UCLA
UCLA Electronic Theses and Dissertations

Title
Design Automation and Optimization for Memory-Bound Application Accelerators

Permalink
https://escholarship.org/uc/item/404825zp

Author
Chi, Yuze

Publication Date
2021

Peer reviewed|Thesis/dissertation

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

UNIVERSITY OF CALIFORNIA

Los Angeles

Design Automation and Optimization for Memory-Bound Application Accelerators

A dissertation submitted in partial satisfaction

of the requirements for the degree

Doctor of Philosophy in Computer Science

by

Yuze Chi

2021

© Copyright by

Yuze Chi

2021

ABSTRACT OF THE DISSERTATION

Design Automation and Optimization for Memory-Bound Application Accelerators

by

Yuze Chi
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2021
Professor Jason Cong, Chair

As we witness the breakdown of Dennard scaling, we can no longer get faster computers by shrinking transistors without increasing power density. Yet, the amount of data to be processed has never stopped growing. The limited power budget builds a "power wall" between the ever-increasing demand for computation and the available computer hardware, which forces computer scientists to seek not only performant, but also power efficient computation solutions, especially in data centers. Moreover, the wide performance gap between the computation units and the memory builds a "memory wall" and limits performance from another dimension.

In the past decade, field-programmable gate arrays (FPGAs) have been rapidly adopted in data centers, thanks to their low power consumption and the reprogrammability that assemble highly power-efficient accelerators for memory-bound applications. Meanwhile, C-based high-level synthesis (HLS) has been growing as the FPGA acceleration market grows, bringing "hard-to-program" FPGA accelerators to a broader community in many application domains. However, to create efficient customized accelerators, FPGA-related expertise is still required for the domain experts when they write HLS C. To make it worse, even for experienced FPGA programmers, C-based HLS is often less productive compared with higher-level software languages, especially

when an application cannot be easily programmed using the compiler directives designed for data-parallel programs.

This dissertation aims to address these two issues for domain-specific customizable accelerators for memory-bound applications with both regular and irregular memory access patterns. For memory-bound applications with regular memory accesses, we select stencil application as a representative for their complex data dependency that is challenging to optimize. We present SODA (Stencil with Optimized Dataflow Architecture) as a domain-specific compiler framework for FPGA accelerators. We show that by adopting theoretical analysis, model-driven design-space exploration, and domain-specific languages, programmers without FPGA expertise can build highly efficient stencil accelerators that outperform multi-thread CPUs by up to 3.3× with the memory bandwidth utilization improved by 1.65× on average. For memory-bound applications with irregular memory accesses, we select graph applications as a representative for their widespread presentation in various application domains. We first present TAPA (TAsk-PArallel) as a language extension to HLS, showing that convenient programming interfaces, universal software simulation, and hierarchical code generation can greatly improve productivity for task-parallel programs and reduce programmers' burden. We then extend our effort to support dynamically scheduled memory accesses to cover more applications and further improve productivity. Finally, we show with two case studies from real-world graph applications, i.e., single-source shortest path for neural image reconstruction and graph convolutional neural network for learning on graph structure, that customizable accelerators can achieve up to 4.9× speedup over state-of-the-art FPGA accelerators and 2.6× speedup over state-of-the-art multi-thread CPU implementation.

The dissertation of Yuze Chi is approved.

Anthony John Nowatzki

Glenn Reinman

Yizhou Sun

Jason Cong, Committee Chair

University of California, Los Angeles

2021

To my family.

In particular, to my parents, grandparents, and fiancée.

TABLE OF CONTENTS

1 Introduction **1**

1.1 Applications with Regular Memory Accesses 2

1.2 Applications with Irregular Memory Accesses 7

1.3 Summary 9

2 Background **12**

2.1 High-Level Synthesis 12

2.2 Memory-Bound Applications 14

2.2.1 Digital Image Processing 15

2.2.2 Numerical Analysis 16

2.2.3 Neural Image Reconstruction 17

2.2.4 Learning on Graph Structure 18

2.3 Domain-Specific Languages 19

2.3.1 Halide 19

2.3.2 HeteroCL 19

2.4 Related Work 21

2.4.1 Stencil Microarchitecture Templates 21

2.4.2 Common Subexpression Elimination 22

2.4.3 Image Processing Domain-Specific Languages 24

2.4.4 Task-Level Parallelism 26

2.4.5 Single-Source Shortest Path 33

3 Theoretical Analysis of Stencil Applications **38**

3.1 Optimal Communication Reuse 38

3.1.1 Definitions and Problem Formulation 38

3.1.2 SODA Microarchitecture 40

3.1.3 Communication Reuse Optimality 43

3.1.4 Dataflow Architecture 45

3.2 Optimal and Heuristic Computation Reuse 48

3.2.1 Definitions and Problem Formulation 48

3.2.2 Optimal Reuse by Dynamic Programming (ORDP) 49

3.2.3 Heuristic Search–Based Reuse (HSBR) 51

3.3 Summary 54

4 Model-Driven DSE for Stencil Applications **55**

4.1 Design-Space Exploration for Communication Reuse 55

4.1.1 Programming Model 56

4.1.2 Configurable Parameters 57

4.1.3 Resource Model 59

4.1.4 Performance Model 61

4.1.5 Design-Space Pruning 62

4.1.6 Experimental Evaluation 62

4.2 Design-Space Exploration for Computation Reuse 66

4.2.1 Total Reuse Distance for a Complex Stencil Kernel 67

4.2.2 Minimizing Total Reuse Distance 67

4.2.3 Experimental Results 68

4.3 Design-Space Exploration for Frequency/Area Trade-Off 74
4.3.1 No Module Merging 75
4.3.2 Fine-Grained Module Merging 75
4.3.3 Coarse-Grained Module Merging 76
4.3.4 Full-Scale Module Merging 76
4.3.5 Module Merging Based on Physical Hierarchy 77
4.3.6 Experimental Results 78
4.4 Summary 80

5 Enabling High-Level DSLs for Stencil Applications 81
5.1 Halide Compilation Flow 84
5.2 HeteroHalide Compilation Flow 84
5.2.1 Algorithm Transformation 85
5.2.2 Schedule Transformation 86
5.2.3 Extensions on Halide Schedules 89
5.3 Evaluation 91
5.3.1 Programming Efficiency 92
5.3.2 Accelerator Performance 93

6 Extending High-Level Synthesis for Task-Parallel Programs 96
6.1 Motivating Example 97
6.2 TAPA Framework 103
6.2.1 Programming Model and Interface 103
6.2.2 Hierarchical Finite-State Machine Model 103

6.2.3 Software Simulation . . . 108
6.2.4 RTL Code Generation . . . 110
6.2.5 TAPA Automation Overview . . . 111
6.3 Evaluation . . . 112
6.3.1 Benchmarks . . . 112
6.3.2 Lines of Kernel Code . . . 114
6.3.3 Lines of Host Code . . . 116
6.3.4 Software Simulation Time . . . 116
6.3.5 RTL Code Generation Time . . . 117
6.4 Alternative Compilation Paths . . . 118
6.5 Summary . . . 122

7 Supporting Dynamically Scheduled Programs . . . 123
7.1 Motivating Examples . . . 123
7.1.1 Dynamic Off-Chip Memory Accesses . . . 123
7.1.2 Dynamic On-Chip Memory Accesses . . . 124
7.2 Supporting Dynamic Off-Chip Memory Accesses . . . 125
7.3 Supporting Dynamic On-Chip Memory Accesses . . . 128
7.4 Summary . . . 132

8 Tackling Irregular Access Pattern: Graph Applications . . . 134
8.1 Single-Source Shortest Path . . . 134
8.1.1 The SPLAG Accelerator . . . 136
8.1.2 The SPLAG Algorithm . . . 137

8.1.3 The Coarse-Grained Priority Queue 139
8.1.4 The Customized Vertex Cache 149
8.1.5 The Edge Fetcher 152
8.1.6 Evaluation 153
8.1.7 Using SPLAG for Neural Image Reconstruction 162
8.2 Graph Convolutional Network 167
8.2.1 Theoretical Analysis 168
8.2.2 Practical Challenges 175
8.2.3 Evaluation 177
8.3 Summary 181

9 Conclusions 182

References 183

LIST OF FIGURES

1.1 Memory access pattern of the stencil kernel in Listing 1.1. Circles represent input data elements. Arrows represent the traversal order of the loop, which also indicates the movement of the sliding window (*stencil window*). The solid circles are being accessed by the current loop iteration, which composes the current stencil window. The shaded circles have been accessed by a previous iteration, and will be accessed in a future iteration as well. 3

1.2 For iterative stencil kernels, redundant computation increases linearly as the number of iteration increases because the border (halo) increases in each iteration. 4

1.3 In the 5-point Jacobi kernel shown in Listing 1.1 and Figure 1.1, inputs from two different loop iterations overlap with each other, and the overlap part performs the same computation in the two different iterations. This demonstrates a computation reuse opportunity. 5

1.4 A task-parallel accelerator design for Dijkstra's algorithm [61], which is challenging to implement using Vitis HLS alone. 8

1.5 Quadrants of memory-intensive applications [73, 101, 109, 119]. 10

2.1 Typical HLS compilation flow. 13

2.2 FPGA accelerator development flows with and without HLS. 14

2.3 The basic camera post-processing pipeline [146]. 15

2.4 Illustration of the Heat equation [137]. 16

2.5 Neurons reconstructed and segmented from the raw 3D image [119]. 17

2.6 Zachary's karate club network [174] and its embeddings obtained from a 3-layer GCN model [101]. 18

2.7 Overlapping pattern in a 5-point Jacobi kernel. 23

3.1 SODA reuse buffer microarchitecture. 42

3.2 Dataflow modules in a SODA microarchitecture. 46

3.3 Overview of a complete SODA accelerator. 47

3.4 Reduction trees of $a + b + c$ augmented from $a + b$. 49

3.5 Reduction tree of Formula 2.3. 50

3.6 Different operand selections on a 4×3 uniform-weight stencil kernel. 53

4.1 SODA automation framework. 58

4.2 Impact of heuristics in HSBR. 71

4.3 Performance of iterative kernels. 72

4.4 Resource usage reduction. The normalization baseline is SODA [20]. BRAM* excludes erosion and xcorr to avoid being biased. Truncated bars are marked with values. . . 73

4.5 Polynomial scalability of HSBR. Different points in the same shape correspond to different benchmarks. 73

4.6 Optimality gap of HSBR on 5281 artificial 3×3 kernels. 74

4.7 A SODA kernel with 3 PEs per iteration and 2 iterations in total. 75

4.8 Fine-grained module merging for the example in Figure 4.7. 75

4.9 Coarse-grained module merging for the example in Figure 4.7. 76

4.10 A SODA kernel with 3 PEs per iteration and 2 iterations in total. 76

4.11 Performance of iterative kernels with frequency improved by applying module merging and coarse-grained floorplanning (**bold** items in Table 4.6). 79

6.1 Example PageRank accelerator design. 98

6.2 TAPA automation flow overview. 111

6.3 LoC comparison for kernel code. Lower is better. 115

6.4 LoC comparison for host code. Lower is better. 115

6.5 Simulation time comparison. Lower is better. The sequential simulator fails to simulate `cannon` and `pagerank` correctly. The Intel OpenCL multi-thread simulator cannot simulate `gaussian` due to its large number of task instances. 116

6.6 RTL code generation time. Lower is better. 117

7.1 Synchronous off-chip memory accesses without burst. 125

7.2 Synchronous off-chip memory accesses with burst. 126

7.3 Asynchronous off-chip memory accesses. 126

8.1 Change of the number of active vertices as edges are traversed. The g500 graphs are power-law graphs and the road graphs are planar graphs. 135

8.2 Architecture overview of the SPLAG accelerator. 137

8.3 A graph with 5 vertices and 7 edges. 139

8.4 Sizes of 32 buckets as edges are traversed in the g500-15 dataset. Each line represents a bucket. 140

8.5 An example of the CGPQ orchestrating chunks of vertices. (a) Initially the CGPQ contains one chunk of four vertices stored off-chip and its reference stored on-chip. The on-chip reference stores the bucket number and a pointer to the off-chip memory position. (b) Another chunk of four vertices is added. The new chunk belongs to Bucket 0 and thus has higher priority. Therefore, it is stored on-chip at a position with a higher priority. The on-chip reference of the old chunk is moved to a position with a lower priority. Meanwhile, the off-chip memory only needs to append the newly added vertices without moving existing ones. (c) Another chunk for Bucket 5 is added. (d) There can be multiple chunks for the same bucket. (e) The chunk with the highest priority is removed. The chunk reference is popped from the on-chip priority queue and the pointer is used to read the vertices from the off-chip memory. On-chip chunk references are reorganized to maintain the priority queue structure while off-chip data are not moved. 141

8.6 A CGPQ with two push ports and two pop ports. The number of ports can be different for push and pop and is larger than two in the actual design (Table 8.3 on page 155). 142

8.7 The chunk buffer in Figure 8.6. Data paths in bold transfer multiple data elements in lockstep. 144

8.8 Data layout of the chunk buffer in Figure 8.7. Terminologies are summarized in Table 8.1. Each push port requires a bucket partition and each pop port requires a URAM bank, so there are two bucket partitions and each bucket partition has two banks. Each bucket buffer (BB) is used as a circular buffer. The numbers in brackets are the array indices of the vertices in each BB. 145

8.9 A customized vertex cache with two banks. Both the on-chip and off-chip memory are partitioned into banks. Vertices are cyclically assigned to each bank. The two switch networks route requests based on the bank ID. 150

8.10 The finite-state machine for memory reads in the CVC. *Hit* means the requested cache line has the same vertex ID as the requested vertex. *Miss* means the opposite of hit. The initial invalid state can only transfer to reading for updating because each vertex cannot be filtered before being updated first. Dirty and writing states can be managed independent of the states for reading and are not included in the figure. Miss on reading will stall the request until the request until the entry is no longer reading, so there is no state transition from reading states on miss. 151

8.11 Edge fetcher with two banks. Each bank stores edges whose vertices in the corresponding vertex partition. Bold lines are coalesced data paths that transfer multiple vertices in lockstep. 153

8.12 Percentage of spilled vertices among all vertices pushed to the CGPQ. 157

8.13 Percentage of CVC idling. Low idling suggests that the CGPQ can pop vertices with a high throughput. 157

8.14 Read and write hit rate of the CVC. 158

8.15 Percentage of active vertices among all traversed edges. The active vertices are either ① discarded by CVC filtering, or ② processed by the edge fetcher. The rest of traversed edges did not generate active vertices during CVC updating. 158

8.16 Throughput achieved by SPLAG. “Traversal” throughput measures the number of traversed edges over the kernel execution time. “Algorithm” throughput measures the number of *undirected* edges in the connected component over the kernel execution time. 160

8.17 Normalized amount of work achieved by SPLAG. This is defined as the number of traversed edges divided by the number of *directed* edges in the connected component. Lower is better. Some benchmarks have < 1 amount of work because of the *never-look-back* optimization (Section 8.1.2). 161

8.18 A complete neural image reconstruction pipeline with Recut [124]. 162

8.19 Sparse representation of a 3D neural image. 163

8.20 SPLAG edge data layout. 163

8.21 Recut [124] pipeline with SPLAG integrated for the SSSP part. 164

8.22 Execution time breakdown for Recut with SPLAG integration. 165

8.23 Execution time breakdown with varying number of active voxels. 166

8.24 Example of single phase update. Both B and C are partitioned into $p = 4$ partitions, and A is partitioned into $p^2 = 16$ partitions accordingly. Partitions of A are streamed from the off-chip memory while partitions of B and C are preloaded or pre-allocated on-chip. By switching partitions of B first, each partition of C is written to the off-chip memory only once. 169

8.25 Example of dual-phase update. Both B and C are partitioned into $p = 4$ partitions, and A is partitioned into $p^2 = 16$ partitions accordingly. (a) In the scatter phase, partitions of A are streamed from the off-chip memory, partitions of B are preloaded on-chip, while the resulting partial sums corresponding to the partition of A are streamed to the off-chip memory. (b) In the gather phase, partial sums corresponding to the partitions of A are streamed from the off-chip memory while partitions of C are pre-allocated on-chip. By storing the partial sums off-chip, each partition of B and C are only accessed once in the off-chip memory. 170

8.26 Example of direct update. Matrix C is partitioned into $p = 4$ partitions, and matrix A is partitioned accordingly. Matrix B is not partitioned and is read directly and randomly from the off-chip memory without using a scratchpad memory while A is streamed from the off-chip memory. Partitions of C are pre-allocated on-chip and are written to the off-chip memory when the corresponding partition of A is finished. Each partition of A and C are only accessed once in the off-chip memory, but B is accessed m/n times on average at a reduced memory bandwidth (because random access bandwidth is lower than sequential access bandwidth). 171

8.27 Out-of-order scheduling to handle read-after-write dependency [158]. 176
8.28 Throughput (a) and execution time (b) of SpMM on K80, Sextans, V100, and Sextans-P, with varying problem sizes (number of floating-point operations). 179

LIST OF TABLES

2.1 Summary of related work of TAPA. 32

2.2 Summary of related work for SSSP. MTEPS measures the algorithm throughput, which is defined as the number of undirected edges in the connected component divided by the execution time. Since some systems did not report the execution time, an upper-bound estimation (Section 8.1.6.4) is listed here. 37

4.1 Benchmarks used for communication reuse. 63

4.2 Average error rate of model prediction. 64

4.3 Performance comparison of SODA and previous work. 65

4.4 Stencil benchmarks used in the experiments. 69

4.5 Operation reduction. **Bold** items are verified to be optimal. 70

4.6 Comparison among different module merging strategies. **Bold** items indicate the best (lowest LUT usage or highest clock frequency) merging strategy of that benchmark. 78

5.1 Schedule primitives and the corresponding transformation methods supported by Halide vs HeteroHalide. 90

5.2 Applications used in the evaluation. 91

5.3 LoC at different levels in the flow. HeteroHalide and HeteroCL counts are algorithm + schedule. Numbers in parentheses are ratios over HeteroHalide. 92

5.4 LoC comparison between HeteroHalide and Xilinx xfOpenCV Library [170]. HeteroHalide counts are algorithm + schedule. xfOpenCV counts include only the core functions, not the utility libraries. Numbers in parentheses are ratios over HeteroHalide. 93

5.5 Accelerators generated by HeteroHalide compared with plain Halide on 28 CPU cores. 94

5.6 Performance comparison between HeteroHalide and Halide-HLS. 95

6.1 Task-parallel programming models. 104

6.2 TAPA communication interface. 106

6.3 Number of tasks and channels in each benchmark. Each task may be instantiated multiple times, so the number of task instances is greater than the number of tasks. . 113

7.1 TAPA asynchronous memory-mapped interface. 127

7.2 Summary of the TAPA dependency detector API. 130

7.3 Comparison of naïve and optimized versions of the micro-benchmarks with dynamic memory accesses. 132

8.1 Terminologies used in the chunk buffer. 143

8.2 Graph datasets evaluated on SPLAG. 154

8.3 Design parameters of the SPLAG accelerator. 155

8.4 Post-implementation results of the SPLAG accelerator on U280. 156

8.5 SPLAG compared against other SSSP systems. 162

8.6 Random access bandwidth on the Alveo U280 FPGA. 172

8.7 Comparison of off-chip memory accesses among SPU, DPU, and DU on 8 real-world datasets used in [176] and 4 synthetic datasets. **Bold** items have the minimum memory accesses on DDR-based systems and underlined items have the minimum memory accesses on HBM-based systems. 174

8.8 Hardware platforms used for SpMM evaluation. 178

8.9 Speedup of Sextans [158] on a U280 FPGA over 32 CPU threads running at 4.4GHz. The kernel speedup measures only the FPGA computation time. The end-to-end (E2E) speedup considers the communication overhead imposed by the offloaded computation kernel. The total speedup measures the speedup of the whole GCN inference, including computation that is not offloaded to the FPGA, e.g., the nonlinear activation function (ReLU). 180

ACKNOWLEDGMENTS

I appreciate my privilege to pursue my doctoral degree. I am especially honored that my doctoral research was advised by Professor Jason Cong. When I decided to start this journey in March 2016, I was told that "Jason Cong is a very bright guy", and it was a great decision to join his group. At that time, I did not know, "bright" meant the sharpest insights in his field, the most patient guidance on his students, and the highest level of dedication to his career. Professor Cong did not just lead me to the research field of design automation and optimization, but also taught me the important skills to deliver my research to the audience. I deeply thank Professor Cong for his supervision in the past five and a half years, and will continue to live up to what he has demonstrated to me for the rest of my life.

I would like to express my appreciation to my doctoral committee members, Professor Tony Nowatzki, Professor Glenn Reinman, and Professor Yizhou Sun, for their time, patience, and suggestions to improve the quality of this dissertation.

This dissertation would have been impossible without the close collaborations with my colleagues. In particular, I thank

- Peipei Zhou, Peng Wei, Jiajie Li, Cody Hao Yu, Jie Wang, Zhe Chen, Professor Zhiru Zhang, and his student Yi-Hsiang Lai, for the collaboration on the SODA project;
- Licheng Guo, Jason Lau, Young-kyu Choi, and Jie Wang, for the collaborative effort on the TAPA project, especially with the usage of HBM;
- Karl Marrett and Licheng Guo, for their support on the SSSP acceleration project;
- Professor Yizhou Sun and her students Yunsheng Bai and Ziniu Hu, for teaching me the basics of graph convolutional networks, and Atefeh Sohrabizadeh and Linghao Song, for the joint effort on the GCN acceleration project.

I appreciate Di Wu, Joel Coburn, and Peng Wei, for hosting my internships at Falcon Computing Solutions and Google LLC.

Jason Lau did an excellent job maintaining our server infrastructures, and I really appreciate it. Without his effort, we would have been using a much slower cluster with a much higher maintenance burden.

I would like to express my gratitude to Alexandra Luong for her help with my graduate student paper work and maintaining miscellaneous supplies in the lab.

I would like to thank Janice Martin-Wheeler and Marci Baun for editing my manuscript submissions and improving my writing skills.

This dissertation is partially supported by the NSF/Intel CAPA program (CCF-1723773), AWS credits from Amazon, the NSF RTML program (CCF1937599), a Google Faculty Award, the NIH Brain Initiative (U01MH117079), the Xilinx Adaptive Compute Clusters (XACC) program, CRISP, one of the six centers of JUMP, and the contributions from the CDSC industrial partners.

VITA

2012–2016 B.E. in Electronic Engineering, Tsinghua University, Beijing.

PUBLICATIONS

Y. Chi, L. Guo, J. Cong. Accelerating SSSP for Power-Law Graphs. In *FPGA*, 2022.

L. Guo, P. Maidee, Y. Zhou, C. Lavin, J. Wang, **Y. Chi**, W. Qiao, A. Kaviani, Z. Zhang, J. Cong. RapidStream: Fast HLS-to-Bitstream Timing Closure through Parallelized and Physical-Integrated Compilation. In *FPGA*, 2022.

L. Song, **Y. Chi**, A. Sohrabizadeh, Y. Choi, J. Lau, J. Cong. Sextans: A Streaming Accelerator for General-Purpose Sparse-Matrix Dense-Matrix Multiplication. In *FPGA*, 2022.

Y. Chi, L. Guo, J. Lau, Y. Choi, J. Wang, J. Cong. Extending High-Level Synthesis for Task-Parallel Programs. In *FCCM*, 2021.

L. Guo, **Y. Chi**, J. Wang, J. Lau, W. Qiao, E. Ustun, Z. Zhang, J. Cong. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *FPGA*, 2021. (**Best Paper Award**)

Y. Choi, **Y. Chi**, W. Qiao, N. Samardzic, J. Cong. HBM Connect: High-Performance HLS Interconnect for FPGA HBM. In *FPGA*, 2021.

L. Guo, J. Lau, **Y. Chi**, J. Wang, C. H. Yu, Z. Chen, Z. Zhang, J. Cong. Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency. In *DAC*, 2020.

Y. Chi, J. Cong. Exploiting Computation Reuse for Stencil Accelerators. In *DAC*, 2020.

J. Li, **Y. Chi**, J. Cong. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. In *FPGA*, 2020.

Y. Choi, **Y. Chi**, J. Wang, J. Cong. FLASH: Fast, ParalleL, and Accurate Simulator for HLS. In *TCAD*, 2020.

Y. Lai, **Y. Chi**, Y. Hu, J. Wang, C. H. Yu, Y. Zhou, J. Cong, Z. Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *FPGA*, 2019. (**Best Paper Award**)

Y. Chi, Y. Choi, J. Cong, J. Wang. Rapid Cycle-Accurate Simulator for High-Level Synthesis. In *FPGA*, 2019.

Y. Chi, J. Cong, P. Wei, P. Zhou. SODA: Stencil with Optimized Dataflow Architecture. In *ICCAD*, 2018. (**Best Paper Nominee**)

G. Dai, **Y. Chi**, Y. Wang, H. Yang. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *FPGA*, 2016.

Y. Chi, G. Dai, Y. Wang, G. Sun, G. Li, H. Yang. NXgraph: An Efficient Graph Processing System on a Single Machine. In *ICDE*, 2016.

CHAPTER 1

Introduction

The demand for computing power has been ever-increasing in the past few decades. Yet, the technology scaling [58] that drives Moore's Law has been slowing down significantly, and the "power wall" created by heat dissipation has been limiting the growth of parallel computing. Another significant limiting factor is the "memory wall" [169], which is created by the slower scaling of memory performance than that of processors. Communication among computation units and the memory system becomes the performance bottleneck in many applications domains. As such, domain-specific customization has been adopted in more and more areas, for its highly customizable microarchitecture and the high power efficiency and memory utilization brought by the customization. Field-programmable gate arrays (FPGA) have been playing a key role in the rapid adoption of customizable accelerators in data centers [34, 88, 145] for their reprogrammability. In the meantime, FPGA accelerator development productivity has been greatly improved by the latest advances and adoption of C-based high-level synthesis (HLS) [37, 41, 76, 77], bringing "hard-to-program" FPGA accelerators to a broader community.

However, creating efficient accelerators is still non-trivial for programmers. The current C-based HLS relies highly on the programmer-inserted compiler directives ("pragmas") to achieve good quality of result (QoR). Learning to use these compiler directives requires background knowledge of the underlying microarchitecture, which takes a long time for the experts in each application domain. Worse still, even for experienced FPGA programmers, C-based HLS is often less productive compared with higher-level software languages, especially for memory-bound applications that require design-space exploration and optimizations beyond what compiler di-

rectives have to offer. In this dissertation, we focus on memory-intensive applications with both regular and irregular memory accesses, and address the design automation and optimization challenges.

1.1 Applications with Regular Memory Accesses

Even for applications bounded by very regular memory accesses, creating efficient accelerators can still be challenging. Take stencil computation [50] as an example. Stencil computation is often intuitively defined as the type of computation that uses a sliding window of the input array to compute the output array. Such computation patterns are widely used in many areas, including image processing [9, 15] and solving partial differential equations [179]. Listing 1.1 shows a 2-dimensional 5-point stencil kernel, which can be used for blurring images or solving the Jacobi equation numerically [179]. Figure 1.1 shows the corresponding memory access pattern.

```
1 void Jacobi(float input[N][M], float output[N][M]) {
2 for (int j = 1; j < N - 1; ++j)
3   for (int i = 1; i < M - 1; ++i)
4     output[j][i] = (input[j-1][i] + input[j][i-1] + input[j][i] + input[j][i+1] +
5                     input[j+1][i]) * 0.2f;
6 }
```

Listing 1.1: A 2-dimensional 5-point Jacobi kernel.

Although memory accesses in a stencil kernel are very regular (Figure 1.1) and can be statically determined at compilation time, it is non-trivial to optimize for performance, due to the high communication/computation ratio [122] and the memory system that cannot keep up with the computation units. Moreover, a stencil computation kernel can be composed of several stages or performed iteratively, which further complicates data dependency and makes communication optimizations harder to achieve. Researchers have been optimizing stencil kernels in the following three aspects:

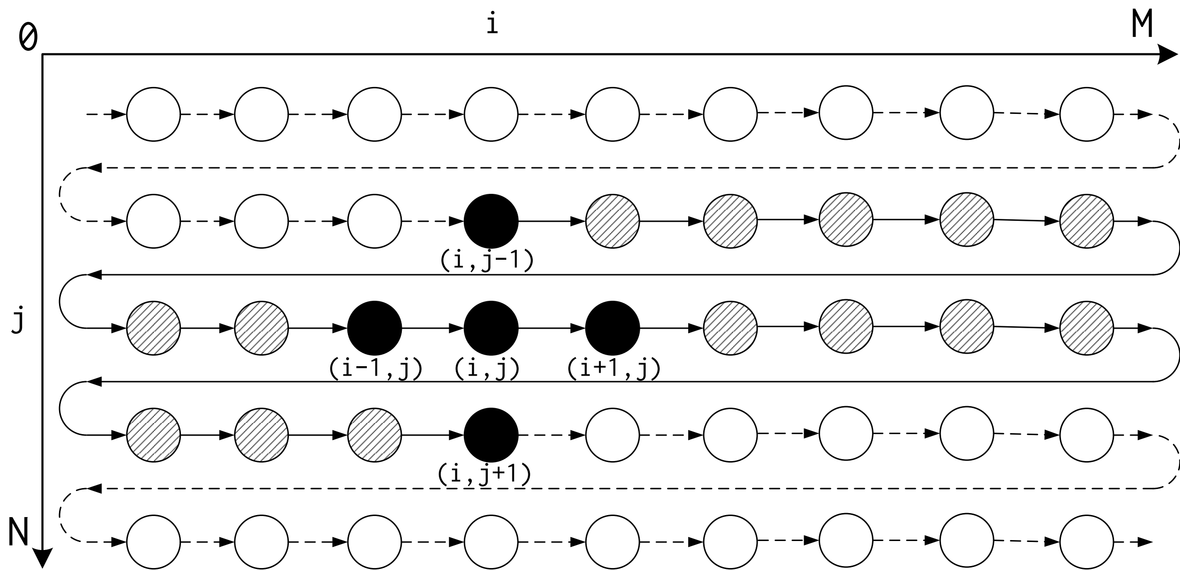


Figure 1.1: Memory access pattern of the stencil kernel in Listing 1.1. Circles represent input data elements. Arrows represent the traversal order of the loop, which also indicates the movement of the sliding window (*stencil window*). The solid circles are being accessed by the current loop iteration, which composes the current stencil window. The shaded circles have been accessed by a previous iteration, and will be accessed in a future iteration as well.

- **Parallelization**. Stencil computation has a large degree of inherent parallelism, including both *spatial parallelism*, i.e., parallelism among spatial elements within a stage or iteration, and *temporal parallelism*, i.e., parallelism among multiple temporal stages or iterations. However, the complex dependencies among elements in different stages make it hard to fully utilize the available parallelism [7, 85, 105, 125, 179].

- **Communication reuse**. The sliding window pattern of stencil kernels makes it possible to reuse data and reduce memory communication. On instruction-based processors (CPU, GPU), this translates into improving locality [2, 179] and reducing inter-core communication [168]. On accelerators (FPGA, ASIC) where data paths can be fully customized, Cong et al. [35, 36] present a microarchitecture that uses the least-possible buffer size for each

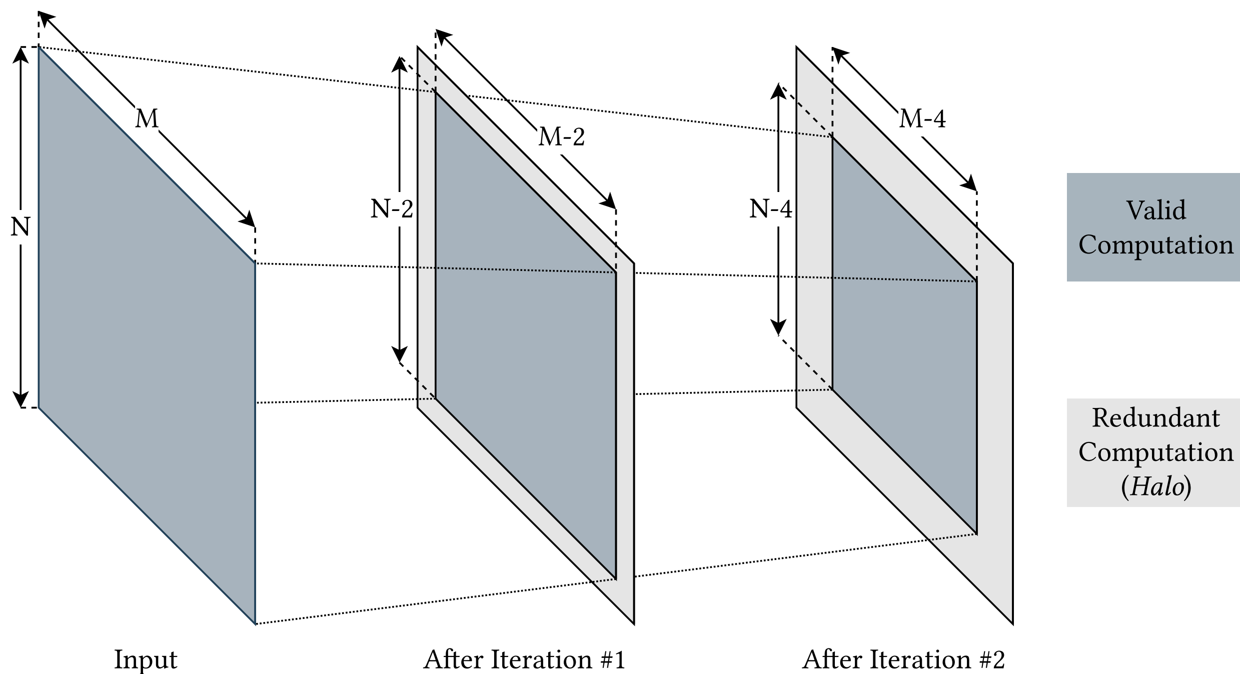


Figure 1.2: For iterative stencil kernels, redundant computation increases linearly as the number of iteration increases because the border (halo) increases in each iteration.

processing element (PE), achieving full communication reuse but without considering parallelization.

- **Computation reuse**. For stencil kernels that are composed of multiple stages or iterations, full communication reuse for the whole kernel is often impossible due to limited computational resources. Fortunately, almost all such stencil kernels perform commutative and associative reduction operations, thus making it possible to reuse some computation [15, 43, 55, 62, 63, 106]. As a motivating example, for a 17×17 kernel used in calcium image stabilization [15], the number of multiplication operations can be dramatically reduced from 197 to only 30, while yielding the same throughput. However, that design was done with extensive manual optimization. Our goal is to automate such optimization process.

Ideally, a domain-specific stencil compiler should be able to optimize all the three aspects col-

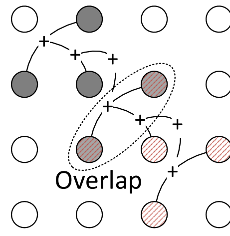


Figure 1.3: In the 5-point Jacobi kernel shown in Listing 1.1 and Figure 1.1, inputs from two different loop iterations overlap with each other, and the overlap part performs the same computation in the two different iterations. This demonstrates a computation reuse opportunity.

laboratively. However, there are three major challenges that have not been addressed thoroughly in state of the art for stencil kernels. The first challenge is that existing accelerator designs are suboptimal when multiple PEs are used for a single stage. Existing stencil accelerators [166, 183] replicate the on-chip buffers along with the PEs to enable concurrent accesses. With a buffer size proportional to the number of PEs, both the maximum achievable number of processing elements (PEs) and the maximum achievable input tile size are suboptimal. Suboptimal number of PEs will under-utilize the computation resources and therefore results in suboptimal performance. Suboptimal input size causes performance loss due to the fact that the borders (halos) of stencil kernels need to be retransmitted. The latter problem is especially severe for 3D or even higher dimensional kernels, since their halos take larger portion of the input size [183]. When temporal parallelism (multiple iterations) are implemented at the same time, this becomes even worse since the halo size increases linearly as the number of iteration increases [183]. This is illustrated in Figure 1.2.

The second challenge is that computation reuse is not thoroughly explored. Most stencil compilers [43, 55, 62, 63, 106] are designed for instruction-based processors and do not explore the complete design space for computation reuse, due to the fact that parallelization and communication reuse have more impact on performance and computation reuse is often just a by-product [2, 179]. However, for accelerators, computation reuse can be fully decoupled from parallelization and communication reuse via data path customization. An ideal stencil compiler for

```
void Module1Func(hls::stream<float>& input_q, hls::stream<float>& output_q) {
  ap_uint<9> ptr_delay_512 = 0;   // M = 512
  ap_uint<1> ptr_delay_1 = 0;
  ap_uint<9> ptr_delay_511 = 0;
  float input_delayed_512_buf[512];   // Communication reuse buffer
  float input_delayed_513_buf[1];
  float rc_delayed_511_buf[511];
  for (;;) {  // Termination condition left out for brevity
#pragma HLS pipeline II = 1
#pragma HLS dependence inter true variable = input_delayed_512_buf distance = 512
#pragma HLS dependence inter true variable = input_delayed_513_buf distance = 1
#pragma HLS dependence inter true variable = rc_delayed_511_buf distance = 511
    if (!input_q.empty()) {   // Required for flushing the pipeline
      const float input = input_q.read();
      const float input_delayed_512 = input_delayed_512_buf[ptr_delay_512];
      const float input_delayed_513 = input_delayed_513_buf[ptr_delay_1];
      const float rc_delayed_511 = rc_delayed_511_buf[ptr_delay_511];
      const float rc = input_delayed_513 + input;   // Reused computation
      output_q.write((rc_delayed_511 + rc + input_delayed_512) * 0.2f);
      input_delayed_512_buf[ptr_delay_512] = input;
      input_delayed_513_buf[ptr_delay_1] = input_delayed_512;
      rc_delayed_511_buf[ptr_delay_511] = rc;
      ptr_delay_512 < 511 ? (++ptr_delay_512) : (ptr_delay_512 = 0);
      ptr_delay_1 < 0 ? (++ptr_delay_1) : (ptr_delay_1 = 0);
      ptr_delay_511 < 30 ? (++ptr_delay_511) : (ptr_delay_511 = 0);
    }   // if not empty
  }   // for
}   // Module1Func
```

Listing 1.2: Optimized HLS code for the kernel in Listing 1.1 with communication reuse and computation reuse but without parallelization. Code that handles off-chip memory accesses and program termination is left out for brevity. With parallelization, the lines of code will increase further.

accelerators should be capable of finding the optimal computation reuse if possible. Moreover, since no stencil compiler uses an accelerator-oriented model to evaluate the computation-storage trade-off, it is hard to guide the design-space pruning and find the best solution.

The third challenge that has not been thoroughly addressed is the lack of complete automation and systematic design-space exploration. Due to the difficulty of programming FPGAs, vast design space, and high time-consumption of logic synthesis, having a fully automated design flow and analytical model-based design-space exploration is crucial. Many existing works are either manually designed or template-based, which lacks the flexibility of designing various stencil kernels agilely [166, 183]. Moreover, the complexity of the source code can quickly grow intractable. Listing 1.2 shows the computation part of optimized HLS code for the 6-line kernel code in Listing 1.1, which is already 4.7× longer that the original kernel with parallelization and off-chip memory optimizations being omitted. Although domain-specific languages (DSL) have been developed to facilitate stencil accelerator design on FPGAs [82, 144, 147], there still lacks a systematic approach to model the resource and performance, explore the design space, and optimize for performance.

1.2 Applications with Irregular Memory Accesses

For applications that are bounded by irregular memory accesses, e.g., graph applications, we no longer have the luxury to analyze and schedule memory accesses statically at compile/synthesis time. Instead, the design and optimization heavily relies on task-level parallelism, where computation units run different programs on different data. On instruction-based processors, such applications are often bottlenecked by the communication among the heterogeneous computation units and the memory system. The memory system designed for general homogeneous workload is usually poorly utilized for task-parallel programs. In contrast, customizable accelerators are advantageous because programmers can customize the data paths and schedule many accesses in parallel to improve memory utilization. However, even for algorithms as simple as Dijkstra's algorithm [61], the current HLS tools are greatly limited for task-parallel programs due to the following reasons:

- **Poor programmability**. Due to the lack of convenient application programming inter-

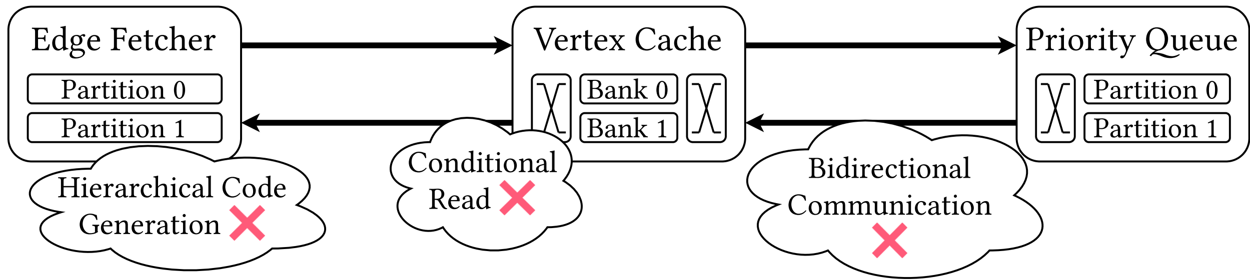


Figure 1.4: A task-parallel accelerator design for Dijkstra's algorithm [61], which is challenging to implement using Vitis HLS alone.

faces (API), programmers are often forced to write more code than they have to. For example, in Figure 1.4, multi-stage switch networks are used to route data among different partitions. Such a network switch needs to read packets conditionally based on their content and the availability of output ports. Without an API to read packets without consuming them (a.k.a., "peek") from the ports, programmers have to manually and carefully create a buffer and maintain a small state machine to keep track of incoming packets. This not only elongates the development cycle, but also makes the code error-prone.

- **Restricted software simulation**. As the key to fast correctness verification, software simulation is not always available to task-parallel programs. For example, Vitis HLS does not support debugging the accelerator shown in Figure 1.4 via software simulation due to the bidirectional communication between the vertex memory and the priority queue, while Intel OpenCL does not support more than 256 concurrent kernels [90] in software simulation. Lack of fast software simulation forces programmers to resort to RTL simulation for correctness verification, significantly elongating the development cycle.

- **Slow code generation**. We found that current HLS compilers view task-parallel code as a monolithic design and processes each instance of the same task as if they are different. For designs that instantiate the same task multiple times (e.g., in a systolic array or different partitions in each component shown in Figure 1.4), this leads to repetitive compilation on

each task and unnecessarily slows down code generation. One may argue that programmers can manually synthesize tasks separately and instantiate them in RTL, but doing so requires debugging RTL code, which is time-consuming and error-prone. We think such processes should be automated by the compiler.

Limited productivity for task-parallel programs significantly elongates the development cycles and undermines the benefits brought by HLS. One may argue that programmers should always go for data-parallel implementations when designing FPGA accelerators using HLS, but data-parallelism may be inherently limited, for example, in applications involving graphs. Moreover, researches show that even for data-parallel applications like neural networks [38, 163] and stencil computation [19, 20], task-parallel implementations show better scalability and higher frequency than their data-parallel counterparts due to the localized communication pattern [40]. In fact, at least 6 papers [53, 93, 118, 151, 157, 173] among the 28 research papers published in the ACM FPGA 2020 conference use task-parallel implementation with HLS, and another 3 papers [12,140,175] use RTL implementation that would have required task-parallel implementation if written in HLS.

1.3 Summary

In this dissertation, we aim to address the design automation and optimization challenges for memory-bound applications with both regular and irregular access patterns. For applications with regular memory access patterns, we use stencil applications as a case study. We show that with theoretical analysis, model-driven exploration, and high-level domain-specific languages, the complexity for scheduling memory accesses can be completely hidden by the compiler, and domain exports can program efficient stencil accelerators without having to know about the design and optimization techniques. For applications with irregular memory access patterns, we first present a highly-productive HLS language extension named TAPA to enable agile development for mapping such applications to accelerators. TAPA can support irregular applications

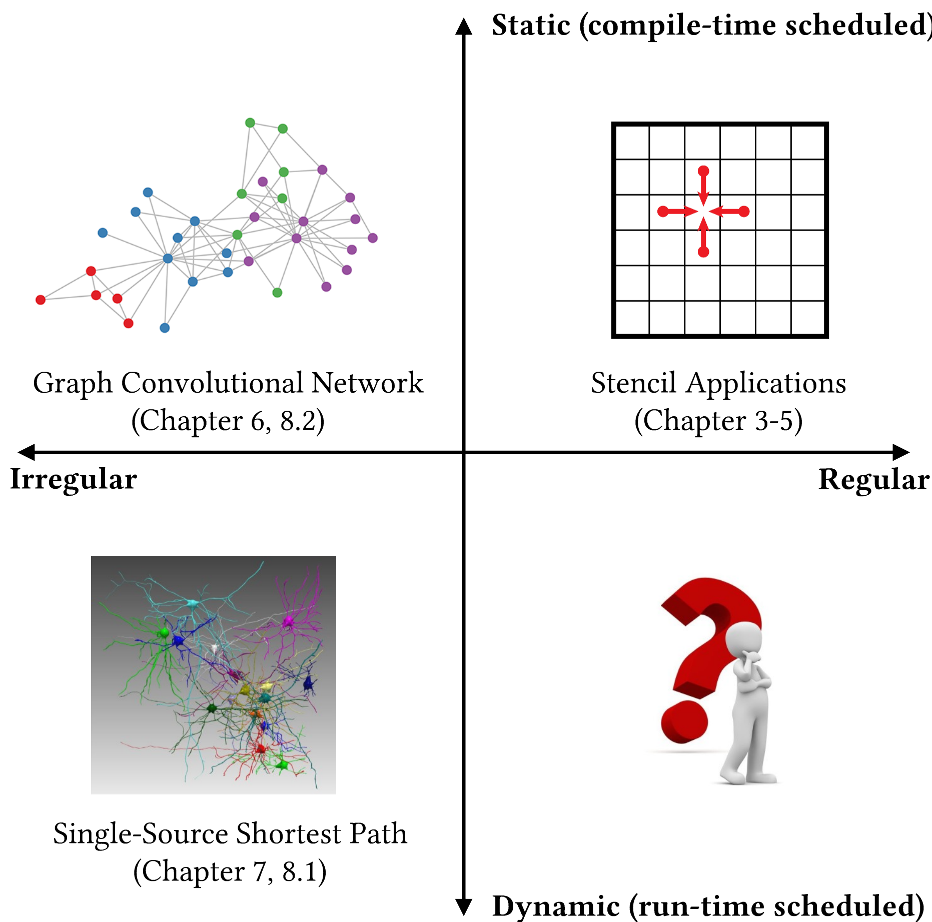


Figure 1.5: Quadrants of memory-intensive applications [73, 101, 109, 119].

by reordering and scheduling memory accesses statically at compile-time. For applications with less compile-time knowledge, we further extend TAPA to support dynamically scheduling memory requests. We use real-world graph applications to demonstrate the high productivity and decent quality of result of TAPA. Figure 1.5 visualizes the three quadrants of memory-intensive applications. For applications with regular memory accesses, the regular access pattern generally enables compile-time scheduling. Therefore, we leave out the fourth quadrant for the scope of this dissertation.

The remainder of the dissertation is organized as follows. Chapter 2 introduces the background of the high-level synthesis (HLS) technology, real-world applications motivating the two

application domains, and the summary of related work. Chapter 3 analyzes the design and optimizations for stencil applications from a theoretical point of view. Under the guidance of the theoretical analysis, Chapter 4 presents model-driven design-space exploration (DSE) for the stencil applications. Chapter 5 raises the abstraction level and presents an end-to-end approach compiling stencil applications from a popular image-processing domain-specific language to efficient hardware accelerators. Chapter 6 presents an HLS language extension that significantly improves the productivity for task-parallel accelerator design. To cover more applications and further improve productivity, Chapter 7 extends the effort to support dynamically scheduled memory accesses. Leveraging the tools presented in previous chapters, Chapter 8 presents two case studies of task-parallel programs. Chapter 9 concludes the dissertation.

CHAPTER 2

Background

This chapter presents background information and related work for this dissertation. We first introduce the high-level synthesis (HLS) technology that significantly boosts the productivity for FPGA accelerator design in Section 2.1. Section 2.2 briefly introduces the real-world memory-bound application drivers, while Section 2.3 introduces two relevant domain-specific languages. Section 2.4 summarizes the related work for this dissertation.

2.1 High-Level Synthesis

A field-programmable gate array (FPGA) is a piece of integrated circuit that contains reprogrammable building blocks. Typically, such building blocks include lookup tables (LUT), flip-flops (FF), digital signal processors (DSP), and block random access memories (BRAM). The functionalities of each block, as well as their interconnect, are all reprogrammable, making it possible to customize a single piece of circuit into accelerators that are tailored to different applications. Such hardware reconfiguration capability gives full control over computation and data paths without the overhead of general-purpose instructions, lowering the "power wall" and "memory wall" for a broad class of application kernels.

A notable disadvantage of FPGA accelerators, i.e., the poor programmability, however, has impeded their wide adoption for decades. The conventional programming abstraction of FPGA devices is the register-transfer level (RTL) description, which not only requires programmers to design and optimize the functionality of the program, but also forces programmers to manu-

ally break and schedule the functionalities into several pipelines stages. In contrast, software programmers only needed to specify the behavior of their programs with very few details of hardware abstraction, making the development and iteration cycles of software programs significantly shorter than that of hardware accelerators. As such, FPGA researchers and vendors have been adopting the high-level synthesis (HLS) technology [37] in the past decade, which relieves programmers from having to manually plan and optimize the timing of their circuit design. A typical HLS compilation flow is shown in Figure 2.1. The HLS code is parsed and compiled by the compiler first, which is then used for resource allocation, binding, and scheduling. RTL code is generated as the output of HLS compilation. A report containing resource consumption and performance estimation is usually generated together with the RTL code, providing guidelines for quality of result (QoR) assessment and optimizations.

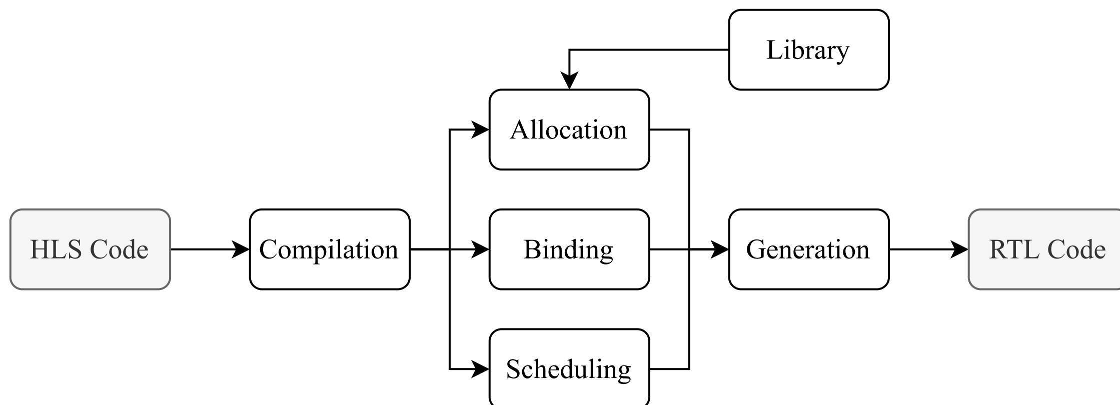
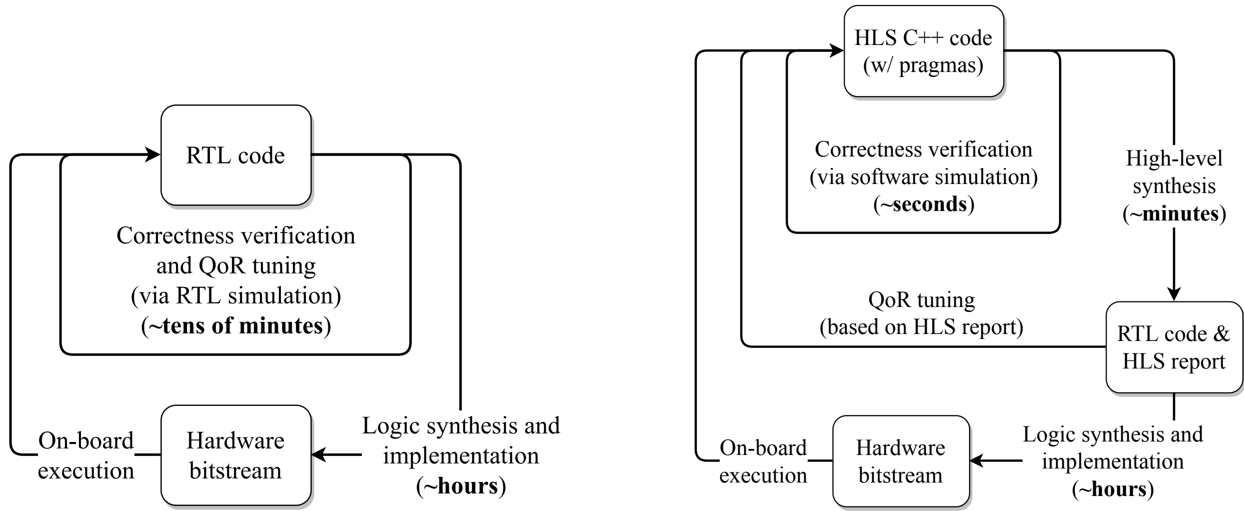


Figure 2.1: Typical HLS compilation flow.

