found that the following configurations work best for this application on AMD Xilinx VCU1525. The systolic array for our standard conv module is organized as an  $8 \times 8$  array with a SIMD factor of 8. For the rest of the modules, we use the same SIMD

factor. The maximal tiling factors for  $Tn, Tm, Th, Tw$  are 64, 64, 12, and 96, respectively. Table 3.2 shows the frequency and resource utilization under this configuration.

Table 3.2: Frequency and resource utilization of FlexCNN implementing OpenPose-V2 on the AMD Xilinx VCU1525 board.

Precision	Frequency	LUT	FF	BRAM	URAM	DSP
float 32-bit	242.9MHz	43%	40%	60%	15%	50%

Table 3.3 shows the benefits of dynamic tiling and data layout transformation. We can see that these optimizations increase the performance by  $2.3\times$ . Figure 3.1 depicts the performance gain of using dynamic tiling in a layer-by-layer fashion for the first 24 convolutional layers. Table 3.4 shows how applying dynamic tiling and dynamic data layout affects the tiling factors and the effective DRAM bandwidth for the first layer of the last RBB in OpenPose-V2 compared to a design without these optimizations. The filter size for this layer is  $1 \times 1$  which means it can use the optimized data layout with burst length of  $Tn(k) \times Tw(k) \times Th(k)$  as described in Section 3.3.2. This data layout, along with the best tiling factor used for this layer increases the effective DRAM bandwidth and CTC ratio by  $2.8\times$ , leading to a performance improvement of  $6.1\times$ .

Table 3.3: Performance of FlexCNN under different settings tested on OpenPose-V2. All Dynamic refers to the case where dynamic tiling and data layout reorganization are applied. All Uniform maintains a uniform tiling factor and data layout across all layers, specifically chosen for optimal performance within the uniform setting.

Model	Precision	Frequency (MHz)	Runtime (ms)	
			(1)	(2)
All Uniform	float 32-bit	237	57.7	41.5
All Dynamic	float 32-bit	242.9	35.6	24.7

(1): Without applying DRAM organization for concatenation layers

(2): With applying DRAM organization for concatenation layers

We further test the DSP efficiency of our design on a given convolutional layer. Of all



Table 3.4: Performance impacts of dynamic tiling and data layout transformation for a given layer in OpenPose-V2 with  $1 \times 1$  filter size.

Model	Th	Tw	Tn	Tm	Effective DRAM Bandwidth (GB/s)	CTC	Performance (GFLOPs)
All Uniform	12	48	32	32	4.31	14.9	24.4
All Dynamic	12	24	48	48	12.05	41.3	149.2 (6.1 $\times$ )

the DSPs, 78.7% of them are used in the Standard Conv module and 11.2% in the Depth Conv module. We measure DSP efficiency using two factors: the total number of DSPs in the design and the number of DSPs of the main computation modules used by that layer. All the tests are on a  $256 \times 384 \times 384$  input, producing 256 output channels. Table 3.5 summarizes the results. DSC layers take  $K^2 \times$  less computation, making them communication-bound as shown in Figure 3.10. This figure depicts that DSC layers fall in the memory-bound region of the roofline model since they have lower CTC ratio. Therefore, we achieve lower computation efficiency in these layers. Additionally, it shows that the data layout optimization for the DSC with the  $1 \times 1$  filter increases the burst length. This helps to increase the effective DRAM bandwidth, leading to a performance improvement over the  $3 \times 3$  DSC.

Table 3.5: Performance of FlexCNN across different convolutional layers on the AMD Xilinx VCU1525 board.

Layer	Runtime (ms)	Performance (GFLOPs)	DSP <sub>total</sub> efficiency	DSP <sub>used</sub> efficiency
Conv 3x3	709.3	245.2	73.8%	93%
Conv 1x1	80.2	240.9	72.6%	91.4%
DSC 3x3	113.4	176.3	53.1%	58.6%
DSC 1x1	84.1	230.8	69.5%	76.7%

### 3.6.3 Integration Optimizations

In this section, we evaluate the effect of our integration optimization. FlexCNN runs at 24.7ms, which translates to a peak performance of 40.5FPS. However, without proper opti-

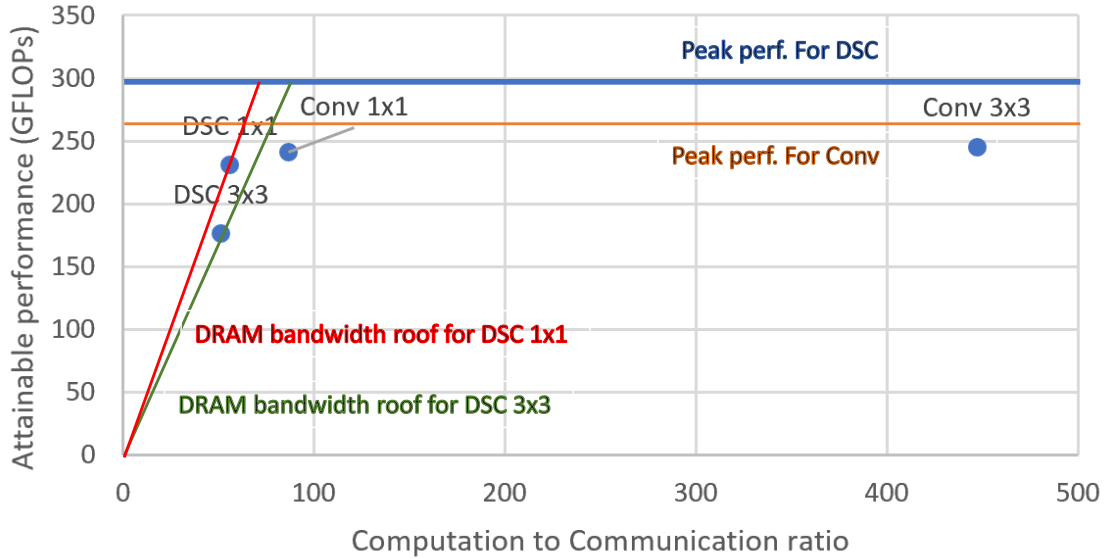


Figure 3.10: Layers in Table 3.5 under the roofline model.

mization, the direct integration into the TensorFlow framework only leads to the performance of 4.8FPS, as shown in Table 3.6. Table 3.6 summarizes the impacts of two-level pipelining on the overall performance. We are using a batch of 16 for the OpenPose-V2 network to enable pipelining of the FPGA computation with the host to FPGA memory transactions since it produces the best performance and smoothest output when displaying the result. With the two-level pipelining, we achieve up to  $5\times$  speedup, which leads to the final performance of 23.8FPS.

Table 3.6: Performance impacts of the integration (FlexCNN to TensorFlow) optimizations. The first level overlaps TensorFlow’s overheads with the FPGA-related steps. The second one further overlaps FPGA’s computation with data movement steps.

Version	Runtime frame (ms)	Perf. (FPS)	Speedup
Original	208.8	4.8	1
1st pipeline	97.1	10.3	2.1
2nd pipeline	42	23.8	5

### 3.6.4 Comparative Studies

To the best of our knowledge, there is only one work [Aki+18] that has implemented a variant of OpenPose on FPGA. However, they take a different approach. They reduce the computation cost of the original network by making the weights sparse and using only two stages after the backbone network. Furthermore, they quantized the data to a 16-bit fixed point and stored all feature maps and weights on-chip. Although exploiting on-chip storage for all the parameters and feature maps have been an attractive approach [Sud+16; Li+16], it limits the scalability of the design. Besides, after these modifications, they neither reported their network’s computation cost nor their architecture’s resource utilization. Thus, we cannot compare our results to theirs directly. Instead, we have compared our results against the network implementation using TensorFlow on CPU and GPU.

The target CPU is a 56-core Intel Xeon CPU E5-2680 v4 that operates at 2.40GHz. For GPU, we use the NVIDIA Tesla V100 GPU and it uses cuDNN [Che+14] to run the network. To have a fair comparison of the latency of running the network on different platforms, we measure the runtime of a single image inference using OpenPose-V2 network. Table 3.7 summarizes the results. The runtime considers only the CNN inference time on RGB images of size  $384 \times 384$ . For both the FPGA and GPU, the time to transfer the data from host to device and device to host is excluded from the measurement. We also measure the dynamic power on each platform, which is calculated as the difference of the hardware power when running and not running the application. Both GPU and FPGA suffer from the low data reuse and degree of parallelism of this network (this is why FlexCNN’s performance on this network is only 117GFLOPs). However, FlexCNN is  $3.8\times$  more energy efficient than GPU.

## 3.7 Conclusion

The rapid evolution of CNN networks has brought new challenges to FPGA acceleration. In this chapter, we identify two major challenges including the performance disparity of different CNN layers and the high overheads of integrating the FPGA into an ML framework.

Table 3.7: Performance comparison of different hardware platforms running OpenPose-V2.

<b>Platform</b>	<b>Frequency (GHz)</b>	<b>Runtime (ms)</b>	<b>Dynamic Power (W)</b>
CPU	2.4	99.3	17
GPU	1.4	25.3	38
FPGA (FlexCNN)	0.243	24.7	10

To tackle these two challenges, we propose an accelerator named FlexCNN which employs dynamic tiling and data layout optimization to improve the hardware efficiency across layers. These two techniques achieve  $2.3\times$  speedup on the studied Openpose-V2 network. Furthermore, we propose a two-level integration pipeline to reduce the integration overheads. It adds another  $5\times$  speedup of the overall performance. At last, we are able to meet the requirement of real-time processing with 23.8FPS using these optimization techniques.

## CHAPTER 4

# StreamGCN: Accelerating Graph Convolutional Networks with Streaming Processing

In the previous chapter, we developed and implemented an efficient and composable architecture named FlexCNN to accelerate the most popular deep learning algorithm on images, CNN. While there have been many studies on hardware acceleration for deep learning on images, there has been a rather limited focus on accelerating deep learning applications involving graphs. The unique characteristics of graphs, such as irregular memory access and dynamic parallelism, impose several challenges when the algorithm is mapped to a CPU or GPU. To address these challenges while exploiting all the available sparsity, we propose a flexible architecture called *StreamGCN* for accelerating Graph Convolutional Networks (GCN), the core computation unit in deep learning algorithms on graphs. The architecture is specialized for streaming processing of many small graphs for graph search and similarity computation. The experimental results demonstrate that StreamGCN can deliver a high speedup compared to a multi-core CPU and a GPU implementation, showing the efficiency of our design.

### 4.1 Introduction

Graphs serve as the foundational data structure within data centers and find extensive applications across diverse domains such as recommender systems, social networks, and the World Wide Web. Despite their widespread use, graphs are mainly unstructured and have high dimensionality, resulting in computational complexity during processing. Notably, nu-

merous graph algorithms, such as Graph Edit Distance [SF83] and Maximum Common Subgraphs [BS98], are known to be NP-complete [Zen+09; Kan92]. This challenge has prompted researchers to leverage deep learning techniques for graphs, aiming to extract structured low-dimensional features. Within this context, GCNs [KW16] have gained widespread adoption. GCNs assign feature vectors, termed "node embeddings", to graph nodes to represent their roles. Similar to CNNs, GCNs employ multiple layers where node features propagate, aggregating rich information about the input graph. In each layer, the GCN updates node features by aggregating neighbors' features and passing the result through a filter. GCNs have demonstrated success across various domains, including but not limited to traffic prediction [Zha+19], enabling web-scale recommender systems [Yin+18a], molecular footprint calculation [Duv+15], and solving EDA tasks [Hua+21].

While some graph data tend to scale rapidly, many graph data are naturally limited in size, for example, chemical compounds and molecules [NCI04; SI15; Bol+08; Che+19] that have a wide application in different domains including drug development, quantum mechanics, physical chemistry, biophysics [Wu+18; Che+19]; the GREC database, which represents symbols from architectural and electronic drawings [DV05]; the Fingerprint database [WW92], etc [Wu+18; RB08]. The average number of nodes for graphs within these databases typically ranges from 5 to 50.

Because of the vast application of small graphs, numerous algorithms have been proposed to obtain their information [Ish+21; Li+19; Ma+20a; QBS20; Bai+19]. In particular, SimGNN [Bai+19] proposed a GCN-based approach to learn a similarity score for such graphs. It demonstrates that a GCN-based approach can approximate the GED [SF83] with high accuracy; hence, it expedites the graph similarity computation significantly for many applications. SimGNN targets graphs from real-world graph databases, such as AIDS [NCI04], LINUX [Wan+12], and IMDB [YV15]. The target graphs are relatively small, with 10 nodes on average, but the database contains millions of graph pairs, creating a large number of graph-matching queries. Although the CPU implementation can finish each SimGNN query in milliseconds, processing millions of queries can take several hours; hence, it requires

customized acceleration. Such a workload of graph searching/mining is increasing in importance. For example, searching for antivirus chemical compounds is an important step in drug repurposing for COVID-19.

Despite the popularity and effectiveness of GNN approaches, there has been limited research on developing an accelerator for them (e.g., [Yan+20a; Gen+20; ZP20]). This is due to the inherent challenges posed by GNN in the design of accelerators, including:

- **Irregular memory access and low data reuse:** As opposed to images, the neighbors of a node in a graph may be stored in any location in memory. This will result in many irregular memory accesses to all levels of the memory hierarchy. Furthermore, GNNs have much lower data reuse compared to CNNs. As such, the countless CNN accelerators proposed in the literature (e.g., [Zha+18b; Wei+18; SFM17a; SWC20]) are incompatible here. Compared to traditional graph algorithms such as Breadth-First Search (BFS), the nodes have long feature vectors instead of a single scalar value. Therefore, not only is the access pattern different, but we can also exploit new kinds of parallelism and data reuse. As a result, most graph-based accelerators proposed in the literature (e.g., [Dai+17; Ham+16; WHN19; Yan+20b; Li+17; Muk+18; Bas+19; Dad+19; Wan+16; Als+16]) are ill-suited for GNNs.
- **Computation pattern disparity:** Different steps of the GCN deal with different sparsity rates (see Section 4.3). Besides, a GNN may include other types of computation patterns, such as the neural tensor network in SimGNN (see Section 4.4) to make an end-to-end application. Such variations call for a customized processing unit for each step.
- **Dynamic workload and parallelism:** Since the *number* of neighbors varies across different nodes, there will be a load imbalance between the graph’s nodes.

In addition to the challenges mentioned above, dealing with small graphs requires special design considerations as we will explain in Section 4.3. To solve these challenges, we present *StreamGCN* as an efficient and flexible GCN accelerator for streaming small graphs

Table 4.1: Properties of StreamGCN compared to the state-of-the-art GCN accelerators.

Work	Graph Size	Layer Customization	Sparse Engine for Feature Transformation	On-the-fly Sparsity Pruning	Read Each Element Only Once	Parallelization		
						Inter-layer	Feature-level (Sparse Part)	Node-level (Sparse Part)
HyGCN [Yan+20a]	Large	✗	✗	✗	✗	✗	✓	✓
GraphACT [ZP20]	Small	✗	✗	✗	✗	✗	✓	✗
BoostGCN [ZKP21]	Large	✗	✓	✗	✗	✗	✓	✓
AWB-GCN [Gen+20]	Large	✓	✓	✓	✗	✓	✗	✓
<b>Ours (StreamGCN)</b>	Small	✓	✓	✓	✓	✓	✓	✓

- from the different levels of memory and even through the network - and exploiting all the available sparsity. Then, we apply it to accelerate the entire pipeline of SimGNN as an end-to-end application. Since we are facing a *memory-bounded* application, we reduce the global memory transactions to the least amount. To deal with *the irregular memory access*, we utilize a scratchpad memory to store the matrices that need random access. Because of the *computation pattern disparity*, we analyze the requirements of all the steps of the computation pipeline and, accordingly, develop a dedicated architecture for each of them. We further propose an efficient workload distribution mechanism to alleviate the *load-imbalance* problem.

Concisely, we fuse all the stages together and employ a very deep pipeline with three different levels of nested customizable parallelization as listed in Table 4.1. We compared our approach to the previous GCN accelerators in detail in Section 2.4.2 which is summarized in Table 4.1. While we use SimGNN for illustrating our approach, the same optimizations can be applied to other GCN-based networks dealing with small graphs (e.g., [QBS20; Ish+21; Bai+20]) as well. We implement StreamGCN on three different FPGAs showing its flexibility and adaptivity to different platforms with different global memory bandwidth.

In summary, the key contributions of this chapter are:



- We design and develop StreamGCN, a flexible architecture for accelerating GCN specialized for streaming processing of small graphs and exploiting all the available sparsity.
- We adopt StreamGCN to accelerate SimGNN as an end-to-end application, resulting in an efficient architecture with a very deep pipeline and three levels of parallelization.
- We demonstrate the flexibility of our architecture by mapping and customizing it to three different FPGAs with different capacities and memory systems.
- Experimental results suggest that our accelerator can outperform multi-core CPU by  $18.2\times$  and GPU by  $26.9\times$ , demonstrating the efficiency of our design.

## 4.2 Background

### 4.2.1 Graph Convolutional Network Computation Details

We described a high-level view of the GCN computation in Section 2.3.1.1. More specifically, layer  $l$  of a GCN [KW16] takes an undirected graph  $G(V, E, H^l)$  as the input, where  $V$  ( $E$ ) denotes the nodes (edges) of the graph.  $H^l \in \mathbb{R}^{|V| \times f_l}$  is the matrix of the *input node embeddings* for this layer, with each row containing the embedding of one of the nodes where  $f_l$  indicates the number of features of each node at layer  $l$ . The core computation of a GCN layer to produce the *output node embeddings* is as follows:

$$\begin{aligned} \tilde{A} &= A + I_N, \quad \tilde{D}_{ii} = \sum_j \tilde{A}_{ij}, \quad A' = \tilde{D}^{-\frac{1}{2}} \cdot \tilde{A} \cdot \tilde{D}^{-\frac{1}{2}} \\ H^{l+1} &= \sigma_{act}(A' \cdot H^l \cdot W^l), \quad H^{l+1} \in \mathbb{R}^{|V| \times f_{l+1}} \end{aligned} \tag{4.1}$$

where  $\sigma_{act}(\cdot)$  is an activation function which typically is a ReLU and  $W^l$  is a layer-specific *trainable weight matrix*.  $A'$  is the *normalized adjacency matrix with added self-connections* that is calculated using  $A$  and  $I_N$  which are the *adjacency* and the *identity matrix*, respectively.  $\tilde{D}$  is a *diagonal matrix* where  $\tilde{D}_{ii}$  is the degree of node  $i$  plus one.

As Eq. 4.1 suggests, the first step in the computation ( $A' \cdot H^l$ ) gathers the neighbors' information for each of the nodes. As  $A'$  is a normalized matrix, the computation here is a weighted aggregation. After the Aggregation step, the node embeddings are transformed by applying a pretrained set of weights and finally passed through a ReLU unit. The time complexity of layer  $l$  can be seen to be  $O(|E|f_l f_{l+1})$ , where  $|E|$  denotes the number of edges including the self-connection ones [KW16].

### 4.3 StreamGCN Architecture

We can compute Eq. 4.1 either as  $(A' \times H^l) \times W^l$  or  $A' \times (H^l \times W^l)$ . We have chosen the latter since it results in a fewer number of operations. Intuitively, this is because both matrices  $A'$  and  $H^l$  are sparse, but their multiplication creates a dense matrix. As a result, in the former, we end up doing a dense-dense multiplication for the second multiplication. However, if we go with the latter, both multiplications are sparse-dense that as shown in AWB-GCN [Gen+20], reduces the number of operations. Figure 4.3 illustrates the high-level view of the GCN architecture in StreamGCN. In this section, we employ a bottom-up approach to highlight the optimization opportunities when GCN is applied to small graphs and how we used them to build the GCN accelerator as demonstrated in Figure 4.3.

#### 4.3.1 StreamGCN Design Principles

StreamGCN is designed:

- To exploit all the available sparsity.
- To reduce the number of times we access the global memory to the least amount possible. In our final architecture, each input element is read only once and there is no need to store any of the intermediate results in the global memory.
- To employ a deep pipeline with varying levels and degrees of parallelization for matching the workload of different stages and maximizing the overall performance.

- To efficiently handle and stream small graphs.

### 4.3.2 Baseline Architecture

In this section, we describe a baseline architecture to accelerate the processing of GCNs inspired by the previous GCN accelerators [ZP20; Yan+20a; Lia+20]. Although these optimizations are necessary, they are not enough when dealing with many small graphs. Hence, we propose to apply further optimizations in the subsequent sections.

#### 4.3.2.1 Feature Transformation (FT)

In this step, one must multiply matrices  $H^l \in \mathbb{R}^{|V| \times f_{in}}$  and  $W^l \in \mathbb{R}^{f_{in} \times f_{out}}$ , where  $f_{in}$  and  $f_{out}$  denote the number of input and output features, respectively. Here, adopting an inner-product-based matrix multiplication results in updating the same output feature in the consecutive iterations which introduces Read-After-Write (RAW) dependency between them. Since the floating-point addition cannot finish in one cycle, our pipeline cannot achieve an Initiation Interval (II) of one, meaning that we cannot schedule new operations in each clock cycle which degrades the efficiency of our design.

*Optimized Scheduling:* Read the Weight Matrix Row-wise; Stream the Embeddings Matrix Column-wise. To alleviate the RAW dependency problem, we perform Cartesian product as in [Par+17]. This means that we design a Processing Element (PE) consisting of SIMD Multiplication and Accumulation (MAC) units. At each cycle, we update different output locations by taking an element from  $H^l$  (read as a stream) and broadcasting that to parallel MAC units while each MAC unit reads different elements of the  $W^l$  matrix. To read each element only once and increase data reuse, for each fetched element of  $H^l$ , (i.e.,  $h_{00}$ ), we schedule all the operations it is involved with, before its eviction. We add a second level of parallelization by duplicating the SIMD PE by a *Duplication Factor (DF)* which parallelizes the node dimension. To avoid RAW dependencies between the PEs, we read  $H^l$  in column order. Note that if we read it rather in row order, we update the same location every  $\frac{f_{out}}{SIMD}$  iterations instead of every  $\frac{|V|}{DF} \times \frac{f_{out}}{SIMD}$  iterations. Reading in column order also

lets us cache and reuse the corresponding row of the weight matrix. Fig. 4.1 illustrates the final execution order of this step. The arrows denote the high-level ordering of traversing different dimensions, and the numbers show the elements that are accessed at their respective cycles. It is important to traverse the input feature dimension ( $f_{in}$ ) last (*arrow III*) since it is the dimension causing the dependencies.

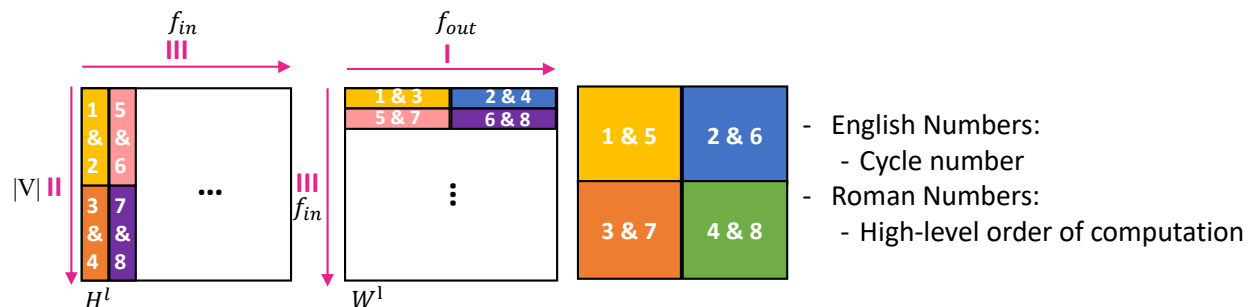


Figure 4.1: The optimized computation order to reduce II for the Feature Transformation step in GCN.

#### 4.3.2.2 Aggregation

In this step, we must multiply matrices  $A' \in \mathbb{R}^{|V| \times |V|}$  and  $X^l \in \mathbb{R}^{|V| \times f_{out}}$  where  $X^l$  is the result of the FT step. Due to the highly irregular access to the matrix  $X^l$  to aggregate features of the neighbors, we cache it in a scratchpad memory. Matrix  $A'$ , is often ultra sparse [Gen+20]. To reduce the number of both transferred elements and operations, we prune this matrix and only pass its non-zero elements, which represent edges, to the FPGA. Instead of dedicating an on-chip memory for storing the edges, we read them as a stream and update all the features of the destination node, before retiring the edge. It helps us with freeing up the storage for caching  $X^l$  which is the same matrix that needs to be cached for the FT step. We further rearrange the edges, as a step of preprocessing, before sending them to the FPGA, so that the ones with the same destination node are at least  $L$  (the latency of the functional unit causing the dependency) locations apart to make sure there is not more than one update to the same node within the window of  $L$  cycles. As edge-level parallelism can result in bank conflicts since they update random nodes, we only make use

of feature-level parallelism to distribute the workload here. Nevertheless, one can include that by adding another level of preprocessing by further reordering the edges.

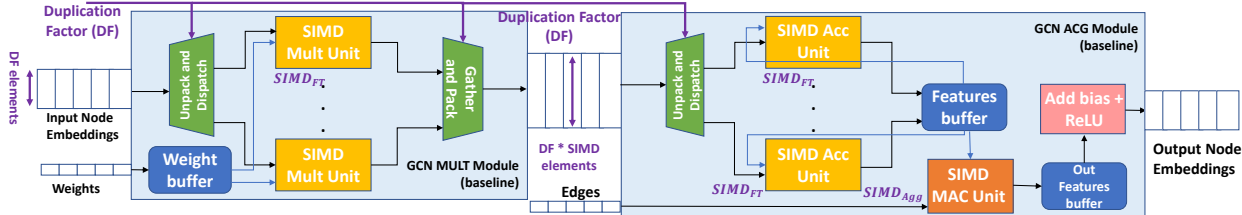


Figure 4.2: Baseline architecture for accelerating GCN: intra-layer pipelining between MULT module (multiplication unit for Feature Transformation step) and ACG Module (accumulation unit for Feature Transformation step + Aggregation step)

### 4.3.2.3 Intra-layer Pipelining

To further boost the performance, we add intra-layer pipelining by connecting the modules as a dataflow architecture. As a result, the overall latency will be close to the latency of the slowest module. In addition, we can avoid off-chip memory accesses in between these modules. The MULT module, depicted in Fig. 4.2, is responsible for doing all the multiplications of the FT step. It has a local buffer to store the weights and streams the elements of  $H^l$  from input FIFO. Each entry of this FIFO is a concatenation of DF elements. Once the multiplication results are ready, they are packed and sent in a FIFO to the ACG module (Fig. 4.2). In this module, we merge the ACC unit of the FT step and the Aggregation step to save memory resources since they share the matrix  $X^l$ . After fetching the output of the MULT module, the ACG module unpacks the data based on the same DF and dispatches SIMD elements to each SIMD ACC Unit with the same SIMD factor. Once the additions are done, it will store the partial results in the local buffer *features buffer*. After all the updates are committed to the *features buffer*, the matrix  $X^l$  is computed and the Aggregation step can start. The SIMD factor of this step is higher than the one in the FT step since we only exploit feature-level parallelization here. After this step is finished, the elements of the *out features buffer* are added with a bias, passed through a ReLU unit ( $\max(0, \cdot)$ ), and stored

in the global memory. Note that in the baseline architecture, we reuse the same modules for all the GCN layers.

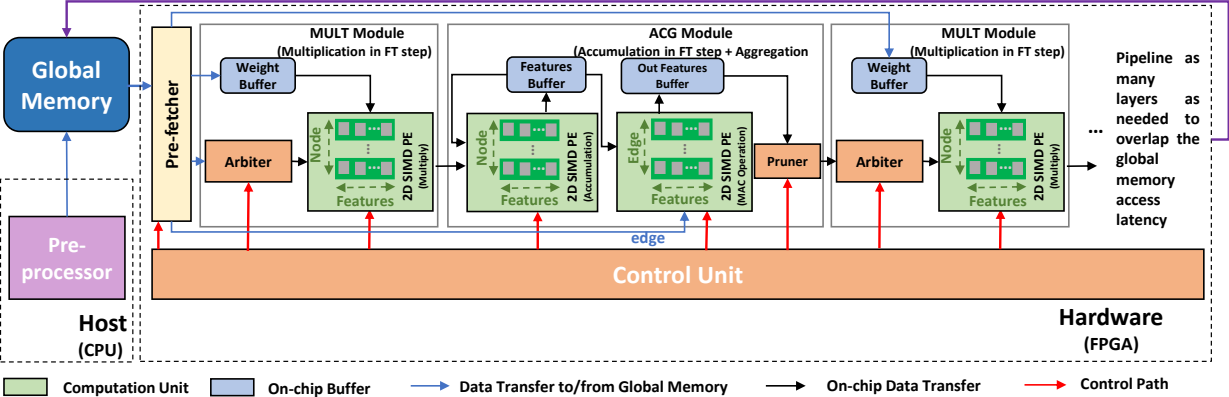


Figure 4.3: High-level overview of GCN accelerator architecture in StreamGCN.

### 4.3.3 Extension 1: Multi-layer Support and Inter-layer Pipelining

As it is commonly practiced ([Yan+20a; ZP20; Lia+20]), in the baseline architecture, we only exploit intra-layer pipelining and reuse the modules for all the GCN layers. However, this is not sufficient when we are dealing with small graphs. The off-chip communication is a serious burden for this application since it deals with small-sized inputs. To alleviate this problem, we intend to reduce the number of accesses to the off-chip (global) memory as much as possible. The baseline architecture is inefficient in this regard since, at the end of each layer, the output should be stored in the global memory and read back again for the next layer. To avoid these redundant accesses, we extend the dataflow architecture described in Section 4.3.2.3 to all the layers of GCN. To realize this, we instantiate new modules for each layer and connect them with FIFOs as depicted in Fig. 4.3. Fusing the computation for all the layers by enabling dataflow architecture has several benefits such as: 1) we can avoid writing the intermediate results to the global memory by forwarding them to the next layer through FIFOs. 2) The operations will be dynamically scheduled since each module can perform its operation whenever it has data available. 3) Since we are instantiating different modules for each layer, we can customize the parallelization factors of each module based

on the workload of their respective GCN layer. 4) As the adjacency matrix of a graph does not change across different layers, we can read the edges from the global memory only once for the first layer and reuse them for the subsequent ones by transferring them through the on-chip FIFOs.

#### 4.3.4 Extension 2: In Situ Sparsity Support in FT Step

The input node embedding to the first layer of GCN usually contains many zero elements since it often uses one-hot encoding for assigning initial vectors to the nodes. Furthermore, since there is a ReLU unit at the end of each GCN layer, the matrix generated by each layer, which is the input to the next layer, is sparse. In fact, we saw 52% and 47% sparsity on average for the input to the second and the third layers of GCN in SimGNN for randomly drawn graphs from our target dataset. Therefore, the FT step also needs to have support for sparse computation. To reduce the number of operations, we prune the zero elements and only pass the non-zero ones to the next layer. As a result, the updates to the output buffer may come in random cycles; thus, it is necessary to store the buffer containing the partial results on-chip to enable random access. For the same reason, we pack the node features with their address which includes their row and column ID. Packing the elements with their address helps to make the dispatch unit simpler since each SIMD PE is free to work with any data and knows which partial result should be updated; hence, there is no need to take special considerations to navigate the data to the correct PE. We only need to make sure that at all times each SIMD PE is working with a different memory bank. We employ an arbiter for this matter, as explained below.

As mentioned in Section 4.3.2, to reduce the number of RAW dependencies, we chose to stream the node embedding matrix and broadcast it to different Computation Units (CU) which read the weight matrix as a batch. Since the node embedding is a sparse matrix, reading it as a stream facilitates the pruning mechanism we employ and enables us to distribute the workload more efficiently. Fig. 4.4 demonstrates a toy example illustrating this. The colored squares show the non-zero elements of the node embedding matrix. By

mapping the weights, which are non-zero, to the SIMD dimension, all the CUs in the PE would execute useful operations and we can skip all the operations involving a zero node embedding.

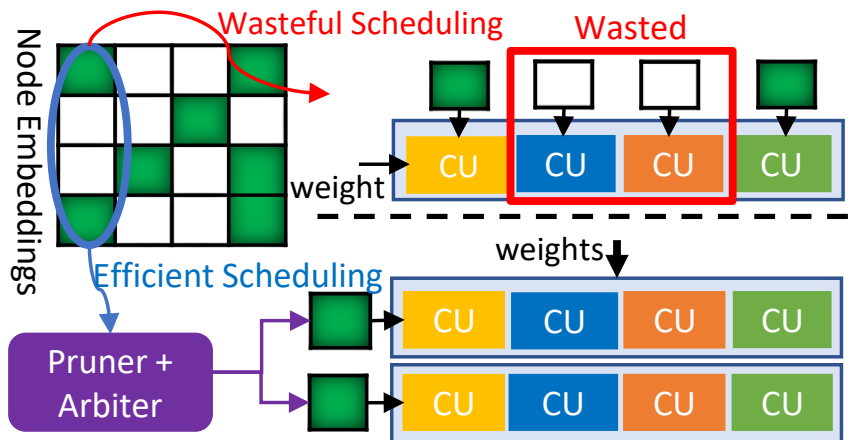


Figure 4.4: The benefit of streaming the node embeddings and mapping the weights to the SIMD dimension when processing the Feature Transformation step of a GCN layer. CU stands for computation unit. Colored cells in the matrix show non-zero values and white cells show zero values.

When skipping the zero node embeddings, the dependency distance for output elements may change dynamically since the number of non-zero inputs between the updates to the same location can be different. Even though the scheduling discussed in Section 4.3.2 increases the dependency distance as much as possible by doing all the operations when a non-zero input is encountered (each non-zero element would fill  $\frac{f_{out}}{SIMD_{FT}}$  cycles of the dependency window), there still may be some cases where the dependency distance is less than  $L$  after this optimization. Instead of setting the II to  $L$  to ensure correctness, we first insert  $L$  registers to store the partial results of CU at the end of each of its pipeline stages; hence, we can schedule a new set of operations at each clock cycle (II=1). There may be cases where the new scheduled operations want to update a location whose old value is still in the registers and have not updated the buffer. To ensure the correctness, we added a control unit that keeps track of the last cycle that each of the output locations was updated. If the number of cycles between two updates to the same location is less than  $L$ , the control unit



will insert bubbles into the pipeline until the previous update is committed.

We insert a unit for pruning zeros at the end of the *ACG* module. As Fig. 4.5 demonstrates, at each cycle, we evaluate  $P$  elements of the node embeddings and pass each to a FIFO if it is not zero. The *MULT* module of the next layer takes the  $P$  FIFOs as the input and uses an arbiter to fetch, at most,  $DF$  of them ( $DF \leq P$ ) for passing to  $DF$  SIMD PEs. An arbiter keeps track of the FIFO whose turn it is to be read first in the next cycle. It then uses a round-robin ordering for dispatching the elements from the non-empty FIFOs. After dispatching the inputs, it checks for the RAW dependency by scanning the *prev iter* buffer which contains the last cycle when each element was seen as the input. If the distance is less than  $L$ , it will insert bubbles in the pipeline until the previous input has committed its update. If there is no dependency, for each memory bank at most one element from the dispatched inputs will be issued to a SIMD PE and the current cycle number will be stored in *prev iter* buffer for that input.

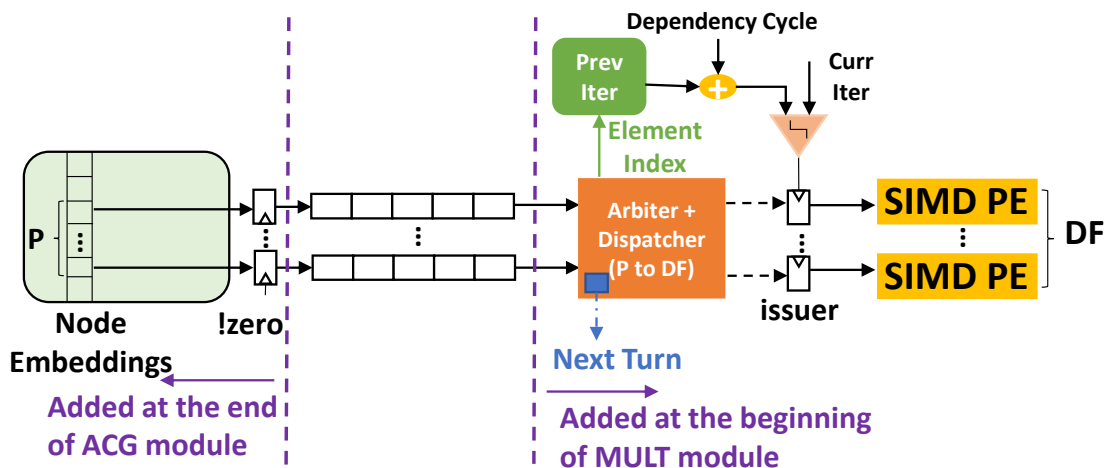


Figure 4.5: StreamGCN architecture support for sparse computation in the Feature Transformation step of GCN.

The StreamGCN architecture provides flexibility in choosing the parallelization factors. Table 4.2 lists the parameters that can be tuned for each GCN layer based on its workload. The SIMD factors correspond to feature-level parallelization, while  $DF$  and  $P$  map to node dimension.

Table 4.2: Summary of architecture parameters for the accelerator of each GCN layer in StreamGCN.

Design Parameter	Explanation
$SIMD_{FT}$	SIMD factor of the FT step
$SIMD_{Agg}$	SIMD factor of the Aggregation step
DF	Duplication factor of the PEs in FT step
P	Number of input FIFOs to the arbiter

FT: Feature Transformation

## 4.4 StreamGCN Application to Graph Matching

In Section 4.3, we proposed an architecture for GCN specialized for small graphs. In this section, we extend our architecture to accelerate an end-to-end application, SimGNN, which introduces new computation patterns beyond GCN.

### 4.4.1 SimGNN

Bai et al. [Bai+19] proposed a neural-network-based approach to assign a similarity score to two graphs. Its computation pipeline consists of four major stages. The first stage has three layers of GCN to extract the *node embeddings*  $H \in \mathbb{R}^{|V| \times F}$  where  $F$  is the number of features of the last layer. In the second stage, it uses a *Global Context-Aware Attention layer (Att)* to combine the node embeddings and generate a single embedding per graph  $h_G \in \mathbb{R}^F$ . For this matter, it adapts an attention mechanism to find out the importance of each of the nodes. The graph embedding, then, can be calculated by taking a weighted sum of the node embeddings using the attention weights. The following formula summarizes the computation in this stage:

$$h_G = \sum_{n=1}^{|V|} \sigma(h_n^T \cdot \tanh(\frac{1}{|V|} W_{Att} \sum_{n=1}^{|V|} h_n)) \cdot h_n \quad (4.2)$$

where  $\sigma(\cdot)$  denotes the sigmoid function to produce the attention weights and  $W_{Att} \in \mathbb{R}^{F \times F}$  is a learnable *weight*. The time complexity of this stage can be seen to be  $O(|V|F)$ . The

third stage is a *Neural Tensor Network (NTN)* that calculates a vector of similarity scores between the two graphs:

$$s(h_{G_1}, h_{G_2}) = \sigma(h_{G_1}^T W_{NTN}^{[1:K]} h_{G_2} + V \cdot \text{concat}(h_{G_1}, h_{G_2}) + b) \quad (4.3)$$

where  $W_{NTN}^{[1:K]} \in \mathbb{R}^{F \times F \times K}$ ,  $V \in \mathbb{R}^{K \times 2F}$ , and  $b \in \mathbb{R}^K$  are learnable *weight tensor*, *weight matrix*, and *bias vector*, respectively.  $K$  is a hyper-parameter that controls the number of similarity scores. The time complexity of this stage is  $O(F^2K)$ . The last stage uses a Fully Connected Network (FCN) to gradually reduce the similarity vector to only one score.

The non-GCN stages make use of *exp* and *tanh* functions which are expensive to have on FPGA that can limit their parallelism rate. Furthermore, the computation complexity of the different stages shows that the GCN step is the most computation-intensive one; hence, when pipelining all the stages together, the accelerator will be bottlenecked by the GCN step. Therefore, we do not aggressively parallelize the rest of the steps and rather focus on reducing their resource utilization.

