DISS. ETH NO. 27995

Productive FPGA Programming for High-Performance Computing

A thesis submitted to attain the degree of

DOCTOR OF SCIENCES of ETH ZURICH
(Dr. sc. ETH Zurich)

presented by

JOHANNES DE FINE LICHT

Master of Science in Computer Science
University of Copenhagen

born on 08.11.1991

citizen of Denmark

accepted on the recommendation of

Prof. Dr. Torsten Hoefler (ETH Zurich), examiner
Prof. Dr. Gustavo Alonso (ETH Zurich), co-examiner
Dr. Michaela Blott (Xilinx), co-examiner
Dr. Michael Kinsner (Intel), co-examiner

2021

“I will go mad!” [Arthur] announced. (...)
“I went mad for a while,” said Ford, “did me no end of good.”

Life, the Universe and Everything
Douglas Adams

Abstract

For decades, the computational performance of processors has grown at a faster rate than the available memory bandwidth. As a result, most transistors in modern processors are spent on managing data movement via caches and registers. Spatial computing architectures can omit general purpose caches, registers, and control logic by implementing application-specific dataflow, where computations are laid out spatially. Programmable spatial architectures, such as FPGAs, can implement application-specific dataflow, but the steep learning curve of hardware programming prevents widespread adoption in high-performance computing (HPC). In this dissertation, we address this programmability gap. High-level synthesis (HLS) has increased productivity when designing FPGA architectures, but traditional software optimizations are insufficient to implement high-performance hardware architectures. To alleviate this, we present a set of key transformations for HLS, targeting scalable architectures for HPC applications, identifying classes of transformations and their effect in hardware, and boost the productivity of HLS developers with the hlslib open source project of productivity tools. Using these techniques, we present a model-based, end-to-end example of optimizing matrix multiplication for FPGAs, which yields competitive performance in practice and is published as an open source project.

Venturing beyond HLS, we propose a new way to develop, optimize, and compile FPGA programs. The Data-Centric parallel programming (DaCe) framework allows applications to be defined by their dataflow and control flow through the Stateful DataFlow multiGraph (SDFG) representation, exposing a plethora of optimization opportunities. We unify general, domain-specific, and platform-specific optimizations in this flow, and present the FPGA backends of DaCe, emitting efficient HLS code for both Xilinx and Intel devices. Building on this infrastructure, we present StencilFlow, an end-to-end framework that maps general directed acyclic graphs of heterogeneous stencil operators to distributed FPGA architectures, maximizing temporal locality and ensuring deadlock freedom. We show the highest performance recorded for stencil programs for either FPGA vendor to date, and study a complex stencil program from a production weather simulation application. With the toolbox of transformations, open source software, and programming abstractions provided in this dissertation, we contribute to the productivity of HLS developers, performance engineers, domain scientists, and compiler engineers alike, bridging the gap for bringing spatial computing systems into the mainstream of HPC.

Zusammenfassung

Die Rechenleistung von Prozessoren wächst seit Jahrzehnten schneller als die verfügbare Speicherbandbreite. Infolgedessen werden die meisten Transistoren in modernen Prozessoren für die Verwaltung der Datenbewegung über Caches und Register verwendet. Spatial Computing-Architekturen können auf Allzweck-Caches, -Register und -Steuerlogik verzichten, indem ein anwendungsspezifischer Datenfluss implementiert wird, bei dem Berechnungen in Schaltkreise übersetzt werden. Programmierbare Architekturen wie FPGAs können anwendungsspezifischen Datenfluss implementieren, aber die steile Lernkurve der Hardwareprogrammierung behindert eine weit verbreitete Einführung in High-Performance Computing (HPC). In dieser Dissertation schlagen wir Lösungen vor, die das Programmieren von FPGAs vereinfachen. High-Level Synthesis (HLS) hat die Produktivität bei der Entwicklung von FPGA-Architekturen erhöht, aber bekannte Softwareoptimierungen reichen nicht aus, um leistungsstarke Hardwarearchitekturen zu implementieren. Wir präsentieren eine Reihe zentraler Transformationen für HLS, die auf skalierbare Architekturen für HPC-Anwendungen abzielen, wir identifizieren Transformationsklassen und deren Auswirkungen in der Hardware, und steigern die Produktivität von HLS-Entwicklern mit dem Open-Source-Projekt hlslib. Mit diesen Techniken konstruieren wir ein komplettes Beispiel für die Optimierung von Matrixmultiplikation, das konkurrenzfähige Leistung in der Praxis erbringt und als Open-Source veröffentlicht wird. Wir gehen über HLS hinaus und schlagen einen neuen Ansatz zur Entwicklung, Optimierung und Kompilierung von FPGA-Programmen vor. Das Data-Centric Parallel Programming (DaCe)-Framework ermöglicht die Definition von Programmen via ihrem Datenfluss und Kontrollfluss durch die Stateful DataFlow multiGraph (SDFG)-Repräsentation. Hierdurch wird eine Vielzahl von Optimierungsmöglichkeiten offengelegt. Wir vereinen allgemeine, domänenspezifische und plattformspezifische Optimierungen in diesem Prozess und präsentieren die FPGA-Backends von DaCe, welche effizienten HLS-Code sowohl für Xilinx-, als auch Intel-Geräte ausgeben. Basierend auf dieser Infrastruktur präsentieren wir StencilFlow, ein Framework das beliebige gerichtete azyklische Graphen heterogener Stencil-Operationen auf verteilte FPGA-Architekturen abbildet, die Temporal Locality maximiert und Deadlock-Freiheit gewährleistet. Wir zeigen die bisher höchste Leistung für Stencil-Programme für beide FPGA-Anbieter und untersuchen ein komplexes Stencil-Program aus einer in Verwendung stehenden Applikation zur Wettersimulation. Mit dieser Toolbox von Transformatio-

nen, Open-Source-Software und Programmierabstraktionen, die in dieser Dissertation erstellt werden, tragen wir zur Produktivität von HLS-Entwicklern, Performance Engineers, Naturwissenschaftler und Compiler-Entwicklern bei und tragen damit dazu bei Spatial Computing-Systeme in den Mainstream der HPC zu führen.

Acknowledgements

All my work was done under the expert guidance of Prof. Torsten Hoefler, whose insightful advice, feedback, and ideas inspired me and helped me navigate through the quirks and pitfalls of academia. His drive to always look into the future to stay ahead of the curve, maintain the big picture even when working on the smallest niche, and his commitment to high quality in all endeavours, has been instrumental in building my academic personality.

Research is in its essence a collaborative effort. This work presented in this dissertation builds on collaboration and conversations with numerous brilliant researchers and engineers, including (in alphabetical order), but not limited to: Alexandros “Alexnick” Nikolaos Ziogas, Andreas Kuster, Carl Johnsen, Chris Pattison, Dario Korolija, David Sidler, Dominic Hofer, Grzegorz “Greg” Kwasniewski, Jakub “Kuba” Beránek, Kaan Kara, Maciej Besta, Manuel Burger, Muhsen Owaida, Nick Fraser, Philipp Schaad, Sabela Ramos, Simon Meierhans, Timo Schneider, Tobias Gysi, Yaman Umuroglu, Yishai Oltchik, and Zsolt István, with special thanks to Tal Ben-Nun for his tireless mentorship throughout my PhD, always motivated and ready to help in the most dire of deadline crunches; to Tiziano “Tizi” De Matteis for being my loyal partner in suffering during our many joint projects; to Alexandru “Lex” Calotoiu for his valuable feedback on this document; to the many bright bachelor and master students who helped build and inspire much of our work (full list on page xi); to the SPCL Slide Committee; and to all of my SPCL and Systems Group colleagues for the many fun activities, cold beer, conference excursions in dubious white vans, and dank memes that we have shared throughout the years.

My gratitude goes out to Michaela Blott and Blake Pelton for their mentorship during my internships at the Xilinx research labs in Dublin, and at Microsoft in Redmond, respectively. The inspiration, knowledge, and perspective that I gathered from them, and the excellent teams that they have assembled at their respective institutions, have been invaluable for my growth as a researcher and as an engineer. Thank you to Gustavo for his mentorship during our joint time in the Systems Group, and to Mike, who along with Michaela kindly agreed to be on my examination committee, fostering exciting discussions on the future of programming spatial devices both during and after the defense. A warm thank you to Børge Svane Nielsen for his crucial mentorship in my early years, starting my international journey by setting up my internship at CERN, where my passion for HPC

and the international community was kindled, leading me to where I am today.

I wish to thank Xilinx for generous donations of software, hardware, and access to the Xilinx Adaptive Compute Cluster (XACC) at ETH Zurich, with special thanks to Cathal McCabe and Mario Ruiz Noguera from the Xilinx University Program for their excellent work on connecting with academia. I also wish to thank the Swiss National Supercomputing Center (CSCS) for providing computing infrastructure, with special thanks to Hussein Harake for his support on the Greina cluster and receiving me on my visits to Lugano, and Mark Klein for his tireless support on the Ault cluster. I also want to thank the Paderborn Center for Parallel Computing (PC^2) for the generous allocation of distributed FPGA computing infrastructure on the Noctua cluster, with special thanks to Christian Plessl and Tobias Kenter for their support, and for hosting me on a visit to Paderborn.

My work was supported by the European Research Council under the European Union's Horizon 2020 programme (grant agreement DAPP, No. 678880).

Finally, a heartfelt thank you to my parents, Irene Spranger and Nikolaj de Fine Licht, and my grandmother, Lene Steners, for your unconditional and unfaltering love and support in my every undertaking; to my grandfather, Kjeld de Fine Licht, who set the precedent as the only person in our family before me to pursue doctoral studies; to my amazing siblings, Theresa and Teodor; to Anna, Erik, Keren, Lilia, Magnus, Oliver, and Philip for being my extra family in Copenhagen and always having your doors open to me; to all of the Christmas/Easter Lunch crew for your loyalty and for being plain awesome; and to Alexandra Rollings, for your love and companionship on our parallel doctorate journeys.

Publications

Publications that have contributed content and/or results to this dissertation:

- Johannes de Fine Licht, Maciej Besta, Simon Meierhans, Torsten Hoefler. "Transformations of High-Level Synthesis Codes for High-Performance Computing". In *IEEE Transactions on Parallel and Distributed Systems (TPDS)* vol. 32, no. 5, pp. 1014–1029, May 2021 [36].

- Johannes de Fine Licht, Torsten Hoefler. "hlslib: Software Engineering for Hardware Design". Presented at the Fifth International Workshop on Heterogeneous High-Performance Reconfigurable Computing (H2RC'19). Extended abstract available at arXiv:1910.04436, November 2019 [37].

- Johannes de Fine Licht, Grzegorz Kwasniewski, Torsten Hoefler. "Flexible Communication Avoiding Matrix Multiplication on FPGA with High-Level Synthesis". In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*, pp. 244—254, February 2020 [39].

- Tal Ben-Nun, Johannes de Fine Licht, Alexandros Nikolaos Ziogas, Timo Schneider, Torsten Hoefler. "Stateful Dataflow Multigraphs: A Data-Centric Model for Performance Portability on Heterogeneous Architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*, article 81, ACM, November 2019 [16].

- Alexandros Nikolaos Ziogas, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Tiziano De Matteis, Johannes de Fine Licht, Luca Lavarini, Torsten Hoefler. "Productivity, Portability, Performance: Data-Centric Python". In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'21)*, November 2021 [164].

- Johannes de Fine Licht, Andreas Kuster, Tiziano De Matteis, Tal Ben-Nun, Dominic Hofer, Torsten Hoefler, "StencilFlow: Mapping Large Stencil Programs to Distributed Spatial Computing Systems". In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'21)*, pp. 315–326, March 2021 [38].

When relevant, the dissertation will contain a statement of authorship, declaring the content's relation to the publications listed here, and the scope of each collaborator's contribution. Prof. Hoefler's role as advisor to all my work is implicitly assumed.

Other publications that I authored or co-authored during my doctoral studies, in descending chronological order of publication:

- Tiziano De Matteis, Johannes de Fine Licht, Torsten Hoefler. "FBLAS: Streaming Linear Algebra on FPGA". In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20)*, November 2020 [41].
- Tiziano De Matteis, Johannes de Fine Licht, Jakub Beránek, Torsten Hoefler. "Streaming Message Interface: High-Performance Distributed Memory Programming on Reconfigurable Hardware". In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '19)*, article 82, ACM, November 2019 [40].
- Maciej Besta, Dimitri Stanojevic, Johannes de Fine Licht, Tal Ben-Nun, Torsten Hoefler. "Graph Processing on FPGAs: Taxonomy, Survey, Challenges". arXiv:1903.06697, April 2019 [20].
- Maciej Besta, Marc Fischer, Tal Ben-Nun, Dimitri Stanojevic, Johannes de Fine Licht, Torsten Hoefler. "Substream-Centric Maximum Matchings on FPGA". In *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 13, no. 2, pp. 152–161, February 2019 [19].

Student Projects

During my doctorate, I supervised or co-supervised numerous student projects that inspired and contributed to our research, including (in descending chronological order):

- Jannis Widmer, “Accelerating FPGA Applications in DaCe with High-Bandwidth Memory”, Bachelor Thesis, August 2021. Co-supervised with Tiziano De Matteis and Tal Ben-Nun.
- Marc Widmer, “Double-Pumping with DaCe-RTL”, Student Research Project, August 2021. Co-supervised with Tiziano De Matteis and Tal Ben-Nun.
- Manuel Burger, “Data-Centric FPGA Programming for Machine Learning with Multi-Level Design”, Research in Data Science project, July 2021. Co-supervised with Tiziano De Matteis and Tal Ben-Nun.
- Andreas Kuster, “StencilFlow: Stencil Dataflow on Reconfigurable Hardware”, Bachelor Thesis, August 2020. Contributed to our CGO’21 paper [38].
- Manuel Burger, “Portable Linear Algebra on FPGA using Data-Centric Parallel Programming”, Bachelor Thesis, June 2020. Co-supervised with Tiziano De Matteis and Tal Ben-Nun. **Winner of the PhD category of the Xilinx Open Hardware Competition 2020**[1], co-authored with Johannes de Fine Licht.
- Pascal Störzbach, “Specialized Hardware Architectures for Dense Linear Algebra”, February 2019. Co-supervised with Maciej Besta.
- Dimitri Stanojevic, “High-Performance Graph Processing on FPGAs”, Master Thesis, February 2019. Co-supervised with Maciej Besta.
- Houssam Naous, “A Modular Framework for Training Deep Neural Networks on FPGAs using OpenCL”, Master Thesis, September 2018. Co-supervised with Tal Ben-Nun.
- Simon Meierhans, “Massively Parallel N-Body Simulation on FPGA”, Student Research Project, April 2018. Contributed to our TPDS paper [36].
- Roman Cattaneo, “High-Level Synthesis of Dense Matrix Operations on FPGA”, Master Thesis, July 2017.

---

[1] `http://www.openhw.eu/2020-results.html`

Contents

Abstract iii

Zusammenfassung v

Acknowledgements vii

Publications ix

Student Projects xi

1 Introduction 1

1.1 The Evolution of Hardware Scaling 1

1.2 Spatial Computing Architectures 4

1.3 Programming Spatial Architectures 5

2 Transforming High-Level Synthesis Codes for High-Performance Computing 9

2.1 Optimization of Hardware Programs 9

2.1.1 From Imperative Code to Hardware 10

2.1.2 Key Transformations for High-Level Synthesis 12

2.1.3 The Importance of Pipelining 13

2.1.4 Optimization Goals 14

2.2 Pipeline-Enabling Transformations 14

2.2.1 Accumulation Interleaving 15

2.2.1.1 Full Transposition 15

2.2.1.2 Tiled Accumulation Interleaving 17

2.2.1.3 Single-Loop Accumulation Interleaving 18

2.2.1.4 Batched Accumulation Interleaving 18
2.2.2 Delay Buffering 19
2.2.3 Random Access Buffering 21
2.2.4 Pipelined Loop Fusion 22
2.2.5 Pipelined Loop Switching 24
2.2.6 Pipelined Loop Flattening/Coalescing 25
2.2.7 Inlining 26
2.3 Scalability Transformations 26
2.3.1 Horizontal Unrolling (Vectorization) 27
2.3.2 Vertical Unrolling 28
2.3.3 Dataflow 29
2.3.4 Tiling 32
2.4 Memory Access Transformations 33
2.4.1 Memory Access Extraction 33
2.4.2 Memory Buffering 35
2.4.3 Memory Striping 35
2.4.4 Type Demotion 36
2.5 Software Transformations in HLS 37
2.6 End-to-End Examples 37
2.6.1 Stencil Code 39
2.6.2 Matrix Multiplication Code 40
2.6.3 N-Body Code 41
2.7 Related Work 42
2.8 Toolflow of Xilinx vs. Intel 44
2.9 Summary 45

3 hlslib: Software Engineering for Hardware Design 47

3.1 High-Level Synthesis Programming 47

3.2 Improving the HLS Workflow 48
  3.2.1 CMake Integration 48
  3.2.2 Portable OpenCL Host Code 49
  3.2.3 Emulating Multiple Processing Elements in Software 50
3.3 Object-Oriented Hardware Design 52
  3.3.1 Streams/Channels 52
  3.3.2 Wide Data Buses and Vectorization 53
  3.3.3 Shift Registers with Parallel Access 53
  3.3.4 Tree Reduction with Functors 55
3.4 Projects Using hlslib 55
3.5 Summary 56

4 A Case Study with Matrix Multiplication 57
4.1 Minimizing I/O of Linear Algebra Kernels 57
  4.1.1 Minimizing I/O on FPGAs 57
4.2 Optimization Goals 59
  4.2.1 Optimizing Computations 59
  4.2.2 Optimizing I/O 60
  4.2.3 Optimizing Resources 60
  4.2.4 Notation 60
4.3 Optimization Models 62
  4.3.1 Computation Model 62
  4.3.2 I/O Model 63
    4.3.2.1 Memory Resources 64
    4.3.2.2 FPGA-constrained I/O Minimization 64
  4.3.3 Resource Model 65
  4.3.4 Parallelism and Memory Resources 67
4.4 Hardware Mapping 68

4.4.1 Layout of Processing Elements 69

4.4.1.1 Collapsing to a 1D array 69

4.4.1.2 Double buffering 70

4.4.2 Handling Loop-carried Dependencies 71

4.4.3 Optimizing Column-wise Reads 72

4.4.4 Writing Back Results 72

4.4.5 Final Module Layout 72

4.5 Evaluation 73

4.5.1 Parameter Selection 73

4.5.2 Code Complexity 74

4.5.3 Experimental Setup 74

4.5.4 Results 75

4.6 Related Work 78

4.7 Summary 80

5 Data-Centric Design of Spatial Architectures 83

5.1 Data-Centric Parallel Programming 83

5.2 A Primer on SDFGs 85

5.2.1 Tasklets 86

5.2.2 Access Nodes 87

5.2.3 Dataflow Edges 87

5.2.4 Maps 88

5.2.5 Dataflow States 89

5.2.6 Control Flow Edges 89

5.2.7 Library Nodes 91

5.2.8 Nested SDFGs 91

5.3 Transforming the SDFG 92

5.3.1 Example: Vectorization 95

5.3.2 Example: Tiling . . . 97
5.3.3 Example: GPU Offloading . . . 98
5.3.4 Example: Control Flow Loops to Parallel Maps . . . 99
5.4 The DaCe Framework . . . 102
5.4.1 Frontends . . . 102
5.4.2 Code Generation . . . 103
5.4.2.1 Graph Traversal . . . 103
5.4.2.2 Scheduling Computations . . . 104
5.4.2.3 Computations . . . 104
5.4.2.4 Memory Allocation . . . 104
5.4.2.5 Memory Accesses . . . 105
5.4.3 Build System . . . 105
5.4.4 Instrumentation . . . 106
5.4.5 Summary . . . 106
5.5 Multi-Level Design with Library Nodes . . . 107
5.5.1 Domain-Specific Optimizations . . . 110
5.5.2 Targeting Vendor Libraries . . . 111
5.5.3 A Note on MLIR . . . 112
5.6 Data-Centric Programming for FPGAs . . . 112
5.7 From SDFG to Spatial Architecture . . . 113
5.7.1 Code Generating SDFGs . . . 113
5.7.2 Parallelism, Pipelining, and Unrolling with Maps . . . 114
5.7.3 Representing FPGA Kernels . . . 115
5.7.4 Processing Elements . . . 116
5.7.5 Channels/Streams/FIFOs . . . 118
5.7.6 Parametric Processing Elements: Systolic Arrays . . . 119
5.7.7 Memory Hierarchy . . . 120

5.8 Optimizations for Spatial Architectures 121
  5.8.1 Transformation: Offloading to the FPGA 122
  5.8.2 Architecture Specialization 123
  5.8.3 Transformation: Memory Access Extraction 126
  5.8.4 Transformation: Pipeline Fusion 127
  5.8.5 `AXPYDOT` Evaluation 129
5.9 Automatically Compiling NumPy to FPGAs 130
5.10 Summary 133

6 StencilFlow: Domain-Specific Dataflow 135

6.1 Definition of a Stencil Program 137
6.2 Mapping to Distributed Hardware 138
  6.2.1 Mapping to Hardware 138
  6.2.2 Mapping to the Distributed Setting 139
6.3 From DAG to Dataflow 141
  6.3.1 Internal Buffers for Intra-Stencil Reuse 142
  6.3.2 Delay Buffers for Inter-Stencil Reuse 143
  6.3.3 Vectorization 144
6.4 Data-Centric Abstract Representation 145
  6.4.1 Spatial Stencil Fusion 146
6.5 Architecture Generation 147
  6.5.1 Intel Stencil Node Expansion 148
  6.5.2 Xilinx Stencil Node Expansion 149
  6.5.3 Reference Expansion 151
  6.5.4 Generating Distributed Programs 151
6.6 Workflow and Artifacts 151
6.7 Benchmarks 152
  6.7.1 Computing Expected Runtime 153

6.7.2 Experimental Platforms . . . 153
6.7.3 Iterative Stencil Performance . . . 155
6.7.4 Off-Chip Memory Bandwidth . . . 157
6.8 Weather Simulation Application Study . . . 158
6.8.1 Horizontal Diffusion Analysis . . . 160
6.8.2 Horizontal Diffusion Benchmark . . . 161
6.8.3 Silicon Efficiency . . . 162
6.8.4 Spatial Tiling . . . 162
6.9 Related Work . . . 163
6.10 Summary . . . 163

7 Conclusion **165**
7.1 Future Work . . . 167

Bibliography **167**

Chapter 1

Introduction

1.1 The Evolution of Hardware Scaling

Until the mid-2000's, the evolution of computing hardware was dominated by Dennard scaling [45], which postulates that the power density of transistors stay constant when the feature size is shrunk, while voltage and current is reduced, improving performance due to higher transistor counts and higher operating frequency, without increasing the effective power consumption. This was in turn fueled by Moore's "law", which was, in practice, an empirical observation followed by a qualitative guess on how the trend might continue [108]:

> "The complexity for minimum component costs has increased at a rate of roughly a factor of two per year (see graph). Certainly over the short term this rate can be expected to continue, if not to increase. Over the longer term, the rate of increase is a bit more uncertain, although there is no reason to believe it will not remain nearly constant for at least ten years."