Compared with the traditional RTL paradigm (Figure 2.2a) where programmers often spend tens of minutes just to verify the correctness of a code modification, with HLS, programmers can follow a rapid development cycle (Figure 2.2b). Programmers can write code in C and leverage fast software simulation to verify the functional correctness. Such a correctness verification cycle can take as few as just 1 second, allowing functionalities to be iterated at a fast pace. Once the HLS code is functionally correct, programmers can then generate RTL code, evaluate the quality of result (QoR) based on the generated performance and resource reports, and modify the HLS code



(a) FPGA accelerator development flow without HLS. Programmers often spend tens of minutes after code modification to evaluate the correctness and quality of result.

(b) FPGA accelerator development flow with HLS. Programmers spend seconds after code modification to verify the correctness. Quality of result can usually be obtained in less than 10 minutes from the HLS report.

Figure 2.2: FPGA accelerator development flows with and without HLS.

accordingly. Such a QoR tuning cycle typically takes only a few minutes. Thanks to the advances in HLS scheduling algorithms [16, 17, 41, 79, 86] and timing optimizations [14, 77, 96], HLS can not only shorten the development cycle, but also generate programs that are often competitive in cycle count [39], and more recently in clock frequency as well [77]. Moreover, FPGA vendors provide host drivers and communication interfaces for kernels designed in HLS [90, 171], further reducing programmers' burden to integrate and offload workload to FPGA accelerators.

2.2 Memory-Bound Applications

Due to the slower performance scaling of memory devices than that of computation units in the past few decades, today the main memory is generally much slower than the CPU. Modern CPU thus heavily relies on fast but tiny on-chip storage to keep its fast operating pace. With

such a hierarchical memory system, many applications would run faster if the memory system becomes faster. Such applications are called *memory-bound applications*. Low *computational intensity*, which can be defined as the ratio between the number of basic computation operations and the amount of data moved between fast and slow memory, makes applications memory-bound, because the fast computation units have to frequently stop and wait for the slow memory system. The large performance gap between the memory and the CPU makes many important real-world applications memory-bound. This section briefly introduces four memory-bound application drivers that motivate our research. Section 2.2.1 and Section 2.2.2 introduce two stencil applications whose memory accesses are regular, while Section 2.2.3 and Section 2.2.4 introduce two graph applications whose memory accesses are irregular.

2.2.1 Digital Image Processing

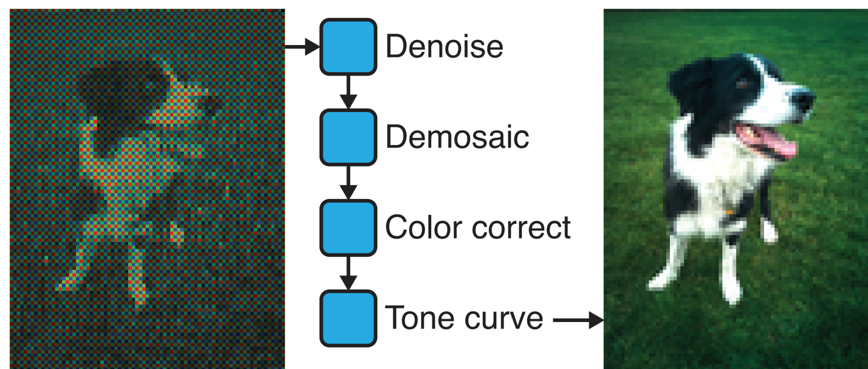


Figure 2.3: The basic camera post-processing pipeline [146].

The advances of computer technology has made cameras and other visual sensors ubiquitous in modern life: smartphones, tablets, and self-driving cars, to name a few. Lying in the core of these is digital image processing, where the visual images are quantized and converted into multidimensional arrays of digital signals (*pixels*) that are easy for computers to store and process. As an example, raw data produced by an image sensor in a camera must be processed to produce pictures that human-beings can view (Figure 2.3) [146]. Such a camera pipeline is composed of 4 stages: *denoise*, *demosaic*, *color correct*, and *tone curve*. Each of the stages either is point-wise,

or considers its nearby neighbors. As a typical memory-bound application, the whole camera pipeline has such a low communication/computation ratio that one has to compute the entire pipeline in small tiles in order to improve memory locality on instruction-based processors, at the cost of redundant computation in the overlapping tile boundaries. We shall show in Section 3.1 that this is not necessary on FPGA accelerators, where external memory accesses can be optimally minimized without loss of scalability.

2.2.2 Numerical Analysis

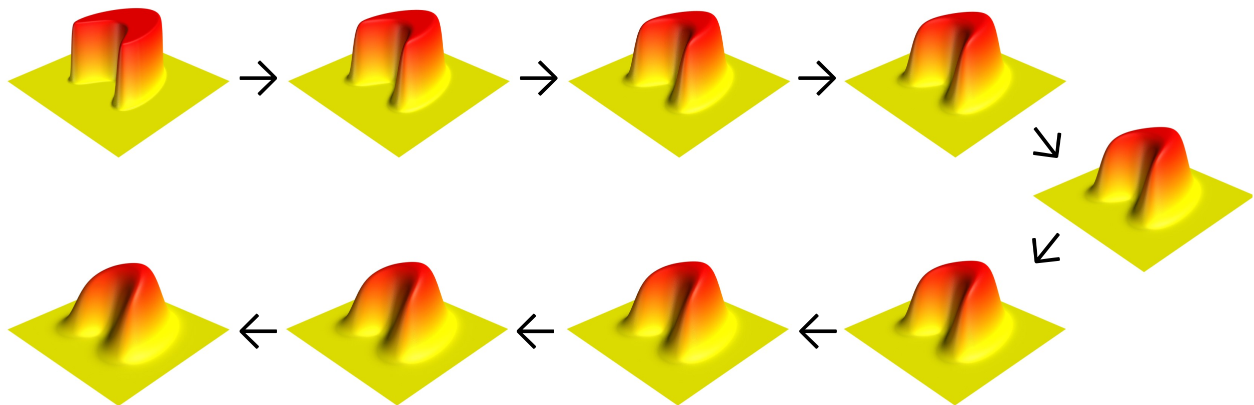
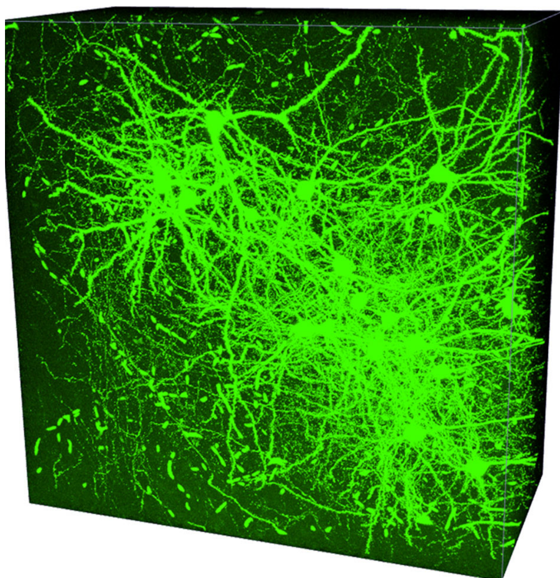


Figure 2.4: Illustration of the Heat equation [137].

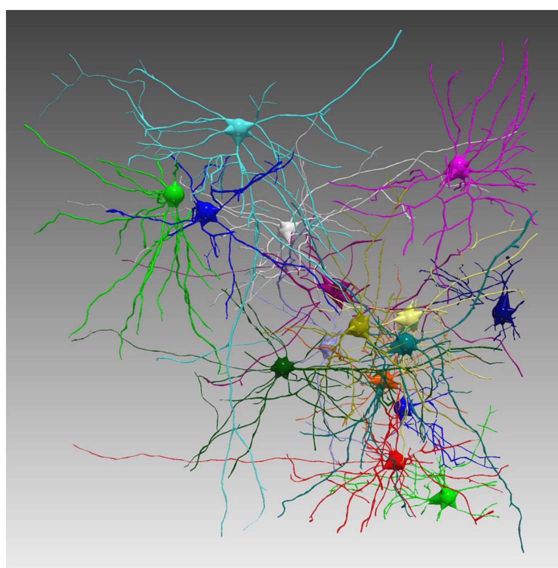
As many mathematical problems do not have known closed-form solutions, mathematicians often resort to numerical methods to solve such problems approximately. A common example is to solve partial differential equations (PDE) using finite element methods, where the numerical space is partitioned into a multidimensional array of finite elements, and the derivative at each point in space is numerically calculated as a weighted sum of neighboring elements. The solution can then be obtained by applying the derivatives iteratively until convergence, as illustrated in Figure 2.4. The shape and weights of the neighbors are determined by the equations, while the size of arrays and the number of neighbors are determined by the accuracy requirement. The communication-computation trade-off for instruction-based processors discussed in Section 2.2.1 is not only still applicable, but even further complicated by the possible computation reuse opportunity [179].

We shall show in Section 3.2 that, however, computation reuse can be solved independently and near-optimally for FPGA accelerators.

2.2.3 Neural Image Reconstruction



(a) Raw 3D image of nerons.

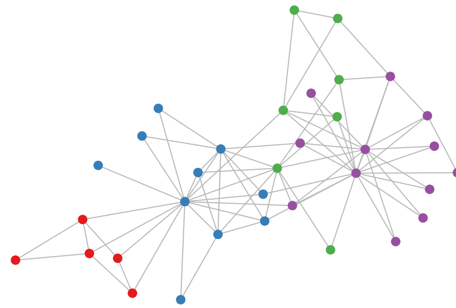


(b) Reconstructed and segmented neurons.

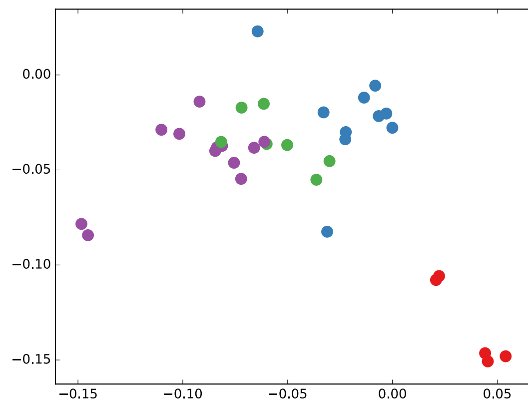
Figure 2.5: Neurons reconstructed and segmented from the raw 3D image [119].

The quest to learn and understand the biological functionality of animal brains has never ended. Towards this goal, understanding the morphology of neurons is one of the most important tasks. Today, researchers can obtain 3-dimensional images from a brain using imaging technologies, but reconstructing the morphology, e.g., recognizing and segmenting individual neurons from a cluster of interweaving neurons, remains a not only challenging, but also time-consuming task. State-of-the-art research [119] shows the great potential of a computational approach based on Dijkstra's shortest path algorithm, but due to the irregular and intensive memory access demand of graph algorithms, scaling the algorithm to large-scale high-precision neural images remains a challenging task.

2.2.4 Learning on Graph Structure



(a) Karate club network.



(b) Embeddings obtained from GCN.

Figure 2.6: Zachary's karate club network [174] and its embeddings obtained from a 3-layer GCN model [101].

Convolutional neural networks (CNN) have shown great success in many application domains, including image and video recognition, image classification [156], natural language processing, etc., and many hardware accelerators emerged rapidly towards fast and energy-efficient training and inference of CNN. However, CNN only learns on regular and dense data structures, yet information embedded in the relationship among objects is not well understood. More re-

cently, the concept of convolutional neural networks was extended to graph data structures [101]. The unique computational challenge imposed by such workload is the combination of both irregular, sparse graph operations and regular, dense matrix operations, which makes optimized frameworks designed for only one of them insufficient to adequately accelerate GCN workload.

2.3 Domain-Specific Languages

Domain-specific languages (DSL) are mini-languages that are created for specific application domains. Since they are *domain-specific*, DSLs often have simpler syntax than general-purpose languages with many convenient language constructs, and are thus much easier to learn and use. In this section, we introduce two emerging DSLs that are involved in this dissertation.

2.3.1 Halide

Halide [146] is an open-source DSL for fast and portable computation on images and tensors. It is designed to help programmers write high-performance image processing code easily on modern machines. One of the biggest advantages of Halide is that it decouples the algorithm description of the program from the scheduling — its execution strategy. When trying to optimize Halide code, programmers can simply modify the code of the scheduling part without changing the algorithm part to change how the program is executed. Listing 2.1 shows an example of Halide program, where programmers can easily change the traversal order using `reorder`. For equivalent C or C++ code, programmers have to change the whole loop of their code. Halide currently targets CPU and GPU, which are both software platforms without hardware customization capability.

2.3.2 HeteroCL

HeteroCL [110] is a Python-based DSL for software-defined reconfigurable accelerators. Similar to Halide, HeteroCL separates the algorithms and the schedules, as shown in Listing 2.2. One

```
// Algorithm
Var x("x"), y("y");
Func B("B");
B(x, y) = (A(x, y) + A(x + 1, y) + A(x + 2, y)) / 3;

// Schedule
B.reorder(y, x);
```

Listing 2.1: A piece of Halide program.

major difference between HeteroCL and Halide is that HeteroCL supports a compilation flow targeting FPGA with multiple backends, including SODA [20], PolySA [38], and Merlin Compiler [33]. Therefore, the customized schedule can be migrated to the hardware design in subsequent HLS/RTL code generated. HeteroCL categorizes the hardware customization into three types: compute, data types, and memory architectures, which allows programmers to explore performance/area/accuracy trade-offs.

```
# Algorithm
A = heterocl.placeholder((height, weight), name='A')
B = heterocl.compute(A.shape,
                     lambda x, y: (A[x, y] + A[x + 1, y] + A[x + 2, y]) / 3,
                     name='B')

# Schedule
s = heterocl.create_schedule([A, B])
s.reorder(B.axis[1], B.axis[0])
```

Listing 2.2: A piece of HeteroCL program.

The heterogeneous backend supported by HeteroCL generates HLS code, which is then synthesized to RTL code using vendor tools. These backends target different types of programs and achieves decent performance. SODA [20] targets stencil computation, which will be detailed in Chapter 3 and Chapter 4. PolySA [38] targets systolic arrays, an architecture consisting of a

group of identical processing elements (PE). This architecture is applicable to a wide range of applications, including convolution computation and matrix multiplication. Apart from the above, Merlin Compiler [33] is a more general backend that can generate optimized HLS code for both Intel and Xilinx platforms and greatly enhance the generality of HeteroCL.

2.4 Related Work

This section summarizes the related work for this dissertation. Section 2.4.1 discusses stencil accelerators with communication-reuse microarchitecture templates, while Section 2.4.2 introduces computation reuse techniques known for stencil computation. In the image processing domain where stencil computation is used extensively, domain-specific languages are widely adopted and are summarizes in Section 2.4.3. Section 2.4.4 summarizes state-of-the-art programming models, interfaces, tools, and languages for task-parallel programs. Section 2.4.5 discusses the related work for single-source shortest path.

2.4.1 Stencil Microarchitecture Templates

Non-uniform memory partitioning-based line buffers are widely used to enable communication data reuse and reduce the external memory accesses for stencil computation. Cong et al. [36] proved that it requires least buffer size for a single PE. Wang and Liang [166] propose to adopt the OpenCL model for iterative stencil algorithms. While the coarse-grained, tile-level parallelism increases the on-chip buffer usage and therefore limits the tile size, OpenCL pipes are used in [166] to alleviate the performance degradation brought by overlapping tile borders. Natale et al. [135] propose to implement multiple temporal iterations as multiple stages and connect them to form a dataflow architecture. This approach scales well as the number of iterations increases, but does not provide parallelism within a single iteration. Zohouri et al. [183] propose to use multiple processing elements (PEs) for each iteration in addition to implementing multiple temporal iterations as multiple stages. However, the reuse buffers are replicated along with the PEs

in each stage to provide concurrent accesses in [183]. We shall show later on in Section 3.1.3 that it is in fact suboptimal.

While the line buffer-based approach is widely used, there are other approaches proposed by researchers. Hegde and Kapre [83] present a soft vector processor for accelerating stencil kernels for OpenCV on FPGAs. Escobedo and Lin [64] use a graph theory-based approach to achieve minimum number of memory banks for a wide range of stencil kernels. However, this approach does not generalize to all stencil kernels. Stitt et al. [159] present a scalable window generator for high bandwidth FPGA systems, which can be used for stencil applications. However, implementing kernels with different window shapes is still non-trivial due to the manual RTL design approach of [159].

2.4.2 Common Subexpression Elimination

Computation reuse is a well-known concept in compiler optimization, more commonly known as common subexpression elimination (CSE). The classical CSE technique is based on expression analysis of the program or value numbering. For example, to evaluate two expressions `x=a·b+c` and `y=a·b+d`, a compiler is expected to find that the two expressions for `x` and `y` share the same subexpression `a·b`, which can be evaluated only once by evaluating a new expression `tmp=a·b` before `x=tmp+c` and `y=tmp+d`.

While the classical CSE is powerful and effective, we notice that it can only achieve *spatial* computation reuse, i.e., common subexpressions exposed independently of the *temporal* loop variables. For example, in Formula 2.1 (which corresponds to Listing 3.1 on page 41), there is no common subexpressions in the classical sense, but there actually is computation that can be reused across loop iterations, i.e., *temporal* reuse.

$$Y[j][i] = (X[j-1][i] + X[j][i-1] + X[j][i] + X[j][i+1] + X[j+1][i]) \times 0.2 \tag{2.1}$$

This is because when iterating over arrays, different array references from different loop iterations may be referring to the same data element of arrays. For example, in Formula 2.1,

the same computation `X[1][2]+X[2][1]` is done twice, $X[j][i+1] + X[j+1][i]$ for `Y[1][1]` and $X[j-1][i] + X[j][j-1]$ for `Y[2][2]`. Figure 2.7 visualizes the above reuse by showing the over-

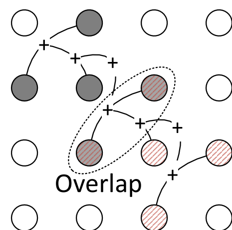


Figure 2.7: Overlapping pattern in a 5-point Jacobi kernel.

lapping inputs used for producing `Y[1][1]` and `Y[2][2]`. With computation reuse, the new kernel becomes a 2-stage kernel (Formula 2.2), which requires only 3 additions per output. As a comparison, the original kernel (Formula 2.1) needs 4 additions per output.

$$\begin{aligned} T[j][i] &= X[j-1][i] + X[j][i-1] \\ Y[j][i] &= (T[j][i] + X[j][i] + T[j+1][i+1]) \times 0.2 \end{aligned} \tag{2.2}$$

Note that there is an implication: when processing such computation reuse, the compiler must recognize the reduction operation and select operands for reuse from a proper computation order, e.g.

$$((X[j-1][i] + X[j][i-1]) + X[j][i]) + (X[j][i+1] + [j+1][i]) \tag{2.3}$$

otherwise the binary + operator will not expose subexpressions like $X[j][i+1] + X[j+1][i]$ due to its default left-to-right associativity.

In summary, a compiler must perform both *temporal exploration* among different loop iterations and *spatial exploration* among reduction operands to find the best design point for computation reuse. However, previous work on computation reuse has limited temporal and/or spatial exploration over the computation reuse design space.

On the temporal exploration side, Ernst [63] and Kronawitter et al. [106] find reuse among iterations via loop unrolling plus spatial CSE, which is suboptimal, e.g., for Formula 2.1 it may

only reuse 1 addition operation per 2 outputs, resulting in 3.5 additions per output, as opposed to 3 achieved by Formula 2.2. Chen et al. [15], Koraei et al. [103], and Zhao et al. [179] only reuse the pointwise scaling operations among iterations, resulting in redundant reduction operations.

On the spatial exploration side, Chen et al. [15], Hammes et al. [81], and Kronawitter et al. [106] do not consider commutativity and associativity. Cooper et al. [43] and Hammes et al. [81] only consider operands spanning in the horizontal direction (corresponding to the innermost loop variable). Deitz et al. [55] only considers operands spanning the horizontal or vertical directions (corresponding to the loop variable of each level of the loop nest). Ding et al. [62] additionally considers diagonal directions, i.e., all loop variables incrementing by the same value. Yet, computation reuse could appear along any spatial direction of the stencil window, which is likely missed by the prior work mentioned above (e.g., Figure 3.6 on page 53).

Besides, previous work on computation reuse heavily focuses on CPU and/or GPU [43, 55, 62, 63, 106, 179], where the trade-off between computation and storage relies heavily on register pressure [43] and/or cache [2, 179] analysis, which is generally hard to characterize quantitatively due to the close yet unmanaged interaction between the computation units and the memory system. Ding et al. [62] find from experimental results that although computation reuse adds <1% space overhead in most cases on CPU, it is also common to have 100% space overhead for other cases. For accelerators, we shall show in Section 3.1 that parallelization and communication can be fully decoupled and present a microarchitecture that requires the Pareto-optimal on-chip buffer size w.r.t. the degree of parallelism. However, it does not remove any redundant computation. We shall show further in Section 3.2 that computation reuse can be applied independent of parallelization and communication reuse, and how to obtain the Pareto-optimal on-chip buffer size with computation reuse being taken into consideration.

2.4.3 Image Processing Domain-Specific Languages

In the image processing domain, stencil kernels are ubiquitous. There are domain-specific compilers that can generate efficient FPGA accelerators from high-level image processing domain-

specific languages (DSL). Hegarty et al. [82], Reiche et al. [147], and Chugh et al. [28] create their own image DSLs and provide compilation flow from the DSLs to hardware code, generating efficient FPGA and/or ASIC designs. While these DSLs significantly reduce the burden of writing high-quality FPGA codes, analytical model-based efficient design-space exploration (DSE) and systematic performance optimization for stencil computations, especially for iterative stencil computations, are not present. We shall show in Chapter 4 that such design-space exploration can be very important to obtain good quality of result, and the SODA DSE framework [19, 20] can automatically find the performance-optimized configuration for a given stencil kernel under the platform resource constraints.

Another limitation of the previous works is that the effort to rewrite existing programs in another language is not negligible. Pu et al. [144] presents an alternative approach named Halide-HLS, which is a compiler that takes as input an existing popular DSL, Halide [146], and provides an automatic pass to synthesize Halide programs to hardware accelerators. Different from other image DSLs, HeteroHalide leverages the existing Halide [146] infrastructure and keeps algorithms decoupled from schedules, which greatly improves the portability and composability of code. The moderate modifications on the scheduling part of existing Halide programs enables fast adoption of FPGA accelerators for the vast number of Halide programs. However, the Halide-HLS compiler is not designed in a composable and hierarchical way, i.e., its scheduling primitives and the corresponding code generators are tightly and directly coupled with the underlying microarchitecture template. This makes it difficult to leverage state-of-the-art accelerator microarchitectures for the best performance and adapt to behavior changes in the vendor tools. We shall show in Chapter 5 that by leveraging HeteroCL [110] as an intermediate representation, our work, HeteroHalide [118] scales better due to its capability of taking advantage of the SODA microarchitecture.

2.4.4 Task-Level Parallelism

Task-level parallelism is a form of parallelization of computer programs across multiple processors. In contrast to data parallelism where the workload is partitioned on data and each processor executes the same program (e.g., OpenMP [46]), different processors in a task-parallel program often behave differently, while data are passed between processors. Examples of task-parallel programs include image processing pipelines [20, 144], graph processing [167, 181], and network switching [140]. Software programs usually implement tasks as threads and/or processes and rely on the operating system to schedule execution and handle communication. This often leads to poor performance caused by inefficient inter-task communication and frequent context switch [4]. Hardware programs, on the other hand, can be much more efficient due to the massive amount of inherently parallel logic units. Moreover, HLS-based FPGA accelerators written in task-parallel form often outperform their data-parallel counterparts [20, 38] due to higher clock frequency and better scalability as a result of local communication [40].

This section discusses background and related work of task-level parallelism and task-parallel accelerators. The rest of this section is organized as follows. Section 2.4.4.1 briefly discusses the programming models for task-parallel programs, while Section 2.4.4.2 introduces two existing programming interfaces. State-of-the-art HLS tools support task-parallel programs by allowing the programmers to launch multiple tasks and stream data between them. We shall discuss them and focus on the inter-task communication interface, task instantiation interface, system integration interface, and software simulation mechanism in Section 2.4.4.3. Besides the general HLS tools, there are frameworks developed specifically for pure streaming applications, which are the discussed in Section 2.4.4.4.

2.4.4.1 Programming Models

Task-parallel programs are often described as communicating sequential processes [84] or using dataflow models [97, 112, 142]. Kahn process network (KPN) [97] is one of the most popular mod-

els used. Under the KPN model, tasks are called *processes*. Processes communicate only through unidirectional *channels*. Data exchanged between channels are called *tokens*. KPN requires that ① each process is deterministic, i.e., the same input sequence must produce the same output sequence; ② channels are unbound, read blocks if and only if the channel is empty, and write always succeed immediately; ③ a process cannot test an input channel for existence of tokens without consuming them (i.e., *peeking* a channel). Consequently, KPN processes are not only *deterministic*, but also *monotonic* with respect to the input sequences: the same input token sequences always produce the same output sequences regardless of the timing of their arrival, and partial inputs always produce partial outputs. Synchronous dataflow (SDF) [112], as a special case of KPN that produces/consumes fixed number of tokens per firing, further allows such a task-parallel program to be scheduled statically.

While existing task-parallel programming models have been successful in scheduling tasks on parallel processors, we shall show in Section 6 that, when applied to model task-parallel HLS programs, such models lack good programmability support: not allowing peeking restricts the expressiveness of programs; not modeling the channel capacity leads to mismatching behavior between the programs and their implementations. In this dissertation, we borrow the terms *process*, *channel*, and *token* used in the KPN formulation, but are not limited to KPN or any dataflow model. In fact, we shall describe our programming model as a hierarchical finite state machine in Section 6.2.2 to overcome expressiveness limitations of existing programming models.

2.4.4.2 Programming Interfaces

SystemC is a set of C++ classes and macros that provide detailed hardware modeling and event-driven simulation. It supports both cycle-accurate and untimed simulation and many simulator implementations are available [29, 152]. Some HLS tools support a subset of untimed SystemC as the input [171]. SystemC supports task-parallel programs natively via the `sc_module` constructs and `tlm_fifo` interfaces. Listing 2.3 shows an example using the accelerator that will be discussed in Section 6.1. Compared with other C-like HLS languages, SystemC can model more hardware

details but is more verbose and less productive due to its special language constructs: for the code snippets shown in Listing 6.4 and Listing 6.5, equivalent SystemC code is 37% longer, as illustrated in Listing 2.3.

```
1  SC_MODULE(VertexHandler) {
2    sc_core::sc_port<tlm::tlm_fifo_gett_if<VertexReq>> vertex_req;
3    ...
4    SC_CTOR(VertexHandler) { SC_THREAD(thread); }
5    void thread() {
6      VertexReq req;
7      if (vertex_req.nb_put(req)) {
8        ...
9      }
10   }
11 };
12
13 SC_MODULE(Ctrl) {
14   sc_core::sc_port<tlm::tlm_fifo_put_if<VertexReq>> vertex_req;
15                                            // declare communication interface
16   ...
17   SC_CTOR(Ctrl) { SC_THREAD(thread); }
18   void thread() { ... }                                  // task description
19     while (...) {
20       VertexReq req();
21       vertex_req.put(req);
22       ...
23     }
24   }
25 };
26
27 SC_MODULE(PageRank) {
28   tlm::tlm_fifo<VertexReq> vertex_req{/*depth=*/2};       // instantiate channels
29   ...
30   Ctrl ctrl;                                             // instantiate tasks
31   VertexHandler vertex_handler;
32   ...
33   SC_CTOR(PageRank) {
34     ctrl.vertex_req(vertex_req);      // bind channels to communication interfaces
35     vertex_handler.vertex_req(vertex_req);
36     ...
37   }
```

```
38 };
```

Listing 2.3: SystemC TLM API example.

Pthread API is a set of widely used standard APIs that can be used to implement task-parallel programs using threads. Pthread requires programmers to explicitly create and join threads, and arguments need to be manually packed and passed. Listing 2.4 shows an example using the accelerator that will be discussed in Section 6.1. Compared with the `tapa::invoke` API used by TAPA, our extension to HLS language that will be presented in Chapter 6, the pthread APIs require more effort to program: for the code snippets shown in Listing 6.4 and Listing 6.5, equivalent pthread-based code is 78% longer, as demonstrated in Listing 2.4.

```
1  struct Ctrl_Arg {                                  // task communication interface
2    channel<VertexReq>* vertex_req;
3    ...
4  };
5
6  void Ctrl(void* arg) {                                        // task description
7    Ctrl_Arg* ctrl_arg = (Ctrl_Arg*)arg;                        // unpack arguments
8    channel<VertexReq>* vertex_req = ctrl_arg->vertex_req;
9    ...
10   while (...) {
11     VertexReq req(...);
12     vertex_req->write(req);
13     ...
14   }
15   pthread_exit(NULL);
16 }
17
18 struct VertexReq_Arg {
19   channel<VertexReq>* vertex_req;
20   ...
21 };
22
23 void VertexReq(void* arg) {
24   VertexReq_Arg* vertex_req_arg = (VertexReq_Arg*)arg;
25   channel<VertexReq>* vertex_req = ctrl_arg->vertex_req;
26   VertexReq req;
27   if (vertex_req->nb_read(req)) {
```

```
28      ...
29    }
30    pthread_exit(NULL);
31  }
32
33  void PageRank(...)
34    channel<VertexReq> vertex_req;                          // instantiate channels
35    ...
36    Ctrl_Arg Ctrl_arg;
37    Ctrl_arg.vertex_req = &vertex_req;                           // pack arguments
38    ...
39    VertexReq_Arg VertexReq_arg;
40    VertexReq_arg.vertex_req = &vertex_req;
41    ...
42    pthread_t Ctrl_pid, VertexReq_pid...;
43    pthread_create(&Ctrl_pid, NULL, Ctrl, (void*)&Ctrl_arg);       // launch threads
44    pthread_create(&VertexReq_pid, NULL, VertexReq, (void*)&VertexReq_arg);
45    ...
46    pthread_join(&Ctrl_pid, NULL);                                  // join threads
47    ...
48  }
```

Listing 2.4: Pthread API example.

In summary, while the existing API alternatives are widely used in some domains, we shall show in Chapter 6 that they are more verbose and thus less productive compared with our work, TAPA.

2.4.4.3 HLS Support for Task-Parallel Programs

Intel HLS compiler supports two different inter-task communication interfaces, `ihc::pipe` and `ihc::stream`. `ihc::pipe` implements a light-weight hardware FIFO with `data`, `valid`, and `ready` signals, while `ihc::stream` implements an Avalon-ST interface that supports transactions. Tasks are instantiated using `ihc::launch` and `ihc::collect`. Software simulation is done via launching multiple threads. Instances of the same task are synthesized separately.

Intel OpenCL compiler supports light-weight FIFO via two APIs, i.e., standard OpenCL `pipe`

and Intel-specific channel. Tasks are instantiated by defining OpenCL __kernels, which forces instances of the same task to be synthesized separately as different OpenCL kernels. OpenCL runtime handles the software simulation by launching multiple threads.

Vivado HLS provides two different streaming interfaces: ap_fifo and axis. The ap_fifo interface generates light-weight FIFO interface. Tasks are instantiated by invoking the corresponding functions in a dataflow region, and instances of the same task are synthesized separately. Software simulation is done by sequentially executing the tasks. The axis interface generates AXI-Stream interface with transaction support. It requires the programmers to instantiate channels and tasks in a separate configuration file when running logic synthesis and implementation. This allows different instances of the same task to be synthesized only once, but takes longer time to learn and implement compared with ap_fifo. OpenCL runtime handles the software simulation for tasks instantiated with the axis interface by launching multiple threads.

Xilinx OpenCL compiler supports standard OpenCL pipe, which generates AXI-Stream interfaces similar to Vivado HLS axis, but pipe does not provide APIs to support transactions. Like Vivado HLS axis, software simulation of pipe is handled by the OpenCL runtime by launching multiple threads.

LegUp compiler provides legup::FIFO, which implements light-weight FIFOs. Tasks are instantiated using pthread API (Section 2.4.4.2). Software simulation is accomplished by launching multiple threads. Instances of the same task are synthesized separately.

Merlin compiler [33] allows programmers to call the FPGA kernel as a C/C++ function and provides OpenMP-like simple pragmas with automated design-space exploration based on machine learning. To support task-parallel programs, Merlin leverages its backend vendor tools' programming interfaces. Software simulation is done by sequentially executing the tasks.

In summary, as pointed out in Table 2.1, none of the state-of-the-art HLS tools provide peeking support. Only Intel HLS ihc::stream and Vivado HLS axis support transactions. Only Merlin allows the accelerator kernel to be called as if it is a C/C++ function. Vivado HLS and Merlin

Table 2.1: Summary of related work of TAPA.

Related Work	Programmability			Software Simulation	RTL Code Generation
	Peeking	Transaction	Host Iface.		
Fleet [161]	No	No	N/A	Sequential	N/A
Intel HLS (`ihc::pipe`)	No	No	N/A	Multi-thread	Monolithic
Intel HLS (`ihc::stream`)	No	Yes	N/A	Multi-thread	Monolithic
Intel OpenCL	No	No	OpenCL	Multi-thread	Monolithic
LegUp [11]	No	No	N/A	Multi-thread	Monolithic
Merlin [33]	No	No	C++	Sequential	Monolithic
ST-Accel [150]	No	No	VFS	Sequential	Hierarchical
Vivado HLS (`ap_fifo`)	No	No	OpenCL	Sequential	Monolithic
Vivado HLS (`axis`)	No	Yes	OpenCL	Multi-thread	Manual
Xilinx OpenCL	No	No	OpenCL	Multi-thread	Monolithic
TAPA (Chapter 6)	Yes	Yes	C++	Coroutine	Hierarchical

execute tasks sequentially for simulation while others launch multiple threads. All HLS tools treat a task-parallel program as a monolithic design and generate RTL code for each instance of task separately, except that Vivado HLS `axis` allows programmers to manually instantiate tasks using a configuration file when running logic synthesis and implementation.

2.4.4.4 Streaming Frameworks

Streaming applications are a special type of task-parallel applications that do not require complex control over inter-task communication and often expose massive data parallelism in addition to task parallelism. There are previous works that focus specifically on such applications.

ST-Accel [150] is a high-level programming platform for streaming applications that features

highly efficient host-kernel communication interface exposed as a virtual file system (VFS). It uses Vivado HLS as its backend for hardware generation and its software simulation is done by sequential execution.

Fleet [161] is a massively parallel streaming framework for FPGAs that features highly efficient memory interfaces for massive instances of parallel processing elements. Programmers write Fleet programs in a domain-specific RTL language based on Chisel [3]. The programs can be simulated in Scala (in which Chisel is embedded).

In summary, while these frameworks are specialized for streaming patterns, neither of them provide peeking and transaction interface in the kernel. Both run software simulation sequentially, which does not have correctness problem for streaming applications but will be restrictive for general task-parallel programs.

2.4.5 Single-Source Shortest Path

Given a directed graph[1] $G = (V, E)$ where each edge $e_{i,j} \in E$ has a non-negative[2] weight $w_{i,j} \geq 0$, a *path* P is a sequence of vertices $P = (v_1, v_2, \cdots, v_n) \in V \times V \times \cdots \times V$ such that v_i and v_{i+1} are connected by an edge $e_{i,i+1} \in E$ for $1 \leq i < n$. Such a path is called a path from $u = v_1$ to $v = v_n$. The *shortest path* between u and v is the path that minimizes the *distance* from u to v, i.e., $\sum_{i=1}^{n-1} w_{i,i+1}$. The *single-source shortest path* (SSSP) problem aims to find the shortest path from a given vertex u (called the *root*) to all vertices in the graph. That is to say, for each vertex $v \in V$, we not only need to find the shortest distance from u, but also the sequence of vertices that connects u to v. This can be effectively represented by storing the parent (v_{i-1}) of each v_i in the output, allowing us to reconstruct the shortest path tree from u.

[1] An undirected graph is modeled as a directed graph with bidirectional edges.

[2] Dijkstra's algorithm and its variants, including SPLAG presented in this dissertation, cannot be used on negative-weighted graphs. In practice, the edge weights represent distances, and are often non-negative by definition, e.g., network latency, strength of connection, etc.

2.4.5.1 Single-Source Shortest Path Algorithms

Dijkstra's algorithm [61] keeps two sets of vertices, the *visited* set and the *active* set. Initially, the visited set is empty, and the active set contains only the root vertex. All vertices are initialized with a *tentative* distance of ∞, except that the root has distance 0. The algorithm iteratively removes vertex u with the minimum distance from the active set, traverses the neighbors of u, and moves u to the visited set. For each neighbor v of u, a new tentative distance can be generated by adding the edge weight to the tentative distance of their parent vertex u. If this new tentative distance is smaller than the previously known distance, v will get a new tentative distance. If v is not in the active set nor the visited set, it will be moved to the active set. This compare-and-update operation is called *relaxation*. The algorithm terminates when the active set is empty.

The original Dijkstra's algorithm uses a list to store the active vertices, which necessitates $\Theta(|V|)$ time to find the minimum-distance vertex. This can be improved by leveraging a priority queue to store the active vertices, which decreases the time complexity of this step to $\Theta(\log|V|)$ [69, 94]. If all weights are small integers bound by a constant C, using a bucket queue can further decrease the time complexity to $\Theta(C)$ for finding the minimum-distance vertex (Dial's algorithm [60]). Dijkstra's algorithm and its priority queue–based variants guarantee each edge is visited *at most once*, however, at the cost of being *inherently sequential* and hard to parallelize, since edges from only one vertex can be relaxed at a time.

The *Bellman-Ford algorithm* [155] employs a different and parallelizable approach to solve the SSSP problem. Instead of selecting the edges from the minimum active vertex for relaxation, this algorithm traverses and relaxes all edges iteratively. Allowing parallel relaxation on all vertices enables massive parallelism, although doing so will relax each edge many times and thus is work-inefficient. Unlike Dijkstra's algorithm and its variants, the Bellman-Ford algorithm can handle negative weights and detect negative cycles. However, its worse-case time complexity of $\Theta(|V||E|)$ makes it highly inefficient when all the edge weights are non-negative, which is quite common in real-world applications. We will show in Section 8.1.6.4 that even with pruning and early-

termination, the Bellman-Ford algorithm is still not as efficient as other SSSP algorithms.

The trade-off between parallelism and work-efficiency has motivated many researchers in the past few decades. The eager version of Dijkstra's algorithm [56] exposes more parallelism by relaxing edges in parallel from more than one vertex with minimal distances. Crauser et al. [44] further define heuristics to decide edges from how many vertices should be relaxed in parallel. *Δ-stepping* [127] and its variants [51, 164] generalize Dial's algorithm [60] by dividing the active vertices into buckets based on their distances and only select active vertices from the first non-empty bucket with the smallest distances.

2.4.5.2 Other Shortest Path Problems

The single-source shortest path problem is not the only possible type of shortest path problems. In fact, we can define four shortest path problems on a given graph G:

- The *single-pair shortest path* problem finds the shortest path from a given source vertex u to a given destination vertex v.
- The *single-source shortest path* problem finds the shortest path from a given vertex u to all vertices in the graph.
- The *single-destination shortest path* problem finds the shortest path from all vertices to a given vertex v in the graph.
- The *all-pairs shortest path* problem finds the shortest path between all pairs of vertices.

The single-pair shortest path problem can be solved using Dijkstra's algorithm with an early termination condition. For multiple single-pair shortest path queries on the same graph, one can solve them more efficiently by pre-computing the SSSP of some selected landmarks [74]. The single-destination shortest path problem can be reduced to SSSP by reverting the direction of edges. The all-pairs shortest path problem utilizes a different algorithm than SSSP [8]. However, due to its $\Theta(|V|^3)$ time complexity, a complete solution to the all-pairs shortest path problem is

often computationally intractable for large-scale graphs, in which cases SSSP of selected/sampled vertices can be used [10]. Altogether, solving the SSSP problem efficiently can be helpful for all four types of shortest path problems.

2.4.5.3 Dijkstra's Algorithm Accelerators

Takei et al. [160] accelerates the original Dijkstra's algorithm and parallelizes both relaxation and the linear search for the minimum active vertex with a SIMD architecture. Lei et al. [114] implements Dijkstra's algorithm with an on-chip systolic priority queue [115]. Since the systolic priority queue operates on every data element on each clock cycle, it cannot leverage dense on-chip storage elements (e.g., BRAMs) and its capacity does not scale well. A two-level linear-search structure is used when the number of active vertices grows beyond the capacity of the queue. Chronos [1] exploits massive speculative parallelism and can implement the eager version of Dijkstra's algorithm efficiently. Chronos uses an on-chip pipelined heap [5] to store the active vertices, which scales better than the systolic priority queue but still is limited by the size of on-chip storage. Only planar graphs are evaluated for the above accelerators.

2.4.5.4 Bellman-Ford Algorithm Accelerators

HitGraph [181] and its earlier version [180] implement an edge-centric graph accelerator. Leveraging the larger sequential bandwidth, HitGraph writes the intermediate relaxation results to DRAM when generated and reads them back when needed. ThunderGP [13] is an HLS-based graph processing template that implements highly-parallel graph accelerators under the vertex-centric gather-apply-scatter model. Unlike HitGraph, ThunderGP updates on-chip vertices directly without generating off-chip intermediate results. GraphLily [87] is an HLS-based graph linear algebra overlay implemented on an FPGA equipped with high-bandwidth memory (HBM). GraphLily can implement the Bellman-Ford algorithm along with other graph linear algebra algorithms without reprogramming the FPGA. In summary, the Bellman-Ford algorithm accelerators

are evaluated using large-scale power-law graphs, but their work-efficiency is overlooked. Only the raw edge traversal throughput is reported, which demonstrates the performance of the graph processing system, but not the algorithm itself. Even worse, to reduce the bandwidth requirement, none of the accelerators records the parent vertex together with the shortest distance. Without the parent vertex as part of the output, we will not be able to construct the shortest path tree out of the result, which reduces the usefulness of the result.

Table 2.2: Summary of related work for SSSP. MTEPS measures the algorithm throughput, which is defined as the number of undirected edges in the connected component divided by the execution time. Since some systems did not report the execution time, an upper-bound estimation (Section 8.1.6.4) is listed here.

System	Lang.	Work-eff.?	Power-law?	Priority queue?	Vertex cache?	MTEPS
Chronos [1]	RTL	Yes	No	P-heap [5]	App.-agnostic	360
GraphLily [87]	HLS	No	Yes	No	Scratchpad	<232
HitGraph [181]	RTL	No	Yes	No	Scratchpad	46.9
Lei et al. [114]	RTL	Yes	No	ExSAPQ [114]	No	9.2
Takei et al. [160]	RTL	Yes	No	No	On-chip only	0.4
ThunderGP [13]	HLS	No	Yes	No	Scratchpad	<122
SPLAG	HLS	Yes	Yes	CGPQ (Sec. 8.1.3)	CVC (Sec. 8.1.4)	763

In this dissertation, we shall present our work named SPLAG in Section 8.1. SPLAG uses a coarse-grained priority queue (CGPQ) to manage the active vertices in the SSSP problem. The CGPQ organizes active vertices in chunks, stores them in the external memory, and orchestrates the chunks with an on-chip priority queue. SPLAG also employs a customized vertex cache (CVC) with application-specific push and pop operations, which reduces both on-chip and off-chip memory traffic. Table 2.2 compares our work with the related works for SSSP.

CHAPTER 3

Theoretical Analysis of Stencil Applications

As a representative type of memory-bound application, stencil applications operate on regular data arrays, which allows us to perform theoretical analysis at compilation/synthesis time to optimize data paths and schedule memory accesses pertinently. In this chapter, we present the SODA (Stencil with Optimized Dataflow Architecture) microarchitecture and analyze it from a theoretical point of view. Section 3.1 presents the design objectives and the algorithm used to generate the SODA microarchitecture, followed by a proof that it requires the least possible buffer size to achieve full communication reuse. Section 3.2 presents an optimal algorithm and a heuristic algorithm that can find computation reuse in stencil kernels, further reducing memory traffic compelled by resource limitations.

3.1 Optimal Communication Reuse

3.1.1 Definitions and Problem Formulation

Stencil kernel [36]: An n-point, m-dimensional stencil kernel A defines a spatial window

$$\{\vec{a}^{(s)} | s \in \{0, 1, 2, \dots, n-1\}\}$$

and a function which produces output at spatial coordinate $\vec{y}$ where

$$\vec{y} = (y_0, y_1, y_2, \dots, y_{m-1})$$

by consuming inputs at spatial coordinates

$$\{\vec{x}^{(s)} | s \in \{0, 1, 2, \dots, n-1\}\} = \{\vec{y} + \vec{a}^{(s)} | s \in \{0, 1, 2, \dots, n-1\}\}$$

$\vec{a}^{(s)}$ denotes the offset between the s-th input and the output. For convenience' sake, we use the spatial coordinate to represent the data element at that position in this dissertation.

$\{\vec{a}^{(s)} | s \in \{0, 1, 2, \ldots, n-1\}\}$ is defined as the *stencil window*. The *stencil window size* in dimension d, S_d, is defined as

$$S_d = \max_s \left(a_d^{(s)} \right) - \min_s \left(a_d^{(s)} \right) + 1$$

In our 5-point 2-dimensional example in Listing 3.1,

$$\{\vec{a}^{(s)}\} = \{(0,-1), (-1,0), (0,0), (1,0), (0,1)\}, S_0 = S_1 = 3$$

Data linearization: Modern computer memory systems use a linear address space. An m-dimensional data must be linearized before it is stored in a memory system. Without loss of generality, a vector coordinate $\vec{x}$ can be linearized to be a scalar offset x:

$$x = x_0 + x_1 T_0 + x_2 T_0 T_1 + \cdots + x_{m-1} \prod_{d=0}^{m-2} T_d \tag{3.1}$$

where $\vec{T} = (T_0, T_1, T_2, \ldots, T_{m-1})$ is the m-dimensional size of the input data. Similarly, each coordinate vector $\vec{a}^{(s)}$ of A can also be linearized as

$$a^{(s)} = a_0^{(s)} + a_1^{(s)} T_0 + a_2^{(s)} T_0 T_1 + \cdots + a_{m-1}^{(s)} \prod_{d=0}^{m-2} T_d$$

Under the above linearization convention, we shall use scalars x and $a^{(s)}$ instead of vectors $\vec{x}$ and $\vec{a}^{(s)}$ in the following parts of this dissertation. **Reuse distance** D_r can then be defined as

$$D_r = \max_s \left(a^{(s)} \right) - \min_s \left(a^{(s)} \right) + 1$$

which represents the linearized distance between the first and the last access of each input data element. In our example, $\{a^{(s)}\} = \{-M, -1, 0, 1, M\}$, $D_r = 2M + 1$.

Stencil computation: Given an n-point, m-dimensional stencil kernel A and input set $\{x\}$, find all outputs $\{y\}$ by applying A on all inputs $\{x\}$. Note that due to the border effect, the number of valid output data elements in dimension d is always $S_d - 1$ smaller than the input, where S_d is the stencil window size in dimension d. This disappeared region is often referred to as the *halo*.

Complex stencil kernel: Two or more stencil kernels can be connected to compose a complex stencil kernel, where the output of the former is used as the input of the latter. Each stencil kernel component of the complex stencil kernel is defined as a *stage* of the whole kernel. Stages are sometimes regarded as the temporal dimension, in analogy to the spatial dimensions of data elements. In particular, stencil kernels can be computed repeatedly where the output of an iteration is used as the input of the next iteration. Such kernels are defined as *iterative*. For the sake of simplicity, the term stage is also used to refer to an iteration of an iterative stencil kernel in this dissertation. Note that the halo size in each dimension is the sum of halo sizes among all the stages in that dimension.

Problem formulation: Given a stencil computation task and the resource constraints on a hardware platform, design an accelerator that achieves the maximum sustained throughput.

3.1.2 SODA Microarchitecture

At the microarchitecture level, we aim to optimize the memory resource consumption of one stage as a building block, while allowing multiple stages or iterations to connect with each other so that spatial parallelism can be exploited. The input size and throughput constraint is assumed to be given. Choices of tile size and number of PEs will be discussed in Section 4.1.2. For the proposed microarchitecture, we have four design objectives.

- **Full pipelining.** Pipelining can increase the throughput with very little resource overhead. Every PE should be fully pipelined and able to consume the input data in one cycle and be ready for the input for the next cycle.
- **Scalable, fine-grained parallelism.** Compared with coarse-grained parallelism, fine-grained parallelism enables resource sharing and reusing, which make it more efficient and scalable.
- **Minimum external memory access.** Compared with on-chip memory, external memory access usually has less bandwidth and longer latency. The proposed design fully reuses the

input data so that every input data element only needs to be transferred once for a given tile. The streamlined access also enables dataflow optimization between stages.

- **Minimum reuse buffer size.** We can prove that the proposed microarchitecture achieves the minimum reuse buffer size with given input size and throughput requirement. Compared with other suboptimal architectures, this enables SODA to use more resources to achieve better performance under a given resource constraint.

```
1 void Jacobi(float input[N][M], float output[N][M]) {
2 for (int j = 1; j < N - 1; ++j)
3   for (int i = 1; i < M - 1; ++i)
4     output[j][i] = (input[j-1][i] + input[j][i-1] + input[j][i] + input[j][i+1] +
5                     input[j+1][i]) * 0.2f;
6 }
```

Listing 3.1: A 5-point 2-dimensional Jacobi stencil kernel.

The reuse buffer design plays a crucial role in achieving these objectives. Listing 3.1 shows a 5-point, 2-dimensional Jacobi kernel on an $M \times N$ input as an example. In the SODA microarchitecture, there are k consecutive output elements generated in each clock cycle, where k is the number of PEs. Suppose the k outputs are $\{y, y+1, y+2, ..., y+k-1\}$. To compute the k outputs, all the needed input data elements are

$$\bigcup_{l=y}^{y+k-1} \left\{ l + a^{(s)} | s \in \{0, 1, 2, \dots, n-1\} \right\}$$

Let g_k be the number of input data elements needed when producing k outputs. All the needed input data elements can then be represented as

$$\{y + a_{u,k} | u \in 0, 1, 2, \dots, g_k - 1\}$$

where $a_{u,k}$ denotes the offset of the u-th input data element needed in each clock cycle when producing k outputs.

To generate efficient line buffers, elements in $\{a_{u,k}\}$ are divided into k sets according to their remainder modulo k. Each set will be synthesized as a *reuse chain*. The *reuse buffer* is the collection of all reuse chains. In each cycle, there will be k new inputs fed into the reuse buffer, and each reuse chain will take one input.

Elements in each remainder set cut each reuse chain into several integer intervals. Each interval corresponds to an FF or a FIFO. Since numbers in each set have the same remainder modulo k, the minimum interval length will be k. If the interval length is k, there will be no data elements in-between and the FIFO is actually a register. If the interval length is larger than k, there will be buffered data in-between, and a FIFO is used. FFs/FIFOs in each set are then connected sequentially to form a complete reuse chain.