#### 4.4.2 Att Architecture

The SimGNN pipeline applies the GCN stage to two graphs for each comparison query. Instead of duplicating the architecture in Fig. 4.3, we process the graphs serially and reuse the GCN module for the two input graphs in the query. Reusing the GCN module enables us to map the design to small FPGAs as well. We improve the performance of processing one query by overlapping the *GCN* computation of one graph with the *Att* computation of the other one. Thus, the total performance will be bottlenecked with the performance of GCN and we can focus on reducing the area and reusing the resources for *Att*. In computing  $v = W_{Att} \sum_{n=1}^{|V|} h_n$ , we first must add  $h_n$  vectors and then do a Matrix-Vector Multiplication (MVM). Instead of instantiating separate adders for the first additions and the ones in MVM, we rewrite the equation as follows to reuse the adders:

$$v = W_{Att} \sum_{n=1}^{|V|} h_n = \sum_{n=1}^{|V|} W_{Att} h_n = \text{sum}(H \cdot W_{Att}^T, 2) \quad (4.4)$$

where  $H \cdot W_{Att}^T$  shows the matrix multiplication between the weight matrix and the matrix of node embeddings,  $sum(H \cdot W_{Att}^T, 2)$  denotes the reduction of the resulting matrix across its second dimension (columns), meaning that all the multiplications associated with a column of  $H$  should be added together. Fig. 4.6 demonstrates an overview of the *Att* module. As in the GCN stage, we divide the matrix multiplication to two different modules, one responsible for multiplications and the other for additions. Again, we use SIMD PEs to implement these modules. However, the SIMD factor here can be set to a different value compared to the GCN stage since they have different computation complexities. The *Repack* module is responsible for adjusting the output of GCN with the SIMD factor of this stage. For *tanh* and *exp* functions, we adopt their implementation in the *AMD Xilinx HLS Math* library. Note that the last summation in Eq. 4.2 can be seen as  $H^T \cdot a$  where  $a \in \mathbb{R}^{|V|}$  contains the sigmoid results. Hence, we use an MVM unit at the end.

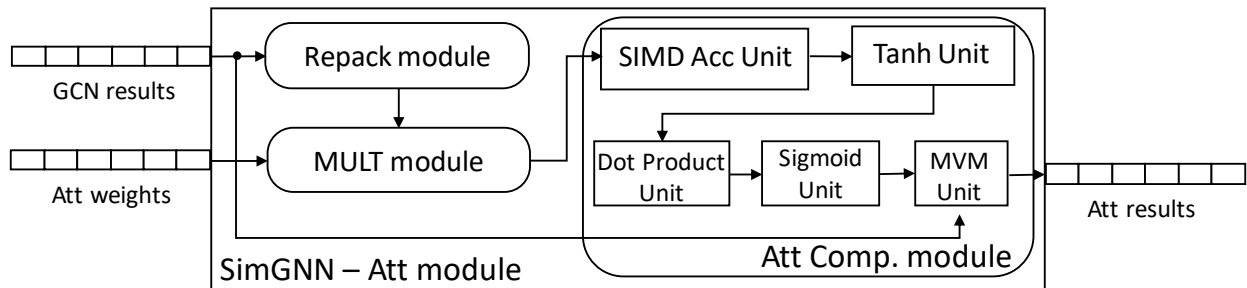


Figure 4.6: Architecture overview of the second stage of SimGNN in StreamGCN: Att module implementing the global context-aware attention layer.

#### 4.4.3 NTN + Fully-connected Network Architecture

The computation in the NTN stage is rather simple since it is a series of fixed-size MVMs followed by a bias addition and an activation function. Furthermore, the layers of the FCN in the last stage either need an MVM unit or a reduction tree to lower a vector to a scalar. Like the previous stages, we implement all the sub-modules of these two stages in a dataflow manner. Fig. 4.7 depicts the architecture of these two steps.

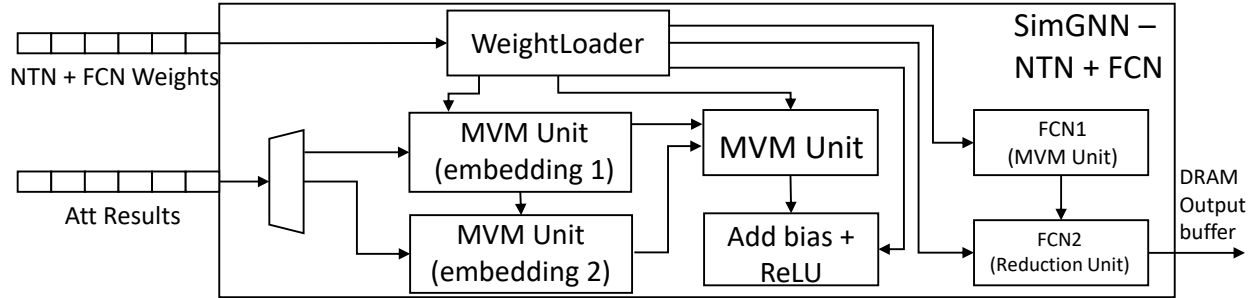


Figure 4.7: Architecture overview of the last two stages of SimGNN in StreamGCN: NTN and FCN module implementing the neural tensor network and fully connected network.

#### 4.4.4 Putting It All Together

The whole computation pipeline of SimGNN is implemented as a three-level dataflow architecture. The first two levels resemble an inter-stage pipelining while the last one is for intra-stage pipelining. The first level enables a task-level parallelization by grouping the graph-related steps, the *GCN* (Section 4.3) and *Att* (Section 4.4.2) modules, and overlapping them with the rest, *NTN\_FCN module* (Section 4.4.3). The second level of dataflow architecture overlaps the *GCN* stage with the *Att*. Finally, the last level applies dataflow architecture to each of the *GCN*, *Att*, and *NTN\_FCN* modules as shown in Fig. 4.3, 4.6, and 4.7, respectively. We apply three optimizations for reducing the off-chip communication latency: 1) each input buffer can be mapped to a different DRAM bank or HBM channel to enable parallel access to them. 2) The available global memory bandwidth is fully utilized by applying memory coalescing. Memory burst is also applied to amortize the initialization overhead, 3) The computation modules overlap with the modules accessing global memory by implementing the accelerator as a dataflow architecture.

## 4.5 Experimental Results

### 4.5.1 Benchmark

We consider a real-life graph dataset, AIDS [NCI04], for benchmarking our design. AIDS contains 42,687 antivirus chemical compounds gathered by the Developmental Therapeutics Program at NCI/NIH. The graphs in AIDS have 25.6 (27.6) nodes (edges) on average. We randomly form 10,000 queries of them for testing. The kernel time and end-to-end (E2E) time reported in this section are the average of all queries.

### 4.5.2 Experimental Setup

The StreamGCN architecture is described using Vivado HLS C++ [Xila]. The design is synthesized and implemented using AMD Xilinx Vitis 2019.2 on three different target platforms: AMD Xilinx Alveo U50, AMD Xilinx Alveo U280, and AMD Xilinx Kintex UltraScale+ KU15P. The first two are equipped with HBM2 and, ideally, can achieve a bandwidth of 316 GB/s (460 GB/s) with a TDP of 75W (225W); while the last one utilizes DDR4 as the global memory. Table 4.3 compares the hardware resources of these boards. For comparison to CPU and GPU, the PyTorch-based implementation of SimGNN from [Roz18] is used that is built using the state-of-the-art PyTorch Geometric (PyG) library [FL19] which is commonly used as a baseline by previous works [Yan+20a; Gen+20; Lia+20]. For the Aggregation step, PyG exploits sparsity and edge-level parallelism by adapting the PyTorch Scatter library [Fey24]. For the Feature Transformation step, it uses Intel MKL [Int24b] and NVIDIA cuBLAS library [NVI24] for CPU and GPU respectively, making it a reasonable and optimized baseline. The target CPU in our experiments is Intel(R) Xeon(R) CPU E5-2699 v4 running at 2.2 GHz. For testing on GPU, we use an AWS p3.2xlarge instance which has an NVIDIA V100 GPU.

Table 4.3: Properties of the FPGAs used to implement StreamGCN.

Platform	BRAM (Mb)	LUT (K)	FF (K)	DSP	URAM (Mb)	Max BW (GB/s)
AMD Xilinx Kintex UltraScale+ KU15P	34.6	523	1045	1968	36	19.2
AMD Xilinx Alveo U50	47.3	872	1743	5952	180	316
AMD Xilinx Alveo U280	70.9	1304	2607	9024	270	420

### 4.5.3 Impact of GCN Architecture Optimizations

#### 4.5.3.1 Inter-Layer Pipelining

Table 4.4: Impact of the GCN architecture optimizations tested on AMD Xilinx Alveo U280 board. The meaning of design parameters is summarized in Table 4.2. Baseline design shows a single set of design parameters because it uses the same hardware for all the layers.

Architecture	Design Parameters (L1 / L2 / L3)				LUT / FF / DSP / BRAM / URAM (%)	Freq. (MHz)	Kernel (ms)	Kernel $\times$ DSP
	$SIMD_{FT}$	$SIMD_{Agg}$	DF	P				
Baseline	16	32	8	-	9.8 / 7.7 / 7.4 / 6.8 / 0	265	0.599 (1 $\times$ )	4.46 (1 $\times$ )
+Inter-Layer Pipeline	32/16/16	32/32/16	8/8/8	-	14 / 12 / 18 / 3.6 / 2.5	271	0.383 (1.56 $\times$ )	6.74 (0.66 $\times$ )
+Extended Sparsity	32/32/16	32/32/16	2/1/1	8/2/2	4.8 / 6.0 / 4.4 / 4.8 / 3.1	300	<b>0.264 (2.27<math>\times</math>)</b>	<b>1.15 (3.88<math>\times</math>)</b>

Table 4.4 shows the resource usage and performance of the StreamGCN architecture when accelerating three GCN layers of SimGNN on the U280 FPGA. The baseline uses the same hardware for all the GCN layers. With inter-layer pipelining added, all 3 GCN layers run in parallel as a coarse-grained pipeline. Since each layer utilizes different pieces of hardware, we can customize the design parameters to match the throughput of each layer. As a result, the 3 layers require 2.4 $\times$  more DSPs compared with the baseline. We distribute the storage units needed between BRAM, URAM, and LUT to obtain a better frequency. The GCN kernel time is reduced by 36% with inter-layer pipelining added to the baseline. However, if we look at the latency-area product metric, i.e., Kernel $\times$ DSP, we can see that the performance improvement does not catch up with the computation units (DSP) increment, suggesting potential for further optimizations.

### 4.5.3.2 In Situ Sparsity for the Feature Transformation Step

Although using  $P$  queues (Section 4.3.4) help the arbiter fetch non-zero elements more frequently, it may still not be enough to dispatch data to all the  $DF$  PEs. Furthermore, by increasing the  $DF$ , we may need to insert more bubbles in the pipeline to avoid RAW dependency since it reduces the number of cycles between the updates to the same location. As a result, there is a trade-off in choosing the right  $DF$  for each layer. The best parallelization factors are summarized in Table 4.4. When  $DF$  is set to 1, we no longer need to have separate banks in the row dimension of the buffers which can lessen the number of needed memory blocks. This makes it more efficient to use dense memory blocks (BRAM and URAM) as opposed to LUTs for the buffers. As Table 4.4 shows, extending sparsity to feature transformation over the inter-layer pipeline has further reduced the kernel time by 31%, while decreasing the DSP usage by  $4.09\times$ . The results clearly suggest that, since this is a memory-bounded application, throwing more resources into the architecture is not helpful. Instead, the memory access latency should be reduced and the computation units shall be used more efficiently. Since a large number of zero elements and required DSPs are excluded, there is a  $2.27\times$  speedup over the baseline and the latency-area metric ( $\text{Kernel}\times\text{DSP}$ ) is greatly improved by  $3.88\times$ .

### 4.5.4 End-to-end Acceleration of SimGNN

#### 4.5.4.1 Flexibility of Mapping to Different FPGAs

We implement the whole pipeline of SimGNN on 2 HBM FPGAs and KU15P that uses DDR memory. Fig. 4.8 compares the resource breakdown of the modules at the top hierarchy of our design when mapped to U280. We allocate most of the resources to the GCN stage as it is the computation-intensive part of the network. Table 4.5 shows the resource usage and performance for the three FPGA platforms. We can see that the kernel runs faster on HBM FPGAs compared to KU15P. This is mainly because HBM FPGAs can achieve a better frequency as they have more resources and the Vitis tool has more freedom in placement



and routing (PnR) of the design to optimize the timing. In fact, the cycle counts of the same kernel running on a common FPGA but using different types and numbers of banks for global memory are almost the same. This suggests that after our optimizations the bottleneck is no longer at the memory level.

Table 4.5: Performance and resource utilization of a StreamGCN design accelerating SimGNN on different target FPGAs.

FPGA	LUT / FF / DSP / BRAM / URAM (%)	Freq. (MHz)	Kernel (ms)	E2E (ms)	E2E (query/s)
KU15P	34 / 29 / 35 / 30 / 23	201	0.786	1.135	881
U50	17 / 16 / 12 / 16 / 4.7	279	0.423	0.538	1858
U280	11 / 10 / 7.7 / 10 / 3.1	290	0.327	0.509	1965

E2E: End-to-End

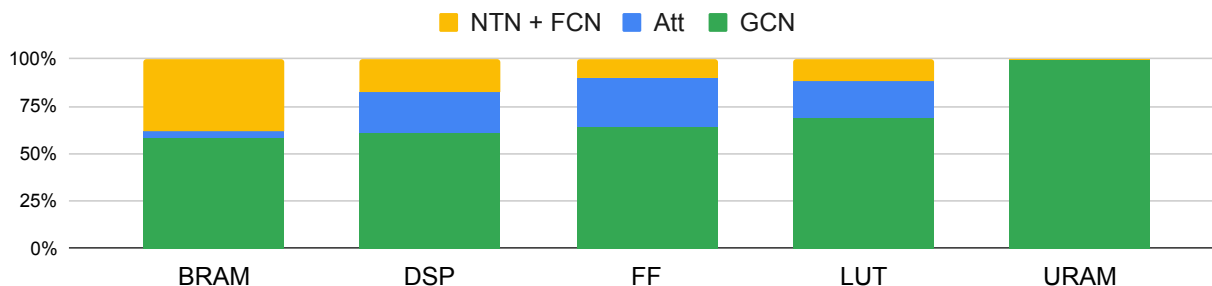


Figure 4.8: Resource breakdown of SimGNN accelerator on AMD Xilinx Alveo U280 board.

#### 4.5.4.2 StreamGCN vs CPU and GPU

We test the performance of the whole pipeline of SimGNN on the CPU and GPU described in Section 4.5.2. In this section, we assume that the inputs are already stored in the host memory, and we want to offload the graph comparison queries to either of the target platforms. The goal is to compare the performance of these platforms for processing a graph-matching query. Table 4.6 summarizes the average runtime per query. The queries are started sequentially, and the end-to-end time of all the platforms is the time interval between two

consecutive queries being started. This includes the runtime for any preprocessing steps as well. For FPGA and GPU, it also involves the host-kernel communication via the PCIe link, writing data to FPGA/GPU’s global memory, reading the results from that, and the overheads for using the APIs (OpenCL for FPGA and PyTorch for CPU/GPU). We use the end-to-end time for comparison since these overheads are inevitable and should be accounted for. The kernel time on CPU/GPU is measured with the PyTorch profiler.

Table 4.6: Performance comparison of running SimGNN on different hardware platforms.

Platform	Max BW (GB/s)	Kernel (ms)	E2E (ms)	Speedup (vs. CPU)	Speedup (vs. GPU)
KU15P (StreamGCN)	19.2	0.786	1.135	8.2	12.1
U50 (StreamGCN)	316	0.423	0.538	17.2	25.5
U280 (StreamGCN)	460	<b>0.327</b>	<b>0.509</b>	<b>18.2</b>	<b>26.9</b>
PyG-CPU	76.8	5.85	9.27	1	1.5
PyG-GPU	900	9.68	13.7	0.68	1

BW: Bandwidth, E2E: End-to-End

The results demonstrate that our FPGA solution can outperform both CPU and GPU significantly. As discussed in Section 4.1, this is partly because of the dynamic load balance and the irregular memory access of the graph structure. Furthermore, since we target small graphs, it results in extreme under-utilization of GPU. In fact, the profiling results indicate that the GPU utilization does not go higher than 6% and, for the most part, the PyG-GPU only uses 1 Streaming Multiprocessor (SM) since the matrices are small. Because of this and the fact that the GPU runs at a lower frequency (1.3GHz) compared to the CPU (2.2 GHz), the GPU version of this application is even slower than the CPU. The *nvprof* profiling results show that PyG-GPU runs 225 kernels for accelerating this application that on average have 4.6 *KFLOPs*. With this low computation intensity, the overhead of running the kernel (such as *cudaLaunchKernel*) is larger than the actual kernel runtime which greatly impacts the GPU performance. Designing the GPU kernel manually can alleviate some of these shortcomings, but the underlying problem still exists due to the coarse-grained execution model of GPU. In contrast, our FPGA solution suffers from kernel initialization overheads

only once since we develop a deep pipeline across all stages of the computation by fusing them in one kernel. This pipelining has several other benefits as explained in Section 4.3.3. Note that both FPGA and GPU have enough resources left for batch processing, so it is meaningful to compare their single query execution.

#### 4.5.4.3 Discussion on Scalability

Table 4.5 illustrates that the available resources allow us to instantiate 6 StreamGCN architectures with U280 before hitting the 80% resource usage upper-bound. 80% is an empirical threshold that beyond that the AMD Xilinx tool would have a hard time mapping the design to the FPGA. Since U280 is equipped with HBM which makes use of pseudo channels that can be accessed *independently*, this batch processing can be done completely in parallel. This does not change the latency of each graph query, but it would increase the throughput by  $6\times$ . In addition, although the graphs in our target benchmark have 25.6 nodes on average and we designed our accelerator for them, we can use the unused resources for increasing the target graph size or processing more GCN layers. Obviously, increasing either the batch number, graph size, or number of GCN layers limits the other values. If all the other options are fixed, we can increase these three parameters by 6, 150, and 20, respectively, when targeting the U280 board for SimGNN.

## 4.6 Other Related Works

In Section 2.3.1.1, we reviewed prior studies presenting GCN accelerators and compared the key components across representative works, as outlined in Table 4.1. In this section, we delve into additional research that emphasizes harnessing the sparsity of the application.

**SpMM and SCNN Accelerators:** Apart from the works focusing on GCN, there has been a lot of research on sparse MM either for pruned CNNs or normal MM [Sri+20a; Han+16; KMZ19; Par+17; Sri+20b]. They all rely on the fact that the sparse matrix is known offline, and they can preprocess it. For example, EIE [Han+16] proposes a sparse

matrix-vector multiplier for the fully-connected layers. It reorganizes the sparse matrix in Compressed Sparse Column (CSC) format and preloads that into on-chip memory. As another example, Kung et al. [KMZ19] pre-process the data by merging multiple sparse columns of the weight matrix into one and pruning all the weights except for the most significant ones, resulting in some accuracy loss. These approaches are not feasible for GCN in which the sparse matrix (i.e., the node embeddings) is generated *while running* the algorithm; whereas we proposed a technique to prune the zeros *on-the-fly*.

## 4.7 Conclusion

In this chapter, we analyzed and examined the optimization opportunities when GCN is applied to small graphs. We presented an efficient architecture, StreamGCN, and developed an accelerator for SimGNN based on that as an end-to-end application, which demonstrated significant speedup over the CPU and GPU results. StreamGCN is ideal for real-time or near real-time graph search and similarity computation for many biological, chemical, or pharmaceutical applications.

Part II

# Machine Learning for Designing Customized Accelerators

In the previous part, we focused on creating specialized accelerators for specific applications. While efficient, this approach is not scalable. Here, we turn our attention to automating microarchitecture optimization for general applications as summarized in Fig. P.2. Chapter 5 introduces AutoDSE, a model-free optimizer that uses a bottleneck-guided heuristic to identify high-quality designs. In Chapter 6, we present GNN-DSE, which uses a surrogate for the HLS tool to evaluate design candidates quickly in milliseconds. In Chapter 7, we create the HLSyn database to provide the large training data required. Model-based techniques can speed up optimization but may lose accuracy in estimating design quality. To mitigate this, Chapter 8 introduces HARP for more robust representation learning.

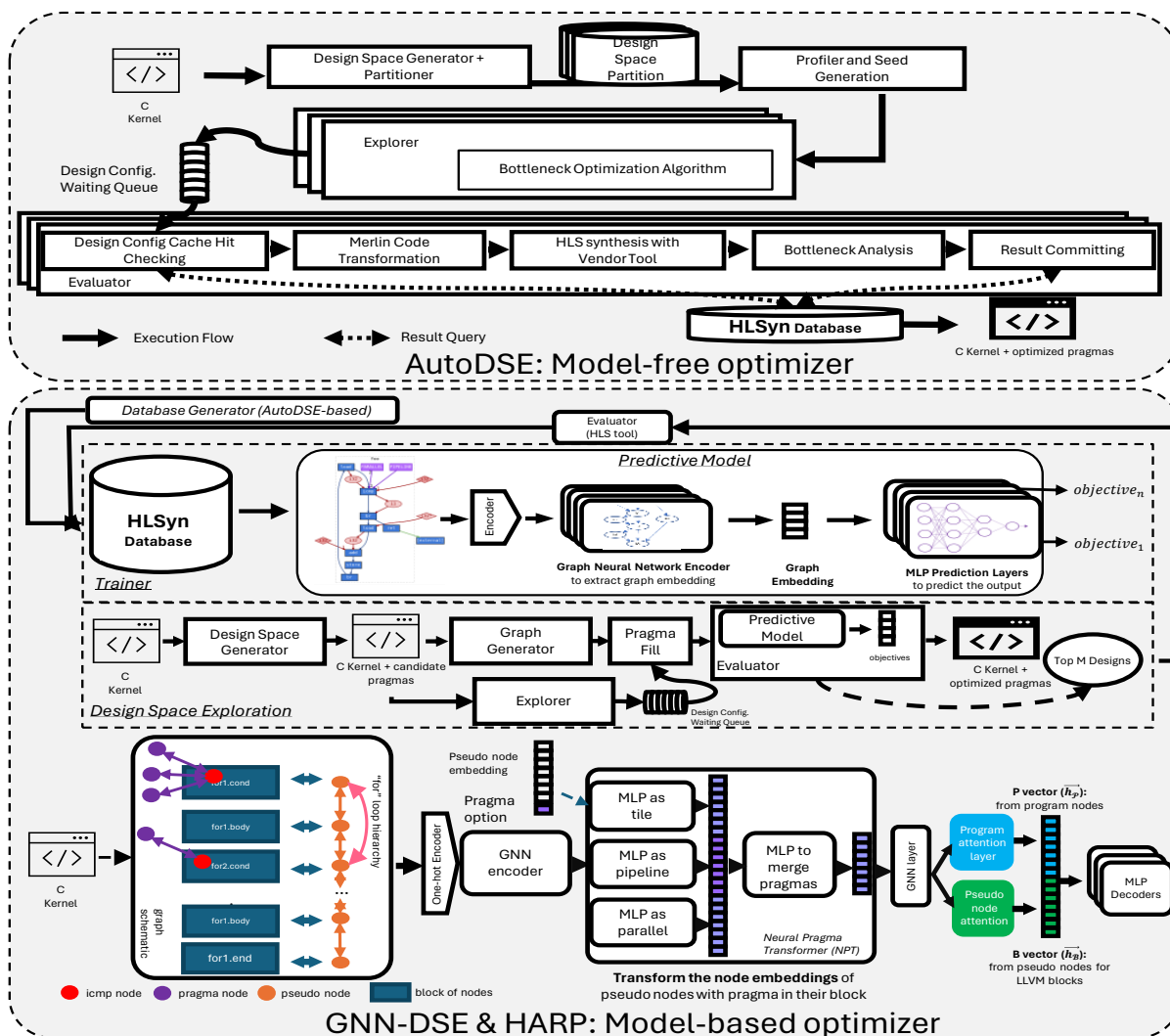


Figure P.2: Overview of Part 2 – General Microarchitecture Optimization.

## CHAPTER 5

# AutoDSE: Enabling Software Programmers to Design Efficient FPGA Accelerators

In the first part of the dissertation, we designed architecture templates for accelerating two well-known deep learning algorithms, CNN and GCN. While it is a promising approach for accelerating popular programs, we cannot afford to do it for every program. The complexity involved in programming FPGAs creates a serious burden for general software programmers in adopting them in their applications. Even with the help of HLS, accelerator designers still have to manually perform code reconstruction and cumbersome parameter tuning to achieve optimal performance. To address this problem, we propose an automated design space exploration framework named *AutoDSE* that leverages a bottleneck-guided coordinate optimizer to systematically find a better design point. In other words, AutoDSE detects the bottleneck of the design in each step of the optimization and focuses on high-impact parameters to overcome it. The experimental results show that AutoDSE can identify the design point that achieves, on the geometric mean,  $19.9\times$  speedup over one CPU core for MachSuite and Rodinia benchmarks. Compared to the manually optimized HLS vision kernels in the Xilinx Vitis libraries, AutoDSE can reduce their optimization pragmas by  $26.38\times$  while achieving similar performance. With less than one optimization pragma per design on average, we are making progress towards democratizing customizable computing by enabling software programmers to design efficient FPGA accelerators<sup>1</sup>.

---

<sup>1</sup>AutoDSE codes are available at <https://github.com/UCLA-VAST/AutoDSE>

## 5.1 Introduction

As we explained in chapters 1 and 2, HLS [Con+11; Zha+08] can help improve the programmability of the FPGAs by raising the abstraction level. Code 5.1 shows an intuitive HLS C implementation of one forward path of a CNN. AMD Xilinx Vitis generates about 5800 lines of RTL code from Code 5.1 with the same functionality. As a result, it is much more convenient and productive for designers to evaluate and improve their designs in HLS C/C++.

While HLS is well-suited for hardware experts to quickly implement an optimal design, it poses challenges for most general software designers with limited FPGA expertise. The hardware architecture inferred from a syntactic C implementation might be ambiguous, leading current commercial HLS tools to typically generate architecture structures based on specific HLS C/C++ code patterns. Consequently, not every C program results in an optimal microarchitecture, and designers must manually reconstruct the HLS C/C++ kernel with specific code patterns and hardware-specific pragmas to achieve high performance. In fact, the generated FPGA accelerator from Code 5.1 is  $80\times$  slower than a single-thread CPU. However, the optimized code (shown in Code A.1 in Appendix A.1) can achieve more than  $7,000\times$  speedup ( $88\times$  speedup over single-threaded CPU) after we analyze and resolve several performance bottlenecks listed in Table 5.1 by applying code transformations and inserting 28 pragmas.

It turns out that the bottlenecks listed in Table 5.1 occur for most C/C++ programs developed by software programmers, and similar optimizations have to be repeated for each new application, which makes HLS C/C++ design not scalable. In general, there are three levels of optimization that one needs to employ to get to a high-performance FPGA design. The first level is for increasing the data reuse or reducing/removing the data dependency by *loop* transformations, which is common in CPU performance optimizations as well (e.g., for cache locality); therefore, it is well accepted by software programmers and we expect them to apply such transformations manually without any problems. The second level



is required to enable repetitive architectural optimizations that most of the designs benefit from, such as memory burst and memory coalescing, as mentioned in reasons 1-2 in Table 5.1. Fortunately, the recently developed Merlin Compiler <sup>2</sup> [Con+16b; Con+16a; Fal] (reviewed in Section 2.1.1) from Falcon Computing Solutions [Fal], which was acquired by Xilinx in late 2020 [Bus20], can automatically take care of these kinds of code transformations.

Code 5.1: CNN HLS C code snippet.

---

```

1 // Skip const variable initialization and macro definitions for brevity
2 void CnnKernel(const float* input ①, const float* weight①,
3               const float* bias ①, float* output ①) {
4
5     float C[ParallelOut][ImSize][ImSize];
6     for (int i = 0; i < NumOut / ParallelOut; ++i) { ④
7         // Initialization
8         for (int h = 0; h < ImSize; ++h) {
9             for (int w = 0; w < ImSize; ++w) {
10                for (int po = 0; po < ParallelOut; ++po)
11                    C[po][h][w] = bias[(i << shift) + po]; ②
12            } }
13        // Convolution
14        for (int j = 0; j < NumIn; ++j) { ⑤
15            for (int h = 0; h < ImSize; ++h) { ⑤
16                for (int w = 0; w < ImSize; ++w) { ⑤
17                    for (int po = 0; po < ParallelOut; ++po) { ⑤
18                        for (int p = 0; p < kKernel; ++p) { ⑤
19                            for (int q = 0; q < kKernel; ++q) ⑤
20                                C[po][h][w] += weight(i, po, j, p, q) * input(j, h + p, w + q); ② ③
21                    } } } } }
22        // ReLU + Max pooling
23        for (int h = 0; h < OutImSize; ++h) { ⑤
24            for (int w = 0; w < OutImSize; ++w) { ⑤
25                for (int po = 0; po < ParallelOut; ++po) { ⑤
26                    output(i, po, h, w) = max(0.f, C, po, h, w);
27                } } }
28    }
29 }

```

---

The final and the most critical level deals with FPGA-specific architectural optimizations, detailed in reasons 3-5 in Table 5.1, that vary from application to application. Although

<sup>2</sup>The Merlin Compiler is open-sourced at <https://github.com/Xilinx/merlin-compiler>

Table 5.1: Analysis of poor performance in Code 5.1.

	Reason	Required Code Changes for Higher Performance
①	Low bandwidth utilization	Manually apply memory coalescing using HLS built-in type <code>ap_int</code> .
②	High access latency to global memory	Manually allocate local buffers and use <code>memcpy</code> to enable memory burst.
③	Does not hide communication latency	Manually create load/compute/store functions and double buffering.
④	Lack of parallelism	Manually create parallel coarse-grained processing elements by wrapping the inner loops as a function and setting proper array partition factors.
⑤	Sequential execution	Apply <code>#pragma HLS pipeline</code> and <code>#pragma HLS unroll</code> with proper array partition factors for each processing element.

the Merlin Compiler also helps alleviate this problem to some extent by introducing a few high-level optimization pragmas and applying source-to-source code transformation to enable them (as explained in Section 2.1.1), these optimizations are much more difficult for software programmers to learn and apply effectively. More specifically, choosing the right part of the program to optimize, deciding the type of optimization and the pragmas to apply for enabling it, and tuning the pragma to get to the design with the highest quality complicate this level.

The requirement of mastering all three levels of optimizations makes the bar for general software programmers to use FPGA extremely high. Hence, general software programmers will lean towards other popular accelerators such as power-consuming GPUs with less consideration over FPGAs. These obstacles consequently result in huge barriers to the adoption of FPGA in data centers, the expansion of the FPGA user community, and the advances of FPGA technology. One possible solution is to apply an automated microarchitecture optimization. Thus, everyone with decent knowledge of programming can try customized

computing with minimum effort. To free accelerator designers from the iterations of HLS design improvement, automated Design Space Exploration (DSE) for HLS attracts more and more attention. However, existing DSE methods face the following challenges:

**Challenge 1: The large solution space:** The solution space grows exponentially by the number of candidate pragmas. In fact, only applying pipeline, unroll, and array partition pragmas to Code 5.1 produces  $10^{20}$  design points. This huge number of combinations creates a serious impediment to exploring the whole design space.

**Challenge 2: Non-monotonic impact of design parameters on performance and/or area:** As pointed out by Nigam et al. [Nig+20], we cannot assume that an individual design parameter will affect the performance/area in a smooth and/or monotonic way.

**Challenge 3: Correlation of different characteristics of a design:** When different pragmas are employed together in a design, they do not affect only one characteristic of a design. We will use the convolution part of the Code 5.1 as an example. If we apply fine-grained (*fg*) pipeline to *w* loop and parallelize the loop with a factor of 2, it results in a loop with initiation interval (II) of 2 synthesized by Vivado HLS 2018.3 [AMD24b]. However, when we change the parallel factor to 4, the HLS tool increases the II to 3 to optimize resource consumption by reusing some of the logic units instead of doubling the resource utilization. The analytical models usually fail to capture these cases. Furthermore, pipelining the *j* loop is part of the best design configuration. However, it does not improve the performance until after the *fg* pipelining is applied on the *w* loop. It suggests that the order of applying the pragmas is crucial in designing the explorer.

**Challenge 4: Implementation disparity of HLS tools:** The HLS tools from different vendors employ different implementation strategies. Even within the same vendor, the optimization and implementation rules keep changing across different versions. For example, the past Xilinx SDAccel versions consistently utilize *registers* to implement array partitions with small sizes to save BRAMs. However, the latest ones use *dual-port BRAMs* for implementation to support two reads in one cycle for achieving full pipelining, or  $II = 1$ , even if the array size is small. Such implementation details are hard to capture and maintain in

analytical models and make it difficult to port an analytical model built on a specific tool to the other.

**Challenge 5: Long synthesis time of HLS tools:** HLS tools usually take 5-30 minutes to generate RTL and estimate the performance—and even longer if the design has a high performance. This emphasizes the need for a DSE that can find the Pareto-optimal design points in fewer iterations.

In this chapter, as our first step to lowering the bar for general software programmers to make FPGA programming universally accessible, we focus on automating the final level of optimization. To solve challenges 2 to 4 mentioned above, instead of developing an analytical model, we treat the HLS tool as a black box. Challenges 1 and 5 imply that we need to explore the solution space intelligently. For that, we first apply the coordinate descent with the finite difference method to guide the explorer. However, we show that the general application-oblivious approaches fail to perform well for the HLS DSE problem. As a result, we present the AutoDSE framework that adapts a bottleneck-guided coordinate optimizer to systematically search for better configurations. Bottleneck-guided optimization approaches have been used successfully for CPU and GPU compilation optimization [Par+04; Hon+18], however the runtime for evaluation of the optimized code on CPUs and GPUs is much shorter. To speed up the process, we incorporate a flexible list-comprehension syntax to represent a grid design space with all invalid points marked which can help us prune the design space. In addition, we also partition the design space systematically to address the local optimum problem caused by Challenge 2.

In summary, this chapter makes the following contributions:

- We propose two strategies to guide DSE. One adapts the commonly used coordinate descent with the finite difference method and the other exploits a bottleneck-guided coordinate optimizer.
- We incorporate list-comprehension syntax to represent a smooth, grid design space with all invalid points marked.

- We develop the AutoDSE framework on top of the Merlin Compiler to automatically perform DSE using the bottleneck optimizer, enabling a systematic approach to pinpoint high-QoR design points.
- To the best of our knowledge, we are the first ones to evaluate our tool using the Xilinx optimized vision library [Xilb]. Evaluation results indicate that AutoDSE can achieve the same performance, yet with  $26.38\times$  reduction of their optimization pragmas resulting in less than one required optimization pragma per kernel, on the geometric mean.
- We evaluate AutoDSE on 11 computational kernels from MachsSuite [Rea+14] and Rodinia [Che+09] benchmarks and one convolution layer of AlexNet [KSH12], showing that we can achieve, on the geometric mean,  $19.9\times$  speedup over a single-thread CPU—only a 7% performance gap compared to manual designs.

## 5.2 Problem Formulation

Our goal is to expedite the hardware design by automating its exploration process. In general, there are two types of pragmas (using Vivado HLS as an example) that are applied to a program. One type is the *non-optimization* pragmas, which are relatively easy for software programmers to learn and apply. The other type is *optimization* pragmas, including PIPELINE and UNROLL pragmas. These pragmas require knowledge of FPGA devices and microarchitecture optimization experience, which are usually much more challenging for a software programmer to learn and master as explained in Section 5.1. The goal of this research is to minimize or eliminate the need to apply optimization pragmas *manually* and let AutoDSE insert them *automatically*. More formally, we formulate the HLS DSE problem as the following:

**Problem 1: Identify the Design Space.** Given a C program  $\mathcal{P}$  as the FPGA accelerator kernel, construct a design space  $\mathbb{R}_{\mathcal{P}}^K$  with  $K$  parameters that contains possible combinations of HLS pragmas for  $\mathcal{P}$  as design configurations.

**Problem 2: Find the Optimal Configuration.** Given a C program  $\mathcal{P}$ , we would like to insert a minimal number of optimization pragmas manually to get a new program  $\mathcal{P}'$  as the FPGA accelerator kernel along with its design space set  $\mathbb{R}_{\mathcal{P}'}^K$ , which is identified in Problem 1, and we let the DSE tool insert the rest of the pragmas automatically. More specifically, having a vendor HLS tool  $\mathbf{H}$  that estimates the execution cycle  $Cycle(\mathbf{H}, \mathcal{P}')$  and the resource utilization  $Util(\mathbf{H}, \mathcal{P}')$  of the given  $\mathcal{P}'$  as a black-box evaluation function, the DSE must find a configuration  $\theta \in \mathbb{R}_{\mathcal{P}'}^K$  in a given search time limit so that the generated design  $\mathcal{P}'(\theta)$  with  $\theta$  can fit in the FPGA and the execution cycle is minimized. Formally, our objective is:

$$\min_{\theta} Cycle(\mathbf{H}, \mathcal{P}'(\theta)) \quad (5.1)$$

subject to

$$\begin{aligned} \theta &\in \mathbb{R}_{\mathcal{P}'}^K \\ \forall u \in Util(\mathbf{H}, \mathcal{P}'(\theta)), u &< T_u \end{aligned} \quad (5.2)$$

where  $u$  is the utilization of one of the FPGA on-chip resources and  $T_u$  is a user-available resource threshold on FPGAs. We set all  $T_u$  to 0.8, an empirical threshold, in our experiments. Beyond 0.8, the design will suffer from high clock frequency degradation due to the difficulty in placement and routing. In addition, the rest of the resources are left for the interface logic of the vendor HLS tool.

Note that we introduce two optimization objectives; one minimizes the optimization pragmas that have to be inserted manually to obtain  $\mathcal{P}'$ , and the other maximizes the performance of  $\mathcal{P}'$  using AutoDSE by applying pragmas automatically. Obviously, there is a trade-off between the two. An expert designer can always get an optimized microarchitecture to achieve the best performance by inserting enough HLS optimization pragmas. However, it is time-consuming and not feasible for software programmers with little or no FPGA design experience. In our evaluation, our goal is to match the performance of well-designed HLS library code (typically written by experts) yet insert much fewer optimization pragmas *manually*. Indeed, our experimental results in Section 5.5 show that we can achieve this with

26.38× pragma reduction on the geometric mean, requiring less than 1 optimization pragma per kernel.

## 5.3 The AutoDSE Framework

To reduce the size of the design space, we build our DSE on top of the Merlin Compiler [Con+16b; Con+16a; Fal]. We reviewed the Merlin Compiler in Section 2.1.1 and discussed how it can facilitate the accelerator design. We will provide more detail on how it can help our DSE and present the solution for Problem 1 in Section 5.3.1. Then, we give an overview of AutoDSE in Section 5.3.2.

### 5.3.1 Merlin Compiler and Design Space Definition

As elaborated in Section 2.1.1, the Merlin Compiler [Con+16b; Con+16a; Fal] was developed to make FPGA programming easier by requiring a reduced set of high-level optimization directives and automatically generating the respective HLS code with the necessary HLS pragmas and code transformations. Since the number of pragmas required by the Merlin Compiler is much smaller, it defines a more compact design space, which makes it a better fit for developing a DSE as shown in [Con+18a; Yu+18]. For instance, Code 5.2 shows the CNN kernel with Merlin pragmas. With inserting only four lines of pragmas and no further *manual* code transformation, the Merlin Compiler can transform Code 5.2 to a high-performance HLS kernel with the same performance as the manually optimized design written in HLS C which has 28 pragmas and 150 lines of code (Section A.1).

The Merlin Compiler, by default, applies code transformations to address the bottlenecks 1 and 2 listed in Table 5.1 and provides high-level optimization pragmas for the rest of them. For example, instead of rewriting Code 5.1 to test whether double buffering would help the performance as described in reason 3 in Table 5.1, we just need to use the `cg pipeline` pragma and the Merlin Compiler will rewrite the code to satisfy it. As a result, our focus in this work is on finding the best location of each of these high-level pragmas and tuning

## Code 5.2: Optimized CNN code snippet in Merlin C.

---

```
1 void CnnKernel(const float input[NumIn][InImSize][InImSize],
2               const float weight[NumOut][NumIn][kKernel][kKernel],
3               const float bias[NumOut], float output[NumOut][OutImSize][OutImSize]) {
4     float C[ParallelOut][ImSize][ImSize];
5     for (int i = 0; i < NumOut/ParallelOut; i++) {
6         // Initialization
7         for (int h = 0; h < ImSize; ++h) {
8 #pragma ACCEL parallel factor=4
9             for (int w = 0; w < ImSize; ++w){
10                for (int po = 0; po < ParallelOut; po++)
11                    C[po][h][w] = 0.f;
12            } }
13        // Convolution
14 #pragma ACCEL pipeline
15        for (int j = 0; j < NumIn; ++j) {
16            for (int h = 0; h < ImSize; ++h) {
17 #pragma ACCEL parallel factor=4
18 #pragma ACCEL pipeline FLATTEN
19                for (int w = 0; w < ImSize; ++w) {
20                    for (int po = 0; po < ParallelOut; po++){
21                        float tmp = 0;
22                        for (int p = 0; p < kKernel; ++p) {
23                            for (int q = 0; q < kKernel; ++q){
24                                tmp += ... } }
25                        C[po][h][w] += tmp;
26                    } } } }
27        // Skip ReLU + Max Pooling for brevity
28    } }
```

---

them, automatically; hence, we can address reasons 3-5 in Table 5.1 as well by enabling the architectural optimizations along with the best pipelining and parallelization attributes. Consequently, our solution to Problem 1 is defined as in Table 5.2. We identify the design space for each kernel by analyzing the kernel Abstract Syntax Tree (AST) to gather loop trip counts, available bit widths, etc. The rules we enforce in building this design space are listed in Section 5.4.3.