Gordon Moore made this statement in 1965, thus making no attempt at projecting beyond 1975 (a time frame in which, indeed, his projection held remarkably well). However, Moore's law is still being cited when discussing the trend in reduction of transistor density 46 years after its stated expiry. Whether it is alive, dead, or anything in between is claimed ad hoc by microchip vendors to suit their current technological competitiveness and marketing narrative, but the core issue is clear: Increases in performance no longer come for free, and most of the burden of progress has been shifted away from microprocessor technology, and onto computer architecture design, along with all the consequences that this brings for programmability.

In the mid-2000's, the era of Dennard scaling came to an end. In fact, recently, shrinking the feature size even *increases* the energy loss in modern microchips. This meant that operating frequencies could no longer be cranked up without significantly increasing the power consumption, so further performance increases had to come from other sources. This marked the beginning of the multi-core era, starting with putting a handful of traditional CPU cores on the same die, and culminating in the golden age of many-core architectures, utilizing hundreds or thousands of small, heavily vectorized cores to yield high throughput at moderate clock frequencies, with GPUs as the main workhorse of modern HPC clusters.

While multi-core and many-core architectures still make up the majority of deployed microchips today, the core concept has yielded its benefit and is now mature, with new generations relying on more incremental architectural improvents to continue improving performance in an era of dead/weakened/not-quite-alive Moore's law.

We have now entered the era of specialized hardware. Instead of waiting for Dennard or Moore to yield higher frequencies and smaller ("better") transistors, boosting performance must now increasingly come from finding new ways to better utilizing the available power budget. Fueled by the explosion of interest (and thereby funding) in machine learning (now more commonly referred to as "AI"), the largest leaps in performance claims recently have come from building architectures that are specialized to solve very specific problems, very efficiently. The most notable first example was Google's TPU, but since its introduction, specialized machine learning hardware has made its way onto the architectures of traditional architectures as well, including Tensor Cores and low-precision instructions on Nvidia GPUs, and the VNNI and bfloat16 instruction sets on Intel CPUs. This hardware reduces the number of instructions needed for the same number of fundamental computations, and most importantly, reduces data movement, both due to fewer instructions required per work done and by shrinking data types altogether, using custom types with properties that favor machine learning workloads [35].

Unfortunately, for most domains outside of machine learning, building application-specified integrated circuits (ASICs) or adding specialized hardware to general purpose processors, is not economically feasible, or not adaptable enough to evolving models and changing requirements. Although some applications might be lucky to find uses for the specialized hardware created with machine learning in mind [68], most smaller domains are not reaping the biggest benefits of the era of specialization.

Simultaneously, another problem has been compounding itself: for decades, the bandwidth of off-chip memory technology has increased at a much slower pace than computational performance [154]. As a result, on most workloads, the increase in performance of the processor effectively has a much smaller proportional benefit to the end user, as most of a program's runtime is often spent waiting on cache misses. This can be observed in state-of-the-art processors today, which only get anywhere close to their advertised peak performance on a very small set of ideal applications, with many workloads sitting in the single-digit percentage of peak — despite the fact that the majority of transistors are already being spent on cache and register files.

The consequences of the memory bottleneck is clear when looking inside a modern processor architecture. Most currently available processor architectures are *load/store architectures* (commonly referred to as "von Neumann" architectures), where data is loaded/stored across a central memory bus to/from a number of central processing cores. These architectures are *temporal*, in that the sequence of instructions of the program are executed on the same central hardware unit, utilized in different ways depending on the given instruction executed. When executing a numerical workload, we can think of the steps required in the processor as some variation of the following steps:

1. Instruction are fetched from the central memory bus, or if possible, from the instruction cache. If a cache miss occurs, the instruction is written to the cache.
2. Instructions are decoded to the format that can be executed by the processor, and addresses of memory locations are translated to their physical addresses.
3. Data required to execute the instruction is loaded from the central memory bus, or if possible, from (one of the multiple layers of) cache, into one or more registers. If a cache miss occurs, the data is written into (one or more layers of) cache.
4. The ALU performs the computation using the data stored in registers.
5. The result of the computation is stored from registers to the central memory bus, or if possible, to cache, where it will be flushed at a later stage.

Out of the steps above, 4 is arguably the *only* step that performs what we would consider a "useful" computation. The remaining steps are primarily orchestration, necessary to route the right data to the ALU and perform the right operations in the right order, before routing the data back to the right place. *All this orchestration exists to fight the memory bottleneck.* As a result, the power consumption of today's load/store architectures are mostly due to the cost of data movement, control logic within the processor, and addressing general purpose registers and cache: On a 45 nm process node, out of 70 pJ spent on an instruction, Marc Horowitz estimated that 0.9 pJ are spent on a 32-bit floating point addition [71]: just 1.3% of the total instruction energy — even before considering off-chip memory, which would be responsible for another 1–2 orders of magnitude more power than the processor instruction. While GPUs reduce this effect by executing more elementary operations per instruction issued, they still rely on general purpose registers, cache, and instructions, and thus do not evade the fundamental issue of power consumption in the load/store paradigm.

1.2 Spatial Computing Architectures

To address the inherent power inefficiency of load/store architectures, we have to consider a fundamentally different paradigm, which eliminates or greatly reduces the presence of *general purpose* cache, registers, and control logic — ideally only utilizing the cache, registers, and control logic that are strictly necessary to execute the program. We can think of such an architecture as a *spatial* architecture (as opposed to a temporal architecture), where computations are laid out spatially across the physical hardware, rather than temporally streaming as a sequence of instructions to the same central hardware unit.

While a spatial architecture that is specialized to solve a target application would be superior to a load/store architecture in terms of power efficiency, the issue of economics and hardware expertise usually makes it infeasible build ASICs for smaller application domains, as is typically the case in scientific applications. Because this dissertation is targeted at high-performance computing workloads, we will thus focus on *programmable* spatial architectures (also referred to as *reconfigurable* hardware). The crucial ongoing effort for such an architecture to be viable is to find a trade-off between programmability and efficiency that offers enough benefits in performance and energy savings to justify the effort required to program it.

The most common class of *programmable* spatial devices are *field-programmable gate arrays* (FPGAs). FPGAs are programmable at a very fine granularity, trading off some of the efficiency gained by shedding the load/store overhead for generality [90]. This allows them to implement arbitrary data types and operations, connected by arbitrarily sized data paths, laid out by expensive placement and routing procedures. Other programmable spatial architectures include the Cerebras [78] deep neural network accelerator, Xilinx' AI Engines [55], and Intel's announced (but not delivered) Configurable Spatial Accelerator [4]. Being the only programmable spatial architecture widely available during the majority of my doctoral studies, FPGAs will be the primary platform of evaluation throughout this dissertation.

1.3 Programming Spatial Architectures

To improve the viability of spatial architectures in high-performance computing in terms of the trade-off between efficiency and programmability, we must either improve the inherent benefit of using such an architecture, or make it easier to achieve the benefits offered. This dissertation focuses on the latter. The performance capabilities of a device are meaningless to a scientist if he cannot program it. If device B promises much higher performance than device A, but device B is much harder to program, most programmers will stick with device A (worse yet, the degrees of "much higher" and "much harder" are rarely known a priori). Our goal is thus to provide scientists with knowledge and tools to productively program spatial systems for high-performance computing, reducing the resistance to explore targeting current and new spatial architectures with their applications.

The primary class of programmable spatial architecture that we will use for evaluation in this work, namely FPGAs, have been around for decades, where they have been used as a vessel to exploit some of the advantages provided by spatial architectures [131, 23]. However, FPGAs are notoriously hard to program [14]. Traditionally, FPGAs have been programmed in hardware description languages (HDLs), such as VHDL or (System)Verilog. These are *structural* languages, which describe the state of every register and wire at a per-cycle level, based on the state of other registers and wires, rather than having a notion of the temporal progression of a program. For programmers that have not been brought up in the hardware domain, HDLs are a challenging change of paradigm with a steep learning curve. Because they exist in a fundamentally different paradigm, HDLs do not benefit from the majority of software engineering techniques that improve programmer productivity and code reliability. While there are still many use cases for HDLs, such as designing latency critical components or working with highly resource constrained designs, the lack of adoption of FPGAs in the HPC community suggests that they are unlikely to become the language of choice for the majority of developers.

High-level synthesis (HLS) tools have established themselves as an alternative to the low-level hardware development offered by HDLs [105, 30], allowing programmers to use familiar procedural languages such as C++ [24] or OpenCL [32] to program FPGAs. HLS abstracts away platform-specific details such as how bus protocols, memory controllers, or floating point units are implemented, introducing some portability between different platforms, and allowing the developer to focus more on the functional aspect of their code. To do so, HLS tools must compile from the procedural, temporal paradigm expressed by

C-like languages into the structural representation of a circuit, which requires the tool to reason about both how the circuit is laid out, and about how the temporally expressed computations are mapped to the circuit. Developers can typically give hints to this process in the form of inline pragmas and compilation, which work to constrain the generated hardware. However, the size of the design space remains huge. When compared to the relationship between source code and CPU assembly, the abstraction of the source code and the abstraction of the architecture are much further removed, requiring the developer to have a better understanding of this mapping and the underlying hardware to achieve satisfactory solutions. As a result, traditional optimization techniques are insufficient for optimizing HLS programs for high-performance computing applications; a shortcoming which we will address in this work, before raising the level of abstraction further.

The remainder of this dissertation is organized into progressive stages of improving the productivity of programming spatial architectures for high-performance computing:

- Chapter 2 surveys the transformations required to optimize the current state-of-the-art programming model for programming spatial devices, namely high-level synthesis, for high-performance computing workloads.
- Chapter 3 gives a short overview of the hlslib open source project, which improves programmer productivity within the HLS programming model.
- Chapter 4 walks through a comprehensive case study of the optimization process of arguably the most common high-performance computing kernel, matrix-matrix multiplication, using the techniques described in Chapter 2, achieving state-of-the-art performance on the target FPGA device.
- Chapter 5 introduces the Data-Centric Parallel Programming (DaCe) framework as a new way to program spatial architectures, raising the level of abstraction of programs from HLS to a dataflow-oriented, graph-based intermediate representation, where data movement is a first-class citizen. In this paradigm, we can optimize programs and run them on both FPGA vendors without changing the input program.
- Chapter 6 shows how a domain-specific language can be built on top of DaCe by introducing the StencilFlow stack, further raising the level of abstraction to a simple JSON-based specification of stencil programs, which is compiled all the way down to state-of-the-art accelerators on both Xilinx and Intel FPGAs.

Combined, the work in this dissertation sets out to empower the workflow of multiple classes of developers: the **HLS programmer**; the **performance engineer**; the **domain scientist**; and the **compiler engineer**.

Chapter 2

Transforming High-Level Synthesis Codes for High-Performance Computing

The work in this chapter is based on our publication in TPDS [36], and has both influenced and been influenced by our tutorial series "Productive Parallel Programming for FPGA with High-Level Synthesis"[1], given by Prof. Hoefler and myself at numerous occasions, including PPoPP'18, SC'18, SC'19, HiPEAC'20, SC'20, ISC'20, SC'21, and twice at ETH Zurich. The tutorial is now also available on YouTube[2], and includes "live" demos of a series of example codes available on GitHub[3]. Maciej Besta assisted with the visual presentation of the original paper, and Simon Meierhans came up with the algorithm that the N-body example code is based on for a project with us during his bachelor degree.

2.1 Optimization of Hardware Programs

For applications where computational performance is a primary goal, this is typically achieved through careful tuning by specialized performance engineers using well-understood optimizing transformations when targeting CPU [13] and GPU [123] architectures. **For high-level synthesis (HLS) codes, a comparable collection of guidelines and principles for code optimization is yet to be established.** Optimizing codes for hardware is drastically different from optimizing codes for software. In fact, the optimization space is *larger*, as it contains most known software optimizations, in addition to HLS-specific transformations that let programmers manipulate the underlying hardware architecture. To make matters worse, the low clock frequency, lack of cache, and fine-grained configurability, means that *naive* HLS codes typically perform poorly compared to naive software codes, and must be transformed considerably before the advantages of specialized hardware can be exploited. **Thus, the established set of traditional transformations is insufficient, as it does not consider aspects of optimized hardware design, such as pipelining and decentralized fast memory.**

---

[1] `https://spcl.inf.ethz.ch/Teaching/hls-tutorial/`

[2] `https://youtu.be/2UvUP2hxMyI`

[3] `https://github.com/spcl/hls_tutorial_examples`

In this chapter, we survey and define a set of key transformations that optimizing compilers or performance engineers can apply to improve the performance of hardware layouts generated from HLS codes. This set combines transformations extracted from previous work, where they were applied either explicitly or implicitly, with additional techniques that fill in gaps to maximize completeness. We characterize and categorize transformations, allowing performance engineers to easily look up those relevant to improving their HLS code, based on the problems and bottlenecks currently present. The transformations have been verified to apply to both the Intel OpenCL and Xilinx Vitis/Vivado HLS toolflows, but are expected to translate to any pragma-based imperative HLS tool. Most transformations are also relevant to general (non-HLS) hardware design, but the details on how they are applied in a given language will differ.

In addition to identifying and defining the surveyed transformations, we describe and publish a set of end-to-end "hands-on" examples, optimized from naive HLS codes into high performance implementations. This includes a stencil code, matrix multiplication (which we discuss briefly here, but will cover in detail in Chapter 4), and the N-body problem, all available on GitHub. The optimized codes exhibit dramatic cumulative speedups of up to 29,950× relative to their respective naive starting points, showing the crucial necessity of hardware-aware transformations, which are not performed automatically by today's HLS compilers. As FPGAs are currently the most commonly targeted platform by HLS tools in the high-performance computing (HPC) domain, transformations are discussed and evaluated in this context. **Our work provides a set of guidelines and a cheat sheet for optimizing high-performance codes for spatial architectures using HLS languages, guiding both performance engineers and compiler developers to efficiently exploit these devices.**

2.1.1 From Imperative Code to Hardware

Before diving into transformations, it is useful to form an intuition of the major stages of the source-to-hardware stack, to understand how they are influenced by the HLS code:

❶ **High-level synthesis** converts a (typically pragma-assisted) procedural description (C++, OpenCL) to a functionally equivalent behavioral description (Verilog, VHDL). This requires mapping variables and operations to corresponding constructs, then scheduling operations according to their inter-dependencies. The dependency analysis is concerned with creating a hardware mapping such that the throughput requirements are satisfied, which for pipelined sections might require the circuit to consume a new input every one

			Enables Pipelining	**Enables Data Reuse**	**Enables Parallelism**	**Optimizes Memory**	**Resource Usage**	**Impairs Routing**	**Impairs Schedule**	**Impairs Code Complexity**	**Loop-Carried Deps.**	**Increase Data Reuse**	**Increase Parallelism**	**Increase Bandwidth Util.**	**Optimize Pipelines**	**Improve Routing**	**Reduce Resources**
Transformations			**Characteristics**								**Objectives**						
Pipelining	Accumulation interleaving	§2.2.1	👍	–	–	∼	👎	–	👎	∼	👉	–	–	–	–	–	–
	Delay buffering	§2.2.2	👍	👍	(👍)	👍	👎	(👎)	👎	👎	👉	👉	–	–	–	–	–
	Random access buffering	§2.2.3	👍	👍	(👍)	👍	👎	👎	👎	👎	👉	👉	–	👉	–	–	–
	Pipelined loop fusion	§2.2.4	👍	(👍)	–	∼	∼	(👎)	–	👎	–	–	–	–	👉	–	–
	Pipelined loop switching	§2.2.5	👍	(👍)	–	∼	∼	(👎)	–	∼	–	–	–	–	👉	–	👉
	Pipelined loop flattening	§2.2.6	👍	–	–	👍	∼	∼	–	👎	–	–	–	–	👉	–	–
	Inlining	§2.2.7	👍	–	(👍)	–	(👎)	–	–	👍	👉	–	–	–	–	–	–
Scaling	Horizontal unrolling	§2.3.1	–	(👍)	👍	👍	👎	👎	👎	(👎)	–	–	👉	–	–	–	–
	Vertical unrolling	§2.3.2	–	👍!	👍!	–	👎!	👎!	👎	👎	–	👉	👉	–	–	–	–
	Dataflow	§2.3.3	–	–	(👍)	–	(👎)	👍!	–	👍	–	👉	–	–	👉	👉	–
	Tiling	§2.3.4	–	👍	–	∼	👎	∼	👎	👎	👉	👉	–	–	–	👉	👉
Memory	Mem. access extraction	§2.4.1	(👍)	–	–	👍	👎	👍	–	👎	👉	–	–	👉	–	–	–
	Mem. buffering	§2.4.2	–	–	–	👍	👎	–	–	👎	–	–	–	👉	–	–	–
	Mem. striping	§2.4.3	–	–	–	👍	👎	👎	–	👎	–	–	–	👉	–	–	–
	Type demotion	§2.4.4	–	–	–	👍	👍	👍	–	👍	–	–	–	👉	–	–	👉

Table 2.1: Overview of **transformations**, the **characteristics** of their effect on the HLS code and the resulting hardware, and the **objectives** that they can target. The center group of column marks characteristics of each transformation as indicated by the columns, while the right group of columns marks **objectives** that can be targeted by transformations. The latter can be used as a cheat sheet when attempting to address a bottleneck, while the former describes how the transformation will affect the code and resulting architecture.

or more cycles. Coarse-grained control flow is typically implemented as sequential logic, while computations and fine-grained control flow are organized in (predicated) pipelines. ❷ **Hardware synthesis** maps the register-level circuit description to components and wires present on the specific target *architecture*. At this stage and onwards, the procedure is both vendor and architecture specific. ❸ **Place and route** maps the logical circuit description to concrete locations on the target *device*, by performing a lengthy heuristic-based optimization that attempts to minimize the length of the longest wire and the sum of all wire lengths. The longest propagation time between two registers including the logic between them (i.e., the critical path of the circuit), will determine the maximum obtainable frequency. ❹ **Bitstream generation** translates the final circuit description to a binary format used to configure the device.

Most effort invested by an HLS programmer lies in guiding the scheduling process in ❶ to implement deep, efficient pipelines, but ❷ is considered when choosing data types and buffer sizes, and ❸ can ultimately bottleneck applications once the desired parallelism has been achieved, requiring the developer to adapt their code to aid this process.

2.1.2 Key Transformations for High-Level Synthesis

This chapter identifies a set of optimizing transformations that are essential to designing scalable and efficient hardware kernels in HLS. An overview is given in Tab. 2.1. We divide the transformations into three major classes: **pipelining** transformations, that enable or improve the potential for pipelining computations; **scaling** transformations that increase or expose additional parallelism; and **memory** enhancing transformations, which increase memory utilization and efficiency. Each transformation is further classified according to a number of characteristic effects on the HLS source code, and on the resulting hardware architecture (central columns). To serve as a cheat sheet, the table furthermore lists common *objectives* targeted by HLS programmers, and maps them to relevant HLS transformations (rightmost columns). Characteristics and objectives are discussed in detail in relevant transformation sections.

Throughout this chapter, we will show how each transformation is applied manually by a performance engineer by directly modifying the source code, giving examples before and after it is applied. However, many transformations are also amenable to automation in an optimizing compiler.

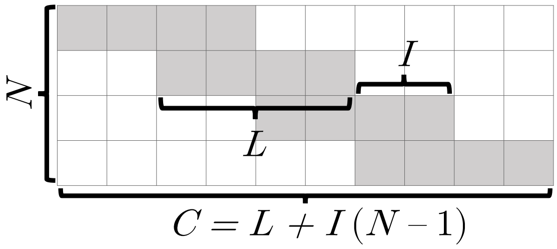


Figure 2.1: Pipeline characteristics.

2.1.3 The Importance of Pipelining

Pipelining is essential to efficient hardware architectures, as expensive instruction decoding and data movement between memory, caches and registers can be avoided, by sending data directly from one computational unit to the next. We attribute two primary characteristics to pipelines:

- **Latency** (L): the number of cycles it takes for an input to propagate through the pipeline and arrive at the exit, i.e., the number of **pipeline stages**.
- **Initiation interval** or **gap** (I): the number of cycles that must pass before a new input can be accepted into the pipeline. A perfect pipeline has $I = 1\,\text{cycle}$, as this is required to keep all pipeline stages busy. Consequently, the initiation interval can often be considered the *inverse throughput* of the pipeline; e.g., $I = 2\,\text{cycles}$ implies that the pipeline stalls every second cycle, reducing the throughput of *all* pipelines stages by a factor of $\frac{1}{2}$.

To quantify the importance of pipelining in HLS, we consider the number of cycles C it takes to execute a pipeline with latency L (both in [cycles]), taking N inputs, with an initiation interval of I [cycles]. Assuming a reliable producer and consumer at either end, we have:

$$C = L + I \cdot (N - 1) \text{ [cycles]}. \tag{2.1}$$

This is shown in Fig. 2.1. The time to execute all N iterations with clock rate f [cycles/s] of this pipeline is thus C/f.

For two pipelines in sequence that both consume and produce N elements, the latency is additive, while the initiation interval is decided by the “slowest” actor:

$$C_0 + C_1 = (L_0 + L_1) + \max(I_0, I_1) \cdot (N - 1)$$

When I_0=I_1 this corresponds to a single, deeper pipeline. For large N, the latencies are negligible, so this deeper pipeline increases pipeline parallelism by adding more computations *without increasing the runtime*; and without introducing additional off-chip memory traffic. **We are thus interested in building deep, perfect pipelines to maximize performance and minimize off-chip data movement.**

2.1.4 Optimization Goals

We organize the remainder of this chapter according to three overarching optimization goals, corresponding to the three categories marked in Tab. 2.1:

- **Enable pipelining** (Sec. 2.2): For compute bound codes, achieve I=1 cycle for all essential compute components, to ensure that all pipelines that materially impact performance run at maximum throughput. For memory bound codes, guarantee that memory is always consumed at line rate.
- **Scaling** (Sec. 2.3): Reduce the total number of iterations N by scaling up the parallelism of the design to consume more elements per cycle, thus cutting the total number of cycles required to execute the program.
- **Memory efficiency** (Sec. 2.4): Saturate pipelines with data from memory to avoid stalls in compute logic. For memory bound codes, maximize bandwidth utilization.