The reuse buffer size—i.e., the total number of data elements stored in the reuse buffer—can be calculated as

$$\max_u (a_{u,k}) - \min_u (a_{u,k}) + 1 =$$
$$\max_s \left(a^{(s)} + k - 1\right) - \min_s \left(a^{(s)}\right) + 1 = D_r + k - 1$$

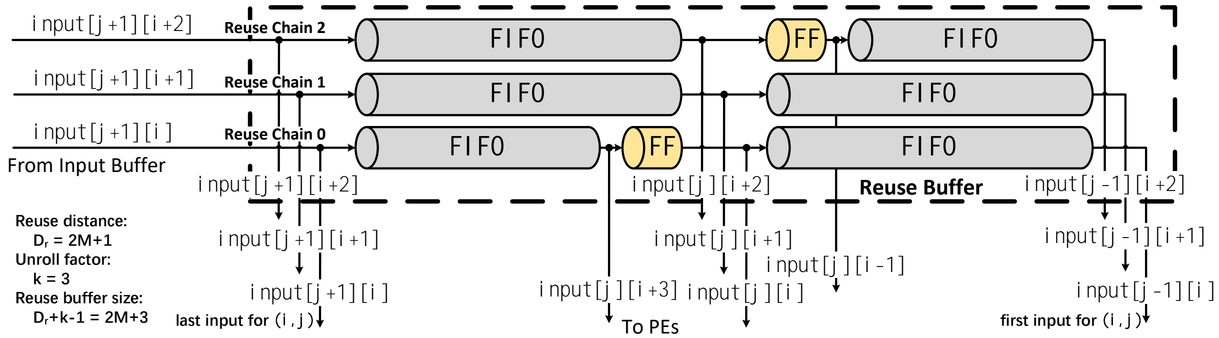


Figure 3.1: SODA reuse buffer microarchitecture.

Figure 3.1 shows the proposed microarchitecture with the example of 5-point 2-dimensional

stencil in Listing 3.1. In our $k = 3$ example,

$$\begin{aligned}\{a_{u,3}\} &= \{-M, -1, 0, 1, M\} \\ &\bigcup\{-M+1, 0, 1, 2, M+1\} \bigcup\{-M+2, 1, 2, 3, M+2\} \\ &= \{-M, -M+1, -M+2, -1, 0, 1, 2, 3, M, M+1, M+2\}\end{aligned}$$

where $g_3 = 11$. In the example of Listing 3.1 where $M = 9$, there will be $k = 3$ remainder sets / reuse chains:

$$\{-M, 0, 3, M\}, \{-M+1, 1, M+1\}, \{-M+2, -1, 2, M+2\}$$

Chain 0 $\{-M, 0, 3, M\}$ uses two FIFOs for $[-M, 0]$ and $[3, M]$ and an FF for $[0, 3]$. The length of the two FIFOs are $[0-(-M)]/3 = M/3$ and $(M-3)/3 = M/3 - 1$, respectively.

Chain 1 $\{-M+1, 1, M+1\}$ uses two FIFOs for $[-M+1, 1]$ and $[1, M+1]$. The length of the two FIFOs are $[1-(-M+1)]/3 = (M+1-1)/3 = M/3$.

Chain 2 $\{-M+2, -1, 2, M+2\}$ uses two FIFOs for $[-M+2, -1]$ and $[2, M+2]$ and an FF for $[-1, 2]$. The length of the two FIFOs are $[-1-(-M+2)]/3 = M/3 - 1$ and $(M+2-2)/3 = M/3$, respectively.

The reuse buffer size in this case is $M + 2 - (-M) + 1 = 2M + 3$.

3.1.3 Communication Reuse Optimality

The optimality of the proposed microarchitecture is proven based on the assumptions that the stencil kernel itself, A, and the size of the input, $\vec{T}$, are given. The input can be tiled, and the design choice of tile size will be discussed in Section 4.1.2. In this section we shall discuss the optimality within an input tile. For the proof of optimal reuse buffer size, we further assume that the number of PEs, k, is given.

Optimal memory utilization: The minimum requirement on input data is to feed all input data elements at least once; our microarchitecture achieves this by storing the input data on-chip until the last time it is accessed. Therefore, the proposed microarchitecture achieves the optimal

memory utilization.

Optimal reuse buffer size: Cong et al. [36] gives a mathematical proof under the polyhedral model that when there is only one PE, the line buffer design has the minimum reuse buffer size equal to the maximum reuse distance, D_r. Based on that, we have the following definition and theorems:

Definition 1 (Lexicographic Order [65])**.** *The lexicographic order relation $\prec$ of two m-dimensional coordinate vectors $\vec{i}$ and $\vec{j}$ is defined as*

$$\vec{i} \prec \vec{j} \Longleftrightarrow (i_{m-1} < j_{m-1}) \lor (i_{m-1} = j_{m-1} \land i_{m-2} < j_{m-2}) \lor (i_{m-1} = j_{m-1} \land i_{m-2} = j_{m-2} \land i_{m-3} < j_{m-3}) \lor \ldots$$
$$\lor (i_{m-1} = j_{m-1} \land i_{m-2} = j_{m-2} \land \cdots \land i_1 = j_1 \land i_0 < j_0)$$

Let A_i represent an n-point stencil window accessing inputs on $\{\vec{i} + \vec{a}^{(0)}, \vec{i} + \vec{a}^{(1)}, \vec{i} + \vec{a}^{(2)}, \ldots, \vec{i} + \vec{a}^{(n-1)}\}$ and producing output on $\vec{i}$. The lexicographic order relation $\prec$ of two stencil windows A_i and A_j is defined as

$$A_i \prec A_j \Longleftrightarrow \vec{i} \prec \vec{j}$$

Under the linearization convention in Section 3.1.1, the lexicographic order of $\vec{i}$ and $\vec{j}$ is the same as the scalar ascending order of linearized i and j. For convenience' sake, A_i is also used to denote the input elements of the stencil window A_i and the offset vector $\vec{i}$ is also denoted as i in the following parts.

Lemma 1. *The minimum reuse buffer size can only be achieved with PEs producing outputs in* ***consecutive*** *lexicographic order, if the offset vector $\vec{i}$ follows the lexicographic order.*

Proof. Suppose an implementation achieving the minimum buffer size is not implemented with PEs producing outputs in consecutive lexicographic order, which means with the k PEs that produce output elements $\{i + p_1, i + p_2, ..., i + p_k\}$ and access input elements $A_{i+p_1} \prec A_{i+p_2} \prec \cdots \prec A_{i+p_k}$ at offset i, $\exists\ p' \notin \{p_1, p_2, ..., p_k\}$ $s.t.$ $A_{i+p_1} \prec A_{i+p_2} \prec \cdots \prec A_{i+p'} \prec \cdots \prec A_{i+p_k}$. According to Property 1 in [36], data elements are accessed in lexicographic order as long as the offset vector i follows the lexicographic order. Therefore, if we allocate the k PEs for $p_1, p_2, \ldots, p', \ldots, p_{k-1}$, the buffer size can

be reduced by at least one since PE p_k accesses at least one data element lexicographically greater than any of the other PEs. Once PE p_k is replaced by p', the data element accessed only by p_k can be removed from the reuse buffer, which is a contradiction to the assumption of the minimum buffer size for the given implementation. Therefore, we know that Lemma 1 is true. □

Lemma 2. *The minimum reuse buffer size with $k+1$ PEs is at least the minimum reuse buffer size with k PEs plus 1.*

Proof. Given an optimal buffer size design with k PEs, if another PE is to be added but no additional input data element is necessary, the additional PE must be lexicographically between the existing PEs. According to Lemma 1, we know that the given design must not be an optimal buffer size design since its PE inputs are not in consecutive lexicographic order. Therefore, by contradiction, there must be at least one additional data element added to the buffer. □

Based on Lemma 1, Lemma 2, and [36], we know by induction that

Theorem 1. *For a given stencil kernel that has reuse distance D_r and a given throughput lower bound of producing k output elements at a time, the minimum possible buffer size required to achieve full data reuse is $D_r + k - 1$.*

3.1.4 Dataflow Architecture

The SODA microarchitecture can be efficiently implemented as dataflow modules. The dataflow implementation enables high-frequency synthesis result and accurate resource modeling, due to its localized communication [40] and modularized structure. It also enables the flexibility to connect multiple stages together in a single accelerator. Figure 3.2 shows the dataflow modules of 1 iteration of the Jacobi kernel shown in Listing 3.1.

As shown in Figure 3.2, the forwarding modules (`FW`) forward and distribute input data to proper destination modules. Each forwarding module either directly forwards data from the input, or implements a FIFO or FF as part of the reuse buffer. Each FIFO or FF in Figure 3.1

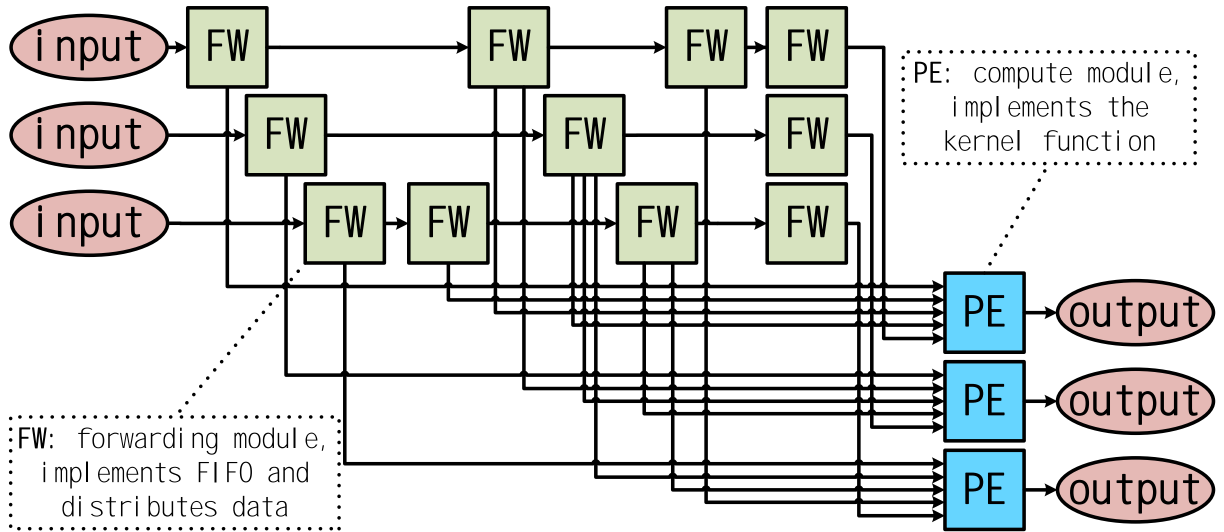


Figure 3.2: Dataflow modules in a SODA microarchitecture.

corresponds to a forwarding module shown in Figure 3.2. The structure of a forwarding module is only determined by the data type, FIFO depth, and fan-out. On FPGAs, FIFOs can be implemented with either shift register lookup tables (SRLs) or block RAMs (BRAMs). On our Xilinx platform, we use `hls::stream` provided in Vivado HLS to implement FIFO. Large FIFO whose capacity is larger than 1024 bits is implemented with BRAM and small FIFO is implemented with SRL. The compute modules (`PE`) are the processing elements and implement the kernel function. Each compute module contains 1 PE, which produces 1 output data element per cycle. For a given stencil kernel, all compute modules in the same stage have the same structure. The dataflow architecture enables the flexibility of cascading multiple stages together. The inputs and outputs can be connected to DRAM or to another stage's outputs or inputs. Figure 3.3 shows the overview of an example of a complete SODA accelerator.

As a common type of external memory on FPGAs, DRAMs have a burst I/O mode which provides higher bandwidth [27], but it also puts some restrictions on the data. On our Xilinx platform, burst-mode DRAM access is fully pipelined, and in each cycle $W_b = 512$ bits are read/written for the maximum throughput. Therefore, 8-bit, 16-bit, or 32-bit input data must be coalesced be-

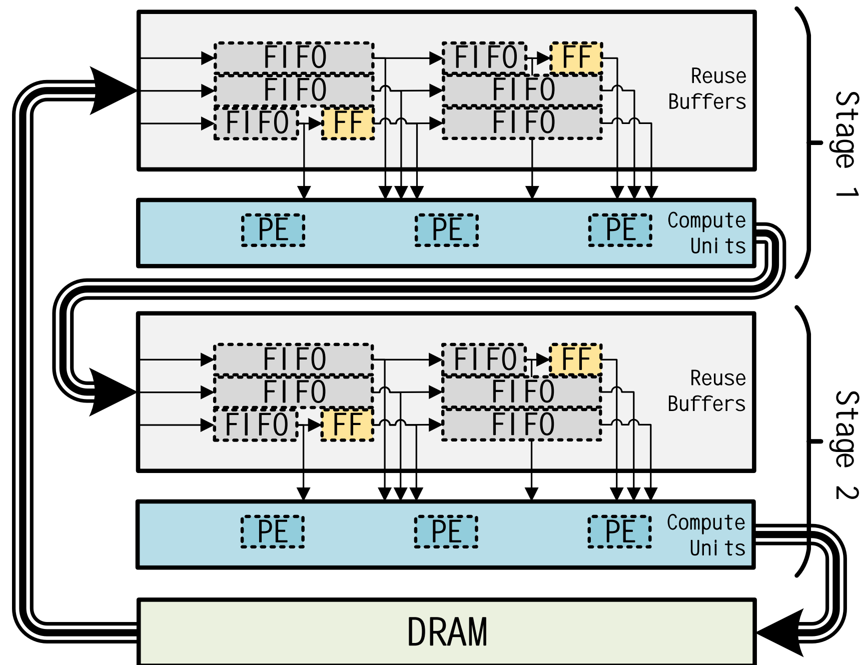


Figure 3.3: Overview of a complete SODA accelerator.

fore sent/received to/from DRAM to achieve the maximum throughput. The SODA automation framework automatically generates modules that handle the memory coalescing and the corresponding host-side data layout manipulation code, which improves external memory throughput without adding complexity to the programming model.

Also, although burst I/O are fully pipelined, the latency is quite long. Therefore, to hide this latency, burst length—i.e., the number of data elements read/written in each DRAM access—has to be large enough. Thanks to the dataflow implementation, the DRAM access can be automatically performed in burst mode with sufficiently long burst length, without the need of coarse-grained pipelining and double buffering as discussed in [27].

3.2 Optimal and Heuristic Computation Reuse

3.2.1 Definitions and Problem Formulation

Reduction operations are operations that are *commutative* and *associative*[1]. Given an n-point stencil kernel with reduction operations

$$Y[y] = g(f_0(X[y + a_0]) \oplus \ldots \oplus f_{n-1}(X[y + a_{n-1}]))$$

The reduction *expression* we are interested in is

$$f_0[a_0] \oplus \ldots \oplus f_{n-1}[a_{n-1}] \tag{3.2}$$

where $f_s[a_s] = f_s(X[y+a_s])$, meaning to apply a point-wise scaling function f_s on the input element in X with an offset of a_s relative to the output element. We shall, e.g., use $[-1][0]$ or $X[-1][0]$ to represent $X[j-1][i]$ when it is clear from the context.

Expressions and schedules: A reduction *expression* defined by Formula 3.2 in does not define a specific computation order. This means the number of non-redundant operations required to compute the expression may vary. To account for that, we define the reduction *schedule* as a specific computation order of an expression. A schedule has a well-defined computational cost in terms of the number of reduction operations $\oplus$ and the number of scaling operations f, e.g., a naïve left-to-right schedule would require $(n-1)$ $\oplus$ operations and n f operations. Although different schedules produce the same computational result mathematically, the computational cost can be different. Note that even if two schedules have the same computational cost, the storage requirement on accelerators can still be different.

Problem formulation: We aim to find a schedule of an expression with 1) the least possible number of $\oplus$ reduction operations, and 2) the least possible number of f scaling operations. Notice that f operations can be reused optimally by creating an intermediate array for each scaled

[1] $\oplus$ is commutative iff $a \oplus b \equiv b \oplus a$. $\oplus$ is associative iff $(a \oplus b) \oplus c \equiv a \oplus (b \oplus c)$. We treat floating-point additions as if they were associative.

operand, we shall focus on the optimal reuse of ⊕ operations in the following part of this section. Furthermore, if multiple schedules have the same number of operations, we aim to find the schedule with the least storage requirement, which will be discussed in Section 4.2.

3.2.2 Optimal Reuse by Dynamic Programming (ORDP)

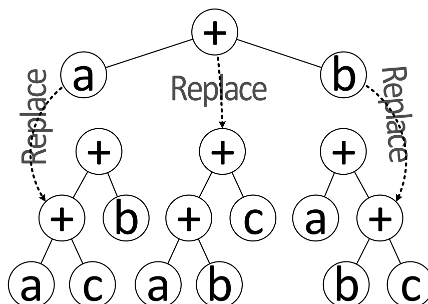


Figure 3.4: Reduction trees of $a + b + c$ augmented from $a + b$.

To discover the optimal reuse, we enumerate over all possible schedules of a reduction expression $opd_0 \oplus opd_1 \oplus ... \oplus opd_{n-1}$ and count the number of unique subexpressions as the number of ⊕ operations. Notice that a schedule with n-operands corresponds to a binary reduction tree, whose n leaf nodes correspond to the n operands and $n - 1$ non-leaf nodes correspond to the $n - 1$ ⊕ operations, we can enumerate all schedules via dynamic programming. As an example, let $a + b + c$ be a 3-operand reduction expression. The schedules of $a + b + c$ can be constructed by adding the third operand c to the existing schedules of $a + b$, while $a + b$ only has 1 trivial schedule, which corresponds to the binary tree shown in the upper part of Figure 3.4. To obtain schedules of $a + b + c$ from $a + b$, we need to replace one node of $a + b$ with a new node, whose children are the original node and c. Since the reduction tree of $a + b$ has 3 nodes, there are 3 replacement outcomes, too. The lower part of Figure 3.4 shows all the 3 reduction trees obtained in this way. The 3 trees correspond to $(a+c)+b$, $(a+b)+c$, and $a+(b+c)$, respectively. In general, assume we have enumerated all schedules of the first k operands, $k = 2, 3, ..., n-1$. To enumerate all schedules of the first $k + 1$ operands, all we need to do is to replace one of the nodes in the

k-operand reduction tree with a new node, whose children are the newly added operand and the original node. By doing so for all $2k-1$ nodes of all k-operand reduction trees, we can obtain all schedules of the first $k+1$ operands. By induction, we can enumerate all schedules of n operands. Such enumeration achieves spatial exploration of computation reuse.

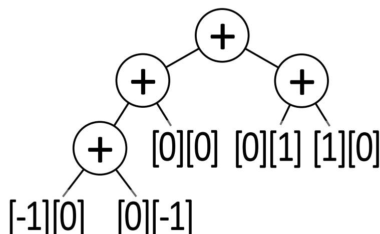


Figure 3.5: Reduction tree of Formula 2.3.

To count the number of operations required for a schedule, we need to count the number of *unique* subexpressions. Since subexpressions can be relatively shifted, we align their access offsets (array references) for comparison. The *aligned access offsets* are obtained by subtracting the least-lexicographical-order [35] offset from all access offsets. As an example, for the schedule given in Formula 2.3, there are 4 subexpressions (including the whole expression itself), each of which corresponds to a non-leaf node shown in Figure 3.5. Among them, two subexpressions, $[-1][0] + [0][-1]$ and $[0][1] + [1][0]$, align to the same $[0][0] + [1][-1]$, which means they can be reused. A hash table is used to count the number of unique subexpressions, where the hash table is keyed by the aligned access offsets and the scaling functions. Subexpressions with the same key indicate reduction operation reuse opportunities. In the previous example, the number of unique subexpressions is 3, which matches the analysis in Section 3.2.1. Access offset alignment achieves temporal exploration of computation reuse.

The number of all possible schedules of an $(n+1)$-operand expression is $(2n-1)!! = 1 \times 3 \times 5 \times \cdots \times (2n-1)$, which is $(2n-1)\times$ that of an n-operand expression, as discussed in the dynamic programming algorithm presented above. This is asymptotically $O\left(\left(\frac{2n-1}{e}\right)^n\right)$. The optimal solution works well when n is not large ($n \leq 10$) but does not scale. Next, we shall present an efficient

heuristic-based solution.

3.2.3 Heuristic Search–Based Reuse (HSBR)

In this section, we present a heuristic search–based reuse (HSBR) discovery algorithm that can help us find near-optimal solutions with polynomial time and space complexity. HSBR is a variant of beam search [2] and is composed of three steps, namely 1) *reuse discovery*, 2) *candidate generation*, and 3) *iterative invocation*.

Reuse discovery enumerates all *pairs* of operands to find potential reuse. If an operand pair appears more than once after alignment, it would be a *reuse pattern* that leads to computation reuse. Note that although we only select pairs of operands, larger patterns are considered since each operand itself can be a subexpression that is composed of multiple operands (e.g., Figure 3.6). For the example of Formula 2.3, after enumerating all pairs of operands, we would find both [−1][0] + [0][−1] and [0][1] + [1][0] align to [0][0] + [1][−1], indicating a reuse opportunity. If no reuse is found in this step, the algorithm terminates.

Candidate generation creates candidate schedules by replacing reuse patterns with new, non-leaf operands. Such non-leaf operands correspond to the intermediate arrays created for reuse, e.g., T in Formula 2.2. Since there can be many different combinations of reuse patterns, this step may generate a large number of candidates. For example, for Formula 2.3, in addition to reusing [−1][0] + [0][−1] for [0][1] + [1][0], we would also generate a candidate schedule that reuses [−1][0] + [0][1] for [0][−1] + [1][0]. For each candidate, we evaluate how much computation is reused by counting the number of unique subexpressions and how much storage is required as will be discussed in Section 4.2. The best W candidates will be selected for the next step. The constant W is the *beam width* in the beam search algorithm.

Iterative invocation enqueues each selected candidate for the next iteration of HSBR. New reuse patterns are found for each candidate separately, but all next-generation candidates are subject to the same constant bound of beam width W. Since the number of selected candidates

in each iteration is $O(W)$ and the number of iterations is $O(n)$ where n is the number of operands in the kernel, the total number of candidates generated and evaluated will be $O(Wn)$. For each candidate, the number of operation required is $O(n^2)$, because we enumerate all pairs of operands. Overall, HSBR is $O(Wn^3)$, which guarantees scalability.

In the remaining part of this subsection, we discuss some optimizations that reduce exploration time and improve quality of result.

3.2.3.1 Operand Selection

We maximize the number of reused operand pairs in each iteration so that the number of iterations is reduced, resulting in faster completion of HSBR, especially for large stencil kernels. This greedy optimization is applied in two places of the candidate generation step. First, for each reuse pattern, we replace as many operand pairs as possible. For example, given $X[0] + 2X[1] + X[2] + 2X[3]+X[4]+2X[5]$, the reuse discovery step would find that the reuse pattern $T[0] = X[0]+2X[1]$ can be reused for $X[0] + 2X[1]$, $X[2] + 2X[3]$, and $X[4] + 2X[5]$. In the candidate generation step, we greedily replace all operand pairs for reuse (i.e., we replace the aforementioned operand pairs with $T[0]$, $T[2]$, and $T[4]$, respectively). Second, in addition to the operand pairs that reuse the same reuse pattern, we also try to apply other reuse patterns if permissible. For example, given $X[0] + 2X[1] + X[2] + 2X[3] + 3X[4] + 4X[5] + 3X[6] + 4X[7]$, we reuse both $T_1[0] = X[0] + 2X[1]$ and $T_2[0] = 3X[0] + 4X[1]$ and generate $T_1[0] + T_1[2] + T_2[4] + T_2[6]$ directly in a single iteration.

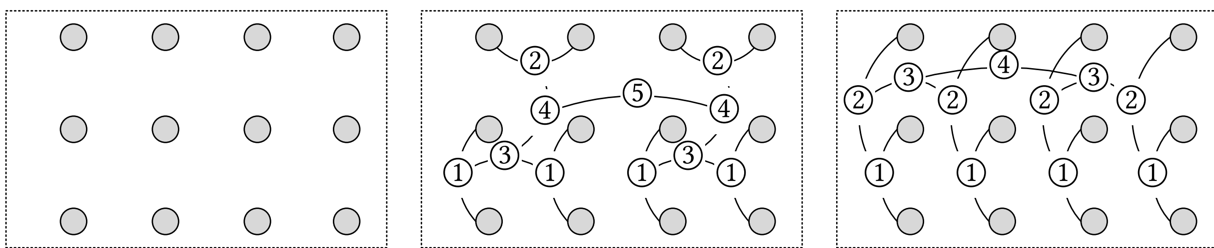
3.2.3.2 Conflict Resolution

When selecting operand pairs for reuse, sometimes not all valid pairs can be selected at the same time. For example, given [0] + [1] + [2] + [3] + [4] + [5], we'll find that [0] + [1] can be reused for 5 different operations, i.e., [0] + [1], [1] + [2], [2] + [3], [3] + [4], and [4] + [5]. However, since these operand pairs overlap, e.g., the first two share the same operand [1], we cannot possibly select all of them for reuse. Although it seems that this problem can be formalized as a graph matching

problem, where the nodes are the operands and the edges are the operand pairs, it cannot be solved using the standard minimum matching because the weight (computational cost) of an edge is not static and may vary in different matchings due to the sharing nature of the computational cost. HSBR resolves the conflicts as follows. Notice that we only consider pairs of operands, for the same reuse pattern, each pair can conflict with at most two other pairs, making the conflict graph bipartite, i.e., there are two conflict-free subsets for each group of conflicting operands. In the previous example, the two choices of conflict-free subsets are $\{[0]+[1],[2]+[3],[4]+[5]\}$ and $\{[1]+[2],[3]+[4]\}$. In the candidate generation step, we generate candidates from both choices. To account for the conflicts between different reuse patterns, we generate multiple candidates prioritizing each reuse pattern while greedily selecting other non-conflicting reuse patterns.

3.2.3.3 Regularity Exaction

Reusing operands spanning multiple dimensions may break regularity and lead to sub-optimality. Take a 4×3 uniform-weight kernel as an example (Figure 3.6a), where the output is the average of the 12 inputs. The aforementioned greedy algorithm selects two reuse patterns (labeled ① and ②) in the same iteration, which ends up with a total of 5 ⊕ operations, as shown in Figure 3.6b. Non-leaf nodes that correspond to the same reuse pattern are labeled with the same number. However,



(a) A 4×3 stencil kernel. (b) A possible operand selection. (c) A better operand selection.

Figure 3.6: Different operand selections on a 4×3 uniform-weight stencil kernel.

if we manually look for reuse, it is not hard to figure out a schedule with only 4 ⊕ operations (Figure 3.6c), which could be generated if we only select patterns along the vertical dimension

(①) in the first iteration of the algorithm. To address this, when the number of reuse patterns exceeds a threshold (e.g., number of operands), candidate generation becomes less greedy and only selects reuse patterns along the same direction.

3.3 Summary

In this chapter, we discussed the three important optimizations for stencil applications: parallelization, communication reuse, and computation reuse. We have demonstrated that, with our novel SODA microarchitecture, communication reuse can be achieved optimally with any parallelization factor. Moreover, we show that computation reuse can be applied on top of parallelization and communication reuse, using an optimal algorithm for small kernels with up to 10 operands, or a heuristic algorithm for large kernels with more than 10 operands. Given the theoretical analysis, we shall discuss the design-space exploration for stencil applications in the next chapter.

CHAPTER 4

Model-Driven DSE for Stencil Applications

For the regular memory accesses in stencil applications, with the theoretical analysis presented in Chapter 3, we can achieve full communication reuse and computation reuse and minimize memory traffic to the lowest possible in theory. However, the remaining design space is still too large to explore manually. In this chapter, we present model-driven design-space exploration (DSE) for stencil applications to guide optimization. Section 4.1 and Section 4.2 discuss DSE for communication reuse and computation reuse, respectively. Section 4.3 explores the trade-off between accelerator frequency and area imposed by different implementation granularity of the same microarchitecture.

4.1 Design-Space Exploration for Communication Reuse

In this section, the programming model for SODA and the corresponding automation framework are discussed first in Section 4.1.1. Under the proposed programming model, the configurable parameters are then discussed in Section 4.1.2. Since these parameters form a large design space and synthesizing an FPGA accelerator is very time-consuming, a resource model and a performance model are proposed in Section 4.1.3 and Section 4.1.4 to predict the post-synthesis resource utilization and the on-board execution performance, respectively. With these models, the large design space can be pruned effectively, which is discussed in Section 4.1.5.

4.1.1 Programming Model

To simplify accelerator kernel design, SODA defines a domain-specific language (DSL) to specify the design parameters as well as the stencil kernel in a concise and high-level way.

As shown in Listing 4.1, the `kernel` statement specifies the name of the stencil kernel. The `input` statement specifies the name, type, and tile size of the input data. Note that the last dimension of tile size is `*` because it is not needed for data linearization and therefore is given at runtime. The `output` statement specifies the name and type of output data as well as the stencil kernel expression to compute it.

```
1  kernel: jacobi2d
2  input float: in(3000, *)  # specifies the tile size
3  output float: out(0, 0) = (
4    in(0, -1) + in(-1, 0) + in(0, 0) + in(1, 0) + in(0, 1)) * 0.2f
5  unroll factor: 3
6  iterate factor: 2
7  # SODA supports multiple stages:
8  # local float: tmp(0,0) = (
9  #   in(0, -1) + in(-1, 0) + in(0, 0) + in(1, 0) + in(0, 1)) * 0.2f
10 # output float: out(0,0) = (
11 #   tmp(0, -1) + tmp(-1, 0) + tmp(0, 0) + tmp(1, 0) + tmp(0, 1)) * 0.2f
12 # SODA supports multiple arrays as input:
13 # local float: t(0, 1) = in(0, 0) + tmp(0, 2)
```

Listing 4.1: 2-dimensional Jacobi kernel in SODA DSL.

If the kernel contains more than 1 stage, intermediate stages can be specified with `local` statements. The `unroll factor` statement specifies the number of PEs in each stage. The `iterate factor` statement automatically implements the specified number of iterations, which simplifies the expression of iterative kernels. Note that the expressions are not restricted to have only 1 input array; the SODA compiler (`sodac`) is capable of processing kernels taking multiple arrays as inputs. To connect with user-defined code, the SODA automation framework provides a concise C/C++ binding of the generated accelerator.

To reduce the burden of programming FPGAs, we develop a fully automated framework to generate efficient hardware accelerators for stencil computations. Currently, the SODA automation framework interfaces with Xilinx SDAccel implementation flow. The automation framework takes a high-level DSL as user input, implements tiling automatically, uses the SODA microarchitecture discussed in Section 3.1.2 as building blocks for implementing stages, automatically solves the dependencies among the stages, and connects multiple stages or iterations with dataflow optimization.

The complete SODA automation framework is shown in Figure 4.1. The SODA compiler (`sodac`) parses the SODA DSL, does a source-to-source transformation, and generates the HLS C++ code as the kernel and the OpenCL API code for the host. Then `gcc` will be invoked to compile and link the OpenCL API with user-defined application and `xocc` will be invoked to launch the Xilinx SDAccel flow to do HLS, logic synthesis, placement, and routing. Host program and FPGA bitstream will be the eventual synthesis results, ready for execution on any compatible environment. In addition, there is a standalone design-space exploration (DSE) framework provided by SODA, which is used to automatically tune the kernel configuration parameters for optimal performance.

4.1.2 Configurable Parameters

Tile sizes $T_0, T_1, ..., T_{m-2}$. To generate valid accelerators, the linearization convention has to be determined before synthesis. Since size of all but the last dimension appear in Formula 3.1, the size of the first $m - 1$ dimensions of the input must be determined before synthesis. However, the exact input size may not be determined at the time of accelerator design. Moreover, the input may be too large to fit the on-chip storage. Therefore, tiling is a reasonable design choice. Note that our automation framework automatically does tiling with the size specified in the DSL and the user does not have to do tiling manually. In this dissertation, we argue that with the ever-increasing resolution of sensors, the input is sufficiently large, and the tile size is only limited by the on-chip storage size.

Unroll factor k. To avoid the confusion with the total number of PEs in a complex kernel, the term *unroll factor* is used to represent the number of PEs in each stage, in analogy to the case where the number of PEs generated for an unrolled loop is determined by the unroll factor. Since all building-block modules in our dataflow architecture can be fully pipelined (which is optimal for throughput), we argue that it does not make sense to use different unroll factors among different stages, which will cause throughput mismatch among stages and increase the initiation interval (II) of the stage with higher unroll factor. The SODA automation framework implements all stages with the same unroll factor.

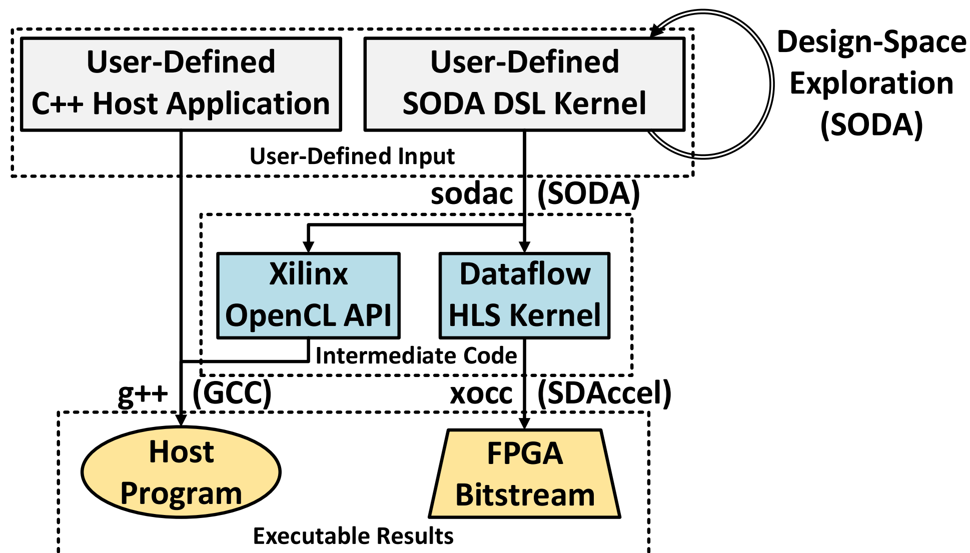


Figure 4.1: SODA automation framework.

Iterate Factor q. For iterative stencil kernels, *iterate factor* is the number of iterations implemented in the accelerator. In practice, the number of iterations required by the application is often much greater than the number of iterations can be implemented in an accelerator. To complete all required iterations, the execution kernel must be invoked multiple times. In this dissertation, we assume that the number of such invocation is sufficiently large and all iterate factors that can be implemented under the resource constraint are permissible.

4.1.3 Resource Model

The SODA DSE framework models all four types of resources on FPGAs, i.e. LUT, FF, DSP, and BRAM. To make the model more practical, we target post-synthesis resource utilization instead of the HLS result. Since the HLS report contains modularized details and can be obtained within minutes whereas it is hard to distinguish user-level modules from the post-synthesis report, and it takes hours to obtain, as the first step we take the HLS report and model the HLS resource utilization. As the second step, based on the HLS report and post-synthesis report, we then make adjustments to the HLS model accordingly so that the final model can reflect the post-synthesis resource utilization. Thanks to the modularized dataflow implementation, the resource utilization of a SODA accelerator can be accurately modeled at the dataflow module level. Those modules can be divided into three categories.

Compute modules consume the majority of resources for computation. The SODA DSE framework obtains the resource utilization of each compute model by running HLS. Since compute modules in the same stage for all iterations have the same structure, they only need to go through HLS once.

Forwarding modules consume the majority of the remaining resources for communication. For a forwarding module with a ϕ-bit wide data type, LUT and FF consumption grows linearly with fanout κ. For those who implement a FIFO of depth $\eta > 0$, there is a constant LUT and FF overhead for the control logic of the FIFO, in addition to the SRL or BRAM used to implement the FIFO. Since only small FIFOs are implemented as SRLs with LUTs, LUTs used for this purpose are much less than those used to implement logics. Thus, the SRL contribution to the LUT utilization is ignored in the model. The coefficients in the model are kernel-independent and can be obtained from a series of pre-executed HLS results. η, ϕ, and κ are determined for each module by `sodac`, according to the tile size and unroll factor configuration and the stencil kernel itself. Forwarding modules do not use any DSP.

We observe that it is very hard to develop a closed-form analytical formula to predict the

BRAM usage $\Theta(\phi, \eta)$ from data width ϕ and depth η, due to the undocumented optimizations performed by Xilinx tools. For example, we observe that a 16-bit × 8K FIFO is implemented with 8 BRAMs whereas a 16-bit × 16K FIFO is implemented with only 15 BRAMs. To accurately predict BRAM utilization, SODA invokes Xilinx Vivado to synthesize a single FIFO and obtains its BRAM utilization from the synthesis report. Such a simple synthesis only takes 3 to 4 minutes and SODA keeps the results in a database so that the result can be reused. Notice that $\Theta(\phi, \eta)$ is a step function of η and does not depend on module fanout κ, the database can be frequently reused and rarely updated. Among all design-space exploration performed in the experiments done in Section 4.1.6, only 115 entries are needed. Since these entries are shared for different kernels and can be generated in parallel, time spent on the $\Theta(\phi, \eta)$ model is negligible compared with the time needed to synthesize a complete accelerator.

Auxiliary modules, including the interconnections with DRAM, control signals, and memory coalescing modules, constitute the remainder of resource consumption. The I/O modules and control modules are independent with the tile size and the iterate factor, whereas the unroll factor has a very weak influence on memory coalescing modules, which is negligible in size compared with the total resource utilization. Consequently, resource utilization of auxiliary modules is considered as a constant and can be obtained from the pre-executed HLS results.

With a given kernel and configuration, the number and parameter of all modules can be determined analytically via the microarchitecture generation algorithm presented in Section 3.1.2. The total resource consumption is the sum of resource consumption of each module. Note that the BRAM and DSP models obtained above are already reflecting the post-synthesis results. Experimental results in Section 4.1.6.2 show 1.84% and 0% average prediction errors for BRAM and DSP, respectively. For the LUT and FF utilization, we observe that the post-synthesis utilization of LUT and FF have a linear relationship with the utilization reported by HLS. Moreover, this linear relationship does not depend on the application kernel or configuration parameters. Therefore, we adjust our model for LUT and FF with a linear adjustment function to get post-synthesis models. Experimental results in Section 4.1.6.2 show 6.23% and 7.58% average prediction errors for

LUT and FF, respectively.

4.1.4 Performance Model

As established in Section 3.1.1, our optimization objective is the sustained throughput H, which can be measured by the number of output data elements per unit time. For an accelerator running at frequency f and having tile size $\vec{T} = (T_0, T_1, ..., T_{m-1})$, unroll factor k, and iterate factor q, the ideal throughput of the kernel is

$$H_{\text{ideal}}(k, q, \vec{T}) = kqf \prod_{d=0}^{m-1} \frac{T_d - q \cdot (S_d - 1)}{T_d} \tag{4.1}$$

where m is the number of dimensions and S_d is the stencil window size in each iteration. For non-iterative stencil kernels, $q \equiv 1$.

H_{ideal} may not be achievable since the hardware put constraints on H in two aspects: 1. External bandwidth limits the effective unroll factor 2. Available resource limits the achievable tile size, unroll factor, and iterate factor. The constraints are modeled as

$$H \leq H_{\text{ideal}}(k, q, \vec{T}) \tag{4.2}$$

$$kf\,W \cdot \frac{H}{H_{\text{ideal}}(k, q, \vec{T})} \leq B^{\text{MAX}} \tag{4.3}$$

$$R^{(\text{AUX},\chi)} + kqR^{(\text{CP},\chi)} + q \cdot R^{(\text{FW},\chi)}(k, \vec{T}) \leq R^{(\text{MAX},\chi)} \tag{4.4}$$

$\chi \in \{\text{LUT}, \text{FF}, \text{DSP}, \text{BRAM}\}$ represents the type of resource. H is the achieved throughput. B^{MAX} is the maximum available DRAM bandwidth. W is the total width of input and output data types. $R^{(\text{MAX},\chi)}$ is the maximum available χ resource. $R^{(\text{AUX},\chi)}$ is the resource consumption of auxiliary modules. $R^{(\text{CP},\chi)}$ is the resource consumption of compute modules of a single PE in a single iteration. $R^{(\text{FW},\chi)}(k, \vec{T})$ is the resource consumption of forwarding modules in a single iteration, which is a function of k and $\vec{T}$. Note that as a result of the dataflow implementation, the frequency of accelerators achieved is always within 10% of the target, according to our experimental results. Therefore, in the performance model we treat f as a constant. Experimental results in Section 4.1.6.2 show 4.22% average prediction error for performance.

4.1.5 Design-Space Pruning

As shown in Section 4.1.4, both the objective function and the constraints are non-linear. Therefore, we do not seek closed-form solution. Instead, we prune the design space and use branch-and-bound method to find the optimal configuration.

The design space of unroll factor k is limited by the external interface. On our evaluation platform, the input and output interface are both 512 bits wide. To avoid complex multiplexers, k is restricted to powers of 2. With the minimum data type width being 8 bits, this effectively reduces the design space of k to have at most 6 points. For iterative kernels, the iterate factor q is a positive integer, but it is bounded by the constraint in Formula 4.4. That is, the total number of PEs is bounded by the available resources. On our evaluation platform, for the simplest PEs, $kq \le 10^2$. For non-iterative kernels, $q \equiv 1$. The design space of tile sizes $T_0, T_1, ..., T_{m-2}$ is much larger compared with k and q, especially for high-dimensional stencils. Nevertheless, notice that the bound of H is monotonically increasing with respect to T_d in each dimension d for every given k and q, $T_0, T_1, ..., T_{m-2}$ can be efficiently searched via branch-and-bound. Note that T_{m-1} is not part of the design space because it is determined by the input and is a runtime parameter instead of a design parameter. With the on-chip storage size being several megabytes, the total size of design space can then be reduced to less than 10^6 and can be explored within a few minutes.

4.1.6 Experimental Evaluation

4.1.6.1 Experiment Setup

We evaluate SODA with a Xilinx Kintex UltraScale FPGA. The AlphaData ADM-PCIE-KU3 board used in our experiments is equipped with XCKU060 FPGA and 2 × 8 GB 1600 MT/s DDR3 DRAMs. High-level synthesis is performed by Xilinx Vivado HLS. Xilinx SDAccel 2017.2, the latest version offering support for this platform, is used for system integration. The target clock frequency is 250 MHz. The CPU experiments are conducted on a server with two Intel Xeon E5-2620 v3 CPUs and 4 × 16 GB 2133 MT/s DDR4 DRAMs.

We use 3 non-iterative benchmarks and 4 iterative benchmarks to evaluate our SODA framework. The benchmarks cover a wide range of application domains and have been used in previous published works [35, 135]. Among the 7 benchmarks, SOBEL 2D is used for edge detection in the image processing domain. DENOISE 2D/3D are used in the medical imaging domain [35]. The four iterative benchmarks, namely JACOBI 2D/3D, SEIDEL 2D, and HEAT 3D, are used in the linear algebra domain [135].

Table 4.1: Benchmarks used for communication reuse.

Benchmark	Iterative?	Data Type	Optimal Configuration		
			Tile Size	Unroll Factor	Iterate Factor
SOBEL 2D [35]	No	`uint16_t`	524302	32	—
DENOISE 2D [35]	No	`float` (2 in, 1 out)	21846	8	—
DENOISE 3D [35]	No	`float` (2 in, 1 out)	156×157	4	—
JACOBI 2D [135]	Yes	`float`	16392	8	10
JACOBI 3D [135]	Yes	`float`	181×182	8	6
SEIDEL 2D [135]	Yes	`float`	32768	8	9
HEAT 3D [135]	Yes	`float`	256×257	8	5

4.1.6.2 Model Validation

In Section 4.1.3 and Section 4.1.4 we proposed a resource model and a performance model. In this section, we run two sets of experiments to validate our model. In the first set of experiments, we fix the tile size and explore the unroll factor. For iterative benchmarks, we also explore different iterate factors. In the second set of experiments, we fix the unroll factor and explore the tile size. In total, 75 different configurations are synthesized. The average of the achieved frequency is 245.66MHz with the lowest being 229.1MHz. To validate the resource model, we compare the model prediction against the *post-synthesis* resource utilization. To validate the per-

formance model, we run *on-board* experiments with 4 different sizes of input and obtain sustained throughput via linear regression of the execution time and the number of pixels processed. The measurement errors of throughput are within 1% for all configurations. The average error rate of the model prediction is listed in Table 4.2.

Table 4.2: Average error rate of model prediction.

Prediction Item	BRAM	DSP	LUT	FF	Throughput
Average Error	1.84%	0%	6.23%	7.58%	4.22%

4.1.6.3 Performance Analysis

Table 4.3 compares the performance of the optimal configurations found by the SODA DSE framework with four baselines. To make the CPU baseline realistic, all benchmarks are rewritten in Halide [146] DSL and optimized via tiling, parallelization, and vectorization. The resulting CPU code is able to utilize all 24 hyper-threads on our server. We implement the FPGA baselines using the methodologies proposed in [35], [135], and [183]. Note that [135] and [183] do not target non-iterative stencil algorithms and [35] does not target iterative stencil algorithms. The #Op column shows the number of operations per iteration.

As shown in Table 4.3, the 24-thread CPU baseline outperforms SODA for non-iterative benchmarks. This is because non-iterative benchmarks are bounded by communication. The CPU platform has 4 DDR4 channels with 68.3 GB/s theoretical bandwidth in total whereas the FPGA platform only has 2 DDR3 channels with 25.6 GB/s theoretical bandwidth. If they have the same external memory bandwidth, SODA can outperform CPU by 1.65x on average. For iterative benchmarks, SODA (and other FPGA platforms) can compute multiple iterations without extra accesses to the external memory whereas the CPU platform has to make a trade-off between memory access and redundant computation. Consequently, SODA outperforms the CPU baseline by 2.76x on average.

Table 4.3: Performance comparison of SODA and previous work.

Benchmark	#Op	Platform	Throughput pixel/ns	Throughput Op/ns	Performance (Norm. to 24t-CPU)
SOBEL 2D	16	CPU	6.66	106.59	1
		[35]	0.25	4.00	0.04
		SODA	5.37	85.86	0.81
DENOISE 2D	45	CPU	1.86	83.55	1
		[35]	0.25	11.07	0.13
		SODA	1.05	47.07	0.56
DENOISE 3D	57	CPU	1.91	109.01	1
		[35]	0.25	14.24	0.13
		SODA	0.93	53.16	0.49
JACOBI 2D	5	CPU	5.49	27.44	1
		[135]	16.67	83.34	3.04
		[183]	17.20	86.01	3.13
		SODA	18.01	90.04	3.28
JACOBI 3D	7	CPU	4.24	29.66	1
		[135]	4.72	33.01	1.11
		[183]	9.86	69.04	2.33
		SODA	12.00	83.98	2.83
SEIDEL 2D	6	CPU	5.82	34.90	1
		[135]	15.03	90.18	2.58
		[183]	15.99	95.95	2.75
		SODA	16.22	97.34	2.79
HEAT 3D	15	CPU	4.21	63.18	1
		[135]	4.70	70.57	1.12
		[183]	6.65	99.70	1.58
		SODA	8.99	134.91	2.14

Thanks to scalable, fine-grained parallelism provided by the SODA microarchitecture, SODA shows 9.82x speed up on average compared with [35]. Compared with [135], SODA achieves 1.08x average speedup on 2D benchmarks and 2.23x average speedup on 3D benchmarks. Compared

with [183], SODA achieves 1.03x average speedup on 2D benchmarks and 1.28x average speedup on 3D benchmarks. The speedup comes from three aspects: 1. SODA uses less resources for communication and can therefore implement more PEs. 2. SODA can accommodate larger tile sizes and is thus less sensitive to the halo size (which is proportional to the iterate factor). 3. SODA provides scalable, fine-grained spatial parallelism and can reduce the halo size caused by temporal parallelism. The difference of the speedup on 2D and 3D benchmarks is due to (2), where 3D benchmarks have much smaller tile sizes in a single dimension compared with 2D ones (as shown in Table 4.1) and are thus more sensitive to large halo size. The optimal microarchitecture and systematic DSE brought by SODA give more speedup for 3D benchmarks compared with [183]. In addition, the optimal configuration for SODA can be obtained from fast and automated DSE, which previous accelerator designs do not provide.

4.2 Design-Space Exploration for Computation Reuse

Given the number of parallel processing elements (i.e., the parallel factor), Section 4.1 discusses Pareto-optimal communication reuse buffers and Section 3.1.3 proves that the minimum on-chip storage required by a stencil kernel is determined by the sum of the reuse distance and the parallel factor. Since the parallel factor is an additive term and can be chosen independently, the storage requirement of a stencil kernel can be fully characterized by the reuse distance, independent of the underlying hardware platform or microarchitecture. For a complex multi-stage kernel, which is common after computation reuse is applied (e.g., Formula 2.2), the conclusion from Section 3.1.3 still holds, and the total storage requirement can be characterized by the total reuse distance[1]. However, the total reuse distance is no longer a constant attribute of the kernel. We shall show an example of such case, followed by an algorithm that minimizes it. The minimum total reuse distance obtained will be used to characterize the storage requirement with computation reuse.

[1]W.l.o.g. we assume all element sizes are the same for conciseness.

4.2.1 Total Reuse Distance for a Complex Stencil Kernel

We start with the following example involving two input arrays X_1, X_2, an intermediate array T, and an output array Y:

$$\begin{aligned} T[2] &= X_1[0] + X_1[1] + X_2[0] + X_2[1] \\ Y[0] &= X_1[3] + X_2[3] + T[0] + T[2] \end{aligned} \tag{4.5}$$

The reuse distances for X_1, X_2, and T are $X_1[0] \cdots X_1[3] = 3$, $X_2[0] \cdots X_2[3] = 3$, and $T[0] \cdots T[2] = 2$, respectively. The total reuse distance is $3 + 3 + 2 = 8$. Notice that Formula 4.5 implies $T[2]$ and $Y[0]$ are produced at the same time, we can shift the production of T and make $T[4]$ be produced at the same time as $Y[0]$, i.e.

$$\begin{aligned} T[4] &= X_1[2] + X_1[3] + X_2[2] + X_2[3] \\ Y[0] &= X_1[3] + X_2[3] + T[0] + T[2] \end{aligned} \tag{4.6}$$

The reuse distances become $X_1[2] \cdots X_1[3] = 1$, $X_2[2] \cdots X_2[3] = 1$, and $T[0] \cdots T[4] = 4$, respectively. The total reuse distance becomes $1 + 1 + 4 = 6 < 8$. Obviously, the total reuse distance for a complex stencil kernel may vary as the relative offset between stages change. Section 4.2.2 will discuss how to minimize it.

4.2.2 Minimizing Total Reuse Distance

Assume we implement our stencil accelerator with a synchronous clock. Given a stencil kernel in which q arrays $\{Y_t | t = 0, ..., q-1\}$ are involved, let $\{Y_s\}$ be the children of Y_t and $Y_s[0]$ consume $\{Y_t[a_u]\}$ from Y_t. For example, in Formula 4.5, X_1, X_2, T, Y are the arrays involved. T is a child of both X_1 and X_2. $T[0]$ consumes X_1 and X_2 at $X_1[-2]$, $X_1[-1]$ and $X_2[-2]$, $X_2[-1]$, respectively. Let $\{Y_t[p_t] | t = 0, ..., p-1\}$ be produced at the same cycle. $\{p_t\}$ are the variables to be determined. The reuse distance of each Y_t is

$$D_t = p_t - \min_{s,u} (p_s + a_u | s \in \text{children}(t), u \in \text{accesses}(t \longrightarrow s))$$

Our goal is to minimize the total reuse distance $\sum_t D_t$. For Formula 4.5,

$$T[p_T] = X_1[p_T - 2] + X_1[p_T - 1] + X_2[p_T - 2] + X_2[p_T - 1]$$

$$Y[p_Y] = X_1[p_Y + 3] + X_2[p_Y + 3] + T[p_Y] + T[p_Y + 2]$$

$$D_{X_1} = p_{X_1} - \min(p_T - 2, p_Y + 3) \qquad D_T = p_T - p_Y$$

$$D_{X_2} = p_{X_2} - \min(p_T - 2, p_Y + 3)$$

The constraint is that an array cannot be consumed before produced:

$$p_t \geq p_s + a_u, \forall t, s \in \text{children}(t), u \in \text{accesses}(t \longrightarrow s)$$

For Formula 4.5, the constraints are:

$$p_{X_1} \geq p_T - 2 \quad p_{X_1} \geq p_T - 1 \quad p_{X_1} \geq p_Y + 3 \quad p_T \geq p_Y$$

$$p_{X_2} \geq p_T - 2 \quad p_{X_2} \geq p_T - 1 \quad p_{X_2} \geq p_Y + 3 \quad p_T \geq p_Y + 2$$

Notice that each constraint is of the type $x_i - x_j \leq c_{ij}$, this is a systems of difference constraints (SDC) problem and can be solved optimally in polynomial time [41]. For Formula 4.5, the solution is $p_{X_1} = p_{X_2} = p_Y + 3, p_T = p_Y + 4$, which gives the minimum total reuse distance of 6 and matches Formula 4.6.