Now that we have defined the design space in Table 5.2 for *Problem 1*, we focus on *Problem 2* in the remainder of this chapter. Although to some extent, Merlin pragmas



alleviate the manual code reconstruction overhead, a designer still has to manually search for the best option for each pragma, including position, type, and factors. In fact, there are a total of 27 candidate pragmas for the CNN design in Code 5.1, which result in  $\sim 10^{16}$  design configurations. The large design space motivates us to develop an efficient approach to find the best configuration more systematically.

Table 5.2: Design space definition based on the Merlin pragmas.

Factor	Design Space (Values)
CG-loop parallel	$\left\{ u \mid 1 < u \leq TC(L), u \mid TC(L) \text{ or } u = 2^k \text{ for } k \in \mathbb{Z} \right\}$
FG-loop parallel	$\left\{ u \mid \begin{cases} 1 < u \leq TC(L), u \mid TC(L) \text{ or } u = 2^k \text{ for } k \in \mathbb{Z}, & TC(L) > 16 \\ u = TC(L), & \text{otherwise} \end{cases} \right\}$
loop pipeline	$\left\{ p \mid p \in \{off, cg, fg\} \right\}$
loop tiling	$\left\{ t \mid 1 < t < TC(L), t \mid TC(L) \right\}$

CG: Coarse-grained; FG: Fine-grained; TC: Loop trip-count

### 5.3.2 Framework Overview

We develop and implement AutoDSE as a push-button framework, depicted in Fig. 5.1, based on the strategies explained in Section 5.4. The framework first automatically builds a design space by analyzing the kernel AST according to the rules and the syntax described in Section 5.4.3. Then, it profiles and selects representative partitions using K-Means as mentioned in Section 5.4.4. For each partition, AutoDSE’s explorer performs DSE using the proposed bottleneck-based coordinate strategy in Section 5.4.2 and the parameter ordering explained in Section 5.4.2.1. The explorer can be tuned to evaluate the quality of design points based on different targets such as performance, resource, or finite difference (Eq. 5.5). When the explorer finishes exploring a partition, it stores the best configuration found by that partition and reallocates the working threads to other partitions to keep the resource utilization high. Finally, when all partitions are finished, AutoDSE outputs the design configuration with the best QoR among all partitions.

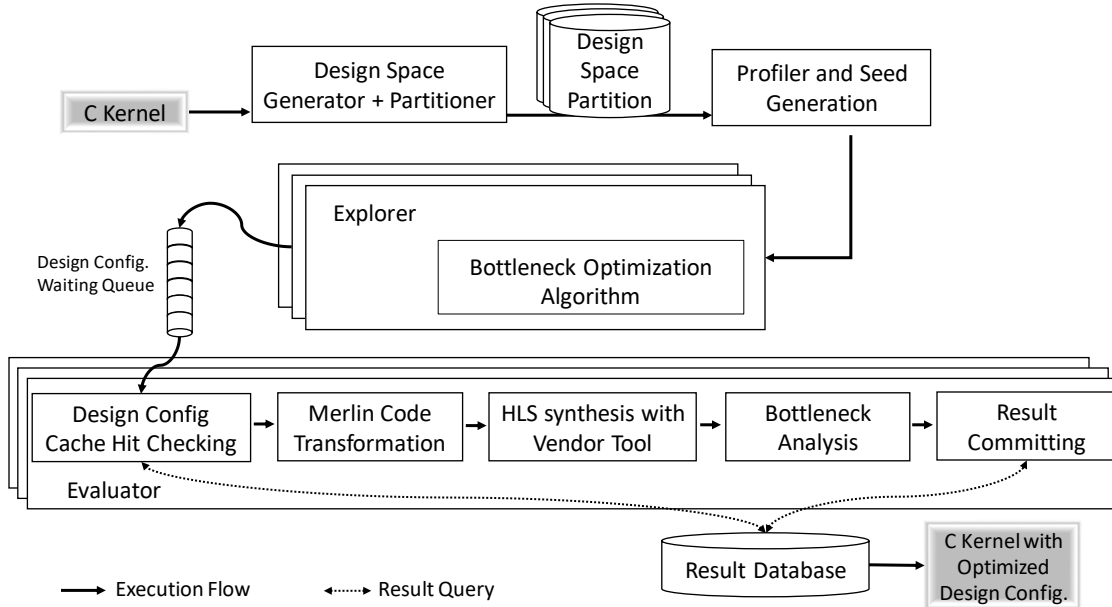


Figure 5.1: The AutoDSE framework overview.

## 5.4 AutoDSE Methodology

In this section, we first examine the efficiency of the application-oblivious heuristics in Section 5.4.1. As we will discuss, the main drawback of these heuristics for the HLS DSE problem is the fact that they do not have any knowledge of the semantics of the program’s parameters. This problem can potentially linger the DSE process since the explorer may waste a lot of time on parameters with no impact on the results at that stage of optimization. As a result, in Section 5.4.2, we present a bottleneck-guided coordinate optimizer that can mimic an expert’s optimization method and generate high-QoR design points in fewer number of iterations. We propose several optimizations in the remainder of this section to further improve the performance of our framework.

### 5.4.1 Application-oblivious Heuristics

A prominent prior art, S2FA [Yu+18], adapted OpenTuner [Ans+14], a popular search engine, for the HLS DSE problem. OpenTuner leverages the Multi-Armed Bandit (MAB)

approach [Fia+10] to assemble multiple meta-heuristic algorithms for high generalization. These meta-heuristics include uniform greedy mutation, differential evolution genetic algorithm, particle swarm optimization, and simulated annealing for the case of S2FA. At each iteration, the MAB selects the meta-heuristic with the highest credit and updates the meta-heuristic’s credit based on the QoR. This means that the meta-heuristic that can find high-quality design points will be rewarded and activated more frequently by the MAB, and vice versa.

S2FA also employs the Merlin Compiler as its backend and further applies more strategies to improve the OpenTuner efficiency when performing DSE for HLS. We use S2FA to perform the DSE for 24 hours and depict the speedup of our benchmark cases compared to the corresponding manual design over time in Fig. 5.2. The black dot indicates the time that S2FA finds the overall best design point. We can see that S2FA requires on average 16.8 hours to find the best solution. We further analyze the exploration process and find that most designs have an obvious performance bottleneck (e.g., low utilization of global memory bandwidth, insufficient parallel factors, etc.), which usually dominates more than half of the overall cycle counts and is controlled by only one or two design parameters (pragmas). In this situation, the performance gain of tuning other parameters is often very limited, but it is hard for the problem-independent searching algorithm to learn that. In fact, it needs many iterations to identify the key parameter and tune it to resolve the performance bottleneck. After that, it has to spend a large number of iterations again to find the next key parameter. Since we rely on the HLS tool to assess a design point, it makes the search process very time-consuming. This phenomenon motivates us to develop a new search algorithm that is guaranteed to optimize the key parameter (high-impact parameter) prior to others.

Coordinate descent is another well-known iterative optimization algorithm for finding a local minimum point. It is based on the idea that one can minimize a multi-variable function by minimizing it along one direction at a time and solving single-variable optimization problems. At each iteration, we generate a set of candidates,  $\Theta_{cand}$ , as the input to the algorithm. Each candidate is generated by advancing the value of each parameter in the

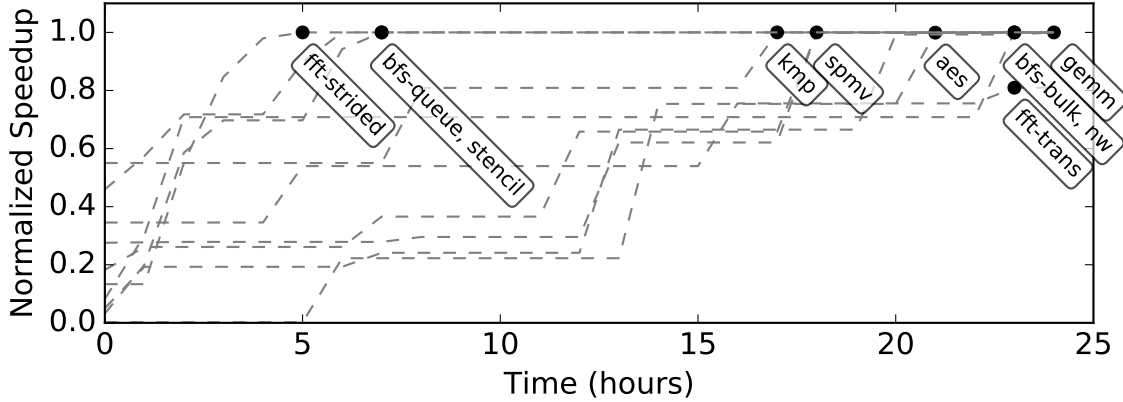


Figure 5.2: Performance speedup of the design found by S2FA [Yu+18] compared to the manual design over time.

current configuration by one step. Formally, the  $c$ -th candidate generated from design point  $\theta_i$  is:

$$\theta_i^c = [p_0, p_1, \dots, next(p_c), \dots, p_K] \quad (5.3)$$

where  $K$  is the total number of parameters,  $p_c$  is the value of  $c$ -th parameter in  $\theta_i$ ,  $next(p_c)$  denotes the next value of this parameter (the next numeric factor for the `parallel` and `tiling` pragmas and the next mode of pipelining for the `pipeline` pragma). Accordingly, at each iteration, we will generate  $K$  candidates, forming the pool of candidates  $\Theta_{cand}$ , and execute the HLS tool  $K$  times to determine the next configuration as follows:

$$\theta_{i+1} = \underset{\theta_i^c \in \Theta_{cand}}{argmin} g(\theta_i^c, \theta_i) \quad (5.4)$$

We leverage the finite difference method to approximate the coordinate value by treating the HLS tool as a black box. That is, given a candidate configuration  $\theta_j$  deviated from the current configuration  $\theta_i$ , the coordinate value is defined as:

$$g(\theta_j, \theta_i) \sim \frac{Cycle(\mathbf{H}, \mathcal{P}(\theta_j)) - Cycle(\mathbf{H}, \mathcal{P}(\theta_i))}{Util(\mathbf{H}, \mathcal{P}(\theta_j)) - Util(\mathbf{H}, \mathcal{P}(\theta_i))} \quad (5.5)$$

We calculate  $Util(\mathbf{H}, \mathcal{P}(\theta))$  by taking into account all the different types of resources using the following formula:

$$Util(\mathbf{H}, \mathcal{P}(\theta)) = \sum_u 2^{\frac{1}{1-u}} \quad (5.6)$$

where  $u$  is the utilization of one of the FPGA resources. We use exponential function to penalize the over-utilization of FPGA more seriously. Note that Eq. 5.5 considers not only performance gain but also resource efficiency, so it could reduce the possibility of being trapped in a local optimum. For example, we may reduce 10% execution cycle by spending 30% more area if we increase the parallel factor of a loop (configuration  $\theta_1$ ); we can also reduce 5% execution cycle by spending 10% more area if we enlarge the bit-width of a certain buffer (configuration  $\theta_2$ ). Although  $\theta_1$  seems better in terms of the execution cycle, it may be more easily trapped by a locally optimal point because it has a relatively limited resource left to be further improved. On the other hand, the finite difference values for the two configurations are  $g(\theta_1, \theta_0) = \frac{-10\%}{30\%} = -0.3$  and  $g(\theta_2, \theta_0) = \frac{-5\%}{10\%} = -0.5$ , so the system prioritizes the second configuration for better long-term performance.

By leveraging the coordinate descent with a finite difference method, we expect to find a better design point every  $K$  HLS runs. Unfortunately, as mentioned in Challenge 2 of Section 5.1, the performance trend is not always smooth, so the coordinate process can easily be trapped by a low-quality locally optimal design point. Moreover, the efficiency of using the coordinate-based approach for DSE is limited by the number of parameters. More specifically, at each iteration, we need to evaluate  $K$  design points, where  $K$  is the total number of tuning parameters, to determine the next step. On the other hand, in most cases, only a few of the  $K$  tuning parameters have a high impact on the performance, so we should evaluate only the  $K'$  impactful parameters at each iteration where  $K' < K$ . For instance, the design space generator will instrument Code 5.1 with 27 pragmas based on the rules explained in Section 5.4.3 and the coordinate-based approach proposed in this section needs to assess the quality of 27 new designs in each iteration. However, in the early iterations, the convolution part takes more than 85% of the total cycle counts of the kernel. As a result, changing the pragmas outside of this part will have an insignificant effect on the performance; hence, it is wasteful to explore them at this stage.

### 5.4.2 AutoDSE Exploring Strategy - Bottleneck-guided Coordinate Optimizer

Two main inefficiencies of the approaches reviewed in the previous section are: 1) they must evaluate many design points to identify the performance bottleneck, and 2) they have no knowledge of the semantics of the parameters, so they have no way of differentiating them and prioritizing the important ones. Identifying the key parameters is not straightforward. Although the HLS report may provide the cycle breakdown for the loop and function statements, it is hard to map them to tuning parameters due to the applications of several code transformations applied by the Merlin Compiler. Fortunately, the Merlin Compiler includes a feature that transmits the performance breakdown reported by the HLS tool to the user input code. More specifically, when performing code transformation, the Merlin Compiler records the code change step by step so that it can propagate the latency estimated by the HLS tool back to the user input code. This feature allows us to identify the performance bottleneck by traversing the Merlin report and mapping the bottleneck statement to one or a few tuning parameters.

By exploiting the cycle breakdown, we can develop a bottleneck analyzer to resolve the issues mentioned above. We first build a map from the loop or function statements in the user input code to design parameters so that we know which parameters can have the highest impact on a particular statement. To identify the critical path and type, we start with the kernel top function statement and build hierarchy paths of the design by traversing the Merlin report using Depth-First Search (DFS). More specifically, for each hierarchy level, we first check to see if the current statement has child loop statements and sort them by their latency. Then, we traverse each of the child loops and repeat this process. In the case of a function call statement, we dive into the function implementation to further check its child statements for building the hierarchy paths. Finally, we return a list of paths in order. Note that since we sort all loop statements according to their latency by checking the Merlin report, the hierarchy paths we created will also be sorted by their latency.

Subsequently, for each statement, we check the Merlin report again to determine whether

its performance bottleneck is memory transfer or computation. The Merlin Compiler obtains this information by analyzing the transformed kernel code along with the HLS report. A cycle is considered to be a memory transfer cycle if it is consumed by communicating to the global memory. As a result, we can not only figure out the performance bottleneck for each design point but also identify a small set of effective design parameters for consideration. Therefore, we can significantly improve the efficiency of our search algorithm.

When we obtain an ordered list of critical hierarchy paths from the bottleneck analyzer, we start from the innermost loop statement (because of the DFS traversal) of the most critical entry and identify its corresponding parameters as candidate parameters to explore, if they are not already tuned. Based on the bottleneck type, provided by the bottleneck analysis (i.e., memory transfer or computation), we pick a subset of the parameters mapped to that statement to evaluate. For example, we may have design parameters of `parallel` and `tiling` at the same loop level. When the bottleneck type of the loop is memory transfer, we focus on the `tiling` parameter for the loop; otherwise, we focus on the `parallel` parameter. In other words, we reduce the number of candidate design parameters not only by the bottleneck statement but also by the bottleneck type.

We define each design point as a data structure containing the following information:

---

```
curr_point = DesignPoint(configuration, tuned, result, quality, children)
```

---

where *configuration* contains the value of all the parameters and *tuned* lists the parameters that the algorithm has explored for the current point. *quality* stores the quality of design measured by finite difference value and *result* includes all the related information gathered from the HLS tool including the resource utilization and the cycle count. Finally, each design point stores a stack of the configurations for its unexplored children where each child is generated by advancing one of the parameters by one step. The children are pushed to the stack in the order of their importance (from least to most important) as computed by the bottleneck analyzer so that by popping the stack, we get to work with the child who can change the parameters and possibly have the most promising impact.

We define level  $n$  as a point where we have fixed the value of  $n$  parameters, so the

maximum level in our algorithm is equal to the total number of parameters. For each level, we define a heap of the pending design points that can be further explored and push the design points by their *quality* into the heap. Since new design points are sorted by their quality values when they were pushed into the heap, the design point with a better quality value will be chosen for tuning more of its parameters prior to other points. As mentioned above, the next point to be explored is chosen by popping the stack of the unexplored children of this design point so that at each step, we get to evaluate the most promising design point.

Algorithm 1 presents our exploring strategy. As we will explain in Section 5.4.4, we partition the design space to alleviate the local optimum problem. For each partition, we first get its default point and initialize the heap of the first level (lines 2 to 7). Then, at each iteration of the algorithm, AutoDSE gets the heap with the highest level, peeks the first node of the heap, and pops its stack of unexplored children to get the new candidate (lines 9 to 12). Next, each option of the new focused parameter will be evaluated and the result will be passed to the bottleneck analyzer to generate a new set of focused parameters for making new children (lines 14 to 19). Since the number of fixed parameters is increased by one, it will be pushed to the heap of the next level if there is still a parameter left that has not been tuned yet (lines 20 to 24). When the stack of unexplored children of the current design point is empty, it will be popped out of the heap (lines 26 to 28). The algorithm continues either until all the heaps are empty or when the DSE has reached a runtime threshold (Line 8).

As an example, when AutoDSE optimizes Code 5.1, it will see that the *convolution* part of the code takes 85.2% of the overall cycle counts. Since that section of the code is a series of nested loops, the parameters of the innermost loop will take the top of the list produced by the bottleneck analyzer. We shall explain in Section 5.4.3 (Rule 1) that we do not consider loops with a trip count of less than or equal to 16 in our DSE since the HLS tool can automatically optimize these loops well. As a result, the `w` loop in Line 16 would be the innermost loop with parameters which the Merlin report tells us it is a computation-bound loop. According to the parameter ordering described in Section 5.4.2.1, AutoDSE first tries



---

**Algorithm 1** AutoDSE Explorer: Bottleneck-guided Coordinate Optimizer

---

**Require:** A C program  $\mathcal{P}$  with top function  $top\_func$  and a set of design space partitions  $\mathbb{P}$ .

**Ensure:** A design configuration  $\theta$  with the best QoR.

```
1: for all  $P \in \mathbb{P}$  do
2:    $best\_cfg = cfg \leftarrow GetDefaultPoint(P)$ 
3:    $report, hier \leftarrow Evaluate(cfg)$ 
4:    $parameter\_order \leftarrow BottleneckAnalysis(report, hier, top\_func, \emptyset)$ 
5:    $children \leftarrow GetChildren(cfg, parameter\_order)$ 
6:    $LevelHeap \leftarrow \emptyset; LevelHeap.append(\emptyset)$ 
7:    $LevelHeap[0].push(DesignPoint(cfg, \emptyset, report.result, 0, children))$ 
8:   while  $LevelHeap \neq \emptyset$  and  $elapsed\_time < DSE\_TIMEOUT$  do
9:      $curr\_level = GetLastLevel(LevelHeap)$ 
10:     $curr\_point \leftarrow LevelHeap[curr\_level].peek()$ 
11:     $tuned\_parameters = curr\_point.tuned$ 
12:     $candidate\_cfg, focused\_parameter \leftarrow curr\_point.children.pop()$ 
13:    for all  $option \in focused\_parameter$  do
14:       $new\_cfg \leftarrow Manipulate(candidate\_cfg, focused\_parameter, option)$ 
15:       $new\_tuned \leftarrow tuned\_parameters + [(focused\_parameter, option)]$ 
16:       $report, hier \leftarrow Evaluate(new\_cfg)$ 
17:       $quality \leftarrow CalQuality(report.result, "FiniteDifference")$ 
18:       $best\_cfg \leftarrow UpdateBest(new\_cfg, quality)$ 
19:       $parameter\_order \leftarrow BottleneckAnalysis(report, hier, top\_func, new\_tuned)$ 
20:      if  $len(parameter\_order) > 0$  then
21:         $children \leftarrow GetChildren(new\_cfg, parameter\_order)$ 
22:         $new\_point \leftarrow DesignPoint(new\_cfg, new\_tuned, report.result, quality, children)$ 
23:         $LevelHeap[curr\_level + 1].push(new\_point)$ 
24:      end if
25:    end for
26:    if  $LevelHeap[curr\_level].peek().NumChildren == 0$  then
27:       $LevelHeap[curr\_level].pop()$ 
28:    end if
29:  end while
30: end for
31: return  $best\_cfg$ 
```

---

to apply `fg pipeline` which would be a successful attempt. In the next iteration, the last level heap will contain the design point that was just optimized and since the *convolution* part is still the bottleneck, AutoDSE would try parallelizing the `w` loop and will choose `factor=4` since it achieves the highest *quality* value. Although `factor=8` can reduce the cycle count by 11%, it increases the overall area (Eq.5.6) by 63% which results in a worse quality; therefore, AutoDSE picks `factor=4` to make room for further improvement. By adopting Algorithm 1, AutoDSE can improve the performance by 218× very quickly, only after 2 iterations of the algorithm.

#### 5.4.2.1 Parameter Ordering

It often happens that each bottleneck type has more than one applicable design parameter. In these situations, we sort the parameters by a pre-defined priority. For example, if the bottleneck of a loop statement is determined to be its computation, one can apply `fg` or `cg` pipelining/parallelization, in general. In this case, we treat the `pipeline` pragma as two different parameters based on its mode and choose the order of applying the pragmas to be `fg pipeline`, `parallel`, and `cg pipeline` which is a heuristic approach to improve the performance by utilizing more fine-grained parallelization units since the HLS tool handles such optimizations better. Here, measuring the quality of design points with the finite difference value (Eq. 5.5) helps AutoDSE not to over-utilize the FPGA. For a configuration, when the gain of the achieved speedup is not comparable to the loss of available resources, the quality of design decreases; hence, AutoDSE will not tune that parameter and the resources are left for applying a design parameter with a higher impact.

Moreover, as mentioned in Challenge 3 of Section 5.1, the order of applying the pragmas is crucial in order to get to the best design. Our experiments show that prioritizing the evaluation of the fine-grained optimizations helps AutoDSE reach the best design point in fewer iterations. This is mainly because HLS tools schedule fine-grained optimizations better than coarse-grained ones. Table 5.3 shows how the performance and resource utilization change when `fg pipeline` and `parallel` pragmas are applied to line 16 in Code 5.1 compared to

Table 5.3: Performance and area compared to the base design when the parameters of Line 16 in Code 5.1 change. TIMEOUT is set to 60 minutes. The results suggest that applying fine-grained optimization first lets the HLS tool synthesize the design easier.

Optimization	Status	Perf	BRAM	LUT	DSP	FF
Pi-fg	PASS (24 min)	175×	+7%	+23%	+24%	+15%
PF=4	TIMEOUT	-	-	-	-	-
Pi-fg + PF=4	PASS (28 min)	218×	+17%	+44%	+33%	+25%

Pi: Pipeline, PF: Parallel Factor, fg: fine-grained

the base design where all the pragmas are off. The time limit to run the HLS tool is set to 60 minutes. The results suggest that in order to get to the optimal configuration for this loop, we must first apply fine-grained pipelining. This way, the HLS tool can better schedule the loop when parallelization is further applied and its synthesis will finish in 28 minutes. However, if we first apply the other pragma which results in a coarse-grained parallelization, the synthesis will be timed out and AutoDSE does not tune this pragma at this stage.

Note that we do not prune the other design parameters. We just change the order of the parameters to be explored as these rules cannot be generalized to all cases due to the unpredictability of the HLS tools. If the bottleneck of a design point is memory transfer, AutoDSE prioritizes `cg pipeline` over the `tiling` pragma. The Merlin Compiler, by default, caches the data and the former will further overlap the communication time with computation by applying double buffering; however, the latter can be used to change the size of the cached data.

### 5.4.3 Efficient Design Space Representation

To further facilitate the bottleneck-based optimizer, we seek to reduce the ineffective parameters. Intuitively, we can build a grid design space from the Merlin pragmas by treating each pragma as a tuning parameter and searching for the best combination. However, many points in this grid space may be infeasible. For example, if we have determined to perform

coarse-grained pipelining at the outermost loop of a loop nest, the Merlin Compiler will apply double-buffering on the loop. In this case, the physical meaning of double-buffering at the outermost loop is to transfer a batch of data from DRAM to BRAM, which cannot be further parallelized (assuming we only have one DRAM bank with a single read port). As a result, pipeline and parallel pragmas are mutually exclusive for this loop. We propose an efficient approach to creating a design space that preserves the grid design space but invalidates infeasible combinations.

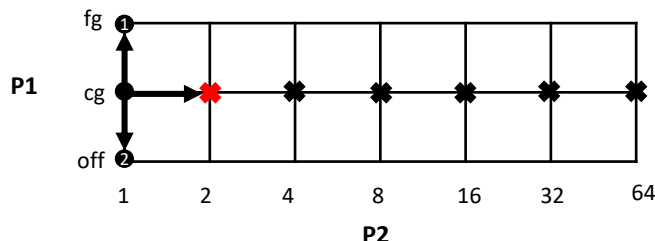


Figure 5.3: The proposed design space representation and its impact on DSE.  $P1$  and  $P2$  denote the pipeline and parallel pragmas of loop  $j$  in Code 5.1, respectively.

Fig. 5.3 illustrates the goal of an efficient design space representation. In this example, we attempt to explore the best parameter with the best option for loop  $j$  of Code 5.1 with pragma  $P1$  and  $P2$  denoting the pipeline and parallel pragmas, respectively. Pragma  $P1$  and  $P2$  are exclusive when  $P1$  is used in `cg` mode; therefore, only one of them should be inserted at a time. A good design space representation must preserve the grid design space but invalidate infeasible points. An example of such representation is presented in Fig. 5.3. Assume that we are at the configuration  $(P1, P2) = (cg, 1)$ , we only have two candidates to explore in the next step because the configuration  $(P1, P2) = (cg, 2)$  is invalid. This representation is exploration-friendly and, it is easy to enforce rules regarding the infeasible points.

To represent a grid design space with invalid points, we introduce a *Python* list comprehension syntax to AutoDSE. The *Python* list comprehension is a concise approach for creating lists with conditions. It has the following syntax:

---

```
list_name = [expression for item in list if condition]
```

---

Formally, we define the design space representation for Merlin pragmas with list comprehensions as follows:

---

```
#pragma ACCEL <pragma-type> <attribute-key>=auto{
  options: parameter_name=list-comprehension-expression;
  default: default_value }
```

---

For our example, the design space can be represented using list comprehensions as follows:

---

```
1 // Skip the rest due to the page limit
2 #pragma ACCEL pipeline auto{
3   options: P1 = [x for x in [off, cg, fg]];
4   default: 'off' }
5 #pragma ACCEL parallel factor=auto{
6   options: P2 = [x for x in [1, 2, 4, 8, 16, 32, 64] if x == 1 or P1 != cg];
7   default: 1 }
8 for (int j = 0; j < NumIn; ++i) {
9 // Skip the rest due to the page limit
```

---

where line 6 indicates that the two pragmas are exclusive. In other words, when we set  $P1 = cg$ , the available option for  $P2$  is only the default value, which is 1 in this case. Note that the default value of each pragma turns it off.

There are three main advantages to adopting list comprehension-based design space representations. First, we are able to represent a design space with exclusive rules to greatly reduce its size. Second, the *Python* list comprehension is general. It provides a friendly and comprehensive interface for higher layers such as polyhedral analysis [Zuo+13] and domain-specific languages to generate an effective design space in the future. Third, the syntax of this representation is *Python* compatible. This means we can leverage the *Python* interpreter to evaluate the design space and improve the overall stability of the DSE framework.

The Design Space Generator, depicted in Fig. 5.1, adapts the Rose Compiler [Lab] to analyze the kernel AST and extract the required information for running the DSE such as the loops in the design, their trip-count, and available bit-width. Artisan [Van+20] employs

a similar approach for analyzing the code. However, it only considers the unroll pragma in code instrumentation. Our approach, on the other hand, considers a wider set of pragmas as listed in Table 2.2 and exploits the following rules to prune the design space:

1. Ignore the fine-grained loops with a Trip Count (TC) of less than or equal to 16 as the HLS tool can schedule these loops well.
2. Tiling Factors (TF) should be integer divisors of their loop TC.
3. The allowed Parallel Factors (PF) for a loop are all powers of two and sub-divisors of the loop TC up to  $\min(128, TC)$  plus the TC itself. PF of larger than 128 causes the HLS tool to run for a long time and it usually does not result in a good performance.
4. For each loop, we should have  $TF * PF \leq TC$ .
5. When `fg pipeline` is applied on a loop, no other pragma is allowed for the inner loops since this parameter wants to unroll all the inner loops completely.
6. A `parallel` pragma is invalid for a loop nest when `cg pipeline` is applied on that loop.
7. A `tiling` pragma is added only to the loops with an inner loop.

#### 5.4.4 Design Space Partitioning

Unfortunately, the third inefficiency of the approaches reviewed in Section 5.4.1 also exists in our bottleneck-guided optimizer. We still cannot identify whether the current option of a parameter is locally or globally optimum. The most promising solution is breaking the dependency between the options and searching for a set of them in parallel. Although we need to evaluate multiple design points at every iteration, each design point will provide the maximum information for improving the performance because we always evaluate the parameters that have the largest impact on the performance bottleneck.

By partitioning the design space based on the likely distribution of locally optimal points and exploring each partition independently, we alleviate the local optimum issue caused by the non-smooth performance trend (Challenge 2 in Section 5.1) since each partition starts exploring from a different design point. Intuitively, we could partition the design space according to the range of values of every parameter in a design, but it may generate thousands of partitions and result in a long exploration time. Instead, we partition the design space based on the pipeline mode, as `fg pipeline` unrolls all the sub-loops while the `cg pipeline` exploits double buffers to implement coarse-grained pipelining. These two modes could significantly influence the resulting architecture, with anticipated differences in performance and resource utilization being unrelated. According to the pipeline modes in each loop, we use the tree partition and generate  $2^m$  partitions from a design space with  $m$  non-innermost loops.

Supposing we use  $t$  working threads to perform, at most,  $h$  hours of DSE for  $2^m$  design space partitions, we need  $\frac{2^m}{t} \times h$  hours to finish the entire process. On the other hand, some partitions that are based on an insignificant pipeline pragma may have a similar performance, so it is more efficient to only explore one of them. As a result, we profile each partition by running HLS with minimized parameter values to obtain the minimum area and performance and use K-means clustering with performance and area as features to identify  $t$  representative partitions among all  $2^m$  partitions.

## 5.5 Evaluation

### 5.5.1 Experimental Setup

Our evaluation is performed on Amazon Elastic Compute Cloud (EC2) [Ama24]. We use `r4.4xlarge` instance with 16 cores and 122 GiB memory to perform the DSE and generate accelerator bitstreams. The generated FPGA accelerators are evaluated on an F1 instance (`f1.2xlarge`) with Xilinx Virtex UltraScale+™ VU9P FPGA. We chose the commonly-used MachSuite [Rea+14] benchmark suite and the FPGA-friendly Rodinia [Che+09] benchmark,

along with one convolution layer of Alexnet [KSH12] as our first benchmark. For several common kernels, MachSuite provides C implementation that is programmed without the consideration of FPGA acceleration, which makes it a natural fit for demonstrating our approach. We evaluate the effect of our optimizations and compare the designs generated by our tool to the state-of-the-art works using this benchmark. Furthermore, to the best of our knowledge, we are the first ones to evaluate the performance of our tool on vision kernels of Xilinx Vitits libraries [Xilb] that are optimized for Xilinx FPGAs, based on the OpenCV library [Bra00].

### 5.5.2 Impact of Parameter Ordering

In this section, we examine the parameter ordering we explained in Section 5.4.2.1. It is crucial to note that prioritizing the evaluation of fine-grained pragmas becomes particularly significant in scenarios with multiple nested loops, as exemplified in our running Code 5.1. With increased nesting, the distinction between coarse-grained and fine-grained optimizations becomes more pronounced. Additionally, as the loop hierarchy deepens, applying coarse-grained optimizations becomes more challenging for the HLS tool. Table 5.4 summarizes the performance and area metrics of the generated design based on the pragma ordering described in Section 5.4.2.1 (`fg pipeline`, `parallel`, and `cg pipeline`) versus the reversed ordering. The synthesis timeout is set to 1 hour and we ran AutoDSE for 7 hours. As the results demonstrate, by prioritizing fine-grained optimizations, AutoDSE achieves a significant speedup of  $143\times$ . This clearly shows that by first optimizing the innermost loops and then moving upwards to further optimize the outer loops, we not only get better performance but also will be able to explore more design points. This happens because optimizing the inner loops makes it easier for the HLS tool to integrate that part into a larger microarchitecture.



Table 5.4: Effect of the parameter ordering on the performance and area of the generated optimized design for Code 5.1 after running the DSE for 7 hours.

Ordering	#explored	Perf	BRAM	LUT	DSP	FF
Pi-fg->parallel->Pi-cg	71	143×	68%	40%	39%	24%
Pi-cg->parallel->Pi-fg	63	1×	53%	1%	~0%	~0%

Pi: Pipeline, fg: fine-grained, cg: coarse-grained

### 5.5.3 Evaluation of Optimization Techniques

We first measure the performance of the Merlin Compiler without any pragmas and without the help of AutoDSE to get the impact of its default optimizations. The 1<sup>st</sup> bar of each case in Fig. 5.4 depicts the speedup gained by the Merlin Compiler with respect to CPU. Then, we evaluate the original Coordinate Descent (CD) method described in Section 5.4.1 and the proposed optimization strategies explained in sections 5.4.3 and 5.4.4. The 2<sup>nd</sup> to 4<sup>th</sup> bars in Fig. 5.4 show the speedup gained after tuning the pragmas by each of these optimizations. Note that the chart is on a logarithmic scale. We can see that the default optimizations of the Merlin Compiler are not enough and after applying the candidate pragmas generated by the Original CD, we get 13.52× speedup, on the geometric mean. Moreover, each of the proposed strategies benefits at least one case in our benchmark and together further brings a 2.47× speedup. The list-based design space representation keeps the search space smooth by invalidating infeasible combinations. As a result, we can investigate more design points in a fixed amount of time. This helps AES, NW, KMP, PATHFINDER, KMEANS, and KNN. Design space partition benefits the designs with many loop nests in which the coordinate process is easily trapped by the local optimum when changing pipeline modes—such as AES, GEMM, NW, STENCIL-2D, and STENCIL-3D.

The 5<sup>th</sup> bar shows the speedup of AutoDSE when the bottleneck-guided coordinate optimizer detailed in Section 5.4.2 is adapted along with the parameter order explained in Section 5.4.2.1, design space representation introduced in Section 5.4.3, and design space partitioning described in Section 5.4.4. With this setup, AutoDSE further improves the

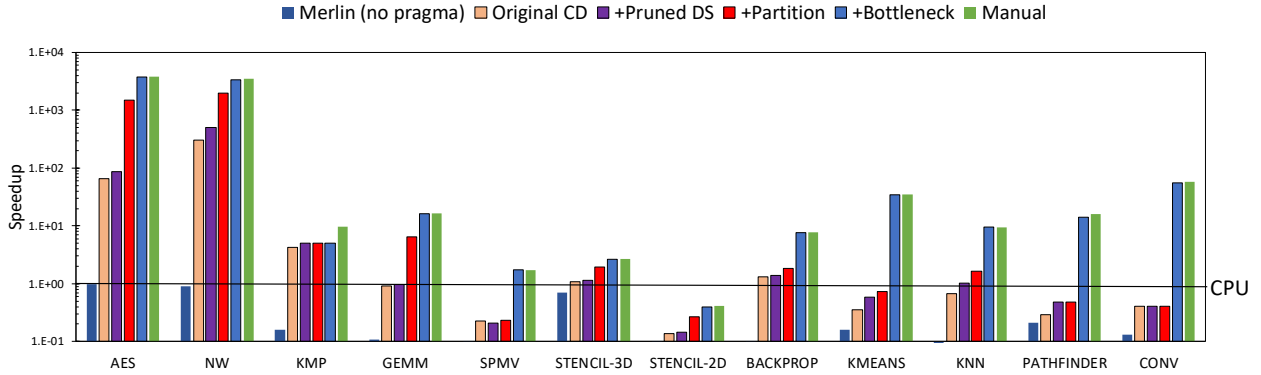


Figure 5.4: Comparison of performance speedup over an Intel Xeon CPU core with different compilation strategies: no pragmas, pragma-augmented design generated by different approaches, and manually optimized design.

result by  $5.5\times$  on the geometric mean bringing the overall speedup compared to when no pragmas are applied to  $182.92\times$ . As a result, AutoDSE can achieve a speedup of  $19.9\times$  over CPU and get to  $0.93\times$  performance of the manual designs while running only for 1.1 hours on the geometric mean. The manual designs, depicted by the 6<sup>th</sup> bar, are optimized by applying the Merlin pragmas *manually* without changing the source programs.

Fig. 5.5 depicts the AutoDSE process for four cases where the bottleneck-guided optimizer showed significant performance improvement. This shows that our approach can rapidly achieve a high-performance design. AutoDSE does not exactly match the performance of manual designs for all of the cases because of the costly process of evaluating each design point with the HLS tool, which can take minutes to hours. Consequently, in our experiments, AutoDSE has only explored a limited portion of the solution space for most of the kernels. Furthermore, the HLS report may not reflect accurate computation cycles when the kernels contain many unbounded loops or while-loops, which in turn affects the Merlin report. To get the importance of the parameters, the bottleneck analyzer (explained in Section 5.4.2) needs to receive the accurate cycle estimation of the design. In the absence of the true cycle breakdown, it cannot determine the high-impact design parameters. Therefore, our search algorithm may focus on unimportant parameters.

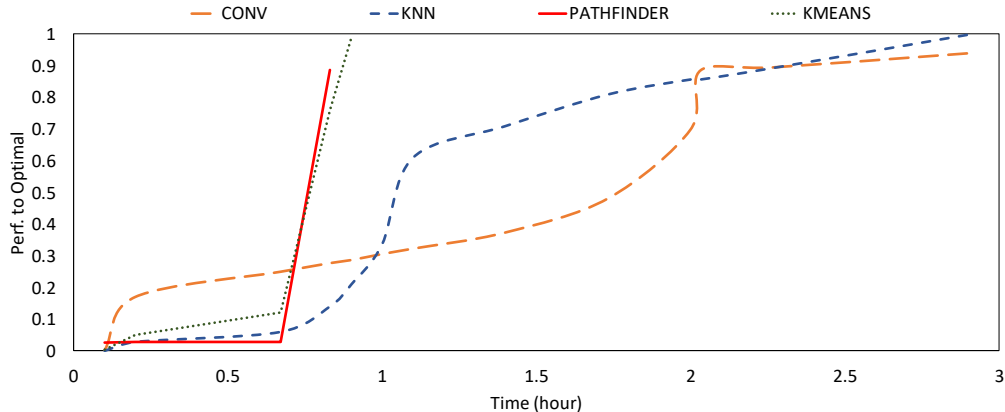


Figure 5.5: Performance speedup of generated designs compared to the manual designs over time using AutoDSE for four cases where the bottleneck-guided optimizer demonstrated significant impact.

#### 5.5.4 Comparison with Other DSE Approaches

We further evaluate the overall performance of the generated accelerator designs by AutoDSE compared to the previous state-of-the-art works including S2FA [Yu+18], lattice-traversing DSE [FAP18b], and Gaussian process-based Bayesian optimization [Sun+21] in Table 5.5. The numbers show the speedup of the design found by AutoDSE compared to the design that their framework found after running the tools for the same allotted time. Note that the performance of the other works is not reported by the authors for all of the kernels we are testing. According to Table 5.5, by utilizing the bottleneck-based approach, we can outperform S2FA, lattice-traversing DSE, and Gaussian process-based Bayesian optimization by  $3.45\times$  ( $86.56\times$ ),  $4.23\times$  ( $5.11\times$ ),  $17.92\times$  ( $43.33\times$ ) respectively, on the geometric (arithmetic) mean.

As we discussed in Section 5.4.1, the deficiency of S2FA stems from how hard it is for the problem-independent learning algorithm to find the key parameters. Lattice-traversing DSE needs an initial sampling step to learn the design space. This takes a long time for our benchmark due to the size of the design space even though the authors only consider unrolling the loops and function inlining. This constraint makes it hard for the tool to start

Table 5.5: Performance speedup of our approach compared to S2FA [Yu+18], Lattice-traversing DSE [FAP18b], and Gaussian process-based Bayesian Optimization [Sun+21]

Approach	AES	NW	GEMM	KMP	SPMV	STENCIL-3D	GEO-Mean	ARITH-Mean
Lattice [FAP18b]	1.63	6.32	7.39	-	-	-	<b>4.23</b>	<b>5.11</b>
S2FA [Yu+18]	512.86	1	1.52	1.74	1	1.26	<b>3.45</b>	<b>86.56</b>
Bayesian [Sun+21]	-	-	100.17	-	2.07	27.75	<b>17.92</b>	<b>43.33</b>

the exploration process before the time limit for DSE is met. The Gaussian process-based Bayesian optimization also has to spend some time to sample the design space and build an initial surrogate model. However, AutoDSE can learn the high-impact directives (pragmas) by exploring the performance breakdown and thus, is able to find a high-performance design in a few iterations.