Sec. 2.5 covers the relationship between well-known software optimizations and HLS, and accounts for which of these apply directly to HLS code. Sec. 2.6 shows the effect of transformations on a selection of kernels, and Sec. 2.7 presents related work.

2.2 Pipeline-Enabling Transformations

As a crucial first step for any HLS code, we cover detecting and resolving issues that prevent pipelining of computations. When analyzing a basic block of a program, the HLS tool determines the dependencies between computations, and pipelines operations accordingly to achieve the target initiation interval. There are two classes of problems that hinder pipelining of a given loop:

1. **Loop-carried dependencies** (inter-iteration): an iteration of a pipelined loop depends on a result produced by a previous iteration, which takes multiple cycles to

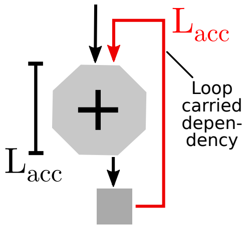


Figure 2.2: Loop-carried dependency.

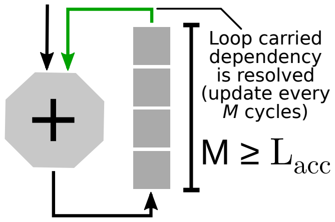


Figure 2.3: Buffered accumulation.

complete (i.e., has multiple internal pipeline stages). If the latency of the operations producing this result is L, the minimum initiation interval of the pipeline will be L. This is a common scenario when accumulating into a single register (see Fig. 2.2), in cases where the accumulation operation takes $L_{\text{acc}}>1$ cycles.

2. **Interface contention** (intra-iteration): a hardware resource with limited ports is accessed multiple times in the same iteration of the loop. This could be a FIFO queue or RAM that only allows a single read and write per cycle, or an interface to external memory, which only supports sending/serving one request per cycle.

For each of the following transformations, we will give examples of programs exhibiting properties that prevent them from being pipelined, and how the transformation can resolve this. All examples use C++ syntax, which allows classes (e.g., "`FIFO`" buffer objects) and templating. We perform pipelining and unrolling using pragma directives, where loop-oriented pragmas always refer to the *following loop/scope*, which is the convention used by Intel/Altera HLS tools (as opposed to applying to *current* scope, which is the convention for Xilinx HLS tools).

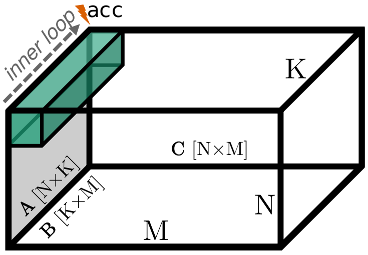
2.2.1 Accumulation Interleaving

For multi-dimensional iteration spaces, *loop-carried dependencies* can often be resolved by reordering and/or interleaving nested loops, keeping state for multiple concurrent accumulations. We distinguish between four approaches to interleaving accumulation, covered below.

2.2.1.1 Full Transposition

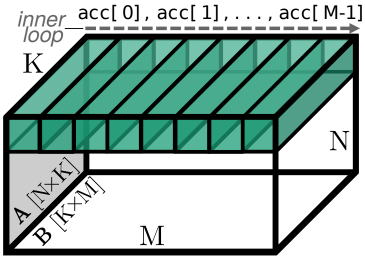
When a loop-carried dependency is encountered in a loop nest, it can be beneficial to reorder the loops, thereby fully transposing the iteration space. This typically also has a

```
for (int n = 0; n < N; ++n)
  for (int m = 0; m < M; ++m) {
    double acc = C[n][m];
    #pragma PIPELINE
    for (int k = 0; k < K; ++k)
      acc += A[n][k] * B[k][m];
    C[n][m] = acc; }
```



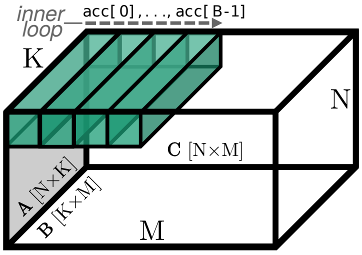
(a) Naive implementation of general matrix multiplication $C{=}AB{+}C$.

```
for (int n = 0; n < N; ++n) {
  double acc[M]; // Uninitialized
  for (int k = 0; k < K; ++k)
    double a = A[n][k]; // Only read once
    #pragma PIPELINE
    for (int m = 0; m < M; ++m) {
      double prev = (k == 0) ? C[n][m]
                             : acc[m];
      acc[m] = prev + a * B[k][m]; }
  for (int m = 0; m < M; ++m) // Write
    C[n][m] = acc[m]; }       // out
```



(b) Transposed iteration space, same location written every M cycles.

```
for (int n = 0; n < N; ++n)
  for (int m = 0; m < M/T; ++m) {
    double acc[T]; // Tiles of size T
    for (int k = 0; k < K; ++k)
      double a = A[n][k]; // M/T reads
      #pragma PIPELINE
      for (int t = 0; t < T; ++t) {
        double prev = (k == 0) ?
            C[n][m*T+t] : acc;
         acc = prev + a * B[k][m*T+t]; }
    for (int t = 0; t < T; ++t) // Write
      C[n][m*T+t] = acc; }   // out
```



(c) Tiled iteration space, same location written every T cycles.

Listing 1: Interleave accumulations to remove loop-carried dependency.

significant impact on the program's memory access pattern, which can benefit/impair the program beyond resolving a loop-carried dependency.

Consider the matrix multiplication code in Lst. 1a, computing $\boldsymbol{C} = \boldsymbol{A} \cdot \boldsymbol{B} + \boldsymbol{C}$, with matrix dimensions N, K, and M. The inner loop $k \in K$ accumulates into a temporary register, which is written back to $\boldsymbol{C}$ at the end of each iteration $m \in M$. The multiplication of elements of $\boldsymbol{A}$ and $\boldsymbol{B}$ can be pipelined, but the addition on line 6 requires the result of the addition in the previous iteration of the loop. This is a loop-carried dependency, and results in an initiation interval of L_+, where L_+ is the latency of a 64-bit floating point addition (for integers $L_{+,\texttt{int}}$=1 cycle, and the loop can be pipelined without further modifications). To avoid this, we can transpose the iteration space, swapping the K-loop with the M-loop, with the following consequences:

- Rather than a single register, we now implement an accumulation buffer of depth M and width 1 (line 2).
- The loop-carried dependency is resolved: each location is only updated every M cycles (with $M \geq L_{\text{acc}}$ in Fig. 2.3).
- $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$ are all read in a contiguous fashion, achieving perfect spatial locality (we assume row-major memory layout. For column-major we would interchange the K-loop and N-loop).
- Each element of $\boldsymbol{A}$ is read exactly once.

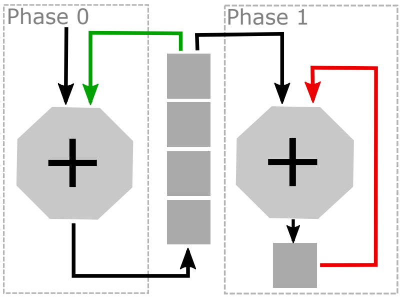
The modified code is shown in Lst. 1b. We leave the accumulation buffer defined on line 2 uninitialized, and implicitly reset it on line 8, avoiding M extra cycles to reset (this is a form of *pipelined loop fusion*, covered in Sec. 2.2.4).

2.2.1.2 Tiled Accumulation Interleaving

For accumulations done in a nested loop, it can be sufficient to interleave across a tile of an outer loop to resolve a loop-carried dependency, using a limited size buffer to store intermediate results. This tile only needs to be of size $\geq L_{\text{acc}}$, where L_{acc} is the latency of the accumulation operation.

This is shown in Lst. 1c, for the transposed matrix multiplication example from Lst. 1b, where the accumulation array has been reduced to tiles of size T (which should be $\geq L_{\text{acc}}$, see Fig. 2.3), by strip-mining the loop over M by a factor of T.

```
double Acc(double arr[], int N) {
  double t[16];
  #pragma PIPELINE
  for (int i = 0; i < N; ++i) { // P0
    auto prev = (i < 16) ? 0 : t[i%16];
    t[i%16] = prev + arr[i]; }
  double res = 0;
  for (int i = 0; i < 16; ++i)  // P1
    res += t[i]; // Not pipelined
  return res; }
```



Listing 2: Two stages required for single loop accumulation.

2.2.1.3 Single-Loop Accumulation Interleaving

If no outer loop is present, we have to perform the accumulation in two separate stages, at the cost of extra resources. For the first stage, we perform a transformation similar to the nested accumulation interleaving, but strip-mine the inner (and only) loop into blocks of size $K \geq L_{\text{acc}}$, accumulating partial results into a buffer of size K. Once all incoming values have been accumulated into the partial result buffers, the second phase collapses the partial results into the final output. This is shown in Lst. 2 for K=16.

Optionally, the two stages can be implemented to run in a coarse-grained pipelined fashion, such that the first stage begins computing new partial results while the second stage is collapsing the previous results (by exploiting dataflow between modules, see Sec. 2.3.3).

2.2.1.4 Batched Accumulation Interleaving

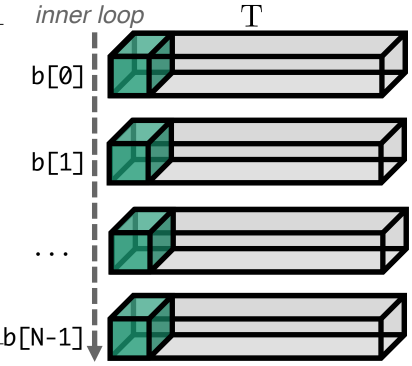
For algorithms with loop-carried dependencies that cannot be solved by either method above (e.g., due to a non-commutative accumulation operator), we can still pipeline the design by processing *batches* of inputs, introducing an additional loop nested in the accumulation loop. This procedure is similar to Sec. 2.2.1.2, but only applies to programs where it is relevant to compute the accumulation for multiple data streams, and requires altering the interface and data movement of the program to interleave inputs in batches.

The code in Lst. 3a shows an iterative solver code with an inherent loop-carried dependency on `state`, with a minimum initiation interval corresponding to the latency L_{Step} of the (inlined) function `Step`. There are no loops to interchange, and we cannot change the order of loop iterations. While there we cannot improve the latency of producing a single result without touching the `Step` function, we can improve the overall throughput by a factor of L_{Step} by pipelining across $N \geq L_{\text{Step}}$ different inputs (e.g., overlap solving for different starting conditions). We effectively inject another loop over inputs, then per-

```
Vec<double> IterSolver(Vec<double> state, int T) {
  #pragma PIPELINE // Will fail to pipeline with I=1
  for (int t = 0; t < T; ++t)
    state = Step(state);
  return state; }
```

(a) Solver executed for T steps with a loop-carried dependency on `state`.

```
template <int N>
void MultiSolver(Vec<double> *in,
                 Vec<double> *out, int T) {
  Vec<double> b[N]; // Partial results
  for (int t = 0; t < T; ++t)
    #pragma PIPELINE
    for (int i = 0; i < N; ++i) {
      auto read = (t == 0) ? in[i] : b[i];
      auto next = Step(read);
      if (t < T-1) b[i] = next;
      else out[i] = next; }} // Write out
```



(b) Pipeline across $N \geq L_{\text{step}}$ inputs to achieve $I=1$ cycle.

Listing 3: Pipeline across multiple inputs to avoid loop-carried dependency.

form transposition or tiled accumulation interleaving with this loop. The result of this transformation is shown in Lst. 3b, for a variable number of interleaved inputs `N`.

2.2.2 Delay Buffering

When iterating over regular domains in a pipelined fashion, it is often sufficient to express buffering using delay buffers, expressed either with cyclically indexed arrays, or with constant offset delay buffers, also known from the Intel ecosystem as *shift registers*. These buffers are only accessed in a FIFO manner, with the additional constraint that elements are only be popped once they have *fully* traversed the depth of the buffer (or when they pass compile-time fixed access points, called “taps”, in Intel OpenCL). Despite the “shift register” name, these buffers do not need to be implemented in registers, and are frequently implemented in on-chip RAM when large capacity is needed, where values are not *physically* shifted.

A common set of applications that adhere to the delay buffer pattern are stencil applications such as partial differential equation solvers [133, 139, 51], image processing pipelines [70, 121], and convolutions in deep neural networks [17, 92, 31, 140, 21], all

```
float north_buffer[M];  // Line
float center_buffer[M]; // buffers
float west, center; // Registers
for (int i = 0; i < N; ++i) {
  #pragma PIPELINE
  for (int j = 0; j < M; ++j) {
    auto south = memory[i][j]; // Single memory read
    auto north = north_buffer[j];         // Read line buffers
    auto east = center_buffer[(j + 1)%M]; // (with wrap around)
    if (i > 1 && j > 0 && j < M - 1) // Assume padding of 1
      result[i - 1][j] = 0.25*(north + west + south + east);
    north_buffer[j] = center; // Update both
    center_buffer[j] = south; // line buffers
    west = center; // Propagate
    center = east; // registers
  }
}
```

(a) Delay buffering using cyclically indexed line buffers.

```
float sr[2*M + 1]; // Shift register buffer
for (int i = 0; i < N; ++i) {
  #pragma PIPELINE
  for (int j = 0; j < M; ++j) {
    #pragma UNROLL
    for (int k = 0; k < 2*M; ++k)
      sr[k] = sr[k + 1]; // Shift the array left
    sr[2*M] = memory[i][j]; // Append to the front
    if (i > 1 && j > 0 && j < M - 1) // Initialize/drain
      result[i-1][j] = 0.25*(sr[0] + sr[M-1] + sr[M+1] + sr[2*M]);
  }
}
```

(b) Delay buffering using an Intel-style shift register.

Listing 4: Two ways of implementing delay buffering on an $N \times M$ grid.

of which are typically traversed using a *sliding window* buffer, implemented in terms of multiple delay buffers (or, in Intel terminology, a shift register with multiple *taps*). These applications have been shown to be a good fit to spatial computing architectures [54, 166, 146, 79, 111, 110, 52], as delay buffering is cheap to implement in hardware, either as shift registers in general purpose logic, or in RAM blocks.

Lst. 4 shows two ways of applying delay buffering to a stencil code, namely a 4-point stencil in 2D, which updates each point on a 2D grid to the average of its north, west, east, and south neighbors. To achieve perfect data reuse, we buffer every element read in sequential order from memory until it has been used for the last time – after two rows, when the same value has been used as all four neighbors.

In Lst. 4a we use cyclically indexed line buffers to implement the delay buffering pattern, instantiated as arrays on lines 1-2. We only read the south element from memory each iteration (line 7), which we store in the center line buffer (line 13). This element is then reused after M cycles (i.e., “delayed” for M cycles), when it is used as the east value (line 9), propagated to the north buffer (line 12), shifted in registers for two cycles until it is used as the west value (lines 14–15), and reused for the last time after M cycles on line 8. The resulting circuit is illustrated in Fig. 2.4.

Lst. 4b demonstrates the shift register pattern used to express the stencil buffering scheme, which is supported by the Intel OpenCL toolflow. Rather than creating each individual delay buffer required to propagate values, a single array is used, which is “shifted” every cycle using unrolling (lines 6-7). The computation accesses elements of this array using *constant indices only* (line 10), relying on the tool to infer the partitioning into individual buffers (akin to loop idiom recognition [13]) that we did explicitly in Lst. 4a. The implicit nature of this pattern requires the tool to specifically support it. For more detail on buffering stencil codes we refer to Chapter 6.

Opportunities for delay buffering often arise naturally in pipelined programs. If we consider the transposed matrix multiplication code in Lst. 1b, we notice that the read from `acc` on line 8 and the write on line 9 are both sequential, and cyclical with a period of M cycles. We could therefore also use the shift register abstraction for this array. The same is true for the accumulation code in Lst. 3b.

2.2.3 Random Access Buffering

When a program unavoidably needs to perform random accesses, we can buffer data in on-chip memory and perform random access to this fast memory instead of to slow off-chip

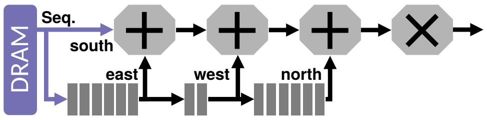
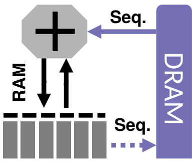


Figure 2.4: A delay buffer for a 4-point stencil with three *taps*.

```
1 unsigned hist[256] = {0}; // Array of bins
2 #pragma PIPELINE // Will have II=2
3 for (int i = 0; i < N; ++i) {
4   int bin = CalculateBin(memory[i]);
5   hist[bin] += 1; // Single cycle access
6 } // ...write result out to memory...
```



Listing 5: Random access to on-chip histogram buffer.

memory. A random access buffer implemented with a general purpose replacement strategy will emulate a CPU-style cache; but to benefit from targeting a spatial system, it is usually more desirable to *specialize* the buffering strategy to the target application [26, 158]. This can enable off-chip memory accesses to be made contiguous by loading and storing data in stages (i.e., tiles), then exclusively performing random accesses to fast on-chip memory.

Lst. 5 outlines a histogram implementation that uses an on-chip buffer (line 1) to perform fast random accesses reads and writes (line 5) to the bins computed from incoming data, illustrated in Fig. 5. Note that the random access results in a loop-carried dependency on `histogram`, as there is a potential for subsequent iterations to read and write the same bin. This can be solved with one of the interleaving techniques described in Sec. 2.2.1, by maintaining multiple partial result buffers.

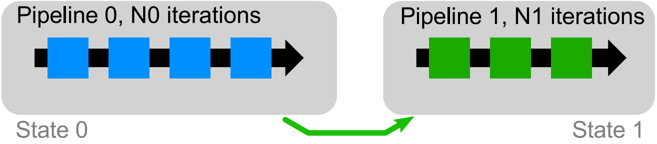
2.2.4 Pipelined Loop Fusion

When two pipelined loops appear sequentially, we can fuse them into a single pipeline, while using loop guards to enforce any dependencies that might exist between them. This can result in a significant reduction in runtime, at little to no resource overhead. This transformation is closely related to loop fusion [86] from traditional software optimization.

For two consecutive loops with latencies/bounds/initiation intervals $\{L_0, N_0, I_0\}$ and $\{L_1, N_1, I_1\}$ (Lst. 6a), respectively, the total runtime according to Eq. 2.1 is $(L_0 + I_0(N_0{-}1)){+}(L_1{+}I_1(N_1{-}1))$. Depending on which condition(s) are met, we can distinguish between three levels of pipelined loop fusion, with increasing performance benefits:

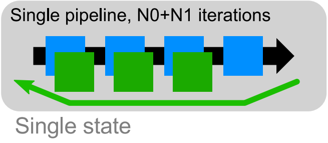
1. $I{=}I_0{=}I_1$ (true in most cases): Loops can be fused by summing the loop bounds, using

```
// Pipelined loops executed sequentially
#pragma PIPELINE
for (int i = 0; i < N0; ++i) Foo(i, /*...*/);
#pragma PIPELINE
for (int i = 0; i < N1; ++i) Bar(i, /*...*/);
```



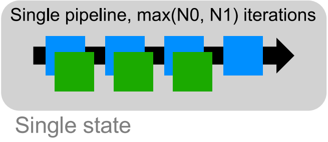
(a) $(L_0 + I_0(N_0{-}1)) + (L_1 + I_1(N_1{-}1))$ cycles.

```
#pragma PIPELINE
for (int i = 0; i < N0+N1; ++i) {
  if (i < N0) Foo(i,      /*...*/);
         else Bar(i - N0, /*...*/); }
```



(b) $L_2 + I(N_0 + N_1{-}1)$ cycles.

```
#pragma PIPELINE
for (int i = 0; i < max(N0, N1); ++i) {
  if (i < N0) Foo(i, /*...*/);   // Omit ifs
  if (i < N1) Bar(i, /*...*/); } // for N0==N1
```



(c) $L_3 + I \cdot (\max(N_0, N_1){-}1)$ cycles.

Listing 6: Two subsequent pipelined loops are fused sequentially (Lst. 6b) or concurrently (Lst. 6c) to reduce the total number of cycles required to execute them.

loop guards to sequentialize them within the same pipeline (Lst. 6b).

2. Condition 1 is met, **and** *only fine-grained or no dependencies* exist between the two loops: Loops can be fused by iterating to the maximum loop bound, and loop guards are placed as necessary to predicate each section (Lst. 6c).
3. Conditions 1 and 2 are met, **and** $N{=}N_0{=}N_1$ (same loop bounds): Loops bodies can be trivially fused (Lst. 6c, but with no loop guards necessary).

An alternative way of performing pipeline fusion is to instantiate each stage as a separate processing element, and stream fine-grained dependencies between them (Sec. 2.3.3).

2.2.5 Pipelined Loop Switching

The benefits of pipelined loop fusion can be extended to coarse-grained control flow by using *loop switching* (as opposed to loop *un*switching, which is a common transformation on load/store architectures [13]). Whereas instruction-based architectures attempt to only execute one branch of a conditional jump (via branch prediction on out-of-order processors), a conditional in a pipelined scenario will result in *both* branches being instantiated in hardware, regardless of whether/how often it is executed. The transformation of coarse-grained control flow into fine-grained control flow is implemented by the HLS tool by introducing *predication* to the pipeline, at no significant runtime penalty.

Lst. 7 shows a simple example of how the transformation fuses two pipelined loops in different branches into a single loop switching pipeline. The transformation applies to *any* pipelined code in either branch, following the principles described for pipelined loop fusion (§2.2.4 and Lst. 6).

The implications of pipelined loop switching are more subtle than the pure fusion examples in Lst. 6, as the total number of loop iterations is not affected (assuming the fused loop bound is set according to the condition, see line 1 in Lst. 7b). There *can* be a (tool-dependent) benefit from saving overhead logic by only implementing the orchestration and interfaces of a single pipeline, at the (typically minor) cost of the corresponding predication logic. More importantly, eliminating the coarse-grained control can enable other transformations that significantly benefit performance, such as fusion [§2.2.4] with adjacent pipelined loops, flattening nested loops [§2.2.6], and on-chip dataflow [§2.3.3].

```
if (condition) {
  #pragma HLS PIPELINE
  for (int i = 0; i < N0; ++i)
    y[i] = Foo(x[i]);
} else {
  #pragma HLS PIPELINE
  for (int i = 0; i < N1; ++i)
    y[i] = Bar(x[i]);
}
```

(a) Coarse-grained control flow.

```
auto N = condition ? N0 : N1;
#pragma HLS PIPELINE
for (int i = 0; i < N; ++i) {
  if (condition) {
    y[i] = Foo(x[i]);
  } else {
    y[i] = Bar(x[i]);
  }
}
```

(b) Control flow absorbed into pipeline.