4.2.3 Experimental Results

We extend the SODA compiler presented in Section 4.1 to implement the computation-reuse algorithms. DSE is written in C++ and runs on a single thread of Intel Xeon E5-2699 v3 CPU. Synthesis is performed by Vivado 2019.1, targeting the Alveo U250 board. The stencil kernels used in the experiments include eight Laplacian kernels used in [20, 103, 135, 179], three image stabilization kernels used in [15], a Gaussian smoother kernel used in pose detection [100], and two biharmonic operator kernels used in [55, 62]. Details about the kernels are listed in Table 4.4. In addition to these real-world benchmarks, we also generate artificial 3×3 kernels to assess the optimality gap between the heuristic algorithm and the optimal one.

[2] Marked benchmarks use 8-bit integers; others use 32-bit `float`.

Table 4.4: Stencil benchmarks used in the experiments.

Name	Computation	Size	Name	Computation	Size
s2d5pt	weighted sum of 5	3×3	s2d33pt	weighted sum of 33	17×17
f2d9pt	weighted sum	3×3	f2d81pt	weighted sum	9×9
s3d7pt	weighted sum of 7	3×3×3	s3d25pt	weighted sum of 25	9×9×9
f3d27pt	weighted sum	3×3×3	f3d125pt	weighted sum	5×5×5
contrast[2]	weighted sum of 197	17×17	erosion[2]	minimum	19×19
xcorr[2]	sum except center	19×19	smoother	weighted sum	25×25
bigbiharm	weighted sum of 25	7×7	lilbiharm	weighted sum of 13	5×5

4.2.3.1 Number of Operation Reduction

Table 4.5 shows the number and type of operations required to produce each output element. The performance of the kernels are fixed to produce 1 output element per clock cycle and all off-chip communication is fully reused. The baseline SODA [20] compiler implements the kernels without computation reuse optimization. Note that SODA outperforms previous papers [35, 135, 183] by up to 9.82× [20]. DCMI [103] is a recent work that synthesizes iterative stencil kernels to FPGAs. DCMI removes redundant multiplication operations, but not the addition (reduction) operations. HSBR shows the result of our heuristic algorithm. Note that for kernels that are small enough (less than 10 points), we are able to verify that *the heuristic algorithm actually produces the optimal result.* On average, our presented algorithm reduces the reduction operations by 58.2%[3].

4.2.3.2 Impact of Design-Space Pruning Heuristics

Figure 4.2 shows the average operation reduction and the design-space exploration (DSE) time with different beam widths and heuristics used in HSBR. Time is normalized per benchmark

[3] $= 1 - \text{GeoMean}\{target/baseline\}$

Table 4.5: Operation reduction. **Bold** items are verified to be optimal.

Kernel	Pointwise Operations		Reduction Operations	
	SODA [20]	DCMI [103]/HSBR	SODA/DCMI	HSBR
s2d5pt	5	1 *(-80%)*	4	**3** *(-25%)*
s2d33pt	33	9 *(-73%)*	32	24 *(-25%)*
f2d9pt	9	3 *(-67%)*	8	**6** *(-25%)*
f2d81pt	81	15 *(-82%)*	80	48 *(-40%)*
s3d7pt	7	1 *(-86%)*	6	**5** *(-17%)*
s3d25pt	25	5 *(-80%)*	24	20 *(-17%)*
f3d27pt	27	4 *(-85%)*	26	14 *(-46%)*
f3d125pt	125	10 *(-92%)*	124	40 *(-68%)*
contrast	197	30 *(-85%)*	196	113 *(-42%)*
erosion	0	0	360	12 *(-97%)*
xcorr	0	0	359	13 *(-96%)*
smoother	625	91 *(-85%)*	624	336 *(-46%)*
bigbiharm	25	5 *(-80%)*	24	14 *(-42%)*
lilbiharm	9	3 *(-67%)*	12	9 *(-25%)*
average[3]	—	*-81%*	—	*-58%*

to obtain meaningful averages over different benchmarks. In general, larger beam width yields better results, but requires longer DSE time. Operation selection speeds up HSBR by reducing the search depth. Conflict resolution adds some over-pruned points back to the design space and thus compensates some quality of result loss caused by operation selection. Regularity exaction further improves the quality and the runtime by prioritizing regular patterns.

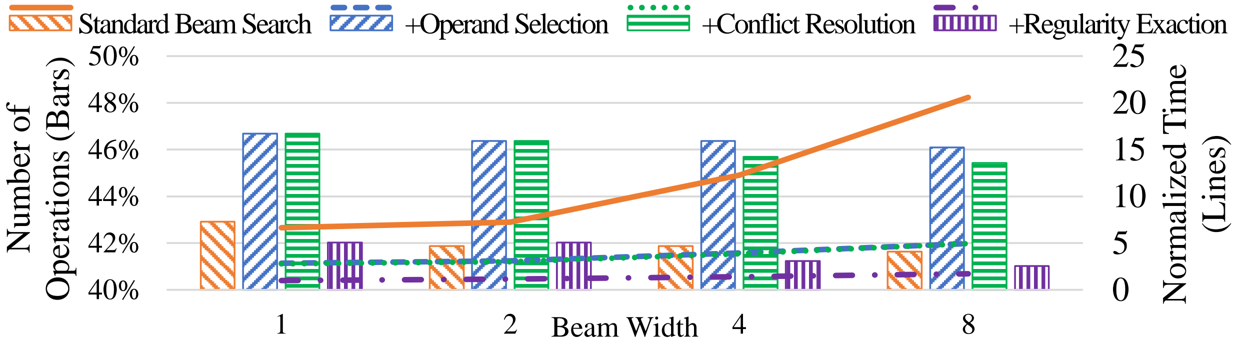


Figure 4.2: Impact of heuristics in HSBR.

4.2.3.3 Performance Boost for Compute-Intensive Stencil

Stencil computation can be compute-intensive if it is iterative [20, 135, 166, 183], or has many operations per output. For compute-intensive stencil kernels, computation reuse can save resources (Section 4.2.3.4) and directly result in a performance boost. We compare the 8 iterative kernels with CPU/GPU results from [179] in Figure 4.3. Note that [179] includes all three aspects of stencil optimizations, i.e., parallelization, communication reuse, and computation reuse. All FPGA implementations are scaled up to use the available DSPs and runs at 100 – 125 MHz[4]. On average, DCMI [103] achieves 1.6× speedup over SODA [20], whereas our proposed HSBR algorithm achieves 2.3×. Moreover, thanks to the highly customized datapaths and fully pipelined microarchitecture, HSBR outperforms multicore Xeon Gold CPU by 10.9×, many-core Xeon Phi processor by 3.17×, and P100 GPU by 1.53× on average, respectively.

4.2.3.4 Resource Consumption Reduction

Figure 4.4 compares the resource usage of the DCMI optimization and our proposed HSBR algorithm with the baseline SODA implementation. Flip-flop (FF) usage is not reported in the figure because it is tightly coupled with look-up table (LUT) usage and is never used more than LUTs.

[4]Designs are HLS-based prototypes and are not fine-tuned for high-frequency [77]. Section 4.3 discusses our effort to improve clock frequency.

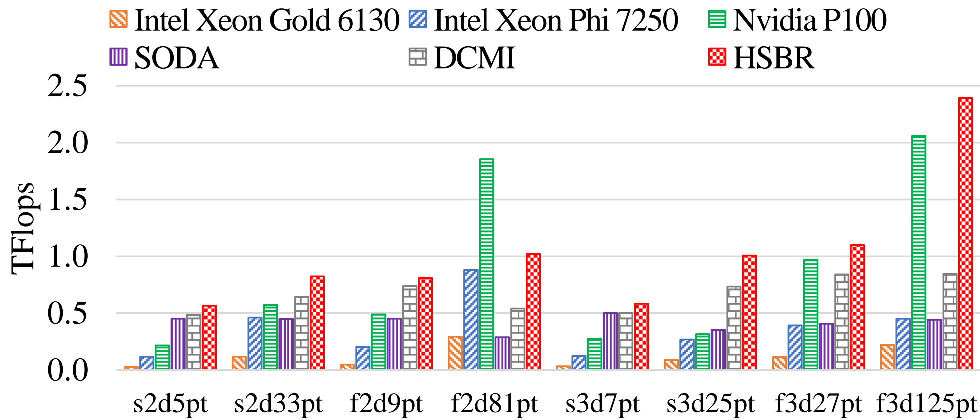


Figure 4.3: Performance of iterative kernels.

From the figure, we can see that both optimizations can save computational resources like LUTs and digital-signal processors (DSP) compared with the baseline implementation, possibly at the cost of storage resources (e.g., block random-access memories, BRAMs). On average, DCMI uses 85.1% LUT and 62.6% DSP with 100.0% BRAM usage (compared with SODA baseline). The reduction on LUT and DSP is from the reuse of multiplication operations, and the BRAM usage is the same as SODA since reusing multiplication can be done without additional storage. The HSBR algorithm, on the other hand, only uses 41.0% LUT and 45.4% DSP with 123.7% BRAM usage (compared with SODA baseline). For large kernels (e.g., contrast, erosion, and xcorr), the baseline implementations generate very deep pipelines that lead to a high BRAM usage. With computation reuse, the kernels are decomposed into smaller ones with shallower pipelines, which can significantly reduce the BRAM usage. The geometric mean of BRAM usage is strongly biased by those cases; after excluding them, the average BRAM usage is 231.9% for HSBR. Note that although the storage (BRAM) usage with computation reuse applied can be as high as 7×, we argue that one can scale up the performance at the cost of computational resources (LUTs and DSPs) without significant increase of storage resources, which makes it reasonable to trade-off storage for computation. Actually, when we scale up each benchmark, we find that BRAM usage never bottlenecks the resource usage; the bottleneck is always DSP (for floating-point numbers) or LUT (for fixed-point numbers).

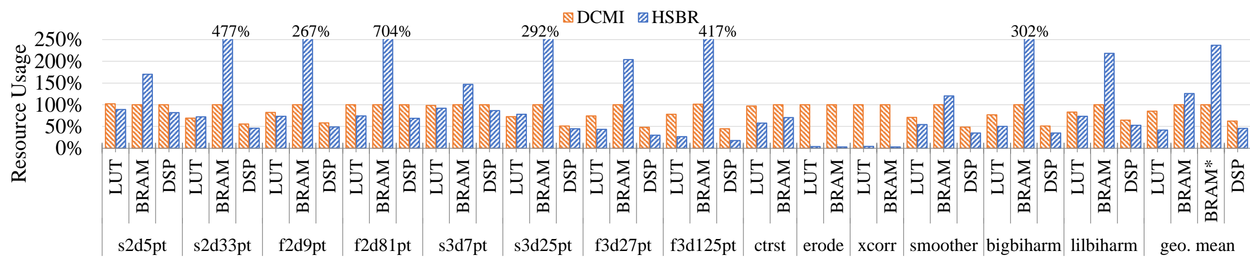


Figure 4.4: Resource usage reduction. The normalization baseline is SODA [20]. BRAM* excludes erosion and xcorr to avoid being biased. Truncated bars are marked with values.

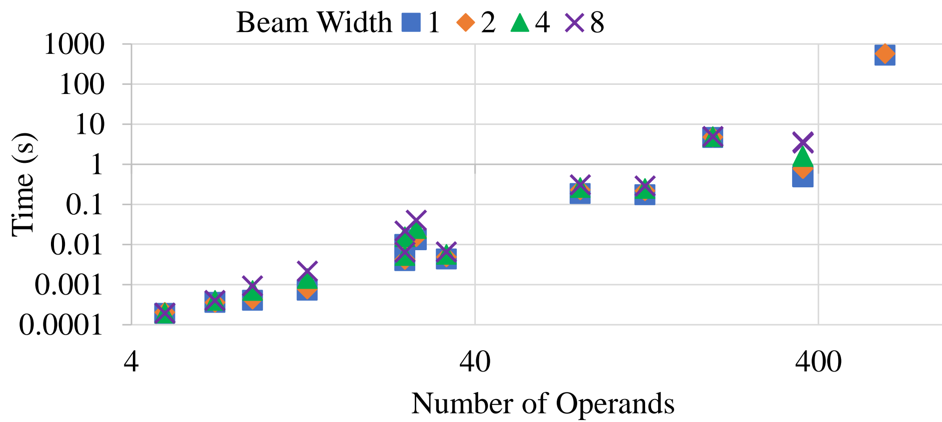


Figure 4.5: Polynomial scalability of HSBR. Different points in the same shape correspond to different benchmarks.

4.2.3.5 Design-Space Exploration Cost

The optimal algorithm scales up to 10-point stencil kernels and runs in 10 minutes with 6 MiB memory. Although the memory usage remains low, scaling to 11 points requires more than 2 hours on our test machine. Figure 4.5 shows the HSBR design-space exploration (DSE) time with various beam widths. Note that since the DSE time is tightly coupled with the kernels, the data points do not align well on a straight line. Since beam search has bounded memory complexity, the memory consumption of HSBR is moderate (< 100 MiB). In general, the cost of the DSE becomes low with the heuristic algorithm.

4.2.3.6 Optimality Gap

Although it is impossible to assess the optimality gap for all kernels, we assess the gap between the heuristic algorithm and the optimal algorithm for small kernels. In addition to the real-world benchmarks, we randomly generate artificial 3×3 kernels to examine how well the heuristics perform. Out of the 11528 kernels we generated randomly, there are 5281 kernels with computation reuse opportunity, and our heuristic algorithm can find *all* of them with the least number of required operations with a beam width of 3. Even with the storage overhead (total reuse distance) taken into consideration, HSBR can yield the optimal reuse buffer size with a beam width of 4. This is shown in Figure 4.6.

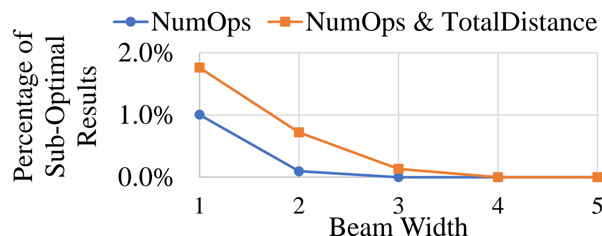


Figure 4.6: Optimality gap of HSBR on 5281 artificial 3×3 kernels.

4.3 Design-Space Exploration for Frequency/Area Trade-Off

Different architectural implementations may lead to very different quality of result for the same theoretically optimal design. In particular, as presented in Section 3.1, SODA uses a fine-grained dataflow microarchitecture, which is composed of many small forwarding modules and compute modules. Each module in SODA is purely functional, i.e., they do not have side effects nor maintain an internal state, which means that we can group small modules into larger ones without changing the functionality of the whole kernel. Smaller modules restrict control signals to each module locally and thus are good for high frequency [77]. Larger modules, on the other hand, can better share and amortize hardware resources for loop control, reuse buffer, etc., and are thus more area-efficient. In this section, we introduce 4 different strategies to merge small modules

and explore the frequency/area trade-off for SODA.

4.3.1 No Module Merging

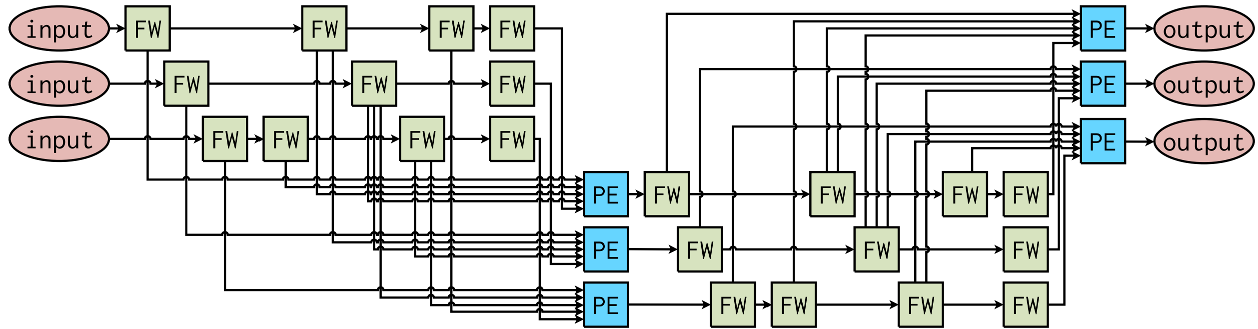


Figure 4.7: A SODA kernel with 3 PEs per iteration and 2 iterations in total.

Figure 4.7 shows an iterative SODA kernel with 3 PEs per iterations and 2 iterations in total. The red ellipses represent input or output modules. The green boxes represent forwarding modules. The blue boxes represent computation modules (processing elements). Without merging, each box is scheduled statically and independently, with ready-valid handshake interfaces for the streaming inputs and outputs (shown as solid arrows connecting the boxes in Figure 4.7). This example is the basis of the examples illustrated in the 4 different merging strategies.

4.3.2 Fine-Grained Module Merging

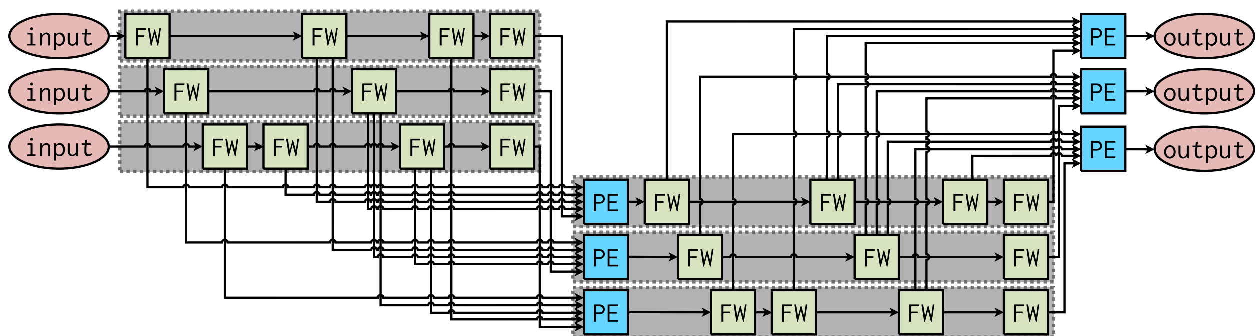


Figure 4.8: Fine-grained module merging for the example in Figure 4.7.

Since each reuse chain must be generated from either an input or a PE (Section 3.1.2), we can always merge the reuse chain together, as shown in Figure 4.8. This is the finest-grained merging strategy based on *logical hierarchy* (the logical dependency and relationship among modules).

4.3.3 Coarse-Grained Module Merging

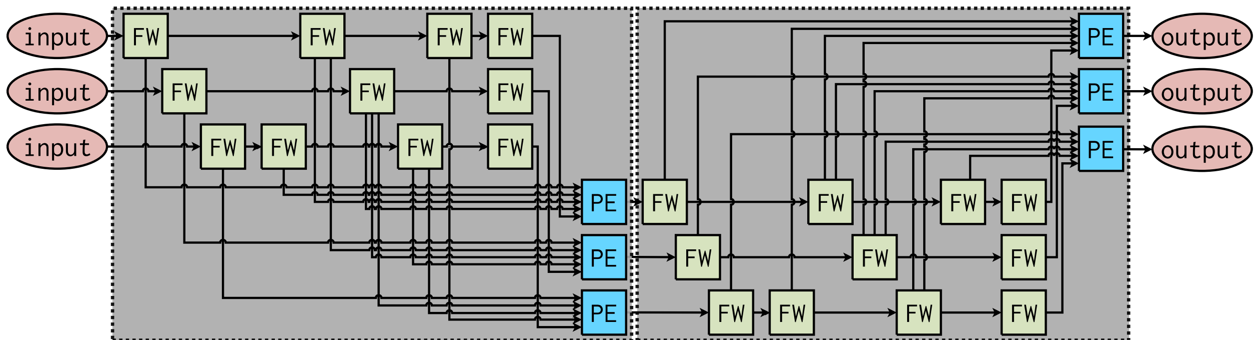


Figure 4.9: Coarse-grained module merging for the example in Figure 4.7.

A coarser-grained merging strategy is to merge everything in each iteration/stage together. This is the same as full-scale merging if there is only one iteration/stage. Figure 4.9 shows the coarse-grained module merging strategy for the example in Figure 4.7.

4.3.4 Full-Scale Module Merging

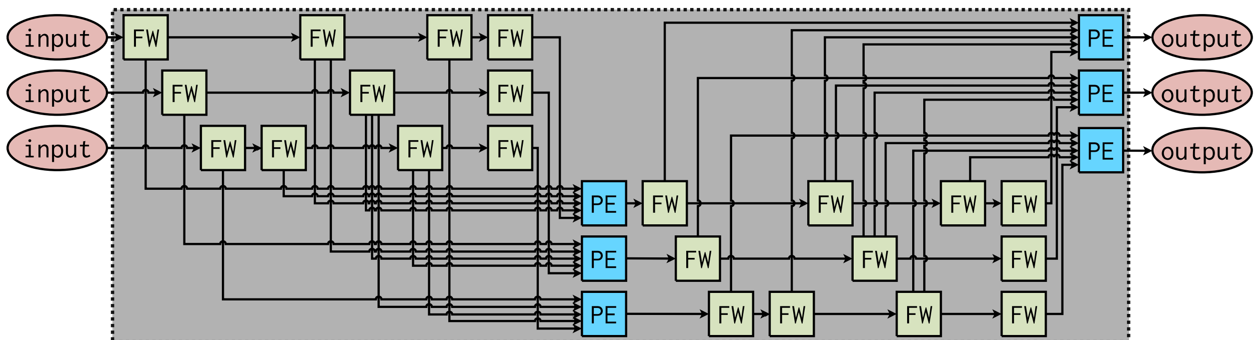


Figure 4.10: A SODA kernel with 3 PEs per iteration and 2 iterations in total.

The full-scale merging strategy aggressively merges all modules together into a single, gigan-

tic module, which removes any inter-module communication channels and minimizes resource utilization. This is the coarsest-grained merging strategy based on logical hierarchy. Figure 4.10 shows the full-scale module merging strategy for the example in Figure 4.7.

4.3.5 Module Merging Based on Physical Hierarchy

The three merging strategies introduced in the previous sections are all based on the *logical* hierarchy of the design. They leverage the logical dependency and relationship of different modules and components in the design, e.g., a reuse chain is always produced by an input or a PE, and all modules in the same iteration/stage collaboratively produce the same output array. Such logical hierarchy may not map well to the two-dimensional layout of modern large FPGAs, since it is usually conceived with little or no consideration of the hardware layout information [32]. An alternative is to merge modules based on *physical* hierarchy that is most suitable for being embedded on a two-dimensional silicon surface [31], e.g., to merge modules so that each merged module fits a physical coarse-grained floorplanning partition while minimizing the number of wires connecting the modules.

To merge modules based on physical hierarchy, we leverage AutoBridge [76]. AutoBridge is capable of coarse-grained floorplanning at the HLS level for dataflow programs. It takes as input a dataflow graph representation of an HLS program, takes into account the physical layout of the underlying device, and tries to produce an optimal floorplan that minimizes the total number of wires between different partitions (pblocks). While AutoBridge can be applied directly on SODA programs without module merging, merging modules in each partition can further reduce slice utilization and potentially improve routability. In case the kernel is small enough and AutoBridge decides to put everything in a single partition, module merging based on physical hierarchy will produce the same result as full-scale merging.

Table 4.6: Comparison among different module merging strategies. **Bold** items indicate the best (lowest LUT usage or highest clock frequency) merging strategy of that benchmark.

Benchmark	LUT Usage/%					Clock Frequency/MHz				
	None	Fine	Coarse	Full	Physical	None	Fine	Coarse	Full	Physical
s2d5pt	58.1	50.9	48.8	Timeout	**43.8**	261	243	245	Timeout	**299**
s2d33pt	52.8	45.4	Failed	Failed	**37.8**	276	273	Failed	Failed	**300**
f2d9pt	52.5	48.7	46.4	Timeout	**43.1**	278	270	290	Timeout	**300**
f2d81pt	49.5	44.6	Failed	Failed	**39.9**	248	260	Failed	Failed	**275**
s3d7pt	47.2	40.0	**37.9**	Timeout	38.1	271	289	**300**	Timeout	264
s3d25pt	49.9	46.5	Failed	**37.6**	Failed	**300**	231	Failed	112	Failed
f3d27pt	52.9	46.6	45.8	Timeout	**41.6**	165	**288**	224	Timeout	282
f3d125pt	39.9	37.2	35.2	Failed	**34.5**	280	274	224	Failed	**294**

4.3.6 Experimental Results

Table 4.6 shows the post-implementation results comparing the five different module merging strategies using the iterative benchmarks in Table 4.4. The target FPGA device is Alveo U250, which has 4 DDR channels. We set the timeout for each design to 72 hours. As demonstrated in Table 4.6, merging smaller modules can clearly reduce resource utilization. The merging strategy based on physical hierarchy proves to be the most resource-efficient in all but one cases because its merging granularity matches both the logical hierarchy of the benchmark kernel and the physical hierarchy of the device. Merging everything into a single module failed to complete in 72 hours for all benchmarks, with only one exception that was under-clocked at 112 MHz, which is the lowest among all successful designs in Table 4.6. This is likely due to the high fan-out of the control signals in the singleton module [77]. The clock frequency does not show a clear trend as in the LUT usage because timing closure can be impacted by many factors. Nevertheless, the merging strategy based on physical hierarchy is a good choice in most cases due to its awareness of the physical hierarchy of the device.

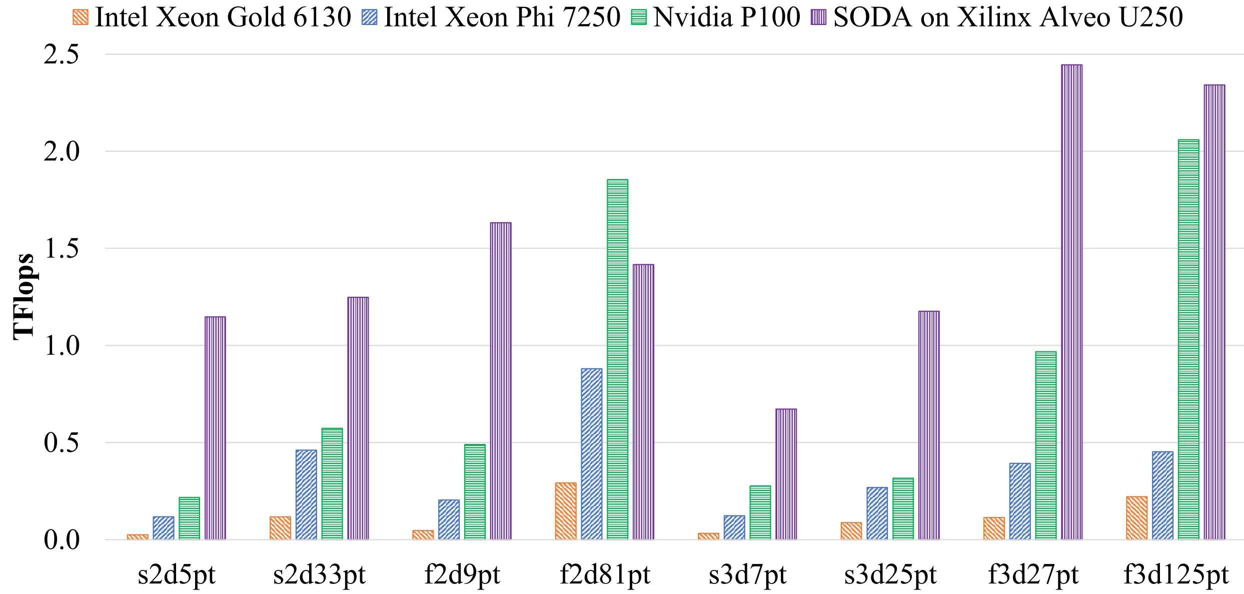


Figure 4.11: Performance of iterative kernels with frequency improved by applying module merging and coarse-grained floorplanning (**bold** items in Table 4.6).

Figure 4.11 compares the performance of SODA with module merging and coarse-grained floorplanning applied on top of parallelization, communication reuse, and computation reuse. The best merging strategy found in Table 4.6 is used. Although the merging strategy based on physical hierarchy is the most likely strategy to produce the best result, it failed to route in one of the benchmarks and did not achieve the highest frequency in two other cases. Note that although the same benchmarks can run on an HBM-equipped FPGA, the DDR-based U250 FPGA has more computational resources and thus can help to achieve higher performance for the iterative benchmarks. Compared with a state-of-the-art software system [179] which also considers parallelization, communication reuse, and computation reuse, the average speedups of SODA on Alveo U250 over multicore Xeon Gold CPU, many-core Xeon Phi accelerator, and P100 GPU are 13.0×, 4.0×, and 1.9×, respectively.

4.4 Summary

In this chapter, we have demonstrated model-driven design-space exploration for both communication reuse and computation reuse. Under the same model, we further explored different implementation granularities by merging the fine-grained modules into larger ones that leads to fewer resource usage. With the latest-generation FPGA board, SODA can outperform multicore CPU, many-core accelerator, and high-end GPU by 13.0×, 4.0×, and 1.9×, respectively. With extensive design-space exploration, we shall raise the abstraction level and present an end-to-end compilation flow from a high-level domain-specific language to efficient FPGA accelerators in the next chapter.

CHAPTER 5

Enabling High-Level DSLs for Stencil Applications

Image processing plays a significant role in lots of applications today, including medical imaging [75], autonomous driving [98], augmented reality [162], and computational photography [121]. Lying in the core of image processing pipelines are local sliding windows [9], i.e., stencil kernels. However, due to large number of processing stages and the complex data dependency, image processing pipelines are often memory-bound and implementing them efficiently is not an easy job. It is difficult and time-consuming for a designer to write image processing algorithms while parallelizing and optimizing for data locality and performance. Halide [146], a widely-used image processing domain-specific language (DSL), partially solves this problem by decoupling the algorithm and scheduling to allow programmers to search for optimized mappings of the resulting pipelines to various parallel architectures and complex memory hierarchies.

The theoretical analysis presented in Chapter 3 and the model-driven DSE presented in Chapter 4 have demonstrated the advantages of FPGA accelerators over CPU and GPU for stencil kernels like image processing pipelines. However, while the Halide community has been growing rapidly in recent years (received over 3,000 stars on GitHub [59]), there is no way to easily migrate the vast number of Halide programs to FPGA accelerators. The direct and traditional way to design FPGA accelerators is to rewrite programs to register-transfer level (RTL) code. This is very time-consuming. Although C-based high-level synthesis (HLS) raises the design abstraction level to untimed specification by automated scheduling, pipelines, and resource sharing [37], it still requires expertise on the microarchitecture to get efficient designs that result in a high threshold for software programmers. The complicated rules of using scheduling primitives for HLS bring

programmers a greater workload as well.

Another approach is to rewrite Halide programs to hardware-oriented DSLs, such as Darkroom [82], Hipacc [147], PolyMage [28], SODA [20], and HeteroCL [110]. But it still takes time to learn these DSLs, not to mention other limitations that exist in these DSLs. For example, Darkroom supports only 1 pixel/cycle pipelines, which may not be acceptable for many image applications. HeteroCL is a promising heterogeneous programming language inspired by Halide, but it takes time for Halide programmers to learn. Some conventions and behaviors of HeteroCL and Halide are not the same. This may cause confusion to programmers who try to manually migrate from Halide to HeteroCL.

The only prior work from Halide to FPGA is Halide-HLS [144]. Halide-HLS is a Halide-to-FPGA compiler and allows programmers to design FPGA accelerators without many modifications. It provides a simple way for Halide programmers to implement their programs on a Xilinx Zynq FPGA. However, due to the lack of active maintenance, the HLS code it generates is no longer supported by the latest FPGA vendor tools, which means no one can actually make use of it now. Even if someone is willing to make it up-to-date, a significant amount of engineering effort would be required because their code generator and the generated architecture is tightly and directly coupled with the backend HLS tool. Nevertheless, this work is an important motivating factor of this study.

In this chapter, we present HeteroHalide, an end-to-end system for compiling Halide programs to FPGA accelerators, making use of both algorithm and scheduling information specified in a Halide program. HeteroHalide not only significantly simplifies the migration effort, but also enables efficient accelerator designs via its flexible backend choices. For stencil applications which is common in image processing pipelines, in addition to parallelization, computation reuse, and communication reuse that SODA can explore, HeteroHalide further enables algorithm-level exploration. More concretely, our contributions are as follows:

- HeteroHalide provides an easy-to-use flow from Halide to FPGA. It only requires moderate

modifications for Halide programs on the scheduling part, instead of algorithm. Compared to other existing approaches, including rewriting in HLS/RTL, this solution of migrating Halide greatly reduces the migration effort.

- HeteroHalide generated accelerators outperform both CPU with 28 cores and the Halide-HLS FPGA compiler. Accelerators generated by HeteroHalide achieve 4.15× speedup on average over CPU, and has 2 ~ 4× peak performance as Halide-HLS, when tested on the same Zynq-7020 board.

- We develop a Halide-to-HeteroCL code generator, which can automatically generate HeteroCL [110] code, with both algorithms and schedules. We choose HeteroCL as the intermediate representation (IR) in HeteroHalide, because of its great hardware customization capability, using the idea of separating algorithms and schedules. HeteroCL supports multiple heterogeneous backends (spatial architectures), including a stencil backend [20], a systolic array backend [38], and the general Merlin compiler backend [33]. Therefore, it is able to generate efficient hardware code according to the type of the applications.

- When Halide is compiled, the scheduling is applied directly at IR level using *immediate transformation*. We make extensions on Halide schedules, allowing some schedules to be lowered with annotations, using *lazy transformation*. By adding this extension to Halide, HeteroHalide can generate specific scheduling primitives at the HeteroCL backend level, thus emitting more efficient accelerators.

The remainder of this chapter is organized as follows. In Section 5.1, we briefly introduce the compilation flow used by Halide. Section 5.2 describes our extensions to the Halide language, which allows the scheduling primitives to be mapped to FPGA. Section 5.3 evaluates the productivity and quality of result.

5.1 Halide Compilation Flow

As discussed in Section 2.3.1, Halide is a domain-specific language that separates schedule from algorithm. Programmers can change the implementation (i.e., *schedule*) of the same algorithm by changing the scheduling primitives, which include loop transformations like `split`, `fuse`, `reorder`, `tile`, etc., and parallelization primitives like `unroll`, `parallel`, `vectorize`, etc. This allows programmers to explore the design space and search for the optimal schedule for different target devices efficiently.

When compiling Halide source code, the Halide compiler analyzes the syntax and semantics of the algorithm part of Halide program and transforms them into an abstract syntax tree (AST). Every node in the AST represents an operation to the variables, such as `Add` and `Store`. In the process of analyzing syntax, an AST node may point to other nodes. For example, a `For` loop node points to a `Stmt` node as its loop body and two `Expr` nodes that store the start and trip count of the loop. Metadata about the loop, such as parallelization type, are also stored in the `For` node. As such, Halide builds the connections between operations and variables by constructing this AST. The AST is used as the IR in Halide. Once the Halide IR is lowered from source code, the Halide compiler will then apply optimization passes according to the scheduling primitives, adding, removing, or modifying IR nodes, and generate the final, optimized Halide IR. Finally, the target's code generator will emit code for the target architecture from the Halide IR.

5.2 HeteroHalide Compilation Flow

HeteroHalide provides an automated flow from Halide to FPGA accelerators. It consists of a Halide-to-HeteroCL code generator, HeteroCL, and multiple backends of HeteroCL. More specifically, HeteroCL generates corresponding hardware domain-specific languages (DSL) to these heterogeneous backends, and then those backends generate hardware code from their hardware DSLs.

Both Halide [146] and HeteroCL [110] separate algorithms and schedules in the code. However, there are still some semantic gaps between them in the scheduling part while the algorithm part is basically consistent during code generation. When a Halide program is compiled, the scheduling primitives are applied at the IR level and are hard to recover, but HeteroCL often needs explicit scheduling information to generate efficient accelerators. To tackle this challenge, we propose two schedule lowering methods and extend Halide by changing the scheduling primitives accordingly. The rest of this section introduces the process of code generation for algorithms and schedules, with examples, and our extensions on schedule transformation.

5.2.1 Algorithm Transformation

This step is straightforward. We use the blur filter as an example to show the compilation from Halide algorithm code to HeteroCL algorithm code. The blur filter consists of two stages. Each stage is described by a Halide function and represents a 3×1 (or 1×3) filter, as shown in Listing 5.1.

```
1 Func blur_x("blur_x");
2 blur_x(x, y) = (input(x, y) + input(x+1, y) + input(x+2, y))/3;
3 Func blur_y("blur_y");
4 blur_y(x, y) = (blur_x(x, y) + blur_x(x, y+1) + blur_x(x, y+2))/3;
```

Listing 5.1: Halide algorithm.

The corresponding HeteroCL algorithm code is shown in Listing 5.2. The generated HeteroCL code uses a top function to describe the overall algorithm. The substage and the following for loop correspond to each `Func` in Halide. Other than the syntax and convention differences, we can see that Halide and HeteroCL share very similar code structures for the algorithms. This makes the transformation relatively simple. We use HeteroCL's imperative programming APIs (`heterocl.Stage`, `heterocl.for_`) instead of the compute API (`heterocl.compute`). In this way, we explicitly represent the computing for loop in the HeteroCL source code that is close to both Halide IR and HeteroCL IR. This improves the scalability and stability of the Halide-to-

```
def top(input_hcl):
  with heterocl.Stage("blur_x"):
    with heterocl.for_(y_min, y_max) as y:
      with heterocl.for_(x_min, x_max) as x:
        tensor_blur_x[x, y] = (input_hcl[x, y] + input_hcl[x + 1, y] +
                               input_hcl[x + 2, y]) / 3

  with heterocl.Stage("blur_y"):
    with heterocl.for_(y_min, y_max) as y:
      with heterocl.for_(x_min, x_max) as x:
        tensor_blur_y[x, y] = (tensor_blur_x[x, y] + tensor_blur_x[x, y + 1] +
                               tensor_blur_x[x, y + 2]) / 3

  return tensor_blur_y
```

Listing 5.2: HeteroCL algorithm.

HeteroCL code generator.

5.2.2 Schedule Transformation

Unlike the algorithm part, effort is needed for code generation for the scheduling part. While Halide implements the schedules directly at the IR level, HeteroCL needs explicit scheduling (customization) information in order to generate efficient FPGA accelerators. Therefore, we propose two methods for schedule lowering: immediate transformation (Section 5.2.2.1), and lazy transformation (Section 5.2.2.2). These two different methods are further explained with examples as follows.

5.2.2.1 Immediate Transformation

Halide schedules are directly implemented at the IR level. As an example, we apply the Halide `unroll` schedule to the blur filter shown in Listing 5.1 to demonstrate the process of immediate transformation. Line 2 in Listing 5.1 represents a computation stage. Without any customized

schedule, its default loop nest is shown in Listing 5.3.

```
1 for y [min=...; extent=...; stride=1]:
2   for x [min=...; extent=...; stride=1]:
3     blur_x(y, x) = ...
```

Listing 5.3: Halide IR of the first loop nest in the blur filter without schedules.

We then apply the Halide schedule `blur_x.unroll(x,4)` to unroll the `x` loop with a factor of 4. It is directly implemented into the Halide IR, and the loop nest is transformed to Listing 5.4.

```
1 for y [min=...; extent=...; stride=1]:
2   for x [min=...; extent=...; stride=4]:
3     blur_x(y, x) = ...
4     blur_x(y, x + 1) = ...
5     blur_x(y, x + 2) = ...
6     blur_x(y, x + 3) = ...
```

Listing 5.4: Halide IR of the first loop nest in the blur filter with `blur_x.unroll(x,4)` applied using immediate transformation.

Via immediate transformation, the schedule is directly implemented into the Halide IR, and the explicit scheduling information is lost at the IR level in the process of lowering.

5.2.2.2 Lazy Transformation

With lazy transformation, Halide schedules are stored explicitly at the IR level as an annotation. The scheduling annotation is further transferred to the subsequent steps of the flow and implemented by the backends. To have a clear comparison with *immediate transformation*, we apply the same unrolling schedule to the blur filter again, but this time we use `lazy_unroll`, which is added as an extension to the existing Halide schedules. With `blur_x.lazy_unroll(x,4)` applied, the corresponding loop nest is shown in Listing 5.5.

```
1  for y [min=...; extent=...; stride=1]:
2    for x [min=...; extent=...; stride=1; unrolled; factor=4]:
3      blur_x(y, x) = ...
```

Listing 5.5: Halide IR of the first loop nest in the blur filter with `blur_x.lazy_unroll(x,4)` applied using lazy transformation.

```
1  schedule = heterocl.create_schedule([input_hcl], top)
2  stage_blur_x = top.blur_x
3  schedule[stage_blur_x].unroll(stage_blur_x.axis[1], 4)
```

Listing 5.6: HeteroCL scheduling code of the first loop nest in the blur filter with `blur_x.lazy_unroll(x, 4)` applied.

In Line 2 of Listing 5.5, the `unrolled` annotation and the corresponding unroll factor are stored in the `For` IR node corresponding to the `x` loop. Thus, explicit scheduling information is maintained at the IR level and can be implemented in subsequent steps of the flow. This is necessary because the HeteroCL backends sometimes need to apply their unique primitives to direct HLS schedules, using the information in the annotations.

These unique scheduling primitives for HeteroCL backends are essential for emitting efficient FPGA code. As HeteroCL supports all those backends, the best way to support the scheduling transformation from Halide to heterogeneous hardware DSLs is to fully utilize HeteroCL and to generate explicit scheduling code for HeteroCL. We keep using the `lazy_unroll` schedule as an example and demonstrate the subsequent compilation flow for this schedule.

Listing 5.6 shows the corresponding HeteroCL schedule. First, a HeteroCL API is called to create a default schedule based on algorithm code of HeteroCL (Line 1). The function defining the algorithm (`top`) and its input(s) (`[input_hcl]`) are passed to the `heterocl.create_schedule` API. Then, in Line 2, our target stage is identified with the algorithm `top` and the stage's name. Line 3 applies the `unroll` scheduling primitive to `stage_blur_x`. `axis[1]` corresponds to loop

```
for (int y = ...; y < ...; y++)
#pragma ACCEL parallel factor = 4 flatten
  for (int x = ...; x < ...; x++)
    blur_x[y][x] = ...
```

Listing 5.7: Merlin C code generated from the HeteroCL code.

x, and the unroll factor is 4. The HeteroCL scheduling code in Listing 5.6 is then transformed to different backend scheduling codes using different HeteroCL backend code generators.

Here, we show two HeteroCL backend schedules as examples. Listing 5.7 shows the loop nest generated by the Merlin C backend and its scheduling primitives. Merlin C is an OpenMP-like programming model used by the Merlin compiler [33] from Falcon Computing Solutions (now acquired by Xilinx). Similar to the Halide IR loop nest with lazy transformation in Listing 5.5, the explicit scheduling information is stored as an annotation (Line 2 in Listing 5.7 and Line 2 in Listing 5.5). Another example is the SODA DSL presented in Section 4.1.1, which can be synthesized into efficient accelerators with scalable parallelism, optimal communication reuse and model-driven computation reuse. A SODA unrolling primitive looks like "`unroll factor: 4`", which directs the SODA compiler to unroll the inner loop of every stage with the same unroll factor.

5.2.3 Extensions on Halide Schedules

In this section, we summarize our extensions on Halide schedules and the design methodology of schedule transformation for different Halide schedules.

To obtain efficient FPGA accelerators, we need to generate scheduling primitives (e.g., Line 2 in Listing 5.7) in the process of compilation. As HeteroCL [110] is a heterogeneous programming platform, maintaining schedule explicitly in the process of Halide-to-HeteroCL is essential. Therefore, we change the lowering method for some Halide schedules (e.g., `lazy_unroll` introduced in Section 5.2.2.2) as extensions. Similar schedule extensions are used for other backend

Table 5.1: Schedule primitives and the corresponding transformation methods supported by Halide vs HeteroHalide.

Primitive	Description	Halide	HeteroHalide
`reorder`	Switch the order of sub-loops in the same nested loop.	Immediate	Immediate
`split`	Split a loop into a two-level nested loop given the extent of the inner loop.	Immediate	Immediate
`fuse`	Fuse a two-level nested loop into a single-level loop.	Immediate	Immediate
`tile`	Split an iteration domain into smaller tiles and iterate over each tile separately.	Immediate	Immediate
`unroll`	Unroll a loop with given factor.	Immediate	Lazy
`parallel`	Schedule a loop in parallel.	Immediate	Lazy

targets as well (e.g., `gpu_tile`).

Table 5.1 lists the schedules supported by Halide and HeteroHalide, and the corresponding schedule lowering methods. By default, Halide uses immediate transformation to implement the schedule directly into the IR. Loop transformation schedule primitives, e.g., `reorder`, `split`, do not have special semantics in HeteroCL, and, therefore, there is no need to create new scheduling primitives for them. However, for the parallelization scheduling primitives, e.g., `unroll`, `parallel`, the explicit scheduling information is required in HeteroCL to generate efficient backend code. For example, if the unrolling schedule is applied immediately and the Halide IR is in the form of Listing 5.4, HeteroCL will not be able to generate SODA DSL and leverage its highly-efficient spatial architecture. Therefore, we create lazily-applied Halide schedules `lazy_unroll` and `lazy_parallel` to transfer scheduling information explicitly to generate efficient FPGA accelerators. Note that not all scheduling primitives are applicable to both Halide and HeteroHalide. For example, `vectorize` is only applicable to Halide, whereas pipelining is implicitly inferred in

HeteroHalide (and thus no explicit scheduling primitive is required nor provided).

5.3 Evaluation

We now present our experiments, followed by the evaluation on two parts: 1) *programming efficiency*, where we show the simplified migration effort from Halide to FPGA accelerators via the lines of code (LoC) comparison, and 2) *accelerator performance*, comparing the throughput of the FPGA code generated by HeteroHalide and the throughput of 28 CPU cores with several real-world applications. This section also compares the peak performance between the accelerators generated by HeteroHalide and those reported by Halide-HLS [144]. The applications we use in the evaluation and the corresponding description are listed in Table 5.2.

Table 5.2: Applications used in the evaluation.

Application	Description
Harris	Harris corner detector.
Gaussian	3 × 3 Gaussian filter.
Unsharp	Unsharp masking filter.
Blur	Average over 3 × 3 window.
Linear Blur	Blur with two linear transformations.
Stencil Chain	3 × 3 kernel chained 3 times.
Dilation	Maximum over 3 × 3 window.
Erosion	Minimum over 3 × 3 window.
Median Blur	Median over 3 × 3 window.
Sobel	Sobel edge detector.
GEMM	General matrix multiplication.
K-Means	K-means clustering.

5.3.1 Programming Efficiency

In this section, we compare the lines of code (LoC) of the same program at different levels in the flow to demonstrate the different workload required when compiling Halide to FPGA accelerators.

Table 5.3: LoC at different levels in the flow. HeteroHalide and HeteroCL counts are algorithm + schedule. Numbers in parentheses are ratios over HeteroHalide.

Application	HeteroHalide	HeteroCL Generated	HLS Generated
Harris	26 + 14	72 + 22 (2.4×)	14224 (355.6×)
Gaussian	8 + 3	23 + 8 (2.8×)	17181 (1561.9×)
Unsharp	13 + 5	46 + 12 (3.2×)	21383 (1187.9×)
Blur	2 + 4	9 + 4 (2.2×)	1455 (242.5×)
Linear Blur	11 + 10	22 + 10 (1.5×)	1072 (51.0×)
Stencil Chain	15 + 10	14 + 8 (0.9×)	9061 (362.4×)
Geo. Mean	—	(2.0×)	(378.6×)

The compilation flow from HeteroHalide consists of the following steps: Halide to HeteroCL, HeteroCL to HLS code, and finally to an FPGA accelerator. It is possible to obtain the same FPGA accelerator by manually writing the corresponding code at any level in the flow, but the required programming effort is significantly different. Table 5.3 shows the LoC comparison among the code at different levels. For most applications, Halide code is more compact than HeteroCL code. Both of them are orders of magnitude more compact than our generated HLS code. The partial reason of this significant difference is HLS code generated by a compiler is redundant compared to optimized HLS code.

Table 5.4 summarizes the LoC comparison between the Halide code and the Xilinx xfOpenCV library [170]. The kernels in the xfOpenCV library are optimized for Xilinx FPGAs and SoCs, based on the OpenCV computer vision library. Compared with the HLS code optimized by experts, Halide code is still more concise and compact. In summary, for both approaches without

Table 5.4: LoC comparison between HeteroHalide and Xilinx xfOpenCV Library [170]. HeteroHalide counts are algorithm + schedule. xfOpenCV counts include only the core functions, not the utility libraries. Numbers in parentheses are ratios over HeteroHalide.

Application	HeteroHalide	xfOpenCV
Harris	26 + 14	117 (2.9×)
Gaussian	8 + 3	104 (9.5×)
Dilation	2 + 1	80 (26.7×)
Erosion	2 + 1	79 (26.3×)
Median Blur	2 + 1	81 (27.0×)
Sobel	3 + 2	208 (41.6×)
Geo. Mean	—	(16.7×)

HeteroHalide, i.e., rewriting Halide in HeteroCL and HLS respectively, the workload for programmers increases, not to mention the additional knowledge required. HeteroHalide greatly simplifies the process of migrating Halide to FPGA accelerators.

5.3.2 Accelerator Performance

In this section, we first evaluate the accelerators generated by HeteroHalide. The experiments for CPU are performed on an Ubuntu 16.04 server with two Intel Xeon 2680v4 CPU (28 cores in total) and 64 GiB DDR4 memory. Our target FPGA is the state-of-the-art Xilinx VU9P FPGA, whose default target frequency is 250 MHz.

Table 5.5 shows the applications and overall evaluation results of each application with certain data sizes and type, including the speedup over CPU, the energy efficiency gain, the accelerator's maximum throughput for stencil applications, and the resource utilization given by the post-synthesis report. Since Halide originated as an image processing DSL and stencil kernels are extensively used, we mainly focus on those in the experiments. To demonstrate the capability of

Table 5.5: Accelerators generated by HeteroHalide compared with plain Halide on 28 CPU cores.

Benchmark	Data Size & Type	#LUT	#FF	#DSP	#BRAM	Throughput	Efficiency	Speedup	Backend
Harris	2448 × 3264, UInt8	55198	64427	264	80	32 pixel/cycle	29.11	10.31	Stencil
Gaussian	2160 × 3840, UInt8	67298	41496	768	0	32 pixel/cycle	17.17	6.08	Stencil
Unsharp	2448 × 3264 × 3, UInt8	47683	33114	400	24	32 pixel/cycle	9.57	3.39	Stencil
Blur	648 × 482, UInt16	6821	8209	32	0	16 pixel/cycle	10.98	3.89	Stencil
Linear Blur	768 × 1280 × 3, Float32	31049	39369	536	16	8 pixel/cycle	12.65	4.48	Stencil
Stencil Chain	1536 × 2560, UInt16	61230	46174	48	192	16 pixel/cycle	4.29	1.52	Stencil
Dilation	6480 × 4820, UInt16	13046	12114	0	64	32 pixel/cycle	4.69	1.66	Stencil
Median Blur	6480 × 4820, UInt16	14388	10066	0	64	32 pixel/cycle	12.51	4.43	Stencil
GEMM	1024 × 1024 × 1024, Int16	454492	800283	2507	932	—	9.97	3.53	Systolic Array
K-Means	320×32, k=16, Int32	212708	235011	1536	32	—	29.00	10.27	General
Geo. Mean	—	—	—	—	—	—	11.71	4.15	—

using multiple backends via leveraging HeteroCL [110], two other applications that are not in the image processing domain are included as well. The energy efficiency gain is the accelerator-to-CPU ratio and is calculated based on the thermal design power. The throughput of accelerators is memory-bounded and is calculated based on the total bandwidth and the data type width of the application. For the CPU execution of the Halide [146] programs, we use the same scheduling strategies as the examples provided in the open-source Halide repository [59]. The specific parameters such as the unroll factor and tiling size are manually fine-tuned in our testing environment. We leverage the auto-scheduling feature of Halide [132] to compare our manual fine-tuned schedules with the optimal strategies generated by Halide auto-scheduler on CPU. The geometric mean of CPU speed ratio between manual and auto scheduling tested on several benchmarks is 1.15, which shows that our manual optimizations are on par with the highly optimized schedules after extensive design-space exploration.

Note that the CPU code generated by Halide is capable of utilizing all 28 cores available on the server. The experimental results show that the accelerators generated by HeteroHalide achieves 4.15× average speedup and 11.71× energy efficiency gain over CPU.

Table 5.6 lists the peak performance of the accelerators generated by HeteroHalide and Halide-

Table 5.6: Performance comparison between HeteroHalide and Halide-HLS.