Moreover, adopting the Merlin Compiler as the backend gives a further advantage to AutoDSE compared to other DSE tools. This allows the tool to exploit the automatic code transformations for applying common optimization techniques such as memory burst, memory coalescing, and double buffering; and focus only on high-level hardware changes. Nonetheless, the performance comparison with S2FA shows that adoption of the Merlin Compiler alone is not enough and we still need to explore the design space more efficiently. Another source of inefficiency in the S2FA framework is in the translation of code from Scala to C code to be used by the Merlin Compiler. The generated C code, in general, may not be efficient enough for HLS, especially for applications like AES that require bit-level optimizations.

### 5.5.5 Comparison with Expert-level Manual HLS Designs

To further evaluate the performance of AutoDSE, we use 33 vision kernels from the Xilinx Vitis Library [Xilb]. These kernels utilize 14 optimization pragmas, on average (by the geomet-

ric mean), which include UNROLL, PIPELINE, ARRAY\_PARTITION, DEPENDENCE, LOOP\_FLATTEN, INLINE, DATAFLOW, and STREAM. For each kernel, we remove all the optimization pragmas except for DATAFLOW and STREAM. The removed pragmas, which are of the first six types mentioned above, are used 13.47 times on average (out of 14). As a result, we require less than one optimization pragma per kernel, on the geometric mean. The only optimization pragmas kept are DATAFLOW and STREAM pragmas. This is because our search space is built on top of the Merlin Compiler and we do not search for the DATAFLOW and STREAM pragmas as these pragmas are not among the Merlin-specified pragmas. The INTERFACE and LOOP\_TRIPCOUNT pragmas are also kept which are not among the HLS optimization pragmas. They are rather used to specify the connection to the AXI bus and the range of the trip count of the loop, respectively.

Table 5.6: Average (geometric mean) performance speedup of the Vitis tool, the Merlin Compiler, and AutoDSE over the manually optimized kernels from Xilinx Vitis libraries. The manual designs are the original kernels from the library and are summarized in the Manually Optimized column for Vitis. The performance of those designs is compared to when the optimization pragmas we search for (UNROLL, PIPELINE, ARRAY\_PARTITION, DEPENDENCE, LOOP\_FLATTEN, and INLINE) are removed and the code is passed to the three different tools. AutoDSE is able to retrieve the removed pragmas automatically.

Compared to Comparison Scenario	Vitis (Manually Optimized)	Vitis (Default)	Merlin Compiler	AutoDSE
Speedup over the Vitis Library with (Original) Manually Inserted Pragmas	1×	0.12×	0.38×	1.04×
Performance Improvement over the Vitis Tool with Default Settings (no pragma)	8.69×	1×	3.29×	9.04×
#pragmas Listed in the Table’s Caption	13.47	0	0	0
Total Optimization Pragma Reduction (including DATAFLOW and STREAM pragmas)	1×	26.38×	26.38×	26.38×

To better understand the effect of our optimizer, we tested the performance of the Vitis tool and the Merlin Compiler on the input to AutoDSE (which does not include the

optimization pragmas mentioned above). The performance comparisons are summarized in Table 5.6. As the results show, while the Merlin Compiler can get to a speedup of  $3.29\times$  compared to the Vitis tool, it still needs the help of AutoDSE to get to the manually optimized kernels in the library. In fact, AutoDSE could achieve a further speedup of  $2.74\times$  by automatically inserting 3.2 Merlin pragmas per kernel, on the geometric mean. As a result, it could improve the performance of the Vitis tool by  $9.04\times$  and  $1.04\times$  when the code with the reduced set of pragmas and the manual code, respectively, are passed to the Vitis tool.

Fig. 5.6 depicts the performance comparison of the design points AutoDSE generated compared to the Xilinx results along with the number of pragmas that we removed in detail. The results show that AutoDSE can achieve the same or better performance yet require  $26.38\times$  fewer optimization pragmas in its input code and it can find the optimal design configuration in 0.3 hours, on the geometric mean; therefore, proving the effectiveness of our bottleneck-based approach and the fact that it can mimic the method an expert would take. For the cases that AutoDSE does not exactly match the performance of Vitis, AutoDSE still finds the best combination of the pragmas. The inequality lies in the different II that Merlin has achieved. For example, the `histEqualize`, `histogram`, and `otsuthreshold` kernels all have a loop that requires the II to be set to 2 when `pipeline` pragma is used. Otherwise, Vivado HLS achieves an `II=3`. However, it is not possible to change the II using the Merlin Compiler. On the other hand, AutoDSE is able to outperform the performance of `customConv` and `reduce` kernels significantly by better detecting the choices and locations for pipelining and parallelization.

### 5.5.6 Frequency of Employed Parameters

We explored the number of candidate pragmas and the number of pragmas that AutoDSE tuned for optimizing the design to find out the most influential parameters. Figure 5.7 illustrates the total count of candidate pragmas in comparison to those that were adjusted to their non-default values for the optimal design identified by AutoDSE for different target kernels. For illustration purposes, we depict the frequency of employed pragmas for those

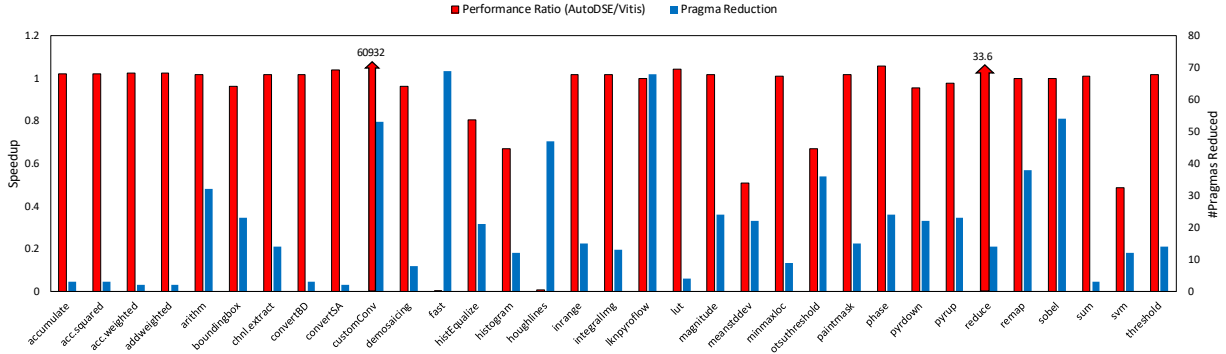


Figure 5.6: Performance speedup and the number of reduced pragmas using AutoDSE compared to the Vision kernels of Xilinx Vitis libraries [Xilb]

who either had the highest speedup or utilized the most number of pragmas, meaning that AutoDSE spent more iterations for optimizing them. As the results suggest, the `tiling` pragma has a special use case, only `CONV` and `STENCIL-2D` utilized it. For the rest of the kernels, it is either not applicable or not necessary. The `parallel` and `pipeline` pragmas are equally important. The `pipeline` pragma is employed more where there is a nested loop, whereas, for single loops, the `parallel` pragma is more preferred. This is because the HLS tools can implement both `fg` and `cg` pipelining better than `cg` parallelization. Also, when a single loop does not have any pragmas, it will be pipelined automatically. Looking at the vision kernels from the Xilinx Vitis Library that we targeted, there are 228, 122, and 18 candidate `parallel`, `pipeline`, and `tiling` pragmas in total, respectively. Since these kernels rarely utilize deep nested loops, AutoDSE chose 65 and 48 of the `parallel` and `pipeline` pragmas in total, respectively; while, it never used any `tiling` pragmas.

## 5.6 Conclusion

In this chapter, we have taken our initial, yet highly significant, attempt towards lowering the barrier for accelerating programs with FPGA, aiming to make FPGAs universally accessible to general software programmers. We analyzed the difficulty of exploring the HLS design space and identified five challenges in Section 5.1. To address challenges 2 to 4 mentioned in

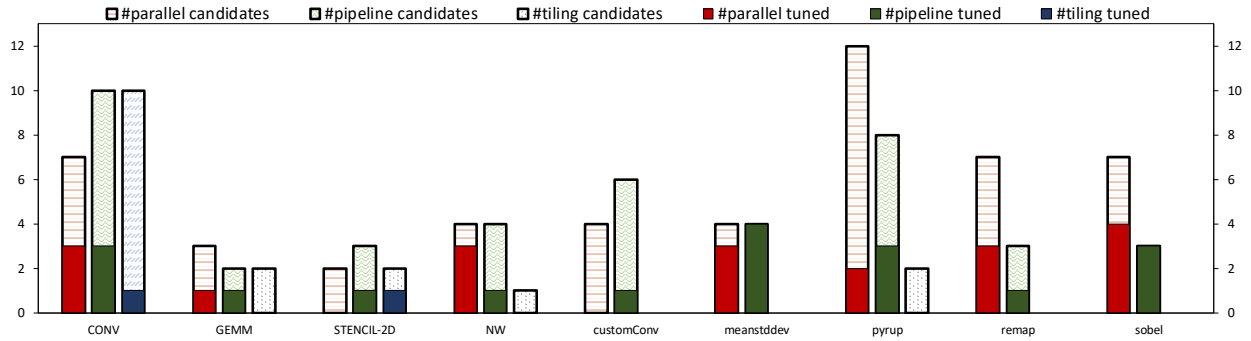


Figure 5.7: The number of candidate pragmas vs. the number of pragmas that were adjusted to their non-default values for the optimal designs of each target kernel.

Section 5.1, we treat the HLS tool as a black box. We use the synthesis results to estimate the QoR rather than the placement and routing ( $PnR$ ) results. This choice is made due to the excessive time required for  $PnR$ , which limits the exploration of an adequate number of design points within a reasonable time frame. According to our observation and analysis, we propose a bottleneck-guided coordinate optimizer and develop a push-button framework, AutoDSE, based on that to systematically approach a better solution. By exploring the solution space efficiently, we address challenges 1 and 5. We propose a heuristic for ordering the parameters that can further help challenges 3 and 5. To eliminate meaningless design points, we incorporate a list comprehension-based design space representation and prune  $24.65\times$  ineffective configurations on average while keeping the design space smooth; hence, further alleviating Challenge 1. Additionally, we employ a partitioning strategy to address the local optimum problem mentioned in Challenge 2. We show that AutoDSE can outperform general hyper-heuristics used in the literature by focusing on high-impact design parameters first. The experimental results suggest that AutoDSE lets anyone with a decent knowledge of programming try customized computing with minimum effort.

AutoDSE is built with the assumption that we can get the performance breakdown of the program from the HLS tool. We expect all HLS tools will provide performance breakdown at some point, as it is important for manual performance optimization (such as the need for Intel VTune Profiler [Int24c] in the case of CPU performance optimization). AMD Xilinx



HLS is already providing such information that the Merlin Compiler leverages. It is likely that other HLS tools [Cad24; Sie24; NEC24; Int24a] will add such information as well in the near future. Hence, we believe, it is reasonable for AutoDSE to take advantage of such information to mimic the human performance optimization process to perform bottleneck-driven DSE.

## CHAPTER 6

# GNN-DSE: Automated Accelerator Optimization Aided by Graph Neural Networks

Using HLS, the hardware designers must describe only a high-level behavioral flow of the design. However, it still can take weeks to develop a high-performance architecture mainly because there are many design choices at a higher level to explore. In the previous chapter, we developed an automated framework, AutoDSE, to explore the solution space. While AutoDSE has proven effective for the HLS DSE problem and outperformed previous state-of-the-art methods, its primary limitation is its reliance on the HLS tool for design point evaluation. This decision stems from the challenge of developing a model capable of accurately capturing the intricate behavior of the HLS tool, given that design choices may not exhibit a linear impact on performance and are often highly interrelated. Nonetheless, the time-intensive nature of HLS tool synthesis, ranging from several minutes to hours per design, limits the exploration of total design points. To solve this problem, we aim to develop a graph neural network model trained to emulate the functionality of the HLS tool for a wide range of applications. The experimental results demonstrate that our approach, termed *GNN-DSE*, can estimate the quality of design in milliseconds with high accuracy, resulting in up to  $79\times$  speedup (with an average of  $44\times$ ) in running the explorer for optimizing the design compared to AutoDSE<sup>1</sup>.

---

<sup>1</sup>All materials are available at <https://github.com/UCLA-VAST/GNN-DSE>.

## 6.1 Introduction

Relying on the HLS tool to evaluate a solution can increase the DSE time significantly, given that each design candidate may require a lengthy evaluation period (ranging from minutes to hours). As a result of this limitation, we can explore only a fraction of the solution space. While utilizing a model can potentially speed up the process, a simple analytical model cannot capture the different heuristics used by the tool [SW19]. Adopting a learning algorithm can help with increasing the accuracy. However, the related works mostly train a separate model for each application which limits the scalability. We rather seek to develop a model in which the knowledge gained from one application can be transferred to another one. A nice effort was made by Kwon et al. [KC20] for transfer learning using a Multi-Layer Perceptron (MLP) network. Nonetheless, they only use the pragma configurations as the input to the model, which can result in considerable loss since the program semantics are missing (see Section 6.4.2.3).

As current HLS tools optimize the design based on specific code patterns, learning to identify the different code patterns and their effect can help with transferring the knowledge across different applications. This property has motivated a few of the very recent works to propose to represent the program as a graph and develop a GNN-based model to predict the design’s quality [WXH22; Ust+20]. They have shown that this approach can help with learning the operation mapping to FPGA’s resources for delay prediction in HLS [Ust+20] or predicting the performance of the program under different resource allocations (DSP or LUT) to its computation nodes [WXH22]. Although their studies clearly demonstrate the value and power of applying GNNs, a complete representation of the program including its pragmas and a model to learn the impact of the pragmas on the design’s quality is still missing.

In this chapter, we aim to automate the design optimization using GNN with the support for *transfer learning* by developing a framework called GNN-DSE. We first build a model to evaluate the design quickly, in milliseconds, without the invocation of the HLS tool. Since

the HLS tools employ many heuristics to optimize a design and the design parameters affect each other, we let a deep learning model learn their impact. Furthermore, as the current HLS tools optimize the design based on specific code patterns, it is important to identify the different code patterns and learn their effect to be able to transfer the knowledge we gained from one application to another. As such, we represent the program as a graph that includes the program semantics in the form of control, data, call, and pragma flows and exploit a GNN to extract the required features of the graph for predicting the objectives. We propose several techniques for improving the accuracy of the model including Jumping Knowledge Network (JKN) [Xu+18], node attention [Li+15], and multi-head objective prediction. To demonstrate the effectiveness of our model, we build a DSE on top of it to find the Pareto-optimal design points. We show that not only can GNN-DSE find the Pareto-optimal designs for the kernels that were included in its training set, but it can also generalize to the kernels outside of its database and detect their Pareto-optimal design points. To the best of our knowledge, this work is the first one to employ a graph representation that captures both the program semantics and the pragmas and to build a *single* predictive model for several applications with transfer learning capability. In this chapter, we target AMD Xilinx FPGAs as an example but our approach is tool-independent and extendable to Intel FPGAs as well.

In summary, this chapter makes the following contributions:

- We propose a graph-based program representation for optimizing FPGA designs which includes both the program context and the pragma flow.
- We develop a learning model based on GNN as a surrogate of the HLS tool for assessing a design point’s quality in milliseconds and propose several techniques for improving its accuracy.
- We build an automated framework, GNN-DSE, to gather a database of FPGA designs, train a learning model for predicting the design’s objectives, and conduct design space exploration using the model to identify the high-performance design points.
- The experimental results indicate that GNN-DSE is capable of identifying Pareto-

optimal design points for the kernels within its database. Furthermore, it demonstrates the ability to optimize *unseen* kernels, similar to those in the training set, by leveraging the knowledge acquired during its training process.

## 6.2 Problem Formulation

In this chapter, our objective is to speed up the DSE process for HLS by facilitating the rapid estimation of the design point’s quality. For this matter, we propose solutions for the following problems:

**Problem 1: Build the Prediction Model.** Let  $\mathcal{P}$  be a C program as the FPGA accelerator kernel with the candidate design configurations ( $\theta$ ), where  $\mathbb{R}_{\mathcal{P}}$  is a set of all the different configurations. Let  $\mathbf{H}$  be a vendor HLS tool that outputs the true execution cycle  $Cycle(\mathbf{H}, \mathcal{P}(\theta))$  and the true resource utilization  $Util(\mathbf{H}, \mathcal{P}(\theta))$  of the program  $\mathcal{P}$ :

$$\mathbf{Q}_{\mathbf{H}}(\mathcal{P}(\theta)) = (Cycle(\mathbf{H}, \mathcal{P}(\theta)), Util(\mathbf{H}, \mathcal{P}(\theta))) \quad (6.1)$$

Find a prediction function ( $\mathbf{F}$ ) that approximates the results of  $\mathbf{H}$  for any given program  $\mathcal{P}$  with any design configurations ( $\theta$ ):

$$\min_{\mathbf{F}} (average_{\theta} (Loss(\mathbf{Q}_{\mathbf{F}}(\mathcal{P}(\theta)), \mathbf{Q}_{\mathbf{H}}(\mathcal{P}(\theta)))) \quad (6.2)$$

In the case of a regression task, the loss function is calculated using Root Mean Squared Error (RMSE) over all the designs. For the classification task, the percentage of misclassified cases and the F1 scores are considered.

**Problem 2: Identify the Optimal Configuration.** Our final goal, like in AutoDSE, is to find the Pareto-optimal design points. The difference is that, here, we aim to use a prediction function  $\mathbf{F}$  that can mimic the HLS tool instead of invoking the tool. More specifically, for the program  $\mathcal{P}$  defined above, our objective is to find a configuration  $\theta \in \mathbb{R}_{\mathcal{P}}$  within a given search time limit such that the generated design  $\mathcal{P}(\theta)$  fits in the FPGA and minimizes the execution cycle. Formally, our objective is:

$$\min_{\theta} Cycle(\mathbf{F}, \mathcal{P}(\theta)) \quad (6.3)$$

subject to

$$\theta \in \mathbb{R}_P^K, \quad \forall u \in Util(\mathbf{F}, \mathcal{P}(\theta)), u < T_u \quad (6.4)$$

where  $u$  is the utilization of one type of the FPGA on-chip resources and  $T_u$  is a user-defined threshold for that type on the FPGA. As explained in Chapter 5, we set all  $T_u$  to 0.8 (80%), an empirical threshold, in our experiments.

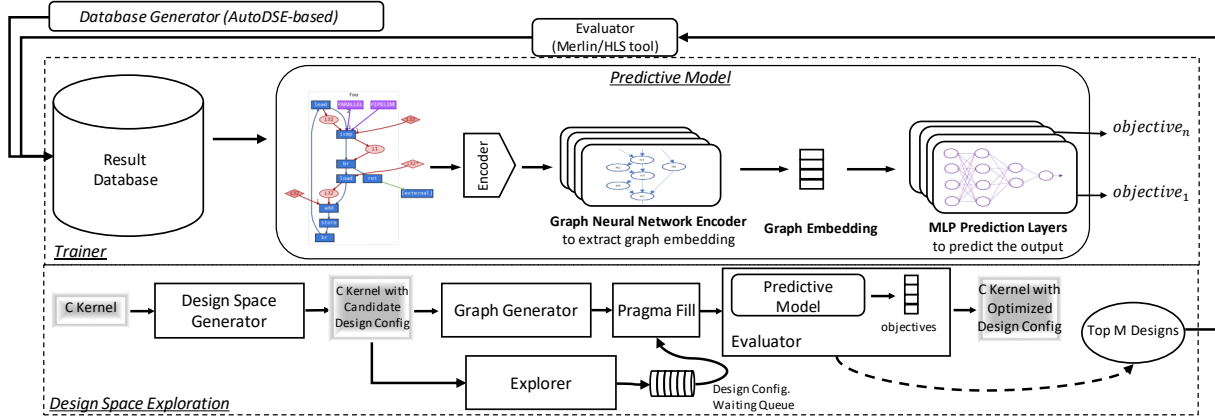


Figure 6.1: High-level overview of our model-based frameworks for optimizing the design.

### 6.3 Our Proposed Methodology

Figure 6.1 presents a broad outline of our model-based frameworks designed to enhance the quality of HLS designs. The *Trainer* component aids in devising solutions to Problem 1, as outlined in Section 6.2, while the *Design Space Exploration* addresses the second problem. We first collect a database from various applications (Section 6.3.1) and represent each design in the database as a graph (Section 6.3.2). Then, we train a *predictive model* for estimating the design’s objectives (Section 6.3.3). Finally, the *predictive model* can be used as a surrogate to the HLS tool to run *inference* and *DSE* stages (Section 6.3.4).

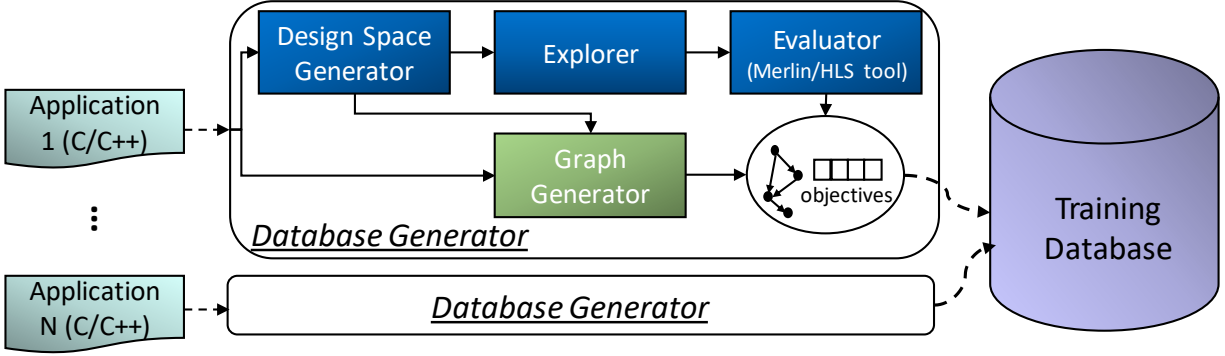


Figure 6.2: Database generator for a model-based DSE framework such as GNN-DSE.

### 6.3.1 Database Generation

We adapt AutoDSE to generate the initial database for each of the applications. Fig. 6.2 demonstrates our approach to it. Each `for` loop can take up to three pragmas: `pipeline`, `parallel`, and `tiling`. Note that the Merlin Compiler automatically inserts the rest of the required HLS pragmas as mentioned in Section 2.1.1. We exploit AutoDSE’s rules for pruning the design configurations (e.g., when fine-grained pipelining is applied on a loop, the inner loops would not take any pragmas) and generate the solution space based on that. Since the model needs to see a variety of design points from “bad” to “good” to learn to distinguish them, GNN-DSE extends AutoDSE to exploit three types of explorers:

- The existing explorer of AutoDSE, the bottleneck-based optimizer, which can find high-quality designs.
- A hybrid explorer integrates bottleneck-based optimization with exhaustive search. It assesses up to  $P$  neighbors of the optimal design point following a  $X\%$  enhancement in quality. In this context, a neighbor is defined as a point where only one pragma option differs from the current point. This exploration approach enables the model to observe the impact of altering only one pragma within each local neighborhood.
- A random explorer that may explore configurations overlooked by the previous two explorers.

Once the explorer picks a design point, it is passed to the Merlin Compiler for evaluation. The result will be committed to a common database along with the program’s graph representation (Section 6.3.2). GNN-DSE gradually collects results from *different* applications in a shared space to be used for training the model. Obtaining the true values of a design’s objectives is time-consuming. This makes gathering the dataset for training the model to be the primary bottleneck in our approach. After building an initial database, we leverage the top points generated by our DSE (Section 6.3.4) to augment the database. It is important to note that the DSE aims to evaluate the model on numerous unseen data points, necessitating a comprehensive representation of all design choices in our database. On the other hand, if our DSE mistakenly believes that an unseen design point has a high QoR, it indicates that the model lacks enough data to generalize across the entire solution space. These particular data points, which led to mispredictions of Pareto-optimal design points, are more likely to contribute to a more accurate representation of the data distribution in subsequent rounds.

### 6.3.2 Program Representation

A popular way of representing a program as a graph is to extract its *Control and Data Flow Graph* (CDFG) from its Intermediate Representation (IR) in LLVM [LA04]. Thus, instead of focusing on the grammar of the code, the semantics of the program flow is captured. In a CDFG, the nodes represent the LLVM instructions that are connected based on the control flow of the program. For the data flow of the program, a second type of edge is added between the nodes based on the operands of the instructions. Note that a CDFG includes many low-level operations like memory management which makes it desirable for FPGA kernels.

On the downside, the CDFGs ignore the precision of the operands and their values, which are crucial in determining the design’s objectives. Recently, a more convenient program representation has been proposed, ProGraML [Cum+21], which extends the CDFG by explicitly assigning separate nodes to operands to retrieve the missing information. It also keeps the function hierarchies by including the design’s call flow. As such, we adapt ProGraML and



extend it by including the *pragma* flow to represent an HLS design. In our DSE works including GNN-DSE, each of the candidate pragmas is defined in either of the following forms:

---

```
#pragma ACCEL pipeline auto{pragma_name}
#pragma ACCEL parallel factor=auto{pragma_name}
#pragma ACCEL tiling factor=auto{pragma_name}
```

---

the `pragma_name` is a placeholder for its option, which can be (`off|cg|fg`) for `pipeline` and a numerical value for the other two as defined in Section 5.3.1. `cg` (`fg`) refers to coarse-grained (fine-grained) pipelining (Section 2.1.1).

For each candidate pragma, we designate a new node. Given that the pragmas are applied to loops, we link this node to one of the instruction nodes associated with the loop: `icmp`. Code 6.1 shows a toy example having a simple `for` loop with two candidate pragmas. Fig. 6.3 depicts its graph representation. We only show the relevant nodes here for illustration purposes. As Fig. 6.3 demonstrates, there are four types of nodes in each graph. The first kind (in blue) is for the LLVM instructions that together demonstrate the control flow of the program. The second and third kinds (in red) exhibit the constant values (diamond shape) and variables (oval shape) that capture the data flow of the program. The pragma nodes (fourth kind) are presented as purple boxes connecting to the respective `icmp` node. The edges also have different kinds which show the different flows of the graph: control (blue), data (red), call (green), and pragma (purple). When there are two or more edges of the same type connected to a node, they are numbered to further distinguish them (see the edges connecting from pragma nodes to the `icmp` node).

Code 6.1: Code snippet of an input toy example to GNN-DSE.

---

```
1 void foo(int input[N]) {
2 #pragma ACCEL pipeline auto{PIPE_L1}
3 #pragma ACCEL parallel factor=auto{PARA_L1}
4   for (int i = 0; i < N; i++) {
5       input[i] += 1;
6   } }
```

---

As discussed in Chapter 5, HLS tools typically produce better outcomes when pragmas

are applied to inner loops, as they find it easier to implement fine-grained optimizations. Thus, the model can enhance its predictive capability if the representation includes information about the loop level for each pragma within a nested loop. Fortunately, our graph representation inherently incorporates loop levels through program control flow. We can further encode this information in each node using the LLVM block ID of the `for` loop. More specifically, each node/edge has the following attributes:

---

```
Node = {'block': LLVM block ID, 'key_text': Node key task, 'function': Function ID, 'type': Node type}
Edge = (Src node ID, Dst node ID, {'flow': Flow type, 'position': Position ID})
```

---

the `type`, `flow`, and `position` attributes encode this information:

<b>type</b>	0: instruction	1: variable	2: constant value	3: pragma
<b>flow</b>	0: control	1: data	2: call	3: pragma
<b>position</b>	0: tiling	1: pipeline	2: parallel	-

The `key_text` attribute shows a keyword corresponding to that node. Here is an example for each of the control, data, and pragma nodes from the graph in Fig. 6.3:

---

```
'full_text': #pragma ACCEL PIPELINE auto{PIPE_L1}, 'key_text': PIPELINE
'full_text': %0 = load i32, i32* %i, align 4, 'key_text': load
'full_text': i32* %input, 'key_text': i32*
```

---

Fig. 6.4 illustrates the *graph generator* of GNN-DSE. It receives a C/C++ code as input and constructs its graph utilizing the LLVM IR of the program alongside its candidate pragmas. In each design configuration, the `auto` variables within the pragma placeholders are substituted with their respective values. Consequently, within the graphs for various design configurations of one application (kernel), only the attributes of their pragma nodes differ.

### 6.3.3 Predictive Model

Fig. 6.5 depicts our model architecture for predicting the design’s objectives. It takes the graph representation of the program as the input and creates the initial node/edge embeddings by concatenating the one-hot encoding of their attributes (Section 6.3.2) and important numeric values, including pragma options and loop trip counts. This encoding helps the

```

1 ...
2 for.cond:
3 %0 = load i32, i32* %i, align 4
4 %cmp = icmp slt i32 %0, 10
5 br i1 %cmp, label %for.body, label %for.end
6
7 for.body:
8 %1 = load i32, i32* %i, align 4
9 %idxprom = sext i32 %1 to i64
10 %arrayidx = getelementptr inbounds [10 x i32],
    [10 x i32]* %a, i64 0, i64 %idxprom
11 %2 = load i32, i32* %arrayidx, align 4
12 %inc = add nsw i32 %2, 1
13 store i32 %inc, i32* %arrayidx, align 4
14 br label %for.inc
15 ...

```

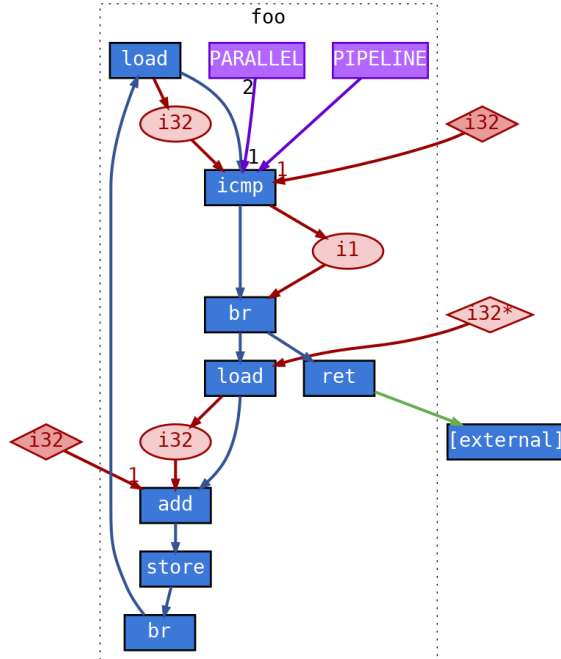


Figure 6.3: Part of the LLVM IR of Code 6.1 and its graph representation illustrating the different kinds of nodes and edges in GNN-DSE’s representation.

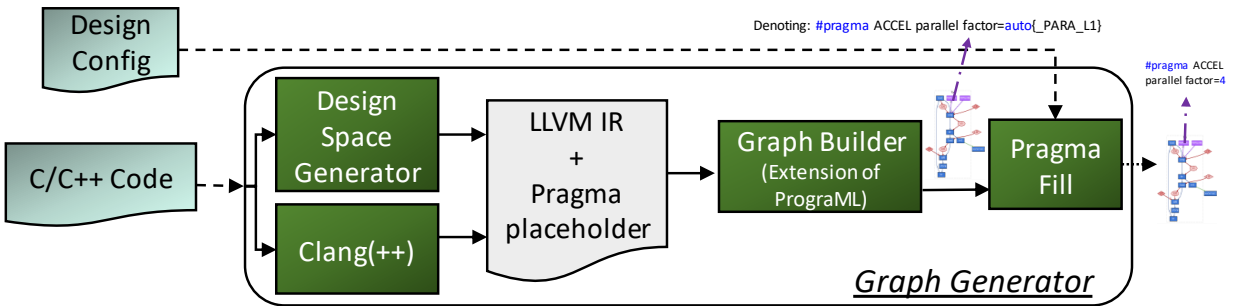


Figure 6.4: Graph generator of GNN-DSE.

model assign a higher weight to the attributes that contribute more to the final prediction. For this matter, the model exploits a *GNN encoder* (Section 6.3.3.1) to update the embeddings. The GNN encoder, then, passes the graph embeddings to a set of MLPs to estimate the outputs (Section 6.3.3.2).

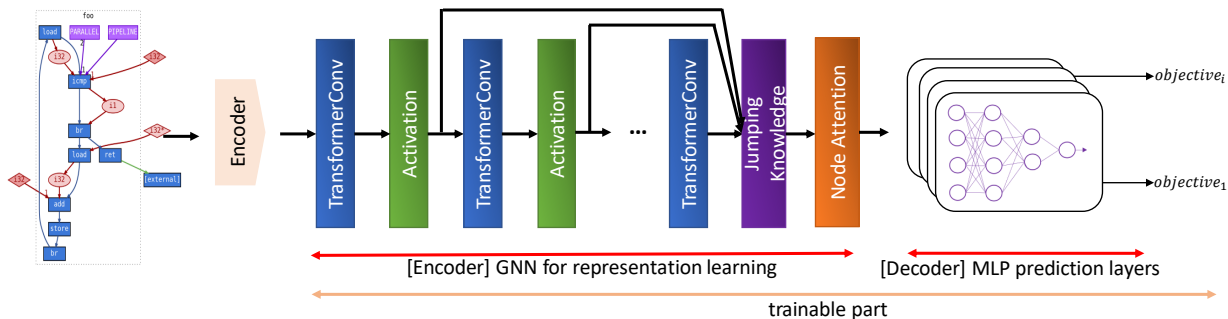


Figure 6.5: The architecture of GNN-DSE’s predictive model.

### 6.3.3.1 GNN Encoder

The GNN encoder assigns  $\mathbf{h}_{\mathcal{G}} \in \mathbb{R}^D$  to a graph  $\mathcal{G}$  via three stages: (1) stacked TRANSFORMERCONV layers to produce node embeddings, (2) a Jumping Knowledge Network for combining the output of different layers to make the final node embeddings with dynamic ranges of neighborhoods, and (3) an attention mechanism to merge the node-level embeddings into a graph-level embedding.

**TRANSFORMERCONV:** We reviewed GCN [KW16] and GAT [Vel+17] in Section 2.3. One drawback of these layers is that they both overlook the edge embeddings. TRANSFORMERCONV [Shi+20], inspired by the Transformer model [Vas+17], is a state-of-the-art GNN architecture, which builds attention coefficients ( $\alpha_{i,j}$ ) for aggregating the neighbors in a different manner than GAT:

$$\alpha_{i,j} = \text{softmax} \left( \frac{(\mathbf{W}_1 \vec{\mathbf{h}}_i)^\top (\mathbf{W}_2 \vec{\mathbf{h}}_j + \mathbf{W}_3 \vec{\mathbf{e}}_{ij})}{\sqrt{D}} \right) \quad (6.5)$$

where  $\mathbf{W}_1$ ,  $\mathbf{W}_2$ , and  $\mathbf{W}_3$  are learnable weight matrices, and  $\mathbf{e}_{ij}$  denotes the embedding of the edge between nodes  $i$  and  $j$ . Including edge attributes is a desirable feature for our task since the edges in our graph representation contain useful information (Section 6.3.2). In addition, TRANSFORMERCONV makes use of gated residual connections when updating the node embeddings that can mitigate potential issues with over-smoothing in the model. Consequently, we adopt TRANSFORMERCONV as the basic building block of our model.

**Jumping Knowledge Network (JKN):** Each layer of a GNN gathers the embeddings

of the first-order neighbors. By adding each layer, the nodes will receive the embeddings from one hop further since their first-order neighbors are now updated with theirs. However, different nodes in the graph may require information from varying neighborhood ranges. For instance, in the graph depicted in Figure 6.3, both the `load` and `add` nodes are influenced by the `pragma` nodes. The `load` node observes the effects of the `pragma` node after three layers, while `add` requires four layers. With the utilization of four layers, the `load` node may accumulate additional noise from new nodes, potentially impacting its embeddings negatively. Consequently, to fully exploit the embeddings produced by different layers of the GNN model, we utilize JKN [Xu+18]. As illustrated in Figure 6.5, JKN considers the output of all layers to selectively choose varying neighborhood ranges for each node. In its simplest form, JKN utilizes max pooling to select the final embeddings of each node from its node embeddings across all previous GNN layers:

$$\vec{h}_i = \max\left(\vec{h}_i^{(1)}, \dots, \vec{h}_i^{(T)}\right) \quad (6.6)$$

where  $\vec{h}_i^{(k)}$  denotes the embedding of node  $i$  after the  $k$ -th layer.

**Node attention-based graph-level embedding generation:** To create a single vector representation for the entire graph, one can simply average all the node embeddings as follows:

$$\vec{h}_G = \frac{1}{N} \sum_{i=1}^N \vec{h}_i. \quad (6.7)$$

where  $\vec{h}_G$  and  $N$  denote the graph-level embedding and the number of nodes in the graph, respectively. However, given the fact that our graph representation contains both the `pragma` nodes and the program context nodes, it is preferable to introduce attention [Li+15] to learn which nodes are more important for the prediction tasks:

$$\vec{h}_G = \sum_{i=1}^N \text{softmax}\left(\text{MLP}_1(\vec{h}_i)\right) \cdot \text{MLP}_2(\vec{h}_i) \quad (6.8)$$

where  $\text{MLP}_1$  maps the node embedding from  $\mathbb{R}^D$  to  $\mathbb{R}$  followed by a global softmax to obtain one attention score per node. The attention scores are then applied to the transformed node embeddings,  $\text{MLP}_2(\vec{h}_i)$ , to obtain the final graph-level embedding. Fig. 6.6 depicts the

graph for a design of the stencil kernel in MachSuite benchmark [Rea+14]. Each node’s circle size is proportional to the attention that its embedding receives in building the graph-level embedding. As we expected, the pragma nodes are among the most *important* nodes. Yet, the model could learn that not all the pragma nodes are equally important here. As the figure suggests, the loop trip count (icmp node and i32 node connecting to it) and other contextual information of the loop determine their importance.

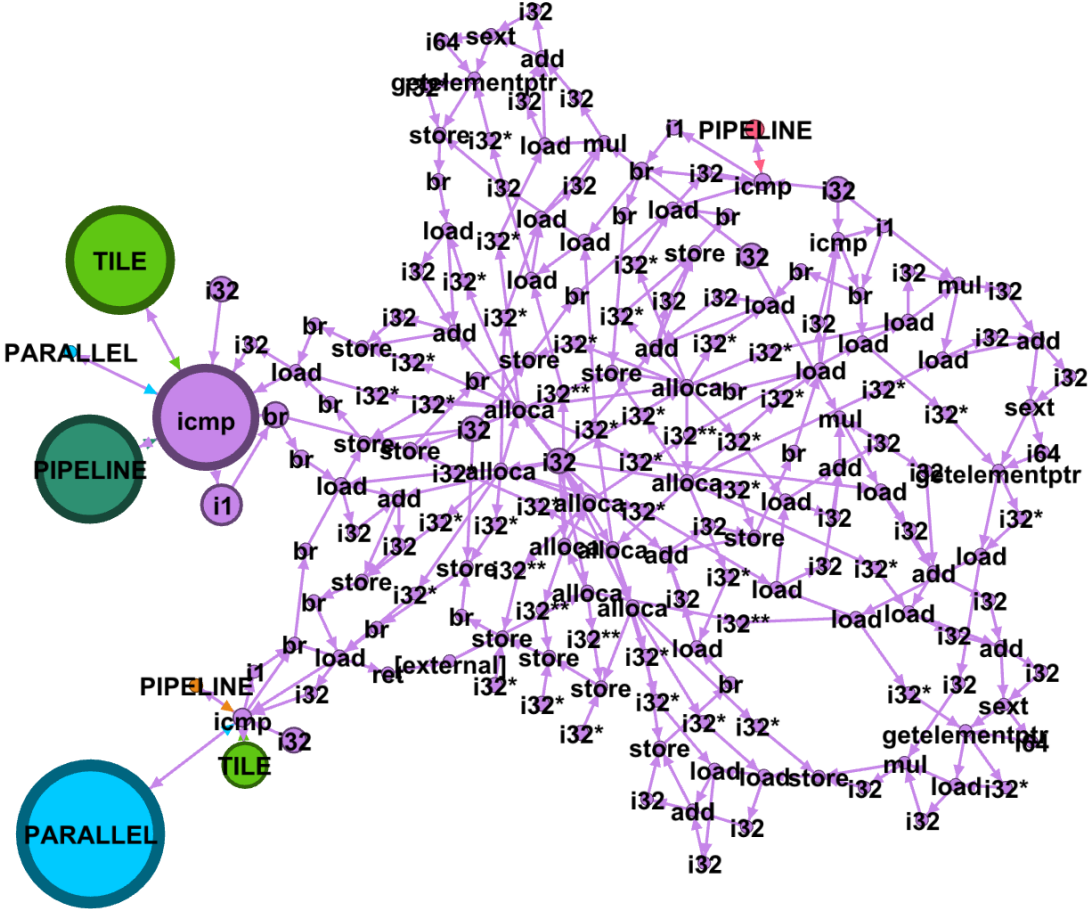


Figure 6.6: Node attention scores of a design of the stencil kernel determined by GNN-DSE’s model. The size of each circle corresponds to its attention score, with larger circles indicating higher attention.