Listing 7: Pipelined loop switching absorbs coarse-grained control flow.

2.2.6 Pipelined Loop Flattening/Coalescing

To minimize the number of cycles spent in filling/draining pipelines (where the circuit is not streaming at full throughput), we can flatten nested loops to move the fill/drain phases to the outermost loop, fusing/absorbing code that is not in the innermost loop if necessary.

Lst. 8a shows a code with two nested loops, and gives the total number of cycles required to execute the program. The latency of the drain phase of the inner loop and the latency of `Bar` outside the inner loop must be *paid at every iteration* of the outer loop. If $N_0 \gg L_0$, the cycle count becomes just $L_1 + N_0 N_1$, but for applications where N_0 is comparable to L_0, draining the inner pipeline can significantly impact the runtime (even if N_1 is large). By transforming the code such that all loops are *perfectly nested* (see Lst. 8b), the HLS tool can effectively *coalesce* the loops into a single pipeline, where next iteration of the *outer* loop can be executed immediately after the previous finishes.

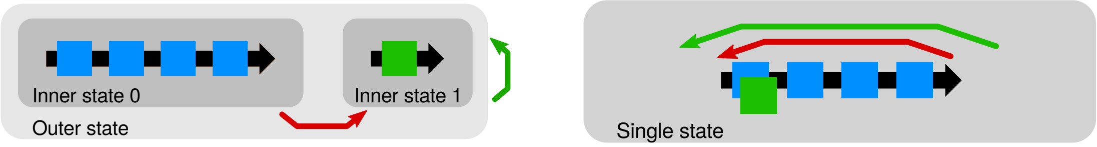
To perform the transformation in Lst. 8, we had to absorb `Bar` into the inner loop, adding a loop guard (line 5 in Lst. 8b), analogous to pipelined loop fusion (§2.2.4), where the second pipelined "loop" consists of a single iteration. This contrasts the loop peeling transformation, which is used by CPU compilers to regularize loops to avoid branch mispredictions and increasing amenability to vectorization. While loop peeling can also be beneficial in hardware, e.g., to avoid deep conditional logic in a pipeline, small inner loops can see a significant performance improvement by eliminating the draining phase.

```
for (int i = 0; i < N1; ++i) {
  #pragma PIPELINE
  for (int j = 0; j < N0; ++i) {
    Foo(i, j);
  }
  Bar(i);
}
```

```
for (int i = 0; i < N1; ++i) {
  #pragma PIPELINE
  for (int j = 0; j < N0; ++i) {
    Foo(i, j);
    if (j == N0 - 1) Bar(i);
  }
}
```

(a) $L_1 + N_1 \cdot (L_0 + N_0{-}1)$ cycles.

(b) $L_2 + N_0 N_1{-}1$ cycles.



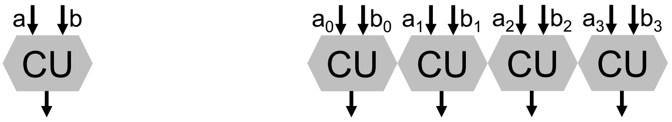
Listing 8: Before and after coalescing loop nest to avoid inner pipeline drains.

2.2.7 Inlining

In order to successfully pipeline a scope, all function calls within the code section must be pipelineable. This typically requires “inlining” functions into each call site, creating dedicated hardware for each invocation, resulting in additional resources consumed for every additional callsite after the first. This replication is done automatically by HLS compilers on demand, but an additional `inline` pragma can be specified to directly “paste” the function body into the callsite during preprocessing, removing the function boundary during optimization and scheduling.

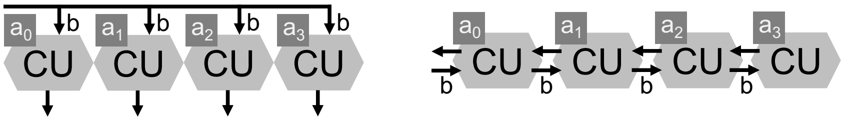
2.3 Scalability Transformations

Parallelism in HLS revolves around *unrolling* loop iterations that would otherwise be executed in a sequential or pipelined manner. In Sec. 2.2.1 we used strip-mining and reordering to avoid loop-carried dependencies by changing the *schedule* of computations in the pipelined loop nest. In this section, we similarly strip-mine and reorder loops, but with additional unrolling of the strip-mined chunks. Pipelined loops constitute the *iteration space*; the size of which determines the number of cycles it takes to execute the program. Unrolled loops, in a pipelined program, correspond to the degree of *parallelism* in the architecture, as every expression in an unrolled statement is required to be executed concurrently in hardware. Parallelizing a code thus means turning sequential/pipelined loops fully or partially into parallel/unrolled loops. This corresponds to cutting the number



(a) Before.

(b) Horizontal unrolling.



(c) Vertical unrolling.

(d) Streaming dataflow.

Figure 2.5: Horizontal unrolling, vertical unrolling, and dataflow, as means to increase parallelism. Rectangles represent buffer space, such as registers or on-chip RAM. **Horizontal:** four independent inputs processed in parallel. **Vertical:** one input is combined with multiple buffered values. **Dataflow:** similar to vertical, but input or partial results are streamed through a pipeline rather than broadcast.

```
for (int i = 0; i < N / W; ++i)
  #pragma UNROLL // Fully unroll inner
  for (int w = 0; w < W; ++w) // loop
    C[i*W + w] = A[i*W + w]*B[i*W + w];
```

(a) Using strip-mining.

```
// Unroll outer loop by W
#pragma UNROLL W
for (int i = 0; i < N; ++i)
  C[i] = A[i] * B[i];
```

(b) Using partial unrolling.

Listing 9: Two variants of vectorization by factor W using loop unrolling.

of sequential iterations, as the number of cycles taken to execute the program are effectively reduced by the inverse of the unrolling factor.

2.3.1 Horizontal Unrolling (Vectorization)

We implement vectorization-style parallelism with HLS by "horizontally" unrolling loops in pipelined sections, or by introducing vector types, dividing the sequential iteration space by the degree of parallelism accordingly. This is the most straightforward way of adding parallelism, as it can often be applied directly to an inner loop without further reordering or drastic changes to the nested loop structure. Vectorization is more powerful in HLS than SIMD operations on load/store architectures, as the unrolled compute units are not required to be homogeneous, and the number of units are not constrained to fixed sizes. Horizontal unrolling increases bandwidth utilization by explicitly exploiting spatial locality, allowing more efficient accesses to off-chip memory such as DRAM.

Lst. 9 shows two functionally equivalent ways of vectorizing a loop over N elements by a horizontal unrolling factor of W. Lst. 9a strip-mines a loop into chunks of W and unrolls the inner loop fully, while Lst. 9b uses partial unrolling by specifying an unroll factor in the pragma. As a third option, explicit vector types can be used, such as those built into OpenCL (e.g., `float4` or `int16`), or custom vector classes (such as those provided by hlslib, see Chapter 3). These provide less flexibility, but are more concise and are sufficient for many scientific applications.

In practice, the unrolling factor W [operand/cycle] is constrained by the bandwidth B [Byte/s] available to the compute logic (e.g., from off-chip memory), according to

$$W_{\max} = \frac{B}{fS},$$

where f [cycle/s] is the clock frequency of the unrolled logic, and S [Byte/operand] is the operand size in bytes. To exploit all available bandwidth in a memory bound application using vectorization, we must set $W \geq \left\lceil \frac{B}{fS} \right\rceil$, while compute bound computations should set $W \leq \left\lfloor \frac{B}{fs} \right\rfloor$ to avoid spending resources on compute units that cannot be fully saturated by the available bandwidth (assuming another source of parallelism is available, see the following). Horizontal unrolling is usually not sufficient to achieve high logic utilization on large chips, where the available memory bandwidth is low compared to the available amount of compute logic. Furthermore, because the energy cost of I/O is orders of magnitude higher than moving data on the chip, it is desirable to exploit on-chip memory and pipeline parallelism instead (this follows in Sec. 2.3.2 and 2.3.3).

2.3.2 Vertical Unrolling

We can achieve scalable parallelism in HLS without relying on external memory bandwidth by exploiting data reuse, distributing input elements to multiple computational units replicated "vertically" through unrolling [127, 166, 39]. *This is the most potent source of parallelism on hardware architectures*, as it can conceptually scale indefinitely with available silicon when enough reuse is possible. Viewed from the paradigm of cached architectures, the opportunity for this transformation arises from temporal locality in loops. Vertical unrolling draws on bandwidth from on-chip fast memory by storing more elements temporally, combining them with new data streamed in from external memory to increase parallelism, allowing more computational units to run in parallel at the expense of buffer space. In comparison, horizontal unrolling requires us to widen the data path that passes through the processing elements (compare Fig. 2.5b and 2.5c).

```
 1 for (int n = 0; n < N / P; ++n) { // Divided by unrolling factor P
 2   for (int m = 0; m < M / T; ++m) { // Tiling
 3     double acc[T][P]; // Is now 2D
 4     // ...initialize acc from C...
 5     for (int k = 0; k < K; ++k) {
 6       double a_buffer[P]; // Buffer multiple elements to combine with
 7       #pragma PIPELINE    // incoming values of B in parallel
 8       for (int p = 0; p < P; ++p)
 9         a_buffer[p] = A[n*P + p][k];
10       #pragma PIPELINE
11       for (int t = 0; t < T; ++t) // Stream tile of B
12         #pragma UNROLL
13         for (int p = 0; p < P; ++p) // P-fold vertical unrolling
14           acc[p] += a_buffer[p] * B[k][m*T+t];
15     }
16     /* ...write back 2D tile of C... */
17   }
18 }
```

Listing 10: P-fold vertical unrolling of matrix multiplication.

When attempting to parallelize a new algorithm, identifying a source of temporal parallelism to feed vertical unrolling is essential to determine whether the design will scale. Programmers should consider this carefully before designing the hardware architecture. From a reference software code, the programmer can identify scenarios where reuse occurs, then extract and *explicitly express* the temporal access pattern in hardware, using a delay buffering [§2.2.2] or random-access [§2.2.3] buffering scheme. Then, if additional reuse is possible, vertically unroll the circuit to scale up performance.

As an example, we return to the matrix multiplication code from Lst. 1c. In Sec. 2.2.1.2, we saw that strip-mining and reordering loops allowed us to move reads from matrix $\boldsymbol{A}$ out of the inner loop, re-using the loaded value across T different entries of matrix $\boldsymbol{B}$ streamed in while keeping the element of $\boldsymbol{A}$ in a register. Since every loaded value of $\boldsymbol{B}$ *eventually* needs to be combined with all N rows of $\boldsymbol{A}$, we realize that we can perform more computations in parallel by keeping *multiple* values of $\boldsymbol{A}$ in local registers. The result of this transformation is shown in Lst. 10. By buffering P elements (where P was 1 in Lst. 1c) of $\boldsymbol{A}$ prior to streaming in the tile of $\boldsymbol{B}$-matrix (lines 8-9), we can divide the outer loop over rows by a factor of P, using unrolling to multiply parallelism (as well as buffer space required for the partial sums) by a factor of P (lines 12-14).

2.3.3 Dataflow

```
void PE(FIFO<float> &in, FIFO<float> &out, int T) {
  // ..initialization...
  for (int t = 0; t < T / P; ++t) { // Divide timesteps T by factor P
    #pragma PIPELINE
    for (/* loops over spatial dimensions */) {
      auto south = in.Pop(); // Value for t-1 from previous PE
      // ...load values from delay buffers...
      auto next = 0.25*(north + west + east + south);
      out.Push(next); // Value for t sent to PE computing t+1
    }
  }
}
```

(a) Processing element for a single timestep. Will be replicated P times.

```
#pragma DATAFLOW // Schedule nested functions as parallel modules
void SystolicStencil(const float in[], float out[], int T) {
  FIFO<float> pipes[P + 1]; // Assume P is given at compile time
  ReadMemory(in, pipes[0]); // Head
  #pragma UNROLL // Replicate PEs
  for (int p = 0; p < P; ++p) {
    PE(pipe[p], pipe[p + 1], T); // Forms a chain
  }
  WriteMemory(pipes[P], out); // Tail
}
```

(b) Instantiate and connect P consecutive and parallel PEs.

Listing 11: Dataflow between replicated PEs to compute P timesteps in parallel.

For complex codes it is common to partition functionality into multiple modules, or *processing elements* (PEs), streaming data between them through explicit interfaces. In contrast to conventional pipelining, PEs arranged in a dataflow architecture are scheduled separately when synthesized by the HLS tool. There are multiple benefits to this:

- *Different functionality runs at different schedules.* For example, *issuing* memory requests, *servicing* memory requests, and *receiving* requested memory can all require different pipelines, state machines, and even clock rates.
- Smaller components are more *modular*, making them easier to reuse, debug and verify.
- The effort required by the HLS tool to schedule code sections increases dramatically with the number of operations that need to be considered for the dependency and pipelining analysis. Scheduling logic in smaller chunks is thus beneficial for compilation time.
- Large *fan-out/fan-in* is challenging to route on real hardware, (i.e., 1-to-N or N-to-1 connections for large N). This is mitigated by partitioning components into smaller parts and adding more pipeline stages.
- The fan-in and fan-out of control signals (i.e., stall, reset) *within* each module is reduced, reducing the risk of these signals constraining the maximum achievable frequency.

To move data between PEs, communication channels with a handshake mechanism are used. These channels double as synchronization points, as they imply a consensus on the program state. In practice, channels are always FIFO interfaces, and support standard queue operations `Push`, `Pop`, and sometimes `Empty`, `Full`, and `Size` operations. They occupy the same register or block memory resources as other buffers (Sec. 2.2.2/Sec. 2.2.3).

The mapping from source code to PEs differs between HLS tools, but is manifested when functions are connected using channels. In the following example, we will use the syntax from Xilinx Vitis HLS to instantiate PEs, where each non-inlined function correspond to a PE, and these are connected by channels that are passed as arguments to the functions from a top-level entry function. Note that this **functionally diverges from C++ semantics** without additional abstraction (we will address this in Sec. 3.2.3), as each function in the dataflow scope is executed in parallel in hardware, rather than in the sequence specified in the imperative code. In Intel OpenCL, dataflow semantics are instead expressed with multiple `kernel` functions each defining a PE, which are connected by global channel objects prefixed with the `channel` keyword.

To see how streaming can be an important tool to express scalable hardware, we apply it in conjunction with vertical unrolling (Sec. 2.3.2) to implement an iterative version of the stencil example from Lst. 4. Unlike the matrix multiplication code, the stencil code has no scalable source of parallelism in the spatial dimension. Instead, we can achieve reuse by unrolling the outer time-loop to treat P consecutive timesteps in a pipeline parallel fashion, each computed by a distinct PE, connected in a chain via channels [54, 124, 166]. We replace the memory interfaces to the PE with channels, such that the memory read and write become `Pop` and `Push` operations, respectively. The resulting code is shown in Lst. 11a. We then vertically unroll to generate P instances of the PE (shown in Lst. 11b), effectively increasing the throughput of the kernel by a factor of P, and consequently reducing the runtime by dividing the outermost loop bound by the unroll factor P (line 3 in Lst. 11a). Such architectures are sometimes referred to as *systolic arrays* [89, 104].

For architectures/HLS tools where large fan-out is an issue for compilation or routing, an already replicated design can be transformed to a dataflow architecture. For example, in the matrix multiplication example in Lst. 10, we can move the P-fold unroll out of the inner loop, and replicate the entire PE instead, replacing reads and writes with channel accesses (for more details, see the case study of optimizing matrix multiplication Chapter 4). $\boldsymbol{B}$ is then streamed into the first PE, and passed downstream every cycle. $\boldsymbol{A}$ and $\boldsymbol{C}$ should no longer be accessed by every PE, but rather be handed downstream similar to $\boldsymbol{B}$, requiring a careful implementation of the start and drain phases, where the behavior of each PE will vary slightly according to its depth in the sequence.

2.3.4 Tiling

Loop tiling in HLS is commonly used to partition large problem sizes into manageable chunks that fit into fast on-chip memory, in an already pipelined program [166]. Rather than making the program faster, this lets the already fast architecture support arbitrarily large problem sizes. This is in contrast to loop tiling on CPU and GPU, where tiling is used to increase performance. Common to both paradigms is that they fundamentally aim to meet fast memory constraints. As with horizontal and vertical unrolling, tiling relies on strip-mining loops to alter the iteration space.

Tiling was already shown in Sec. 2.2.1.2, when the accumulation buffer in Lst. 1b was reduced to a tile buffer in Lst. 1c, such that the required buffer space used for partial results became a constant, rather than being dependent on the input size. This transformation is also relevant to the stencil codes in Lst. 4, where it can be used to restrict the size of the line buffers or shift register, so they are no longer proportional to the problem size.

2.4 Memory Access Transformations

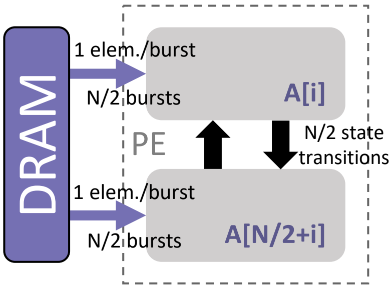
When an HLS design has been pipelined, scheduled, and unrolled as desired, the memory access pattern has been established. In the following, we describe transformations that optimize the efficiency of off-chip memory accesses in the HLS code. For memory bound codes in particular, this is critical for performance after the design has been pipelined.

2.4.1 Memory Access Extraction

By extracting accesses to external memory from the computational logic, we enable compute and memory accesses to be pipelined and optimized separately. Accessing the same interface multiple times within the same pipelined section is a common cause for poor memory bandwidth utilization and increased initiation interval due to interface contention, since the interface can only service a single request per cycle. In the Intel OpenCL flow, memory extraction is done automatically by the tool, but since this process must be conservative due to limited information, it is often still beneficial to do the extraction explicitly in the code [84]. In many cases, such as for independent reads, this is not an inherent memory bandwidth or latency constraint, but arises from the tool scheduling iterations according to program order. This can be relaxed when allowed by inter-iteration dependencies (which can in many cases be determined automatically, e.g., using polyhedral analysis [60]).

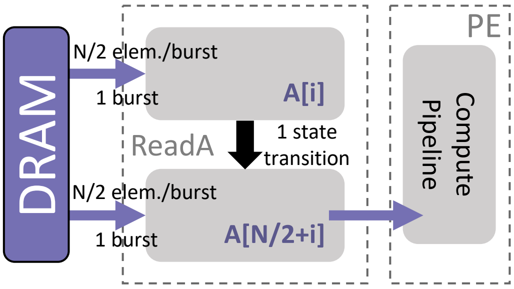
In Lst. 12a, the same memory (i.e., hardware memory interface) is accessed twice in the inner loop. In the worst case, the program will issue two 4 Byte memory requests every iteration, resulting in poor memory performance, and preventing pipelining of the loop. In software, this problem is typically mitigated by caches, always fetching at least one cache line. If we instead read the two sections of A sequentially (or in larger chunks), the HLS tool can infer two so-called “burst” accesses (consecutive sequential accesses) to A of length $N/2$, shown in Lst. 12c. Since the schedules of memory and computational modules are independent, `ReadA` can run ahead of `PE`, ensuring that memory is always read at the maximum bandwidth of the interface (Sec. 2.4.2 and Sec. 2.4.3 will cover how to increase this bandwidth). From the point of view of the computational PE, both A_0 and A_1 are read in parallel, as shown on line 5 in Lst. 12b, hiding initialization time and inconsistent memory producers in the synchronization implied by the data streams.

```
void PE(const int A[N], int B[N/2]) {
  #pragma PIPELINE // Achieves I=2
  for (int i = 0; i < N/2; ++i) {
    // Issues N/2 memory requests of size 1
    B[i] = A[i] + A[N/2 + i];
  }
}
```



(a) Multiple accesses to A cause inefficient memory accesses.

```
void PE(FIFO<int> &A0, FIFO<int> &A1,
        int B[N/2]) {
  #pragma PIPELINE // Achieves I=1
  for (int i = 0; i < N/2; ++i) {
    B[i] = A0.Pop() + A1.Pop());
  }
}
```



(b) Move memory accesses out of computational code.

```
void ReadA(const int A[N], FIFO<int> &A0, FIFO<int> &A1) {
  int buffer[N/2];
  #pragma PIPELINE
  for (int i = 0; i < N/2; ++i)
    buffer[i] = A[i]; // Issues 1 memory request of size N/2
  #pragma PIPELINE
  for (int i = 0; i < N/2; ++i) {
    A0.Push(buffer[i]); // Sends to PE
    A1.Push(A[N/2 + i]); // Issues 1 memory request of size N/2
  }
}
```

(c) Read A in long bursts and stream them to the PE.

Listing 12: Separate memory accesses from computational logic.

An important use case of memory extraction appears in the stencil code in Lst. 11, where it is necessary to separate the memory accesses such that the PEs are agnostic of whether data is produced/consumed by a neighboring PE or by a memory module. Memory access extraction is also useful for performing data layout transformations in fast on-chip memory. For example, we can change the schedule of reads from $\boldsymbol{A}$ in Lst. 10 to a more efficient scheme by buffering values in on-chip memory, while streaming them to the kernel according to the original schedule.

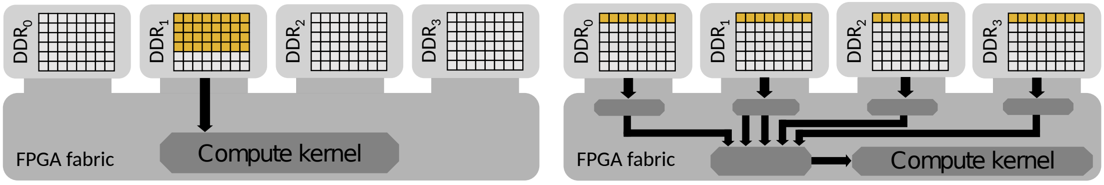
2.4.2 Memory Buffering

When dealing with memory interfaces with an inconsistent data access latency, such as DRAM, it can be beneficial to request and buffer accesses earlier and/or at a more aggressive pace than what is consumed or produced by the computational elements. For memory reads, this can be done by reading ahead of the kernel into a deep buffer instantiated between memory and computations, by either 1) accessing wider vectors from memory than required by the kernel, narrowing or widening data paths when piping to or from computational elements, respectively, or 2) increasing the clock rate of modules accessing memory with respect to the computational elements.