Benchmark	Data Sizes & Type	Halide-HLS	HeteroHalide	Speedup
Harris	640 × 640, UInt8	2 pixel/cycle	4 pixel/cycle	2
Gaussian	640 × 640, UInt8	2 pixel/cycle	8 pixel/cycle	4
Unsharp	640 × 640 × 3, UInt8	1 pixel/cycle	4 pixel/cycle	4
Geo. Mean	—	—	—	3.175

HLS [144] for three applications. Since we were unable to reproduce the results using the current synthesis tools, for Halide-HLS we use the reported numbers in their paper [144]. The throughput of HeteroHalide is obtained using the same Zynq 7020 device as Halide-HLS. The results show that for various applications, HeteroHalide achieves a 2~4× speedup over Halide-HLS. Clearly, HeteroHalide is more efficient to migrate Halide programs to FPGA accelerators. With HeteroHalide and its SODA backend (presented in Chapter 3 and Chapter 4), image processing experts now have an end-to-end solution to create efficient FPGA accelerators capable of communication reuse and computation reuse from a high-level DSL that they are familiar with, significantly improving productivity.

CHAPTER 6

Extending High-Level Synthesis for Task-Parallel Programs

Using stencil applications as a representative example, Chapter 3, Chapter 4, and Chapter 5 has demonstrated how theoretical analysis, model-driven exploration, and high-level languages can benefit accelerator design and optimization for memory-bound applications with regular memory accesses. In this chapter, we will shift our focus to applications with irregular memory accesses. Such applications often do not have a high-degree of data-parallelism, and are often implemented as task-parallel programs. However, not all programs are created equal for HLS. Data-parallel programs can be easily programmed following the sequential C semantics with HLS-specific compiler directives (i.e., "pragma"). The HLS compiler can then leverage the directives to extract the parallelism automatically via static dependency analysis. This enables such applications to be quickly designed and iterated in the fast correctness verification cycle and QoR tuning cycle (Figure 2.2b). However, task-parallel programs are not supported by the native C semantics, and the productivity provided by current HLS tools are greatly limited due to poor programmability, restricted software simulation, and slow code generation. Limited productivity for task-parallel programs significantly elongates the development cycles and undermines the benefits brought by HLS. One may argue that programmers should always go for data-parallel implementations when designing FPGA accelerators using HLS, but data-parallelism may be inherently limited, for example, in applications involving graphs. Moreover, researches show that even for data-parallel applications like neural networks [38] and stencil computation [20], task-parallel implementations show better scalability and higher frequency than their data-parallel counterparts due to the localized communication pattern [40].

In this chapter, we extend the HLS C++ language and present our framework, TAPA (**ta**sk-**pa**rallel)[1], as a solution to the limitations of HLS productivity. Our contributions include:

- **Convenient programming interfaces**: We show that, with peeking and transactions added to the programming interfaces, TAPA can be used to program task-parallel kernels with 22% reduction in lines of code (LoC) on average. By unifying the interface used for the kernel and host, TAPA further reduces the LoC on the host side by 51% on average.
- **Universal software simulation**: We demonstrate that our proposed simulator can correctly simulate task-parallel programs that existing simulators fail to simulate. Moreover, the correctness verification cycle can be accelerated by a factor of 3.2× on average.
- **Hierarchical code generation**: We show that by modularizing a task-parallel program and using a hierarchical approach, RTL code generation can be accelerated by a factor of 6.8× on our server with 32 hyper-threads.

The remainder of this section is organized as follows. A PageRank [139] accelerator motivates our work in Section 6.1. Section 6.2 introduces the details of the TAPA framework. Section 6.3 presents the productivity evaluation.

6.1 Motivating Example

Graph is an important data structure that is critical in many data mining and machine-learning algorithms [57, 101, 117, 126, 139]. While there are many existing FPGA accelerators designed for graph algorithms [47, 48, 99, 138, 175, 177, 181], none of them are programmed in HLS. HLS's lack of good productivity for task-parallel programs is one of the reasons it is not adopted for graph algorithms. In this section, we use a real-world design to illustrate the productivity issues for implementing graph accelerators in HLS, which serve as a motivating example for our work.

---

[1]While a prior work TAPAS [123] and our work TAPA share similarity in name, our work focuses on statically mapping tasks to hardware, yet TAPAS specializes in dynamically scheduling tasks.

Our example accelerator implements PageRank [139] on the Alveo U280 board and leverages the high-bandwidth memories (HBM). The input graph is pre-processed and loaded into the HBM on the FPGA. The accelerator adopts an edge-centric graph programming model [149] and decouples the computation into two phases, i.e., the scatter phase and the gather phase [21, 181]. In the scatter phase, edges are streamed from the HBM to the processing elements (PE) on FPGA. For each edge, an update message is generated to propagate the weighted ranking of the source vertex to the destination vertex. The updates are collected and stored off-chip in the HBM. In the gather phase, the updates are loaded from the HBM and the rankings are accumulated over each vertex. Our PageRank accelerator instantiates multiple PEs. The PEs are connected to a vertex handler and a control module. The control module coordinates accesses to the vertex attributes and iterative execution between the two phases. Figure 6.1 shows the block diagram of the example accelerator.

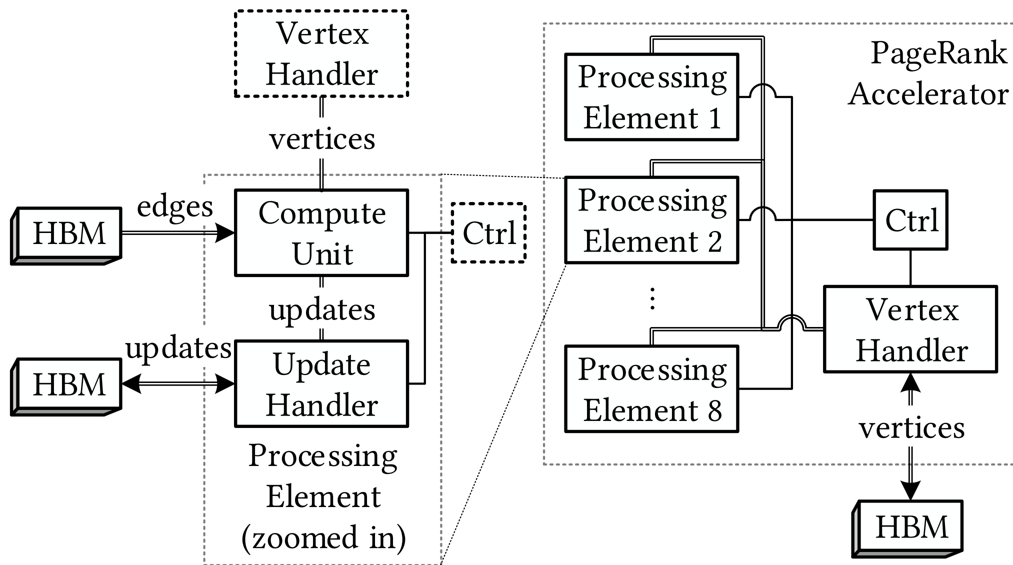


Figure 6.1: Example PageRank accelerator design.

We measured 4.4 GTEPS[2] on-board execution throughput using the accelerator with 19 HBM channels in use. As a comparison, multi-thread CPU performance is around 0.7 GTEPS with 4 DDR4 memory channels [21]. Even if we assume a similar memory bandwidth as the FPGA accelerator and project the CPU performance to 3.5 GTEPS, it would still be more than 20% slower, due to the lack of fine-grained control over communication. While developing this accelerator, we found that the following are missing or hard-to-use in the HLS tools and significantly impact the productivity.

1) *Peeking*. Peeking is defined as reading a token from a channel without consuming it. As mentioned in Section 2.4.4.1, KPN explicitly prohibits such behavior. Yet, such a pattern is common in many applications. For example, in the PageRank accelerator, the `UpdateHandler` module needs to keep track of the number of updates destinated to each vertex partition. Due to the large number of partitions, block RAMs (BRAM) are used for storing the update counts. However, incrementing a value in BRAM cannot be done in a single clock cycle on FPGAs due to the addressing latency, which prevents the loop from being fully pipelined. A workaround is to accumulate the update count in a register for updates with the same partition ID (`pid`) and only write changes to BRAM when the `pid` changes. This requires us to detect conflicts on the addresses and stop reading the input channel when conflict occurs, as shown in the green lines marked with "+" in Listing 6.1. Without a `peek` API, one has to write it as the red lines marked with "-" in Listing 6.1 to manually maintain a buffer for the incoming values. This not only increases the programming burden, but also makes the design prone to errors in state transitions of the buffer.

2) *Transactions*. A sequence of tokens may constitute a single logical communication transaction. Using the same PageRank accelerator example, in the gather phase when the updates are read from HBM, the updates transmitted from `UpdateHandler` to `ComputeUnit` for each vertex partition can be considered a single transaction. Since only `UpdateHandler` knows the number of updates transmitted in each transaction, `ComputeUnit` needs to test for a special token to detect the *end of transaction* (green lines marked with "+" in Listing 6.2). Without an `eot` API, one has

[2] Giga traversed edges per second.

```
 1   1   int counts[kPartitionCount] = {};
 2   2   int count, last_pid = -1, last_last_pid = -1;
 3    -  UpdateWithPid buf;
 4    -  bool buf_valid = false;
 5   3   while (...) {
 6    -    if (buf_valid) {
 7    -      Pid pid = buf.pid;
     4+    Pid pid = input_chan.peek().pid;
 8   5     if (pid != last_pid && pid == last_last_pid) {
 9   6       last_last_pid = -1;  // BRAM conflict on counts
10   7     } else {
11    -        Update update = buf.update;
12    -        // read for next iteration
13    -        buf_valid = input_chan.try_read(buf);
     8+      Update update = input_chan.read().update;
14   9       if (last_pid != pid) {
15  10         // commit change to BRAM
16  11         if (last_pid != -1) counts[last_pid] = count;
17  12         count = counts[pid] + 1;
18  13       } else {
19  14         ++count;  // accumulate over register
20  15       }
21  16       last_last_pid = last_pid;
22  17       last_pid = pid;
23  18       // write update to HBM
24    -        updates[...] = update;
25    -      }
26    -    } else {
27    -      // non-blocking read
28    -      buf_valid = input_chan.try_read(buf);
    19+      updates[...] = input_chan.read().update;
29  20     }
30  21   }
```

Listing 6.1: Code snippets with (green lines marked with “+”) and without (red lines marked with “-”) a peek API. Without the peek API, the code snippet is 33% longer and error-prone.

```
1      - struct UpdateWithEot {
2      -   Update update;
3      -   bool eot;
4      - };
5      -
6    1   // test for end-of-transaction token
7      - while (!input_chan.peek().eot) {
8      -   Update update = input_chan.read().update;
     2+  while (!input_chan.eot()) {
     3+    Update update = input_chan.read();
9    4     ...  // accumulate the updates
10   5   }
```

Listing 6.2: Code snippets with (green lines marked with "+") and without (red lines marked with "-") an end-of-transaction (`eot`) API. Without the `eot` API, the code snippet is 2× longer.

to manually add a special bit to the data structure representing the tokens (red lines marked with "-" in Listing 6.2). Note that the `Update` struct is used elsewhere, and it is infeasible to add the `eot` bit directly to the `Update` struct. An alternative solution, i.e., sending the length of transaction to the token consumer beforehand, is not only more complicated, but also impractical in cases where the tokens are generated dynamically and the length of transaction cannot be determined beforehand.

3) *System integration.* To offload computation kernel from the host CPU to FPGA accelerators, programmers need to write host-side code to interface the accelerator kernel with the host. FPGA vendors adopt the OpenCL standard to provide such a functionality. While the standard OpenCL host-kernel interface infrastructure relieves programmers from writing their own operating system drivers and low-level libraries, it is still inconvenient and hard-to-use. Programmers often have to write and debug tens of lines of code just to set up the host-kernel interface. Task-parallel accelerators often make the situation worse because the parallel tasks are often described as distinct OpenCL kernels [90], which significantly increases the programmers' burden on managing these kernels in the host-kernel interface. For our PageRank accelerator, more than 60 lines of host code are created just for the host-kernel integration, which constitute more than 20 percent of the whole source code. Yet, what we actually need is just a single function invocation of the synthesized FPGA bitstream given proper arguments.

```
1    1  void PageRank(...) {
2    2    ...  // declare channels
3      - #pragma HLS dataflow
4      -   Ctrl(...);
5      -   VertexHandler(...);
6      -   ComputeUnit(...);
7      -   ComputeUnit(...);
     3+    tapa::task()
     4+      .invoke(Ctrl, ...)
     5+      .invoke(VertexHandler, ...)
     6+      .invoke(ComputeUnit, ...)
     7+      .invoke(ComputeUnit, ...)
8    8       ...
9      -   UpdateHandler(...);
10     -   UpdateHandler(...);
     9+      .invoke(UpdateHandler, ...)
    10+      .invoke(UpdateHandler, ...)
11  11       ...
    12+    ;
12  13  }
```

Listing 6.3: Code snippets that instantiate tasks in Vivado HLS (red lines marked with "-") and TAPA (green lines marked with "+"). The instantiation interface in Vivado HLS is not verbose, but software simulation does not work correctly.

4) *Software simulation.* C does not have explicit parallel semantics by itself. Vivado HLS uses the dataflow model and allow programmers to instantiate tasks by invoking each of them sequentially [171]. While this is very concise to write (red lines marked with "-" in Listing 6.3), it will lead to incorrect simulation results. This is because the communication between `ComputeUnit` and `UpdateHandler` are bidirectional, but a sequential simulator can only send tokens from `ComputeUnit` to `UpdateHandler` because of their invocation order. This problem was also pointed out in [18]. In order to run software simulation correctly, the programmer can change the source code to run tasks in multiple threads for software simulation, but doing so requires the same piece of task instantiation code to be written twice for synthesis and simulation, reducing productivity. While other tools that run tasks in parallel threads do not have the same correctness problem, we shall show in Section 6.3.4 that such simulators do not scale well when the number of tasks increase.

5) *RTL code generation.* In our PageRank design, the same processing element is instantiated 8

times. This makes the HLS compiler synthesize the same PE module 8 times, taking 7 minutes per compilation. We can reduce the code generation time to less than 1 minute by manually synthesizing each module separately and connecting the generated RTL code, but doing so forces us to debug RTL code and spend tens of minutes to verify the correctness for each code modification, thus defeats the purpose for adopting HLS.

In this section, we present the TAPA framework that addresses these challenges by providing convenient programming interfaces, universal software simulation, and hierarchical code generation.

6.2 TAPA Framework

TAPA is composed of several components. The programming model and interfaces used by TAPA in introduced in Section 6.2.1, followed by the TAPA coroutine-based software simulator presented in Section 6.2.3. Section 6.2.4 discusses the modularized RTL code generator. Section 6.2.5 gives a holistic overview of the automated TAPA framework.

6.2.1 Programming Model and Interface

Different from many HLS compilers that targets KPN, TAPA adopts a hierarchical finite-state machine model, which is detailed in Section 6.2.2. The inter-task communication interface, task instantiation interface, and the system integration interface are presented in Section 6.2.2.1, Section 6.2.2.2, and Section 6.2.2.3, respectively.

6.2.2 Hierarchical Finite-State Machine Model

Similar to KPN described in Section 2.4.4.1, tasks in TAPA communicate via channels. Unlike KPN, tasks are modeled as hierarchical finite-state machines (FSM). Each task is either a leaf that does not instantiate any channels or tasks, or a collection of tasks and channels with which the

tasks communicate. A task that instantiates a set of tasks and channels is called the *parent task* for that set. Each channel must be connected to exactly two tasks that are instantiated in the same parent task. One of the tasks must act as a *producer* and the other must act as a *consumer*. The producer *streams* tokens to the consumer via the channel in the first-in-first-out (FIFO) order. Each task is an FSM, where the tokens streamed to and from the task are inputs and outputs to the FSM. In case of a parent task, the state of all instantiated channels and tasks constitute its state. The producer of a channel can test the fullness of the channel and append tokens to the channel (*write*) if the channel is not full. The consumer of a channel can test the emptiness of the channel and remove tokens from the channel (*read*), or duplicate the head of token without removing it (*peek*), if the channel is not empty. Read, peek, and write operations can be blocking or non-blocking. A blocking operation on an input (output) channel keeps the task FSM in its current state until the channel becomes non-empty (non-full). A non-blocking operation *tries* to perform the operation and returns whether it is successful as one of the inputs to the task FSM. Each task is implemented as a C++ function, which can communicate with each other via the *communication interface*. A parent task instantiates channels and tasks using the *instantiation interface*. One of the tasks is designated as the *top-level task*, which defines the communication interfaces external to the FPGA accelerator. Table 6.1 summarizes TAPA's hierarchical FSM model and other task-parallel programming models.

Table 6.1: Task-parallel programming models.

Programming Model	Peeking	Channel Capacity
Communicating Sequential Processes [84]	Allowed	Not modeled
Kahn Process Network [97]	Not allowed	Not modeled
Synchronous Data Flow [112]	Not allowed	Can be derived
Hierarchical Finite State Machine (Section 6.2.2)	Allowed	Modeled

6.2.2.1 Communication Interface

Tasks communicate with each other through the communication interface. TAPA provides separated communication APIs for the producer side and the consumer side. The producer and consumer tasks of a channel use `ostream` and `istream` as the interfaces, respectively. The interfaces are templated and can be used for any copyable class. On the consumer side, `istream` provides `peek` that allows the programmer to read a token without removing it from the channel, i.e., the state of the channel is not changed. A special token denoting end-of-transaction (EoT) is available to all channels. A producer process can "close" a channel by writing an EoT to it. The consumer process of the same channel will then be able to detect that there is an EoT in this channel, and know that this channel is in "closed" state. The consumer process can also consume the EoT token from the channel, effectively reset the state of the channel from "closed" to "open". An EoT token does not contain any useful data. This is designed deliberately to make it possible to break from a pipelined loop when an EoT is present (Listing 6.2). Table 6.2 summarizes the communication interfaces provided by TAPA. Listing 6.4 shows an example of how the communication interfaces are used in TAPA.

6.2.2.2 Instantiation Interface

A parent task instantiates channels and tasks using the instantiation interface. Channels are instantiated using `tapa::channel<type, capacity>`. For example, `tapa::channel<VertexReq, 2>` instantiates a channel with capacity 2, meaning up to 2 tokens can be written to this channel without reading them out or blocking the producer. Data tokens transmitted using this channel have type `VertexReq`. Tasks are instantiated using `tapa::task::invoke`. By default, a parent task does not finish until all its children tasks finish. A child task can optionally be invoked with `tapa::detach`, meaning the child task is launched and detached immediately, and the parent does not wait for it to finish. The `tapa::detach` invocation type is particularly useful when a task never terminates, e.g., `VertexHandler` with an infinite loop (Listing 6.4). Listing 6.5 shows

Table 6.2: TAPA communication interface.

tapa::ostream<T>& API	**Producer-side functionality**
bool full();	fullness test
void write(T);	blocking write a non-EoT token
bool try_write(T);	non-blocking write a non-EoT token
void close();	blocking write an EoT token
bool try_close();	non-blocking write an EoT token
tapa::istream<T>& API	**Consumer-side functionality**
bool empty();	emptiness test
T peek();	blocking peek a non-EoT token
bool try_peek(T&);	non-blocking peek a non-EoT token
T read();	blocking read a non-EoT token
bool try_read(T&);	non-blocking read a non-EoT token
bool eot();	return if next token is EoT
bool try_eot(bool&);	return if next token exists and if it is EoT
void open();	blocking read an EoT token
bool try_open();	non-blocking read an EoT token

an example of how channels and tasks are instantiated in TAPA.

6.2.2.3 System Integration Interface

To offload a kernel to an FPGA accelerator, programmers will need to integrate the FPGA into the host CPU system. Thanks to the vendor-provided system drivers and the standard OpenCL accelerator APIs, most programmers only need to follow the OpenCL host-kernel communication specification and invoke proper APIs. However, those OpenCL APIs are still verbose and take a

```
void VertexHandler(tapa::istream<VertexReq>& req_s, ...) {
  for (;;) {
    VertexReq req;
    if (req_s.try_read(req)) {
      ...  // handle requests
    }
  }
}

void Ctrl(tapa::ostream<VertexReq>& vertex_req, ...) {
  ...                          // initial setup
  while (...) {                // iterative execution
    VertexReq req(...);        // request vertices
    vertex_req.write(req);
    ...                        // finish scatter & do gather
  }
}
```

Listing 6.4: TAPA communication interface example.

```
void PageRank(...) {
  tapa::channel<VertexReq, 2> vertex_req;
  ...
  tapa::task()
    .invoke<tapa::detach>(VertexHandler, vertex_req, ...)
    .invoke(Ctrl, vertex_req, ...)
    ...
    ;
}
```

Listing 6.5: TAPA instantiation interface example.

long time to learn and develop. For example, programmers need to learn the concepts of "platform", "context", "queue", and "kernel" in OpenCL and manage them for each accelerator, yet the only thing necessary is usually just find a proper FPGA accelerator or simulation environment

and use it to run the program. This overhead for programmers is exacerbated by task-parallel accelerators, where parallel tasks are often synthesized as concurrent OpenCL kernels that need to be managed separately by the host.

TAPA uses a unified system integration interface to further reduce programmers' burden. To offload a kernel to an FPGA accelerator, programmers only need to call the top-level task as a C++ function in the host code. Since TAPA can extract metadata information, e.g., argument type, from the kernel code, TAPA will automatically synthesize proper OpenCL host API calls and emit an implementation of the top-level task C++ function that can set up the runtime environment properly. As a user of TAPA, the programmer can use a single function invocation in the same source code to run software simulation, hardware simulation, and on-board execution, with the only difference of specifying proper bitstreams.

6.2.3 Software Simulation

State-of-the-Art Approach. There are mainly two state-of-the-art approaches that run fast software simulation for task-parallel applications: the sequential approach and the multi-thread approach. A sequential simulator invokes tasks sequentially in the invocation order [171]. Sequential simulators are fast, but cannot correctly simulate the capacity of channels and applications with tasks communicating bidirectionally, as discussed in Section 6.1. A multi-thread simulator invokes tasks in parallel by launching a thread for each task. This enables the capacity of channels and bidirectional communication to be simulated correctly. However, they may perform poorly due to the inefficiency of inter-thread communication and context switch handled by the operating system. The FLASH simulator [18, 25] proposed an alternative to the above, which relies on the HLS scheduling information to mimic the RTL FSM. While this simulation approach itself is faster than multi-thread simulators, generating simulation executable becomes slower due to the need of the HLS scheduler output for cycle-accuracy, which is not needed for correctness verification.

In this section, we present an alternative approach to run software simulation on task-parallel

applications. Given that the inefficiency of multi-thread execution is mainly caused by the preemptive nature of operating system threads and inspired by the widespread adoption of coroutines in modern software languages [54, 104], we propose an approach that uses collaborative coroutines instead of preemptive threads. Note that fast and/or cycle-accurate debugging in general [92] is out of the scope of this section; we focus on the correctness and scalability issues for task-parallel programs.

Coroutine-Based Approach. Routines in programming languages are the units of execution contexts, e.g., functions in C [102]. Coroutines [42] are routines that execute collaboratively; more specifically, coroutines can be explicitly suspended and resumed. Coroutines can even maintain their own stacks. As a result, each coroutine can invoke subroutines themselves and suspend from and resume to any subroutine [104]. Coroutines that have their own stacks are called *stackful* coroutines. A context switch between coroutines takes only 26ns on modern CPUs [104]. As a comparison, an operating system thread context switch takes 1.2 ~2.2µs [4], which is two orders of magnitude slower.

TAPA leverages stackful coroutines to perform software simulation. When channels are instantiated in the simulator, enough memory space is reserved to ensure the channel capacity can be simulated correctly. When tasks are instantiated, a coroutine is launched but suspended immediately for each task. Once all tasks are instantiated, the simulator starts to resume the suspended coroutines. A resumed task will be suspended again if any input channel is accessed when empty or any output channel is accessed when full, which means that no progress can be made from this task. A different task will then be selected and resumed by the simulator.

For example, in the task instantiation code shown in Listing 6.5, both `VertexHandler` and `Ctrl` are launched as coroutines and suspended immediately by the `invoke` function calls. Once all tasks are instantiated, the simulator starts to pick tasks for execution. `Ctrl` is picked first, which will write vertex requests to `vertex_req`. Once `vertex_req` becomes full, the simulator determines that no progress can be made from `Ctrl`, thus will suspend it and pick another task for execution. `VertexHandler` is then resumed and tokens will be read from `vertex_req`.

Once `vertex_req` becomes empty, the simulator determines that no progress can be made from `VertexHandler`, thus will suspend it and pick the next task for execution.

To better utilize the available CPU cores, we use a thread pool to execute the coroutines. We shall show in Section 6.3.4 that the coroutine-based simulator outperforms the existing simulators by 3.2× on average (Section 6.3.4).

6.2.4 RTL Code Generation

State-of-the-art Approach. Current HLS tools treat the whole task-parallel program as a monolithic design, treat channels as global variables, and compile different instances of tasks as if they are completely unrelated. While this enables instance-specific optimizations, e.g., different constant arguments can be propagated to different instances, it can also lead to a significant amount of repeated work. For example, the dataflow architecture generated by the SODA compiler [19,20] is highly modularized, and many modules are functionally identical. However, both the Vivado HLS backend and Intel FPGA OpenCL backend of SODA generate RTL code for each SODA module separately. When the design scales out to hundreds of modules, RTL code generation can easily run for hours, taking even longer time than logic synthesis and implementation. While we recognize that a programmer can manually generate RTL code for each task and glue them at RTL level to speed up RTL code generation, doing so defeats the purpose of using HLS for high productivity, because the glued RTL code can be error-prone yet cannot be verified using fast software simulation. We also recognize that fast RTL code generation in general is an interesting problem, but we focus on the inefficiency exacerbated by task-parallel programs in this paper.

Modularized Approach. Thanks to the hierarchical programming model, TAPA can keep the program hierarchy, recognize different instances of the same task, and compile each task only once. As such, the total amount of time spent on RTL code generation is reduced. Moreover, modularized compilation makes it possible to compile tasks in parallel, further reducing RTL code generation time on multicore machines. TAPA implements this by doing a source-to-source transformation to generate the vendor HLS code for each task and invoking the vendor tools in

parallel for each task. On average, TAPA reduces HLS compilation time by 4.9× (Section 6.3.5).

6.2.5 TAPA Automation Overview

The TAPA automation flow is shown in Figure 6.2. The TAPA C++ source code can be compiled directly for software simulation and correctness verification. Starting from the same TAPA C++ source code, TAPA extracts the HLS code for each task and the metadata information of the whole design, including the communication topology among tasks, token types exchanged between tasks, and channels' capacity. The vendor HLS tool is then leveraged to generate RTL code and performance/resource report for each task. The extracted metadata is used to instantiate the task instances and connect them together systematically, producing the overall HLS report and kernel RTL code, which can be used for QoR tuning and logic synthesis and implementation, respectively. The same metadata information is also used to create the host-kernel communication interface, which can be used for on-board execution or optionally RTL simulation.

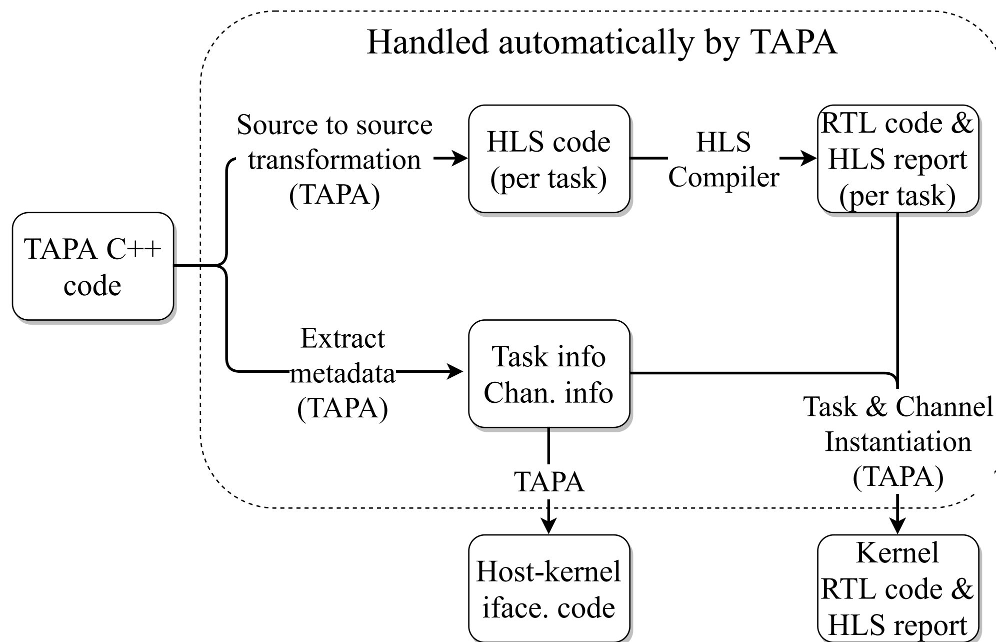


Figure 6.2: TAPA automation flow overview.

6.3 Evaluation

We prototype TAPA on Xilinx devices using Vivado HLS as the backend; support for Intel devices will be added later. Clang compiler infrastructure is modified to extract information about tasks and perform source-to-source transformation to generate Vivado HLS kernel code and OpenCL host code. GCC is used to compile the host executables and the software simulators. We compare the productivity of TAPA with two vendor tools that provide end-to-end high-level programming experience (including host-kernel communication): Xilinx Vitis/Vivado HLS 2019.2 suite and Intel FPGA SDK for OpenCL Pro Edition 19.4. The experimental results are obtained on an Ubuntu 18.04 server with 2 Xeon Gold 6244 processors.

6.3.1 Benchmarks

We used the following benchmarks for comparison. All implementations (Vivado HLS, Intel OpenCL, and TAPA) of each benchmark are written in such a way that tasks in each implementation have one-to-one correspondence, corresponding loops are scheduled with the same initiation interval (II), and each task performs the same computation. This guarantees all tools generate consistent quality of results. Note that we aim to compare the productivity of each of the HLS tools, not the quality of result. In particular, we were unable to guarantee that the generated RTL codes have exactly the same behavior without having access to the HLS compiler's scheduling algorithm. For example, the network switch implemented in TAPA has a total latency of 3 cycles while the Vivado HLS implementation has a total latency of 6. This is inevitable because, using Vivado HLS, one has to manually buffer the incoming packets, forcing an additional latency of 1 cycle at each network stage. Table 6.3 summarizes the number of tasks and channels used in each benchmark.

Cannon's algorithm [113] is a distributed algorithm for matrix multiplication that runs on 2D mesh of processing elements (PE). This benchmark contains 8×8 PEs. Each PE is internally vectorized to perform 8 multiply-accumulate operations per cycle for two 128×128 matrices. Besides

Table 6.3: Number of tasks and channels in each benchmark. Each task may be instantiated multiple times, so the number of task instances is greater than the number of tasks.

Benchmark	#Tasks	#Task Instances	#Channels
cannon	5	91	344
cnn	14	209	366
gaussian	15	564	1602
gcn	5	12	25
gemm	14	207	364
network	3	14	32
page_rank	4	18	89

the 64 PEs, the accelerator also contains 9 data distributor/collector for each matrix. The inputs to the whole accelerator are 1024×1024×1024.

Convolutional neural networks are very popular for many machine learning applications, e.g., image classification [156]. This benchmark implements the third layer of VGG [156] [3] based on a systolic array implementation generated from AutoSA [163]. AutoSA is a polyhedral-based systolic array auto-compilation framework that can generate optimal designs within one hour with performance comparable to state-of-the-art manual designs.

Gaussian filter is often employed for low-pass filtering on input signals or images, or used iteratively for solving linear system of equations. This benchmark is based on a dataflow microarchitecture generated from SODA [20]. SODA is a stencil compiler that can generate optimal communication-reuse buffers with temporal and spatial parallelism. This benchmark performs 8 iterations of Gaussian filtering, each of which is capable of processing 16 input elements in parallel. The input size is 32768×32768.

Graph convolutional network [101] is an emerging type of neural network that processes sparse

[3] Parameters $\{i, o, h, w, p, q\} = \{512, 512, 56, 56, 3, 3\}$.

and irregular data as opposed to dense and regular ones like images. This benchmark implements a forward layer of GCN for the Cora dataset [153], which contains 2708 vertices and 10556 edges. The input and output features have 1433 and 16 dimensions, respectively.

General matrix multiplication is based on a systolic array implementation generated from AutoSA [163]. Compared with Cannon's algorithm, AutoSA avoids feedback data paths in the systolic array, and can support non-square matrices. The input matrices to the accelerator are 1040×1024 and 1024×1024.

Network switching implements an 8×8 Omega network switch [111] that can route packets from any input port to any output port. The packets are 64-bit wide with the first 3 bits being the header and are generated randomly with an even distribution among the 8 destination ports.

PageRank implements the PageRank [139] citation ranking algorithm for general large graphs as described in Section 6.1. We use the Slashdot community graph [117] as the dataset for debugging, which contains 77360 vertices and 905468 edges. The accelerator design itself can scale up to 2^{26} vertices and 2^{28} edges.

6.3.2 Lines of Kernel Code

TAPA simplifies the kernel code in two aspects. First, the TAPA communication interfaces simplify the code with the built-in support for peeking and transactions. This not only simplifies the body of each task definition, but also removes the necessity for many `struct` definitions. Second, the TAPA instantiation interfaces simplify the code by allowing tasks to be launched and detached concisely. Without this functionality, each task in Vivado HLS must be carefully given a termination condition, whereas Intel OpenCL requires verbose kernel instantiation attributes for each instance of task. Figure 6.3 shows the lines of kernel code comparison of each benchmark. On average, TAPA reduces the lines of kernel code by 22%. Note that only synthesizable kernel code is counted; code added for multi-thread software simulation is not counted for Vivado HLS.

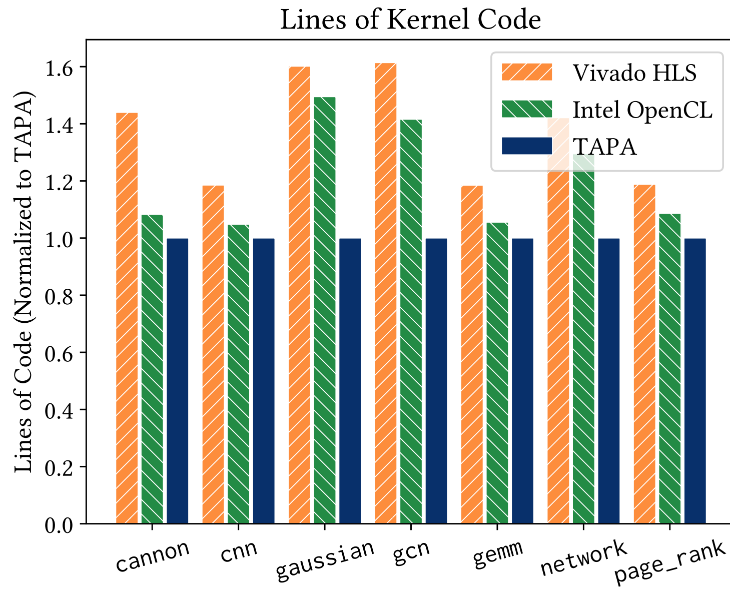


Figure 6.3: LoC comparison for kernel code. Lower is better.

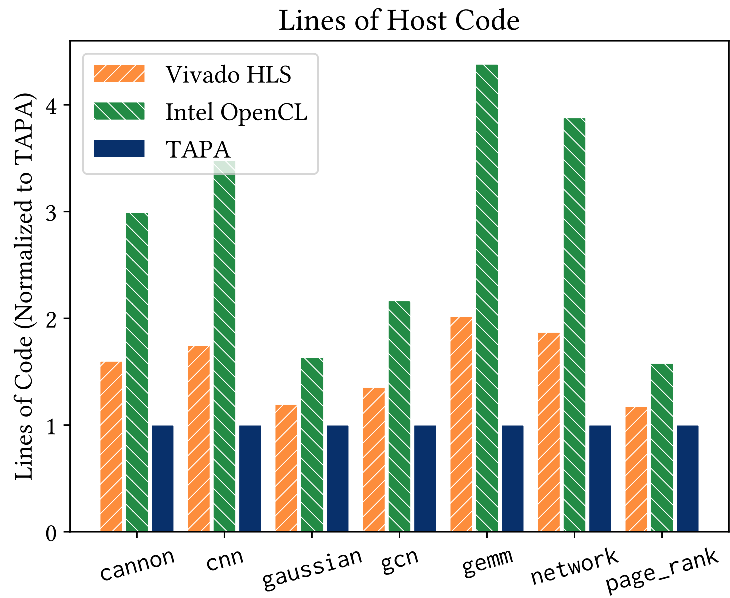


Figure 6.4: LoC comparison for host code. Lower is better.

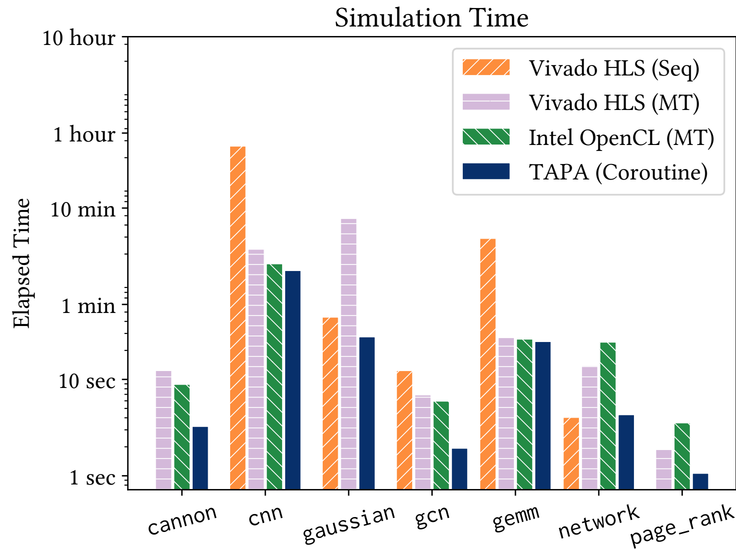


Figure 6.5: Simulation time comparison. Lower is better. The sequential simulator fails to simulate `cannon` and `pagerank` correctly. The Intel OpenCL multi-thread simulator cannot simulate `gaussian` due to its large number of task instances.

6.3.3 Lines of Host Code

The host code used in the benchmarks contains a minimal testbench to verify the correctness of the kernel code. TAPA system-integration API automatically interfaces with the OpenCL host APIs and relieves the programmer from writing repetitive code just to connect the kernel to a host program. Table 6.4 shows the lines of host code comparison. On average, the length of host code is reduced by 51%.

6.3.4 Software Simulation Time

Figure 6.5 shows four simulators, that is, the sequential Vivado HLS simulator, the multi-thread Vivado HLS simulator, the multi-thread Intel OpenCL simulator, and the coroutine-based TAPA simulator. Among the three simulators, the sequential simulator fails to correctly simulate benchmarks that require feedback data paths (`cannon` and `page_rank`). Due to the larger memory

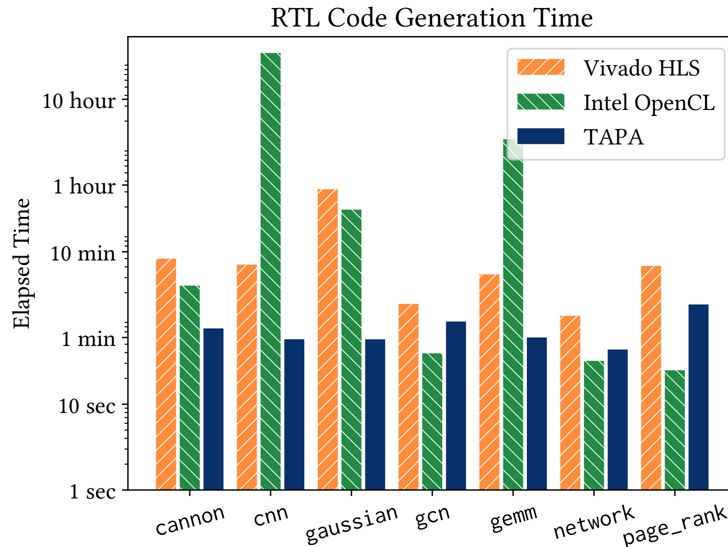


Figure 6.6: RTL code generation time. Lower is better.

footprint required for storing the tokens transmitted between tasks and lack of parallelism, the sequential simulator is outperformed by the coroutine-based simulator in all but one of the benchmarks (`network`). The two multi-thread simulators correctly simulate all benchmarks, except that Intel OpenCL cannot handle `gaussian` because its large number of task instances (564) exceeds the maximum allowed (256) by the simulator. However, the multi-thread simulators perform poorly on benchmarks that are communication-intensive (e.g., `network`) or have more tasks than the number of available threads (e.g., `gaussian`). The coroutine-based TAPA simulator can correctly simulate all benchmarks without significant performance loss for both communication-intensive and computation-intensive tasks with 3.2× average speedup.

6.3.5 RTL Code Generation Time

Figure 6.6 shows the RTL code generation time comparison. Thanks to the hierarchical programming model and modularized code generator, TAPA accelerates the HLS compilation time by 6.8× on average. This is because ① TAPA runs HLS for each task only once even if it is instantiated many times, while Vivado HLS and Intel OpenCL runs HLS for each task instance; ② TAPA runs

HLS in parallel on multicore machines.

6.4 Alternative Compilation Paths

We choose to use C++ as the base language of TAPA because C++ is one of the most commonly used languages for both commercial and academic HLS compilers [11, 37, 91, 95]. Extending HLS C++ makes it easier to migrate existing HLS programs to leverage TAPA. As a future work, we hope to extend the frontend of TAPA to support more than just TAPA HLS C++. A possible extension is to make TAPA take standard Vitis HLS C++ as input. This will enable programmers to benefit from the unified system integration interface and fast hierarchical code generation provided by the TAPA compiler without any change in the kernel code. Task-level parallelism can be exposed using the standard HLS dataflow pragma and the existing source-to-source transformations in TAPA can be reused.

A more challenging possible extension is to support a different task-parallel language/library, which requires additional effort to properly handle the new language constructs. There are task-parallel languages/libraries that support communication based on message passing, e.g., MPI [68] and Go [128], which is consistent with the communication interfaces in TAPA. To support these languages, we will need to insert a common intermediate representation (IR) layer between the TAPA frontend and the HLS backend, and lower the new input language to this common IR. Listing 6.6 shows a merge sort example written in Go that may be transformed into TAPA as illustrated in Listing 6.7. Since the language constructs are similar, the major challenge would be to compile and lower the languages properly.

It would be even more challenging to support task-parallel languages/libraries where tasks communicate using shared memory instead of message passing, e.g., Cilk [6] and OpenMP [46]. On CPU or even HLS-based FPGA accelerators [123], one often relies on a fully-connected cache-coherent shared memory system to enable such inter-task communication. While TAPA allows multiple task instances to share the same memory space, currently it does not provide caching.

```go
func Sort(data_in []int, n int, i int, data_out chan<- int) {
  // sort data_in and write to data_out...
  close(data_out)
}
func Merge(data_in_0 <-chan int, data_in_1 <-chan int, data_out []int, n int) {
  var data_0, data_1 int                 // data buffers
  is_0_valid, is_1_valid := false, false // whether the buffer is valid
  for i := 0; ; i++ {
    // read data if the buffer is not valid and the channel is not closed
    if !is_0_valid {
      data_0, is_0_valid = <-data_in_0
    }
    if !is_1_valid {
      data_1, is_1_valid = <-data_in_1
    }
    if is_0_valid && !(is_1_valid && data_1 < data_0) {
      data_out[i] = data_0
      is_0_valid = false
    } else if is_1_valid {
      data_out[i] = data_1
      is_1_valid = false
    } else {
      break
    }
  }
}
func MergeSort(data []int, n int) {
  sorted_0 := make(chan int, 2)
  sorted_1 := make(chan int, 2)
  var wg sync.WaitGroup
  wg.Add(3)
  go func() { Sort(data, n, 0, sorted_0); wg.Done() }()
  go func() { Sort(data, n, 1, sorted_1); wg.Done() }()
  go func() { Merge(sorted_0, sorted_1, data, n); wg.Done() }()
  wg.Wait()
}
```

Listing 6.6: Merge sort written in Go.

```
void Sort(tapa::mmap<int> data_in, int n, int i, tapa::ostream<int> &data_out) {
  // sort data_in and write to data_out...
  data_out.close();
}
void Merge(tapa::istream<int> &data_in_0, tapa::istream<int> &data_in_1,
           tapa::mmap<int> data_out, int n) {
  int data_0, data_1;                          // data buffers
  bool is_0_valid = false, is_1_valid = false; // whether the buffer is valid
  for (int i = 0;; ++i) {
    // read data if the buffer is not valid and the channel is not closed
    if (!is_0_valid && !data_in_0.eot(nullptr)) {
      data_0 = data_in_0.read();
      is_0_valid = true;
    }
    if (!is_1_valid && !data_in_1.eot(nullptr)) {
      data_1 = data_in_1.read();
      is_1_valid = true;
    }
    if (is_0_valid && !(is_1_valid && data_1 < data_0)) {
      data_out[i] = data_0;
      is_0_valid = false;
    } else if (is_1_valid) {
      data_out[i] = data_1;
      is_1_valid = false;
    } else {
      break;
    }
  }
}
void MergeSort(tapa::mmap<int> data, int n) {
  tapa::stream<int, 2> sorted_0, sorted_1;
  tapa::task()
      .invoke(Sort, data, n, 0, sorted_0)
      .invoke(Sort, data, n, 1, sorted_1)
      .invoke(Merge, sorted_0, sorted_1, data);
}
```

Listing 6.7: Merge sort written in TAPA.

```
void Sort(int* data, int n, int i) {
  // sort data...
}
void Merge(int* data, int n) {
  int i_0 = 0, i_1 = 0, i = 0;
  const int n_0 = n / 2, n_1 = n - n_0;
  for (; i_0 < n_0 && i_1 < n_1; ++i) {
    if (data[n + n_0 + i_1] < data[n + i_0]) {
      data[i] = data[n + i_0];        // input data stored in [n, n+n_0)
      ++i_0;
    } else {
      data[i] = data[n + n_0 + i_1];  // input data stored in [n+n_0, n*2)
      ++i_1;
    }
  }
  for (; i_0 < n_0; ++i_0, ++i) {
    data[i] = data[n + i_0];
  }
  for (; i_1 < n_1; ++i_1, ++i) {
    data[i] = data[n + n_0 + i_1];
  }
}
void MergeSort(int* data, int n) {
  cilk_spawn Sort(data, n, 0);
  cilk_spawn Sort(data, n, 1);
  cilk_sync;
  Merge(data, n, 0);
}
```

Listing 6.8: Merge sort written in Cilk.

Since neither Cilk nor OpenMP supports explicit message passing between task instances, it is challenging for the compiler to extract and optimize parallel communication patterns and create fine-grained message-passing interfaces as used in TAPA programs. With only shared memory interfaces, the program may be suboptimal due to the under-optimized inter-task communication. Listing 6.8 shows an example of such a challenging case of merge sort written in Cilk. Compared with Listing 6.6 (written in Go) or Listing 6.7 (written in TAPA), Listing 6.8 requires addition

external memory accesses to store the intermediate sorting results produced by the two `Sort` tasks. Moreover, because of the additional memory accesses, `Merge` cannot run in parallel with `Sort`.

6.5 Summary

In this chapter, we present TAPA as an HLS C++ language extension to enhance the programming productivity of task-parallel programs on FPGAs. TAPA has multiple advantages over state-of-the-art HLS tools: on average, ① its enhanced programming interface helps to reduce the lines of kernel code by 22%, ② its unified system integration interface reduces the lines of host code by 51%, ③ its coroutine-based software simulator shortens the correctness verification development cycle by 3.2×, ④ its modularized code generation approach shortens the QoR tuning development cycle by 6.8×. As a fully automated and open-source framework, TAPA aims to provide highly productive development experience for mapping task-parallel programs to FPGA accelerators using HLS. However, task-parallel accelerators with dynamically scheduled memory accesses are still often under-optimized. In the next chapter, we shall extend TAPA with a set of high-level libraries to help simplify design automation and optimization for such dynamically scheduled programs.

CHAPTER 7

Supporting Dynamically Scheduled Programs

Chapter 6 solves the problem to map memory-bound applications statically to hardware accelerators, but many applications require dynamic scheduling. For example, Dijkstra's algorithm for single-source shortest path visits vertices in a graph dynamically based on their tentative distances. In such cases, HLS designs often under-utilize the hardware and produce undesired quality of result. In this chapter, we start with two micro-benchmarks as motivating examples, and then propose a set of high-level libraries to simplify design automation and optimization for dynamically scheduled programs. These libraries are used to implement the applications presented in Chapter 8.

7.1 Motivating Examples

7.1.1 Dynamic Off-Chip Memory Accesses

Listing 7.1 shows a micro-benchmark that reads from the off-chip memory with a runtime-determined (*dynamic*) access pattern. This kind of workload is common, e.g., in a customized cache (Section 8.1). While this micro-benchmark can be synthesized with an initiation interval (II) of one and implemented at 300 MHz on an Alveo U280 board, the actual II (measured number of cycles on board over number of iterations) is much larger (11.95) when using addresses generated from a 21-bit linear-feedback shift register (LFSR) pseudo-random number generator. Even though the compiler can statically schedule the loop with II=1, when actually running on board, the loop is stalled very frequently to wait for the dynamic memory requests. This is not

```
1 void ReadFromOffChipMemory(tapa::istream<Addr>& addr_q,
2                            tapa::ostream<Data>& data_q, tapa::mmap<Data> data) {
3   for (int i = 0; i < (1 << 20); ++i) {
4     if (!addr_q.empty()) {
5       data_q.write(data[addr_q.read()]);
6     }
7   }
8 }
```

Listing 7.1: A micro-benchmark that reads from the off-chip memory `data` with a runtime-determined access pattern.

only because off-chip memory accesses have a long latency, but also because the kernel cannot overlap memory accesses well. We shall show in Section 7.2 that this II can be greatly reduced from 11.95 to 4.14, which makes the performance bound by the off-chip memory system alone.

7.1.2 Dynamic On-Chip Memory Accesses

```
1 void AccumulateOverOnChipMemory(tapa::istream<Addr>& addr_q,
2                                 tapa::ostream<Data>& data_q) {
3   Data data[kOnChipDataSize];
4   for (int i = 0; i < (1 << 20); ++i) {
5     if (!addr_q.empty()) {
6       data_q.write(data[addr_q.read()] += 4.2f);
7     }
8   }
9 }
```

Listing 7.2: A micro-benchmark that accumulates values over the on-chip memory `data` with a runtime-determined access pattern.

Listing 7.2 shows a micro-benchmark that accumulates values over the on-chip memory with a dynamic access pattern. This type of workload is common, e.g., in graph applications (Sec-

tion 8.2). Due to the potential read-after-write dependency on elements in array `data`, this microbenchmark can only be synthesized with an II of 11 on an Alveo U280 board running at 300 MHz. Using addresses generated from a 21-bit LFSR, the II measured on board is 11.06, which is consistent with the synthesis result. However, for such random addresses, read-after-write dependency is almost never violated. We shall show in Section 7.3 that this II can be safely decreased from 11.06 to 1.03 by adopting a simple dynamic conflict resolver with only 3.9% resource overhead.

7.2 Supporting Dynamic Off-Chip Memory Accesses

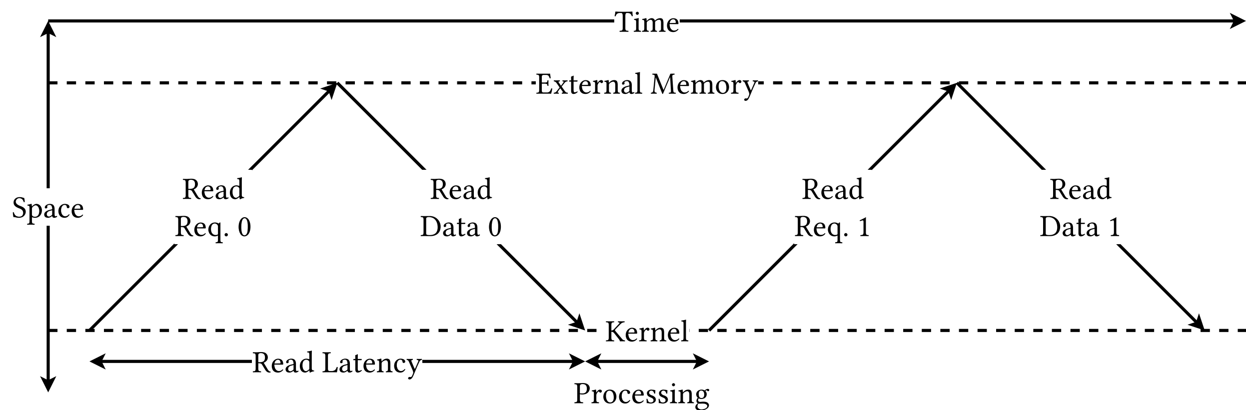


Figure 7.1: Synchronous off-chip memory accesses without burst.