Given that the embeddings are high-dimensional vectors (124/64-D vectors for initial/final embeddings), we employ t-SNE [MH08] for visualization. t-SNE is a powerful technique that can model high-dimensional data by 2-D points in a way that nearby (distant) points

model similar (dissimilar) data. Figure 6.7(a) illustrates the t-SNE plot for the stencil kernel based on its initial embeddings. The graph-level embedding is generated using Eq. 6.7, which computes the average of all node embeddings. Each point denotes a design configuration, color-coded by its latency (cycle counts). Figure 6.7(b) showcases the t-SNE plot when utilizing the graph-level embeddings produced by our GNN encoder. While the initial features exhibit high similarity between design points with significant differences in their latencies, the GNN encoder successfully assigns embeddings to the graphs, ensuring that only designs with similar latencies are clustered together.

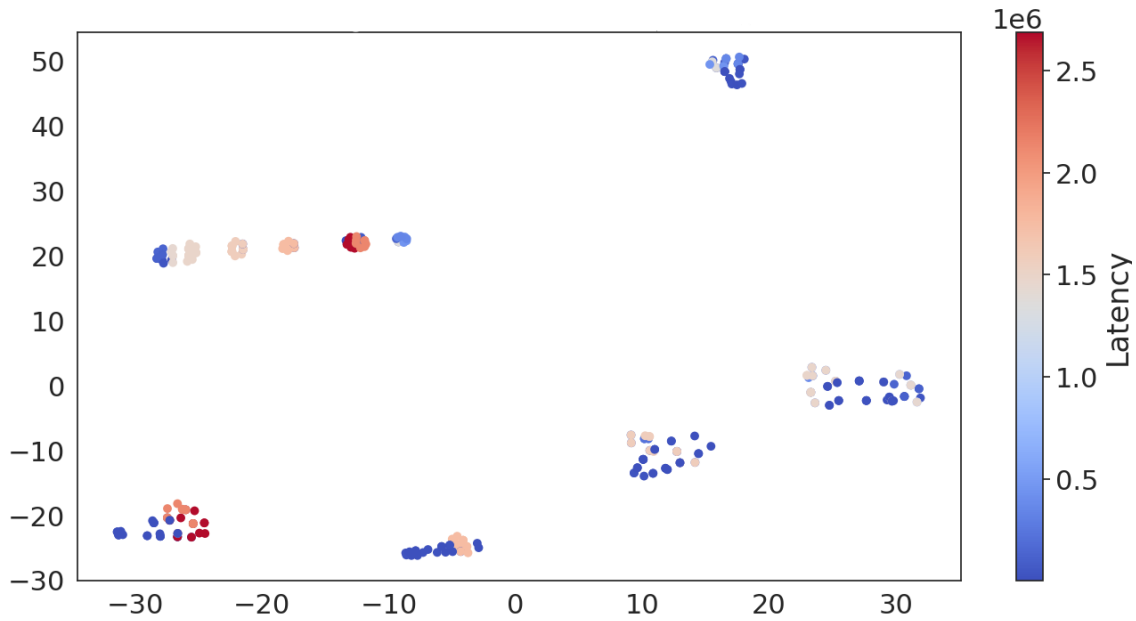
### 6.3.3.2 MLP Prediction Layers

After encoding the graph as a vector, further transformation is needed to perform the final prediction. We have the following learning tasks for assessing a design point:

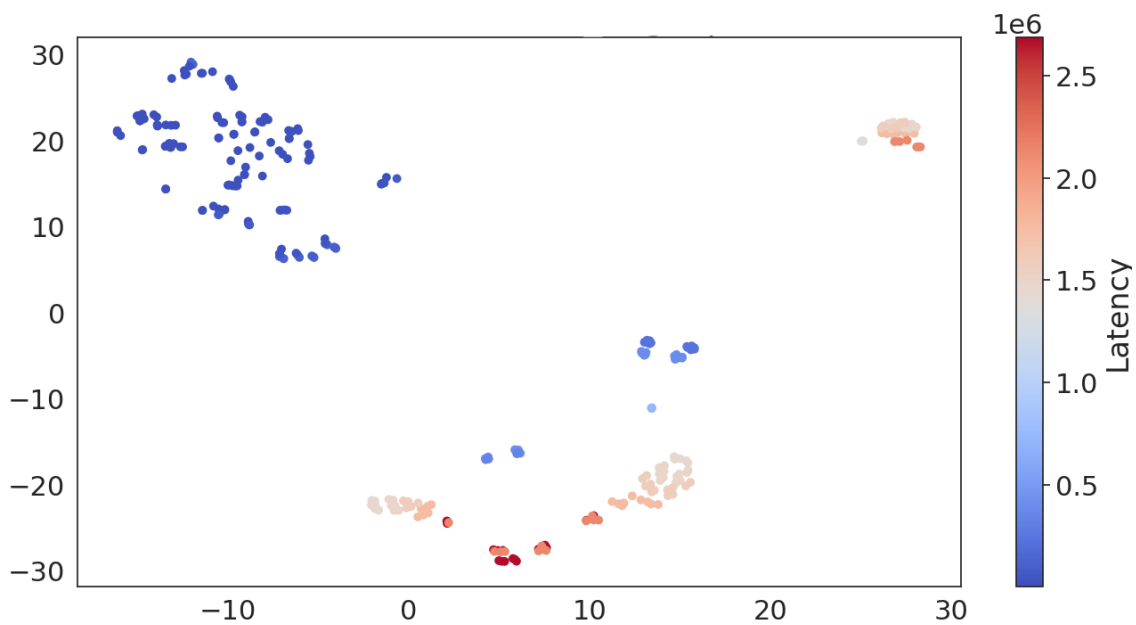
- **Classification task for determining whether a design configuration is valid.**

Invalidity can arise from various factors, including 1) the difficulty for the HLS tool to implement certain combinations of pragmas, leading us to label designs as invalid if they remain unfinished after a certain timeframe. 2) Refusal of the HLS tool to synthesize designs with high parallelization factors. 3) General infeasibility of some pragma combinations. For example, the implementation of coarse-grained pipelining involves applying double buffering, which is non-parallelizable when only a single DRAM port is available. In these instances, the Merlin Compiler issues a warning indicating the inability to apply at least one pragma, which we utilize to designate the design as invalid. While we identified certain invalid cases in developing AutoDSE, not all scenarios may have been accounted for. On the other hand, the learning model can identify patterns within pragma combinations that result in invalid designs, thanks to its exposure to large datasets.

- **Regression task to estimate the design’s objectives.** After determining the validity of a design, we proceed to assess its quality by predicting its cycle count and



(a) Initial Embeddings



(b) Embeddings learned by GNN-DSE

Figure 6.7: t-SNE [MH08] visualization of the design configurations of stencil. Each point represents a different pragma combination with colors indicating its latency value. While the initial embeddings tend to cluster points irrespective of their latency values, GNN-DSE's embeddings achieve a more distinct and informative clustering.



resource utilization, constituting our regression task. The resources considered in this dissertation include Digital Signal Processing (DSP), Block RAM (BRAM), Look-Up Table (LUT), and Flip-Flop (FF). Nonetheless, it is important to highlight that the model can be further trained to predict additional objectives provided there is access to additional labeled data.

For each of these tasks, we exploit MLPs to do the prediction based on the graph-level embedding. Note that our regression task seeks to predict multiple objectives. We can either employ separate models for each objective or use the same GNN encoder as the backbone while adopting different MLP branches for each objective to perform multi-task prediction, as shown in Figure 6.5. In the former approach, each model independently seeks to derive the graph-level embedding necessary to predict its target objective. However, the latter approach is preferable when the objectives exhibit correlation, as they can mutually contribute to the creation of a more refined graph-level embedding.

#### 6.3.4 Design Space Exploration

Once we have an accurate model, we can use it for finding the Pareto-optimal design points. For each design configuration, we first run the classification model to determine its validity. If it is valid, we then run our regression models to predict the design’s objectives. If, for any of the resources, the utilization is higher than 0.8 (80%), we reject that design due to over-utilization. This threshold is empirically set, as exceeding it leads to frequency degradation and mapping difficulties. Subsequently, from the remaining designs, we select the top 10 designs with the lowest latency numbers for evaluation using the HLS tool. Consequently, our model enables us to reduce the number of evaluations with the HLS tool to only 10, rather than invoking it for every design.

Since our models can complete inference for each design point within milliseconds, we can efficiently examine numerous design points. However, when dealing with vast solution spaces, an exhaustive search within a reasonable timeframe might not be feasible. Therefore, we set a time limit for running the DSE and employ a heuristic to prioritize the exploration of the

most promising candidates. As the HLS tools can implement the fine-grained optimizations better, we adapt a BFS-like traversal of the pragmas starting with the inner-most loops to create an ordered list of them. As a result, the pragmas of the inner-most loop levels are evaluated sooner.

If there are multiple pragmas at a loop level, we prioritize `parallel` over `pipeline` over `tile`. If the picked pragma ( $P_p$ ) depends on another pragma ( $P_d$ ) from the same loop level or one loop level further, we move pragma  $P_d$  up in the ordered list. Pragma dependencies are established when defining rules for the solution space to prune invalid pragma combinations, as detailed in Section 5.4.3. For instance, there is always a dependency between the `parallel` pragma of one loop level and the `pipeline` pragma of its upper loop level because `fg` pipelining fully unrolls the sub-loops, making the `parallel` pragma unnecessary. Since there is always a dependency between the `parallel` pragma of one loop level with the `pipeline` pragma of its upper level, for the second-inner-most loop level upwards, this ordering results in evaluating the pipeline pragma before any other optimizations (even before pipelining or tiling of its inner loop). This prioritization is desirable because successful pipelining can lead to either double buffering or full unrolling of the inner loops, both of which are typically preferred over other optimizations on the inner loop. After evaluating this pragma, the same process is repeated for the next loop section until all pragmas are visited.

## 6.4 Evaluation

### 6.4.1 Experimental Setup

We choose our target kernels from the commonly-used Machsuite benchmark [Rea+14], and the Polyhedral benchmark suite (Polybench) [YP]. The initial database is generated as explained in Section 6.3.1, using the AMD Xilinx Virtex Ultrascale+ VCU1525 as the target FPGA synthesized with AMD Xilinx SDAccel 2018.3 (v18) [AMD] and a target frequency of 250 MHz. It consists of kernels with different computation intensities including matrix and vector operations, stencil operation, encryption, and a dynamic programming application

(nw). We then augment this database by including the top design points of three rounds of our DSE as explained in Section 6.3.4, resulting in our final database.

Our model predicts the `latency` in the form of *cycle counts*, and the resource utilization for DSP, BRAM, LUT, and FF. Our framework is deployed and trained using PyTorch [Pas+19] on the NVIDIA Tesla V100 GPU. We use 80% (20%) of the dataset for training (testing), 3-fold cross-validation during training with Adam optimizer [KB14], and a learning rate of 0.001. The reported performance metrics of the model are obtained by executing the inference process on the test set. The initial embeddings, created by concatenating one-hot encoders of various node attributes, consist of 124 features. We train separate models for the classification and regression tasks (defined in Section 6.3.3.2). Each model comprises 6 GNN layers, each with 64 features, followed by different MLP heads for individual objectives. The MLP networks consist of 4 layers, gradually reducing the feature vector from 64 dimensions to a scalar value representing the final objective.

Table 6.1 summarizes the number of pragmas, the total number of design points in the solution space, the total number of configurations in our database, and the number of valid configurations among them for each kernel. It also includes the number of designs after augmenting the database as detailed in Section 6.3.4. In our database, the `latency` is in the range of 660 to 12,531,777 cycles. DSP / BRAM / LUT / FF counts are in the range of 0 / 0 / 913 / 0 to 28,672 / 7,464 / 2,639,487 / 3,831,357 showing a wide range for all the objectives.

## 6.4.2 Model Evaluation

### 6.4.2.1 Preprocessing the Data

We preprocess our data to limit their ranges so that they can contribute to the loss equally. For this matter, we normalize the resource utilizations by dividing them by the available number of resources on the FPGA and apply the following formula for `latency`:

$$T_{latency} = \log_2 \frac{NormalizationFactor}{latency} \quad (6.9)$$

Table 6.1: The statistics for our target kernels including the number of pragmas, the size of the solution space, and the database size utilized for training the GNN-DSE model.

Kernel name	aes	atax	gemm- blocked	gemm- ncubed	mvt	spmv-crs	spmv- ellpack	stencil	nw	Total
# pragmas	3	5	9	7	8	3	3	7	6	-
# Designs configs	45	3,354	2,314	7,792	3,059,001	114	114	7,591	15,288	3,095,613
Initial database (# Total / # Valid)	15 / 15	605 / 101	616 / 149	432 / 149	571 / 180	98 / 35	114 / 60	1,066 / 281	911 / 66	4,428 / 1,036
Final database (# Total / # Valid)	44 / 44	636 / 129	667 / 183	476 / 193	621 / 224	114 / 51	114 / 60	1,098 / 291	982 / 103	4,752 / 1,278

therefore, the model spends more time on reducing the loss for large values of  $T_{latency}$  which corresponds to low latency values, i.e., the high-performance designs. The  $\log_2$  factor is employed to create a more balanced data distribution, compensating for the inherent bias towards low-performance values due to the nature of the problem. While the use of the  $\log_2$  factor may lead to increased prediction losses when reverting the data back to its original range, it does not affect our primary objective of identifying the best design configuration. It is important to note that during the DSE process, we care more about the ranking of the design points based on their latency values rather than their absolute values. After this normalization, the lower range for all the objectives is 0.0 and the upper range is 12.7414 / 4.1900 / 1.7200 / 2.2300 / 1.6600 for `latency` / `DSP` / `BRAM` / `LUT` / `FF`, respectively.

#### 6.4.2.2 Grouping the Objectives

Fig. 6.8 depicts the correlation matrix of the objectives in our database with a correlation coefficient of 1 indicating a perfect positive correlation, -1 indicating a perfect negative correlation, and 0 indicating no correlation. As the figure illustrates, the `LUT`, `FF`, and `DSP` values are highly correlated. The `latency` value ( $T_{latency}$ ) has a weak correlation with each of the `LUT`, `FF`, and `DSP` values but it almost does not correlate with `BRAM`. As a result, the

latency, LUT, FF, and DSP values can help each other to learn a better graph embedding as explained in Section 6.3.3.2. Consequently, we train two separate models to learn the regression objectives: one solely predicts the BRAM utilization, while the other predicts the remaining objectives. Although it is possible to train a single regression model for all objectives, we can get better results by grouping the objectives based on their correlations and training separate models accordingly.

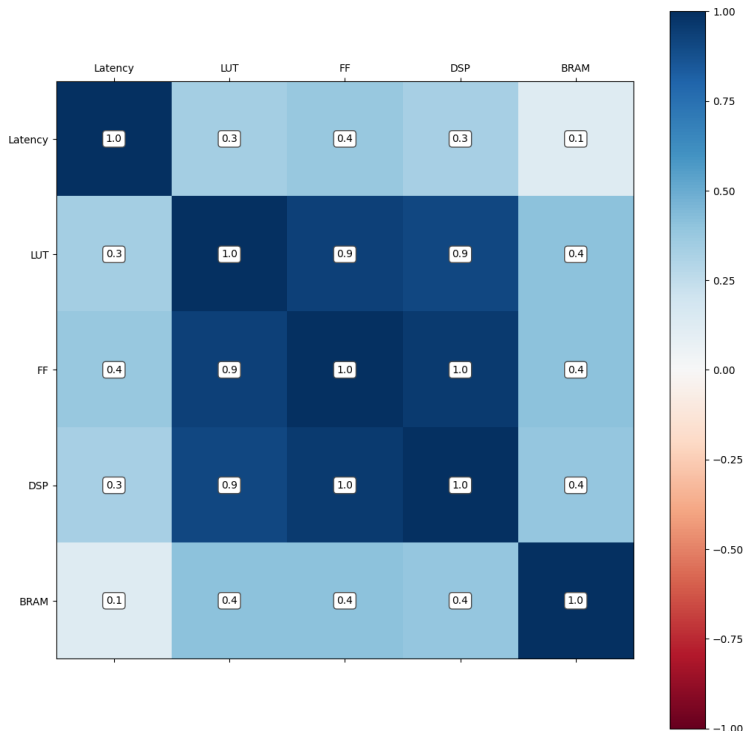


Figure 6.8: Correlation matrix for the database used in training GNN-DSE’s model with 1 (-1) showing a perfect positive (negative) correlation and 0 indicating no correlation.

### 6.4.2.3 Comparative Studies

To test whether our program representation is beneficial for this problem or not, we first test the performance of two other models that only use MLP networks with no considerations for the graph structure. The first one (M1) follows the same approach as in [KC20]<sup>2</sup> and just

---

<sup>2</sup>As the authors were not ready to share their codes at the moment, we reimplemented their model as closely as possible.

Table 6.2: Model evaluation on the test set of our database. RMSE loss is used as the evaluation metric for the regression task. The sum of all losses is reported in the ‘All’ column. For the classification task, the accuracy and F1-score are reported.

Model	Method	Latency (as in Eq. 6.9)	DSP	LUT	FF	BRAM	All	Accuracy	F1
M1	MLP-pragma (as in [KC20])	3.2756	0.5857	0.3115	0.2483	0.3356	4.7567	0.52	0.42
M2	MLP-pragma- program context	2.9444	0.4650	0.2401	0.1349	0.1597	3.9442	0.78	0.40
M3	GCN	1.6825	0.4265	0.1642	0.1277	0.1593	2.5602	0.79	0.51
M4	GAT	1.1819	0.2557	0.1266	0.1009	0.1178	1.7829	0.85	0.68
M5	Tconv	1.1323	0.2540	0.1245	0.0938	0.1231	1.7277	0.85	0.76
M6	M5 + JKN	1.0846	0.2521	0.1112	0.0933	0.0912	1.6324	0.92	0.86
GNN-DSE	M6 + node att.	<b>0.5359</b>	<b>0.1253</b>	<b>0.0762</b>	<b>0.0632</b>	<b>0.0515</b>	<b>0.8521</b>	<b>0.93</b>	<b>0.87</b>

Tconv: TRANSFORMERCONV

uses the pragma settings as the input. The second model (M2) takes all the nodes of the graph with their initial embeddings as the input but does not exploit the GNN techniques for updating the embeddings and rather only uses an MLP. As the results suggest, including the program context in the input is crucial for improving the accuracy of the model since it wants to predict the objectives across applications with different semantics.

Additionally, we assess the effect of our optimizations on the model. We first tested the model’s performance when it used either the GCN, GAT, or TRANSFORMERCONV as the GNN layer with normal averaging to create the graph-level embeddings (M3 to M5). Then, we added the JKN (M6) and replaced the normal averaging with the node attention layer (M7). As Table 6.2 shows, the fact that these models include various design flows (control, data, call, pragma) of the program using a graph structure can help to decrease their loss. The results further demonstrate the effectiveness of our optimizations as explained in Section 6.3.3.1. Specifically, among the GNN models, the TRANSFORMERCONV leads to the best performance by incorporating an attention mechanism for both nodes and edges. The

performance numbers also validate our hypothesis that the model must not only have a way of deciding which program and pragma nodes are more important for the given application but also dynamically adjust neighborhood ranges for each node to update its features effectively.

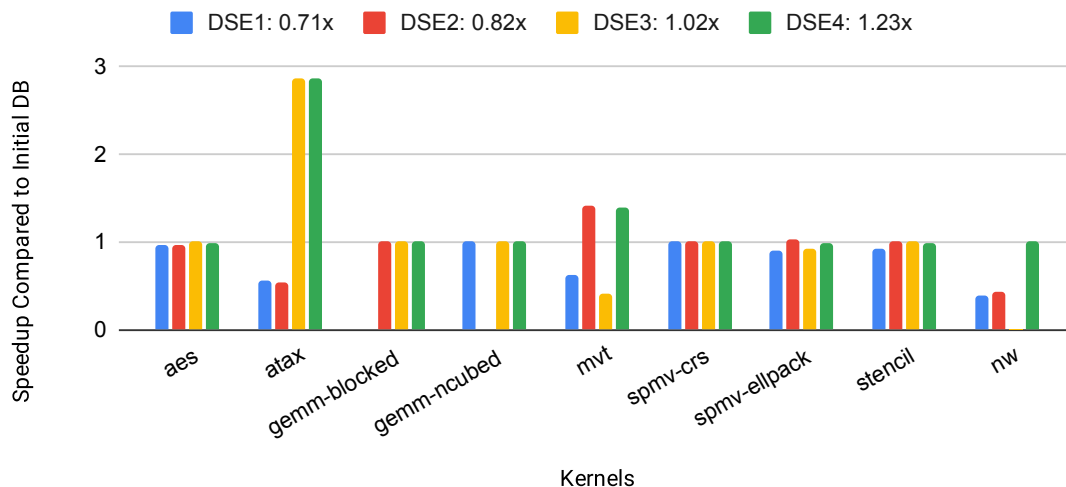


Figure 6.9: Speedup comparison of GNN-DSE relative to the optimal design in the initial database. Following each round of DSE, the top designs are added to the database to retrain the model and improve the discovered optimal designs.

### 6.4.3 Results of Design Space Exploration

Using our models, we are able to run 22 inferences per second. As a result, we can exhaustively search through all the design choices for our target kernels, except for `mvt`, in a few minutes. We adopt the heuristic proposed in Section 6.3.4 to search through `mvt` for one hour. We run the DSE on all the kernels and evaluate their top 10 design points using the HLS tool. Depending on how it performs, we add a various number of design points with their true objectives to the database as explained in Section 6.3.4. Fig. 6.9 depicts the speedup that GNN-DSE could achieve for each kernel compared to the best design in the initial database for different rounds of the DSE. The figure demonstrates that after three rounds of expanding the database, the DSE almost consistently discovers configurations that are almost always better or equivalent to those found by AutoDSE for all kernels. This is

because these newly added data enhance the representativeness of the database, which can help us train a better model capable of identifying equivalent or better configurations in each iteration for the majority of cases. The chart’s legend summarizes the average speedup of all the kernels after each round.

#### 6.4.4 Results on Unseen Kernels

To test whether our tool is extensible to unseen kernels, we have chosen four new kernels from Polybench that were not part of our database but bear similarities to the ones already included: `bicg`, `doitgen`, `gesummv`, and `2mm`. `bicg` is doing two matrix-vector multiplications, `doitgen` multiplies a 3-D tensor with a matrix, `gesummv` has two matrix-vector multiplications and a weighted vector addition, and `2mm` consists of two matrix multiplications. Note that four of the kernels in our database are working with matrix-vector operations, although, in general, they have a different problem size and coding structure. Table 6.3 summarizes the number of pragmas and the design configurations for each of these kernels. Like in Section 6.4.3, we set a time limit of one hour for `2mm`, which has more than  $492M$  design choices. For the rest of the kernels, we exhaustively search through all their configurations which takes less than 2 minutes. For all of them, we then pass the top 10 designs to the Merlin Compiler and run them in parallel to evaluate them. The 4th column of Table 6.3 lists the overall runtime of this process for getting the best design for each kernel.

To measure the quality of top designs generated here, we ran the original explorer of AutoDSE for up to 21 hours (its search for `doitgen` finished after 3 hours). During this time, it explored up to 163 design configurations for each of the cases achieving a maximum speedup of  $350\times$  compared to the design with no optimizations. GNN-DSE could achieve about the same performance (from  $-2\%$  and  $+5\%$  difference with a mean of  $+1\%$ ) but in much less time. Table 6.3 presents a summary of the speedup achieved by GNN-DSE in the total DSE runtime and design synthesis with HLS for each kernel, relative to AutoDSE. The results show that GNN-DSE can accelerate this process by up to  $79\times$  with an average of  $44\times$ . Note that here we are not further fine-tuning our model to adapt to these kernels.



The success of the model depends on how closely the new kernel aligns with our existing database. In the next chapter, we aim to tackle this issue by developing a robust model capable of more systematic adaptation to new environments.

Table 6.3: GNN-DSE’s performance on unseen kernels without further adaptation. The runtime speedup numbers are with respect to AutoDSE, after running it for up to 21 hours. GNN-DSE could achieve about the same performance but in much less time.

<b>Kernel</b>	<b>#pragma</b>	<b>#Design configs</b>	<b>DSE + HLS runtime (m)</b>	<b>#Explored</b>	<b>Runtime speedup</b>
<b>bicg</b>	5	3,536	18	3,536	69×
<b>doitgen</b>	6	179	16	179	11×
<b>gesummv</b>	4	1,581	16	1,581	79×
<b>2mm</b>	14	492,787,501	74	78,676	17×

## 6.5 Conclusion

In this chapter, we developed a push-button framework, GNN-DSE, to build a learning model for predicting the design’s objectives in milliseconds. We proposed a graph-based program representation that includes both the program semantics and the candidate pragmas and implemented a GNN-based model to help us extract the required information for estimating our targets. We exploited our model to optimize the target applications by searching through their different design configurations. The experimental results show that GNN-DSE can build a single model with high accuracy to be used among different domains. They also demonstrate that GNN-DSE is able to not only find the Pareto-optimal designs quickly for the applications in its database but also extend the knowledge it gained from them to optimize new applications from its existing domains. In Chapter 8, we enhance our model-based methodology to create a model with improved adaptability to new environments in a more systematic manner.

## CHAPTER 7

# HLSyn: Benchmark for High-Level Synthesis Targeted to FPGAs

In order to train our models effectively, we require a database containing various HLS designs. However, there are no existing open-source datasets suitable for our needs, so we created our own. We utilize AutoDSE as our primary tool to explore and collect different design points for each application in our training set. As detailed in Section 6.3.1, we enhance AutoDSE’s main explorer with both a hybrid explorer and random search functionalities. This allows us to gather a diverse range of design points, spanning from “bad” to “good”, making the database more representative of the whole space. When a design point is selected by the explorer, it is sent to the Merlin Compiler for evaluation and to generate the design objectives. The outcomes are stored in a shared database, HLSyn, which contains results from various applications along with their corresponding program graph representations. We progressively accumulate results from different applications within this common space, which serves as the foundation for training our model.

The HLSyn dataset includes kernels of intermediate complexity that can be used as building blocks of larger applications. Specifically, we selected 41 kernels from the widely used MachSuite benchmark [Rea+14] and the Polyhedral benchmark (PolyBench)[YP]. They include kernels with different computation intensities including linear algebra operations on matrices and vectors (e.g., BLAS kernels), data mining (`correlation` and `covariance`), stencil operations, encryption (`aes`), and a dynamic programming application (`nw`). For synthesis, we employ three AMD Xilinx HLS tools, SDAccel 2018.3 (v18) [AMD], Vitis 2020.2 (v20), and Vitis 2021.1 (v21) [Xilb], targeting the AMD Xilinx Alveo U200 FPGA

with a frequency of 250MHz. The v20 and v21 datasets are solely collected using AutoDSE after running each of its explorers for a day. Meanwhile, the design points in v18 additionally incorporate results from multiple iterations of GNN-DSE, as explained in Section 6.3.4.

For each design point, we collect the `latency` in terms of cycle counts and resource utilization for DSP, BRAM, LUT, and FF. We normalize the resource usage with the available resources on the board and the latency with  $Norm_{factor} * \log_2(\frac{Latency_{ref}}{latency})$  which we call `perf` with  $Norm_{factor}$  being set to 0.5 and  $Latency_{ref}$  to  $1e7$ . Table 7.1 presents our database statistics. It is important to note that not all combinations of pragmas yield valid design points. Invalid design points include those with excessively long synthesis times (more than 200 minutes) or cases where either the Merlin Compiler or the HLS tool failed to implement them for reasons mentioned in Section 6.3.3.2.

The three versions of the database consist of a total of 41 unique kernels, with 21 kernels existing in all versions. Among the 22 kernels shared between the v18 and v20 databases, the average latency of optimal design found by AutoDSE in v18 is  $5.54\times$  ( $1.36\times$  on the geometric mean) higher than that in v20, suggesting improvements in the heuristics of the HLS tool over time. Similarly, we see an average latency reduction of  $2.68\times$  ( $1.58\times$  on the geometric mean) in the optimal design points found by AutoDSE for the 25 common kernels when we transition from the v20 to the v21 HLS tool.

Table 7.2 provides an overview of the statistics for each kernel in the HLSyn dataset. Notably, certain kernels have two versions with slight variations. The objective is to introduce subtle changes to the code structure, allowing the model to comprehend the impact of such modifications. For example, eliminating a loop dependency may enable greater parallelization, while reordering independent loop nests might not make a difference. These alterations may range from being unimportant, such as defining an unused variable, to significantly influencing the final microarchitecture objectives. Changes affecting the objectives include adjustments to problem size (e.g., arrays and loop trip counts), code modifications to incorporate a reduction tree, and minor alterations to enhance HLS compatibility by removing dependencies. Some kernels undergo both objective-affecting and non-objective-affecting

Table 7.1: Statistics of HLSyn dataset which consists of 41 unique kernels among which 21 of them are shared in all databases. We denote the normalized value of latency as `perf` as it corresponds to the base-2 logarithm of the speedup relative to a reference latency value.

Version	# kernels	#points (All/Valid)	Data range [min – max]					
			Range	latency	BRAM	DSP	LUT	FF
SDAccel 2018.3 v18	35	23,524/ 8,481	Original	[660 – 94,129,840]	[0 – 12,950]	[0 – 57,531]	[0 – 7,739,313]	[0 – 7,558,355]
			Normed	[-1.62 – 6.94]	[0 – 2.99]	[0 – 8.41]	[0 – 6.54]	[0 – 3.19]
Vitis 2020.2 v20	26	11,721/ 4,508	Original	[992 – 1,453,575,296]	[0 – 3,182]	[0 – 45,056]	[0 – 6,611,687]	[0 – 4,411,806]
			Normed	[-3.59 – 6.65]	[0 – 0.73]	[0 – 6.58]	[0 – 5.59]	[0 – 1.86]
Vitis 2021.1 v21	40	45,371/ 10,886	Original	[1,243 – 162,024,512]	[0 – 13,750]	[0 – 89,728]	[0 – 13,288,216]	[0 – 41,661,056]
			Normed	[-2.01 – 6.49]	[0 – 3.18]	[0 – 13.11]	[0 – 11.23]	[0 – 17.61]

changes for experimentation purposes. We suggest that including additional kernels with these types of variations in the future could enrich the database, allowing the model to better understand how code transformations influence the optimal pragma combination. The table also includes the number of nodes and edges corresponding to the hierarchical graph presented in Section 8.2.1.

Table 7.2: Statistics of each kernel in the HLSyn dataset. v18, v20, and v21 refer to AMD Xilinx SDx 2018.3, Vitis 2020.2, and Vitis 2021.1, respectively. Difference columns indicate the description of the application’s functionality, number of nodes and edges in the hierarchical graph representation, number of candidate pragmas, total number of points in the solution space, and the number of points in each version of the database.

Kernel	Description	nodes	edges	pragmas	space	in	in	in
						v18	v20	v21
2mm	2 Matrix Multiplications	380	1,332	14	492M	793	861	1,364

(cont’)

Kernel	Description	nodes	edges	pragmas	space	in v18	in v20	in v21
3mm	3 Matrix Multiplications	526	1,850	21	17T	841	-	1,720
adi	Alternating Direction Im- plicit Solver	1,077	3,880	13	201M	505	-	969
aes	Advanced Encryption Standard	1,784	6,339	3	45	45	43	36
atax-S	Matrix Transpose and Vector Multiplication (small)	249	864	5	2,644	884	891	1,787
atax-M	Matrix Transpose and Vector Multiplication (medium)	241	836	5	9,454	316	-	1,501
bicg-S	BiCG Sub Kernel of BiCGStab Linear Solver (small)	249	866	5	2,975	502	466	727
bicg-M	BiCG Sub Kernel of BiCGStab Linear Solver (medium)	231	802	5	19,536	270	-	1,445
correlation	Correlation Computation	705	2,489	17	513B	1,464	636	1,512
covariance	Covariance Computation	454	1,596	13	970M	-	287	1,589
doitgen	Multiresolution Analysis	266	926	6	179	179	172	179
doitgen-R	Multiresolution Analysis with reduction pattern	252	878	7	595	595	221	595
fdtd-2d-S	2-D Finite Different Time Domain Kernel (small)	555	1,965	16	18B	619	-	1,161
fdtd-2d-M	2-D Finite Different Time Domain Kernel (medium)	555	1,965	16	70B	-	159	1,150
gemm- blocked	Blocked Version of Ma- trix Multiplication	294	1,023	9	2,314	775	440	2,234
gemm- ncubed	Matrix Multiplication	189	655	7	7,152	742	515	1,356
gemm-p-S	Weighted Matrix Multi- plication (small)	235	818	8	409K	1,118	677	1,421
gemm-p-M	Weighted Matrix Multi- plication (medium)	235	818	8	690K	-	149	1,116
gemver-S	Vector Multiplication and Matrix Addition (small)	438	1,532	13	100B	877	641	1,285
gemver-M	Vector Multiplication and Matrix Addition (medium)	447	1,564	13	13B	-	-	428

(cont')

Kernel	Description	nodes	edges	pragmas	space	in v18	in v20	in v21
gesummv-S	Scalar, Vector and Matrix Multiplication (small)	276	968	4	1,543	442	371	1,196
gesummv-M	Scalar, Vector and Matrix Multiplication (medium)	254	890	4	777	292	-	777
heat-3d	Heat Equation over 3D Data Domain	849	3,066	11	71,511	1,620	-	268
jacobi-1d	1-D Jacobi Stencil Computation	212	740	5	2,871	593	-	2,331
jacobi-2d	2-D Jacobi Stencil Computation	412	1,466	11	7M	1,815	-	1,837
md	n-body Molecular Dynamics	409	1,448	3	75	12	-	75
mvt-S	Matrix-Vector Product and Transpose (small)	224	778	8	2M	1,154	1,415	3,132
mvt-M	Matrix-Vector Product and Transpose (medium)	224	778	8	933K	367	-	1,323
nw	Dynamic Programming for Sequence Alignment	479	1,662	6	15,288	1,336	590	45
seidel-2d	2-D Seidel Stencil Computation	329	1,174	7	8,016	1,314	-	184
spmv-crs	Sparse Mat-Vec Mult. w/ Variable-Len. Neighbor	186	644	3	114	114	114	-
spmv-ellpack	Sparse Mat-Vec Mult. w/ Fixed-size Neighbor	168	580	3	114	114	102	114
stencil-2d	A Two-Dimensional Stencil Computation	216	744	7	7,591	1,401	1,010	4,267
stencil-3d	A Three-Dimensional Stencil Computation	375	1,350	5	239	239	239	55
symm	Symmetric Matrix Multiplication	335	1,182	7	50,265	96	100	449
symm-opt-S	Symmetric Matrix Multiplication (HLS friendly) (small)	376	1,324	8	730K	-	285	1,473
symm-opt-M	Symmetric Matrix Multiplication (HLS friendly) (medium)	376	1,324	8	665K	-	-	833
syr2k	Symmetric Rank-2k Operations	293	1,024	8	340K	380	-	526

(cont')

<b>Kernel</b>	<b>Description</b>	<b>nodes</b>	<b>edges</b>	<b>pragmas</b>	<b>space</b>	<b>in v18</b>	<b>in v20</b>	<b>in v21</b>
syrk	Symmetric Rank-k Operations	242	840	8	340K	600	166	775
trmm	Triangular Matrix Multiplication	214	746	7	50,265	176	923	485
trmm-opt	Triangular Matrix Multiplication (HLS friendly)	204	710	7	43,463	934	248	1,651

Table 7.2 Statistics of each kernel in HLSyn dataset. (cont')

## CHAPTER 8

# HARP: Robust GNN-based Representation Learning for HLS

The efficient and timely optimization of microarchitecture for a target application is hindered by the long evaluation runtime of a design candidate, creating a serious burden. To address this issue, in the previous chapter, we introduced a surrogate for the HLS tool by developing a GNN-based model, GNN-DSE. We showed that this model can achieve high accuracy across diverse domains while expediting the optimization process through rapid evaluation of each candidate. This chapter focuses on tackling challenges related to the program’s long dependency range and deeply coupled input program and transformations (i.e., pragmas). We present *HARP* (Hierarchical Augmentation for Representation with Pragma optimization) with a novel hierarchical graph representation of the HLS design by introducing auxiliary nodes to include high-level hierarchical information about the design. Additionally, HARP decouples the representation of the program and its transformations and includes a Neural Pragma Transformer (NPT) approach to facilitate a more systematic treatment of this process. Our proposed graph representation and model architecture for HARP enhance the model’s predictive accuracy, resulting in superior design space exploration outcomes, while also contributing to the development of a more robust and reliable HLS GNN-based representation. This enhancement, in turn, improves its transfer learning capability, simplifying the adaptation process to new environments<sup>1</sup>.

---

<sup>1</sup>All materials are available at <https://github.com/UCLA-VAST/HARP>.



## 8.1 Introduction

In HLS C/C++, the main instruments used to define the microarchitecture are compiler directives in the form of pragmas. An essential research question here is how to incorporate the right combination of pragmas into the code to enhance the QoR. This includes determining the type of required pragmas, where to apply them, and their options, such as the unroll factor or pipelining type. The complexity of this problem arises from the exponential growth in the number of candidate pragmas, the long synthesis time for each design, and the fact that the pragmas do not have a monotonic effect on performance and/or area, which makes it challenging to predict their impact. While the optimal choice of pragmas can yield significant performance improvements in the resulting microarchitecture, such as the 9000× speedup reported in [Chi+22], as we discussed in the previous chapters, identifying the optimal combination of pragmas remains a challenging task [Chi+22; Hua+21].

To address this problem, several previous studies, as outlined in [SW19], have treated the HLS tool as a black box and focused on developing efficient heuristics to explore the solution space more intelligently. In Chapter 5, we introduced AutoDSE as an advanced approach that employs a bottleneck optimizer, emulating the optimization strategies of an expert designer. However, these works suffer from long runtimes as they rely on running the tool directly for evaluating the design configurations, with each run taking minutes to hours. Recognizing the effectiveness of GNNs in the EDA domain [Kha+20; Ren+22; Ust+20; Guo+22; Kou+22], we introduced GNN-DSE in Chapter 6 to speed up the optimization process. GNN-DSE represents the input program along with its pragmas as a graph and utilizes GNNs to capture graph properties and generate vectors for graph/node embeddings. The model then employs a post-processing stage to convert these embeddings into the final objectives we aim to predict, such as performance and/or area. In addition to GNNs, recent advancements in Large Language Models (LLMs) like ChatGPT [Ope24], GPT-4 [Ope23], and AlphaCode [Li+22] make them potential candidates for addressing the HLS optimization problem. However, all of these models take huge computing power to train and none of them

has targeted FPGA accelerator designs with performance optimization in mind. Therefore, for now, GNNs are a more practical solution for the problem at hand and we will consider utilizing LLMs at a later time.

Although GNN-based models have shown promising performance in the EDA domain, there are still some challenges that need to be addressed to make them more effective. One of the main challenges is how to represent the HLS design (C/C++ program with architectural pragmas) in a way that captures all relevant details and makes it informative for the learning model. Additionally, as the design objectives are influenced by both program context and pragmas (i.e., transformations), it can be beneficial to develop a model that can learn the effect of each component separately. In response to these challenges, we propose and implement HARP. To address the first challenge, it includes a novel hierarchical representation of HLS designs. This representation incorporates program semantics and pragmas, while also introducing auxiliary nodes that provide high-level hierarchical information about the design. This graph representation provides a coarsened view of the design, which can assist with coping with the long-range dependencies within the program. In fact, it helps to reduce the average shortest path of our benchmark by a factor of 5. This permits the GNN model to pass the nodes' messages more easily throughout the whole graph. To tackle the second challenge, HARP intends to enhance modeling the pragma optimizations. Hence, we propose two optimizations for decoupling the program representation from its transformations. The first optimization separates the vector representation of the program and pragmas generated by the GNN and employs an autoencoder loop to ensure the pragma vector representation can reconstruct its initial features. The second optimization introduces a Neural Pragma Transformer (NPT), which models pragmas as learnable functions applied to the program representation. This architectural design aligns more naturally with the transformative nature of pragmas. We compare and evaluate these two optimizations in our experiments.

The next challenge emerges when deploying the model in a new environment, where two types of shifts can occur that can lead to different data distributions compared to the training set. First, the *domain shift* arises when the model encounters a kernel that was not

seen by the model during the training process. Second, the *task shift* appears when there is a need to predict a new objective that was not included in the model’s training. Bai *et al.* [Bai+22] discuss how we can leverage meta-learning techniques to tackle domain shift. We aim to adapt to these shifts by simply fine-tuning the model so our goal is to enhance the quality and reliability of representation learning. As a result, we anticipate improved transfer learning capabilities to emerge from these adjustments.