The memory access function Lst. 12c allows long bursts to the interface of A, but receives the data on a narrow bus at $W \cdot S_{\text{int}} = (1 \cdot 4)$ Byte/cycle. In general, this limits the bandwidth consumption to $f \cdot WS_{\text{int}}$ at frequency f, which is likely to be less than what the external memory can provide. To better exploit available bandwidth, we can either read wider vectors (increase W) or clock the circuit at a higher rate (increase f). The former consumes more resources, as additional logic is required to widen and narrow the data path, but the latter is more likely to be constrained by timing constraints on the device, and is not always directly supported by HLS tools.

2.4.3 Memory Striping

When multiple memory banks with dedicated channels (e.g., multiple DRAM modules or HBM lanes) are available, the bandwidth at which a single array is accessed can be increased by a factor corresponding the the number of available interfaces by striping it across memory banks. This optimization is employed by most CPUs transparently by striping across multi-channel memory, and is commonly known from RAID 0 configuration of disks. This optimization can become especially crucial when dealing with memory technologies that expose many small channels to provide very high memory bandwidth,



(a) Memory stored in a single bank.

(b) Memory striped across four banks.

Figure 2.6: Striping memory across memory banks increases available bandwidth.

such as high-bandwidth memory (HBM) stacks.

We can perform striping explicitly in HLS by inserting modules that join or split data streams from two or more memory interfaces. Reading can be implemented with two or more memory modules requesting memory from their respective interfaces, pushing to FIFO buffers that are read in parallel and combined by another module (for writing: in reverse), exposing a single data stream to the computational kernel. This is illustrated in Fig. 2.6, where the unlabeled dark boxes in Fig. 2.6b represent PEs reading and combining data from the four DRAM modules. The Intel OpenCL compiler [32] can apply this transformation automatically with the appropriate flags set.

2.4.4 Type Demotion

We can reduce resource and energy consumption, bandwidth requirements, and operation latency by demoting data types to less expensive alternatives that still meet precision requirements. This can lead to significant improvements on architectures that are specialized for certain types, and perform poorly on others. As of writing, Intel FPGAs only expose 32-bit floating point as hardened units, while Xilinx FPGAs expose no native floating point units at all. Since integer/fixed point and floating point computations on these architectures can thus compete for the same reconfigurable logic, using a data type with lower resource requirements increases the total number of arithmetic operations that can potentially be instantiated on the device. The largest benefits of type demotion are seen in the following scenarios:

- Compute bound architectures where the data type can be changed to a type that occupies less of *the same resources* (e.g., from 64-bit integers to 48-bit integers).
- Compute bound architectures where the data type can be moved to a type that is *natively* supported by the target architecture, such as single precision floating point on

Intel's Arria 10 and Stratix 10 devices [130].

- Bandwidth bound architectures, where performance can be improved by up to the same factor that the size of the data type can be reduced by.
- Latency bound architectures where the data type can be reduced to a lower latency operation, e.g., from floating point to integer.

In the most extreme case, it has been shown that collapsing the data type of weights and activations in deep neural networks to binary [31, 140] can provide sufficient speedup for inference that the increased number of weights makes up for the loss of precision per weight.

2.5 Software Transformations in HLS

In addition to the transformations described in the sections above, we include an overview of how well-known CPU-oriented transformations apply to HLS, based on the compiler transformations compiled by Bacon et al. [13]. These transformations are included in Tab. 2.2, and are partitioned into three categories:

- Transformations directly relevant to the HLS transformations already presented here.
- Transformations that are the same or similar to their software counterparts.
- Transformations with little or no relevance to HLS.

It is interesting to note that the majority of well-known transformations from software apply to HLS. This implies that we can leverage much of decades of research into high-performance computing transformations to also optimize hardware programs, including many that can be applied *directly* (i.e., without further adaptation to HLS) to the imperative source code or intermediate representation before synthesizing for hardware. We stress the importance of support for these pre-hardware generation transformations in HLS compilers, as they lay the foundation for the hardware-specific transformations.

2.6 End-to-End Examples

To showcase the transformations presented here and provide a "hands-on" opportunity for seeing HLS optimizations applied in practice, we will describe the optimization process on

		CPU-oriented Transformations and how they apply to HLS codes.
Directly related to HLS transformations	☞	**Loop interchange** [8, 86] is used to resolve loop-carried dependencies [§2.2].
	☞	**Strip-mining** [148], **loop tiling** [95, 86], and **cycle shrinking** [116] are central components of many HLS transformations [§2.2.1, §2.3.1, §2.3.2, §2.2.1.2].
	☞	**Loop distribution** and **loop fission** [87, 86] are used to separate differently scheduled computations to allow pipelining [§2.3.3].
	☞	**Loop fusion** [157, 86, 152] is used for merging pipelines [§2.2.4].
	☞	**Loop unrolling** [48] is used to generate parallel hardware [§2.3.1, §2.3.2].
	☞	**Software pipelining** [94] is used by HLS tools to schedule code sections according to operation interdependencies to form *hardware* pipelines.
	☞	**Loop coalescing/flattening/collapsing** [115] saves pipeline drains in nested loops [§2.2.6].
	☞	**Reduction recognition** prevents loop-carried dependencies when accumulating [§2.2.1].
	☞	**Loop idiom recognition** is relevant for HLS backends, for example to recognize shift registers [§2.2.2] in the Intel OpenCL compiler [32].
	☞	**Procedure inlining** is used to remove function call boundaries [§2.2.7].
	☞	**Procedure cloning** is frequently used by HLS tools when inlining [§2.2.7] to specialize each function "call" with values that are known at compile-time.
	☞	**Loop unswitching** [7] is rarely advantageous; its *opposite* is beneficial [§2.2.6, §2.2.4].
	☞	**Loop peeling** is rarely advantageous; its **opposite** is beneficial to allow coalescing [§2.2.6].
	☞	**SIMD transformations** is done in HLS via horizontal unrolling [§2.3.1].
	☞	**Short-circuiting**: while the logic for both boolean operands must always be instantiated in hardware, dynamically scheduling branches [81] can effectively "short-circuit" otherwise deep, static pipelines.
Same or similar in HLS	☞	**Loop-based strength reduction** [28, 18, 136], **Induction variable elimination** [6], **Unreachable code elimination** [6], **Useless-code elimination** [6], **Dead-variable elimination** [6], **Common-subexpression elimination** [6], **Constant propagation** [6], **Constant folding** [6], **Copy propagation** [6], **Forwarding substitution** [6], **Reassociation**, **Algebraic simplification**, **Strength reduction**, **Bounds reduction**, **Redundant guard elimination** are all transformations that eliminate code, which is a useful step for HLS codes to avoid generating unnecessary hardware.
	☞	**Loop-invariant code motion (hoisting)** [6] does not save hardware in itself, but can save memory operations.
	☞	**Loop normalization** can be useful as an intermediate transformation.
	☞	**Loop reversal** [6], **array padding and contraction**, **scalar expansion**, and **scalar replacement** yield the same benefits as in software.
	☞	**Loop skewing** [6] can be used in multi-dimensional wavefront codes.
	☞	**Function memoization** can be applied to HLS, using explicit fast memory.
	☞	**Tail recursion elimination** may be useful if eliminating dynamic recursion can enable a code to be implemented in hardware.
	☞	**Regular array decomposition** applies to partitioning of both on-chip/off-chip memory.
	☞	We do not consider transformations that apply only in a distributed setting (**message vectorization**, **message coalescing**, **message aggregation**, **collective communication**, **message pipelining**, **guard introduction**, **redundant communication**), but they should be implemented in dedicated message passing hardware when relevant [40].
Do not apply to HLS	☞	No use case found for **loop spreading** and **parameter promotion**.
	☞	**Array statement scalarization**: No built-in vector notation in C/C++/OpenCL.
	☞	**Code colocation**, **displacement minimization**, **leaf procedure optimization**, and **cross-call register allocation**, are not relevant for HLS, as there are no runtime function calls.
	☞	**I/O format compilation**: No I/O supported directly in HLS.
	☞	**Supercompiling**: is infeasible for HLS due to long synthesis times.
	☞	**Loop pushing/embedding**: Inlining completely is favored to allow pipelining.
	☞	**Automatic decomposition and alignment**, **scalar privatization**, **array privatization**, **cache alignment**, and **false sharing** are not relevant for HLS, as there is no (implicit) cache coherency protocol in hardware.
	☞	**Procedure call parallelization** and **split** do not apply, as there are no forks in hardware.
	☞	**Graph partitioning** only applies to explicit dataflow languages.
	☞	There are no instruction sets in hardware, so **VLIW transformations** do not apply.

Table 2.2: The relation of traditional CPU-oriented transformations to HLS codes.

a sample set of classical HPC kernels, available as open source repositories on GitHub[4]. These kernels are written in C++ for Xilinx Vitis HLS [160] with hlslib (introduced in Chapter 3) extensions, and are built and run using the Xilinx Vitis environment. For each example, we will describe the sequence of transformations applied, and give the resulting performance at each major stage.

The included benchmarks were run on an Alveo U250 board, which houses a Xilinx UltraScale+ XCU250-FIGD2104-2L-E FPGA and four 2400 MT/s DDR4 banks (we utilize 1-2 banks for the examples here). The chip consists of four almost identical chiplets with limited interconnect between them, where each chiplet is connected to one of the DDR4 pinouts. This multi-chiplet design allows more resources (1728K LUTs and 12,288 DSPs), but poses challenges for the routing process, which impedes the achievable clock rate and resource utilization for a monolithic kernel attempting to span the full chip. Kernels were compiled for the xilinx_u250_xdma_201830_2 shell with Vitis 2019.2 and executed with version 2.3.1301 of the Xilinx Runtime (XRT). All benchmarks are included in Fig. 2.7, and the resource utilization of each kernel is shown in Fig. 2.8.

2.6.1 Stencil Code

Stencil codes are a popular target for FPGA acceleration in HPC, due to their regular access pattern, intuitive buffering scheme, and potential for creating large systolic array designs [166]. We show the optimization of a 4-point 2D stencil based on Lst. 4. Benchmarks are shown in Fig. 2.7, and use single precision floating point, iterating over a 8192×8192 domain. We first measure a naive implementation, where all neighboring cells are accessed directly from the input array, which results in no data reuse and heavy interface contention on the input array. We then apply the following optimization steps:

1. Delay buffers [§2.2.2] are added to store two rows of the domain (see Lst. 4a), removing interface contention on the memory bus and achieving perfect spatial data reuse.
2. Spatial locality is exploited by introducing vectorization [§2.3.1]. To efficiently use memory bandwidth, we use memory extraction [§2.4.1], buffering [§2.4.2], and striping [§2.4.3] from two DDR banks.
3. To exploit temporal locality, we replicate the vectorized PE by vertical unrolling [§2.3.2] and stream [§2.3.3] between them (Lst. 11). The domain is tiled [§2.3.4] to limit fast memory usage.

---

[4] `https://github.com/spcl?q=hls`

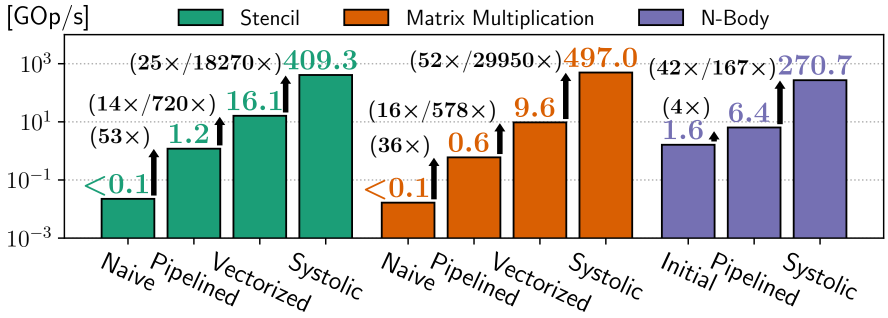


Figure 2.7: Performance progression of kernels when applying transformations. Parentheses show speedup over previous version, and cumulative speedup.

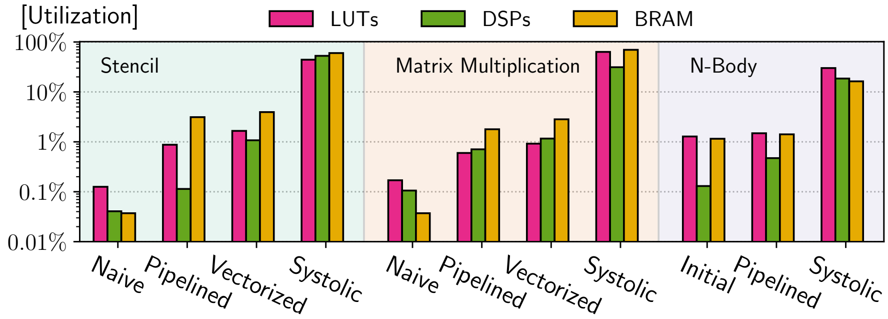


Figure 2.8: Resource usage of kernels from Fig. 2.7 as fractions of available resources. The maxima are taken as 1728K LUTs, 12,288 DSPs, and 2688 BRAM.

Enabling pipelining with delay buffers allows the kernel to throughput ∼1 cell per cycle. Improving the memory performance to add vectorization (using $W = 16$ operands/cycle for the kernel) exploits spatial locality through additional bandwidth usage. The vertical unrolling and dataflow step scales the design to exploit available hardware resources on the chip, until limited by placement and routing. The final implementation is available on GitHub[5].

2.6.2 Matrix Multiplication Code

The full optimization process of matrix multiplication kernel is presented as a comprehensive case study in Chapter 4. Here, we present an overview of performance at progressive stages of optimization for 8192×8192 matrices, shown in Fig. 2.7. Starting from a naive implementation (Lst. 1a), the following optimization stages were applied:

---

[5] https://github.com/spcl/stencil_hls/

1. We transpose the iteration space [§2.2.1.1], removing the loop-carried dependency on the accumulation register, and extract the memory accesses [§2.4.1], vastly improving spatial locality. The buffering, streaming and writing phases are fused [§2.2.4], allowing us to coalesce the three nested loops [§2.2.6].
2. In order to increase spatial parallelism, we vectorize accesses to $\boldsymbol{B}$ and $\boldsymbol{C}$ [§2.3.1].
3. To scale up the design, we vertically unroll by buffering multiple values of $\boldsymbol{A}$, applying them to streamed in values of $\boldsymbol{B}$ in parallel [§2.3.2]. To avoid high fan-out, we partition buffered elements of $\boldsymbol{A}$ into processing elements [§2.3.3] arranged in a systolic array architecture. Finally, the horizontal domain is tiled to accommodate arbitrarily large matrices with finite buffer space.

Allowing pipelining and regularizing the memory access pattern results in a throughput of ∼1 cell per cycle. Vectorization multiplies the performance by W, set to 16 in the benchmarked kernel. The performance of the vertically unrolled dataflow kernel is only limited by placement and routing due to high resource usage on the chip. The final implementation achieves state-of-the-art performance on the target architecture, and is available on GitHub[6]. For more details, see the full case study in Chapter 4.

2.6.3 N-Body Code

Finally, we show an N-body code in 3 dimensions, using single precision floating point types and iterating over 16,128 bodies. Since Vitis HLS does not allow memory accesses of a width that is not a power of two, memory extraction [§2.4.1] and buffering [§2.4.2] was included in the first stage, to support 3-vectors of velocity. We then performed the following transformations:

1. The loop-carried dependency on the acceleration accumulation is resolved by applying tiled accumulation interleaving [§2.2.1.2], pipelining across $T \geq L_+$ different resident particles applied to particles streamed in.
2. To scale up the performance, we further multiply the number of resident particles, this time replicating compute through vertical unrolling [§2.3.2] of the outer loop into P parallel processing element arranged in a systolic array. Each element holds T resident particles, and particles are streamed [§2.3.3] through the PEs.

---

[6] https://github.com/spcl/gemm_hls

The second stage gains a factor of 4× corresponding to the latency of the interleaved accumulation, followed by a factor of 42× from unrolling units across the chip. $T \geq L_+$ can be used to regulate the arithmetic intensity of the kernel. The bandwidth requirements can be reduced further by storing more resident particles on the chip, scaling up to the full fast memory usage of the FPGA. The tiled accumulation interleaving transformation thus enables not just pipelining of the compute, but also minimization of I/O. The optimized implementation is available on GitHub[7].

These examples demonstrate the impact of different transformations on a programmable spatial computing platform. In particular, enabling pipelining, regularizing memory accesses, and vertical unrolling are shown to be central components of scalable hardware architectures. The dramatic speedups over naive codes also emphasize that HLS tools do not yield competitive performance out of the box, making it critical to perform further transformations. For additional examples of optimizing HLS codes, we refer to the numerous works applying HLS optimizations listed below.

2.7 Related Work

Optimized applications. Much work has been done in optimizing C/C++/OpenCL HLS codes for FPGA, such as stencils [166, 146, 79, 149, 144], deep neural networks [138, 159, 140, 21, 31], matrix multiplication [46, 144, 39, 59], graph processing [20, 19], networking [50], light propagation for cancer treatment [158], and protein sequencing [127, 122]. These works optimize the respective applications using transformations described here, such as delay buffering, random access buffering, vectorization, vertical unrolling, tiling for on-chip memory, and dataflow.

Transformations. Zohouri et al. [165] use the Rodinia benchmark to evaluate the performance of OpenCL codes targeting FPGAs, employing optimizations such as SIMD vectorization, sliding-window buffering, accumulation interleaving, and compute unit replication across multiple kernels. We present a generalized description of a superset of these transformations, along with concrete code examples that show how they are applied in practice. The DaCe framework [16] exploits information on explicit dataflow and control flow to perform a wide range of transformations, and code generates efficient HLS code using vendor-specific pragmas and primitives. Kastner et al. [83] go through the implementation of many HLS codes in Vivado HLS, focusing on algorithmic optimiza-

---

[7] `https://github.com/spcl/nbody_hls`

tions. da Silva et al. [33] explore using modern C++ features to capture HLS concepts in a high-level fashion. Lloyd et al. [103] describe optimizations specific to Intel OpenCL, and include a variant of memory access extraction, as well as the single-loop accumulation variant of accumulation interleaving.

Directive-based frameworks. High-level, directive-based frameworks such as OpenMP and OpenACC have been proposed as alternative abstractions for generating FPGA kernels. Leow et al. [100] implement an FPGA code generator from OpenMP pragmas, primarily focusing on correctness in implementing a range of OpenMP pragmas. Lee et al. [99] present an OpenACC to OpenCL compiler, using Intel OpenCL as a backend. The authors implement horizontal and vertical unrolling, pipelining and dataflow by introducing new OpenACC clauses. Papakonstantinou et al. [114] generate HLS code for FPGA from directive-annotated CUDA code.

Optimizing HLS compilers. Mainstream HLS compilers automatically apply many of the well-known software transformations in Tab. 2.2 [11, 62, 63], but can also employ more advanced FPGA transformations. Intel OpenCL [32] performs memory access extraction into “load store units” (LSUs), does memory striping between DRAM banks, and detects and auto-resolves some buffering and accumulation patterns. Recent versions of the Vitis HLS compiler also detects and resolves some floating point accumulation patterns. The proprietary Merlin Compiler [29] uses high-level acceleration directives to automatically perform some of the transformations described here, as source-to-source transformations to underlying HLS code. The HeteroCL [93] framework can automatically optimize certain classes of applications supported by its backend components. Artisan [142] is a meta-programming approach built on HLS, which uses an approach based on separation of concerns to apply some of the transformations described here, such as vertical unrolling, vectorization, delay buffering, dataflow, inlining, and resolving loop-carried dependencies. Schuiki et al. [125] propose LLHD, an intermediate representation designed to sit between hardware description languages and the circuitry that they represent, providing a promising approach to establishing common ground for different hardware compilers and optimization frameworks. Polyhedral compilation is a popular framework for optimizing CPU and GPU loop nests [60], and has also been applied to HLS for FPGA for optimizing data reuse [119]. Such techniques may prove valuable in automating, e.g., memory extraction and tiling transformations. While most HLS compilers rely strictly on static scheduling, Dynamatic [81] considers dynamically scheduling state machines and pipelines to allow reducing the number of stages executed at runtime.

Domain-specific frameworks. Implementing programs in domain specific languages (DSLs) can make it easier to detect and exploit opportunities for advanced transformations. Darkroom [70] generates optimized HDL for image processing codes, and the popular image processing framework Halide [121] has been extended to support FPGAs [120, 101]. Luzhou et al. [104] and StencilFlow [38] propose frameworks for generating stencil codes for FPGAs. These frameworks rely on optimizations such as delay buffering, dataflow, and vertical unrolling, which we cover here. Using DSLs to compile to structured HLS code can be a viable approach to automating a wide range of transformations, as we will show with the StencilFlow framework in Chapter 6. Koeplinger et al. [85] and the FROST [135] DSL framework also follow this approach.

Other approaches. There are other approaches than C/C++/OpenCL-based HLS languages to addressing the productivity issues of hardware design: Chisel/FIRRTL [12, 77] maintains the paradigm of behavioral programming known from RTL, but provides modern language and compiler features. This caters to developers who are already familiar with hardware design, but wish to use a more expressive language. In the Maxeler ecosystem [107], kernels are described using a Java-based language, but rather than transforming imperative code into a behavioral equivalent, the language provides a DSL of hardware concepts that are instantiated using object-oriented interfaces. By constraining the input, this encourages developers to write code that maps well to hardware, but requires learning a new language exclusive to the Maxeler ecosystem.

2.8 Toolflow of Xilinx vs. Intel

When choosing a toolflow to start designing hardware with HLS, it is useful to understand two distinct approaches by the two major vendors: Intel OpenCL wishes to enable *writing accelerators using software*, making an effort to abstract away low-level details about the hardware, and present a high-level view to the programmer; whereas Xilinx' Vitis/Vivado HLS ecosystem provides *a more productive way of writing hardware*, by means of a familiar software language. Xilinx uses OpenCL as a vehicle to interface between FPGA and host, but implements the OpenCL compiler itself as a thin wrapper around the C++ compiler, whereas Intel embraces the OpenCL paradigm as their frontend (although they encourage writing single work item kernels [2], effectively preventing reuse of OpenCL kernels written for GPU).

Vitis HLS has a stronger coupling between the HLS source code and the generated hardware. This requires the programmer to write more annotations and boilerplate code, but can also give them a stronger sense of control over their architecture. Conversely, the Intel OpenCL compiler presents convenient abstracted views, saves boilerplate code (e.g., by automatically pipelining sections), and implements efficient substitutions by detecting common patterns in the source code (e.g., to automatically perform memory extraction [§2.4.1]). The downside is that developers can end up struggling to write or generate code in a way that is recognized by the tool's "black magic", in order to achieve the desired result. Finally, Xilinx' choice to allow C++ gives Vitis HLS an edge in expressibility, as (non-virtual) objects and templating turns out to be a useful tool for abstracting and extending the language (we will see multiple examples of this in Chapter 3). Intel offers a C++-based HLS compiler, but does not (as of writing) support direct interoperability with the OpenCL-driven accelerator flow.