Off-chip memory accesses are more and more important today, due to the widening gap between the on-chip computation unit and the off-chip external memory system. For a state-of-the-art FPGA (e.g., Alveo U280), while the on-chip memory (e.g., URAM) can operate at a short clock period (e.g., 3.3ns), the external memory latency is almost two orders of magnitude longer (e.g., 182ns) [26]. Without any latency-hiding optimizations, off-chip memory accesses can severely limit the performance of an accelerator. This is especially true for high-level synthesis (HLS) designs, where the sequential C/C++ semantics imply synchronous memory accesses, i.e., the next memory operation cannot start until the current one finishes. This is shown in Figure 7.1.

To help exploit the off-chip memory system, Vitis HLS provides an optimization called "burst"

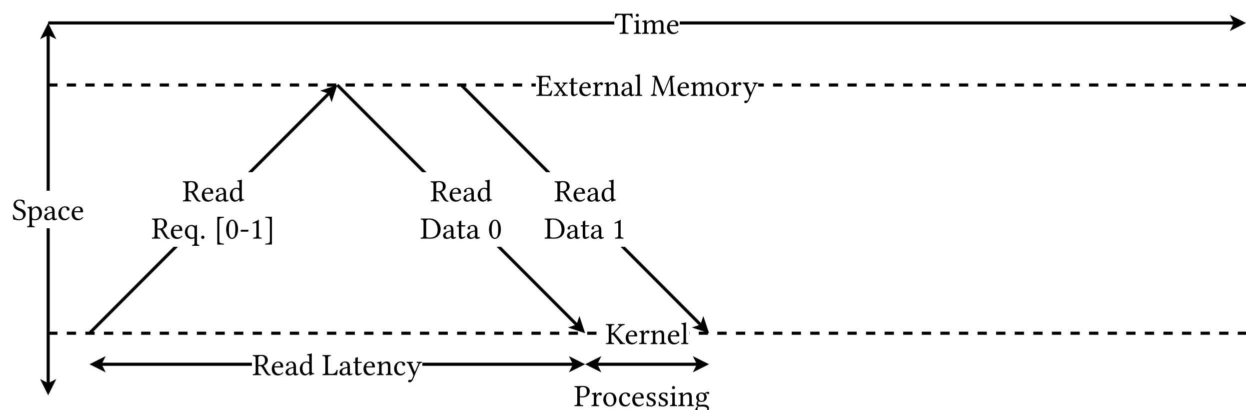


Figure 7.2: Synchronous off-chip memory accesses with burst.

mode accesses to hide the long memory latency. Memory accesses must be statically known and continuous to enable burst mode. In burst mode, since the memory addresses are statically know, the kernel can start many memory accesses using a single request without serializing each access. This is demonstrated in Figure 7.2.

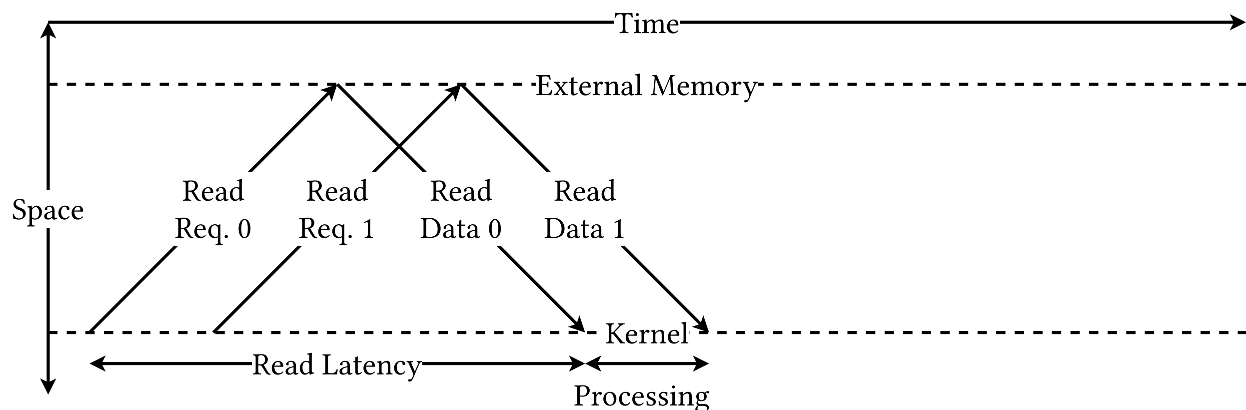


Figure 7.3: Asynchronous off-chip memory accesses.

Burst mode accesses enables full memory utilization, but only for statically know continuous memory accesses. In many cases, it is not always possible to use such memory accesses. To give programmers more choices for the off-chip memory system, we extend the memory interfaces in TAPA (Chapter 6) to support asynchronous memory accesses. As illustrated in Figure 7.3, asynchronous memory accesses allows programmers to start the next memory access without

waiting for the current memory access to finish. As such, programmers are given the flexibility to aggressively issue multiple outstanding memory requests.

Table 7.1: TAPA asynchronous memory-mapped interface.

`tapa::async_mmap<T>` Channel	**Functionality**
`tapa::ostream<uint64_t>& read_addr;`	Write an address to initiate a memory read request
`tapa::istream<T>& read_data;`	Read data fetched from the address given by `read_addr`
`tapa::ostream<uint64_t>& write_addr;`	Write an address to initiate a memory write request
`tapa::ostream<T>& write_data;`	Write data to the address given by `write_addr`
`tapa::istream<uint8_t>& write_resp;`	Read the number of completed write requests

TAPA supports the same synchronous memory interface as Vitis HLS via `tapa::mmap`, which can be accessed via `operator[]` directly. To support asynchronous memory accesses, we extend `tapa::mmap` to `tapa::async_mmap` with 5 streaming channels: two for *read addresses* and *read data*, three for *write addresses*, *write data*, and *write acknowledgments*. Each channel can be accessed using `tapa::istream` and `tapa::ostream` as presented in Section 6.2.2.1. Table 7.1 summarizes the functionality of each channel in `tapa::async_mmap`.

Listing 7.3 illustrates the micro-benchmark shown in Listing 7.1 using TAPA's asynchronous memory-mapped interface. Measured on the same Alveo U280 board, the average II (number of cycles on board over number of iterations) is reduced from 11.95 to 4.14 with 23% more LUTs but 96% less FFs and no BRAM.

TAPA's asynchronous memory-mapped interface (`tapa::async_mmap`) is used to implement the customized vertex cache (CVC) in Section 8.1.4. Without the asynchronous access support, we will not be able to fully pipeline the CVC nor to create the shortest path accelerator (Section 8.1). Section 8.1.6.2 evaluates the hit rate of the CVC. Besides, the random access bandwidth benchmark in Section 8.2.1.4 also takes advantages of `tapa::async_mmap`.

```
void ReadFromOffChipMemory(tapa::istream<Addr>& addr_q,
                           tapa::ostream<Data>& data_q,
                           tapa::async_mmap<Data> mem) {
  for (int i = 0; i < (1 << 20); ++i) {
    if (!addr_q.empty() && !mem.read_addr.full()) {
      mem.read_addr.write(addr_q.read()); //
    }

    if (!mem.read_data.empty() && !data_q.full()) {
      data_q.write(mem.read_data.read());
    }
  }
}
```

Listing 7.3: Optimized version of the micro-benchmark shown in Listing 7.1 using TAPA. The runtime average II is reduced from 11.95 from 4.14 with 23% more LUT usage but 96% less FF usage and no BRAM.

7.3 Supporting Dynamic On-Chip Memory Accesses

```
for (int i = 0; i < n; ++i) {
  const auto edge = edges[i];
  vertices[edge.dst].new_ranking += vertices[edge.src].old_ranking * ... ;
}
```

Listing 7.4: Dynamic on-chip memory accesses with a large initiation interval (II).

While on-chip memory accesses do not suffer from long latency (Section 7.2), supporting dynamic accesses efficiently is still challenging. A notable challenge is to handle read-modify-write operation with dynamic random accesses, which is a common irregular access pattern in graph applications. For example, the PageRank algorithm accumulates weighted rankings from the neighbors. Using an edge-centric traversal model, this can be implemented using a simple

loop shown in Listing 7.4, where the `vertices` array is accessed with a dynamic index. The accumulation operation creates a read-after-write dependency, i.e., the next iteration must not read from an array element until it has been written to the on-chip storage (e.g., URAM), which can take several clock cycles if the accumulation operates on floating point numbers. Although the edges may be reordered arbitrarily, state-of-the-art commercial HLS tools pipeline such a loop pessimistically to make sure it is always correct. This means even if consecutive loop iterations do not really violate any dependency, the pipeline is still stalled for many cycles. Dynamatic [95], an academic HLS tool, also noticed this problem, and proposed to dynamically schedule the whole HLS program as a solution. However, doing so has demonstrated non-trivial resource overhead due to extensive hand shaking and the lack of resource sharing [17].

```
1  constexpr int kDepDist = 5;
2  using AddrType = int;
3  constexpr AddrType kNullAddr = -1;
4  tapa::dependency_detector<AddrType, kDepDist - 1> detector(kNullAddr);
5  for (int i = 0; i < n;) {
6  #pragma HLS dependence true variable = vertices distance = kDepDist
7    const auto edge = edges[i];
8    AddrType addr = kNullAddr;
9    if (detector.is_conflict_free(edge.dst)) {
10     vertices[edge.dst].new_ranking += vertices[edge.src].old_ranking * ...;
11     addr = edge.dst;
12     ++i;
13   }
14   detector.record(addr);
15 }
```

Listing 7.5: Dynamic on-chip memory accesses with a dynamic dependency resolver.

In this dissertation, we solve the dynamic on-chip memory accesses using a different approach. Noticed that HLS tools provide directives to partially disable static analysis and let the programmers handle dependency (e.g., via `#pragma HLS dependence` in Vitis HLS and `#pragma ivdep` in Intel OpenCL for FPGA), we propose to leverage such features and resolve dependency

at runtime. Once such a pragma is applied, the loop shown in Listing 7.4 can be fully pipelined, because the HLS compiler is instructed that each memory address will not appear until `kDepDist` = 12 iterations later. Therefore, the compiler will think that there is no read-after-write dependency. To make sure the same address actually will not appear until 12 iterations later, we add a dependency resolver to record and detect conflicting addresses. The original loop body will be replaced with a bubble if an address has appeared within the latest 12-1=11 iterations, making sure of correctness. Listing 7.5 illustrates how the example in Listing 7.4 can be fully pipelined with the help of a dynamic dependency resolver.

Table 7.2: Summary of the TAPA dependency detector API.

API	`tapa::dependency_detector<T, N>(T null_addr);`
Functionality	Constructs a dependency detector that can detect conflicts of addresses of type `T` with the previous `N` addresses. After construction, the previous addresses are initialized to `null_addr`.
API	`tapa::dependency_detector<T, N>::is_conflict_free(T addr);`
Functionality	Returns true if `addr` is conflict free with the previous `N` addresses.
API	`tapa::dependency_detector<T, N>::record(T addr);`
Functionality	Records an accessed address `addr`.

Table 7.2 summarizes the TAPA dependency detector API. The dependency detector should always be used with `#pragma HLS dependence` with properly determined dependency distance, in order to resolve dependency correctly. Listing 7.6 illustrates how it can be used in the example shown in Listing 7.2. The dynamic dependency detector can be implemented using shift registers. In each loop iteration, a new address is pushed to the shift registers and the oldest address is discarded (Line 14 in Listing 7.5). To detect conflicts (Line 9 in Listing 7.5), the dependency detector simply checks in parallel if the input address exists in any of the shift registers.

An obvious limitation of the dynamic dependency resolver is that it only inserts bubbles to

```cpp
void AccumulateOverOnChipMemory(tapa::istream<Addr>& addr_q,
                                tapa::ostream<Data>& data_q) {
  Data data[kOnChipDataSize];
  constexpr int kDepDist = 12;
  const OnChipAddr kNullAddr = -1;
  tapa::dependency_detector<OnChipAddr, kDepDist - 1> detector(kNullAddr);
  for (int i = 0; i < (1 << 20);) {
#pragma HLS dependence true variable = data distance = kDepDist
    OnChipAddr addr = kNullAddr;
    if (addr_q.try_peek(addr) && detector.is_conflict_free(addr)) {
      addr_q.read();
      data_q.write(data[addr] += 4.2f);
      ++i;
    }
    detector.record(addr);
  }
}
```

Listing 7.6: Optimized version of the micro-benchmark shown in Listing 7.2 using TAPA.

stall the pipeline, and cannot reorder the loop iterations to further reduce stalling. While it is theoretically possible to do *online* reordering, such reordering requires much more complex hardware than the simple dynamic dependency resolver and can easily become the frequency bottleneck since the pipeline dependency resolver itself cannot be pipelined. In Section 8.2.2, we shall introduce an *offline* method that reduces stalling by reordering the addresses and only insert bubbles when necessary.

Listing 7.6 demonstrates the improved version of the micro-benchmark shown in Listing 7.2 using TAPA's dynamic conflict resolver. Measured on the same Alveo U280 board, the II is reduced from 11.06 to 1.03 with only 3.9% more LUTs and FFs.

Table 7.3: Comparison of naïve and optimized versions of the micro-benchmarks with dynamic memory accesses.

Micro-Benchmark	Clock/MHz	II	LUT	FF	BRAM	URAM	DSP
Naïve off-chip accesses	300	11.95	1189	3740	15	0	0
Optimized off-chip accesses	300	4.14	1466	162	0	0	0
Naïve on-chip accesses	300	11.06	3708	5094	0	32	32
Optimized on-chip accesses	300	1.03	3854	5297	0	32	32

7.4 Summary

In this chapter, we further extended TAPA presented in Chapter 6 to support dynamically schedule off-chip and on-chip memory accesses. For off-chip memory accesses, we add an asynchronous memory-mapped interface so that the long off-chip memory latency can be hidden very well. For on-chip memory accesses, we address the potential read-after-write dependency caused by long computation latency compared with the short on-chip memory latency. Table 7.3 summarizes the results using two representative micro-benchmarks for each type of memory accesses.

With the extensions introduced in this chapter, TAPA can now support task-parallel programs with both statically and dynamically scheduled memory accesses. The task instantiation, however, is still static at the language level. One of the major challenges to support dynamic task instantiation is to make sure the dynamically instantiated tasks can always communicate with other tasks whenever necessary. TAPAS [123] partially addresses this problem by using a shared memory system for inter-task communication, but it lacks the support for highly concurrent inter-task communication via hardware channels. TAPA leaves the responsibility of dynamically instantiating tasks to the programmer so that the compiler can leverage programmers' knowledge to optimize the inter-task communication channels. In the next chapter, we shall discuss

two real-world graph applications as representatives for irregular memory-intensive applications, and evaluate the effectiveness of the extended TAPA framework.

CHAPTER 8

Tackling Irregular Access Pattern: Graph Applications

In Chapter 6, we have presented TAPA, a highly productive extension to HLS for task-parallel HLS programs. Chapter 7 further extends TAPA to support dynamically scheduled memory accesses. This enables us to use TAPA to implement memory-intensive applications with irregular memory accesses. In this chapter, we will present two design & optimization case studies of such irregular applications, using two graph analytics applications, i.e., single-source shortest path (SSSP) and graph convolutional networks (GCN). For SSSP, the amount of edge traversal highly depends on the priority scheduling of vertices, which can only be determined at run time. As such, in Section 8.1, we take the opportunity to showcase our support for dynamically scheduled off-chip memory accesses. For GCN, memory accesses can be reordered without changing the total amount of traversal. Therefore, in Section 8.2, we first discuss the best approach to schedule the memory accesses, followed by the challenges imposed by irregular on-chip memory access.

8.1 Single-Source Shortest Path

The graph is a universal data structure that models relationships, connections, and structures. The single-source shortest path (SSSP) problem, one of the most important and well-studied graph problems, finds its prevalent application in road navigation [74], telecom network routing [143], neural image reconstruction [119], and social network analysis [10]. Although we have known Dijkstra's algorithm [61] and its priority queue–based variants [69, 94] for several decades, these algorithms are inherently sequential and are not easily parallelizable, because increasing parallelism is often at the cost of increasing the total amount of work as well. As such, efficient paral-

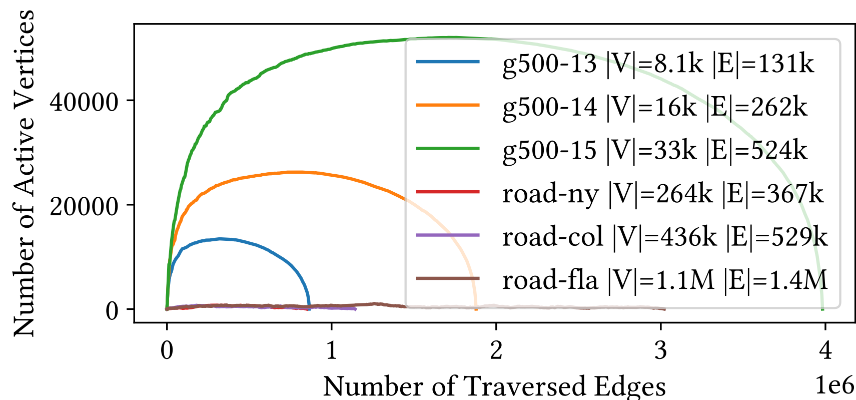


Figure 8.1: Change of the number of active vertices as edges are traversed. The g500 graphs are power-law graphs and the road graphs are planar graphs.

lelization of SSSP algorithms are still an active field of research today [51, 114, 127, 160, 164, 180].

Compared with CPUs and GPUs, FPGAs have the unique capability of customizing the control flow and data paths, which has demonstrated tremendous potential in various application domains, including stencil computations [19, 20, 52], neural networks [163, 178], and general graph algorithms [13,87,165]. This makes the FPGA a naturally good candidate platform for SSSP acceleration, since the high-throughput on-chip priority queues [5, 115] enable effective control over the trade-off between parallelism and the amount of work [1, 114]. However, such on-chip priority queue–based approach has been applied only to uniform-degree planar graphs, yet many real-world graphs have skewed degree distributions, which are often modeled using the power law [30]. Compared with planar graphs, power-law graphs have a much larger frontier of active vertices, which requires a priority queue with a much larger capacity. Even worse, such a capacity requirement increases rapidly as the size of graph grows, making it infeasible to keep the priority queue on-chip. This is demonstrated in Figure 8.1.

Another challenge to implement priority queue–based SSSP algorithms efficiently is that priority-order graph traversal prohibits many reordering techniques used in many graph accelerators [13, 47–49, 87, 154, 181], which are vitally important to reducing external memory traffic and achieving high performance. As such, many graph accelerators [13, 87, 181] implement the

Bellman-Ford algorithm [155] that does not require a priority queue at all. However, these accelerators work best for algorithms whose amount of work is insensitive to the traversal order (e.g., SpMV and PageRank). For SSSP, the total amount of traversed edges (i.e., amount of work) can be very different with different traversal orders. We will show in Section 8.1.6.4 that, while the Bellman-Ford algorithm is good for parallelization and raw traversal throughput, its highly redundant edge traversal leads to a poorer overall performance for SSSP.

In this section, we present SPLAG, an SSSP accelerator for power-law graphs [22]. SPLAG uses a coarse-grained priority queue (CGPQ) to manage the active vertices in the SSSP problem. The CGPQ organizes active vertices in chunks, stores them in the external memory, and orchestrates the chunks with an on-chip priority queue. SPLAG also employs a customized vertex cache (CVC) with application-specific push and pop operations, which reduces both on-chip and off-chip memory traffic. As a cache, the CVC leverages the asynchronous memory-mapped memory interface (Section 7.2) extensively to hide the memory latency.

8.1.1 The SPLAG Accelerator

SPLAG aims to enable high-throughput and work-efficient SSSP queries for large-scale power-law graphs. This is achieved using the architecture shown in Figure 8.2. The whole SPLAG accelerator is composed of three major components:

- The *coarse-grained priority queue (CGPQ)* implements a high-throughput bucket-based priority queue that is scalable to a large capacity by buffering active vertices on-chip and storing excessive vertices in the off-chip *spill memory* as fixed-size chunks. Section 8.1.3 will provide more details about the CGPQ.
- The *customized vertex cache (CVC)* provides high-throughput access to the vertex data, which are initially stored in the off-chip *vertex memory*. Unlike a standard cache with read and write interfaces, the CVC provides application-specific interfaces for *updating* vertices and *filtering* redundant updates. Section 8.1.4 will review the internals of the CVC.

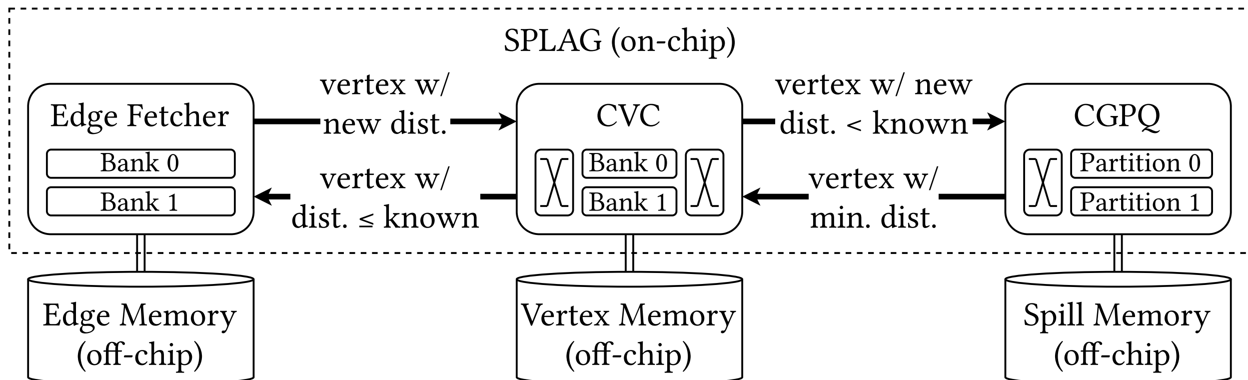


Figure 8.2: Architecture overview of the SPLAG accelerator.

- The *edge fetcher (EF)* traverses neighbors of an active vertex and calculates the new tentative distance. The off-chip *edge memory* stores the edge list in the compressed sparse row (CSR) format. Section 8.1.5 will discuss the edge fetcher.

To enable concurrent processing, we partition all the three major components internally. We use multi-stage switch networks [111] to improve the clock frequency without sacrificing the throughput [24] when all-to-all concurrent communication is required. Besides the three major components, the SPLAG accelerator also contains a dispatcher responsible for injecting the first active vertex, controlling program termination, and collecting statistics. The host program initializes the vertex and edge memory.

8.1.2 The SPLAG Algorithm

The SPLAG architecture implements a variant of Dijkstra's algorithm, shown in Algorithm 1. The algorithm is designed to expose as much parallelism as possible while minimizing the amount of work. This algorithm exploits two levels of parallelism by ① relaxing edges from multiple active vertices at the same time (Line 4 in Algorithm 1), and ② relaxing multiple edges of the same active vertex at the same time (Line 7 in Algorithm 1). Moreover, SPLAG executes the algorithm asynchronously, which means the next iteration of the outer loop (Line 4) can start before the previous one finishes, avoiding load balancing caused by skewed degree distribution.

Algorithm 1 SPLAG's variant of Dijkstra's algorithm.

Require: A graph $G = (V, E)$ and $root \in V$

Ensure: $vertices$ represent the shortest-path tree from $root$

```
1:  vertices = [{dist = ∞, parent = null }, ···]
2:  vertices[root] = [{dist = 0, parent = root}]
3:  queue = [{id = root, dist = 0, parent = root}]
4:  while not queue.empty() in parallel do
5:      u = queue.pop()                                        ▷ CGPQ
6:      if u.dist ≤ vertices[u.id].dist then                   ▷ CVC
7:          for all e = u.id → vid ∈ E in parallel do          ▷ Edge Fetcher
8:              if vid ≠ u.parent then                         ▷ Edge Fetcher
9:                  d = u.dist + e.weight                      ▷ Edge Fetcher
10:                 if d < vertices[vid].dist then[1]          ▷ CVC
11:                     vertices[vid] = {dist = d, parent = u.id}          ▷ CVC
12:                     queue.push({id = vid, dist = d, parent = u.id})    ▷ CGPQ
```

This, however, makes it possible to terminate the program prematurely because the *queue* may be temporarily empty before active vertices are pushed to the *queue* in Line 12. To solve this problem, we delay the program termination by a short, fixed amount of clock cycles to make sure any in-progress operation has been completed. This delay period is chosen based on the latency of memory accesses and depth of pipelines, and is long enough for any in-progress operation to complete.

Highly parallel execution of Dijkstra's algorithm may lead to highly redundant amount of work. That is, the number of edge traversal may be greater than the number of edges in the connected component. SPLAG reduces the amount of work using the conditional statement shown in Line 6 of Algorithm 1. It can filter out vertices that are updated many times. For example, for

[1]Line 10 and Line 11 must be atomic. The CVC takes care of this in SPLAG.

an SSSP query from root vertex A on the graph shown in Figure 8.3, vertex A generates a path to D with a tentative distance of 8 and vertex B generates a path to D with a tentative distance of 3+2=5. Without Line 6 in Algorithm 1, neighbors of vertex D will be traversed twice because D will be popped twice in Line 5 with two different tentative distances. With Line 6 and the priority queue, vertex D with the smaller tentative distance 5 will be popped first, and the second pop will be filtered out because a smaller distance is already known.

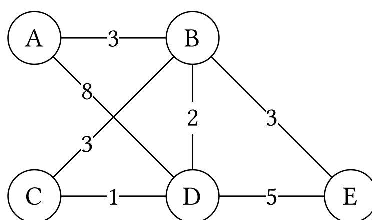


Figure 8.3: A graph with 5 vertices and 7 edges.

To further reduce redundant edge traversal, SPLAG applies another optimization named *never-look-back*. Noticing that a vertex always has a smaller distance than its children in the SSSP tree, SPLAG skips the parent of a vertex v when it traverses the neighbors of v. For example, for an SSSP query from root vertex A in Figure 8.3, A generates a path to B with tentative distance 3 and parent A. When SPLAG traverses neighbors of B, it will skip A and only traverse C, D, and E, since A is the parent of B, and we know A must already have a smaller distance than B.

8.1.3 The Coarse-Grained Priority Queue

A high-throughput and work-efficient SSSP accelerator for power-law graphs requires high-throughput priority-order graph traversal. Therefore, there are two design objectives for the priority queue: ① the priority queue must have a **large capacity** and utilize off-chip memory efficiently, and ② the priority queue must support **high throughput** push and pop operations. In this section, we present our solution named the coarse-grained priority queue (CGPQ). Noticing that a strict priority queue exposes too little parallelism and is not necessary for correctness, we take a coarse-grained *bucket*-based approach to achieve the two design objectives. Using a

pre-selected Δ, we can divide the active vertices into many buckets based on the distance from the root, and, e.g., store a vertex with tentative distance d in bucket $\left[\frac{d}{\Delta}\right]$. Vertices in the same bucket are considered to have the same priority and can be accessed in simple first-in-first-out (FIFO) order.

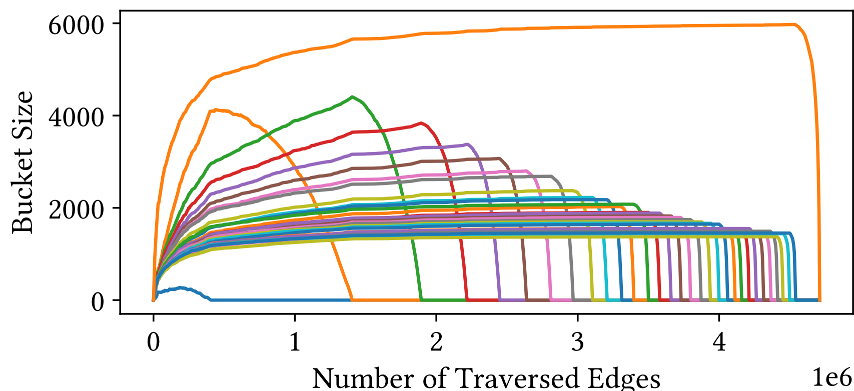


Figure 8.4: Sizes of 32 buckets as edges are traversed in the g500-15 dataset. Each line represents a bucket.

While it seems trivial to implement such a simple bucket-based CGPQ, the dynamic nature of the SSSP problem actually imposes significant challenges: the maximum size of each bucket is unknown before the program execution. While we can pessimistically reserve consecutive memory space for each bucket, doing so will likely result in a significant waste of memory since the overall utilization of memory would be low, which limits the scalability in terms of capacity. Figure 8.4 shows an example of how the sizes of 32 buckets change as edges are traversed. We can see that different buckets are utilized differently. If we reserve memory based on the maximum size of all the buckets, 63% of the reserved memory will be unnecessary. In fact, we must reserve even more because we must account for the worse case among all SSSP queries on all datasets. To avoid such memory waste, one can employ a linked list to allocate memory space dynamically, but such a data structure not only has the storage overhead for the node pointers, but also is slow due to random accesses. A commonly used data structure that achieves a compromise between a fixed-size array and a linked list is often called a double-ended queue (*deque*), which is a linked

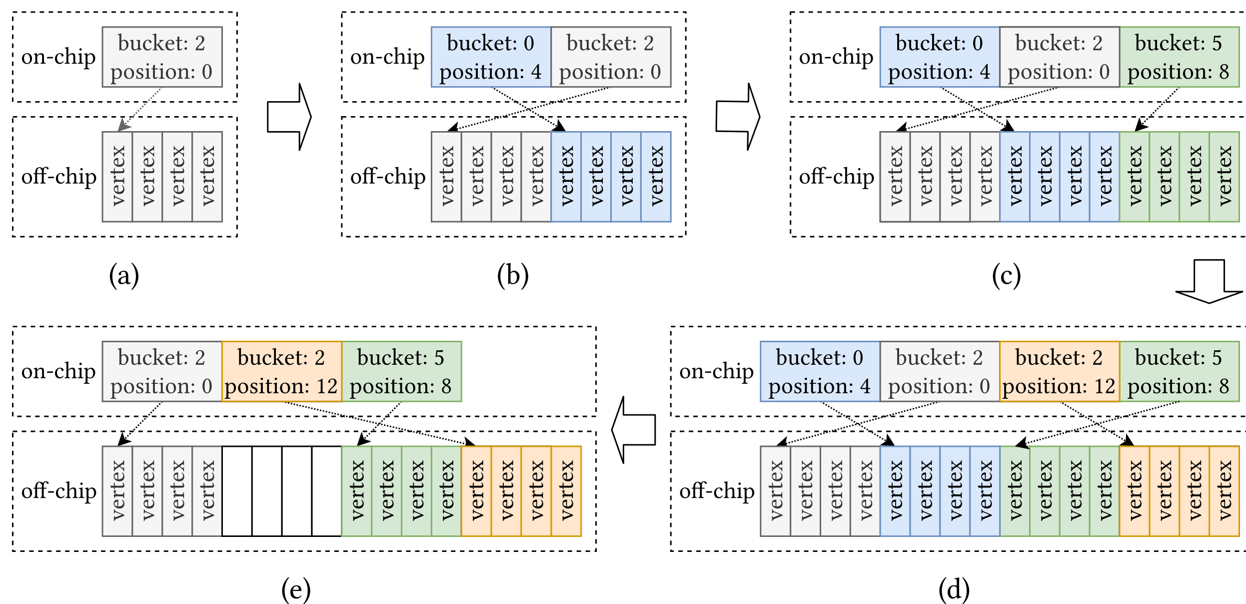


Figure 8.5: An example of the CGPQ orchestrating chunks of vertices. (a) Initially the CGPQ contains one chunk of four vertices stored off-chip and its reference stored on-chip. The on-chip reference stores the bucket number and a pointer to the off-chip memory position. (b) Another chunk of four vertices is added. The new chunk belongs to Bucket 0 and thus has higher priority. Therefore, it is stored on-chip at a position with a higher priority. The on-chip reference of the old chunk is moved to a position with a lower priority. Meanwhile, the off-chip memory only needs to append the newly added vertices without moving existing ones. (c) Another chunk for Bucket 5 is added. (d) There can be multiple chunks for the same bucket. (e) The chunk with the highest priority is removed. The chunk reference is popped from the on-chip priority queue and the pointer is used to read the vertices from the off-chip memory. On-chip chunk references are reorganized to maintain the priority queue structure while off-chip data are not moved.

list of fixed-size arrays. The use of linked lists, however, are still inefficient for implementation on FPGA.

The CGPQ is inspired by the deque data structure. Similar to a deque, the CGPQ manages off-chip active vertices in a unit of fixed-size *chunks*. Unlike a deque, the CGPQ manages the position and priority of the chunks with an on-chip priority queue, instead of linked lists. Figure 8.5

demonstrates how the on-chip *chunk priority queue* (CPQ) orchestrates the off-chip memory accesses to enforce the priority ordering of vertices. While the example shows 4-vertex chunks, in practice, the chunks are typically hundreds or thousands of vertices large to make sure the on-chip priority queue does not overflow. Therefore, the off-chip memory is always accessed in large chunk of vertices, which guarantees high memory bandwidth utilization. As such, the CGPQ can scale to a large capacity without a significant waste of the memory space or bandwidth.

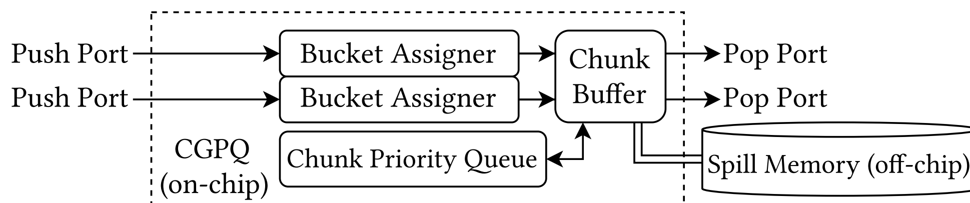


Figure 8.6: A CGPQ with two push ports and two pop ports. The number of ports can be different for push and pop and is larger than two in the actual design (Table 8.3 on page 155).

Figure 8.6 shows the architectural overview of a CGPQ with two push ports and two pop ports, which allows two vertices to be pushed and two vertices to be popped in parallel. Each input vertex will be assigned a bucket by the *bucket assigner* using a pre-selected Δ. The CGPQ then buffers on-chip at least one chunk of active vertices for each bucket, enabling high-throughput push and pop operations. Section 8.1.3.1 discusses more details about the *chunk buffer* and its two-level partitioning mechanism, which further enables concurrent operations and makes it possible to achieve our high-throughput design objective. When the buffer is (almost) full, vertices will be offloaded to the off-chip *spill memory* as a whole chunk. The *chunk priority queue* orchestrates the chunks between the on-chip chunk buffer and the off-chip spill memory, as demonstrated in Figure 8.5. Section 8.1.3.3 further discusses how we dynamically and collaboratively schedule the off-chip operations together with the on-chip push and pop operations. Such dynamic management allows the memory space to be used in a compact way, making it possible to achieve our large-capacity design objective.

8.1.3.1 The Chunk Buffer

The chunk buffer is the key to achieving highly concurrent push and pop throughput while supporting large queue capacity. As a priority queue, the two basic operations are *push* and *pop*. To achieve the large-capacity design objective, the chunk buffer must additionally support the *spill* and *refill* operations to store excessive vertices in the off-chip *spill memory*. The spill operation moves a chunk of vertices from the on-chip chunk buffer to the off-chip spill memory. The refill operation moves a chunk of vertices from the off-chip spill memory back to refill the on-chip chunk buffer. To achieve the high-throughput design objective, all four operations must be able to parallelize. This is done as follows using the terminologies summarized in Table 8.1.

Table 8.1: Terminologies used in the chunk buffer.

Term	**Meaning**
Chunk Buffer	The chunk buffer is part of the CGPQ and buffers vertices on-chip.
Bucket Partition	Vertices in the chunk buffer can be divided into many bucket partitions according to their bucket IDs modulo the bucket partition count. Different bucket partitions are implemented using separate hardware resources and therefore can be accessed in parallel.
Bucket Buffer	Vertices in the chunk buffer can be divided into many bucket buffers according to their bucket ID. Different bucket buffers belong to the same bucket partitions if they have the same bucket ID modulo the bucket partition count.
URAM Bank	Each bucket partition is implemented using several URAM banks. Bucket buffers in the same bucket partition cannot be accessed in parallel because they are stored in the same URAM banks (Figure 8.8).

To support concurrent push operations, the chunk buffer is internally partitioned into many *bucket partitions* so that different buckets can be accessed in parallel. This is called *inter-bucket parallelism*. Figure 8.7 shows the architecture of the chunk buffer with two bucket partitions. In

this example, vertices in Bucket 0, 2, 4, ... belongs to Bucket Partition 0, and vertices in Bucket 1, 3, 5, ... belongs to Bucket Partition 1. Since each vertex may belong to any bucket partition, this requires an all-to-all communication pattern. After each incoming active vertex is assigned a bucket and is sent to the chunk buffer, it will first be routed through the switch network based on its bucket partition ID.

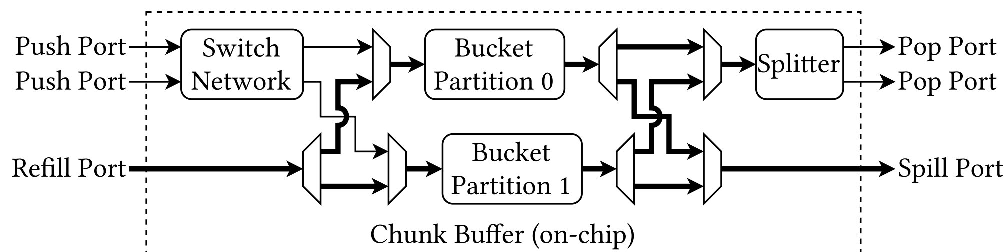


Figure 8.7: The chunk buffer in Figure 8.6. Data paths in bold transfer multiple data elements in lockstep.

Each bucket has its own on-chip storage in the chunk buffer called the *bucket buffer (BB)*. Each BB is accessed in FIFO order as a circular buffer. The chunk buffer maintains the write and read pointers of each BB. Figure 8.8 shows the data layout of the chunk buffer in Figure 8.7. The numbers in brackets are the array indices of each vertex in each BB. For example, the first three active vertices assigned to Bucket 0 will be written to memory positions `[0]`, `[1]`, and `[2]` in BB 0 (Figure 8.8) in three clock cycles, which can happen in parallel when Bucket 1 or Bucket 3 (but not Bucket 2 due to bank conflict with Bucket 0) is being written.

In Figure 8.7, only one vertex may be routed to each bucket partition. While each bucket partition can in fact consume multiple vertices in each clock cycle, we do not exploit the parallelism to push vertices with each bucket partition. The rationale is as follows. On the one hand, we observe that the incoming vertices are roughly evenly distributed among all buckets in the beginning of execution where the push operations are the most intensive. Figure 8.4 shows such an example: almost all bucket sizes increase rapidly before the bucket sizes hit ~500. The switch network can further absorb temporary unbalances. Therefore, pushing only one vertex to each bucket partition does not impose a significant throughput decrease. On the other hand,

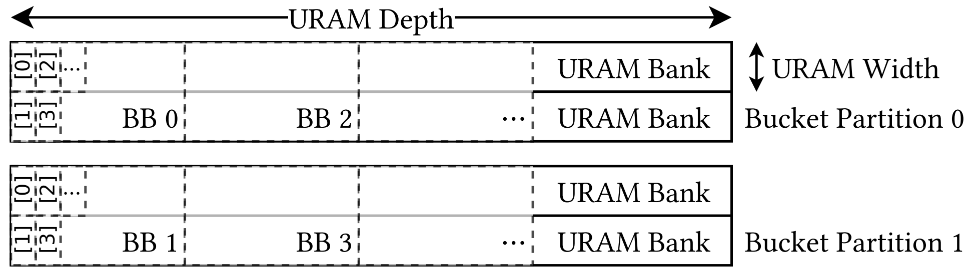


Figure 8.8: Data layout of the chunk buffer in Figure 8.7. Terminologies are summarized in Table 8.1. Each push port requires a bucket partition and each pop port requires a URAM bank, so there are two bucket partitions and each bucket partition has two banks. Each bucket buffer (BB) is used as a circular buffer. The numbers in brackets are the array indices of the vertices in each BB.

we observe that unlike pop operations (which we will discuss later), push operations cannot be coalesced and aligned, since we can never know when/if the next vertex to the same bucket will arrive. Therefore, each incoming vertex may fall in any bank in the bucket partition. Unlike the routing problem among different bucket partitions whose destination is determined solely by its distance and thus can use a multi-stage switch network, the bank ID to which a vertex shall be written is determined by the runtime conditions in the buffer. Restricting the push rate to each bucket partition not only reduces resource utilization, but also removes a potential critical path in the whole accelerator. As such, we only exploit inter-bucket parallelism for push operations.

For pop operations, since we only pop from a single non-empty bucket with the smallest distance, the inter-bucket parallelism among different bucket partitions cannot be exploited. Therefore, we further divide each bucket partition cyclically into *URAM banks* so that multiple vertices can be accessed at the same time. This is called the *intra-bucket parallelism*. With the intra-bucket parallelism, we can then pop multiple vertices from the same bucket in a single clock cycle. These vertices then go through coalesced data paths in lockstep and are eventually split into individual data paths. In Figure 8.7 and Figure 8.8, since there are two pop ports, each bucket partition is divided into two banks and the coalesced data paths are two-element-wide to match the data rate

of pop operations.

Note that a bucket may have fewer valid vertices than the intra-bucket parallel factor. In such cases, *null vertices* padding will be filled in the coalesced data paths. The *splitter* in Figure 8.7 detects and removes the null vertices when sending the coalesced vertices into individual pop ports. To simplify the logic to determine the validity of popped vertices, we require that the pop operations are always aligned; null vertices will be written to the BB in case a pop operation performs a partial read. For example, let there be three valid vertices in Bucket 0 in Figure 8.8 stored in `[0]`, `[1]`, and `[2]`. The first pop operation will read two vertices from `[0]` and `[1]` respectively, which are aligned to the intra-bucket parallel factor 2. The second pop operation will only read one valid vertex from `[2]`. This is not an aligned operation. If we allow such unaligned operations and later a new vertex is written to `[3]` in Bucket 0, we will have to be able to read one single valid vertex from `[3]` while marking vertices from other banks null. This complicates the design. Instead, in case of unaligned operations, we will force alignment and adjust the write pointer in addition to the read pointer to fill in the unaligned locations. Using the same example above, when the second pop operation reads the vertex from `[2]`, it will move both the read pointer and the write pointer to `[4]` so that the next incoming vertex will be stored in `[4]` instead of `[3]`. Note that this alignment enforcement does not sacrifice the maximum capacity of each circular buffer. As such, unaligned pop operations will only insert null vertices in higher locations, which simplifies the pop operation logic without affecting other operations.

Intra-bucket parallelism enables not only concurrent pop operations, but also faster spill/refill operations by reading from/writing to all URAM banks in each bucket partition. Figure 8.7 shows the data paths for the spill and refill operations. Section 8.1.3.3 will discuss how the four operations are scheduled. Note that we cannot guarantee each spill/refill operation is aligned. For example, in Figure 8.8, the read pointer may point to position `[1]` when a spill operation is scheduled, which means the spill operation should read `[1]` and `[2]` in the first clock cycle. This is why we have to partition each bucket partition into individual memory banks instead of reshaping the data structure to use a single-bank memory with a wider width.

8.1.3.2 The Chunk Priority Queue

The chunk buffer is on-chip and limited in size. To accomplish the large capacity design objective, when a bucket buffer (BB) is almost full, it will *spill* to the off-chip memory. When spilling happens, a *chunk reference* will be created with the off-chip memory pointer and the bucket associated with that chunk. This chunk reference will be pushed into an on-chip *chunk priority queue* (CPQ) so that when the BB has enough space, the off-chip chunks can be *refilled* in priority order. Figure 8.5 shows an example of spilling and refilling. Since each chunk contains many vertices, the capacity of the CPQ can be much smaller than the whole CGPQ, and the CPQ can be on-chip only. Since the off-chip access has a long latency (> 100 ns [26]), a regular binary heap suffices for the CPQ. Other on-chip priority queue data structures such as the systolic priority queue or pipelined heap require more memory banks and are thus less efficient, so we do not use those.

8.1.3.3 Scheduling the Operations

The potential bank conflicts and multi-cycle operation of spilling and refilling bring challenges to schedule the four operations correctly without deadlock. Moreover, the spill memory is shared by all bucket partitions to maximize the utilization of the external memory. To avoid deadlock, the general scheduling rules are:

1. always make sure multi-cycle operations, i.e., spilling and refilling, can finish without indefinite stalling. This not only simplifies the inter-operation dependency, but also helps to improve off-chip memory utilization since memory requests will not be stalled by the chunk buffer;

2. prioritize push operations over pop operations since pop operations may generate more push operations;

3. each bucket partition only has one read port and one write port.

Details for scheduling each operation are as follows.

The push operation is scheduled on a bucket partition when an incoming vertex is available on the push port, unless:

1. The write port is occupied by an active refill operation.
2. There is insufficient buffer space for the target bucket. This includes the case when the BB is full and the case when future refill operations exhaust the available space. For example, let each chunk contain 4 vertices and the BB can hold up to 8 vertices. A refill operation is scheduled when the BB only has 2 vertices. Push operations can be scheduled before refilling data are fetched from the off-chip memory (which takes many clock cycles), until there are 4 vertices in the BB. We will not schedule another push operation when the BB contains 4 vertices since if we do, the refill operation would stall indefinitely when no pop operation is scheduled and the BB does not have sufficient space for the last vertex.

The pop operation is scheduled for the non-empty BB with the highest priority, unless:

1. The output pop port is full.
2. The read port is occupied by an active spill operation.
3. There are insufficient vertices.
4. The pop operation is unaligned and the write port is used by a push or refill operation.

A BB is selected for *spilling* if its size exceeds a pre-defined threshold (e.g., 3/4 BB capacity) and there is no spilling or refilling already scheduled (which occupies the off-chip memory). If multiple BBs are almost full, we start spilling the one with the lowest priority. Once a BB is scheduled for spilling, the whole chunk must be moved to the off-chip memory, which takes multiple clock cycles. That BB will continue the spilling operation in the following clock cycles unless the memory channel is busy.

The top bucket in the CPQ is selected for *refilling* if its size is below a pre-defined threshold (e.g., 1/4 BB capacity), and there is no spilling already scheduled. Since there is a long latency

between the off-chip memory read request and the data response, we allow at most one refilling operation to be scheduled while another one is in process, which can help to hide this long latency. Once scheduled, the refilling operation will continue in the follow clock cycles unless the memory channel is not ready with the appropriate data.

Given the above scheduling mechanism, we have the following theorem:

Theorem 2. *The CGPQ operation scheduling is deadlock-free if the spill memory is sufficiently large and deadlock-free.*

Proof. Spilling and refilling operations by-construction will not block indefinitely as long as the spill memory is deadlock-free. Moreover, given a finite number of push operations and properly chosen thresholds, spilling and refilling operations will be scheduled for finite times, all of which will eventually complete. Therefore, assuming the spill memory is sufficiently large, spill operations will eventually unblock push operations blocked by insufficient buffer space, so push operations will not block indefinitely either. As a result, the rest of the accelerator is always able to make progress, which will eventually unblock pop operations blocked by full output. This means none of the four operations will block indefinitely. By definition, the scheduling is deadlock-free. □

8.1.4 The Customized Vertex Cache

With a high-throughput and large-capacity CGPQ, we still need a carefully designed accelerator that can keep up with the throughput. The priority-order graph traversal generates extensive random memory accesses that are infeasible to reorder for better off-chip bandwidth utilization, making it even more challenging to create a fast SSSP accelerator. We could employ a classic memory cache to mitigate the random accesses on the vertex data, but it would produce lower quality of results due to its application-agnostic nature. Noticing that the tentative distance of each vertex monotonically decreases, we can take advantage of this property and simplify the accelerator design. In SPLAG, we create the customized vertex cache (CVC) to help

1. reduce the off-chip memory traffic by caching vertex data on-chip, and
2. reduce on-chip memory requests by taking advantages of the fact that the tentative distance of each vertex is monotonically decreasing.

The CVC provides two basic operations:

1. The *updating* operation consumes as input a vertex with a new distance and its corresponding parent vertex ID. The CVC updates the tentative distance and the tentative parent of a vertex if and only if the input distance is smaller than the existing value. If the update happens, the updated vertex is pushed to the CGPQ.
2. The *filtering* operation takes as input a vertex popped from the CGPQ. The CVC compares the tentative distance of the input and the existing value and checks if the input is "stale", i.e., its tentative distance has been updated to a smaller value. Only if the input vertex is not stale, will the CVC forward the input from the CGPQ to the edge fetcher to traverse its neighbors.

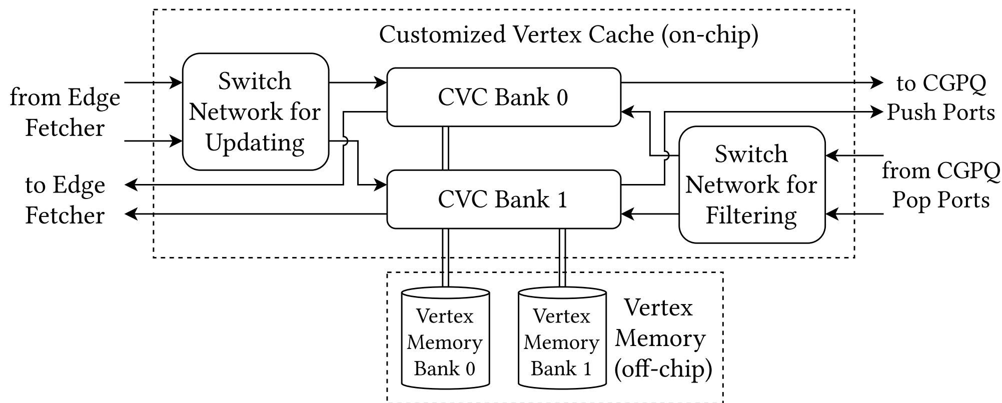


Figure 8.9: A customized vertex cache with two banks. Both the on-chip and off-chip memory are partitioned into banks. Vertices are cyclically assigned to each bank. The two switch networks route requests based on the bank ID.

Figure 8.9 shows the architecture of a CVC. For high-throughput memory accesses, the vertices are cyclically partitioned into multiple banks. Each CVC bank has two pairs of ports, one

pair for updating and another for filtering. The updating ports are connected to the edge fetcher, which is responsible for generating vertices with new tentative distances. Since each vertex may have neighbors in any CVC bank, a switch network is used to route the updating inputs to the correct bank. Similarly, the filtering ports are connected to the CGPQ and a switch network is used to route the filtering inputs to the correct bank.

The CVC is fully pipelined. Each CVC bank can serve one request per cycle on hit. The initiation interval for miss requests is two, because it takes one cycle to send and another cycle to receive the off-chip memory request. The memory requests are pipelined, and their long latency can be hidden by overlapping them with each other. Each CVC bank implements a direct-mapped write-back cache. We do not employ a set-associative cache since the hit rate improvement does not make up the frequency degradation caused by its complexity. Each cache entry keeps a `dirty` bit to indicate whether its content should be written back on cache miss or program termination. Each entry also keeps a `writing` bit to indicate that its content is being written back, and another dirty cache miss must stall until the write finishes.

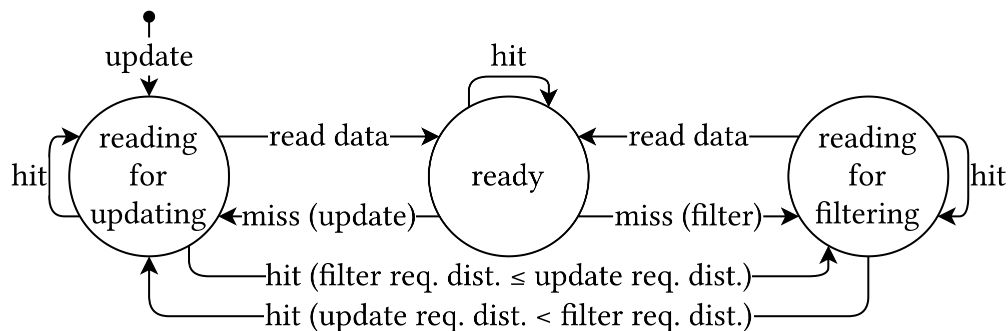


Figure 8.10: The finite-state machine for memory reads in the CVC. *Hit* means the requested cache line has the same vertex ID as the requested vertex. *Miss* means the opposite of hit. The initial invalid state can only transfer to reading for updating because each vertex cannot be filtered before being updated first. Dirty and writing states can be managed independent of the states for reading and are not included in the figure. Miss on reading will stall the request until the request until the entry is no longer reading, so there is no state transition from reading states on miss.