A significant source of task shift occurs when the HLS tool is updated, and the heuristics used in these tools change, which, in turn, impact the design’s objectives. Fig. 8.1 showcases the variations in latency and the rest of the resources’ utilization (including LUT, FF, BRAM, and DSP) for a total of 1145 designs during the transition from SDx 2018.3 to Vitis 2020.2, the HLS tools from AMD Xilinx. The vertical axis represents the objectives obtained using Vitis 2020.2, while the horizontal axis corresponds to the results obtained with SDx 2018.3. To provide a clearer comparison, the outcomes are contrasted with the diagonal line  $y = x$ . The figure shows that all objectives have changed, and the degree of change varies for each objective. Given the cost of regenerating the database and retraining the model, it is preferable to transfer the model using a smaller dataset. Our experimental results show that HARP improves the performance of both the original and transferred models. Even with large datasets, the pretrained model of HARP yields better results after transfer learning. This strong transfer learning capability is due to our novel graph representation and model architecture

In summary, in this chapter, we make the following contributions:

- We propose a novel hierarchical graph representation to combine both a high-level view (combination of C/C++ level and LLVM IR level) and a low-level view (LLVM IR level) of the HLS designs, which can help to reduce the long range of dependencies.
- We design two approaches to decouple the representation of programs and their pragmas, allowing the model to learn the individual impact of each component more effectively.

- We evaluate the effectiveness of our proposed hierarchical graph representation and model architectures for transfer learning by showcasing their capacity to enhance the adaptability of the resulting model to changes in the objectives of HLS designs.
- The experimental results demonstrate that our approach can decrease the prediction loss compared to our previous state-of-the-art (SOTA) GNN-based work by 12-34%.
- When utilized in DSE, HARP delivers an average performance improvement of  $2.13\times$  over the SOTA model-free DSE while operating within a reduced time limit of  $25\times$ . It also outperforms the SOTA model-based approach by  $1.38\times$  on average after transfer learning with limited data.
- Even with large datasets, HARP shows strong transfer learning capability, outperforming the SOTA model-based approach by  $1.26\times$  on average.

## 8.2 HARP Methodology

The objective of our study is to enhance the efficiency of exploring the HLS design space by developing a model capable of predicting the behavior of the HLS tool. Similar to AutoDSE and GNN-DSE, we build our tool on top of the Merlin Compiler (Section 2.1.1). This means that our solution space consists of three types of pragmas (`pipeline`, `parallel`, and `tiling`) which are considered as transformations  $T$  applied to the program (i.e., kernel)  $P$ . In this context, HARP includes a novel hierarchical graph representation, introduced in Section 8.2.1, which facilitates the propagation of graph information throughout the graph. Furthermore, HARP utilizes an advanced model architecture to increase the accuracy of the prediction. Applying traditional machine learning models to determine the objectives may erroneously carry the correlation between program  $P$  and transformations  $T$  in the collected data into the prediction. In contrast, HARP individually learns the impact of each component, as we discuss in Section 8.2.2. In Section 8.2.3, we explain that this attribute can also be advantageous when transitioning to new tasks and domains (kernels) that cause

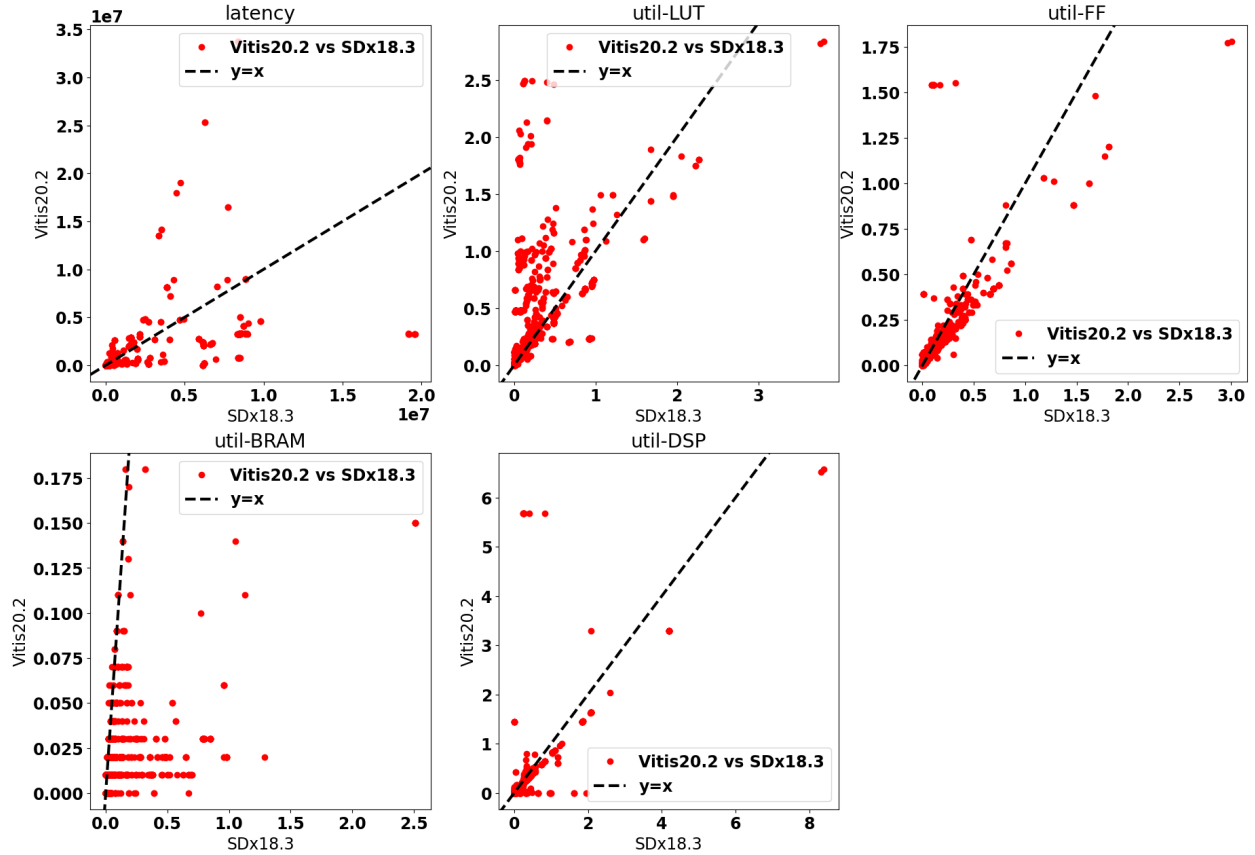


Figure 8.1: The comparison of design objectives when the HLS tool is changed from AMD Xilinx SDx 2018.3 to Vitis 2020.2. Each point on the graph represents a distinct design configuration, plotted against the  $y = x$  line.

shifts in the data distribution. The model can more readily adjust to these shifts, offering an advantage in adaptation.

### 8.2.1 Hierarchical Graph Representation

A common issue in GNNs is that their performance tends to degrade as the number of layers increases, leading to a phenomenon known as over-smoothing. This occurs when repeated graph convolutional layers create too similar node embeddings, thus losing important information about the graph structure. Consequently, GNNs typically have shallow networks, which focus on learning local neighborhoods, leading to limited receptive fields and diffi-

Code 1:

```

for (int i = 0; i < I; i++) {
  for (int j = 0; j < J; j++) {
    for (int k = 0; k < K; k++) {
      ...
    } } }

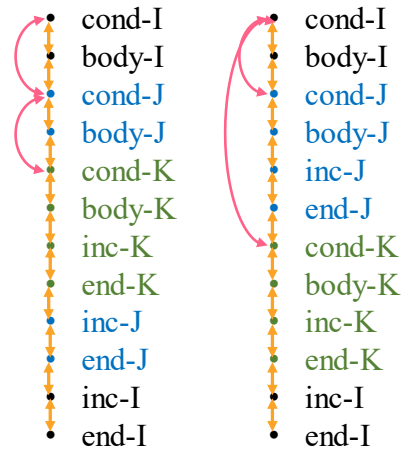
```

Code 2:

```

for (int i = 0; i < I; i++) {
  for (int j = 0; j < J; j++) { ... }
  for (int k = 0; k < K; k++) { ... }
}

```

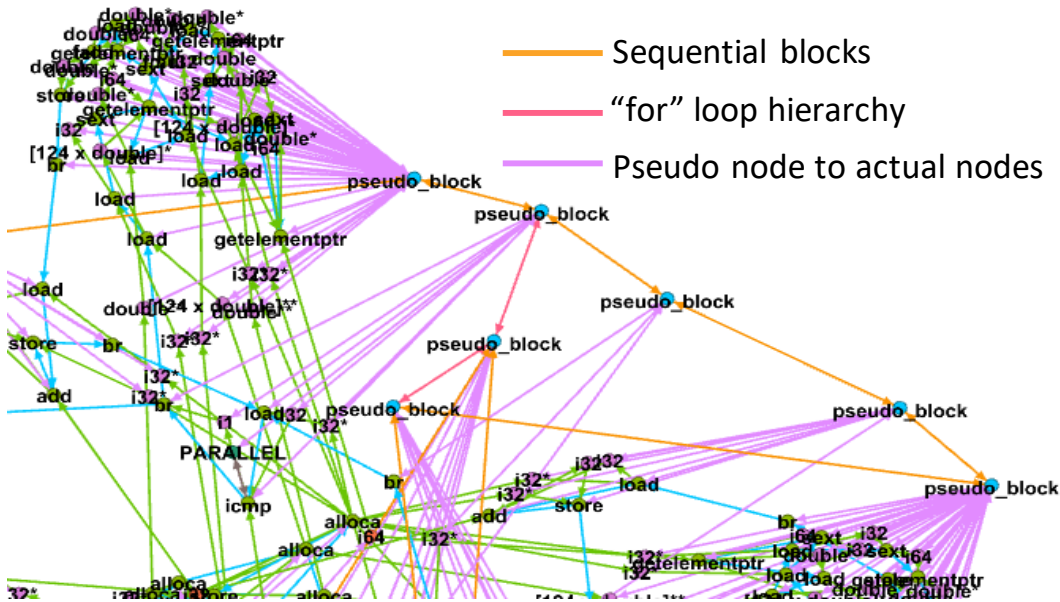


Code 1

Code 2

a) Code snippet 1 and 2

b) Hierarchical structures of sample codes



c) A sample hierarchical graph representation

Figure 8.2: (a) Two sample code snippets; (b) The hierarchical structures of the two sample code snippets, showing only the pseudo nodes and the connections between them; (c) A sample hierarchical graph focusing on demonstrating the pseudo nodes and their connections.

culties in capturing a global view of the graph [Guo+22; LHW18]. This poses a significant challenge in effectively learning programs that are typically characterized by extensive dependency chains, wherein the performance of a given program element depends on the operation of another element located far away in the code.

We aim to tackle this challenge by developing a hierarchical graph representation that integrates both high-level and low-level perspectives of the program, specifically, the HLS design. By introducing nodes in the graph that can establish relationships at various levels, we can coarsen the graph representation to mitigate the impact of the long range of dependencies. To this end, our method incorporates a high-level view that combines the C/C++ code level and LLVM IR [LA04] level and a low-level view that relies solely on the LLVM IR level. We leverage the graph representation provided by GNN-DSE to build the graph from LLVM IR and extend it to incorporate two additional abstraction levels of the program.

In GNN-DSE, separate nodes represent the instructions and their operands (data) in LLVM IR, and they are connected according to the control, data, and call flow of the program. The pragmas are modeled as extra nodes that link to the `icmp` instruction of their respective ‘for’ loop, where the optimization pragmas are applied. To build the second level of representation in the graph, we insert auxiliary nodes (*pseudo nodes*), where each pseudo node corresponds to a distinct LLVM IR block. A block in LLVM IR is a sequence of instructions that end with a terminator instruction, such as a branch, return, or switch. Each basic block in LLVM IR has a single entry point and a single exit point. We define a new node called `pseudo_block` for each block. Fig. 8.2(a) and (b) illustrate two toy examples for showcasing these nodes and the hierarchical structures between them. In LLVM IR, each ‘for’ loop is typically translated into 4 blocks. These blocks consist of the loop condition block, the loop body block, the block for updating the loop iterator, and the final block with a branch instruction to transition to the subsequent block after the loop’s completion. Fig. 8.2 (b) portrays the pseudo nodes assigned to each of these blocks, along with their order and connectivity. The pseudo nodes are linked to one another based on their sequential order. Additionally, the pseudo nodes representing the initial blocks of the ‘for’ loops establish

connections based on their order in the C/C++ code. As demonstrated, each ‘for’ loop is linked to its parent ‘for’ loop (if any) and its first-level children (if any).

Fig. 8.2(c) shows a partial view (due to the space limit) of a graph for a real case. Each `pseudo_block` node has three types of edges. First, it links to all instruction and data nodes within that block. Second, it connects to other pseudo nodes in sequential order, thereby creating the first level of hierarchy. Third, it establishes connections based on the hierarchy level of the ‘for’ loops in the C/C++ code, linking their first blocks according to their hierarchy in the code. This creates the second level of hierarchy in the graph representation. By adopting a hierarchical graph representation that combines high-level and low-level views, our approach can provide a more comprehensive understanding of the design and reduce the complexity of modeling long-range dependencies. This is achieved by decreasing the shortest path between the nodes via the pseudo nodes and their connections, which helps the GNN model to pass messages throughout the graph. For the kernels in our HLSyn benchmark (comprising 41 unique kernels as summarized in Chapter 7), the average shortest path between every two nodes in the graph is reduced from 25.3 (24.5) for the original graph to 5.1 (5) for the hierarchy graph, on average (the geometric mean).

Since we are augmenting the graph with new nodes and edges, we extend the attributes defined in Section 6.3.2. Specifically, we expand the `type` and `flow` attributes to include the new information:

<b>type</b>	0: instruction	1: variable	2: constant value	3: pragma
	4: pseudo node			
<b>flow</b>	0: control	1: data	2: call	3: pragma
	4: pseudo node	5: 1st hierarchy	6: 2nd hierarchy	
	to actual node	(LLVM blocks)	(for loops)	

## 8.2.2 Decoupling Program and Transformation

The input to HLS tools is composed of two primary components that significantly influence the final microarchitecture. The first component is a high-level program description, denoted as P, expressed in C/C++, which defines the semantics and functionality of the DSA to be



designed. The second component is a set of pragmas that include parallelizing, pipelining, and tiling directives, which are applied as transformations (T). These pragmas modify the microarchitecture which in turn affects the performance, power, and/or area of the DSA. The resulting HLS design is a function of both P and T. This work focuses on minimizing the latency  $L(P, T)$  of the design, given the available resource constraints of the FPGA on which the design will be implemented. The resource constraints are determined by the utilization of BRAM, DSP, flip-flops (FF), and lookup-tables (LUT), which are denoted as  $BRAM(P, T)$ ,  $DSP(P, T)$ ,  $FF(P, T)$ , and  $LUT(P, T)$ , respectively, and must be within certain preset thresholds. Thus, the GNN task is to learn the impact of T on P. In our initial effort to build a learning model for this problem, GNN-DSE, we learned a coupled representation vector containing both P and T. In this chapter, we propose to separate the modeling of each component as it allows for a more natural understanding of their individual impacts. In sections 8.2.2.1 and 8.2.2.2, we present two distinct optimizations for effectively implementing such a modeling strategy.

### 8.2.2.1 Separating Vector Representation of Program and Transformation

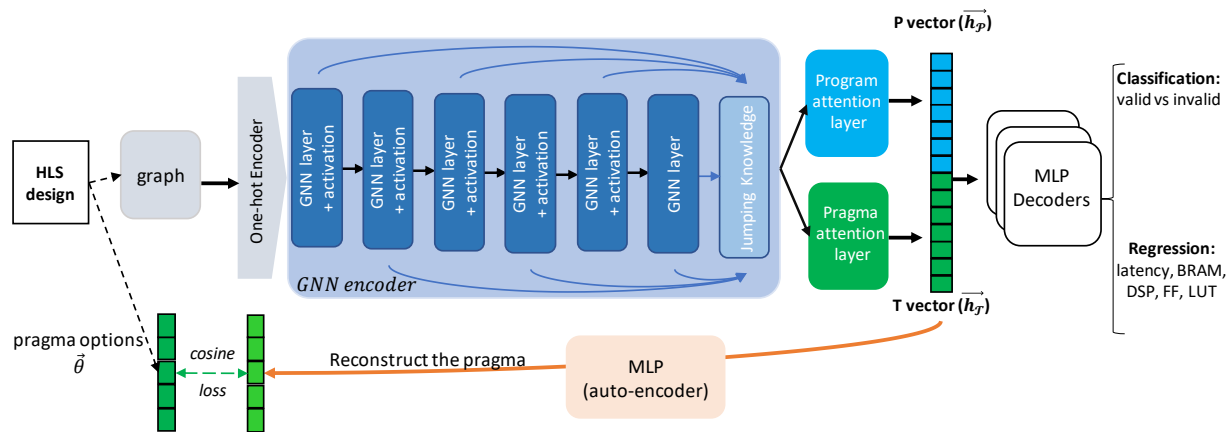


Figure 8.3: Separating the vector representation of the program P and its transformation T. Distinct vectors are generated for each one. A further reconstruction loss with an autoencoder is used to enhance the influence of pragmas on the T vector.

Fig. 8.3 depicts the model architecture for separating the vector representation of P and T. As in GNN-DSE, we start with encoding the node and edge attributes using one-hot encoders. The graph is then passed through a series of GNN layers and a JKN. This JKN enables the model to dynamically adapt the neighborhood range for each node. We demonstrated the effectiveness of this technique in Chapter 6. Once the GNN encoder is finished, the nodes have seen the program and the pragma structure, and their embeddings are produced based on that. We employ two attention layers to build the final P and T vectors. The attention layer is responsible for learning an attention (importance) score for each node and applying a weighted addition accordingly on their embeddings. The *program attention layer* merges the nodes corresponding to the program context ( $N_{\mathcal{P}}$ ) while the *pragma attention layer* pools only the pragma nodes ( $N_{\mathcal{T}}$ ). In addition to separating the learning of the program and its transformation, this architecture helps to amplify the effect of the pragmas in predicting the final objectives. Formally, the computation here can be modeled as:

$$\forall \mathcal{V} \in \mathcal{P}, \mathcal{T} \quad h_{\mathcal{V}} = \sum_{i \in N_{\mathcal{V}}} \text{softmax} \left( \text{MLP}_1(\vec{h}_i) \right) \cdot \text{MLP}_2(\vec{h}_i) \quad (8.1)$$

where  $\vec{h}_i$  represent the embedding vector of node  $i$ ,  $\mathcal{V}$  can denote either the program context  $\mathcal{P}$  or the transformation context  $\mathcal{T}$  and  $N_{\mathcal{V}}$  designates the set of nodes that are in the context of  $\mathcal{V}$ .

To make the T vector ( $h_{\mathcal{T}}$ ) more meaningful, we utilize an autoencoder [HS06] structure. Autoencoders are designed to reconstruct part of the input data given its context. We use them to make sure  $h_{\mathcal{T}}$ , which summarizes the pragmas, can reconstruct the input pragmas stored as a vector  $\vec{\theta}$ . This can help us increase the effect of a change in the input pragma options in the final vector representation. The autoencoder architecture consists of an MLP encoder and an MLP decoder, which takes as input  $h_{\mathcal{T}}$  and aim to produce  $\vec{\theta}$ . Despite the varying number of pragmas in different programs (HLS designs), we employ a fixed-sized vector for  $\vec{\theta}$  to enable training a shared MLP decoder for all programs. In cases where programs have fewer pragmas, the remaining elements of  $\vec{\theta}$  are filled with zeros. The total

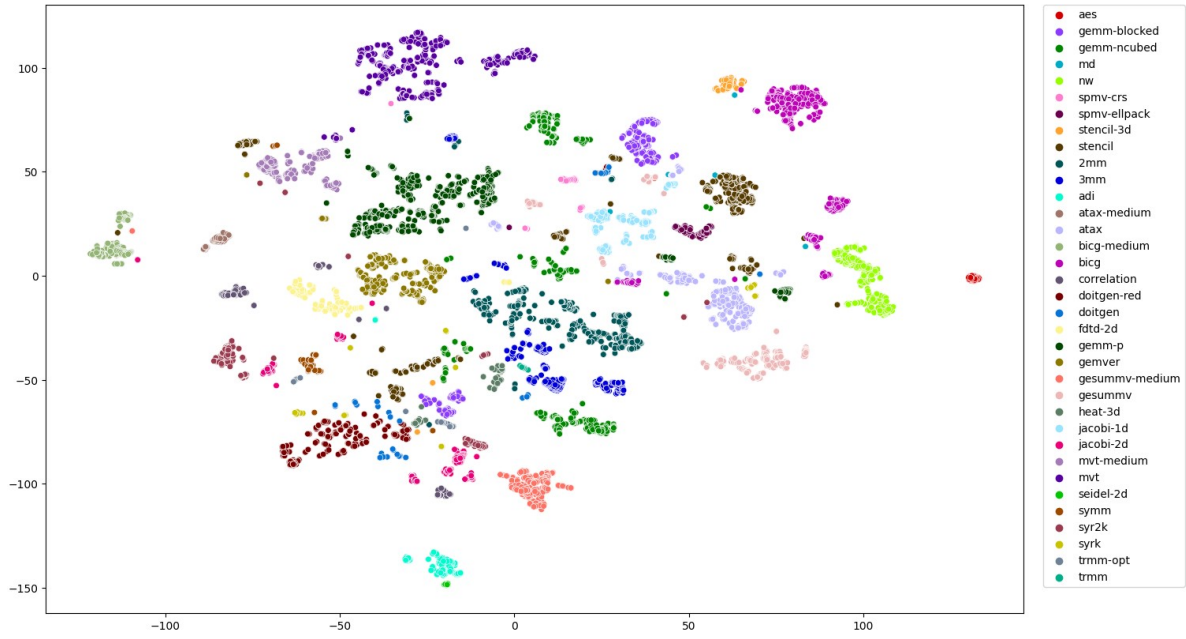
loss of the model would be calculated as:

$$l_T = l_{CE}(\mathbf{AE}(h_{\mathcal{T}}), \vec{\theta}) + \sum_{o \in obj} l_{MLP}(\mathbf{F}_o(P, T), H_o(P, T)) \quad (8.2)$$

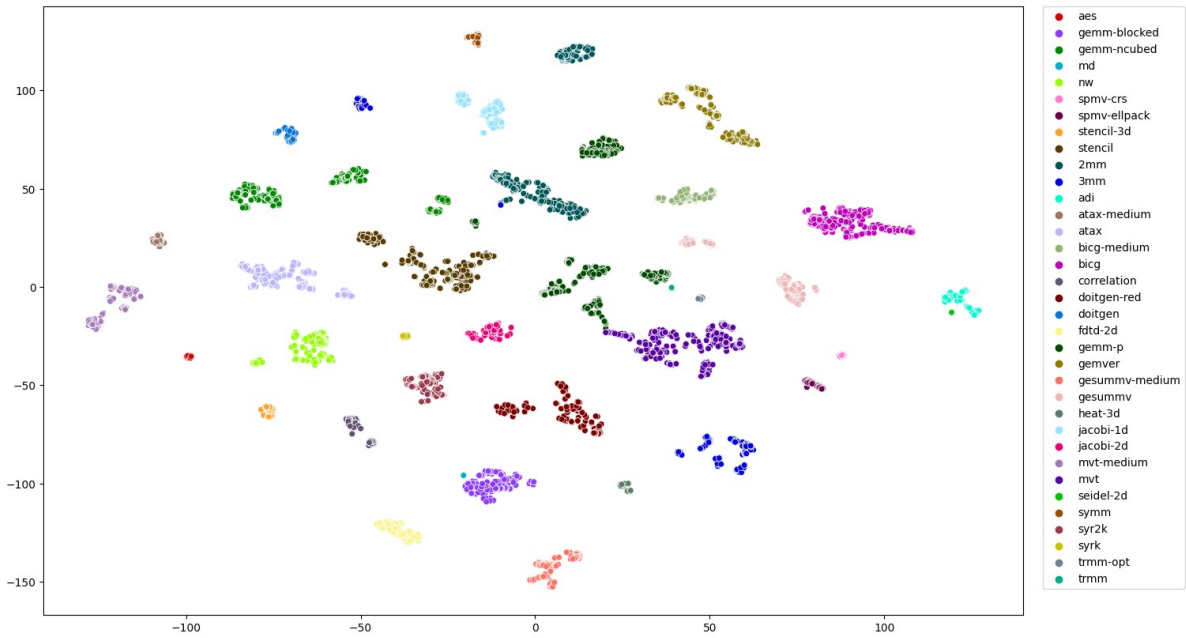
where  $l_{CE}$  and  $l_{MLP}$  denote the cosine error and prediction error of the MLP decoder, respectively.  $l_{MLP}$  is calculated as cross-entropy loss for the classification task and mean squared error for the regression task.  $\mathbf{AE}(h_{\mathcal{T}})$  is the generated vector from the autoencoder.  $\mathbf{F}_o(\cdot)$  and  $H_o(\cdot)$  show the predicted value and the ground-truth value (HLS results) for objective  $o$ , respectively.

The t-SNE [MH08] visualizations of the embeddings generated by GNN-DSE and our new proposed P vector ( $h_{\mathcal{P}}$ ) are compared in Fig. 8.4. t-SNE is a method that is capable of representing data with high dimensionality through 2-D points, where data points that are close together in the 2-D space are indicative of similar data, and those far apart indicate dissimilar data. Each point in the figure represents a different design point from a different kernel and is color-coded based on its kernel name. The embeddings generated from GNN-DSE are interleaved when labeled by kernel name, whereas our proposed model successfully clusters the embeddings based on the kernel they belong to. To quantitatively assess the improvement in clustering, we compute the Euclidean distance between every pair of embeddings for a given kernel and measure the maximum and average distance among them. The average (across kernels) of the average and maximum distance using  $h_{\mathcal{P}}$  decreases by  $3.7\times$  and  $2.5\times$  respectively, compared to the embeddings generated by GNN-DSE. These findings highlight the effectiveness of  $h_{\mathcal{P}}$  in understanding the program scope and its semantics.

Furthermore, Fig. 8.5 shows the t-SNE visualization of  $h_{\mathcal{T}}$  for a random kernel from the MachSuite benchmark [Rea+14], `gemm-blocked`, which is color-coded based on the `perf` value. The `perf` value, defined in Chapter 7, represents the log speedup of the design points to a reference latency value. In order to better illustrate the effectiveness of  $h_{\mathcal{T}}$ , we compare it with visualization using pragma options  $\vec{\theta}$ . As the figure shows, there are some points that are similar to each other when they are compared with their pragma options  $\vec{\theta}$  but have large differences in their `perf` value. This is expected as a small change in the pragma options (for example, changing the pipelining from coarse-grained to fine-grained) can have



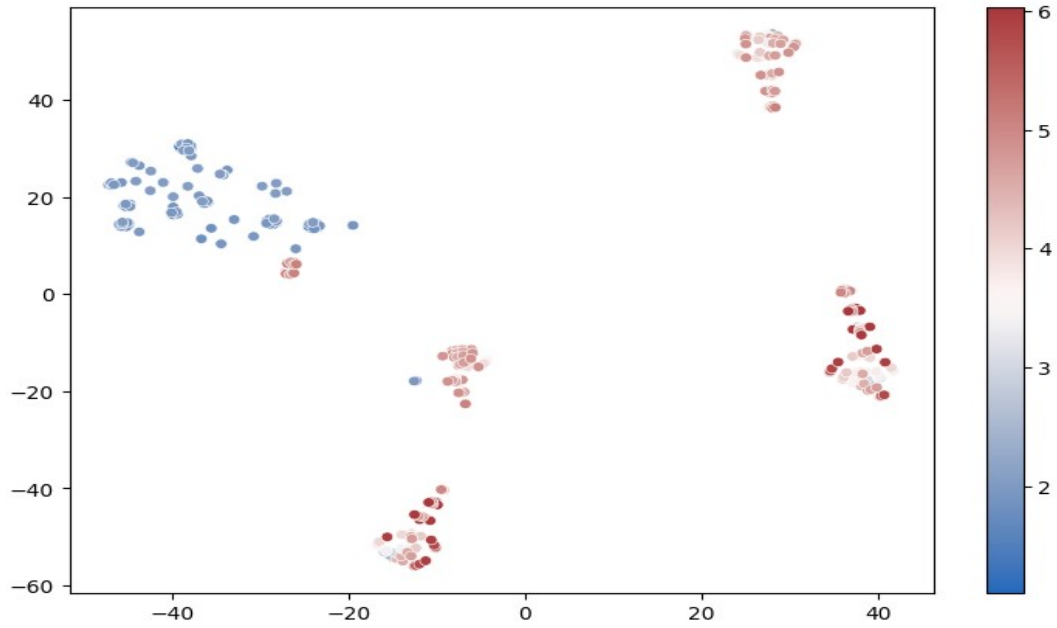
(a) GNN-DSE graph-level embedding.



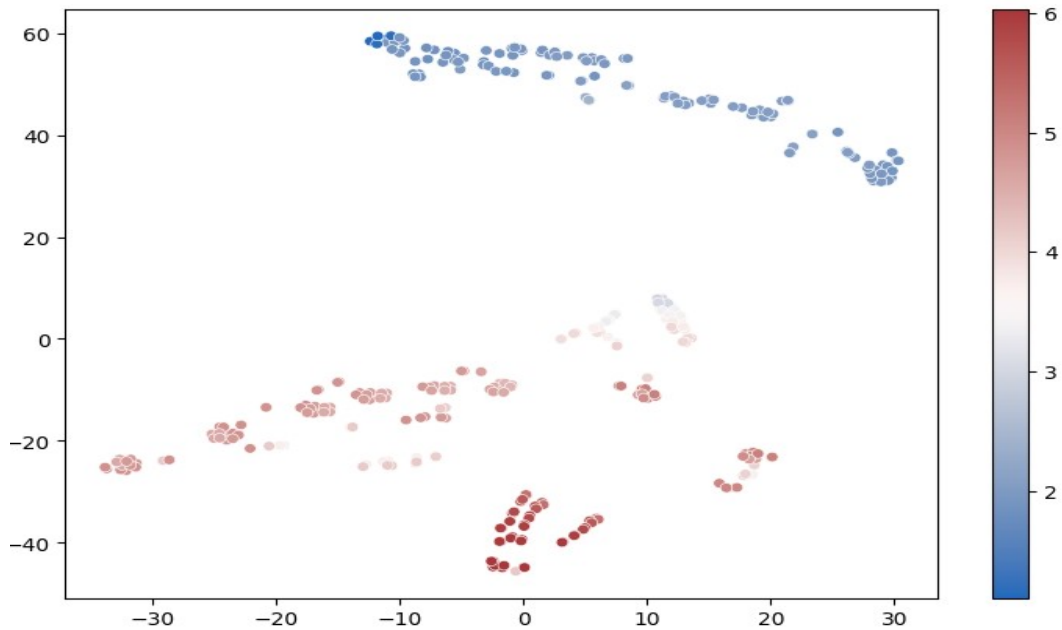
(b) HARP P vector ( $\vec{h}_P$ ) representation.

Figure 8.4: t-SNE visualization of the generated embeddings that are color-coded by the kernel name.

a significant effect on the resulting microarchitecture and the final performance. However,  $\vec{h}_T$  can effectively capture the impact of transformations, leading to improved clustering of



(a) The input pragma options ( $\vec{\theta}$ ).



(b) HARP T vector ( $\vec{h}_{\mathcal{T}}$ ) representation.

Figure 8.5: t-SNE visualization of the T vector compared to input pragma options that are color-coded by the performance value (log of speedup). Warmer colors indicate higher performance (lower latency).

design points. This helps us further in distinguishing the design points within the same program.

Therefore, our proposed P and T vector representations together provide a better understanding of the program scope and the transformations that are applied to it. For the final prediction, we concatenate these two vectors and pass them to MLP decoders. Like GNN-DSE, we define two types of tasks, classification to predict whether a pragma candidate creates a valid design or not, and regression to predict the latency and resource utilization. Experimental results (Section 8.3.2) reveal that this model can decrease the loss by 10-23%.

### 8.2.2.2 Modeling Pragas as Function Transformation via Neural Pragma Transformer (NPT)

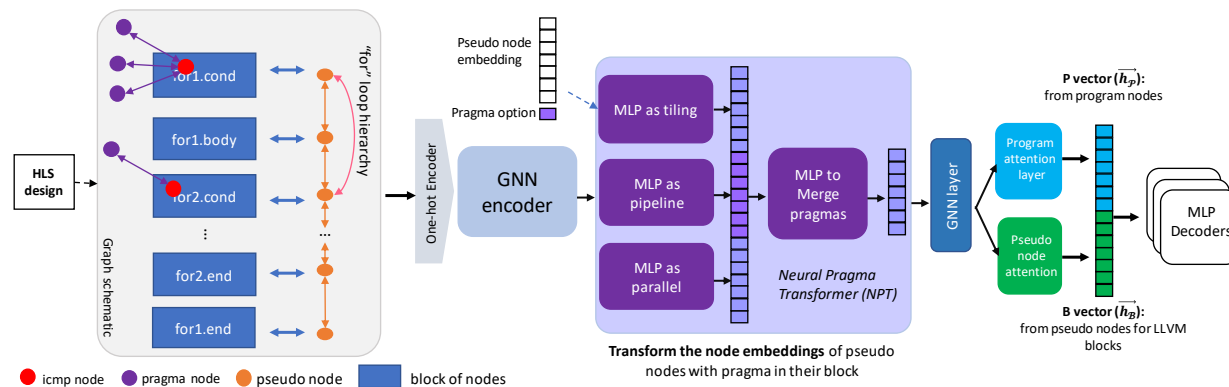


Figure 8.6: Modeling pragmas as function transformations using NPT: each pragma type is modeled as a learnable MLP which takes in the embeddings of the pseudo node of the pragma block along with the pragma option. A second level of MLP is used to merge the results.

The primary goal of this study is to predict the objectives of an HLS design after applying a specific transformation T to its behavioral description in program P. These transformations are applied in the form of pragmas that alter the microarchitecture of the target application (Section 2.1.1). For example, the parallel pragma duplicates statements within a loop and creates parallel units to process them simultaneously. Therefore, it is appropriate to model

the pragmas (T) as functional transformations that are applied to the program P, which is represented as a graph. Our model for achieving this goal is illustrated in Fig. 8.6.

The model in Section 8.2.2.1 can work with both the original graph and the hierarchy graph. However, this model needs to be applied to the hierarchical graph. Since the actual graphs are too crowded to visualize ( $\sim 400$  nodes on average), a schematic of the hierarchical graph is presented in Fig. 8.6. The blue boxes represent the LLVM blocks, and only one representative node, namely, the `icmp` node, is depicted inside each box, which is connected to the `pragma` nodes. Each box has a corresponding pseudo node, and these pseudo nodes are connected with the hierarchical structure of the program as described in Section 8.2.1.

A GNN encoder with the same architecture as the one shown in Fig. 8.3 encodes the graph. This encoder is intended to focus on the program’s structure along with the domain of its pragmas. Therefore, all pragma nodes have the same attribute as their default option (1 for `parallel` and `tiling` pragmas. ‘off’ for `pipeline` pragma). As a result, unlike in Section 8.2.2.1, the input one-hot encoder does not encode the pragma options. After the GNN encoder has finished, the nodes have gained insight into the program’s semantics in addition to the domain of the pragmas. We then utilize the learnable NPT module to apply pragmas as function transformations. NPT takes the embedding of the pseudo nodes that contain a pragma node in their block as the input and transforms it based on the type of the pragmas and their actual options. Each pragma type is modeled using a learnable MLP that accepts the node embedding and the pragma option as input and transforms the node embedding. If a pragma type is not present in the block, the default option is employed. The results of the MLP transformation for each pragma type are concatenated, and another MLP is used to learn their interactions and transform the concatenated result to the final node embedding of the corresponding pseudo node. After this stage, the pseudo nodes have acquired knowledge of the program semantics, the pragma domains, and their options. A further GNN layer is utilized to propagate the new information (pragma options) to the rest of the program nodes via message passing.

Once the final node embeddings have been generated, they are pooled to create the

graph-level embedding. Consistent with the approach used in Section 8.2.2.1, two vectors are generated with the attention mechanism in Eq. 8.1 to represent the program P and transformation T separately. Note that in this architecture, the transformations T are applied to the pseudo nodes. P vector ( $\vec{h}_P$ ) is generated by pooling the program nodes and B vector ( $\vec{h}_B$ ) is the result of pooling the pseudo nodes, which are the primary sources containing the pragma information. As before,  $\vec{h}_P$  and  $\vec{h}_B$  are concatenated, and the result is passed through MLP decoders to predict the final objectives. Experimental results (Section 8.3.2) reveal that this model can decrease the loss by 12-34% compared to GNN-DSE.

### 8.2.3 Transfer Learning

When faced with new programs or tasks, the data distribution may shift from the training data distribution, making the prediction model unreliable. In Section 8.1, we discussed one form of task shift that occurs when the HLS tool, used for synthesizing and implementing the design, changes. In such cases, collecting all the labels again, including the latency and resource usage, and retraining the entire pipeline can be time-consuming. To address this issue, we aim to adapt to the new environment using less labeled data by leveraging transfer learning. Specifically, we use the model trained on the previous version of the tool and fine-tune it to adapt the predictions to the labels of the new version of the tool.

Transfer learning [Zhu+20] can be viewed as a form of task adaptation, where knowledge learned from a source task is transferred to a target task with limited labeled data. In our case, the source task refers to the previous version of the HLS tool, where a large amount of labeled data is available, and the target task refers to the new version of the tool, where limited labeled data is available. Additionally, as we transition to the new tool version, we introduce new kernels to contain domain shifts as well. We speculate that one important requirement for the success of transfer learning in this context is that the model must have a clear understanding of the components that impact optimization results, namely the program semantics and the impact of transformations. By distinguishing between these two components, the model can better update its predictions when the data distribution



shifts.

Fig. 8.7 presents the correlation matrix of the design objectives for the common design points between versions v18 (SDx 2018.3) and v20 (Vitis 2020.2) within our HLSyn benchmark (introduced in Chapter 7). The correlation matrices between versions v18 and v21, as well as between versions v20 and v21, follow the same trend and are included in Appendix A.2. In Section 6.4.2.2, we explained that a correlation coefficient of 1 indicates perfect positive correlation, -1 indicates perfect negative correlation, and 0 indicates no correlation. Notably, we observe a high correlation not only among some objectives within the same version but also among the same objectives across different versions. This suggests potential benefits for pretraining a model on one version of the tool and fine-tuning it on another.

When the pretrained model’s target and the fine-tuned model’s target are highly correlated, it suggests that the knowledge encoded in the pretrained model is relevant to the fine-tuning task. This can accelerate the learning process for new tasks. This correlation suggests that the pretrained model has already acquired representations or features beneficial for predicting the fine-tuned target. During fine-tuning, the weights of the pretrained model are adjusted to better align with the new target task. A strong initialization from the pretrained model can facilitate quicker convergence during fine-tuning and potentially yield improved performance. Moreover, the risk of overfitting during fine-tuning is mitigated, as the model learns generalizable features or patterns relevant to both tasks, aiding in regularization and preventing the fine-tuned model from excessively fitting the training data. Furthermore, fine-tuning may demand less labeled data to achieve satisfactory performance compared to training the model from scratch, as the pretrained model has already acquired useful representations from a potentially larger dataset.