2.9 Summary

The transformations known from software are insufficient to optimize HPC kernels targeting spatial computing systems. We have proposed a new set of optimizing transformations that enable efficient and scalable hardware architectures, and can be applied directly to the source code by a performance engineer, or automatically by an optimizing compiler. Performance and compiler engineers can benefit from these guidelines, transformations, and the presented cheat sheet as a common toolbox for developing high performance hardware using HLS. Much of the material presented here is also available in interactive form as part of our tutorial, "Productive Parallel Programming for FPGA with High-Level Synthesis", available online (see page 9), containing demos of code examples.

In Chapter 4, we will deep dive into the optimization process of a matrix multiplication kernel implemented in an HLS language, showcasing many of the transformations discussed here in a concrete setting, by simultaneously maximizing computational throughput and minimizing data movement of the application. But first, we will provide some concrete software tools to improve the productivity of HLS programming.

Chapter 3

hlslib: Software Engineering for Hardware Design

---

hlslib is an open source project[1]*, which materialized as I implemented features that I found missing or lacking in Vivado/Vitis HLS, while developing kernels such as the ones described in Chapter 2, and has since received support for Intel FPGA. hlslib was presented at H2RC'19 and is published as an extended abstract on arXiv [37]. The GitHub repository has garnered significant attention and received contributions from multiple authors.*

3.1 High-Level Synthesis Programming

Although the productivity of developing for FPGAs has improved significantly with widespread adoption of HLS, working with these tools can be a less-than-smooth experience. The imperative languages primarily used by HLS tools, namely C, C++, and OpenCL, were not designed with hardware development in mind, and the resulting opaque mapping to hardware can be unclear to both software developers (who cannot implement code in the way they are used to), and hardware developers (who struggle to achieve the exact architecture that they have in mind).

We introduce hlslib, an open source collection of tools, modules, and scripts, with the overarching goal of improving the quality of life of HLS developers. An overview of some hlslib features and which stage of development benefits from them is given in Fig. 3.1. While hlslib cannot hope to solve all the issues of HLS development, we hope to smoothen as many steps of the process as possible in a external library, and encourage good practices inspired by traditional software engineering. This chapter gives an overview of the functionality offered by hlslib as of writing, but the library is continuously developed to provide new features and to support newer versions and functionality of the two major vendor tools, Xilinx' Vitis HLS [160] (formerly Vivado HLS), and Intel's OpenCL SDK for FPGAs [32].

---

[1] https://github.com/definelicht/hlslib

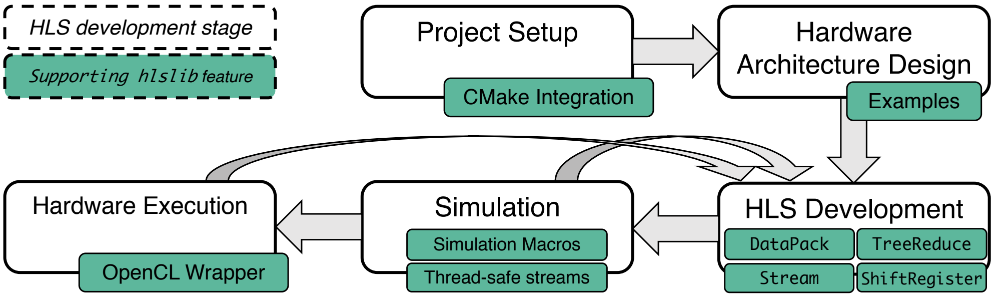


Figure 3.1: Stages of HLS development and the supporting hlslib features.

3.2 Improving the HLS Workflow

3.2.1 CMake Integration

Many existing HLS projects, including example codes published by both Xilinx and Intel through their official channels, rely on manually written GNU makefiles. This method offers poor portability, and does not allow projects to be configured without modifying the makefile or source code. In software development, CMake is a widespread tool used to configure and build C/C++ projects. Users can set project parameters during configuration, as compilation is performed out-of-source, and dependencies are automatically located on the host system in a portable fashion.

hlslib provides supports for FPGA projects in CMake, allow separation of source code and configuration through the `FindVitis.cmake` and `FindIntelFPGAOpenCL.cmake` scripts, required to locate and expose the Xilinx and Intel FPGA ecosystems, respectively. Users gain access to the HLS binaries, as well as compiler flags, header files, and library files required to build the OpenCL host code. Historically, the workflow for building and running FPGA codes with commercial HLS tools has been continuously changing throughout their development. By offloading the responsibility of setting up the HLS environment to hlslib, projects become robust to changes in the setup provided by the vendors.

An example snippet for a `CMakeLists.txt` using hlslib to build an Vitis project with a top-level function “`Top`” is given below, where hardware targets are added with custom targets, using the binaries exposed by the find-scripts:

Examples of the full flow with all relevant files are included in the hlslib repository, for both Xilinx and Intel OpenCL ecosystems.

```
set(TARGET_PLATFORM "xilinx_u250_xdma_201830_2" CACHE STRING "Vitis accelerator platform.")
set(CMAKE_MODULE_PATH hlslib/cmake)
find_package(Vitis REQUIRED)
include_directories(${Vitis_INCLUDE_DIRS} hlslib/include)
add_executable(MyHostCode MyHostCode.cpp)
target_link_libraries(MyHostCode ${Vitis_LIBRARIES})
add_custom_target(compile_hardware COMMAND ${Vitis_COMPILER}
                  --platform ${TARGET_PLATFORM} --kernel Top
                  -c -t hw Kernel.cpp -o Kernel.xo)
add_custom_target(link_hardware COMMAND ${Vitis_COMPILER}
                  --platform ${TARGET_PLATFORM} --kernel Top
                  -l -t hw Kernel.xo -o Kernel.xclbin)
```

Listing 13: Creating custom FPGA targets with hlslib CMake support.

3.2.2 Portable OpenCL Host Code

OpenCL was originally developed for GPUs, and thus follows the GPU model of creating computational kernels, transferring data in bulk between host and device memories, and launching discrete compute kernels. OpenCL is exposed as a host-side interface by both Intel and Xilinx for launching computational kernels and interacting with device DRAM.

Intel and Xilinx have taken slightly different approaches to adapting OpenCL to FPGAs. In order to enable a fully unified interface, hlslib provides an OpenCL wrapper that hides subtle differences between vendors, such as distinct ways of specifying which memory bank a buffer should be instantiated on. An example of a basic hlslib OpenCL host program is given in Lst. 14, which is valid code for both the Intel and Xilinx ecosystems (example file name uses the Xilinx `.xclbin` suffix).

```
using namespace hlslib::ocl;
Context context;  // Sets up the vendor OpenCL runtime
auto program = context.MakeProgram("KernelFile.xclbin");  // Or KernelFile.aocx
std::vector<float> input_host(N, 5), output_host(N);
auto input_device = context.MakeBuffer<float, Access::read>(MemoryBank::bank0, input_host.cbegin(),
                                                            input_end.cend());
auto output_device = context.MakeBuffer<float, Access::write>(MemoryBank::bank1, N);
auto kernel = program.MakeKernel("Kernel", in_device, out_device, N);
kernel.ExecuteTask();  // Synchronous kernel execution
output_device.CopyToHost(output_host.begin());
```

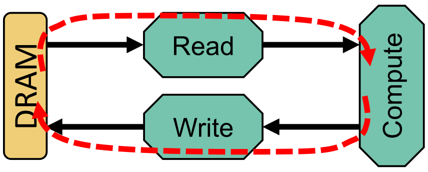
Listing 14: Portable OpenCL host program implemented with the hlslib wrapper.

```
void Top(int const *mem0, int *mem1) {
  #pragma HLS DATAFLOW
  hlslib::Stream<int> s0, s1;
  Read(mem0, s0);    // Sequential in software,
  Compute(s0, s1);  // parallel in hardware
  Write(s1, mem1);
}

void Compute(hlslib::Stream &s0,
             hlslib::Stream &s1) {
  for (int t = 0; t < T; ++t) {
    for (int i = 0; i < N; ++i) {
      #pragma HLS PIPELINE
      int read = s0.Pop();
      int res = /* do compute */;
      s1.Push(res);
    }
  }
}
```

```
void Read(int const *mem0, hlslib::Stream &s) {
  for (int t = 0; t < T; ++t) {
    for (int i = 0; i < N; ++i) {
      #pragma HLS PIPELINE
      s.Push(mem0[i]);
} } }

void Write(hlslib::Stream &s, int *mem1) {
  for (int t = 0; t < T; ++t) {
    for (int i = 0; i < N; ++i) {
      #pragma HLS PIPELINE
      mem1[i] = s.Pop();
} } }
```



Listing 15: Software and hardware behavior is different for cyclic dataflow.

3.2.3 Emulating Multiple Processing Elements in Software

Accurately emulating the semantics of multiple concurrent processing elements (PEs) executing in hardware is critical to the testing process, as multiple PEs are vital to any high performance architecture. PEs typically communicate via blocking channels, implying synchronization points between them. Emulating concurrent PEs thus requires a multi-threaded environment with thread-safe constructs.

In the Intel OpenCL ecosystem, PEs are expressed as OpenCL kernels that are launched separately from the host code, and communication channels are expressed as global objects that are accessed within the kernel codes. When running emulation in software, PEs are thus launched as concurrent threads by the runtime. On the other hand, Xilinx HLS instantiates PEs from functions or loops "called" in a scope annotated with the `DATAFLOW` pragma. While this allows expressing communication between kernels with multiple PEs in a more explicit fashion, it also means that the behavior of executing the code when compiled as C++ code can differ significantly from its behavior when built for hardware. An example of this is shown in Lst. 15, when `mem0` and `mem1` are passed as *pointers to the same address*.

Programs with cyclic dataflow between PEs are not officially supported by Xilinx, but will compile and run in practice, albeit without any guarantees of correctness. Regardless, it can be desirable to write such programs for high-performance implementations of iterative algorithms, such as iterative stencil computations, where the same DRAM memory addresses are read and written multiple times during execution. In such a scenario, a program like Lst. 15 will exhibit different behavior in software and hardware:

- In software, `Read` will execute all $T \cdot N$ iterations before `Compute` is called, which will execute all $T \cdot N$ iterations before `Write` is called. Assuming that the streams `s0` and `s1` are unbounded in software, each iteration t will perform exactly the same computation.
- In hardware, `Read`, `Compute`, and `Write` will run concurrently, and `s0` and `s1` will be bounded with size 1. The PEs will thus stay synchronized. Each iteration t of `Read` will read values written by the previous iteration of `Write`, assuming N is significantly larger than the pipeline depth.

When feedback happens directly between PEs and there is a cycle in the channels interconnecting them, in the best case, programs will crash or not terminate in software. In the worst case, programs like Lst. 15 will produce different results in software and hardware, because the feedback dependency on `mem` is not enforced.

To more accurately emulate kernels with multiple PEs and potential feedback dependencies, hlslib provides a set of thin wrapper macros that mitigate the difference between the compiled C++ and the hardware generated HLS, that can be used in conjunction with **`hlslib::Stream`** wrapper objects (see Sec. 5.7.5) to run PEs concurrently in a communication synchronous fashion. Programs only need to wrap every function call in a `DATAFLOW` section in an hlslib-defined macro, as shown in Lst. 16, which is a modified version of the top-level function from Lst. 15.

```
void Top(int const *mem0, int *mem1) {
  #pragma HLS DATAFLOW
  hlslib::Stream<int> s0, s1; // hlslib streams are thread-safe
  HLSLIB_DATAFLOW_INIT();
  HLSLIB_DATAFLOW_FUNCTION(Read, mem0, s0);  // In simulation mode,
  HLSLIB_DATAFLOW_FUNCTION(Compute, s0, s1); // each call launches
  HLSLIB_DATAFLOW_FUNCTION(Write, s1, mem1); // a separate C++ thread
  HLSLIB_DATAFLOW_FINALIZE(); // Joins C++ threads
}
```

Listing 16: PEs in `DATAFLOW` section annotated to emulate hardware behavior.

Behind the scenes, each `HLSLIB_DATAFLOW_FUNCTION` macro chooses between two kinds of behavior, depending on the compilation mode:

- In hardware, all annotated functions are simply inlined, resulting in code identical to Lst. 15, which will run each function as a parallel processing element.
- In software, each function is executed in a newly launched C++ thread. When `HLSLIB_DATAFLOW_FINALIZE` is called, hlslib will wait on each of the launched threads, returning when all PEs have terminated.

The software behavior means that PEs cannot run ahead of others more than what is allowed by the depth of the channels between them, which also allows debugging deadlocks due to channel sizes (i.e., depth of the FIFOs implementing them in hardware). When a `Pop` or `Push` from/to a channel has waited for a configurable amount of time without receiving data, hlslib will print a warning with the channel name and operation, enabling easier debugging of deadlocks.

3.3 Object-Oriented Hardware Design

Classes can provide excellent encapsulation for hardware concepts, combining data and functionality in the spirit of object-oriented programming, but also allow specializing classes with C++ templates allows parameters to be specified at compile-time, when this is necessary for generating hardware. hlslib uses classes both in the object-oriented sense, and to exploit template metaprogramming.

3.3.1 Streams/Channels

Channel objects are ubiquitous in HLS programming, either as communication primitives between processing elements, or as buffers with FIFO semantics. Channels in Intel OpenCL are global objects, while channels in Vitis HLS are created as templated `hls::stream` objects.

hlslib extends the Vitis HLS built-in `hls::stream` class in the `hlslib::Stream` class, which adds a number of additional features and streamlines the interface. Most notably, hlslib streams are thread-safe, and support the features offered by the hlslib multiple-PE simulation functionality (example usage shown was shown in Lst. 15 and 16). Furthermore, streams are bounded by default, like the hardware implement they represent. If no argument is specified, the default Vitis HLS implementation is used, which is a ping-pong buffer. Any other depth will implement a FIFO using a resource suggested by the tool, or specified by an optional template argument (e.g., SRL, LUTRAM, BRAM, or UltraRAM).

3.3.2 Wide Data Buses and Vectorization

Instantiating wide data paths in HLS is necessary to utilize all available memory bandwidth (Sec. 2.4), and to achieve parallel architectures through vectorization (Sec. 2.3.1). In practice, this is typically done either by unrolling loops and relying on the tool to infer wide data accesses, or by using types that explicitly specify the vector size, such as OpenCL vector types for Intel OpenCL, or `ap_uint` for Vitis HLS. OpenCL vector types only expose a small, limited set of types and vector lengths, and `ap_uint` requires tedious and error-prone casting to implement vector types in hardware.

hlslib provides the templated `DataPack` class for Vitis HLS, which exposes a versatile interface for implementing wide buses, registers, memory interfaces, and computations that consist of multiple data elements. Unlike `ap_uint`, `DataPack` is typed, allowing native indexing of elements for both reading and writing, supports element-wise operations (shown in Lst. 17), and convenience functions for concerting to and from C-style arrays and `ap_uint` types.

```
hlslib::DataPack<float, 4> Direction(hlslib::DataPack<float, 4> &a, hlslib::DataPack<float, 4> &b) {
  auto d = b - a; // Vector operations
  auto len = c[0] + c[1] + c[2] + c[3]; // Indexing
  return 1/len * d; // Element-wise operations
}
```

Listing 17: Overview of `hlslib::DataPack` functionality.

When used as the data type for pointer or stream arguments, `DataPack` enforces bus widths corresponding to the byte width formed by the data type vector size. If used consistently, simply changing the width of a centrally defined `DataPack`-based type will be sufficient to adjust registers, buses, buffers, and interfaces throughout an HLS code.

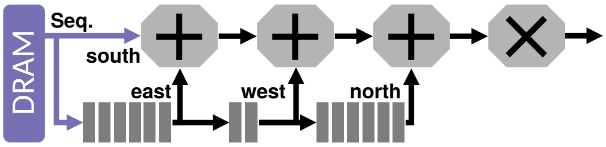
3.3.3 Shift Registers with Parallel Access

A common pattern for FPGA algorithms [165] is to buffer elements streamed in for a *constant* number of cycles, thus “delaying” them for future iterations, as described in Sec. 2.2.2 (e.g., to be used as a different element of a *sliding window* in a stencil computation [52, 166]). This is similar to a FIFO buffer, with the added constraint that elements pushed and popped are at a constant distance (e.g., for a buffer of size 4, an element pushed can only be accessed again when it comes out at the end, i.e., after 4 additional pushes). We will refer to these types of buffers as *shift registers* according to the Intel

FPGA nomenclature, although it also common to implement these in BRAM/M20K on-chip memory. We assume that shift registers have a single input, but can have multiple parallel outputs (known as "taps").

In Intel OpenCL, shift registers are inferred as a pattern when an unrolled loop shifts an array by a constant offset every cycle of a pipelined section, and the remainder of the section only accesses the array using constant indices. The compiler can then infer the distance between each tap, allowing it to instantiate separate buffers in hardware between them, effectively partitioning the single array into multiple smaller buffers. Vitis HLS, on the other hand, does not recognize this as a high-level pattern (as of writing this work), and will textually unroll the shifting loop in the preprocessor and analyze the unrolled code, which does not scale with large shift registers.

We express the parallel shift register abstraction as a templated class in hlslib, transparently managing buffers between each tap. Unlike the Intel ecosystem, hlslib shift registers are explicitly instantiated by the programmer (as opposed to relying on pattern detection), and enforce constant offset access at compile-time, while providing the full abstraction to the Vitis HLS ecosystem, which otherwise requires this pattern to be implemented manually. An implementation of a 4-point 2D stencil code based on an hlslib shift register is shown in Lst. 18.



```
void Stencil(hlslib::Stream<float> &in, hlslib::Stream<float> &out) {
  // Explicitly declare taps as template arguments
  hlslib::ShiftRegister<float, 0, W - 1, W + 1, 2 * W> sr;
  // H and W are compile-time constants
  for (int i = 0; i < H; ++i) {
    for (int j = 0; j < W; ++j) {
      #pragma HLS PIPELINE
      sr.Shift(in.Pop()); // Push new element and shift buffer
      if (i >= 2 && j >= 1 && j < W - 1) { // Ignore boundary
        // Specify tap to access using compile-time indices
        float res = 0.25 * (sr.Get<2 * W>() + sr.Get<W - 1>() +
                            sr.Get<W + 1>() + sr.Get<0>());
        out.Push(res);
} } } }
```

Listing 18: Explicit shift register abstraction provided by hlslib.

```
using Vec = DataPack<float, 8>;
void Reduce(hlslib::Stream<Vec> &in, hlslib::Stream<Vec> &out) {
  for (int i = 0; i < 1024; ++i) {
    #pragma HLS PIPELINE
    auto v = in.Pop();
    auto r = hlslib::TreeReduce<float, hlslib::op::Add<float>, 8>(v);
    out.Push(r);
} }
```

Listing 19: Explicit balanced tree reduction of an array.

Variadic template arguments are used to instantiate taps, where the distance between each consecutive index is used to compute the respective buffer size (as a result, indices must be specified in ascending order).

3.3.4 Tree Reduction with Functors

To perform a fully pipelined reduction of an array of elements for an associative operator, it is common to implement the reduction as a balanced binary tree to minimize latency and resource utilization. Implementing reduction trees in an imperative language requires the compiler to recognize unrolled loops that accumulate into a single variable, and requires explicitly allowing the compiler to reorder non-associative operations, such as floating point addition.

To guarantee that a reduction is performed as a balanced binary tree, hlslib provides the `TreeReduce` templated function, which uses variadic templates to explicitly instantiate the full tree in hardware. The template supports any type, array size, and binary operator. An example is shown in Lst. 19.

hlslib supports a set of common binary operators by default, but custom operators can be implemented with a functor struct that defines the `Apply` binary function and an `identity` for the operator. These functors are conveniently expressible using C++ templated classes.

3.4 Projects Using hlslib

All the features described in this work were tested to meet the demands of concrete HLS codes. The repository holds additional niche features left out here, as well as a compilation of examples testing and demonstrating various concepts.

We maintain a list of projects leveraging hlslib on the repository page. The matrix multiplication code described in Chapter 4 uses hlslib vector types (§3.3.2), the simulation framework (§3.2.3) and thread-safe streams (§5.7.5), and the OpenCL wrapper (§3.2.2). The *Data Centric Parallel Programming* (DaCe) project (Chapter 5) uses hlslib emulation functionality (§3.2.3), streams (§5.7.5), and vector types (§3.3.2) are used to generate code for the Xilinx backend. The OpenCL wrapper (§3.2.2) is used for generating code for host-device interaction, and CMake is used for configuration (§3.2.1). The reference implementation of the *Streaming Message Interface* (SMI) [40], a distributed memory inter-FPGA communication model specification unifying message passing with the streaming model of pipelined HLS codes, also uses hlslib for OpenCL integration.

3.5 Summary

hlslib is a collection of tools that aims to improve the productivity of HLS developers, including CMake integration, and classes for vectorization, thread-safe simulation, reduction patterns, and host/device interaction. As tools develop, hlslib aims to continue to evolve and adapt to new features and incorporate new ideas for how to close the productivity gap. For HLS development to become truly productive, the field must see many open source efforts, with active exchange of knowledge and pooling of developer effort, so that hardware design can reap the benefits of open source development that we know from the software domain.

As of writing, the open source hlslib repository (see link on page 47) has 178 stars, 33 forks, and 5 contributors on GitHub.

Chapter 4

A Case Study with Matrix Multiplication

This chapter is based on work published at FPGA'20 [39]. The theory on minimizing I/O in matrix multiplication that serves as background for the I/O model was developed by Grzegorz Kwasniewski [91]. The model developed for this work incorporates FPGA hardware constants to guide the architecture, and was done in collaboration between Grzegorz Kwasniewski and myself. I designed the hardware architecture, wrote the code, and conducted all experiments. The resulting open source implementation[1] won the Compute Acceleration category of the Xilinx Open Hardware Competition 2020[2].

4.1 Minimizing I/O of Linear Algebra Kernels

In this chapter, we will apply the techniques described in Chapter 2 in a thorough case study of Matrix-Matrix Multiplication (MMM), where we use a theoretical I/O model to optimize data movement in addition to computational performance, following a trend in both academia [134, 126, 44, 5] and industry [73, 9] to reduce data movement to mitigate the diverging evolution of computational speeds and memory bandwidth discussed in Sec. 1.1. Being ubiquitious in both scientific and industrial applications, linear algebra has been the subject of much work on this topic [80, 75, 134, 56, 10]. Minimizing I/O impacts not only performance, but also reduces bandwidth usage in a shared system. MMM is typically used as a component of larger applications [143, 42], where it co-exists with other algorithms, e.g., memory bound linear algebra operations such as matrix-vector/vector-vector operations, which benefit from a larger share of the bandwidth, but do not require large amounts of compute resources.