Coordinating memory reads from DRAM is more complicated than writes, because both update requests and filter requests can generate DRAM reads. On cache miss, we store the tentative distance and parent in the incoming vertex as the tentatively known data, and do not generate the output until the off-chip data are available. The CVC treats off-chip read caused by updating and filtering differently when they arrive, therefore each cache entry in the CVC has two different reading states. Figure 8.10 shows the finite-state machine of a CVC entry for memory reads. When an off-chip read for updating finishes, the CVC compares the distances and generates an update if the incoming vertex has a smaller distance. If another updating request arrives for the same vertex before the read data arrive, the cache entry is updated on-chip to keep only the smallest tentative distance. This application-specific optimization reduces on-chip memory traffic while hiding off-chip memory latency. When an off-chip read for filtering finishes, the CVC compares the distances and discards the request if the incoming vertex has a greater distance (which means the popped vertex is "stale"). If another filtering request arrives for the same vertex before the read data arrive, the cache entry is updated on-chip to keep only the smallest tentative distance. If an updating request arrives when an entry is reading for a filtering request, the CVC compares the distances and marks the purpose of the reading updating if the updating request has a smaller distance. This is because if the updating request has a smaller distance, the filtering request would become stale and should be discarded. Otherwise, the updating request is not generating an update and the filtering request should continue. Figure 8.15 on page 158 shows the statistics of requests discarded by the CVC. Similarly, if a filtering request arrives when an entry is reading for an updating request, the CVC compares the distances and marks the purpose of the reading filtering if the filtering request's tentative distance is smaller than or equal to the updating request.

8.1.5 The Edge Fetcher

The edge fetcher traverses neighbors of active vertices filtered by the CVC and calculates the new tentative distances of the neighbors. Figure 8.11 shows the architecture of the edge fetcher. The

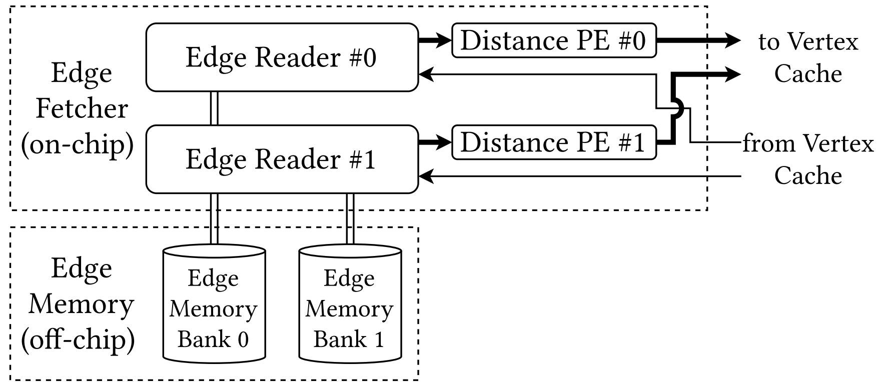


Figure 8.11: Edge fetcher with two banks. Each bank stores edges whose vertices in the corresponding vertex partition. Bold lines are coalesced data paths that transfer multiple vertices in lockstep.

edge fetcher exploits two levels of parallelism:

1. the edges are partitioned into multiple banks based on the source vertex ID so that neighbors of different vertices are traversed at the same time, and

2. the edges are coalesced into wide vectors so that multiple neighbors are traversed at the same time.

The edge fetcher is fully pipelined without turnaround time for different input vertices, meaning if there are no bubbles in the input vertices, the edge fetcher will not insert any bubbles to the output (unless the off-chip memory does not keep up with the data rate).

8.1.6 Evaluation

We evaluate SPLAG with both synthetic and real-world graph datasets. Table 8.2 shows the details of the datasets. All graphs are undirected. For each dataset, we sample 64 vertices that are connected to at least one other vertex and report the harmonic mean since the metrics are ratios. All experimental results are collected from on-board execution. Performance counters are inserted to the accelerator to collect the relevant metrics.

Table 8.2: Graph datasets evaluated on SPLAG.

Dataset	$\|V\|$	$\|E\|$	Maximum degree	Average degree	Source
amzn	2.1M	5.8M	12k	2.7	Amazon product ratings [131]
dblp	540k	15M	3.3k	28	DBLP Paper Coauthors [71]
digg	872k	4.0M	31k	4.5	Users from digg.com [148]
flickr	2.3M	33M	34k	14	Flicker users [129]
g500-N	2^N	2^{N+4}	$2^{0.6N+5}$	16	Graph 500 datasets [133]
hlwd-09	1.1M	58M	12k	50	Actor collaboration [148]
orkut	3.0M	106M	28k	36	Orkut social network [148]
rmat-21	2.1M	91M	214k	44	A Kronecker graph [116]
wiki	274k	2.9M	3.4k	11	Wiki article–word graph [148]
youtube	3.2M	12.2M	130k	3.8	YouTube users [130]

We implement SPLAG using TAPA [23] (Chapter 6 and Chapter 7), to leverage the convenient peeking interfaces, decoupled DRAM request and response interfaces, simplified host-kernel interface, and AutoBridge [76,78] floorplanning. Our implementation targets the Alveo U280 board with 32 high-bandwidth memory (HBM) channels. Table 8.3 summarizes the design parameters. We determine the design parameters as follows: to maximize the utilization of the switch networks, we only select powers of 2 for *#bank* and *#HBM*. We allocate as many *#HBM* as possible to CVC for its intensive random accesses, and evenly distribute the rest between EF and CGPQ. For CVC, *capacity/bank* maximizes the URAM utilization. For EF, *coalescing factor* matches *#bank* of CVC and EF. For CGPQ, *#port* matches *#bank* of CVC. *CPQ capacity* and *#bucket* are maximized without exceeding the timing critical path in CVC. *Chunk size* matches the capacity of HBM and CPQ. *BB capacity* doubles the chunk size. *Spill (refill) threshold* is empirically chosen as ¾ (¼) of *BB capacity*.

We use Vitis 2021.1 for hardware implementation. The post-implementation reports suggest

that the whole accelerator, including the Vitis shell platform, utilizes 75% CLBs, 6.3% DSPs, 14% BRAMs, and 83% URAMs with a 45W power budget (including the HBM). There are 162840 CLBs, 9024 DSPs, 2016 BRAMs, and 960 URAMs available in total. The accelerator is clocked at 130MHz with critical paths caused by the extensive usage of URAMs in the CVC. The CPU baseline is a multi-thread Δ-stepping implementation written in the latest Galois [107]. The server has two Xeon Gold 6244 CPUs, which have 32 threads in total running at 4.4 GHz.

Table 8.3: Design parameters of the SPLAG accelerator.

Component	Parameter	Value	Rationale
CVC	#HBM	16	Maximum power of 2 & < 32 HBMs in total
	#Bank	16	Matches #HBM
	Capacity/Bank	64k	Maximizes the URAM utilization
EF	#HBM	8	= (32 - 16) / 2
	#Bank	8	Matches #HBM
	Coalescing Factor	2	Matches #bank of CVC / #bank of EF
CGPQ	#HBM	8	= (32 - 16) / 2
	#Push Port	16	Matches #bank of CVC
	#Pop Port	16	Matches #bank of CVC
	CPQ Capacity	256k	Maximized without exceeding critical path in CVC
	#Bucket	128	Maximized without exceeding critical path in CVC
	Chunk Size	1024	Matches the capacity of HBM and CPQ
	BB Capacity	2048	Doubles the chunk size
	Spill Threshold	1536	Empirically chose as ¾ of BB capacity
	Refill Threshold	512	Empirically chose as ¼ of BB capacity

Table 8.4: Post-implementation results of the SPLAG accelerator on U280.

	LUT	FF	DSP	BRAM	URAM	Power	Clock
SPLAG	48% / 621k	25% / 660k	6.3% / 564	14% / 548	83% / 800	45 W	130 MHz
CGPQ	277k	222k	528	2	288	—	—
CVC	121k	90k	0	0	512	—	—
Edge Fetcher	15k	17k	32	0	0	—	—
Dispatcher	5k	27k	0	2	0	—	—
Vitis Shell Platform	203k	303k	4	546	0	—	—
Total Available on U280	1303k	2607k	9024	4032	960	225 W	300 MHz

8.1.6.1 Evaluation of the CGPQ

Figure 8.12 shows the percentage of spilled vertices among all vertices pushed to the CGPQ. We can see that for large datasets, almost all active vertices are spilled to the off-chip memory. Moreover, the scaling from g500-15 to g500-22 matches the trend of active vertices shown in Figure 8.1 on page 135. This suggests that our CGPQ design has accomplished the large-capacity design objective.

Figure 8.13 shows the percentage of idling cycles of the CVC. Note that CVC idling can be caused by either empty CGPQ or insufficient pop throughput; the performance counters cannot tell the reason for idling. Moreover, the CVC never stalls because the CGPQ push port is full in any of the evaluations. <8% CVC idling and 0% CVC stalling caused by the CGPQ suggest that our CGPQ design has accomplished the high-throughput design objective.

8.1.6.2 Evaluation of the CVC

Figure 8.14 shows the CVC read and write hit rate. We found that the hit rate highly depends on the number of vertices of the dataset: the g500-N series show a clear dropping trend when vertex count increases, and larger datasets (e.g., amzn, flickr, orkut, youtube) tend to have lower hit rate

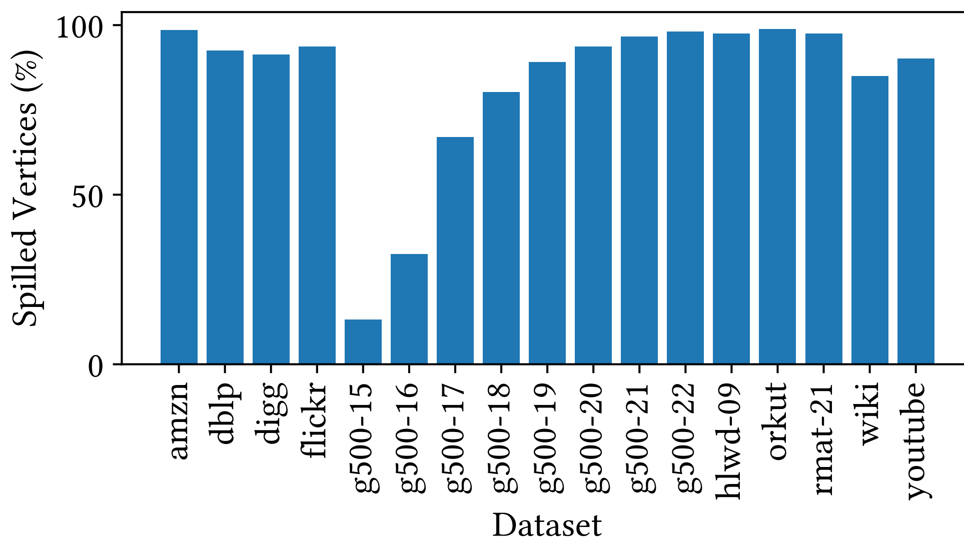


Figure 8.12: Percentage of spilled vertices among all vertices pushed to the CGPQ.

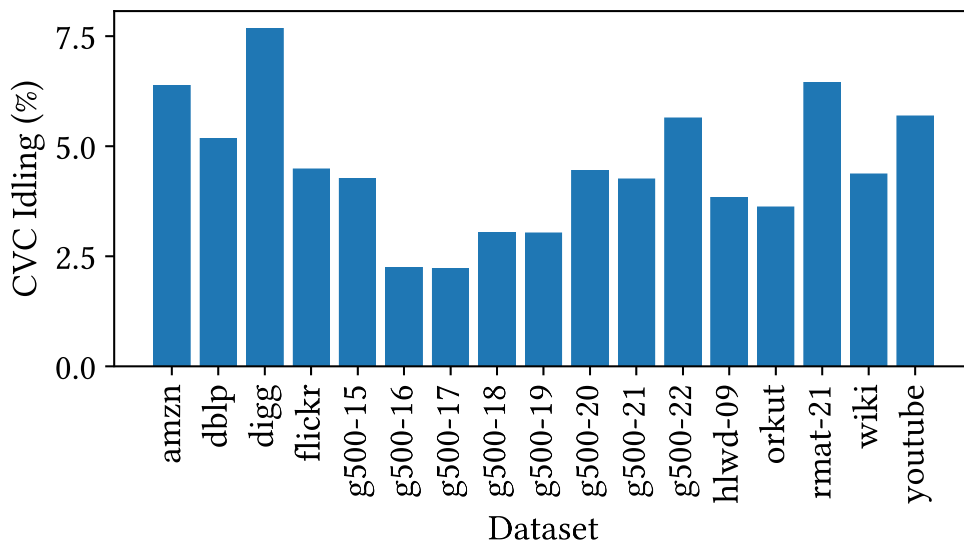


Figure 8.13: Percentage of CVC idling. Low idling suggests that the CGPQ can pop vertices with a high throughput.

in general. Nevertheless, even for the largest datasets, the read and write hit rate is still higher than 80% and 50%, indicating effective caching.

Figure 8.15 shows the percentage of traversed edges that generated an active vertex with a

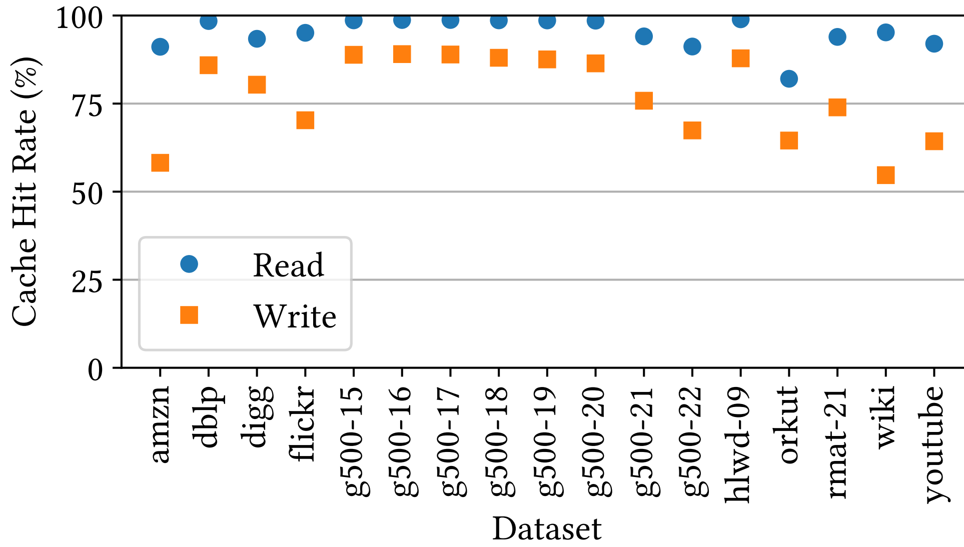


Figure 8.14: Read and write hit rate of the CVC.

new distance. We can see that CVC filtering is very effective in reducing redundant edge traversal.

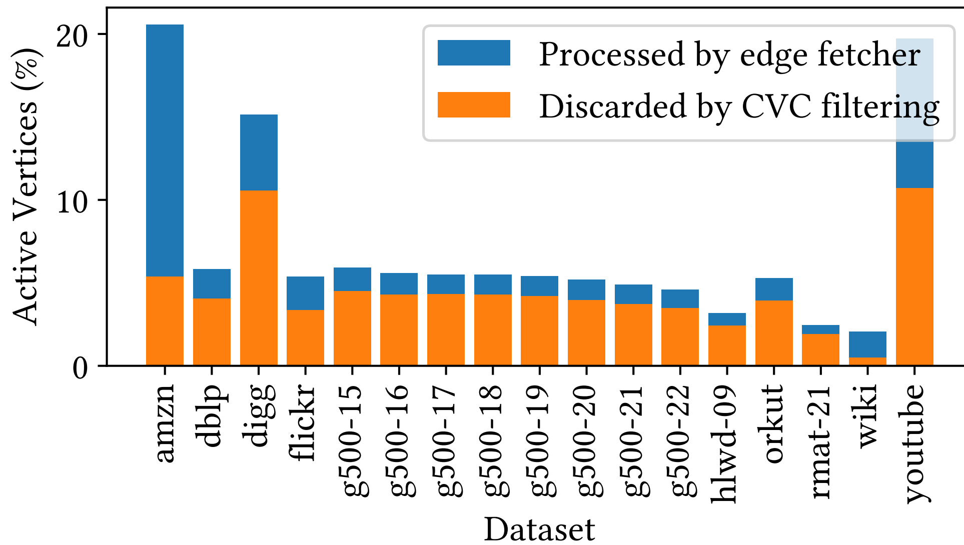


Figure 8.15: Percentage of active vertices among all traversed edges. The active vertices are either ① discarded by CVC filtering, or ② processed by the edge fetcher. The rest of traversed edges did not generate active vertices during CVC updating.

8.1.6.3 Overall Evaluation of SPLAG

Figure 8.16 shows the throughput achieved by SPLAG. The *traversal throughput* is defined as the number of traversed edges divided by the kernel execution time, which reflects the processing capability of the hardware but not the performance of the algorithm itself. The traversal throughput counts directed edges because only one direction can be traversed at a time. The *algorithm throughput* is defined as the number of *undirected* edges in the connected component of the root vertex divided by the kernel execution time, which measures the overall performance of the SSSP algorithm, including both the graph traversal throughput and the work efficiency. The algorithm throughput metric is used by the Graph 500 benchmark for data intensive applications [133]. We measured 504 MTEPS throughput under this metric using the g500-21 dataset, which could be ranked at the 14th position of the Graph 500 June 2021 SSSP list [120]. To the best of our knowledge, SPLAG is the first FPGA accelerator that can achieve such a ranking. The immediate preceding system on that list (at the 13th position) used an 8-node/128-core cluster to achieve 656 MTEPS throughput, while SPLAG works on a single FPGA board with only 45 W power budget. Beyond the Graph 500 datasets, the dblp dataset achieves the highest 763 MTEPS algorithm throughput.

Figure 8.17 further shows the work efficiency achieved by SPLAG. The work efficiency metric, *amount of work*, is normalized to the number of *directed* edges in the traversed connected component. Therefore, Dijkstra's algorithm generally achieves the amount of work of 1. Thanks to the never-look-back optimization (Section 8.1.2), SPLAG can even achieve < 1 amount of work for some datasets.

8.1.6.4 Comparison with Other SSSP Systems

Table 8.5 compares SPLAG against a multi-thread CPU baseline and three state-of-the-art graph accelerators. The multi-thread CPU Δ-stepping [127] implementation achieves better work efficiency than the Bellman-Ford accelerators, but is still less efficient than SPLAG due to its application-

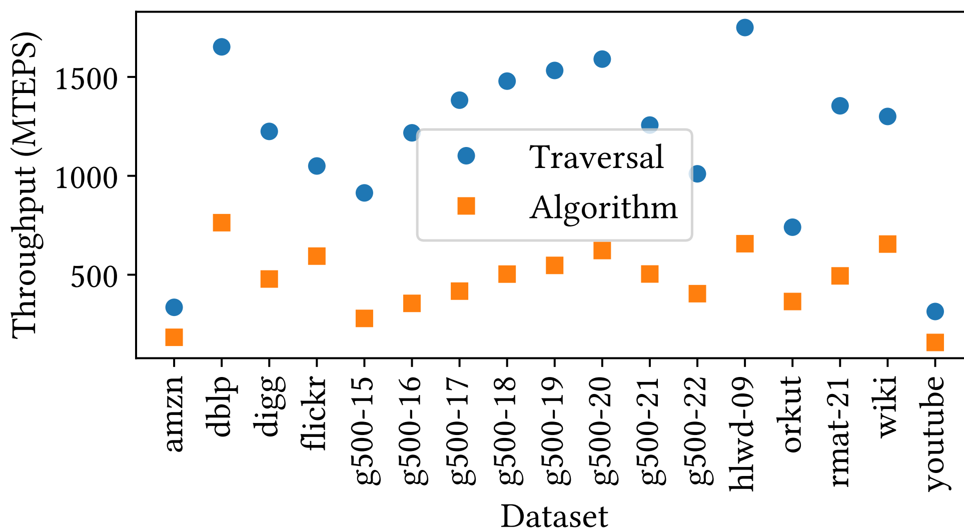


Figure 8.16: Throughput achieved by SPLAG. "Traversal" throughput measures the number of traversed edges over the kernel execution time. "Algorithm" throughput measures the number of *undirected* edges in the connected component over the kernel execution time.

agnostic memory system. Note that previous Dijkstra's algorithm accelerators were not evaluated using power-law graphs. All three FPGA accelerators implement the Bellman-Ford algorithm, which is work-inefficient. We use the performance numbers reported in each paper. Considering the fact that the previous works do not record the parent vertex ID in the vertex data while SPLAG does, we halve the traversal throughput of the previous works. While it is conceptually trivial to additionally record parent ID, doing so has a significant performance penalty, due to the increased attribute size. For HitGraph, adding parent ID doubles the size of each update, and thus halves the throughput of both the scatter phase and gather phase. For ThunderGP, doubling the vertex attribute size (`prop_t` or *read_size* in Formula (1) and (2) in [13]) halves the number of PEs, halving the throughput as well. For GraphLily, doubling the vertex attribute size halves the pack size (Section IV-A-(1) in [87]), thus halving the throughput, too. Since ThunderGP [13] and GraphLily [87] do not report the absolute execution time, we estimate the upper-bound of their algorithm throughput based on a CPU implementation of the Bellman-Ford algorithm. The CPU implementation applies push-based graph traversal and edges are traversed only if the ver-

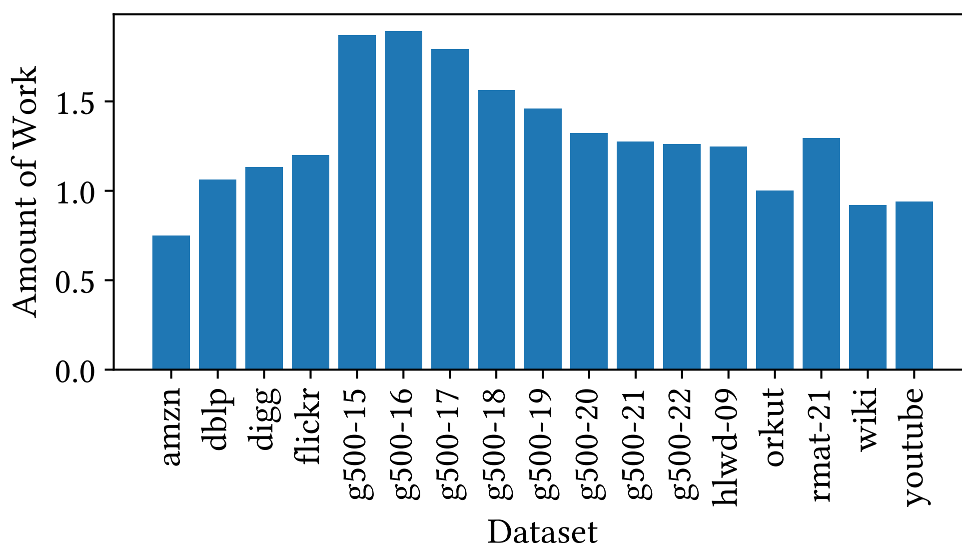


Figure 8.17: Normalized amount of work achieved by SPLAG. This is defined as the number of traversed edges divided by the number of *directed* edges in the connected component. Lower is better. Some benchmarks have < 1 amount of work because of the *never-look-back* optimization (Section 8.1.2).

tex is updated in the previous iteration. Due to lower parallelism, push-based traversal usually is only adopted when the graph traversal frontier is small [87], and pull-based traversal generates more redundant traversal. Therefore, our estimation gives a lower-bound of the number of traversed edges. For fair comparison with ThunderGP and HitGraph which use DDR-based FPGA, we ported SPLAG to an Alveo U250 board with 4 DDR channels. Still, SPLAG is at least 2.3× faster. The three FPGA baselines used for comparison are powerful general-purpose graph processing systems. Many graph algorithms (e.g., PageRank) are very well-accelerated by these systems, yet SPLAG is not capable of the same. However, while they can support some application-specific optimizations like pruning and early-termination for SSSP, further customization by SPLAG (esp. with efficient support of order-sensitive edge traversal) leads to better performance at the expense of some lose of generality.

Table 8.5: SPLAG compared against other SSSP systems.

Dataset	System	Throughput (MTEPS) Traversal	Throughput (MTEPS) Algorithm	SPLAG's Speedup
hlwd-09	Galois (Δ-stepping on 32-t 4.4 GHz CPU) [107, 127]	1229	211	**2.6×**
	GraphLily [87] (32 HBM)	4670	< 232	> **2.3×**
	SPLAG (32 HBM)	**1744**	**543**	1×
	ThunderGP [13] (4 DDR)	2454	< 122	> **2.6×**
	SPLAG (4 DDR)	756	315	1×
rmat-21	Galois (Δ-stepping on 32-t 4.4 GHz CPU) [107, 127]	930	254	**1.9×**
	GraphLily [87] (32 HBM)	2823	< 195	> **2.5×**
	SPLAG (32 HBM)	**1354**	**494**	1×
	HitGraph [181] (4 DDR)	2152	46.9	**4.9×**
	SPLAG (4 DDR)	**533**	**228**	1×

8.1.7 Using SPLAG for Neural Image Reconstruction

As discussed in Section 2.2.3, one of our application drivers for SPLAG is neural image reconstruction. In this section, we integrate SPLAG with Recut [124], a state-of-the-art concurrent framework for sparse neural reconstruction.

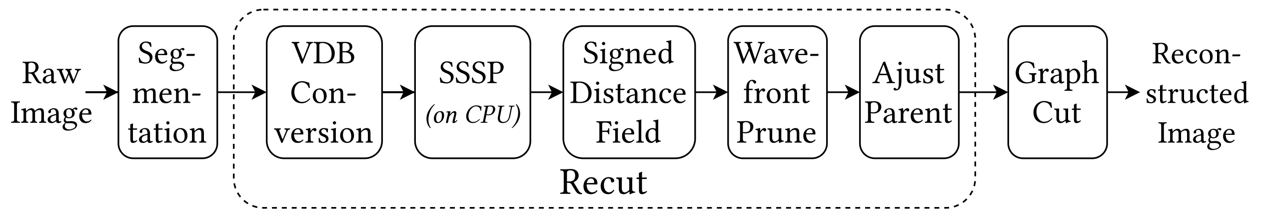


Figure 8.18: A complete neural image reconstruction pipeline with Recut [124].

In neural reconstruction, the SSSP algorithm is used to differentiate and identify entangled

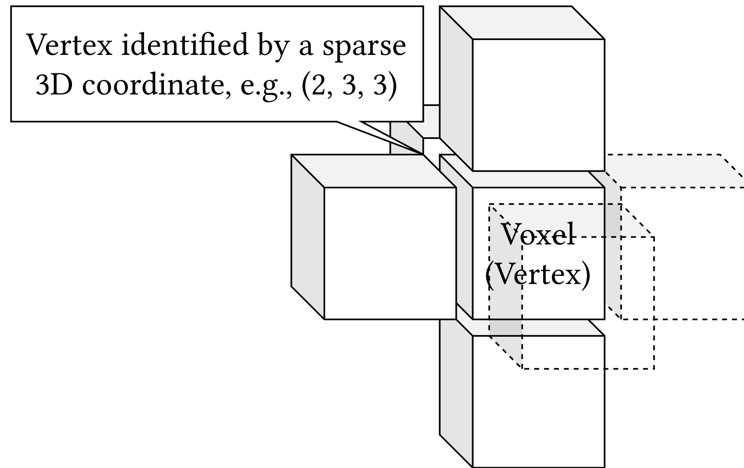


Figure 8.19: Sparse representation of a 3D neural image.

neurons from each other. The 3D neural image can be treated as a graph as follows: a voxel [2] that represents part of a neuron is considered a valid vertex. Neighbor voxels are also neighbors in the graph. The edge weight (distance) of an edge is determined by the average image intensity of the two vertices (voxel) of the edge. The cell bodies (somas) of the neurons can be easily identified, which will be set as the root vertices in the SSSP problem. Figure 8.19 shows an example of a 3D neural image, with five valid voxels (solid) and two invalid voxels (dotted). The center voxel has four neighbors.

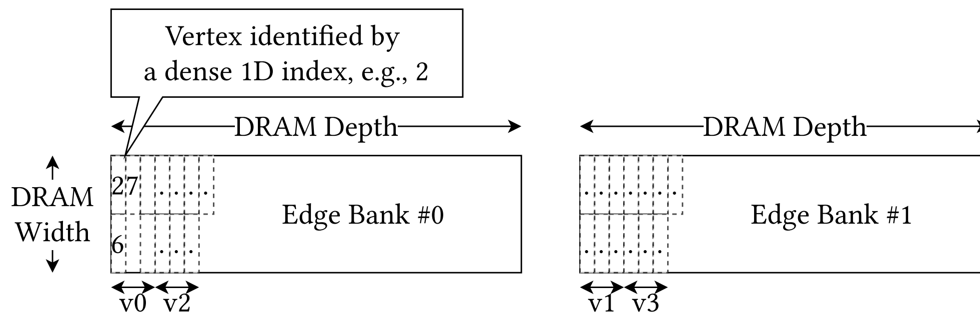


Figure 8.20: SPLAG edge data layout.

[2]The word *voxel* is analogous to *pixel*, with *vo* representing *volume* (instead of *picture*) and *el* representing *element* [67].

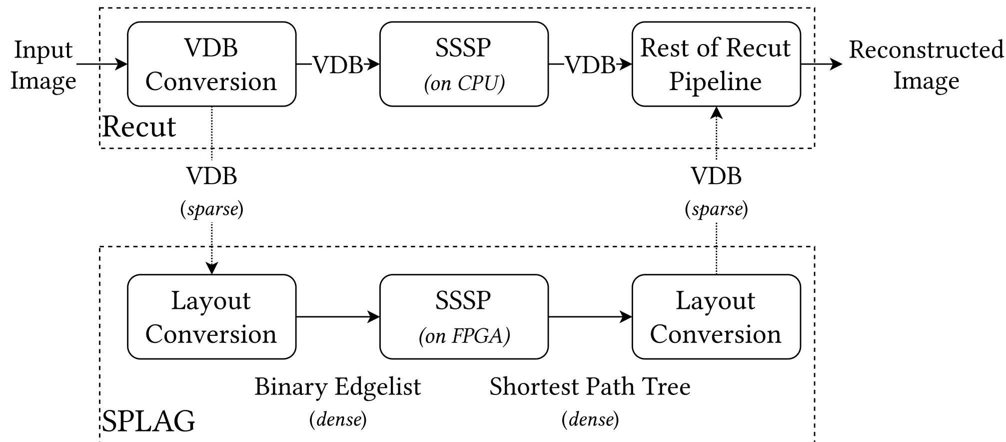


Figure 8.21: Recut [124] pipeline with SPLAG integrated for the SSSP part.

Figure 8.18 shows the Recut [124] processing pipeline. Recut uses the VDB [134] format to store the sparse structure efficiently. It first converts the raw input images to the VDB format, and then performs transformation passes, e.g., SSSP, on that sparse data structure. However, SPLAG uses a dense representation of graphs, as illustrated in Figure 8.20 (also discussed in Section 8.1.5). Therefore, we have to convert the data layout before sending the image from Recut and after SPLAG has computed the shortest path tree. While this conversion takes time, the highly efficient edge traversal provided by SPLAG makes up the overhead. The overall workflow for Recut with SPLAG integration is demonstrated in Figure 8.21. On-board execution of a real-world neural image demonstrates a total of 34.8s runtime. As a comparison, it takes 78.1s for the CPU-based Recut for the same workload [124], which makes the end-to-end speedup 2.2×. This speedup includes the "accelerator tax" we paid, not only for the host-device data transfer, but also the layout transformation.

Figure 8.22 visualizes the breakdown of the end-to-end execution time of Recut with SPLAG integration. This execution time can be divided into three components:

1. pre-processing layout conversion,

2. accelerator execution, and

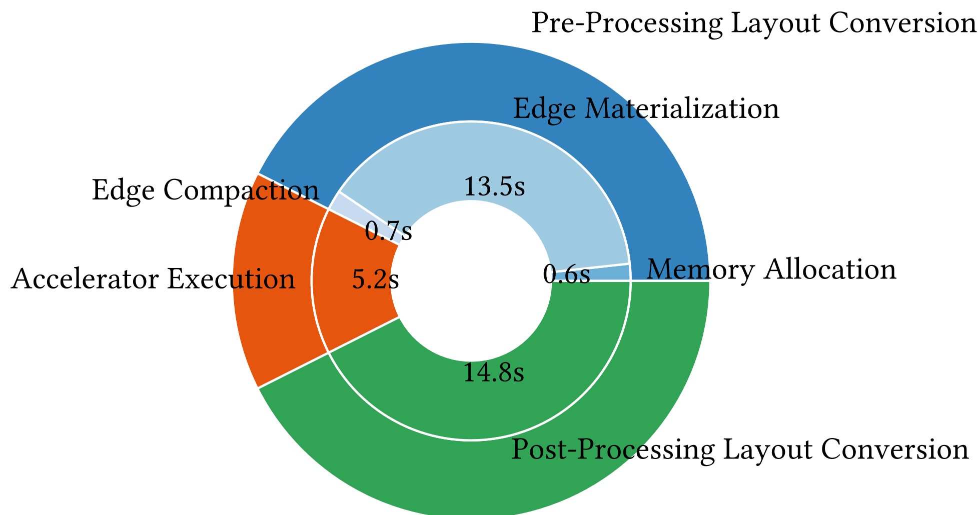


Figure 8.22: Execution time breakdown for Recut with SPLAG integration.

3. post-processing layout conversion.

Pre-processing layout conversion consists of three steps.

1. Memory allocation. This step allocates memory for the data structures on the host memory for the layout conversion, including the dense binary edge list, a sparse array mapping sparse 3D coordinates to dense 1D indices, and a dense array mapping dense 1D indices back to sparse 3D coordinates. The sparse *coord-to-index* mapping leverages VDB [134] and is initialized empty. The dense *index-to-coord* mapping uses `std::vector` and is initialized with enough space for all voxels (we know the number of active voxels via VDB). Each voxel can only have up to 6 neighbors, but we do not know the exact number of edges at this point. Therefore, we reserve memory for 6 neighbors, and rely on a compaction step to release unnecessary memory. This bookkeeping step takes very little time in pre-processing.

2. Edge materialization. This step traverses the sparse 3D image, assign indices to voxels if none has been assigned, records the assigned indices in the two mappings, calculates the

edge weights from the voxel intensity, and stores the edges in the pre-allocated memory space. Due to the sparse data structures, edge materialization step contributes the most time in pre-processing.

3. Edge compaction. After edge materialization, we will know exactly how many neighbors each vertex has. In the edge compaction step, we remove the allocated but unused memory space by overwriting them with edges for the next vertex. Since the accesses are dense and sequential, this step takes much less time than edge materialization.

Accelerator execution includes the host-device data transfer time in addition to the FPGA accelerator execution. Post-processing layout conversion traverses the shortest path tree stored in form of the distances and parents of vertices and reconstructs the sparse VDB format, leveraging the *index-to-coord* mapping obtained from pre-processing.

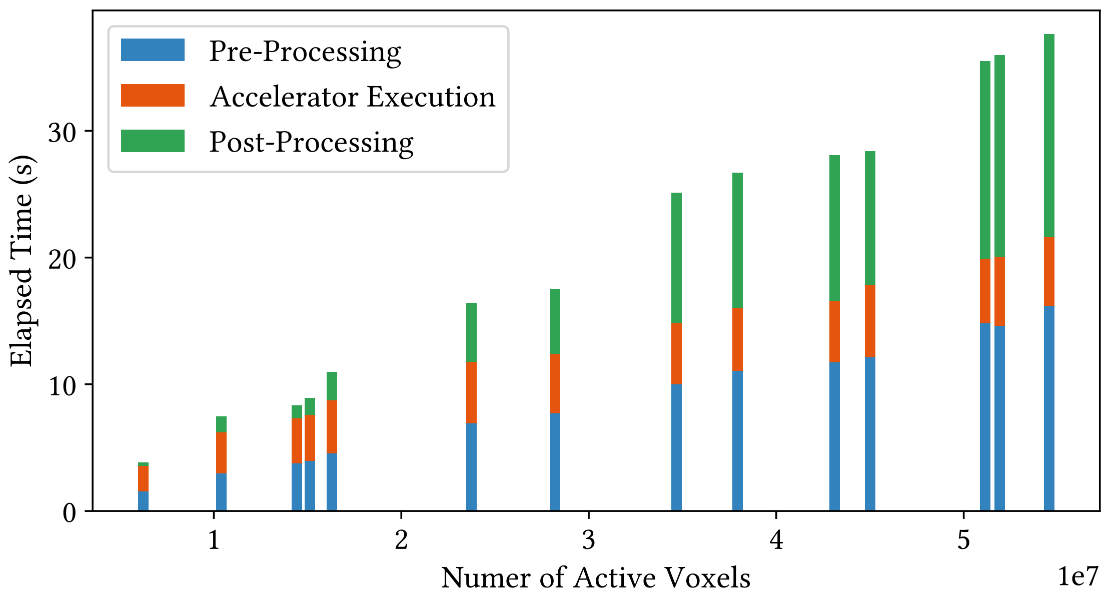


Figure 8.23: Execution time breakdown with varying number of active voxels.

The neural images evaluated in this section are produced by a Dragonfly imaging system [89], which images a full mouse striatum brain region at 0.2 μm × 0.2 μm × 1 μm resolution. Due to constraints in this modality of imaging, the striatum is sliced into ~16 coronal physical sections.

Each evaluated image is half of one section of the striatum, or 1/32 of a single animal striatum, at full resolution without down-sampling. To evaluate the scalability of our integration approach, we test 14 different images with varying number of active voxels. Figure 8.23 illustrates that the execution time scales linearly as the number of active voxels (i.e., the number of vertices) increases. As demonstrated in Figure 8.22 and Figure 8.23, currently, pre-processing and post-processing dominate the SSSP stage of the Recut pipeline. We hope to offload more stages in the Recut pipeline to the FPGA in order to amortize the overhead for data layout transformation.

8.2 Graph Convolutional Network

Graph convolutional network (GCN) [101] is an emerging graph application. As a generalization of neural networks, GCN has been widely used for learning properties from graph data structures since it was first proposed in the 2010s. Unlike traditional neural networks that work on regular, dense matrices of values, GCNs work on irregular, sparse graph data.

While there are many variants of the original GCN [101], most of the variants share similar computational pattern. For these variants, the goal is to learn features on a graph $G = (V, E)$, which takes as input ① an input feature matrix X_{in} with $|V|$ rows of f_{in} features per row, and ② a representation of the graph G, e.g., in form of an adjacency matrix A, and produces an output feature matrix X_{out} with $|V|$ rows of f_{out} features per row [101]. Similar to deep convolutional neural networks, GCNs are also layered. Each layer can be written as a non-linear function $H^{(l+1)} = f(H^{(l)}, A)$, where different GCN models choose different $f(\cdot,\cdot)$, and $H^{(l)}$ is the feature matrix of layer l. The non-linear function proposed by Kipf and Welling [101] is

$$f\left(H^{(l)}, A\right) = \text{ReLU}\left(\hat{D}^{-\frac{1}{2}}\hat{A}\hat{D}^{-\frac{1}{2}} \cdot H^{(l)} \cdot W^{(l)}\right) \tag{8.1}$$

with $\hat{A} = A + I$ where I is the identify matrix and $\hat{D}$ is the diagonal degree matrix of $\hat{A}$. $W^{(l)}$ is the weight matrix of each layer, which is obtained from training and is usually dense. $H^{(l)}$ is the feature matrix of each layer, which is usually dense unless, e.g., it is a one-hot encoded feature matrix. $\hat{A}$ is usually very sparse since it is an adjacency matrix for a graph.

As shown in Formula 8.1, lying in the core of GCN is sparse matrix–dense matrix multiplication (SpMM), where the sparse matrix brings the challenges for memory-intensive applications. In this section, we will present theoretical analysis and practical challenges for an SpMM accelerator written in TAPA (Section 6) with optimizations discussed in Section 7.3, followed by experimental evaluation on the SpMM computation kernel.

8.2.1 Theoretical Analysis

Consider $C = A \times B$ where A is a sparse matrix and B, C are dense matrices. Let the sparse matrix be $n \times n$ with m non-zero elements, and the dense matrices be $n \times F$. For GCN, n would be the number of vertices and m corresponds to the number of edges (including self-edges added by the identify matrix in Formula 8.1). The total amount of computation for SpMM $C = A \times B$ is $m \times F$ multiply-accumulate operations. The total amount of communication includes:

1. reading m numbers for the sparse matrix,
2. reading $n \times F$ numbers for the dense matrix, and
3. writing $n \times F$ numbers for the dense matrix.

Assuming the A matrix is so sparse that the whole multiplication is memory-bound and the on-chip storage is not large enough to store all matrices, in this section, we evaluate three possible SpMM update schemes and model the amount of off-chip memory traffic in order to compare the three update schemes. The three schemes have been seen in various previous works for GCN acceleration and traditional graph processing in the literature [13, 21, 47, 48, 72, 80, 108, 149, 172, 176, 181, 182].

8.2.1.1 Single Phase Update (SPU)

The single phase update (SPU) scheme assumes partitions of the input and output dense matrix are both on-chip so that each partition is updated in a single phase without an intermediate step.

This scheme is seen in [21, 47, 48, 72, 80, 172, 176, 182]. To this end, we have to divide the vertices into partitions to fit partitions on-chip. Let the number of such partitions be p. We will load each input partition $p - 1$ times, once per output partition, minus one for reusing the input partition when the output partition is switched. Reusing the input partition can be achieved by alternating the replacement direction of the input matrix, e.g., using snake-shaped replacement as illustrated in Figure 8.24. The output partitions are stored only once. Moreover, we can partition the $n \times F$ dense matrices B and C into k partitions, each of which has size $n \times \frac{F}{k}$. This allows us to decrease p by processing the k partitions of B and C one-by-one, at the cost of reading the sparse matrix A k times. If S is the on-chip storage size, $p = \frac{nF}{kS}$.

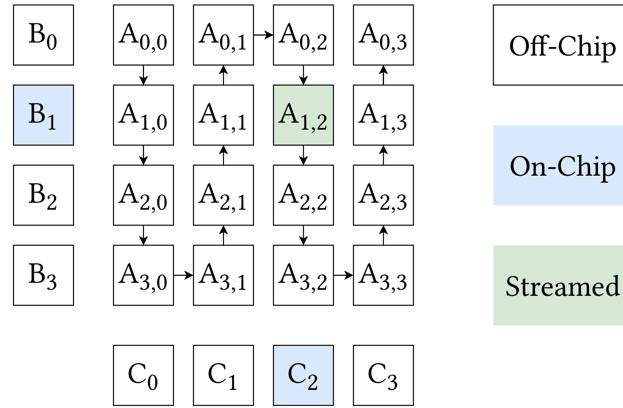


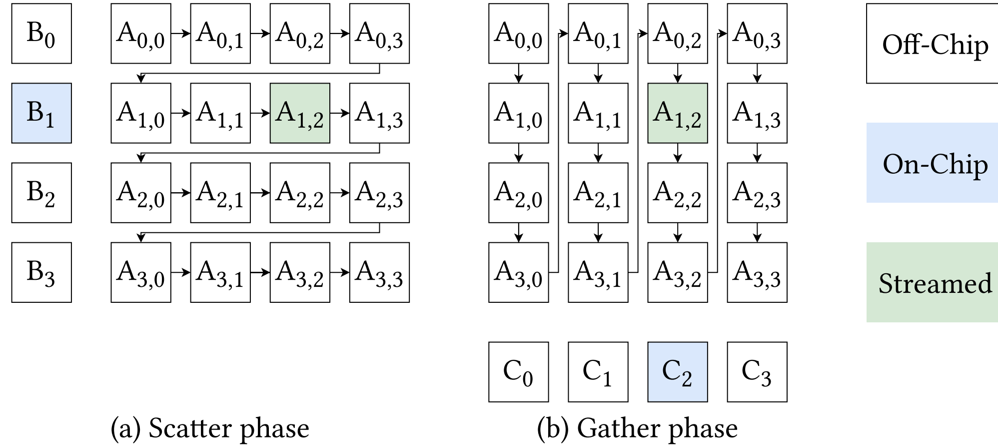
Figure 8.24: Example of single phase update. Both B and C are partitioned into $p = 4$ partitions, and A is partitioned into $p^2 = 16$ partitions accordingly. Partitions of A are streamed from the off-chip memory while partitions of B and C are preloaded or pre-allocated on-chip. By switching partitions of B first, each partition of C is written to the off-chip memory only once.

Figure 8.24 illustrates an example of SPU. The total communication required by SPU include ① $m \times k$ for reading the sparse matrix, ② $n \times F \times (p - 1)$ for reading the input dense matrix, ③ $n \times F$ for writing the output dense matrix, or in total

$$mk + npF = mk + \frac{(nF)^2}{kS} \tag{8.2}$$

8.2.1.2 Dual-Phase Update (DPU)

The dual-phase update (DPU) scheme writes partial sums to the external memory and reads them back to accumulate them. This scheme is seen in [21, 149, 181]. While the intermediate step doubles the memory traffic, we only need to read the input vertices once regardless of the on-chip storage size.



(a) Scatter phase (b) Gather phase

Figure 8.25: Example of dual-phase update. Both B and C are partitioned into $p = 4$ partitions, and A is partitioned into $p^2 = 16$ partitions accordingly. (a) In the scatter phase, partitions of A are streamed from the off-chip memory, partitions of B are preloaded on-chip, while the resulting partial sums corresponding to the partition of A are streamed to the off-chip memory. (b) In the gather phase, partial sums corresponding to the partitions of A are streamed from the off-chip memory while partitions of C are pre-allocated on-chip. By storing the partial sums off-chip, each partition of B and C are only accessed once in the off-chip memory.

Figure 8.25 shows an example of DPU. The total communication required by DPU include ① m for reading the sparse matrix A, ② $n \times F$ for reading the input dense matrix B, ③ $m \times F$ for writing the intermediate results, ④ $m \times F$ for reading the intermediate results back, and ⑤ $n \times F$ for writing the output dense matrix C, or in total

$$m + 2(m + n)F \tag{8.3}$$

8.2.1.3 Direct Update (DU)

Direct update scheme reads the input dense matrix from the external memory directly as it works on the output matrix. This scheme is seen in [13, 108]. The n rows of the input matrix will be loaded m times in total.

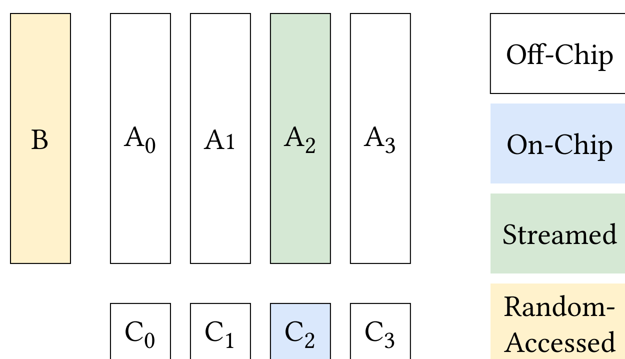


Figure 8.26: Example of direct update. Matrix C is partitioned into $p = 4$ partitions, and matrix A is partitioned accordingly. Matrix B is not partitioned and is read directly and randomly from the off-chip memory without using a scratchpad memory while A is streamed from the off-chip memory. Partitions of C are pre-allocated on-chip and are written to the off-chip memory when the corresponding partition of A is finished. Each partition of A and C are only accessed once in the off-chip memory, but B is accessed m/n times on average at a reduced memory bandwidth (because random access bandwidth is lower than sequential access bandwidth).

Figure 8.26 demonstrates an example of DU. The total communication required by DU include ① m for reading the sparse matrix A, ② $m \times F$ for reading the input dense matrix B, ③ $n \times F$ for writing the output dense matrix C. Note that unlike SPU and DPU where only the on-chip memory is accessed randomly and the off-chip memory is always accessed sequentially, DU requires direct random accesses on the off-chip memory for reading B, which is lower than the sequential access bandwidth (details in Section 8.2.1.4). Therefore, we add a coefficient $\alpha > 1$ to the amount of communication corresponding to the input matrix B. The total communication

required by DU is then

$$m + (\alpha m + n)F \tag{8.4}$$

8.2.1.4 Random Access Ratio

The random access ratio α defined in Section 8.2.1.3 is a function of the granularity of each access (F). This function $\alpha(F)$ depends on the properties of the external memory and needs to be measured from experiments. Leveraging the asynchronous memory interface (Section 7.2), we can measure the random access throughput by issuing as many outstanding requests as possible until the address channel is blocked by the external memory. Table 8.6 summarizes the result.

Table 8.6: Random access bandwidth on the Alveo U280 FPGA.

Burst Size (Byte)	**Bandwidth (GB/s)**		$\frac{\textbf{Sequential Bandwidth}}{\textbf{Random Bandwidth}}$	
	DDR	HBM	DDR	HBM
64	3.9	3.3	4.2	4.0
128	6.1	6.6	2.7	2.0
256	7.6	10.7	2.1	1.2
512	10.6	12.6	1.5	1.0
1024	13.5	12.7	1.2	1.0
2048	15.2	12.8	1.1	1.0
4096	16.3	13.0	1.0	1.0

8.2.1.5 Comparison of the Three Update Schemes

We can compare Formula 8.2, Formula 8.3, and Formula 8.4 and find the best update scheme with the least amount of memory traffic. To enable comparison without the impact of absolute dataset size, we divide each formula by nF to obtain the normalized amount of memory traffic. Let $d = \frac{m}{n}$ be the average degree of the graph corresponding to the sparse matrix. The normalized amount

of memory traffic for SPU is

$$\frac{nF}{kS} + \frac{kd}{F} \tag{8.5}$$

Since k is a design parameter of our choice, we shall choose $k = F\sqrt{\frac{n}{Sd}}$ to minimize Formula 8.5 to

$$2\sqrt{\frac{m}{S}} \tag{8.6}$$

The normalized amount of memory traffic for DPU is

$$\frac{d}{F} + 2d + 2 \tag{8.7}$$

The normalized amount of memory traffic for DU is

$$\frac{d}{F} + \alpha d + 1 \tag{8.8}$$

As illustrated in Formula 8.6, Formula 8.7, Formula 8.8, and Table 8.6, the optimal choice among the three schemes depends on both the hardware platform (S) and the input matrices (m, d, and F). We quantitatively examined 8 representative real-world GCN models used in [176] and 4 synthetic datasets, and conservatively assume that half of URAMs on an Alveo U280 board can be used for storing the output dense matrix ($S = 3932160$ 32-bit floating point numbers). Table 8.7 summarizes the result. It shows that for all 8 representative GCN models, SPU is the optimal choice under our analytical model. The reason is that modern FPGAs have very large on-chip storage that reduces the number of required partitions for SPU, while the examined real-world graph datasets are not sparse enough so that DVU or DU would be more efficient. The synthetic datasets demonstrate that if the datasets are two orders of magnitude larger in the number of vertices, SPU will lose its optimality. In the remainder of this section, we will discuss some practical challenges to implement the SPU scheme and how we address them using TAPA (presented in Chapter 6 and Chapter 7).

Table 8.7: Comparison of off-chip memory accesses among SPU, DPU, and DU on 8 real-world datasets used in [176] and 4 synthetic datasets. **Bold** items have the minimum memory accesses on DDR-based systems and underlined items have the minimum memory accesses on HBM-based systems.