Our experimental results demonstrate that indeed our graph representation and model architecture are effective in improving the model’s performance after transfer learning. Specifically, our approach achieves significant performance gains in terms of both the model accuracy and the DSE results when fine-tuned on the limited labeled data (in this case, about

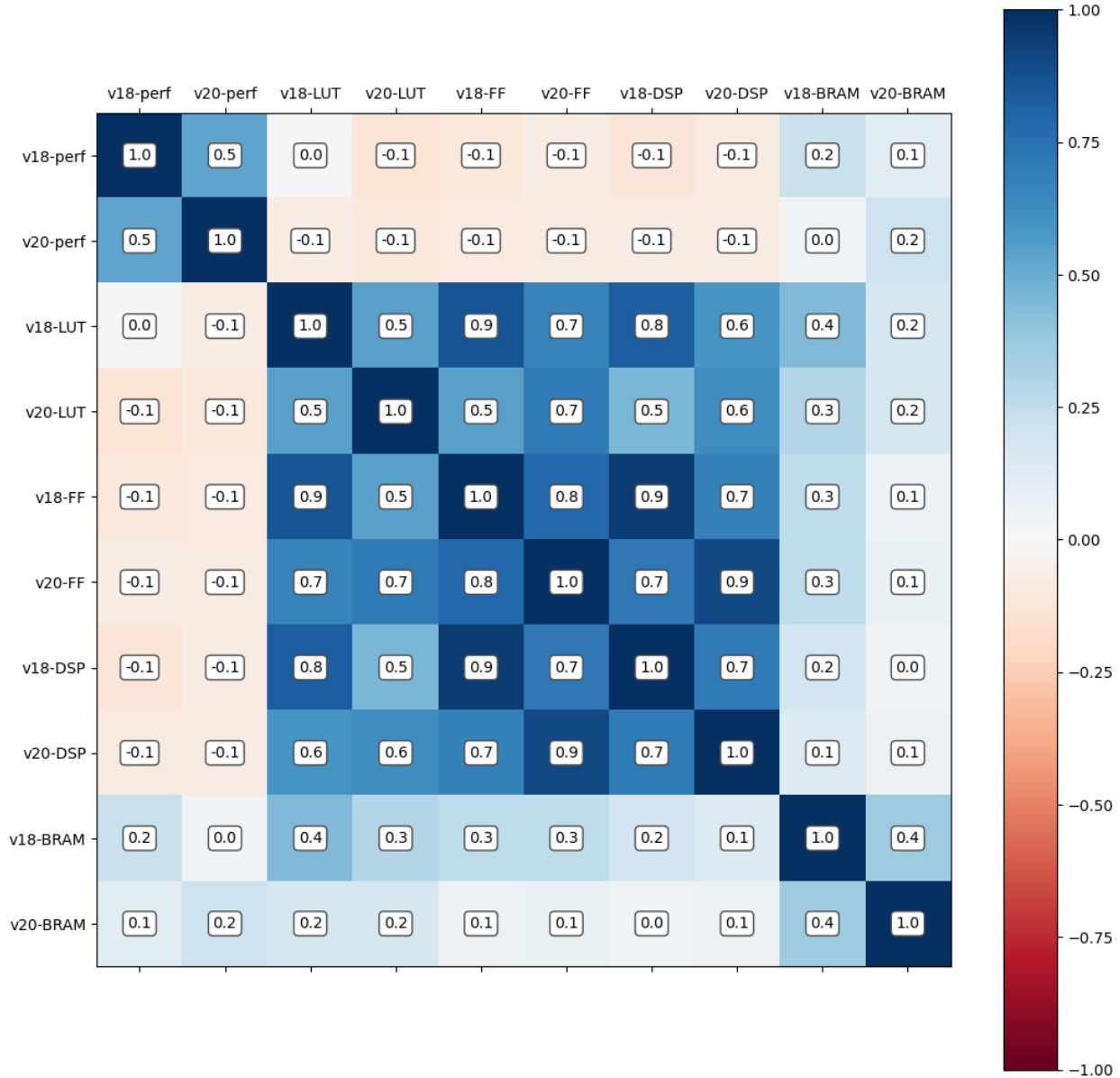


Figure 8.7: Correlation matrix of the design objectives resulted from AMD Xilinx SDx 2018.3 (v18) and Vitis 2020.2 (v20). The data is taken from our HLSyn dataset (Chapter 7).

half the size of the previous dataset) from the new version of the tool. We also demonstrate that even in scenarios with ample data availability, transfer learning through fine-tuning remains highly effective, further enhancing our performance metrics. Importantly, our HARP design exhibits significantly greater capability in transfer learning compared to GNN-DSE, underscoring the importance of the optimizations we implemented in the graph representa-

tion and model architecture. These optimizations enable better learning of each problem component, ultimately leading to a robust representation.

## 8.3 Experimental Results

### 8.3.1 Experimental Setup

Our framework is implemented and trained using PyTorch [Pas+19] on NVIDIA Tesla V100 GPUs. We use our HLSyn dataset, introduced in Chapter 7, for training. The dataset is split into 70% for training, 15% for validation, and 15% for testing. We employ the Adam optimizer [KB14] with a maximum learning rate of 1e-3, which is linearly increased from zero over the first 2000 updates and then annealed to zero using a cosine schedule. Separate models are trained for classification and regression tasks. The classification/regression model is trained for 200/1500 epochs (taking less than 10h with 1 GPU) for the first version of the database (v18) and 200 epochs for transfer learning to the v20 database. We pick the model with the lowest validation loss and report its performance on the test set. The initial embeddings have 154 features. We utilize 6 TRANSFORMERCONV [Shi+20]) with a feature dimension of 64 for the GNN encoder. The final objective prediction is performed using 4 MLP layers (one MLP network for each objective). The GNN and MLP layers are followed by ELU activation [CUH15]. To mitigate overfitting, we apply dropout with a probability of 0.1 to the neurons in the GNN layers. The NPT module utilizes two layers for each of the MLPs. The autoencoder is an MLP with 4 layers that gradually reduces the feature size from 64 to 8 and then increases it to 21 which is the dimension of the vector containing the pragma options. When fine-tuning the model for transfer learning to the v20 database, we freeze the first GNN layer and update the rest of the network.

### 8.3.2 Model Accuracy

We conducted a series of experiments to evaluate the effectiveness of various components of our approach. Table 8.1 includes the model definitions that were tested in our ablation

Table 8.1: Descriptions of models indicating the graph representation type and network architecture.

Name	Graph	Model
GNN-DSE	Original	Coupled P&T (Chapter 6)
M2	Original	Separate P&T
M3	Hierarchy	Separate P&T
M4	Hierarchy	Sequential pragma as Neural Pragma Transformer (NPT)
M5	Hierarchy	Parallel & merge as Neural Pragma Transformer (NPT)
HARP	Hierarchy	Parallel & merge as Neural Pragma Transformer (NPT) + post GNN layer

Table 8.2: Total Root Mean Squared Error (RMSE), Mean Absolute Error (MAE), and perf ranking (tau) of the models. For RMSE and MAE, the lower the better. For tau, the higher the better. The percentage of difference is measured with respect to GNN-DSE (Chapter 6).

Table 8.1 contains the definitions of the models.

Name	v18 database			v20 database					
	Train from scratch			Train from scratch			Fine-tuned from v18		
	RMSE	MAE	perf tau	RMSE	MAE	perf tau	RMSE	MAE	perf tau
GNN-DSE	1.104	0.357	0.90	1.253	0.770	0.78	0.955	0.479	0.85
M2	0.991 (-10%)	0.307 (-14%)	0.92	1.330 (+6%)	0.790 (+3%)	0.76	0.796 (-17%)	0.368 (-23%)	0.89
M3	<b>0.975</b> (-12%)	<b>0.257</b> (-28%)	<b>0.93</b>	1.443 (+15%)	0.948 (+23%)	0.70	0.872 (-9%)	0.348 (-27%)	0.89
M4	1.083 (-2%)	0.339 (-5%)	0.91	1.502 (+20%)	0.938 (+22%)	0.73	0.876 (-8%)	0.449 (-6%)	0.86
M5	0.989 (-10%)	0.277 (-23%)	0.92	1.073 (-14%)	0.636 (-17%)	0.81	0.739 (-23%)	<b>0.309</b> (-35%)	0.89
HARP	<b>0.974</b> (-12%)	0.295 (-18%)	<b>0.93</b>	<b>1.015</b> (-19%)	<b>0.601</b> (-22%)	<b>0.82</b>	<b>0.679</b> (-29%)	0.317 (-34%)	<b>0.90</b>

studies. Firstly, we retrained the GNN-DSE model using our HLSyn database as the baseline. Then, we developed M2 by replacing the model architecture of GNN-DSE with our

proposed approach described in Section 8.2.2.1. This involved generating separate vector representations for the program (P) and the transformation (T). Additionally, we constructed M3 by replacing the graph representation with our hierarchical graph. Furthermore, we implemented HARP based on the approach outlined in Section 8.2.2.2. Note that it also exploits the idea of separating vector representations discussed in Section 8.2.2.1. We also examined two variations of this model: M5, where the last GNN layer after the NPT module was excluded, and M4, which additionally applied the pragmas sequentially instead of using the existing parallel and merge structure of the NPT module. For each model, we evaluated its performance under three different scenarios. The first two scenarios involved training the model on datasets v18 and v20, respectively. The third scenario involved utilizing the model pretrained on dataset v18 and fine-tuning it on dataset v20. Our empirical results demonstrated that freezing the parameters of the first GNN layer, which helps reduce the number of parameters requiring updates, resulted in the best performance after fine-tuning.

Table 8.2 summarizes the performance of each model, using three metrics to assess their effectiveness. The first metric uses Root Mean Squared Error (RMSE) for each objective and calculates the total loss by summing the losses of all objectives. The second one utilizes Mean Absolute Error (MAE) instead. For both metrics, we also provide the percentage difference compared to the results obtained from GNN-DSE. Since our primary objective is to conduct DSE for design optimization, the ranking of the `perf` values holds significant importance. Therefore, we employ Kendall’s tau [Ken38], a correlation coefficient that measures the similarity between two variables’ rankings. A value of 1 indicates a perfect positive association. Hence, for RMSE and MAE, lower values indicate better performance, while for tau, higher values indicate superior performance.

The analysis of the results reveals several key observations. Firstly, when we employ separate learning of representations for program P and transformation T (M2), we observe a decrease in both losses and an improvement in the tau ranking of `perf`. However, an exception occurs when the model is trained from scratch on the v20 database. In this case, the increased number of parameters in the new model makes it harder to converge

in a limited training budget (dataset and training time). Nonetheless, when utilizing the pretrained model from the v18 database, the performance is able to catch up and even surpass GNN-DSE. A similar trend is observed when incorporating the hierarchical graph (M3), which further improves the results. Additionally, our findings highlight that the optimal architecture for the NPT involves modeling the pragmas as parallel learnable MLPs, with another MLP responsible for managing their interaction and merging their results. Finally, the most effective model for all scenarios (HARP) utilizes the hierarchy graph and consists of NPT employing the parallel and merge structure, followed by an additional GNN layer to propagate the pragma options throughout the program. It is important to note that this architecture, as depicted in Fig. 8.6, also generates separate embeddings for program P and pseudo nodes B, which contain the pragma (transformation) information here.

Moreover, the results in Table 8.2 align with our expectations, indicating a correlation between the objectives obtained from the two different versions of the HLS tool. Importantly, we observe that the pretrained model from one version can effectively enhance the performance of the other version. This eliminates the need to regenerate the whole training set with each new version of the tool, streamlining the adaptation process. In addition, the results demonstrate that HARP exhibits the best graph representation and model architecture for effectively adapting to task shifts. This validates our hypothesis that by decoupling the learning and representation of the program and its transformations (i.e., pragmas), the model not only acquires a deeper understanding of each component but also enhances its adaptability to new environments.

### 8.3.3 DSE Results

To verify the effectiveness of our model, we use it to identify the Pareto-optimal design points by performing a DSE of the design parameters. We adopt the same exploration technique as GNN-DSE in searching through the solution space. Specifically, we employ a bottom-up approach that utilizes a BFS traversal of the pragmas, starting from the innermost loops. This exploration strategy has shown to be very effective for this problem as it prioritizes the

exploration of fine-grained optimizations over coarse-grained ones. HLS tools can usually perform better for such optimizations, making this approach particularly relevant.

We employ HARP for conducting the DSE with a classification model to assist in pruning invalid design points. Given the ample points available for training the classification model and the relatively simpler task involved, we opt to train a model from scratch for each version using its respective dataset. These models exhibit high accuracy, with rates of 95% for the v18 database and 93% for the v20 database. Given their already high accuracy, we do not employ additional transfer learning methods for them. During the DSE, the classification model first determines the validity of the point, and if deemed valid, the regression model assesses its quality. The DSE seeks to optimize the `perf` value (minimize the latency) while ensuring that resource utilizations remain below 80%. We set a time limit of 1h/kernel on our exploration and can explore approximately 100,000 points during this time. Once the exploration is finished, we synthesize the top 10 points using the HLS tool to get their true labels for comparison.

We also run DSE utilizing the GNN-DSE approach, trained on HLSyn datasets in the same fashion. For the baseline comparison, we employ AutoDSE, which directly runs the HLS tool to evaluate design points. Due to the nature of this approach, AutoDSE requires a more extended runtime. Thus, we set a time limit of 25h/kernel for its DSE. During this period, AutoDSE typically explores an average of 250 valid points. Note that not all of the explored points can finish the synthesis as some of them may be invalid points.

Table 8.3 summarizes the DSE results obtained using versions v18 and v20 of the HLS tool. The DSE is conducted on a total of 35 kernels for SDx 2018.3 (v18) and 26 kernels for Vitis 2020.2 (v20). It is important to note that among the 22 kernels shared between the v18 and v20 databases, the average latency of optimal design in v18 is  $5.54\times$  ( $1.36\times$  on the geometric mean) higher than that in v20, suggesting improvements in the heuristics of the HLS tool over time. Due to space limitations, we only report the arithmetic (avg) and geometric mean (geo mean) of the speedup of the optimal design found by each DSE with respect to the best design discovered by AutoDSE. As the model-based DSEs get to explore

a much larger space, they can find better points compared to a model-free DSE. Notably, for 3mm kernel from PolyBench with a solution space of over 17 trillion points, both HARP and GNN-DSE demonstrate speedups of  $70\times$ . The results reveal that HARP outperforms both AutoDSE and GNN-DSE. Specifically, HARP showcases its competence in adapting to new versions of the HLS tool (v20 kernels), surpassing the performance of GNN-DSE by an average (geometric mean) speedup of  $1.38\times$  ( $1.35\times$ ). It is important to highlight that when transitioning to the v20 dataset, in addition to the task shift, we also have included 4 new kernels, resulting in a domain shift as well. The results validate our hypothesis that the hierarchical graph structure in addition to the decoupling of program and transformation learning contributes to better adaptation capabilities in the face of shifts from the original training data distribution.

Approach	Time Limit	v18 kernels (#:35)		v20 kernels (#:26)	
		avg	geo mean	avg	geo mean
AutoDSE	25h/kernel	$1\times$	$1\times$	$1\times$	$1\times$
GNN-DSE	1h/kernel	$3.51\times$	$0.99\times$	$0.84\times$	$0.78\times$
<b>HARP</b>	1h/kernel	$3.61\times$	$1.23\times$	$1.16\times$	$1.05\times$

Table 8.3: The performance of the best design found by each DSE with respect to the best one found by AutoDSE in 25h.

### 8.3.4 Ablation Study: Transfer Learning with Abundant Data

In Section 8.3.2, we demonstrated that higher prediction accuracy can be achieved through transfer learning when limited data is available. This in turn leads to improved DSE results. To investigate the effectiveness of transfer learning with a large dataset, we developed a larger dataset with Vitis 2021.1 (v21) as detailed in Table 7.1. Similar to previous findings, training a classification model on this dataset yielded 99% accuracy, eliminating the need for further transfer learning. However, for the regression model, we pursued two different approaches. Initially, we fine-tuned the v20 model using the v21 dataset for 400 epochs. Then, considering the large number of data available for this version, we trained another



model from scratch on the dataset for 1500 epochs until it nearly matched the accuracy of the fine-tuned model on the test set.

We use the same experimental setting as in Section 8.3.3. Similar to the improvement we saw in designs’ performances when we transitioned from v18 to v20, we see an average latency reduction of  $2.68\times$  ( $1.58\times$  on the geometric mean) in the optimal design points for the 25 common kernels when we transition from the v20 to the v21 HLS tool. This further shows the improvements in the HLS tools with each new version.

Table 8.4 presents a summary of the arithmetic and geometric mean speedup achieved by each approach compared to the best results obtained by AutoDSE. Notably, model-based approaches demonstrate strong performance when trained from scratch due to their extensive training data. This, coupled with their ability to explore a larger solution space, enables them to discover better design points. However, fine-tuning a pretrained model from a previous version of the HLS tool yielded no advantages for GNN-DSE. It is noteworthy to mention that alongside the task shift, we also encounter a domain shift, with 15 new kernels introduced compared to v20 and 6 new kernels compared to v18 (the pretrained model for v20). In contrast, HARP exhibits more robust results and notable improvements post-transfer learning.

This feature is highly desirable as it allows us to leverage previously gathered data, a process that incurred significant costs. Rather than discarding efforts and expenses associated with collecting datasets from prior HLS tool versions and kernels, we can utilize them to achieve enhanced optimization outcomes. Note that in the current setup, the model trained from scratch nearly underwent  $3.75\times$  more epochs to achieve the same test set accuracy as the fine-tuned model. Nevertheless, the fine-tuned model demonstrates superior generalization across the entire solution space, which includes many unseen points. Notably, this trend is not observed for GNN-DSE, indicating that HARP exhibits better generalization capabilities. This validates our hypothesis that the hierarchical graph structure in addition to the decoupling of program and transformation learning contributes to a more robust representation and better adaptation capabilities in the face of shifts from the original training.

Approach	Time	v21 kernels (#:40)			
	Limit	Trained from scratch		Fine-tuned from v20	
	h/kernel	avg	geo mean	avg	geo mean
AutoDSE	25	1×	1×	1×	1×
GNN-DSE	1	1.17×	1.04×	1.16×	0.98×
<b>HARP</b>	1	1.19×	0.82×	1.46×	1.15×

Table 8.4: The transfer learning results when a larger dataset is present. The performance comparison shows the best design found by each DSE with respect to the best one found by AutoDSE in 25h.

### 8.3.5 Ablation Study: Impact of the Classification Model

In Section 8.2.2, we discussed that numerous pragma combinations result in an invalid design. We defined an invalid design as one that either cannot be synthesized, exceeds a given synthesis time limit, or at least one of the pragmas could not be applied. Table 7.1 shows that in our database, only 30% of the points created a valid design point. Therefore, it is crucial for our optimizer to identify and discard design points that cannot produce a valid microarchitecture. Our classification model is responsible for this task. Table 8.5 compares the speedup performance we achieve compared to AutoDSE when our classification model is absent. Here, we focus solely on the top 10 designs generated by the model. Acknowledging that they may all be invalid designs, we calculate a modified version of the geometric mean (denoted as *geo mean\**), where we add one to all speedup values and subtract one from the resulting geometric mean. The results show that the classification model can help to improve the DSE results since it can prune the invalid points.

### 8.3.6 Ablation Study: Alternative Hierarchy Structure and Larger Model

We tested two additional alternatives to determine if they could improve our results. First, we explored a slightly different hierarchy structure in the graph representation. Instead of connecting the first `pseudo_blocks` of each ‘for’ loop to form the second level of the

Dataset	Without Classification		With Classification	
	avg	geo mean*	avg	geo mean*
v18 #:35	3.14×	0.89×	3.61×	1.37×
v20 #:26	0.93×	0.88×	1.16×	1.05×
v21 #:40	1.34×	0.92×	1.46×	1.09×

Table 8.5: Impact of the classification model. The baseline is AutoDSE after running for 25h.

hierarchy, we introduced new pseudo nodes to represent the ‘for’ loops and connected them based on their hierarchy. Second, we examined the effect of making the model larger by increasing the number of model parameters. We achieved this by increasing the dimension of the hidden layers in the GNN, meaning that we increased the number of features in the graph/node embeddings. Additionally, we added extra layers to the subsequent MLPs to gradually reduce the size of these embeddings for the final prediction. The model was trained from scratch on the v21 dataset, using a similar approach as before.

Figure 8.8 shows the MSE loss of each model on the test set. The results indicate that using a higher dimension and more parameters generally reduces the loss. As the dataset size increases, we could potentially increase these parameters further. While the new hierarchy structure showed some improvement in loss for two cases, this trend was not consistent across all cases. We further ran DSE using models with a dimension of 128, as they performed well with both graph representations. Table 8.6 summarizes the DSE results for these models and compares them to the HARP model from Section 8.3.4. The DSE results indicate that increasing the parameter size improves model performance. However, the new hierarchy structure does not provide consistent benefits. It is important to note that the DSE process evaluates more than just the MSE loss; it also considers prediction accuracy for ranking of points based on their latency.

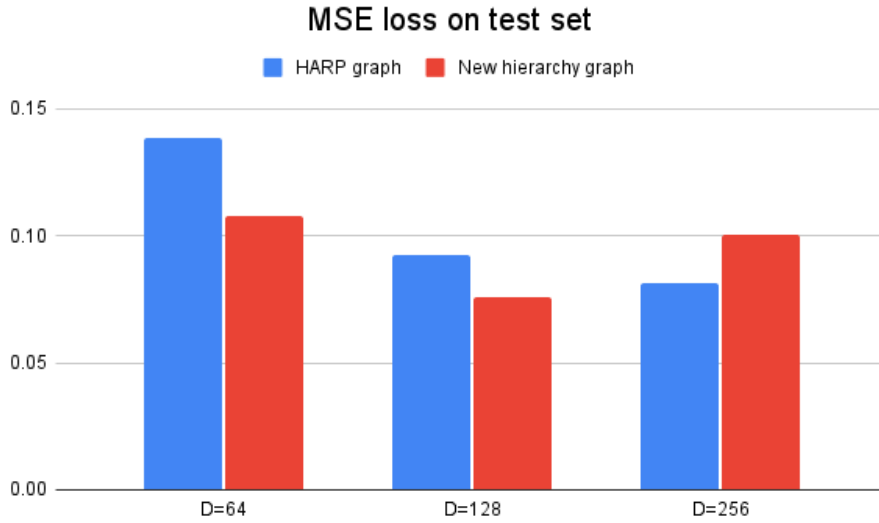


Figure 8.8: Impact of alternative hierarchy structure and larger model on the Mean Squared Error (MSE) on test set. In the new hierarchy graph, we define new pseudo nodes for ‘for’ loops.

Model	v21 kernels (#:40)	
	avg	geo mean
D=64, HARP graph	1.19×	0.82×
D=128, HARP graph	1.22×	0.99×
D=128, New hierarchy graph	1.12×	1.01×

Table 8.6: Impact of alternative hierarchy structure and larger model. In the new hierarchy graph, we define new pseudo nodes for ‘for’ loops. The baseline is AutoDSE after running for 25h.

## 8.4 Conclusion

In this chapter, we discussed three key challenges in developing a GNN-based model for HLS and developed HARP for addressing them. Firstly, we tackle the long-range dependency issue in HLS kernels by proposing a hierarchical graph structure, reducing the average shortest path in our benchmark kernels by 5×. Secondly, recognizing that the final objec-

tives are influenced by two main components, program structure and its transformations in the form of pragmas, we decouple their representation to enhance the model's performance. This improved graph representation and model architecture enable better adaptation to the inevitable domain and task shifts.

## CHAPTER 9

### Concluding Remarks

This dissertation aims to enhance the accessibility of customized computing by creating automated frameworks capable of producing efficient accelerators. Given the importance of machine learning in today’s world and its applications across various industries, we believe it is essential to develop specialized accelerators to process them efficiently. Therefore, in the first part of the dissertation, we concentrate on building highly efficient architectures tailored for handling well-known machine learning tasks. While this method holds great promise for creating accelerators suited to specific purposes, repeating all the necessary steps for each new application requires substantial manual effort and may not be practical. Fortunately, machine learning can help us develop optimizers that target general applications. Consequently, in the second part of the dissertation, we aim to explore how to use machine learning to automate the process of designing microarchitectures, making it applicable to any given application.

In this regard, our initial effort focused on developing FlexCNN in Chapter 3, a flexible and composable architecture designed for processing CNN applications. We demonstrated that using a uniform tiling factor and data layout across all network layers leads to low performance due to varying layer characteristics impacting each layer’s computation-to-communication ratio. To adapt to these variations, FlexCNN employs dynamic tiling and data layout optimization strategies to improve hardware efficiency across different layers. We built a DSE engine guided by analytical latency and resource models to determine optimal hardware configurations for the target network. We were able to develop an analytical model because we worked with a fixed architecture template and only needed to modify a few parameters, such as the number of processing elements and tiling factors. FlexCNN uses

this information to generate the best architecture for the network, fuse different network operations to determine which ones can be executed in one architecture pass, and generate instructions for running the entire network. We also developed a library to integrate FlexCNN with TensorFlow, providing an end-to-end framework. This library reads the network description, generates the optimal architecture, and offloads CNN processing to an FPGA. We identified significant overhead when connecting the FPGA to TensorFlow, so we developed a two-level pipelining system to overlap their execution. We showed that all these optimizations result in  $11.5\times$  speedup for our target network, OpenPose.

Subsequently, we shifted our focus towards analyzing and exploring optimization opportunities involved in the application of GCN to small graphs. We noted that GCNs have different computational complexities and memory access patterns compared to CNNs and traditional graph algorithms, suggesting they could benefit from a specialized accelerator. Therefore, we developed an efficient architecture named StreamGCN in Chapter 4, specifically designed for streaming processing of small graphs. It is especially suitable for facilitating real-time or near real-time search and similarity computations of graphs, with applications across fields such as biology, chemistry, and pharmaceuticals. We identified two types of sparsity in GCN computations. The first one is due to the sparsity of the adjacency matrix, which has been exploited in previous works, while the second one involves the node embedding matrix, which has been overlooked by previous studies. The adjacency matrix is known in advance, while the node embedding is generated dynamically, so we lack information on its sparsity. Consequently, these two require different approaches to exploiting their sparsity. We preprocess the adjacency matrix to prune zero elements and reorder them to eliminate dependencies, allowing us to schedule new operations in each cycle. For the node embeddings, we developed a mechanism that uses a pruner and an arbiter to prune zero elements on-the-fly. As this is a memory-bound application, we further minimized global memory access by reading each element once and scheduling all associated computations before evicting it. We implemented StreamGCN with a deep pipeline that offers varying levels and degrees of parallelization, including node, edge, feature, and batch parallelization,

to effectively manage the workload across different stages. Our optimizations resulted in  $2.3\times$  speedup while requiring  $1.7\times$  less computation resources (DSP).

The second part of this dissertation focuses on democratizing FPGA acceleration for general-purpose programs. We aim to simplify the adoption of customized computing by optimizing an application’s microarchitecture through code augmentation with high-level architectural pragmas. We began this line of work with the development of AutoDSE in Chapter 5. Given the disparities in HLS tool implementations and the unpredictable effects of pragmas on design objectives, we opted to treat the HLS tool as a black box. However, this approach resulted in long evaluation times for each design point. This was particularly problematic as the solution space grows exponentially with the number of pragmas. We found that application-oblivious search heuristics were too slow because they lacked knowledge of the various types of pragmas and their impact on different parts of the code. To address this challenge, we proposed a bottleneck-guided coordinate optimizer and developed AutoDSE as a push-button framework around this concept to systematically and efficiently explore the solution space. To enhance the representation of solutions, we introduced a list comprehension-based design space representation and formulated rules to eliminate meaningless design points and shrink the size of the solution space. Additionally, we implemented a partitioning strategy to mitigate the local optimum problem stemming from the non-monotonic effects of pragmas. We demonstrated that AutoDSE can find design points that closely match manual performance across a wide range of applications. AutoDSE outperformed the previous state-of-the-art works by a factor of  $3.45\times$  to  $17.92\times$ . On manually-optimized vision kernels from the Xilinx Vitis Library [Xilb], AutoDSE achieved slightly better performance while requiring  $26.38\times$  fewer optimization pragmas.

While AutoDSE excels in identifying Pareto-optimal points, its reliance on running the HLS tool for evaluating design candidates makes it a very time-consuming approach. Consequently, it explores only a small fraction of the solution space. To speed up the evaluation process, we leveraged GNNs to build a learning model capable of predicting the design’s objectives within milliseconds. We developed a heterogeneous graph-based program repre-



sensation based on LLVM IR that integrates both program semantics and candidate pragmas, leading to the development of the GNN-DSE framework in Chapter 6. GNN-DSE further employs a carefully-designed GNN-based model for extracting essential information required for estimating our desired targets. The GNN model applies message passing, inspired by the Transformer [Vas+17] architecture, on the attributes we defined for each node and edge to create embeddings that show the role of each node in the graph. It then dynamically selects the neighborhood range for each node and uses an attention mechanism to evaluate the importance of each node in the graph, pooling the node embeddings into a single graph-level embedding, accordingly. We extended AutoDSE to sample a representative database for training a learning-based model. We also described how to use the model directly in an active learning setting to select more informative points for the database. We demonstrated that this approach achieves high accuracy and yields superior DSE results compared to AutoDSE, primarily since it can explore a significantly larger solution space. Additionally, we compiled the HLSyn database in Chapter 7, consisting of 41 distinct kernels and employing three different AMD Xilinx HLS tools to generate labels, resulting in over 81,000 labeled designs, to serve as a large training dataset for training the models.

While GNN-DSE showcased the effectiveness of developing a learning-based optimizer, we identified and addressed three challenges to enhance the robustness of representation learning. This led to the development of HARP in Chapter 8, which stands for Hierarchical Augmentation for Representation with Pragma optimization. First, we found that the graph representation had long-range dependencies, whereas GNNs, with their shallow networks, are more suited for applications requiring a local perspective and often miss capturing a global view. To mitigate this, we introduced a hierarchical graph structure that decreased the average shortest path of HLSyn kernels by  $5\times$ . Specifically, we retrieved missing information about the design’s hierarchy that was lost when converting it to LLVM IR. This additional information helped reduce the shortest path and bring the entire graph within the receptive field of our GNN. Second, we discussed that the final design’s objectives depend on both the program structure and the pragmas applied as program transformations. We showed that

model prediction could be improved by separating the learning of these two components and proposed effective ways to achieve this. Lastly, we discussed two types of shifts that can occur during deployment: domain shift, when encountering a new kernel, and task shift, when facing new labels to predict, such as when changing the HLS tool. We showed that the combination of our hierarchical graph and new model architecture not only improves prediction accuracy but also leads to more robust representations and better adaptability to shifts. These optimizations enable the model to accumulate knowledge from different sources and steadily strengthen its predictive capabilities. This is crucial given the difficulty of collecting large labeled datasets of HLS designs, allowing us to leverage and integrate knowledge from different sources rather than discarding the previous dataset and starting over.

Even though our focus in this dissertation is on FPGAs, our design decisions are not dependent on them. We believe that our approach can be applied to other platforms and HLS tools as well. Moving forward, there are several promising directions for extending the work in this dissertation:

We recognize that machine learning plays a significant role in today’s world. Also, manually analyzing the optimization space of an algorithm and designing a specialized architecture can yield the most efficient accelerator. Though manually designing an accelerator for each application may not be scalable, we believe it is worthwhile to do for widely used machine learning applications. The architecture should be flexible enough and supported by a compiler that can optimize it to match the computation and communication requirements of the target network, similar to our approach in the FlexCNN work.

Although machine learning for EDA holds great potential, applying it to design such large microarchitecture for machine learning applications may not be feasible at the moment. This is primarily due to the intractable solution space that emerges from numerous new optimization opportunities, such as code transformation for data layout optimizations, enabling task-level pipeline parallelism across design stages, etc. To address these challenges automatically, further enhancements to the learning-based framework are necessary, as dis-

cussed later. Nevertheless, a hybrid approach can still be taken by manually breaking down computations into smaller stages and allowing the learning model to optimize them more quickly, freeing designers from this task.

The most critical problem in adopting machine learning for EDA is the lack of open-source datasets and the significant time investment required to generate them using EDA tools, which are typically slow. To advance this field, we believe it is crucial to develop more efficient methods for selecting data points that need labeling, as suggested in [SS17]. In addition, in our HARP work, we demonstrated that the model can benefit from a pretrained model when designed appropriately. Building on this, we suggest leveraging pretrained models through self-supervised learning techniques [Liu+22] or by creating an artificial dataset of HLS kernels using analytical models. Although we discussed that the analytical models may not precisely capture HLS implementations, they can serve as a primitive form of HLS tool, aiding in the understanding and differentiation of various programs and transformations. Subsequently, fine-tuning can help refine predictions and adapt them to the actual HLS tool.

On the representation side, we developed our graph representation based on LLVM IR and aimed to retrieve missing hierarchical information that can enhance the GNN performance. The Multi-Level IR (MLIR) compiler framework [Lat+20] provides a promising alternative for building more informative graphs, as this work also aims to retrieve information that might be lost when using a traditional compiler. Utilizing hierarchical GNNs [Yin+18b] could also prove beneficial, allowing us to divide the graph into sub-graphs, summarize local information, and gradually integrate them to form a global view and facilitate compositional objective prediction. We also see significant potential in combining GNNs with Large Language Models (LLMs) [Zha+23b], as this approach enables us to leverage various design modalities: graph representation and source code. Moreover, state-of-the-art LLMs, trained on vast datasets of diverse code, offer invaluable resources for our task, which is constrained by small datasets.

Given our successful development of a learning model to predict the quality of an HLS design, we believe that applying this data-driven approach to the search process could also

be effective. In our work on AutoDSE, we observed that traditional search heuristics such as simulated annealing and genetic search are not efficient for this problem. By carefully analyzing the space and the problem, we could develop a highly efficient heuristic to explore the search space. Although this analysis was done manually, a data-driven approach in the DSE stage could automate the process and enhance exploration by training a reinforcement learning agent [Wan+20] or employing black-box optimization [KMG22].

# APPENDIX A

## Appendix

### A.1 Optimized HLS Code for CNN

Code A.1 shows the optimized HLS code for the CNN algorithm in Code 5.1 after applying the code transformations and pragmas listed in Table 5.1.

Code A.1: Optimized CNN HLS C Code Snippet

---

```
1 // Skip const variable initialization for brevity
2
3 void CnnKernel(const ap_uint< 128 > * input, float weight,
4               const ap_uint< 512 > * bias, ap_uint< 512 > * output){
5 #pragma HLS INTERFACE m_axi port=input bundle=gmem1 depth=3326977
6 #pragma HLS INTERFACE s_axilite port=input bundle=control
7 // Skip the rest for brevity
8
9 float bias_buf[ParallelOut][ParallelOut];
10 #pragma HLS array_partition variable=bias_buf complete dim=2
11
12 float C[ParallelOut][ImSize][ImSize];
13 #pragma HLS array_partition variable=C cyclic factor=8 dim=3
14 #pragma HLS array_partition variable=C cyclic factor=2 dim=2
15 #pragma HLS array_partition variable=C complete dim=1
16
17 LoadBurst(bias, bias_buf);
18
19 for (int i = 0; i < NumOut / ParallelOut; i++) {
20 float weight_buf[NumOut / ParallelOut][NumIn][kKernel][kKernel];
21 #pragma HLS array_partition variable=weight_buf complete dim=4
22 #pragma HLS array_partition variable=weight_buf complete dim=3
23 #pragma HLS array_partition variable=weight_buf complete dim=1
24
25 float output_buf[NumOut / ParallelOut][OutImSize][OutImSize];
26 #pragma HLS array_partition variable=output_buf cyclic factor=16 dim=3
```

```

27 #pragma HLS array_partition variable=output_buf complete dim=1
28
29     LoadBurst(weight, weight_buf);
30     // Initialization
31     for (int h = 0; h < ImSize; ++h) {
32         for (int w = 0; w < ImSize / 4; ++w) {
33 #pragma HLS dependence variable=C array inter false
34 #pragma HLS pipeline
35         for (int w_sub = 0; w_sub < 4; ++w_sub) {
36 #pragma HLS unroll
37         for (int po = 0; po < ParallelOut; po++) {
38 #pragma HLS unroll
39             C[po][h][w * 4 + w_sub] = 0.f;
40         } } } }
41     // Convolution
42     for (int j = 0; j < NumIn; ++j) {
43         float input_buf[InImSize][InImSize];
44 #pragma HLS array_partition variable=input_buf cyclic factor=8 dim=2
45 #pragma HLS array_partition variable=input_buf cyclic factor=5 dim=1
46         LoadBurst(input, input_buf);
47         for (int h = 0; h < ImSize; ++h) {
48             for (int w = 0; w < ImSize / 4; ++w) {
49 #pragma HLS dependence variable=C array inter false
50 #pragma HLS pipeline
51             for (int w_sub = 0; w_sub < 4; ++w_sub) {
52 #pragma HLS unroll
53             for (int po = 0; po < ParallelOut; po++) {
54 #pragma HLS unroll
55                 float tmp = 0.f;
56                 for (int p = 0; p < kKernel; ++p) {
57 #pragma HLS unroll
58                     for (int q = 0; q < kKernel; ++q) {
59 #pragma HLS unroll
60                         tmp += ...;
61                     } }
62                 C[po][h][w * 4 + w_sub] += tmp;
63             } } } } }
64     // ReLU + Max pooling
65     for (int h = 0; h < OutImSize; ++h) {
66         for (int w = 0; w < OutImSize; ++w) {
67 #pragma HLS dependence variable=output_buf array inter false
68 #pragma HLS pipeline
69         for (int po = 0; po < ParallelOut; po++) {

```

```

70 #pragma HLS unroll
71     output_buf(h, w, po) = ...
72 } } }
73 StoreBurst(output, output_buf);
74 } }

```

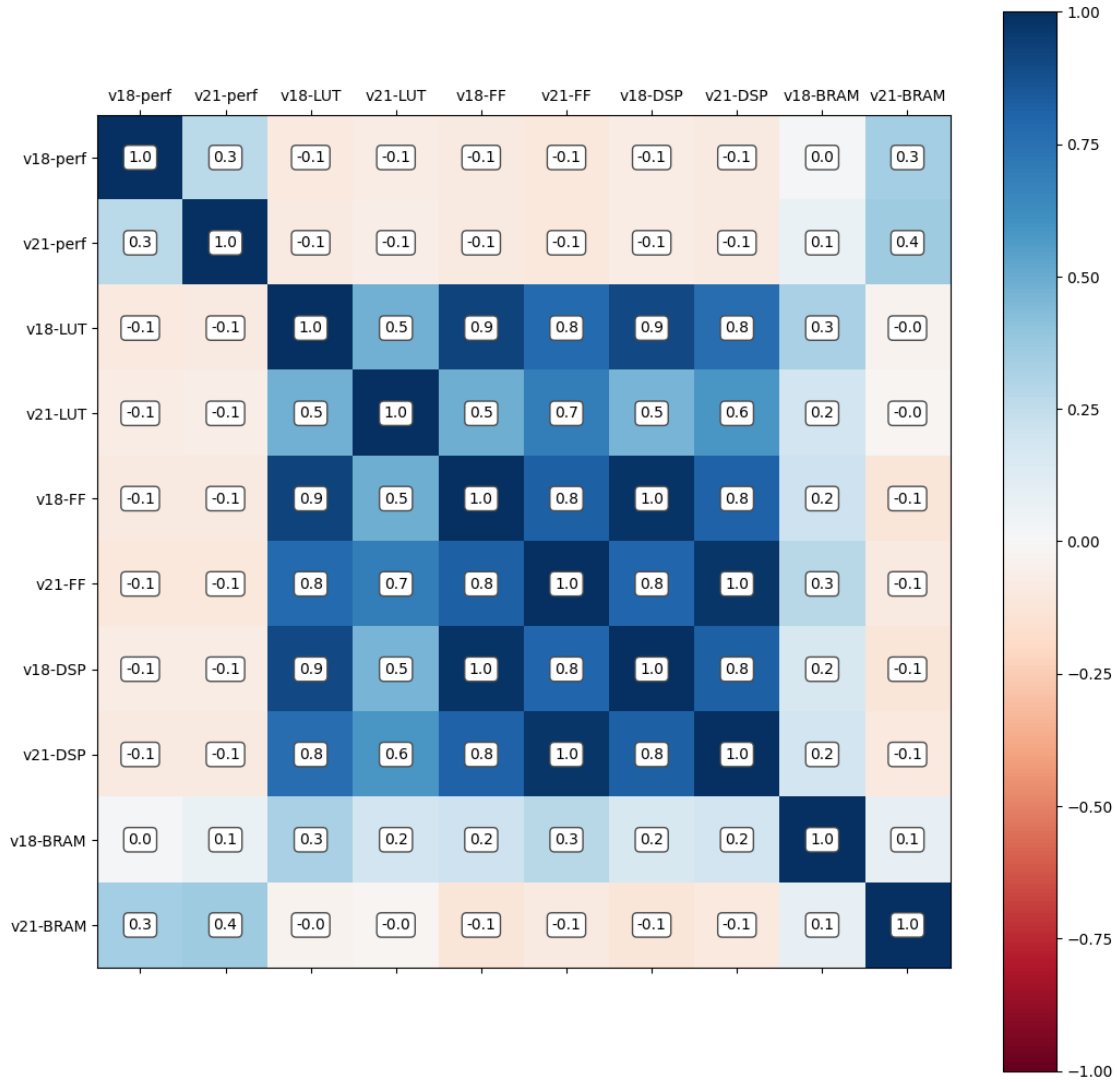


Figure A.1: Correlation matrix of the design objectives resulted from AMD Xilinx SDx 2018.3 (v18) and Vitis 2021.1 (v21).

## A.2 Correlation Matrices of HLSyn Benchmark

We depicted the correlation matrix between versions v10 and v20 in Section 8.2.3. Fig. A.1 shows the correlation matrix between versions v18 and v21 of our HLSyn benchmark, while Fig. A.2 illustrates the correlation matrix between versions v20 and v21. Across successive versions, we observe a high correlation among the same objectives, suggesting that fine-tuning could be highly effective, as discussed in Section 8.2.3.