4.1.1 Minimizing I/O on FPGAs

FPGAs are an excellent platform for accurately modeling performance and I/O to guide algorithm implementations. In contrast to software implementations, replacing cache with

---

[1] `https://github.com/spcl/gemm_hls`

[2] `http://www.openhw.eu/2020-results.html`

explicit on-chip memory, and isolation of the instantiated architecture, can yield fully deterministic behavior in the circuit: accessing memory, both on-chip and off-chip, can be done explicitly, rather than by a cache replacement scheme fixed by the hardware. The models established so far, however, pose a challenge for their applicability on FPGAs. They often rely on abstracting away many hardware details, assuming several idealized processing units with local memory and all-to-all communication [134, 5, 80, 75]. Those assumptions do not hold for FPGAs, where the physical area size of custom-designed processing elements (PEs) and their layout are among most important concerns in designing efficient FPGA implementations [98]. Therefore, performance modeling for reconfigurable architectures requires taking constraints like logic resources, fan-out, routing, and on-chip memory characteristics into account.

With an ever-increasing diversity in available hardware platforms, and as low-precision arithmetic and exotic data types are becoming key in modern DNN [17] and linear solver [68] applications, extensibility and flexibility of hardware architectures will be crucial to stay competitive. Existing high-performance FPGA implementations [109, 82] are implemented in hardware description languages (HDLs), which drastically constrains their maintenance, reuse, generalizability, and portability. Furthermore, the source code is not disclosed, such that third-party users cannot benefit from the kernel or build on the architecture.

In this chapter, we address the above issues, while showcasing a real-world scenario of applying the toolbox of transformations presented in Chapter 2. We present a high-performance, communication avoiding MMM algorithm, which is based on both computational complexity theory [80] (Sec. 4.3.2), and on detailed knowledge of FPGA-specific features (Sec. 4.4). Our architecture is implemented in pure C++ with a small and readable code base, and to the best of our knowledge, was the first open source, high-performance MMM FPGA code. We do not assume the target hardware, and allow easy configuration of platform, degree of parallelism, buffering, data types, and matrix sizes, allowing kernels to be specialized to the desired scenario. The contributions of this chapter are:

- We model a decomposition for matrix multiplication that simultaneously targets maximum performance and minimum off-chip data movement, in terms of hardware constants.
- We design a mapping that allows the proposed scheme to be implemented in hardware, using the model parameters to lay out the architecture.

```
for (i = 0; i < M; i++)
    for (j = 0; j < N; j++)
        for (k = 0; k < K; k++)
            C[i, j] = C[i, j] + A[i, k]*B[k, j];
```

Listing 20: Classical MMM algorithm.

- We provide a plug-and-play, open source implementation of the hardware architecture in pure HLS C++, enabling portability across FPGA and demonstrating low code complexity.
- We include benchmarks for a wide range of floating point and integer types, and show the effect of adjusting parallelism and buffer space in the design, demonstrating the design's flexibility and scalability.

4.2 Optimization Goals

In this section we introduce *what* we optimize. In Sec. 4.3.2, 4.3.3, and 4.4 we describe *how* this is achieved. We consider optimizing the schedule of a *classical* MMM algorithm, that is, given a problem of finding C, where $C = AB, A \in \mathbb{R}^{m\times k}, B \in \mathbb{R}^{k\times n}, C \in \mathbb{R}^{m\times n}$, an algorithm performs $F = mnk$ multiplications and additions (pseudocode shown in Lst. 20). We therefore exclude Strassen-like routines [137] from our analysis, as the classical algorithms often perform better on practical problems and hardware [34]. We require that the optimal schedule: (1) achieves *highest performance* (has the lowest time-to-solution time), while (2) performing the *least number of I/O operations*, by (3) *making the most efficient use of resources*.

4.2.1 Optimizing Computations

On load/store architectures, such as CPUs and GPUs, optimizing the computations of scientific applications typically only require targeting the native vector units and spawning enough parallel threads to saturate all vectorized processor cores. In contrast, exploiting all available computational logic on an FPGA requires significantly more engineering effort, as computations must be grouped and distributed on the chip in a routable fashion, and the degree of parallelism that must be exploited is very high due to the (relatively) low clock frequencies. Laying out the computational logic must thus be taken into account when decomposing the target application for hardware execution.

4.2.2 Optimizing I/O

The decomposition of a problem for execution on a parallel architecture is constrained by the data movement between the memory subsystem and the computational units, and by the data movement between computational units. Both aspects are related to the relevant FPGA resources (e.g., off-chip bandwidth and on-chip memory) and the dependencies between operators on the chip (e.g., partitioning of on-chip memory, and fan-in/fan-out to off-chip and on-chip memory).

4.2.3 Optimizing Resources

When targeting a high utilization design on FPGA, it is critical to maintain characteristics that aid the routing process. Routing reacts poorly to large fan-in or fan-out, which typically occurs when these are dependent on the degree of parallelism: that is, if N determines the degree of parallelism in the program, 1-to-N and N-to-1 connections in the architecture should be avoided. This is true both on the granularity of individual logic units, and on the granularity of coarse-grained modules instantiated by the programmer. To accommodate this, we can regulate the size of PEs, and favor PE topologies that are easily mapped to a plane, such as grids or chains. Furthermore, mapping of a hardware architecture to the chip logic and interconnect (placement and routing) may reduce the clock frequency due to long routing paths. Due to the intractable size of the configuration space, this cannot be efficiently modeled and requires empirical evaluation of designs. The routing challenges are exasperated in FPGA chips that consist of multiple “chiplets”, such as the Xilinx UltraScale+ VU9P chip used in the following experiments, which hosts three “super-logical regions” (SLRs). Crossing the chiplets consumes highly limited routing resources and carries a higher timing penalty. Limiting these crossings is thus key to scaling up resource utilization.

4.2.4 Notation

Throughout this chapter, we use a two-level notation for naming parameters (Tab. 4.1). Most parameter names are on the form of α_β, where α refers to some quantity, such as the total number of objects, and β denotes the object of interest. For example, N_c, N_b, s_b are: total number of (N) compute units (c), memory blocks (b), and the size of each memory block (s), respectively.

The target hardware contains d types of different logic resources. This typically consists of

MMM	$\vec{i}$, $\vec{j}$, $\vec{k}$	Unit vectors in the 3D iteration space.
	m, n, k	Matrix sizes in $\vec{i}$, $\vec{j}$, and $\vec{k}$ dimensions, respectively.
	$\boldsymbol{A}$, $\boldsymbol{B}$	Input matrices $\boldsymbol{A} \in \mathbb{R}^{m \times k}$ and $\boldsymbol{B} \in \mathbb{R}^{k \times n}$.
	$\boldsymbol{C} = \boldsymbol{AB}$	Output matrix $\boldsymbol{C} \in \mathbb{R}^{m \times n}$.
naming	α_β	Parameter naming convention. α is some quantity (i.e., size or number of objects), β is an object that α refers to.
	$\alpha_{\beta,\max}$	Hardware limit on a parameter α_β.
	$\alpha_{\text{tot}} = \prod_\beta \alpha_\beta$	The product of all tile sizes.
$\boldsymbol{\alpha}$	N	Total number of objects.
	x, y	Number of objects in the **i**th **j**th dimension of the given tile.
	s	Intrinsic size of an object.
	w	Bit width of object (e.g., of port or data type).
	$\vec{r}$	Vector of logic resources.
$\boldsymbol{\beta}$	c	Compute units in a processing element.
	p	Processing elements in a compute tile.
	t	Compute tiles in a block tile.
	b	Block tiles in a memory tile.
optimization	$S = N_b \cdot s_b$	Total size of available on-chip memory.
	$N_c \leq N_{c,\max}$	Total number of compute units.
	Q	Total number of off-chip memory transfers.
	$F = n \cdot m \cdot k$	Total number of multiply-addition operations required to perform MMM.
	$f \leq f_{\max}$	Achieved and maximum design frequency.
	$T = \frac{F}{f \cdot N_c}$	Design total execution time.

Table 4.1: The most important symbols used in this chapter.

general purpose logic, such as *lookup tables* (LUTs), and more specialized arithmetic units, such as *digital signal processing units* (DSPs). We represent a quantity of these resources as a vector $\vec{r}_{\max} = [r_{1,\max}, ..., r_{d,\max}]$. As a basic logical entity, we consider a "compute unit", which is a basic circuit able to perform a single multiply-addition operation in a single cycle. Each unit is implemented on the target hardware using some combination of logic resources $\vec{r}_c$. Depending on the numerical precision, different numbers of computational resources $\vec{r}_c$ are needed to form a single compute unit, so the maximum number of compute units that can be instantiated, N_c, may vary. The compute units are organized into N_p *processing elements* (PEs), which encapsulate a closed logical task (e.g., a vector operation) of $x_c \cdot y_c$ compute units. Each PE additionally consumes $\vec{r}_p$ logic resources as orchestration overhead. This gives us the following constraint, which enforces that the total amount of resources consumed by compute units and their encompassing PEs should not exceed the total resources available:

$$\begin{aligned} \forall_{1 \leq i \leq d} N_c r_{i,c} + N_p r_{i,p} \leq r_{i,\max}, \\ \text{or equivalently } \forall_{1 \leq i \leq d} N_p (r_{i,p} + r_{i,c} \cdot x_c y_c) \leq r_{i,\max} \end{aligned} \tag{4.1}$$

where d is the dimensionality of the resource vector.

4.3 Optimization Models

In this section, we describe the models for computation, I/O, resource usage, and constraints by the resources used that achieve the goals defined in Sec. 4.2 in terms of FPGA hardware constants, which lay the ground for the decomposition that will guide the implementation.

4.3.1 Computation Model

To optimize computational performance we minimize the total execution runtime, which is a function of achieved parallelism (total number of compute units N_c) and the design clock frequency f. The computational logic is organized into N_p PEs, and we assume that every PE holds $x_c \cdot y_c$ compute units in dimensions **x** and **y** (see Tab. 4.1 for an overview of all symbols used). We model the factor N_c directly in the design, and rely on empirically fixing f, which is limited by the maximum size of data buses between PEs (i.e., $x_c w_c \leq w_{p,\max}$ and $y_c w_c \leq w_{p,\max}$, where $w_{p,\max}$ depends on the architecture, and typically takes values up to 512 bit). Formally, we can write the computational optimization problem as follows:

$$\begin{aligned}
\text{minimize } T = \frac{F}{f \cdot N_c} &= \frac{mnk}{f \cdot N_p \cdot x_c y_c} \\
\text{subject to:}& \\
\forall_{1 \le i \le d} N_p(r_{i,p} + r_{i,c} \cdot x_c \cdot y_c) &\le r_{i,\max} \\
x_c w_c &\le w_{p,\max} \\
y_c w_c &\le w_{p,\max} \\
f &\le f_{\max}
\end{aligned} \tag{4.2}$$

That is, the time to completion T is minimized when $f \cdot N_c$ is maximized, where the number of parallel compute units N_c is constrained by the available logic resources $\vec{r}_{\max}$ of the design (this can be the full hardware chip, or any desired subset resource budget). We respect routing constraints by observing a maximum bus width $w_{p,\max}$, and must stay within the target frequency $f_{\max}$.

4.3.2 I/O Model

Previous work on I/O models for linear algebra [80, 74, 134] assume a system with a fixed number of processors p, each with a fixed size local memory of some constant size S. Under these assumptions, COSMA [91] shows that each processor should perform mnk/S subcomputations (a subcomputation is denoted V_i), each loading $2\sqrt{S}$ operands from matrices A and B, then performing S multiply-addition operations. When considering FPGAs, these assumptions no longer hold: computational logic and fast memory is distributed on the chip, such that the granularity of "processors" (or processing *elements*) is determined by the programmer and depends on the types of resources available on the target device, and on the types of operations performed (thus varying the number of processing elements p). The mapping between these processing elements and the fast memory that they access is furthermore constrained by the port widths, capacities, and routing constraints of the types of on-chip memory resources used (such as registers and BRAM).

For the I/O model developed here, we make the assumption that compute and memory resources must be equally distributed among processing elements, posing additional restrictions on a number of available resources and their distribution for each subcomputation to secure maximum arithmetic throughput and routing feasibility:

1. The number of parallel compute units N_c is maximized.

2. The work is load balanced, such that each compute unit performs the same number of computations.
3. Each memory block is routed to only one compute unit (i.e., they are not shared between compute units).
4. Each processing element p performs the same logical task, and consumes the same amount of computational and memory block resources.

4.3.2.1 Memory Resources

To model the memory resources of the FPGA, we consider the bit-length of each operand, w_c, which depends on the chosen precision (e.g., 16 bit for half precision floating point, or a 64 bit for long unsigned integers). The machine contains N_b on-chip memory blocks, each capable of holding s_b words of the target data type, yielding a maximum of

$$S = N_b \cdot s_b$$

words that can be held in on-chip memory. s_b takes different values depending on w_c (the relationship between s_b is not necessarily linear, as memory resources can expose a fixed number of “configurations” that map one to the other). We model each memory block as supporting one read and one write of up to w_b bits in a single cycle in a pipelined fashion.

4.3.2.2 FPGA-constrained I/O Minimization

We denote each subcomputation V_i as a *memory tile* M, as its size in $\vec{i}$ and $\vec{j}$ dimensions determines the memory reuse. To support a hierarchical hardware design, each M is further decomposed into several levels of tiling. This decomposition encapsulates hardware features of the chip, and imposes several restrictions on the final shape of M. The tiling scheme is illustrated in Fig. 4.1. We will cover the purpose and definition of each layer in the hierarchy shortly in Sec. 4.3.3.

Following the result from COSMA [91], a schedule that minimizes the number of I/O operations loads x_{tot} elements of one column of matrix $\boldsymbol{A}$, y_{tot} elements of one row of matrix $\boldsymbol{B}$ and reuses $x_{\text{tot}}y_{\text{tot}}$ previous partial results of $\boldsymbol{C}$, thus computing an outer product of the loaded row and column. The arithmetic intensity of the program then corresponds to the arithmetic intensity of a single subcomputation, which we can write in terms of x_{tot} and y_{tot} as:

$$\frac{x_{\text{tot}}y_{\text{tot}}}{x_{\text{tot}} + y_{\text{tot}}}. \tag{4.3}$$

```
// Memory tiles m
for ($i_m = 1$; $i_m \leq n$; $i_m = i_m + x_{\texttt{tot}}$)
  for ($j_m = 1$; $j_m \leq m$; $j_m = j_m + y_{\texttt{tot}}$)
    for ($k = 1$; $k \leq k$; $k = k + 1$) // Full dimension k
// [Sequential] Block tiles b in memory tile
      for ($i_b = i_m$; $i_b \leq i_m + x_{\texttt{tot}}$; $i_m = i_m + x_t x_c x_p$)
        for ($j_b = j_m$; $j_b \leq j_m + y_{\texttt{tot}}$; $j_m = j_m + y_t y_p y_c$)
// [Sequential] Compute tiles t in block tile
          for ($i_t = i_b$; $i_t \leq i_b + x_t x_p x_c$; $i_b = i_b + x_c x_p$)
            for ($j_t = y_b$; $j_t \leq j_b + y_t y_p y_c$; $j_t = j_t + y_c y_p$)
// [Parallel] Processing elements p in compute tile
              forall ($i_p = i_t$; $i_p \leq i_t + x_p x_c$; $i_t = i_t + x_c$)
                forall ($j_p = j_t$; $j_p \leq j_t + y_p y_c$; $j_p = j_p + y_c$)
// [Parallel] Compute units c in processing element
                  forall ($i_c = i_p$; $i_c \leq i_p + x_c$; $i_c = i_c + 1$)
                    forall ($j_c = j_p$; $j_c \leq j_p + y_c$; $j_c = j_c$+1)
                      $\boldsymbol{C}(i_c, j_c) = \boldsymbol{C}(i_c, j_c) + \boldsymbol{A}(i_c, k) \cdot \boldsymbol{B}(k, j_c)$
```

Listing 21: Pseudocode of the tiled MMM algorithm.

To minimize I/O, we maximize the arithmetic intensity given the total amount of fast memory available on the device S, yielding

$$\begin{aligned} &\text{maximize } \frac{x_{\text{tot}} y_{\text{tot}}}{x_{\text{tot}} + y_{\text{tot}}} \\ &\text{subject to: } x_{\text{tot}} + y_{\text{tot}} + x_{\text{tot}} y_{\text{tot}} \leq S, \end{aligned} \tag{4.4}$$

and the total number of I/O operations performed by the program, denoted as the communication volume Q is

$$Q = mn + \frac{mnk}{x_{\text{tot}}} + \frac{mnk}{y_{\text{tot}}} = mn \left(1 + k \left(\frac{1}{x_{\text{tot}}} + \frac{1}{y_{\text{tot}}} \right) \right). \tag{4.5}$$

This expression is minimized when:

$$x_{\text{tot}} = y_{\text{tot}} = \sqrt{S} \tag{4.6}$$

That is, a memory tile M should be a square of area S with sides x_{tot} and y_{tot}. Eq. 4.5 therefore gives us a theoretical lower bound on $Q \leq 2mnk/\sqrt{S}$, assuming that all available memory can be used effectively. However, the assumptions stated in Sec. 4.3.2 constrain the perfect distribution of hardware resources, which we model in Sec. 4.3.3.

4.3.3 Resource Model

Based on the I/O model and the FPGA constraints, we create a logical hierarchy which encapsulates various hardware resources, which will guide the implementation to maximize

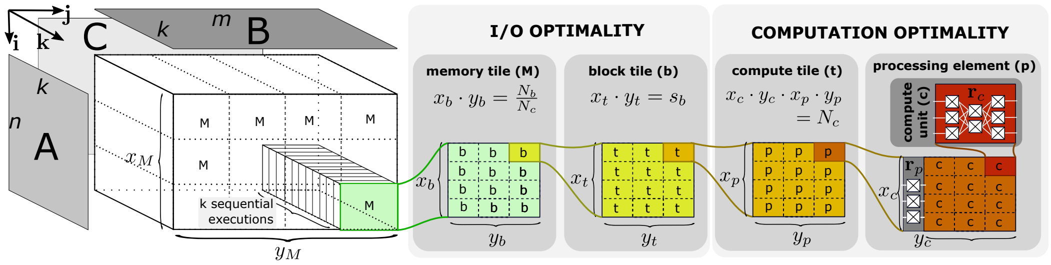


Figure 4.1: Decomposition of MMM achieving both performance and I/O optimality in terms of hardware resources.

I/O and performance. We assume a chip contains $\vec{r}_{\max} = \{r_{1,\max}, \ldots, r_{t,\max}\}$ different hardware resources (see Sec. 4.2.4). The dimensionality and length of a vector depends on the target hardware – e.g., Intel Arria 10 and Stratix 10 devices expose native floating point DSPs, each implementing a single operation, whereas a Xilinx UltraScale+ device requires a combination of logic resources. We model fast memory resources separately as memory blocks (e.g., M20K blocks on Intel Stratix 10, or Xilinx BRAM units). We consider a chip to contains N_b memory blocks, where each unit can store s_b elements of the target data type and has a read/write port width of w_b bits. The scheme is organized as follows (shown in Fig. 4.1):

1. A compute unit c consumes $\vec{r}_c$ hardware resources, and can output a result of a single multiplication and addition per cycle. Their maximal number

   $$N_{c,\max} \leq \min_{1 \leq i \leq t} \left(\frac{r_{i,\max}}{r_{i,c}} \right)$$

   for a given numerical precision is a hardware constant, given by the available resources $\vec{r}_{\max}$.

2. A processing element p encapsulates $x_c \cdot y_c$ compute units. Each processing element requires additional $\vec{r}_p$ resources for overhead logic.

3. A compute tile t encapsulates $x_p \cdot y_p$ processing elements. One compute tile contains all available compute units
   $x_c \cdot y_c \cdot x_p \cdot y_p = N_c$.

4. A block tile b encapsulates $x_t \cdot y_t = s_b$ compute tiles, filling the entire internal capacity s_b of currently allocated memory blocks in a cyclic pattern (§2.2.2).

5. A memory tile M encapsulates $x_b \cdot y_b = \left\lfloor \frac{N_b}{N_{b,\mathrm{min}}} \right\rfloor$ block tiles (discussed below), using all available N_b memory blocks.

Pseudocode showing the *iteration space* of this decomposition is shown in Lst. 21, consisting of 11 nested loops. Each loop is either a *sequential* `for`-loop, meaning that no iterations will overlap, and will thus correspond to *pipelined* loops in the HLS code; or a *parallel* `forall`-loop, meaning that every iteration is executed every cycle, corresponding to *unrolled* loops in the HLS code (§2.3.1 and §2.3.2). We require that the sequential loops are coalesced (§2.2.6) into a single pipeline, such that no overhead is paid at iterations of the outer loops.

4.3.4 Parallelism and Memory Resources

The available degree of parallelism, counted as a number of simultaneous computations of line 17 in Lst. 21, is determined by the number of compute units N_c. Every one of these compute units must read and write an element of $\boldsymbol{C}$ from fast memory *every cycle*. This implies a minimum number of *parallel* fast memory accesses that must be supported in the architecture. Memory blocks expose a limited access width w_b (measured in bits), which constrains how much data can be read from/written to them in a single cycle. We can thus infer a minimum number of memory blocks necessary to serve all compute units in parallel, given by:

$$N_{b,\mathrm{min}} = x_p y_p \cdot \left\lceil \frac{w_c \cdot x_c y_c}{w_b} \right\rceil, \tag{4.7}$$

where w_c is the width of the data type in bits, and $x_c y_c$ denotes the granularity of a processing element. Because all $x_c y_c$ accesses within a processing element happen in parallel, accesses to fast memory can be coalesced into long words of size $w_c \cdot x_c y_c$ bits. For cases where w_b is not a multiple of w_c, the ceiling in Eq. 4.7 may be significant for the resulting $N_{b,\mathrm{min}}$. When instantiating fast memory to implement the tiling strategy, Eq. 4.7 defines the minimum "step size" we can take when increasing the tile sizes.