Dataset	n	d	F	k	**SPU**	**DPU**	**DU (DDR)**	**DU (HBM)**
Cora	2.7k	3.8	16	1	**0.2**	9.8	17.1	16.4
Cora	2.7k	3.8	128	2	**0.1**	9.6	6.7	4.8
Citeseer	3.3k	2.6	16	1	**0.2**	7.4	12.3	11.7
Citeseer	3.3k	2.6	128	2	**0.1**	7.3	5.0	3.7
Pubmed	19k	4.4	16	1	**0.4**	11.0	19.6	18.8
Pubmed	19k	4.4	128	4	**0.3**	10.8	7.6	5.4
PPI	14k	29	128	1	**0.7**	61.0	45.3	30.6
PPI	14k	29	256	2	**0.7**	60.9	36.4	30.5
Flickr	89k	9.8	128	8	**1.0**	21.7	15.8	10.9
Flickr	89k	9.8	256	16	**1.0**	21.6	12.8	10.8
Reddit	0.23M	92	128	4	**4.7**	186.7	139.7	93.7
Reddit	0.23M	92	256	8	**4.7**	186.4	111.8	93.4
Yelp	0.72M	19	128	16	**3.9**	41.0	30.3	20.6
Yelp	0.72M	19	256	32	**3.9**	41.0	24.4	20.5
Amazon	1.6M	176	128	8	**17.5**	355.4	266.4	178.4
Amazon	1.6M	176	256	16	**17.5**	354.7	212.9	177.7
Synthetic	2^{24}	32	128	64	**24.5**	66.3	49.3	33.3
Synthetic	2^{25}	32	128	64	**33.1**	66.3	49.3	33.3
Synthetic	2^{26}	32	128	128	**49.1**	66.3	49.3	33.3
Synthetic	2^{27}	32	128	128	66.1	66.3	**49.3**	33.3

8.2.2 Practical Challenges

Following the analysis in Section 8.2.1, we will store the dense matrix on-chip and load the sparse matrix from off-chip memory directly. While all off-chip memory accesses are sequential, which is good for bandwidth utilization, the intensive yet irregular on-chip memory accesses impose two challenges. First, each BRAM or URAM bank only has up to 2 ports, and concurrent accesses must be scattered among many banks to ensure full pipelining. However, the sparse matrix has irregular structure that do not always allow such concurrent accesses. Second, floating-point accumulation takes multiple clock cycles on the FPGA, which creates true read-after-write dependency for SpMM. Without special handling, the loop initiation interval can be severely bottlenecked by the latency of the floating-point accumulation.

8.2.2.1 Resolving Bank Conflicts

As discussed in Section 8.2.1, given the state-of-the-art FPGAs and the typical properties of sparse matrices, the best scheme to perform SpMM is to partition the matrices so that each partition of the dense matrix fit on-chip, and read the sparse matrix from the off-chip memory directly. In this section, we assume both B and C are stored on-chip. Consider the core operation of SpMM, i.e., the multiply-accumulation operation (MAC): $c_{ij} = a_{ik} \cdot b_{kj} + c_{ij}$, with a_{ik} from the sparse matrix A, b_{kj} from the input dense matrix B, and c_{ij} from the output dense matrix C. To read a_{ik} only once with sufficient memory coalescing, we fully unroll the traversal of the j loop and partially unroll the traversal of the i loop, too, to guarantee sufficient coalescing factor for A. Since A is sparse, each a_{ik} needs to carry not only the 32-bit floating point value but also the indices i and k. Therefore, the coalescing factor for A is set to 8 for the optimal 512-bit coalesced access. We partition C cyclically to expose sufficient parallelism and resolve bank conflicts for c_{ij}. Doing so also requires reordering of a_{ik} so that the i indices of a_{ik} processed in each clock cycle falls in different banks. To resolve the bank conflicts on b_{kj}, we duplicate B in addition to partitioning. Duplication is necessary because while the j index is consecutive and can be cyclically assigned

to each different banks, different i index can still result in random accesses on the k index of b_{kj}.

8.2.2.2 Handling Read-after-Write Dependency

Section 7.3 has discussed the reason and a solution for the large initiation interval (II) caused by floating-point accumulations. Although we can employ exactly the same dynamic resolution approach, for SpMM where the sparse matrix A is known offline, we can employ a less expensive offline dependency resolution via scheduling a_{ik} out of order. This offline resolution step can be performed while the sparse matrix is being partitioned, by reordering a_{ik} so that any consecutive L elements do not access the same location in a bank. If we cannot find a suitable element, we will add a bubble to the pipeline by inserting an invalid a_{ik} (marked using a special bit).

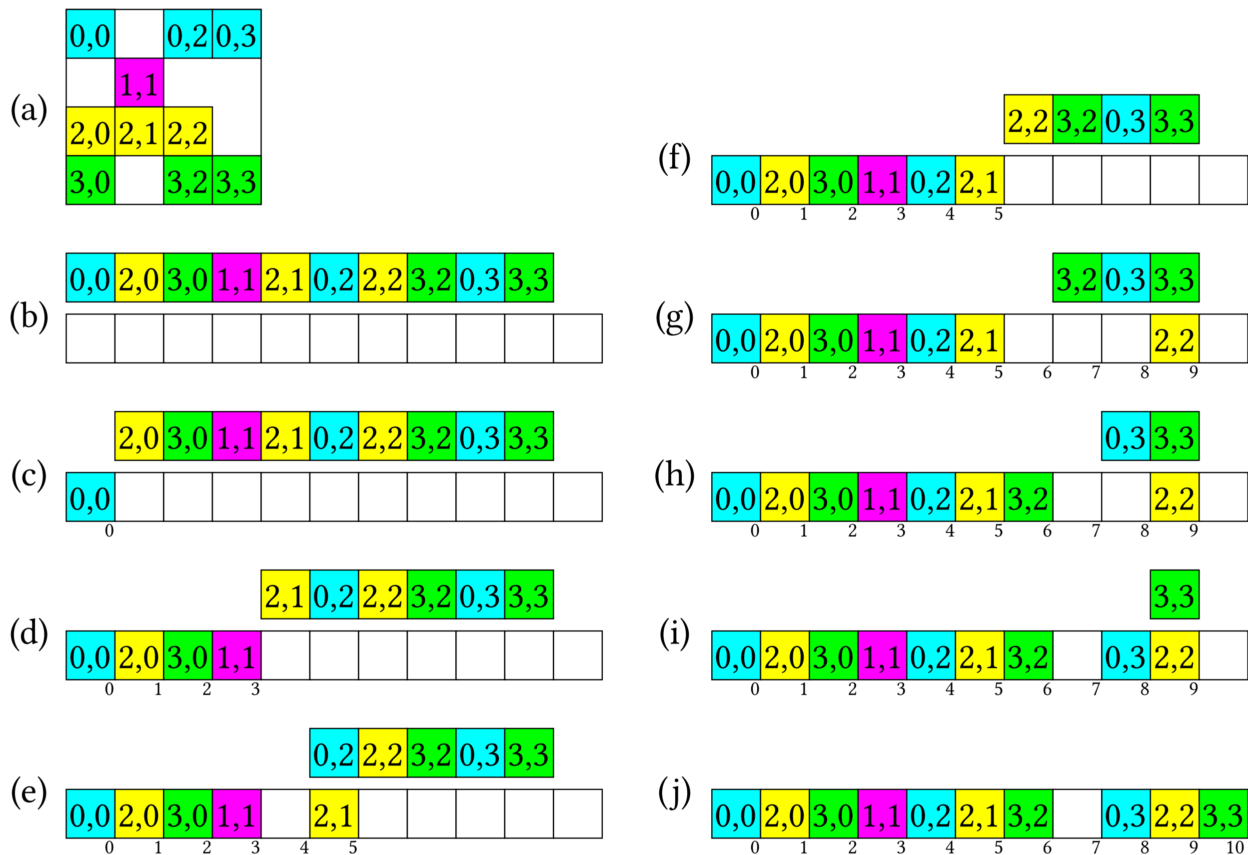


Figure 8.27: Out-of-order scheduling to handle read-after-write dependency [158].

Figure 8.27 demonstrates an example of the offline reordering of a 4×4 A matrix, assuming the RAW dependency distance L is 4. For simplicity, we further assume the i loop is pipelined without partial unrolling, and all rows of C (corresponding to rows of A) belong to the same bank. Different rows of A in Figure 8.27 corresponds to different i index of a_{ik} and are marked with different colors, as demonstrated in Figure 8.27a. Since all rows of C belong to the same bank, accesses to the same row must be separated by at least $L = 4$ cycles. Since a_{ik} and c_{ij} accesses the same rows, a_{ik} from the same rows of A must be separated by at least $L = 4$ cycles. Figure 8.27b–Figure 8.27d shows that the first 4 elements in A can be arranged without adding bubbles. Figure 8.27e shows that $a_{2,1}$ is delayed by 1 cycle due to dependency from $a_{2,0}$. Figure 8.27f shows that $a_{0,2}$ fills the bubble left by $a_{2,1}$ in Figure 8.27e. Figure 8.27j shows the eventual ordering of A with 10 elements in 11 loop iterations (with 1 bubble inserted). With the dependency pragma, this loop can be pipelined correctly with an II of 1, and the 11 iterations can finish in $11 \times 1 + 4 = 15$ cycles. As a comparison, without the offline reordering, the loop would be pipelined with an II of 4, which will take $10 \times 4 + 4 = 44$ cycles.

8.2.3 Evaluation

In this section, we demonstrate the effectiveness of the optimization techniques presented in Section 7.3 and Section 8.2.2 using Sextans [158]. Sextans is a streaming accelerator for general-purpose SpMM written in TAPA with major contributions from our lab mate, Linghao Song. In addition to the bank conflict and offline dependency resolution optimizations discussed in Section 8.2.2, Sextans also features multi-level memory optimizations with high-bandwidth memory (HBM) for efficient accessing and streaming, and hardware flexibility to support different matrices using the same hardware without reconfiguration.

Table 8.8: Hardware platforms used for SpMM evaluation.

System	Hardware	Tech. (nm)	Freq. (MHz)	Bw. (GB/s)	TDP/W
CuSPARSE [136]	Tesla K80	28	562	480	300
Sextans [158]	Alveo U280	16	189	460	225
CuSPARSE [136]	Tesla V100	12	1297	900	300
Sextans-P [158]	Projection	16	350	900	300

8.2.3.1 Evaluation using SpMM Kernel

Sextans is a general-purpose SpMM accelerator. We shall evaluate Sextans using standalone SpMM kernels in this section. Section 8.2.3.2 will discuss the end-to-end performance in a complete GCN where SpMM is a computational kernel. As a general-purpose SpMM accelerator, Sextans is evaluated using 200 different sparse matrices, with the sizes varying from 5 to 5.1×10^5. The number of non-zeros in the sparse matrices varies from 10 to 3.7×10^7, and the sparsity varies from 5.97×10^{-6} to 0.4. Each sparse matrix is evaluated using 7 different dense matrices, whose width are powers of 2 from 8 to 512. Table 8.8 summarizes the hardware platforms used for SpMM evaluation.

Figure 8.28 demonstrates the evaluation results. The peak throughput of K80, Sextans, V100, and Sextans-P are 128 GFLOP/s, 181 GFLOP/s, 688 GFLOP/s, and 344 GFLOP/s, respectively. The geometric mean speedups of the four platforms normalized to K80 are 1.0×, 2.5×, 4.3×, and 4.9×, respectively. The geometric mean speedup of Sextans-P to V100 is 1.1×.

8.2.3.2 Evaluation using Complete GCN

In this section, we interface the Sextans accelerator with PyTorch Geometric (PyG) [66] leveraging the convenient host-kernel interface API provided by TAPA (Section 6.2.2.3), and evaluate the end-to-end speedup brought by Sextans using a complete GCN model [101]. PyG is a popular

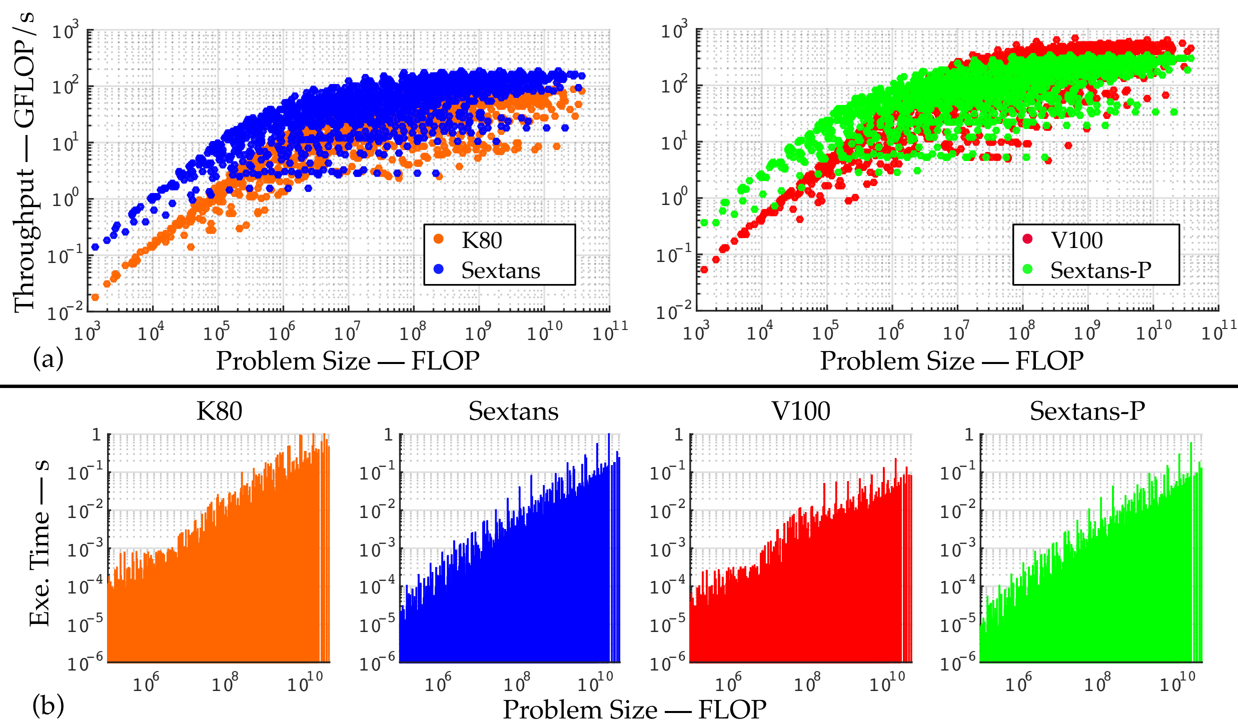


Figure 8.28: Throughput (a) and execution time (b) of SpMM on K80, Sextans, V100, and Sextans-P, with varying problem sizes (number of floating-point operations).

library built upon PyTorch [141] to easily write and train graph neural networks for a wide range of applications related to structured data. The baseline implementation of the original GCN is provided by PyG, and we modify its SpMM implementation to provide an option to offload the computation to Sextans. PyG caches the preprocessing results for the input graph (i.e., the sparse matrix A for Sextans). We take advantages of this feature to amortize the overhead brought by the offline read-after-write dependency resolution (Section 8.2.2.2).

Table 8.9 summarizes the datasets and the performance comparison. For each benchmark, we evaluate two representative settings of the hidden layer feature size as presented in [176]. We can see that if we only compare the SpMM computational kernel (L1 Kernel and L2 Kernel in Table 8.9), Sextans can always achieve a speedup for all datasets and all layers, and the speedup is significant (107.3× and 35.4× geometric mean for the first and second layer, respectively). How-

Table 8.9: Speedup of Sextans [158] on a U280 FPGA over 32 CPU threads running at 4.4GHz. The kernel speedup measures only the FPGA computation time. The end-to-end (E2E) speedup considers the communication overhead imposed by the offloaded computation kernel. The total speedup measures the speedup of the whole GCN inference, including computation that is not offloaded to the FPGA, e.g., the nonlinear activation function (ReLU).

Dataset	**#Vertices**	**#Features**		**FLOP**		**Sextans Speedup**				
		L1	L2	L1	L2	L1 Kernel	L1 E2E	L2 Kernel	L2 E2E	Total
Cora	2.7k	16	7	0.42M	0.19M	4.5×	0.1×	2.3×	0.3×	0.5×
Cora	2.7k	128	7	3.4M	0.19M	23.2×	1.0×	2.8×	0.3×	1.0×
Citeseer	3.3k	16	6	0.40M	0.15M	5.8×	0.1×	2.2×	0.3×	0.6×
Citeseer	3.3k	128	6	3.2M	0.15M	26.1×	0.5×	2.2×	0.3×	0.7×
Pubmed	19k	16	3	3.5M	0.65M	26.9×	1.0×	6.5×	0.7×	1.0×
Pubmed	19k	128	3	27.7M	0.65M	156.3×	2.4×	12.4×	0.4×	1.9×
PPI	14k	128	121	0.11G	0.10G	44.6×	3.5×	40.6×	4.9×	2.2×
PPI	14k	256	121	0.22G	0.10G	93.8×	7.2×	53.3×	5.6×	3.6×
Flickr	89k	128	7	0.25G	14M	264.2×	4.3×	59.4×	3.4×	3.7×
Flickr	89k	256	7	0.51G	14M	200.6×	2.6×	39.2×	1.9×	2.4×
Reddit	0.23M	128	41	6.0G	1.9G	1341.4×	40.8×	900.9×	47.7×	19.0×
Reddit	0.23M	256	41	12G	1.9G	1537.3×	37.3×	846.0×	43.3×	20.6×
Yelp	0.72M	128	100	3.6G	2.8G	816.3×	8.6×	736.8×	8.2×	7.0×
Yelp	0.72M	256	100	7.1G	2.8G	1091.0×	8.9×	687.4×	7.4×	7.4×
Geo. Mean	—	—	—	—	—	107.3×	2.6×	35.4×	2.1×	2.6×

ever, if we measure the end-to-end performance (L1 E2E and L2 E2E in Table 8.9), Sextans can only achieve speedup for large datasets with more than ~3.5M floating-point operations, and the geometric mean speedup is decreased to 2.6× and 2.1×, respectively. This speedup accounts for the overhead brought by the additional function calls, data layout conversion, and host-device communication required by the accelerator. If we consider the execution time of the whole GCN inference kernel, the geometric mean speedup is 2.6×, further demonstrating effectiveness of our

methods.

8.3 Summary

In this chapter, we demonstrated accelerator design and optimizations for two graph applications with irregular memory access patterns. Both accelerators leverage the TAPA framework presented in Chapter 6 and the enhancements for dynamic memory accesses in Chapter 7. On the latest HBM-equipped U280 FPGA, the SSSP accelerator benefits from the optimizations for dynamic off-chip memory accesses and shows a 2.2× geometric mean speedup over 32 CPU threads at 4.4 GHz. The GCN accelerator benefits from the optimizations for dynamic on-chip memory accesses, and illustrates a 2.6× geometric mean speedup over the same CPU.

Although TAPA does not support dynamically instantiate tasks at the language level, the two application case studies demonstrate that we can schedule tasks dynamically on top of the statically instantiated hardware, with manually coordinated and optimized inter-task communication channels. More specifically, we use multi-stage networks in Section 8.1, while we use a daisy chain in Section 8.2. Such complex inter-task communication patterns are also seen in many other applications [24, 40, 163], and we think a hardened network-on-chip (NoC) would be beneficial for these applications [45]. Indeed, the next-generation Versal [70] FPGA manufactured by Xilinx would be equipped with such a hardened NoC. We are actively working on an extension of TAPA to support such a new accelerator architecture. Meanwhile, we are also exploring a communication-centric approach to implement FPGA accelerators, where the communication network is planned before the rest of the accelerator [78]. This can enable efficient support for task-parallel overlay accelerators with hardened NoC, e.g., PolyGraph [45].

CHAPTER 9

Conclusions

This dissertation discusses accelerator design automation and optimization techniques for various memory-bound applications, including both regular and irregular access patterns. For regular memory accesses, we start from theoretical analysis to reduce memory traffic and prove that communication reuse can be optimally achieved with computation reuse, and propose the stencil with optimized dataflow architecture (SODA). We then present model-driven design-space exploration to automatically find the best design parameters based on the theoretical analysis. To further simplify accelerator design, we present an end-to-end compiler framework, HeteroHalide, that compiles a high-level domain-specific language to efficient accelerators. For irregular memory accesses, we first enhance the state-of-the-art high-level synthesis tools and present TAPA to better support task-level programs, which are commonly used to implement applications with irregular accesses. To handle applications with dynamically scheduled memory accesses, we extend TAPA to support dynamic accesses. Using the developed tools, we perform case studies on two important real-world graph applications. For the single-source shortest path algorithm, experimental results on various synthetic and real-world graph datasets using an HBM-equipped FPGA demonstrate up to 763 MTEPS overall throughput, a 4.9× speedup over state-of-the-art accelerators, and 2.6× speedup over state-of-the-art multi-thread CPU implementation. For graph convolutional networks, on-board execution illustrates a geometric mean speedup of 2.6× over a widely-used multi-thread CPU implementation.

REFERENCES

[1] Maleen Abeydeera and Daniel Sanchez. Chronos: Efficient Speculative Parallelism for Accelerators. In *ASPLOS*, 2020.

[2] Andrew Adams, Frédo Durand, Jonathan Ragan-Kelley, Karima Ma, Luke Anderson, Riyadh Baghdadi, Tzu-Mao Li, Michaël Gharbi, Benoit Steiner, Steven Johnson, and Kayvon Fatahalian. Learning to optimize halide with tree search and random programs. In *SIGGRAPH*, 2019.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. Chisel: Constructing Hardware in a Scala Embedded Language. In *DAC*, 2012.

[4] Eli Bendersky. Measuring context switching and memory overheads for Linux threads. `https://eli.thegreenplace.net/2018/measuring-context-switching-and-memory-overheads-for-linux-threads/`, 2018.

[5] Ranjita Bhagwan and Bill Lin. Fast and Scalable Priority Queue Architecture for High-Speed Network Switches. In *INFOCOM*, 2000.

[6] Robert D Blumofe, Christopher F Joerg, Bradley C Kuszmaul, Charles E Leiserson, Keith H Randall, and Yuli Zhou. Cilk: An Efficient Multithreaded Runtime System. In *PPoPP*, 1995.

[7] Uday Bondhugula and Vinayaka Bandishti. Diamond Tiling : Tiling Techniques to Maximize Parallelism for Stencil Computations. *TPDS*, 28(5), 2017.

[8] Uday Bondhugula, Ananth Devulapalli, James Dinan, Joseph Fernando, Pete Wyckoff, Eric Stahlberg, and P. Sadayappan. Hardware/Software Integration for FPGA-based All-Pairs Shortest-Paths. In *FCCM*, 2006.

[9] Alan C. Bovik. *The Essential Guide to Image Processing*. 2009.

[10] Ulrik Brandes and Christian Pich. Centrality Estimation in Large Networks. *International Journal of Bifurcation and Chaos*, 17(07), 2007.

[11] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-Level Synthesis for FPGA-Based Processor/Accelerator Systems. In *FPGA*, 2011.

[12] Han Chen, Sergey Madaminov, Michael Ferdman, and Peter Milder. FPGA-Accelerated Samplesort for Large Data Sets. In *FPGA*, 2020.

[13] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. ThunderGP: HLS-based Graph Processing Framework on FPGAs. In *FPGA*, 2021.

[14] Yu Ting Chen, Jin Hee Kim, Kexin Li, Graham Hoyes, and Jason H. Anderson. High-Level Synthesis Techniques to Generate Deeply Pipelined Circuits for FPGAs with Registered Routing. In *FPT*, 2019.

[15] Zhe Chen, Hugh T Blair, and Jason Cong. LANMC: LSTM-Assisted Non-Rigid Motion Correction on FPGA for Calcium Image Stabilization. In *FPGA*, 2019.

[16] Jianyi Cheng, Shane T. Fleming, Yu Ting Chen, Jason H. Anderson, and George A. Constantinides. EASY: Efficient Arbiter SYnthesis from Multi-threaded Code. In *FPGA*, 2019.

[17] Jianyi Cheng, Lana Josipović, George A. Constantinides, Paolo Ienne, and John Wickerson. Combining Dynamic & Static Scheduling in High-level Synthesis. In *FPGA*, 2020.

[18] Yuze Chi, Young-kyu Choi, Jason Cong, and Jie Wang. Rapid Cycle-Accurate Simulator for High-Level Synthesis. In *FPGA*, 2019.

[19] Yuze Chi and Jason Cong. Exploiting Computation Reuse for Stencil Accelerators. In *DAC*, 2020.

[20] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. SODA: Stencil with Optimized Dataflow Architecture. In *ICCAD*, 2018.

[21] Yuze Chi, Guohao Dai, Yu Wang, Guangyu Sun, Guoliang Li, and Huazhong Yang. NXgraph: An Efficient Graph Processing System on a Single Machine. In *ICDE*, 2016.

[22] Yuze Chi, Licheng Guo, and Jason Cong. Accelerating SSSP for Power-Law Graphs. In *FPGA*, 2022.

[23] Yuze Chi, Licheng Guo, Jason Lau, Young-kyu Choi, Jie Wang, and Jason Cong. Extending High-Level Synthesis for Task-Parallel Programs. In *FCCM*, 2021.

[24] Young-kyu Choi, Yuze Chi, Weikang Qiao, Nikola Samardzic, and Jason Cong. HBM Connect: High-Performance HLS Interconnect for FPGA HBM. In *FPGA*, 2021.

[25] Young-kyu Choi, Yuze Chi, Jie Wang, and Jason Cong. FLASH: Fast, ParalleL, and Accurate Simulator for HLS. *TCAD*, 2020.

[26] Young-kyu Choi, Yuze Chi, Jie Wang, Licheng Guo, and Jason Cong. When HLS Meets FPGA HBM: Benchmarking and Bandwidth Optimization. `https://arxiv.org/abs/2010.06075`, 2020.

[27] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *DAC*, 2016.

[28] Nitin Chugh, Vinay Vasista, Suresh Purini, and Uday Bondhugula. A DSL Compiler for Accelerating Image Processing Pipelines on FPGAs. In *PACT*, 2016.

[29] Moo Kyoung Chung, Jun Kyoung Kim, and Soojung Ryu. SimParallel: A High Performance Parallel SystemC Simulator Using Hierarchical Multi-threading. In *ISCAS*, 2014.

[30] Aaron Clauset, Cosma Rohilla Shalizi, and M. E. J. Newman. Power-Law Distributions in Empirical Data. *SIAM Review*, 51(4), 2009.

[31] Jason Cong. An Interconnect-Centric Design Flow for Nanometer Technologies. *Processings of the IEEE*, 89(4), 2001.

[32] Jason Cong. Timing Closure Based on Physical Hierarchy. In *ISPD*, 2002.

[33] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. Software Infrastructure for Enabling FPGA-Based Accelerations in Data Centers. In *ISLPED*, 2016.

[34] Jason Cong, Muhuan Huang, Di Wu, and Cody Hao Yu. Heterogeneous Datacenters: Options and Opportunities. In *DAC*, 2016.

[35] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers. In *DAC*, 2014.

[36] Jason Cong, Peng Li, Bingjun Xiao, and Peng Zhang. An Optimal Microarchitecture for Stencil Computation Acceleration Based on Non-Uniform Partitioning of Data Reuse Buffers. *TCAD*, 35(3), 2015.

[37] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-Level Synthesis for FPGAs: From Prototyping to Deployment. *TCAD*, 2011.

[38] Jason Cong and Jie Wang. PolySA: Polyhedral-Based Systolic Array Auto-Compilation. In *ICCAD*, 2018.

[39] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. Automated Accelerator Generation and Optimization with Composable, Parallel and Pipeline Architecture. In *DAC*, 2018.

[40] Jason Cong, Peng Wei, Cody Hao Yu, and Peipei Zhou. Latte: Locality Aware Transformation for High-Level Synthesis. In *FCCM*, 2018.

[41] Jason Cong and Zhiru Zhang. An Efficient and Versatile Scheduling Algorithm Based On SDC Formulation. In *DAC*, 2006.

[42] Melvin E. Conway. Design of a Separable Transition-Diagram Compiler. *Communications of the ACM*, 6(7), 1963.

[43] Keith Cooper, Jason Eckhardt, and Ken Kennedy. Redundancy Elimination Revisited. In *PACT*, 2008.

[44] A. Crauser, K. Mehlhorn, U. Meyer, and P. Sanders. A Parallelization of Dijkstra's Shortest Path Algorithm. In *MFCS*, 1998.

[45] Vidushi Dadu, Sihao Liu, and Tony Nowatzki. PolyGraph: Exposing the Value of Flexibility for Graph Processing Accelerators. In *ISCA*, 2021.

[46] Leonardo Dagum and Ramesh Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *IEEE Computational Science and Engineering*, 5(1), 1998.

[47] Guohao Dai, Yuze Chi, Yu Wang, and Huazhong Yang. FPGP: Graph Processing Framework on FPGA A Case Study of Breadth-First Search. In *FPGA*, 2016.

[48] Guohao Dai, Tianhao Huang, Yuze Chi, Ningyi Xu, Yu Wang, and Huazhong Yang. Fore-Graph: Exploring Large-scale Graph Processing on Multi-FPGA Architecture. In *FPGA*, 2017.

[49] Guohao Dai, Tianhao Huang, Yuze Chi, Jishen Zhao, Guangyu Sun, Yongpan Liu, Yu Wang, Yuan Xie, and Huazhong Yang. GraphH: A Processing-in-Memory Architecture for Large-scale Graph Processing. *TCAD*, 2018.

[50] Kaushik Datta, Mark Murphy, Vasily Volkov, Samuel Williams, Jonathan Carter, Leonid Oliker, David Patterson, John Shalf, and Katherine Yelick. Stencil Computation Optimization and Auto-tuning on State-of-the-Art Multicore Architectures. In *SC*, 2008.

[51] Andrew Davidson, Sean Baxter, Michael Garland, and John D. Owens. Work-Efficient Parallel GPU Methods for Single-Source Shortest Paths. In *IPDPS*, 2014.

[52] Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, and Torsten Hoefler. StencilFlow: Mapping Large Stencil Programs to Distributed Spatial Computing Systems. In *CGO*, 2021.

[53] Johannes De Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis. In *FPGA*, 2020.

[54] Ana Lúcia de Moura and Roberto Ierusalimschy. Revisiting Coroutines. *TOPLAS*, 31(2), 2009.

[55] Steven J. Deitz, Bradford L. Chamberlain, and Lawrence Snyder. Eliminating Redundancies in Sum-of-Product Array Computations. In *ICS*, 2001.

[56] Camil Demetrescu, Andrew Goldberg, and David Johnson. *The Shortest Path Problem*, volume 74 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science.* 2009.

[57] Chenhui Deng, Zhiqiang Zhao, Yongyu Wang, Zhiru Zhang, and Zhuo Feng. GraphZoom: A Multi-level Spectral Approach for Accurate and Scalable Graph Embedding. In *ICLR*, 2020.

[58] Robert H. Dennard, Fritz Gaensslen, Hwa-Nien Yu, V. Leo Rideout, Ernest Bassous, and Andre R. LeBlanc. Design of Ion-Implanted Small MOSFET's Dimensions with Very Small Physical Dimensions. *JSSC*, 9(5), 1974.

[59] Halide Developers. Halide. `https://github.com/halide/Halide`, 2019.

[60] RoBERT B Dial. Algorithm 360: Shortest-Path Forest with Topological Ordering. *CACM*, 12(11), 1969.

[61] E. W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1), 1959.

[62] Yufei Ding, Xipeng Shen, North Carolina, and United States. GLORE: Generalized Loop Redundancy Elimination upon LER-Notation. In *OOPSLA*, volume 1, 2017.

[63] Michael D Ernst. *Serializing Parallel Programs by Removing Redundant Computation.* PhD thesis, 1994.

[64] Juan Escobedo and Mingjie Lin. Graph-Theoretically Optimal Memory Banking for Stencil-Based Computing Kernels. In *FPGA*, 2018.

[65] Paul Feautrier. Some Efficient Solutions to the Affine Scheduling Problem. I. One-Dimensional Time. *IJPP*, 21(5), 1992.

[66] Matthias Fey and Jan E Lenssen. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

[67] James D. Foley, Andries van Dam, Steven K. Feiner, and John Hughes. *Computer Graphics: Principles and Practice.* The Systems Programming Series. 1996.

[68] Message Passing Interface Forum. MPI: A Message-Passing Interface Standard. `https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf`.

[69] M.L. Fredman and R.E. Tarjan. Fibonacci Heaps And Their Uses In Improved Network Optimization Algorithms. In *FOCS*, 1984.

[70] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx Adaptive Compute Acceleration Platform: Versal Architecture. In *FPGA*, 2019.

[71] Robert Geisberger, Peter Sanders, and Dominik Schultes. Better Approximation of Betweenness Centrality. In *ALENEX*, 2008.

[72] Tong Geng, Ang Li, Runbin Shi, Chunshu Wu, Tianqi Wang, Yanfei Li, Pouya Haghi, Antonino Tumeo, Shuai Che, Steve Reinhardt, and Martin C. Herbordt. AWB-GCN: A Graph Convolutional Network Accelerator with Runtime Workload Rebalancing. In *MICRO*, 2020.

[73] Gentryx. 2D von Neumann Stencil. https://commons.wikimedia.org/wiki/File:2D_von_Neumann_Stencil.svg, 2011.

[74] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *SODA*, 2005.

[75] Hayit Greenspan, Bram Van Ginneken, and Ronald M. Summers. Guest Editorial Deep Learning in Medical Imaging: Overview and Future Promise of an Exciting New Technique. *TMI*, 35(5), 2016.

[76] Licheng Guo, Yuze Chi, Jie Wang, Jason Lau, Weikang Qiao, Ecenur Ustun, Zhiru Zhang, and Jason Cong. AutoBridge: Coupling Coarse-Grained Floorplanning and Pipelining for High-Frequency HLS Design on Multi-Die FPGAs. In *FPGA*, 2021.

[77] Licheng Guo, Jason Lau, Yuze Chi, Jie Wang, Cody Hao Yu, Zhe Chen, Zhiru Zhang, and Jason Cong. Analysis and Optimization of the Implicit Broadcasts in FPGA HLS to Improve Maximum Frequency. In *DAC*, 2020.

[78] Licheng Guo, Pongstorn Maidee, Yun Zhou, Chris Lavin, Jie Wang, Yuze Chi, Weikang Qiao, Alireza Kaviani, Zhiru Zhang, and Jason Cong. RapidStream: Parallel Physical Implementation of FPGA HLS Designs. In *FPGA*, 2022.

[79] Ameer Haj-Ali, Qijing Huang, William Moses, John Xiang, Krste Asanovic, John Wawrzynek, and Ion Stoica. AutoPhase: Juggling HLS Phase Orderings in Random Forests with Deep Reinforcement Learning. In *MLSys*, 2020.

[80] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A High-Performance and Energy-Efficient Accelerator for Graph Analytics. In *MICRO*, 2016.

[81] J. Hammes, A.P.W. Bohm, C. Ross, M. Chawathe, B. Draper, R. Rinker, and W. Najjar. Loop Fusion and Temporal Common Subexpression Elimination in Window-based Loops. In *IPDPS*, 2001.

[82] James Hegarty, John Brunhaver, Zachary DeVito, Jonathan Ragan-Kelley, Noy Cohen, Steven Bell, Artem Vasilyev, Mark Horowitz, and Pat Hanrahan. Darkroom: Compiling High-Level Image Processing Code into Hardware Pipelines. In *SIGGRAPH*, 2014.

[83] Gopalakrishna Hegde and Nachiket Kapre. Energy-Efficient Acceleration of OpenCV Saliency Computation Using Soft Vector Processors. In *FCCM*, 2015.

[84] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8), 1978.

[85] Justin Holewinski, Louis Noël Pouchet, and P. Sadayappan. High-Performance Code Generation for Stencil Computations on GPU Architectures. In *ICS*, 2012.

[86] Hsuan Hsiao and Jason Anderson. Thread Weaving: Static Resource Scheduling for Multithreaded High-Level Synthesis. In *DAC*, 2019.

[87] Yuwei Hu, Yixiao Du, Ecenur Ustun, and Zhiru Zhang. GraphLily: Accelerating Graph Linear Algebra on HBM-Equipped FPGAs. In *ICCAD*, 2021.

[88] Muhuan Huang, Di Wu, Cody Hao Yu, Zhenman Fang, Matteo Interlandi, Tyson Condie, and Jason Cong. Programming and Runtime Support to Blaze FPGA Accelerator Deployment at Datacenter Scale. In *SoCC*, 2016.

[89] Oxford Instruments. Dragonfly 200 High Speed Confocal Microscope System. `https://andor.oxinst.com/products/dragonfly-confocal-microscope-system`, 2021.

[90] Intel. Intel FPGA SDK for OpenCL Pro Edition: Programming Guide, 2020.

[91] Intel. Intel High Level Synthesis Compiler Pro Edition: User Guide, 2020.

[92] Al Shahna Jamal, Eli Cahill, Jeffrey Goeders, and Steven J. E. Wilton. Fast Turnaround HLS Debugging using Dependency Analysis and Debug Overlays. *TRETS*, 13(1), 2020.

[93] Jiantong Jiang, Zeke Wang, Xue Liu, Juan Gómez-Luna, Nan Guan, Qingxu Deng, Wei Zhang, and Onur Mutlu. Boyi: A Systematic Framework for Automatically Deciding the Right Execution Model of OpenCL Applications on FPGAs. In *FPGA*, 2020.

[94] Donald B. Johnson. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM*, 24(1), 1977.

[95] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically Scheduled High-level Synthesis. In *FPGA*, 2018.

[96] Lana Josipović, Shabnam Sheikhha, Andrea Guerrieri, Paolo Ienne, and Jordi Cortadella. Buffer Placement and Sizing for High-Performance Dataflow Circuits. In *FPGA*, 2020.

[97] Gilles Kahn. The Semantics of a Simple Language for Parallel Programming. In *IFIP*, 1974.

[98] Shinpei Kato, Eijiro Takeuchi, Yoshio Ishiguro, Yoshiki Ninomiya, Kazuya Takeda, and Tsuyoshi Hamada. An Open Approach to Autonomous Vehicles. *Micro*, 35(6), 2015.

[99] Soroosh Khoram, Jialiang Zhang, Maxwell Strange, and Jing Li. Accelerating Graph Analytics by Co-Optimizing Storage and Access on an FPGA-HMC Platform. In *FPGA*, 2018.

[100] Ildoo Kim. TF Pose Estimation. `https://github.com/ildoonet/tf-pose-estimation`, 2019.

[101] Thomas N. Kipf and Max Welling. Semi-Supervised Classification with Graph Convolutional Networks. In *ICLR*, 2017.

[102] Donald Ervin Knuth. *Fundamental Algorithms. The Art of Computer Programming 1*. 1997.

[103] Mostafa Koraei, Omid Fatemi, and Magnus Jahre. DCMI: A Scalable Strategy for Accelerating Iterative Stencil Loops on FPGAs. *TACO*, 16(4), 2019.

[104] Oliver Kowalke. Boost Library Documentation, Coroutine2. `https://boost.org/doc/libs/1_65_0/libs/coroutine2/doc/html/coroutine2/intro.html`, 2014.

[105] Sriram Krishnamoorthy, Muthu Baskaran, Uday Bondhugula, J. Ramanujam, Atanas Rountev, and P. Sadayappan. Effective Automatic Parallelization of Stencil Computations. In *PLDI*, 2007.

[106] Stefan Kronawitter, Sebastian Kuckuk, and Christian Lengauer. Redundancy Elimination in the ExaStencils Code Generator. In *ICA3PP*, 2016.

[107] Milind Kulkarni, Keshav Pingali, Bruce Walter, Ganesh Ramanarayanan, Kavita Bala, and L. Paul Chew. Optimistic Parallelism Requires Abstractions. In *PLDI*, 2007.

[108] Aapo Kyrola, Guy Blelloch, and Carlos Guestrin. GraphChi: Large-Scale Graph Computation on Just a PC. In *OSDI*, 2012.

[109] Peggy Lachmann-Anke and Marco Lachmann-Anke. Question Mark Image. `https://pixabay.com/illustrations/question-mark-question-response-1019820/`, 2011.

[110] Yi-Hsiang Lai, Yuze Chi, Yuwei Hu, Jie Wang, Cody Hao Yu, Yuan Zhou, Jason Cong, and Zhiru Zhang. HeteroCL: A Multi-Paradigm Programming Infrastructure for Software-Defined Reconfigurable Computing. In *FPGA*, 2019.

[111] Duncan H. Lawrie. Access and Alignment of Data in an Array Processor. *ToC*, C-24(12), 1975.

[112] Edward A. Lee and David G. Messerschmitt. Synchronous Data Flow. *IEEE*, 75(9), 1987.

[113] Hyuk-Jae Lee, James P. Robertson, and José A.B. Fortes. Generalized Cannon's Algorithm for Parallel Matrix Multiplication. In *ICS*, 1997.

[114] Guoqing Lei, Yong Dou, Rongchun Li, and Fei Xia. An FPGA Implementation for Solving the Large Single-Source-Shortest-Path Problem. *Transactions on Circuits and Systems II*, 63(5), 2016.

[115] Charles E. Leiserson. Systolic Priority Queues. Technical report, 1979.

[116] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker Graphs: An Approach to Modeling Networks. *Journal of Machine Learning Research*, 11, 2010.

[117] Jure Leskovec, Kevin J. Lang, Anirban Dasgupta, and Michael W. Mahoney. Community Structure in Large Networks: Natural Cluster Sizes and the Absence of Large Well-Defined Clusters. *Internet Mathematics*, 6(1), 2009.

[118] Jiajie Li, Yuze Chi, and Jason Cong. HeteroHalide: From Image Processing DSL to Efficient FPGA Acceleration. In *FPGA*, 2020.

[119] Rui Li, Muye Zhu, Junning Li, Michael S. Bienkowski, Nicholas N. Foster, Hanpeng Xu, Tyler Ard, Ian Bowman, Changle Zhou, Matthew B. Veldman, X. William Yang, Houri Hintiryan, Junsong Zhang, and Hong Wei Dong. Precise segmentation of densely interweaving neuron clusters using G-Cut. *Nature Communications*, 10(1), 2019.

[120] The Graph 500 List. June 2021 SSSP. `https://graph500.org/?page_id=944`, 2021.

[121] Rastislav Lukac. *Computational Photography: Methods and Applications*. 2016.

[122] Crovella M, Bianchini R, LeBlanc T, Markatos E, and Wisniewski R. Using Communication-to-Computation Ratio in Parallel Program Design and Performance Prediction. In *SPDP*, 1992.

[123] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. TAPAS: Generating Parallel Accelerators from Parallel Programs. In *MICRO*, 2018.

[124] Karl Marrett, Muye Zhu, Chris Choi, Yuze Chi, Zhe Chen, Hong-Wei Dong, Chang Sin Park, X. William Yang, and Jason Cong. Recut: A Concurrent Framework for Sparse Reconstruction of Neuronal Morphology. `https://www.biorxiv.org/content/10.1101/2021.12.07.471686v1`, 2021.

[125] Naoya Maruyama, Tatsuo Nomura, Kento Sato, and Satoshi Matsuoka. Physis: An Implicitly Parallel Programming Model for Stencil Computations on Large-Scale GPU-Accelerated Supercomputers. In *SC*, 2011.

[126] Julian Mcauley. Learning to Discover Social Circles in Ego Networks. In *NIPS*, 2012.

[127] U. Meyer and P. Sanders. Δ-stepping: A parallelizable shortest path algorithm. *Journal of Algorithms*, 49(1), 2003.

[128] Jeff Meyerson. The Go Programming Language. *IEEE Software*, 31(5), 2014.

[129] Alan Mislove, Hema Swetha Koppula, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Growth of the Flickr Social Network. In *WOSN*, 2008.

[130] Alan Mislove, Massimiliano Marcon, Krishna P. Gummadi, Peter Druschel, and Bobby Bhattacharjee. Measurement and analysis of online social networks. In *IMC*, 2007.

[131] Arjun Mukherjee, Bing Liu, and Natalie Glance. Spotting fake reviewer groups in consumer reviews. In *WWW*, 2012.

[132] Ravi Teja Mullapudi, Andrew Adams, Dillon Sharlet, Jonathan Ragan-Kelley, and Kayvon Fatahalian. Automatically Scheduling Halide Image Processing Pipelines. In *SIGGRAPH*, 2016.

[133] Richard C. Murphy, Kyle B. Wheeler, Brian W. Barrett, and James A. Ang. Introducing the Graph 500. In *CUG*, 2010.

[134] Ken Museth. VDB: High-Resolution Sparse Volumes with Dynamic Topology. *ACM Transactions on Graphics*, 32(3), 2013.

[135] Giuseppe Natale, Giulio Stramondo, Pietro Bressana, Riccardo Cattaneo, Donatella Sciuto, and Marco D. Santambrogio. A Polyhedral Model-Based Framework for Dataflow Implementation on FPGA Devices of Iterative Stencil Loops. In *ICCAD*, 2016.

[136] Maxim Naumov, L. Chien, Philippe Vandermersch, and Ujval Kapasi. CUSPARSE library. In *GPU Technology Conference*, 2010.

[137] Nicoguaro. Illustration of the Heat equation. `https://commons.wikimedia.org/wiki/File:Heat.gif`, 2017.

[138] Eriko Nurvitadhi, Gabriel Weisz, Yu Wang, Skand Hurkat, Marie Nguyen, James C. Hoe, José F. Martínez, and Carlos Guestrin. GraphGen: An FPGA Framework for Vertex-Centric Graph Computation. In *FCCM*, 2014.

[139] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank Citation Ranking: Bringing Order to the Web. Technical report, 1998.

[140] Philippos Papaphilippou, Jiuxi Meng, and Wayne Luk. High-Performance FPGA Network Switch Architecture. In *FPGA*, 2020.

[141] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *NeurIPS*, 2019.

[142] James L Peterson. Petri Nets. *ACM Computing Surveys*, 9(3), 1977.

[143] Larry L. Peterson and Bruce S. Davie. *Computer Networks: A Systems Approach*. 2010.

[144] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming Heterogeneous Systems from an Image Processing DSL. *TACO*, 14(3), 2017.

[145] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, Michael Haselman, Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi Xiao, and Doug Burger. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services Andrew. In *ISCA*, 2014.

[146] Jonathan Ragan-Kelley, Andrew Adams, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Decoupling Algorithms from Schedules for Easy Optimization of Image Processing Pipelines. In *SIGGRAPH*, 2012.

[147] Oliver Reiche, M. Akif Özkan, Richard Membarth, Jürgen Teich, and Frank Hannig. Generating FPGA-based Image Processing Accelerators with Hipacc. In *ICCAD*, 2017.

[148] Ryan A. Rossi and Nesreen K. Ahmed. The network data repository with interactive graph analytics and visualization. In *AAAI*, 2015.

[149] Amitabha Roy, Ivo Mihailovic, and Willy Zwaenepoel. X-Stream: Edge-centric Graph Processing using Streaming Partitions. In *SOSP*, 2013.

[150] Zhenyuan Ruan, Tong He, Bojie Li, Peipei Zhou, and Jason Cong. ST-Accel: A High-Level Programming Platform for Streaming Applications on FPGA. In *FCCM*, 2018.

[151] Vladimir Rybalkin and Norbert Wehn. When Massive GPU Parallelism Ain't Enough: A Novel Hardware Architecture of 2D-LSTM Neural Network. In *FPGA*, 2020.

[152] Tim Schmidt, Guantao Liu, and Rainer Dömer. Exploiting Thread and Data Level Parallelism for Ultimate Parallel SystemC Simulation. In *DAC*, 2017.

[153] Prithviraj Sen, Galileo Mark Namata, Mustafa Bilgic, Lise Getoor, Brian Gallagher, and Tina Eliassi-Rad. Collective Classification in Network Data. *AI Magazine*, 29(3), 2008.

[154] Zhiyuan Shao, Ruoshi Li, Diqing Hu, Xiaofei Liao, and Hai Jin. Improving Performance of Graph Processing on FPGA-DRAM Platform by Two-level Vertex Caching. In *FPGA*, 2019.

[155] Alfonso Shimbel. Structural parameters of communication networks. *The Bulletin of Mathematical Biophysics*, 15(4), 1953.

[156] Karen Simonyan and Andrew Zisserman. Very Deep Convolutional Networks for Large-Scale Image Recognition. In *ICLR*, 2015.

[157] Atefeh Sohrabizadeh, Jie Wang, and Jason Cong. End-to-End Optimization of Deep Learning Applications. In *FPGA*, 2020.

[158] 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. In *FPGA*, 2022.

[159] Greg Stitt, Abhay Gupta, Madison N. Emas, David Wilson, and Austin Baylis. Scalable Window Generation for the Intel Broadwell + Arria 10 and High-Bandwidth FPGA Systems. In *FPGA*, 2018.

[160] Yasuhiro Takei, Masanori Hariyama, and Michitaka Kameyama. Evaluation of an FPGA-Based Shortest-Path-Search Accelerator. In *PDPTA*, 2015.

[161] James Thomas, Pat Hanrahan, and Matei Zaharia. Fleet: A Framework for Massively Parallel Streaming on FPGAs. In *ASPLOS*, 2020.

[162] Erik Todeschini. Augmented-reality signature capture, 2016.

[163] Jie Wang, Licheng Guo, and Jason Cong. AutoSA: A Polyhedral Compiler for High-Performance Systolic Arrays on FPGA. In *FPGA*, 2021.

[164] Kai Wang, Don Fussell, and Calvin Lin. A Fast Work-Efficient SSSP Algorithm for GPUs. In *PPoPP*, 2021.

[165] Qinggang Wang, Long Zheng, Yu Huang, Pengcheng Yao, Chuangyi Gui, Xiaofei Liao, Hai Jin, Wenbin Jiang, and Fubing Mao. GraSU: A Fast Graph Update Library for FPGA-based Dynamic Graph Processing. In *FPGA*, 2021.

[166] Shuo Wang and Yun Liang. A Comprehensive Framework for Synthesizing Stencil Algorithms on FPGAs using OpenCL Model. In *DAC*, 2017.

[167] Yu Wang, James C. Hoe, and Eriko Nurvitadhi. Processor Assisted Worklist Scheduling for FPGA Accelerated Graph Processing on a Shared-Memory Platform. In *FCCM*, 2019.

[168] Markus Wittmann, Georg Hager, and Gerhard Wellein. Multicore-Aware Parallel Temporal Blocking of Stencil Codes for Shared and Distributed Memory. In *IPDPSW*, 2010.

[169] Wm. A. Wulf and Sally A. McKee. Hitting the Memory Wall. *SIGARCH Comput. Archit. News*, 23(1), 1995.

[170] Xilinx. Xilinx xfOpenCV Library. `https://github.com/Xilinx/xfopencv`, 2019.

[171] Xilinx. Vivado Design Suite User Guide: High-Level Synthesis (UG902), 2020.

[172] Mingyu Yan, Lei Deng, Xing Hu, Ling Liang, Yujing Feng, Xiaochun Ye, Zhimin Zhang, Dongrui Fan, and Yuan Xie. HyGCN: A GCN Accelerator with Hybrid Architecture. In *HPCA*, 2020.

[173] Tanner Young-Schultz, Lothar Lilge, Stephen Brown, and Vaughn Betz. Using OpenCL to Enable Software-like Development of an FPGA-Accelerated Biophotonic Cancer Treatment Simulator. In *FPGA*, 2020.

[174] Wayne W. Zachary. An Information Flow Model for Conflict and Fission in Small Groups. *Journal of Anthropological Research*, 33(4), 1977.

[175] Hanqing Zeng and Viktor Prasanna. GraphACT: Accelerating GCN training on CPU-FPGA heterogeneous platforms. In *FPGA*, 2020.

[176] Bingyi Zhang, Rajgopal Kannan, and Viktor Prasanna. BoostGCN: A Framework for Optimizing GCN Inference on FPGA. In *FCCM*, 2021.

[177] Jialiang Zhang and Jing Li. Degree-aware Hybrid Graph Traversal on FPGA-HMC Platform. In *FPGA*, 2018.

[178] Yichi Zhang, Junhao Pan, Xinheng Liu, Hongzheng Chen, Deming Chen, and Zhiru Zhang. FracBNN: Accurate and FPGA-Efficient Binary Neural Networks with Fractional Activations. In *FPGA*, 2021.

[179] Tuowen Zhao, Mary Hall, Protonu Basu, Samuel Williams, and Hans Johansen. Exploiting Reuse and Vectorization in Blocked Stencil Computations on CPUs and GPUs. In *SC*, 2019.

[180] Shijie Zhou, Charalampos Chelmis, and Viktor K. Prasanna. Accelerating Large-Scale Single-Source Shortest Path on FPGA. In *IPDPSW*, 2015.

[181] Shijie Zhou, Rajgopal Kannan, Viktor K. Prasanna, Guna Seetharaman, and Qing Wu. HitGraph: High-throughput Graph Processing Framework on FPGA. *TPDS*, 2019.

[182] Xiaowei Zhu, Wentao Han, and Wenguang Chen. GridGraph : Large-Scale Graph Processing on a Single Machine Using 2-Level Hierarchical Partitioning. In *ATC*, 2015.

[183] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. Combined Spatial and Temporal Blocking for High-Performance Stencil Computation on FPGAs Using OpenCL. In *FPGA*, 2018.