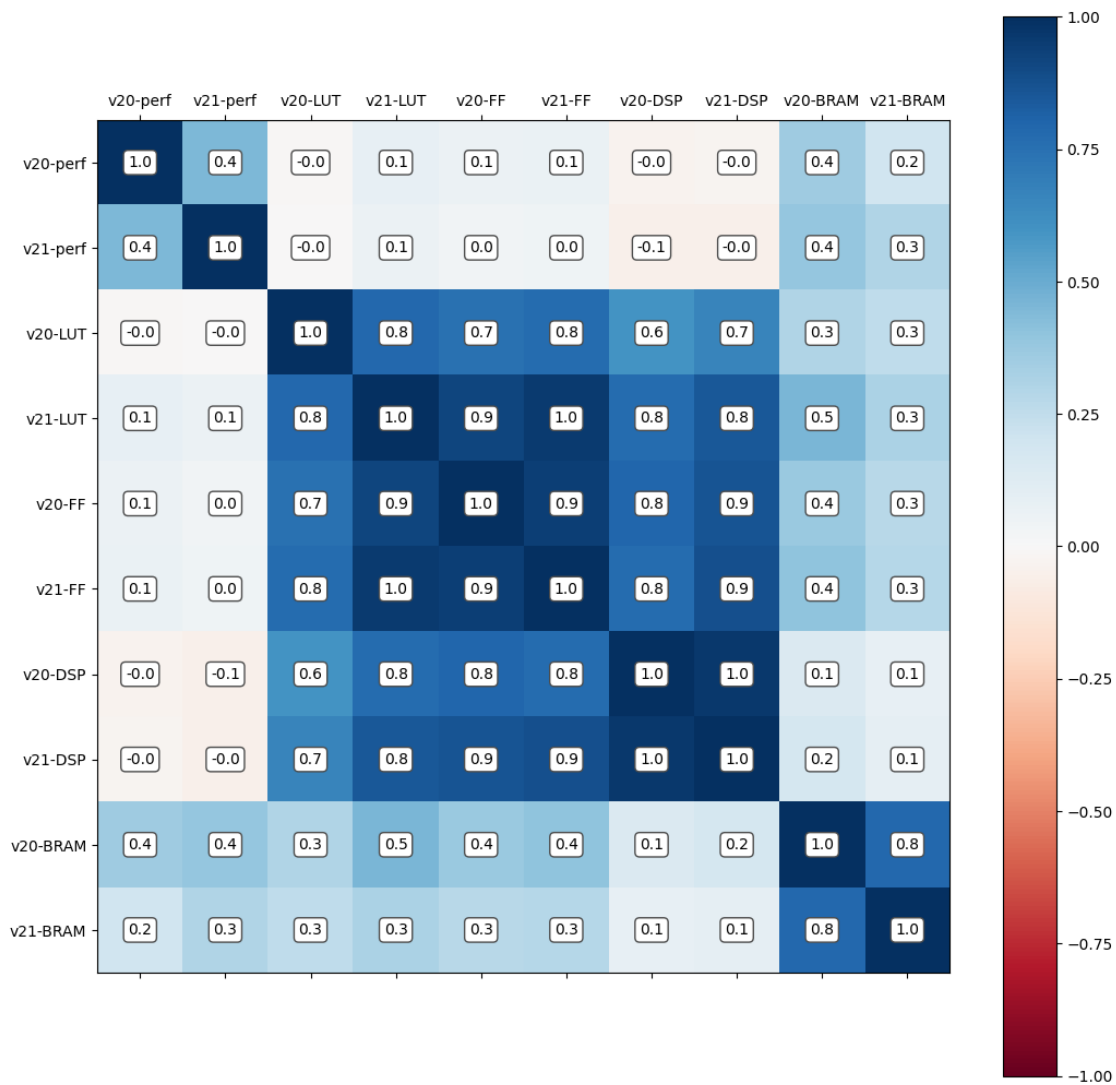


Figure A.2: Correlation matrix of the design objectives resulted from AMD Xilinx Vitis 2020.2 (v20) and Vitis 2021.1 (v21).



## BIBLIOGRAPHY

- [Aba+16] Martín Abadi et al. “Tensorflow: A system for large-scale machine learning”. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 2016, pp. 265–283.
- [Aki+18] Jinguji Akira et al. “An FPGA Realization of OpenPose Based on a Sparse Weight Convolutional Neural Network”. In: *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE. 2018, pp. 310–313.
- [Ala+20] Mohamed Baker Alawieh et al. “High-definition routing congestion prediction for large-scale FPGAs”. In: *2020 25th Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE. 2020, pp. 26–31.
- [Als+16] Johnathan Alsop et al. “Lazy release consistency for GPUs”. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2016, pp. 1–14.
- [Ama24] Amazon. *Amazon EC2 F1 Instance*. 2024. URL: <https://aws.amazon.com/ec2/instance-types/f1/> (visited on 04/30/2024).
- [AMD] AMD/Xilinx SDAccel - Vivado HLS. <https://docs.xilinx.com/v/u/2018.3-English/ug902-vivado-high-level-synthesis>.
- [AMD22] AMD. *AMD Completes Acquisition of Xilinx*. 2022. URL: <https://www.amd.com/en/press-releases/2022-02-14-amd-completes-acquisition-xilinx> (visited on 04/30/2024).
- [AMD24a] AMD. “Vitis Unified Software Platform”. In: (2024). URL: <https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html>.
- [AMD24b] AMD. *Vivado HLS*. 2024. URL: [www.xilinx.com/products/design-tools/vivado](http://www.xilinx.com/products/design-tools/vivado).

- [And+17] Joao Andrade et al. “Design space exploration of LDPC decoders using high-level synthesis”. In: *IEEE Access* 5 (2017), pp. 14600–14615.
- [Ans+14] Jason Ansel et al. “Opentuner: An extensible framework for program autotuning”. In: *Proceedings of the 23rd international conference on Parallel architectures and compilation*. 2014, pp. 303–316.
- [Bai+19] Yunsheng Bai et al. “Simgnn: A neural network approach to fast graph similarity computation”. In: *Proceedings of the twelfth ACM international conference on web search and data mining*. 2019, pp. 384–392.
- [Bai+20] Yunsheng Bai et al. “Learning-based efficient graph similarity computation via multi-scale convolutional set matching”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 34. 04. 2020, pp. 3219–3226.
- [Bai+22] Yunsheng Bai et al. “Improving GNN-based accelerator design automation with meta learning”. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 2022, pp. 1347–1350.
- [Bas+19] Abanti Basak et al. “Analysis and optimization of the memory hierarchy for graph processing workloads”. In: *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2019, pp. 373–386.
- [Bas+23] Suhail Basalama et al. “FlexCNN: An End-to-End Framework for Composing CNN Accelerators on FPGA”. In: *ACM Transactions on Reconfigurable Technology and Systems* 16.2 (2023), pp. 1–32.
- [Ben+10] Mohamed-Walid Benabderrahmane et al. “The polyhedral model is more widely applicable than you think”. In: *International Conference on Compiler Construction*. Springer. 2010, pp. 283–303.
- [Bol+08] Evan E Bolton et al. “PubChem: integrated platform of small molecules and biological activities”. In: *Annual reports in computational chemistry*. Elsevier, 2008.

- [Bra00] Gary Bradski. “The opencv library”. In: *Dr Dobb’s J. Software Tools* 25 (2000), pp. 120–125.
- [BS98] Horst Bunke and Kim Shearer. “A graph distance metric based on the maximal common subgraph”. In: *Pattern Recognition Letters* (1998).
- [Bus20] BusinessWire. *Xilinx Acquires Assets of Falcon Computing Solutions to Advance Software Programmability and Expand Developer Community*. 2020. URL: <https://www.businesswire.com/news/home/20201201005316/en/Xilinx-Acquires-Assets-of-Falcon-Computing-Solutions-to-Advance-Software-Programmability-and-Expand-Developer-Community> (visited on 04/30/2024).
- [BZH18] Lin Bai, Yiming Zhao, and Xinming Huang. “A CNN accelerator on FPGA using depthwise separable convolution”. In: *IEEE Transactions on Circuits and Systems II: Express Briefs* 65.10 (2018), pp. 1415–1419.
- [Cad24] Cadence. *Cadence Stratus High-Level Synthesis*. 2024. URL: [https://www.cadence.com/en\\_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html](https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/synthesis/stratus-high-level-synthesis.html).
- [Cao+17] Zhe Cao et al. “Realtime multi-person 2d pose estimation using part affinity fields”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2017, pp. 7291–7299.
- [CC18] Young-kyu Choi and Jason Cong. “HLS-based optimization and design space exploration for applications with variable loop bounds”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.
- [CES16] Yu-Hsin Chen, Joel Emer, and Vivienne Sze. “Eyeriss: A spatial architecture for energy-efficient dataflow for convolutional neural networks”. In: *ACM SIGARCH Computer Architecture News* (2016).

- [Che+09] Shuai Che et al. “Rodinia: A benchmark suite for heterogeneous computing”. In: *2009 IEEE international symposium on workload characterization (IISWC)*. Ieee. 2009, pp. 44–54.
- [Che+14] Sharan Chetlur et al. “cuDNN: Efficient primitives for deep learning”. In: *arXiv preprint arXiv:1410.0759* (2014).
- [Che+16] Yu-Ting Chen et al. “When Spark meets FPGAs: A case study for next-generation DNA sequencing acceleration”. In: *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2016.
- [Che+18] Tianqi Chen 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)*. 2018, pp. 578–594.
- [Che+19] Guangyong Chen et al. “Alchemy: A quantum chemistry dataset for benchmarking ai models”. In: *arXiv preprint arXiv:1906.09427* (2019).
- [Chi+18] Yuze Chi et al. “SODA: stencil with optimized dataflow architecture”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 2018, pp. 1–8.
- [Chi+22] Yuze Chi et al. “Democratizing Domain-Specific Computing”. In: *Communications of the ACM* 66.1 (2022), pp. 74–85.
- [Con+11] Jason Cong et al. “High-level synthesis for FPGAs: From prototyping to deployment”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 30.4 (2011), pp. 473–491.
- [Con+16a] Jason Cong et al. “Software infrastructure for enabling FPGA-based accelerations in data centers”. In: *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*. 2016, pp. 154–155.
- [Con+16b] Jason Cong et al. “Source-to-source optimization for HLS”. In: *FPGAs for Software Programmers* (2016), pp. 137–163.

- [Con+18a] Jason Cong et al. “Automated accelerator generation and optimization with composable, parallel and pipeline architecture”. In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6.
- [Con+18b] Jason Cong et al. “Best-effort FPGA programming: A few steps can go a long way”. In: *arXiv preprint arXiv:1807.01340* (2018).
- [CUH15] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. “Fast and accurate deep network learning by exponential linear units (elus)”. In: *arXiv preprint arXiv:1511.07289* (2015).
- [Cum+21] Chris Cummins et al. “ProGraML: A Graph-based Program Representation for Data Flow Analysis and Compiler Optimizations”. In: *Proceedings of the 38th International Conference on Machine Learning*. Ed. by Marina Meila and Tong Zhang. Vol. 139. Proceedings of Machine Learning Research. PMLR, July 2021, pp. 2244–2253. URL: <https://proceedings.mlr.press/v139/cummins21a.html>.
- [CW18] Jason Cong and Jie Wang. “PolySA: Polyhedral-based systolic array auto-compilation”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.
- [CWY18] Jason Cong, Peng Wei, and Cody Hao Yu. “From JVM to FPGA: Bridging abstraction hierarchy via optimized deep pipelining”. In: *10th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud)*. 2018.
- [Dad+19] Vidushi Dadu et al. “Towards general purpose acceleration by exploiting common data-dependence forms”. In: *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*. 2019, pp. 924–939.
- [Dai+17] Guohao Dai et al. “ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture”. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2017, pp. 217–226.

- [Den+74] Robert H Dennard et al. “Design of ion-implanted MOSFET’s with very small physical dimensions”. In: *IEEE Journal of solid-state circuits* 9.5 (1974), pp. 256–268.
- [Dep24a] Statista Research Department. *Information technology (IT) spending on data center systems worldwide from 2012 to 2024*. 2024. URL: <https://www.statista.com/statistics/314596/total-data-center-systems-worldwide-spending-forecast/> (visited on 01/05/2024).
- [Dep24b] Statista Research Department. *Volume of data/information created, captured, copied, and consumed worldwide from 2010 to 2025*. 2024. URL: <https://www.statista.com/statistics/871513/worldwide-data-created/> (visited on 01/05/2024).
- [DM98] Leonardo Dagum and Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *IEEE computational science and engineering* 5.1 (1998), pp. 46–55.
- [Dua+18] Javier Duarte et al. “Fast inference of deep neural networks in FPGAs for particle physics”. In: *Journal of Instrumentation* 13.07 (2018), P07027.
- [Duv+15] David K Duvenaud et al. “Convolutional networks on graphs for learning molecular fingerprints”. In: *Advances in neural information processing systems* 28 (2015).
- [DV05] Philippe Dosch and Ernest Valveny. “Report on the second symbol recognition contest”. In: *International Workshop on Graphics Recognition*. Springer. 2005.
- [Fal] Falcon Computing Solutions, Inc. <http://www.falcon-computing.com>.
- [Fan+20] Weili Fang et al. “Computer vision for behaviour-based safety in construction: A review and future directions”. In: *Advanced Engineering Informatics* 43 (2020), p. 100980.

- [FAP18a] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. “Cluster-based heuristic for high level synthesis design space exploration”. In: *IEEE Transactions on Emerging Topics in Computing* 9.1 (2018), pp. 35–43.
- [FAP18b] Lorenzo Ferretti, Giovanni Ansaloni, and Laura Pozzi. “Lattice-traversing design space exploration for high level synthesis”. In: *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE. 2018, pp. 210–217.
- [Fen+23] Siyuan Feng et al. “Tensorir: An abstraction for automatic tensorized program optimization”. In: *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 2023, pp. 804–817.
- [Fey24] Matthias Fey. “PyTorch Scatter”. In: (2024). URL: [https://github.com/rustyls/pytorch\\_scatter](https://github.com/rustyls/pytorch_scatter).
- [Fia+10] Álvaro Fialho et al. “Analyzing bandit-based adaptive operator selection mechanisms”. In: *Annals of Mathematics and Artificial Intelligence* 60.1 (2010), pp. 25–64.
- [Fin+20] Johannes de Fine Licht et al. “Transformations of high-level synthesis codes for high-performance computing”. In: *IEEE Transactions on Parallel and Distributed Systems* 32.5 (2020), pp. 1014–1029.
- [FL19] Matthias Fey and Jan Eric Lenssen. “Fast graph representation learning with PyTorch Geometric”. In: *arXiv preprint arXiv:1903.02428* (2019).
- [Gen+20] Tong Geng et al. “AWB-GCN: A graph convolutional network accelerator with runtime workload rebalancing”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 922–936.
- [Gu+18] Jiuxiang Gu et al. “Recent advances in convolutional neural networks”. In: *Pattern Recognition* 77 (2018), pp. 354–377.

- [Gua+17] Yijin Guan et al. “FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates”. In: *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2017, pp. 152–159.
- [Guo+22] Zizheng Guo et al. “A timing engine inspired graph neural network model for pre-routing slack prediction”. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 2022, pp. 1207–1212.
- [Ham+16] Tae Jun Ham et al. “Graphicionado: A high-performance and energy-efficient accelerator for graph analytics”. In: *2016 49th annual IEEE/ACM international symposium on microarchitecture (MICRO)*. IEEE. 2016, pp. 1–13.
- [Han+16] Song Han et al. “EIE: Efficient inference engine on compressed deep neural network”. In: *ACM SIGARCH Computer Architecture News* (2016).
- [He+16] Kaiming He et al. “Deep residual learning for image recognition”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.
- [Hon+18] Changwan Hong et al. “Gpu code optimization using abstract kernel emulation and sensitivity analysis”. In: *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2018, pp. 736–751.
- [How+17] Andrew G Howard et al. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).
- [HS06] Geoffrey E Hinton and Ruslan R Salakhutdinov. “Reducing the dimensionality of data with neural networks”. In: *science* 313.5786 (2006), pp. 504–507.
- [Hua+21] Guyue Huang et al. “Machine learning for electronic design automation: A survey”. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 26.5 (2021), pp. 1–46.



- [Ign+18] Andrey Ignatov et al. “AI benchmark: Running deep neural networks on android smartphones”. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018.
- [Int24a] Intel. *Intel High-Level Synthesis Compiler*. 2024. URL: <https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html>.
- [Int24b] Intel. “Intel MKL”. In: (2024). URL: <https://software.intel.com/en-us/mkl>.
- [Int24c] Intel. *Intel VTune Profiler*. 2024. URL: <https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html>.
- [Ish+21] Sho Ishida et al. “Graph Neural Networks with Multiple Feature Extraction Paths for Chemical Property Estimation”. In: *Molecules* (2021).
- [JL21] Weiwei Jiang and Jiayun Luo. “Graph neural network for traffic forecasting: A survey”. In: *arXiv preprint arXiv:2101.11174* (2021).
- [Kan92] Viggo Kann. “On the approximability of the maximum common subgraph problem”. In: *Annual Symposium on Theoretical Aspects of Computer Science*. Springer. 1992.
- [KB14] Diederik P Kingma and Jimmy Ba. “Adam: A method for stochastic optimization”. In: *arXiv preprint arXiv:1412.6980* (2014).
- [KC20] Jihye Kwon and Luca P Carloni. “Transfer Learning for Design-Space Exploration with High-Level Synthesis”. In: *2020 ACM/IEEE 2nd Workshop on Machine Learning for CAD (MLCAD)*. IEEE. 2020, pp. 163–168.
- [Ken38] Maurice G Kendall. “A new measure of rank correlation”. In: *Biometrika* 30.1/2 (1938), pp. 81–93.
- [Kha+20] Brucek Khailany et al. “Accelerating Chip Design With Machine Learning”. In: *IEEE Micro* 40.6 (2020), pp. 23–32.

- [Kim18] Ildoo Kim. *tf-pose-estimation*. <https://github.com/ildoonet/tf-pose-estimation>. 2018.
- [Kir+19] Robert Kirby et al. “CongestionNet: Routing congestion prediction using deep graph neural networks”. In: *2019 IFIP/IEEE 27th International Conference on Very Large Scale Integration (VLSI-SoC)*. IEEE. 2019, pp. 217–222.
- [KMG22] Siddarth Krishnamoorthy, Satvik Mehul Mashkaria, and Aditya Grover. “Generative pretraining for black-box optimization”. In: *arXiv preprint arXiv:2206.10786* (2022).
- [KMZ19] HT 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*. 2019, pp. 821–834.
- [Koe+16] David Koeplinger et al. “Automatic generation of efficient accelerators for reconfigurable hardware”. In: *ACM SIGARCH Computer Architecture News* 44.3 (2016), pp. 115–127.
- [Kou+22] Mingyang Kou et al. “GEML: GNN-based efficient mapping method for large loop applications on CGRA”. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 2022, pp. 337–342.
- [KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. “Imagenet classification with deep convolutional neural networks”. In: *Advances in neural information processing systems* 25 (2012).
- [KW16] Thomas N Kipf and Max Welling. “Semi-supervised classification with graph convolutional networks”. In: *arXiv preprint arXiv:1609.02907* (2016).
- [LA04] Chris Lattner and Vikram Adve. “LLVM: A compilation framework for life-long program analysis & transformation”. In: *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE. 2004, pp. 75–86.

- [Lab] Lawrence Livermore National Laboratory. *Rose Compiler Infrastructure*. URL: <http://rosecompiler.org/>.
- [Lai+19] Yi-Hsiang Lai et al. “HeteroCL: A multi-paradigm programming infrastructure for software-defined reconfigurable computing”. In: *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2019, pp. 242–251.
- [Lat+20] Chris Lattner et al. “MLIR: A compiler infrastructure for the end of Moore’s law”. In: *arXiv preprint arXiv:2002.11054* (2020).
- [Lau+20] Jason Lau et al. “HeteroRefactor: Refactoring for heterogeneous computing with FPGA”. In: *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 2020, pp. 493–505.
- [LC13] Hung-Yi Liu and Luca P Carloni. “On learning-based methods for design-space exploration with high-level synthesis”. In: *Proceedings of the 50th annual design automation conference*. 2013, pp. 1–7.
- [LHW18] Qimai Li, Zhichao Han, and Xiao-Ming Wu. “Deeper insights into graph convolutional networks for semi-supervised learning”. In: *Proceedings of the AAAI conference on artificial intelligence*. Vol. 32. 1. 2018.
- [Li+15] Yujia Li et al. “Gated graph sequence neural networks”. In: *arXiv preprint arXiv:1511.05493* (2015).
- [Li+16] Huimin Li et al. “A high performance FPGA-based accelerator for large-scale convolutional neural networks”. In: *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*. IEEE. 2016, pp. 1–9.
- [Li+17] Zhaoshi Li et al. “Aggressive pipelining of irregular applications on reconfigurable hardware”. In: *Proceedings of the 44th Annual International Symposium on Computer Architecture*. 2017, pp. 575–586.

- [Li+19] Yujia Li et al. “Graph matching networks for learning the similarity of graph structured objects”. In: *International conference on machine learning*. PMLR. 2019, pp. 3835–3845.
- [Li+22] Yujia Li et al. “Competition-level code generation with alphacode”. In: *Science* 378.6624 (2022), pp. 1092–1097.
- [Lia+20] Shengwen Liang et al. “Engn: A high-throughput and energy-efficient accelerator for large graph neural networks”. In: *IEEE Transactions on Computers* 70.9 (2020), pp. 1511–1525.
- [Liu+22] Yixin Liu et al. “Graph self-supervised learning: A survey”. In: *IEEE transactions on knowledge and data engineering* 35.6 (2022), pp. 5879–5900.
- [LLS19] Shuangnan Liu, Francis CM Lau, and Benjamin Carrion Schafer. “Accelerating FPGA prototyping through predictive model-based HLS design space exploration”. In: *Proceedings of the 56th Annual Design Automation Conference 2019*. 2019, pp. 1–6.
- [LPL20] Yi-Chen Lu, Sai Pentapati, and Sung Kyu Lim. “VLSI placement optimization using graph neural networks”. In: *Proceedings of the 34th Advances in Neural Information Processing Systems (NeurIPS) Workshop on ML for Systems, Virtual*. 2020, pp. 6–12.
- [Ma+20a] Hehuan Ma et al. “Multi-view graph neural networks for molecular property prediction”. In: *arXiv preprint arXiv:2005.13607* (2020).
- [Ma+20b] Yuzhe Ma et al. “Understanding graphs in EDA: From shallow to deep learning”. In: *Proceedings of the 2020 International Symposium on Physical Design*. 2020, pp. 119–126.
- [Mat+17] De G Matthews et al. “GPflow: A Gaussian process library using TensorFlow”. In: *The Journal of Machine Learning Research* 18.1 (2017), pp. 1299–1304.
- [MH08] Laurens van der Maaten and Geoffrey Hinton. “Visualizing data using t-SNE”. In: *Journal of machine learning research* 9.Nov (2008), pp. 2579–2605.

- [Mir+21] Azalia Mirhoseini et al. “A graph placement methodology for fast chip design”. In: *Nature* 594.7862 (2021), pp. 207–212.
- [MS14] Anushree Mahapatra and Benjamin Carrion Schafer. “Machine-learning based simulated annealer method for high level synthesis design space exploration”. In: *Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn)*. IEEE. 2014, pp. 1–6.
- [Muk+18] Anurag Mukkara et al. “Exploiting locality in graph analytics through hardware-accelerated traversal scheduling”. In: *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2018, pp. 1–14.
- [NCI04] NCI/NIH. “AIDS Antiviral Screen Data.” In: (2004). URL: <https://wiki.nci.nih.gov/display/NCIDTPdata/AIDS+Antiviral+Screen+Data>.
- [NEC24] NEC. *NEC CyberWorkBench*. 2024. URL: <https://www.nec.com/en/global/prod/cwb/index.html>.
- [Net+19] Walter Lau Neto et al. “LSOracle: A logic synthesis framework driven by artificial intelligence”. In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2019, pp. 1–6.
- [NH10] Vinod Nair and Geoffrey E Hinton. “Rectified linear units improve restricted boltzmann machines”. In: *Proceedings of the 27th international conference on machine learning (ICML-10)*. 2010, pp. 807–814.
- [Nig+20] Rachit Nigam et al. “Predictable accelerator design with time-sensitive affine types”. In: *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*. 2020, pp. 393–407.
- [NSW18] Daniel H Noronha, Bahar Salehpour, and Steven JE Wilton. “LeFlow: Enabling flexible FPGA high-level synthesis of TensorFlow deep neural networks”. In: *FSP Workshop 2018; Fifth International Workshop on FPGAs for Software Programmers*. VDE. 2018, pp. 1–8.

- [NVI24] NVIDIA. “NVIDIA cuBLAS”. In: (2024). URL: <https://developer.nvidia.com/cublas>.
- [Ope23] OpenAI. *GPT-4 Technical Report*. 2023. arXiv: [2303.08774](https://arxiv.org/abs/2303.08774) [cs.CL].
- [Ope24] OpenAI. *generatubg: Introducing ChatGPT* <https://openai.com/blog/chatgpt>. 2024.
- [Par+04] David Parello et al. “Towards a systematic, pragmatic and architecture-aware program optimization process for complex processors”. In: *SC’04: Proceedings of the 2004 ACM/IEEE conference on Supercomputing*. IEEE, 2004, pp. 15–15.
- [Par+17] Angshuman Parashar et al. “Scnn: An accelerator for compressed-sparse convolutional neural networks”. In: *ACM SIGARCH Computer Architecture News* (2017).
- [Pas+19] Adam Paszke et al. “PyTorch: An Imperative Style, High-Performance Deep Learning Library”. In: *Advances in Neural Information Processing Systems 32*. Ed. by H. Wallach et al. Curran Associates, Inc., 2019, pp. 8024–8035.
- [PMC19] Dimitri Palaz, Mathew Magimai-Doss, and Ronan Collobert. “End-to-end acoustic modeling using convolutional neural networks for HMM-based automatic speech recognition”. In: *Speech Communication* 108 (2019), pp. 15–32.
- [Pra+16] Raghu Prabhakar et al. “Generating configurable hardware from parallel patterns”. In: *Acm Sigplan Notices* 51.4 (2016), pp. 651–665.
- [Put+14] Andrew Putnam et al. “A reconfigurable fabric for accelerating large-scale data-center services”. In: *ACM SIGARCH Computer Architecture News* 42.3 (2014), pp. 13–24.
- [QBS20] Zongyue Qin, Yunsheng Bai, and Yizhou Sun. “GHashing: Semantic graph hashing for approximate similarity search in graph databases”. In: *Proceedings of the 26th ACM SIGKDD international conference on knowledge discovery & data mining*. 2020, pp. 2062–2072.

- [RB08] Kaspar Riesen and Horst Bunke. “IAM graph database repository for graph based pattern recognition and machine learning”. In: *Structural, Syntactic, and Statistical Pattern Recognition: Joint IAPR International Workshop, SSPR & SPR 2008, Orlando, USA, December 4-6, 2008. Proceedings*. Springer. 2008, pp. 287–297.
- [Rea+14] Brandon Reagen et al. “Machsuite: Benchmarks for accelerator design and customized architectures”. In: *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE. 2014, pp. 110–119.
- [Reg+19] Enrico Reggiani et al. “Pareto optimal design space exploration for accelerated CNN on FPGA”. In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. IEEE. 2019, pp. 107–114.
- [Ren+22] Haoxing Ren et al. “Why are graph neural networks effective for eda problems?”. In: *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design*. 2022, pp. 1–8.
- [Reu22] Reuters. *AMD closes record chip industry deal with estimated 50 billion purchase of Xilinx*. 2022. URL: <https://www.reuters.com/technology/amd-closes-biggest-chip-acquisition-with-498-bln-purchase-xilinx-2022-02-14/> (visited on 04/30/2024).
- [Roz18] Benedek Rozemberczki. “SimGNN.” In: (2018). URL: <https://github.com/benedekrozemberczki/SimGNN>.
- [San+18] Mark Sandler et al. “MobileNetv2: Inverted residuals and linear bottlenecks”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 4510–4520.
- [Sch17] Benjamin Carrion Schafer. “Parallel high-level synthesis design space exploration for behavioral ips of exact latencies”. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 22.4 (2017), pp. 1–20.

- [SF83] Alberto Sanfeliu and King-Sun Fu. “A distance measure between attributed relational graphs for pattern recognition”. In: *IEEE Transactions on Systems, Man, and Cybernetics* (1983).
- [SFM17a] Yongming Shen, Michael Ferdman, and Peter Milder. “Escher: A CNN accelerator with flexible buffering to minimize off-chip transfer”. In: *2017 IEEE 25Th annual international symposium on field-programmable custom computing machines (FCCM)*. IEEE. 2017, pp. 93–100.
- [SFM17b] 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)*. IEEE. 2017, pp. 535–547.
- [Shi+20] Yunsheng Shi et al. “Masked label prediction: Unified message passing model for semi-supervised classification”. In: *arXiv preprint arXiv:2009.03509* (2020).
- [SI15] Teague Sterling and John J Irwin. “ZINC 15–ligand discovery for everyone”. In: *Journal of chemical information and modeling* (2015).
- [Sie24] Siemens. *Siemens Catapult High-Level Synthesis*. 2024. URL: <https://eda.sw.siemens.com/en-US/ic/ic-design/high-level-synthesis-and-verification-platform/>.
- [SM14] Laurent Sifre and Stéphane Mallat. “Rigid-motion scattering for image classification”. In: *Ph. D. dissertation* (2014).
- [Sma24a] SmartNIC. *Cisco Nexus SmartNIC*. 2024. URL: <https://www.cisco.com/c/en/us/products/interfaces-modules/nexus-smartnic/index.html> (visited on 04/29/2024).
- [Sma24b] SmartSSD. *Samsung Computational Storage Drive*. 2024. URL: <https://www.xilinx.com/applications/data-center/computational-storage/smartssd.html> (visited on 04/29/2024).



- [Sno+15] Jasper Snoek et al. “Scalable bayesian optimization using deep neural networks”. In: *International conference on machine learning*. PMLR. 2015, pp. 2171–2180.
- [Sri+20a] Nitish Srivastava et al. “Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product”. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE. 2020, pp. 766–780.
- [Sri+20b] Nitish Srivastava et al. “Tensaurus: A versatile accelerator for mixed sparse-dense tensor computations”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 689–702.
- [SS17] Ozan Sener and Silvio Savarese. “Active learning for convolutional neural networks: A core-set approach”. In: *arXiv preprint arXiv:1708.00489* (2017).
- [Sud+16] Naveen Suda et al. “Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks”. In: *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2016, pp. 16–25.
- [Sun+21] Qi Sun et al. “Correlated Multi-objective Multi-fidelity Optimization for HLS Directives Design”. In: *IEEE/ACM Proceedings Design, Automation and Test in Europe (DATE)*. 2021, pp. 01–05.
- [SW12a] B Carrion Schafer and Kazutoshi Wakabayashi. “Machine learning predictive modelling high-level synthesis design space exploration”. In: *IET computers & digital techniques*. Vol. 6. 3. 2012, pp. 153–159.
- [SW12b] Benjamin Carrion Schafer and Kazutoshi Wakabayashi. “Divide and conquer high-level synthesis design space exploration”. In: *ACM Transactions on Design Automation of Electronic Systems (TODAES)* 17.3 (2012), pp. 1–19.
- [SW19] Benjamin Carrion Schafer and Zi Wang. “High-Level Synthesis Design Space Exploration: Past, Present, and Future”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39.10 (2019), pp. 2628–2639.

- [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*. 2020, pp. 133–139.
- [SZ14] Karen Simonyan and Andrew Zisserman. “Very deep convolutional networks for large-scale image recognition”. In: *arXiv preprint arXiv:1409.1556* (2014).
- [TLH19] Qiaoyu Tan, Ninghao Liu, and Xia Hu. “Deep representation learning for social network analysis”. In: *Frontiers in big Data* 2 (2019), p. 2.
- [Ust+20] Ecenur Ustun et al. “Accurate operation delay prediction for FPGA HLS using graph neural networks”. In: *Proceedings of the 39th International Conference on Computer-Aided Design*. 2020, pp. 1–9.
- [Van+20] Jessica Vandebon et al. “Artisan: A meta-programming approach for codifying optimisation strategies”. In: *2020 IEEE 28th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2020, pp. 177–185.
- [Vas+17] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).
- [Vas+18] Nicolas Vasilache et al. “Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions”. In: *arXiv preprint arXiv:1802.04730* (2018).
- [Vel+17] Petar Veličković et al. “Graph attention networks”. In: *arXiv preprint arXiv:1710.10903* (2017).
- [Wan+12] Xiaoli Wang et al. “An efficient graph indexing method”. In: *2012 IEEE 28th International Conference on Data Engineering*. IEEE. 2012, pp. 210–221.
- [Wan+16] Yangzihao Wang et al. “Gunrock: A high-performance graph processing library on the GPU”. In: *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. 2016.

- [Wan+18] Fanchao Wang et al. “Accelerating coverage directed test generation for functional verification: A neural network-based framework”. In: *Proceedings of the 2018 on Great Lakes Symposium on VLSI*. 2018, pp. 207–212.
- [Wan+20] Hao-nan Wang et al. “Deep reinforcement learning: a survey”. In: *Frontiers of Information Technology & Electronic Engineering* 21.12 (2020), pp. 1726–1744.
- [Wei+17] Xuechao Wei et al. “Automated systolic array architecture synthesis for high throughput CNN inference on FPGAs”. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. ACM. 2017, p. 29.
- [Wei+18] Xuechao Wei et al. “TGPA: Tile-grained pipeline architecture for low latency CNN inference”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.
- [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.
- [WHN19] Yu Wang, James C Hoe, and Eriko Nurvitadhi. “Processor assisted worklist scheduling for FPGA accelerated graph processing on a shared-memory platform”. In: *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE. 2019, pp. 136–144.
- [WLZ17] Shuo Wang, Yun Liang, and Wei Zhang. “FlexCL: An analytical performance model for OpenCL workloads on flexible FPGAs”. In: *Proceedings of the 54th Annual Design Automation Conference 2017*. 2017, pp. 1–6.
- [WM19] Junsong Wang and Steve Matinelli. *AccDNN*. <https://github.com/IBM/AccDNN>. 2019.
- [WS20] Zi Wang and Benjamin Carrion Schafer. “Machine learning to set meta-heuristic specific parameters for high-level synthesis design space exploration”. In: *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE. 2020, pp. 1–6.

- [WSH20] Xiongwei Wu, Doyen Sahoo, and Steven CH Hoi. “Recent advances in deep learning for object detection”. In: *Neurocomputing* 396 (2020), pp. 39–64.
- [Wu+18] Zhenqin Wu et al. “MoleculeNet: a benchmark for molecular machine learning”. In: *Chemical science* (2018).
- [Wu+20] Zonghan Wu et al. “A comprehensive survey on graph neural networks”. In: *IEEE transactions on neural networks and learning systems* 32.1 (2020), pp. 4–24.
- [Wu+22] Nan Wu et al. “High-level synthesis performance prediction using gnns: Benchmarking, modeling, and advancing”. In: *Proceedings of the 59th ACM/IEEE Design Automation Conference*. 2022, pp. 49–54.
- [WW92] Craig I Watson and Charles L Wilson. “NIST special database 4”. In: *Fingerprint Database, National Institute of Standards and Technology* (1992).
- [WXH22] Nan Wu, Yuan Xie, and Cong Hao. “IronMan-pro: Multi-objective design space exploration in HLS via reinforcement learning and graph neural network based modeling”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2022).
- [Xie+18] Zhiyao Xie et al. “RouteNet: Routability prediction for mixed-size designs using convolutional neural network”. In: *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2018, pp. 1–8.
- [Xila] AMD Xilinx. *Vivado Design Suite User Guide - High-Level Synthesis (UG902)*. URL: <https://docs.amd.com/v/u/en-US/ug902-vivado-high-level-synthesis>.
- [Xilb] Xilinx Vitis Libraries. [www.github.com/Xilinx/Vitis\\_Libraries](http://www.github.com/Xilinx/Vitis_Libraries).
- [Xu+18] Keyulu Xu et al. “Representation learning on graphs with jumping knowledge networks”. In: *International Conference on Machine Learning*. PMLR. 2018, pp. 5453–5462.

- [Xu+20] Pengfei Xu et al. “Autodnnchip: An automated dnn chip predictor and builder for both fpgas and asics”. In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2020, pp. 40–50.
- [Xyd+14] Sotirios Xydis et al. “SPIRIT: Spectral-aware Pareto iterative refinement optimization for supervised high-level synthesis”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34.1 (2014), pp. 155–159.
- [Yan+18] Xuan Yang et al. “DNN dataflow choice is overrated”. In: *arXiv preprint arXiv:1809.04070* (2018).
- [Yan+20a] Mingyu Yan et al. “Hygcn: A gcn accelerator with hybrid architecture”. In: *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE. 2020, pp. 15–29.
- [Yan+20b] Yifan Yang et al. “GraphABCD: Scaling out graph analytics with asynchronous block coordinate descent”. In: *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*. IEEE. 2020, pp. 419–432.
- [Yin+18a] Rex Ying et al. “Graph convolutional neural networks for web-scale recommender systems”. In: *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 2018, pp. 974–983.
- [Yin+18b] Zhitao Ying et al. “Hierarchical graph representation learning with differentiable pooling”. In: *Advances in neural information processing systems* 31 (2018).
- [YP] Tomofumi Yuki and Louis-Noël Pouchet. *PolyBench/C*. URL: <https://web.cs.e.ohio-state.edu/~pouchet.2/software/polybench/>.
- [Yu+18] Cody Hao Yu et al. “S2FA: An accelerator automation framework for heterogeneous computing in datacenters”. In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6.
- [Yue+20] Xiang Yue et al. “Graph embedding on biomedical networks: methods, applications and evaluations”. In: *Bioinformatics* 36.4 (2020), pp. 1241–1251.

- [YV15] Pinar Yanardag and SVN Vishwanathan. “Deep graph kernels”. In: *Proceedings of the 21th ACM SIGKDD international conference on knowledge discovery and data mining*. 2015, pp. 1365–1374.
- [YXD18] Cunxi Yu, Houping Xiao, and Giovanni De Micheli. “Developing synthesis flows without human knowledge”. In: *Proceedings of the 55th Annual Design Automation Conference*. 2018, pp. 1–6.
- [Zac+19] Georgios Zacharopoulos et al. “Compiler-assisted selection of hardware acceleration candidates from application source code”. In: *2019 IEEE 37th International Conference on Computer Design (ICCD)*. IEEE. 2019, pp. 129–137.
- [Zen+09] Zhiping Zeng et al. “Comparing stars: On approximating graph edit distance”. In: *Proceedings of the VLDB Endowment* (2009).
- [Zha+08] Zhiru Zhang et al. “AutoPilot: A platform-based ESL synthesis system”. In: *High-Level Synthesis: From Algorithm to Digital Circuit* (2008), pp. 99–112.
- [Zha+15] Chen Zhang et al. “Optimizing FPGA-based accelerator design for deep convolutional neural networks”. In: *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM. 2015, pp. 161–170.
- [Zha+17] Jieru Zhao et al. “COMBA: A comprehensive model-based analysis framework for high level synthesis of real applications”. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE. 2017, pp. 430–437.
- [Zha+18a] Chen Zhang et al. “Caffeine: Toward uniformed representation and acceleration for deep convolutional neural networks”. In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 38.11 (2018), pp. 2072–2085.
- [Zha+18b] Xiaofan Zhang et al. “DNNBuilder: An automated tool for building high-performance DNN hardware accelerators for FPGAs”. In: *Proceedings of the International Conference on Computer-Aided Design*. ACM. 2018, p. 56.

- [Zha+19] Ling Zhao et al. “T-gcn: A temporal graph convolutional network for traffic prediction”. In: *IEEE Transactions on Intelligent Transportation Systems* (2019).
- [Zha+22] Qian Zhang et al. “HeteroGen: transpiling C to heterogeneous HLS code with automated test generation and program repair”. In: *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. 2022, pp. 1017–1029.
- [Zha+23a] Shichang Zhang et al. “A Survey on Graph Neural Network Acceleration: Algorithms, Systems, and Customized Hardware”. In: *arXiv preprint arXiv:2306.14052* (2023).
- [Zha+23b] Wayne Xin Zhao et al. “A survey of large language models”. In: *arXiv preprint arXiv:2303.18223* (2023).
- [Zhe+20] Size Zheng et al. “Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system”. In: *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*. 2020, pp. 859–873.
- [Zho+14] Guanwen Zhong et al. “Design space exploration of multiple loops on FPGAs using high level synthesis”. In: *2014 IEEE 32nd international conference on computer design (ICCD)*. IEEE. 2014, pp. 456–463.
- [Zho+16] Guanwen Zhong et al. “Lin-analyzer: A high-level performance analysis tool for FPGA-based accelerators”. In: *Proceedings of the 53rd Annual Design Automation Conference*. 2016, pp. 1–6.
- [Zho+17] Guanwen Zhong et al. “Design space exploration of FPGA-based accelerators with multi-level parallelism”. In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 1141–1146.
- [Zhu+20] Fuzhen Zhuang et al. “A comprehensive survey on transfer learning”. In: *Proceedings of the IEEE* 109.1 (2020), pp. 43–76.

- [ZKP21] Bingyi Zhang, Rajgopal Kannan, and Viktor Prasanna. “BoostGCN: A Framework for Optimizing GCN Inference on FPGA”. In: *2021 IEEE International Symposium on FCCM*. 2021.
- [ZP20] Hanqing Zeng and Viktor Prasanna. “GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms”. In: *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2020, pp. 255–265.
- [ZPM18] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. “Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL”. In: *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. 2018, pp. 153–162.
- [Zuo+13] Wei Zuo et al. “Improving polyhedral code generation for high-level synthesis”. In: *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ ISSS)*. IEEE. 2013, pp. 1–10.
- [ZZP20] Bingyi Zhang, Hanqing Zeng, and Viktor Prasanna. “Hardware acceleration of large scale GCN inference”. In: *2020 IEEE 31st International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE. 2020, pp. 61–68.