Within a full memory tile, each updated value $C[i, j]$ is reused after all $x_{\mathrm{tot}} \cdot y_{\mathrm{tot}}$ elements in a single memory tile are evaluated (§2.2.2), and computation proceeds to the next iteration of the k-loop (line 4 in Lst. 21). Given the intrinsic size of each memory block s_b, we can thus perform s_b iterations of the compute tile before a single batch of $N_{b,\mathrm{min}}$ allocated memory blocks has been filled up. If the total number of memory blocks $N_{b,\mathrm{max}} \geq 2N_{b,\mathrm{min}}$,

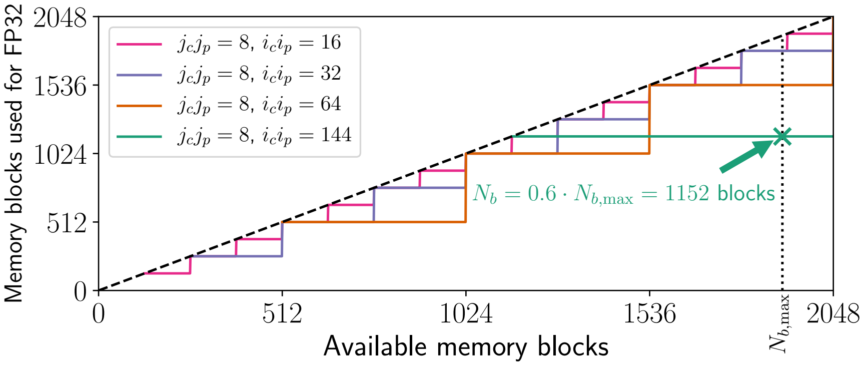


Figure 4.2: Utilization of memory blocks with memory tile size. For $i_cj_c = 8$ and $i_pj_p = 144$, we can utilize $60.4\% \cdot N_{b,\max}$.

i.e., the number of blocks required to support the parallel access requirements is less than the total number of blocks available, we can perform additional $\left\lfloor \frac{N_b}{N_{b,\min}} \right\rfloor$ iterations of the block tile, using all available memory blocks (up to the additive factor of $N_b \mod N_{b,\min}$). However, for large available parallelism N_c, this additive factor may play a significant role, resulting in a part of available on-chip memory not being used. This effect is depicted in Fig. 4.2 for different values of N_c for the case of single precision floating point (FP32) in Xilinx BRAM blocks, where $s_b = 1024$ and $w_b = 36\,\text{bit}$. The total number of memory blocks that can be efficiently used, without sacrificing the compute performance and load balancing constraints, is then:

$$N_b = \left\lfloor \frac{N_{b,max}}{N_{b,\min}} \right\rfloor N_{b,\min}. \tag{4.8}$$

In the worst case, this implies that only $N_{b,\max}/2 + 1$ memory blocks are used. In the best case, $N_{b,\max}$ is a multiple of $N_{b,\min}$, and all memory block resources can be utilized. When $N_c > N_{b,\max}/2$, the memory tile collapses to a single block tile, and the total memory block usage is equal to Eq. 4.7.

4.4 Hardware Mapping

With the goals for compute performance and I/O optimality set by the model, we now describe a mapping to a concrete hardware implementation.

4.4.1 Layout of Processing Elements

For Eq. 4.2 to hold, all N_p PEs must run at maximum throughput for the duration of the kernel execution, computing distinct contributions to the output tile. In terms of the proposed tiling scheme, we must evaluate a full compute tile t (second layer in Fig. 4.1) every cycle, which consists of $x_p \cdot y_p$ PE tiles (first layer in Fig. 4.1, via vertical unrolling of the buffered dimension §2.3.2), each performing $x_c \cdot y_c$ calculations in parallel (via horizontal unrolling §2.3.1), contributing a total of N_c multiplications and additions towards the outer product currently being computed. Assuming that N_p elements of $\boldsymbol{A}$ and a full row of y_{tot} elements of $\boldsymbol{B}$ have been prefetched, we must – for each of the x_p rows of the first layer in Fig. 4.1 – propagate x_c values to all y_p horizontal PEs, and equivalently for columns of $\boldsymbol{B}$. If this was broadcasted directly, it would lead to a total fan-out of $x_p \cdot y_p$ for both inputs.

Rather than broadcasting, we can exploit the regular grid structure, letting each column forward values of $\boldsymbol{A}$, and each row forward values of $\boldsymbol{B}$, in a pipelined fashion, implemented as a dataflow architecture (§2.3.3). Such an architecture is sometimes referred to as a *systolic array*, and is illustrated in Fig. 4.3. In this setup, each processing element has three inputs and three outputs (for $\boldsymbol{A}$, $\boldsymbol{B}$, and $\boldsymbol{C}$), and dedicated `Feed A` and `Feed B` modules send prefetched contributions to the outer product at the left and top edges of the grid, while `Store C` consumes the output values of $\boldsymbol{C}$ written back by the PEs. The number of inter-module connections for this design is $3x_py_p$, but more importantly, the fan-out of all modules is now constant, with 6 data buses per PE. Each PE is responsible for fully evaluating $x_{\text{tot}}y_{\text{tot}}/N_p$ elements of the output tile of $\boldsymbol{C}$. The elements of each PE tile in Fig. 4.1 are stored contiguously (the first layer), but all subsequent layers are not – only the compute tile as a whole is contiguous in $\boldsymbol{C}$. Final results must thus be written back in an interleaved manner to achieve contiguous writes back to $\boldsymbol{C}$.

4.4.1.1 Collapsing to a 1D array

Although the 2D array of PEs is intuitive for performing matrix multiplication, it requires a grid-like structure to be routed on the chip. While this solves the issue of individual fan-out – and may indeed be sufficient for monolithic devices with all logic arranged in a rectangular structure – we wish to map efficiently onto general interconnects, including non-uniform and hierarchical structures, as well as multiple-chiplet FPGAs (or, potentially, multiple FPGAs). To achieve this, we can optionally collapse the 2D array of PEs into a 1D array by fixing $y_p = 1$, resulting in $N_p = x_p$ PEs connected in sequence. Since this results in a long, narrow compute tile, we additionally fix $x_c = 1$, relying on y_c to regulate

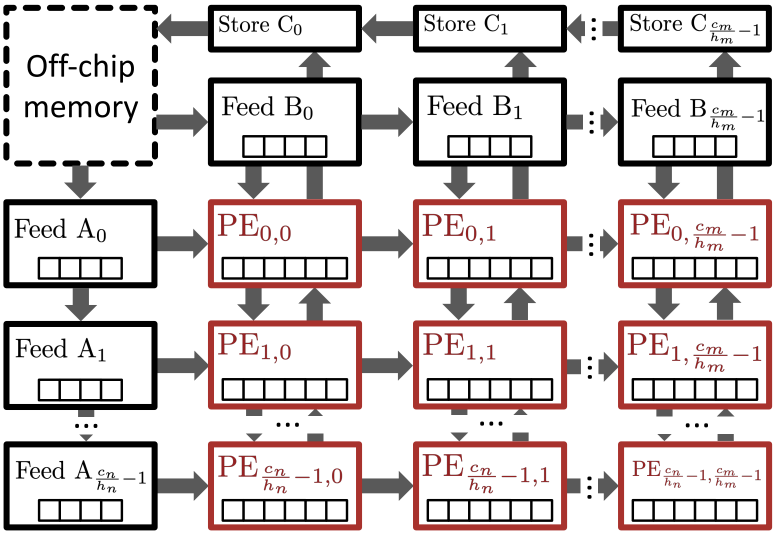


Figure 4.3: Compute arranged in a 2D grid.

the PE granularity. Forming narrow compute tiles is possible without violating Eq. 4.6, as long as x_{tot} and y_{tot} are kept identical (or as similar as possible), which we can achieve by regulating the outer block and tiling layers (the memory and block tile layers in Fig. 4.1).

4.4.1.2 Double buffering

Since each PE in the 1D array now computes one or more full rows of the compute tile, we can buffer values of $\boldsymbol{A}$ in internal registers, rather than from external modules. These can be propagated through the array from the first element to the last, then kept in local registers and applied to values of $\boldsymbol{B}$ that are streamed through the array from a buffer before the first PE. Since the number of PEs in the final design is large, we overlap the propagation of new values of $\boldsymbol{A}$ with the computation of the outer product contribution using the previous values of $\boldsymbol{A}$, by using *double buffering*, requiring two registers per PE, i.e., $2N_p$ total registers across the design.

By absorbing the buffering of $\boldsymbol{A}$ into the PEs, we have reduced the architecture to a simple chain of width 1, reducing the total number of inter-module connections for the compute to $3N_p$, with 3 buses connecting each PE transition. When crossing interconnects with long timing delays or limited width, such as connections between chiplets, this means that only 3 buses must cross the gap, instead of a number proportional to the circumference of the number of compute units within a single chiplet, as was the case for the 2D design. As a situational downside, this increases the number of pipeline stages in the architecture when

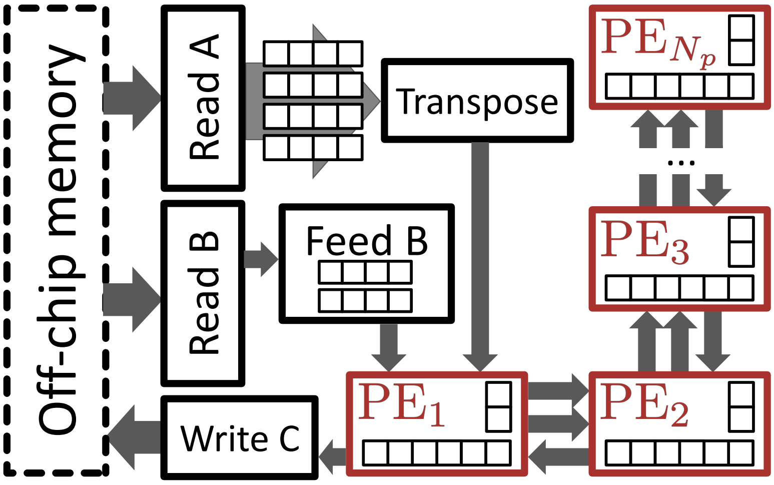


Figure 4.4: Module layout of final kernel architecture.

maximizing compute, which means that the number of compute tiles must be larger than the number of PEs, i.e., $y_t x_t \geq N_p$. This extra constraint is easily met when minimizing I/O, as the block tile size is set to a multiple of s_b (see Sec. 4.3.3), which in practice is higher than the number of PEs, assuming that extreme cases like $x_c = y_c = 1$ are avoided for large N_c.

4.4.2 Handling Loop-carried Dependencies

Floating point accumulation is often not a native operation on FPGAs, which can introduce loop-carried dependencies on the accumulation variable (see Sec. 2.2). This issue is circumvented with our decomposition. Each outer product consists of $x_p x_m \cdot y_p y_m$ inner memory tiles. Because each tile reduces into a distinct location in fast memory, collisions are separated by $x_p x_m \cdot y_p y_m$ cycles, and thus do not obstruct pipelining for practical memory tile sizes (i.e., where $x_p x_m \cdot y_p y_m$ is bigger than the accumulation latency).

For data types such as integers or fixed point numbers, or architectures that support (and benefit from) pipelined accumulation of floating point types, it is possible to make k the innermost loop, optionally tiling n and m further to improve efficiency of reads from off-chip memory. The hardware architecture for such a setup is largely the same as the architecture proposed here, but changes the memory access pattern.

4.4.3 Optimizing Column-wise Reads

In the outer product formulation, the $\boldsymbol{A}$-matrix must be read in a column-wise fashion. For memory stored as row-major, this results in slow and wasteful reads from DDR memory (in a column-major setting, the same argument applies, but for $\boldsymbol{B}$ instead). For DDR4 memory, a minimum of 512 bits must be transferred to make up for the I/O clock multiplier, and much longer bursts are required to saturate DDR bandwidth in practice. To make up for this, we can perform on-the-fly transposition of $\boldsymbol{A}$ as part of the hardware design in an additional module (an advanced form of memory buffering §2.4.2), by reading wide vectors and pushing them to separate FIFOs of depth $\geq x_b x_m$, which are popped in transposed order when sent to the kernel (this module can be omitted in the implementation at configuration time if $\boldsymbol{A}$ is pre-transposed, or an additional such module is added if $\boldsymbol{B}$ is passed in transposed form).

4.4.4 Writing Back Results

Each final tile of $\boldsymbol{C}$ is stored across the chain of processing in a way that requires interleaving of results from different PEs when writing it back to memory. Values are propagated backwards through the PEs, and are written back to memory at the head of the chain, ensuring that only the first PE must be close to the memory modules accessing off-chip memory. In previous work, double buffering is often employed for draining results, at the significant cost of reducing the available fast memory from S to $S/2$ in Eq. 4.5, resulting in a reduction in the arithmetic intensity of $\sqrt{2}$. To achieve optimal fast memory usage, we can leave writing out results as a sequential stage performed after computing each memory tile. It takes nm/y_c cycles to write back values of $\boldsymbol{C}$ throughout kernel execution, compared to nmk/N_c cycles taken to perform the compute. When $k/N_c \gg 1$, i.e., the matrix is large compared to the degree of parallelism, this effect of draining memory tiles becomes negligible.

4.4.5 Final Module Layout

With the constraints and considerations accumulated above, we fix the final hardware architecture. The module layout is shown in Fig. 4.4, and consists of $4 + N_p$ modules. The `Feed B` module buffers the outer product row of $\boldsymbol{B}$, whereas N_p values of $\boldsymbol{A}$ are kept in PE registers. The vast majority of fast memory is spent in buffering the output tile of C (see Sec. 4.3.2), which is partitioned across the PEs, with $\frac{x_{\text{tot}} \cdot y_{\text{tot}}}{N_p}$ elements stored in each. The `Read A` and `Transpose` modules are connected with a series of FIFOs, the

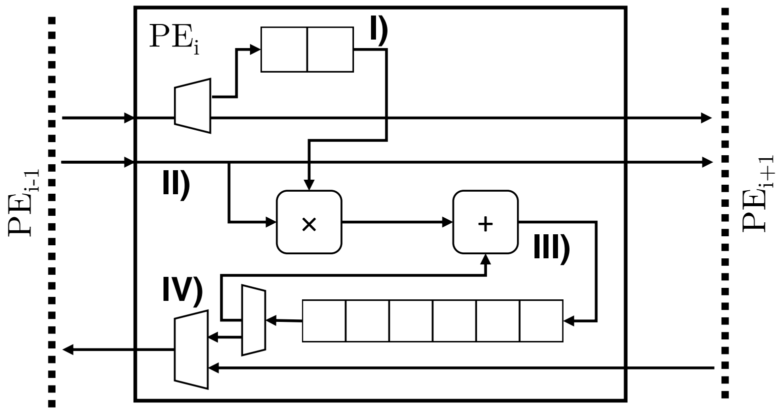


Figure 4.5: Architecture of a single PE.

number of which is determined by the desired memory efficiency in reading $\boldsymbol{A}$ from DRAM. In our provided implementation, PEs are connected in a 1D sequence, and can thus be routed across the FPGA in a "snake-like" fashion [98] to maximize resource utilization with minimum routing constraints introduced by the module interconnect.

The PE architecture is shown in Fig. 4.5. **I)** Each PE is responsible for storing a single double-buffered value of $\boldsymbol{A}$. Values are loaded from memory and passed through the array, while the previous outer product is being computed. **II)** Values of $\boldsymbol{B}$ are streamed through the chain to be used at every PE. **III)** Every cycle accumulates into a different address of the output $\boldsymbol{C}$ until it repeats after $x_t x_b \cdot y_t y_b$ cycles. **IV)** When the outer tile has been computed, it is sent back through the PEs and written back at the memory interface.

4.5 Evaluation

4.5.1 Parameter Selection

Using the performance model and hardware mapping considerations, parameters for kernel builds used to produce results are chosen in the following way, in order to maximize performance and minimize I/O based on available compute and memory resources, respectively:

1. The PE granularity is fixed at $x_c = 1$, and y_c is set as high as possible without impairing routing (determined empirically).
2. fN_c is maximized by scaling up parallelism $N_c = N_p \cdot y_c$ (we fixed $x_c = 1$) when the benefit is not eliminated by reduction in frequency, according to Eq. 4.2.

3. Memory tile sizes are maximized according to Eq. 4.8 to saturate on-chip memory resources.

For a given set of parameters, we build kernels in a *fully automated end-to-end fashion*, leveraging the abstractions provided by the high-level toolflow.

4.5.2 Code Complexity

The MMM kernel architecture used to produce the results in this chapter is implemented in Xilinx' Vivado HLS tool with hlslib (Chapter 3) extensions, and as of writing, consists of 620 and 291 SLOC of C++ kernel and header files, respectively. This is a generalized implementation, and includes variations to support transposed/non-transposed input matrices, variable/fixed matrix sizes, and different configurations of memory bus widths. Additionally, the operations performed by compute units can be specified, e.g., to compute the distance product by replacing multiply and add with add and minimum, respectively. The full source code is available on GitHub under an open source license (see page 57).

4.5.3 Experimental Setup

We evaluate our implementation on a Xilinx VCU1525 accelerator board, which hosts an Virtex UltraScale+ XCVU9P FPGA. The board has four DDR4 DIMMs, but due to the minimal amount of I/O required by our design, a single DIMM is sufficient to saturate the kernel. The chip is partitioned into three chiplets, that have a total of 1,033,608 LUTs, 2,174,048 *flip-flops* (FFs), 6834 DSPs, and 1906 BRAMs available to our kernels. This corresponds to 87%, 92%, 99.9%, and 90% of data sheet numbers, respectively, where the remaining space is occupied by the provided shell.

Our kernels are written in Vivado HLS targeting the `xilinx:vcu1525:dynamic:5.1` platform of the SDAccel 2018.2 framework (both SDAccel and Vivado HLS have been absorbed into the Vitis platform since these experiments were conducted), and `-O3` is used for compilation. We target 200 MHz in Vivado HLS and SDAccel, although this is often reduced by the tool in practice due to congestion in the routed design for large designs, in particular paths that cross between chiplets on the FPGA (see Sec. 4.2.3). Because of the high resource utilization, each kernel build takes between 8 and 24 hours to finish successfully, or between 4 and 24 hours to fail placement or routing.

On the Virtex UltraScale+ architecture, floating point operations are not supported natively, and must be implemented using a combination of DSPs and general purpose logic provided by the toolflow. The resource vector $\vec{r}$ thus has the dimensions LUTs, FFs, and DSPs. The Vivado HLS toolflow allows choosing from multiple floating point implementations, that provide different trade-offs between LUT/FF and DSP usage. In general, we found that choosing implementations of floating point addition that does not use DSPs yielded better results, as DSPs replace little general purpose logic for this operation, and are thus better spent on instantiating more multiplications.

Memory blocks are implemented in terms of BRAM, where each block has a maximum port width of 36 bit of simultaneous read and write access to 18 kbit of storage. For wider data types, multiple BRAMs are coalesced. Each BRAM can store $s_{b,36\,\mathrm{bit}} = 1024$ elements in 36 bit configuration (e.g., FP32), $s_{b,18\,\mathrm{bit}} = 2048$ elements in 18 bit configuration (e.g., FP16), and $s_{b,72\,\mathrm{bit}} = 512$ elements in 72 bit configuration (e.g., FP64). For this implementation, we do not consider UltraRAM, which is a different class of memory blocks on the UltraScale+ architecture, but note that these can be exploited with the same arguments as for BRAM (according to the principles in Sec. 4.3.3). For benchmarked kernels we report the compute and memory utilization in terms of the hardware constraints, with the primary bottleneck for I/O being BRAM, and the bottleneck for performance varying between LUTs and DSPs, depending on the data type.

4.5.4 Results

We evaluate the computational performance and communication behavior of our approach by constructing kernels within varying logic and storage budgets, based on our C++ reference implementation. To explore the scaling behavior with increased parallelism, we measure strong scaling when increasing the number of PEs, shown in Fig. 4.6, by increasing N_c for 16384×16384×16384 matrices. The toolchain does not support specifying a random seed for placement and routing, so the first compiled bitstream is used. We report the median across 20 runs, and omit confidence intervals, as all kernels behaved deterministically, making errors negligible. To measure power efficiency, we sample the direct current power draw of the PSU in the host machine, then determine the FPGA power consumption by computing the difference between the machine at idle with no FPGA plugged in, and the FPGA plugged in while running the kernel. This method includes power drawn by the full VCU1525 evaluation board, *including the integrated fan.* The kernels compile to maximum performance given by each configurations at 200 MHz until the first chiplet/SLR crossing, at which point the clock frequency starts degrading. This indicates that the chiplet crossings are the main contributor to long timing paths in

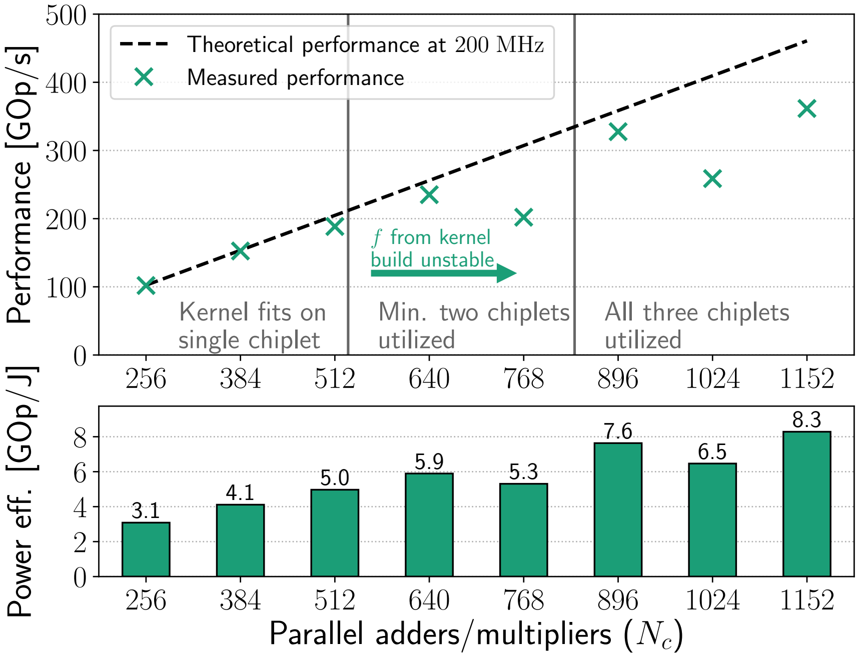


Figure 4.6: Strong scaling for single precision floating point, n=m=k=16384 matrices.

the design that bottleneck the frequency.

Tab. 4.2 shows the configuration parameters and measured results for the highest performing kernel built using our architecture for half, single and double precision floating point types, as well as 8-bit, 16-bit, and 32-bit unsigned integer types. Timing issues from placement and routing are the main bottleneck for all kernels, as the frequency for the final routed designs start to be unstable beyond 33% resource usage, when the number of chiplet crossings becomes significant (shown in Fig. 4.6). When resource usage exceeds 80−90%, kernels fail to route or meet timing entirely. Due to the large step size in BRAM

Type	x_p	y_c	x_{tot}	y_{tot}	**Freq.**	**Perf.**	**Pow. eff.**	**Arith. int.**	**LUTs**	**FFs**	**DSPs**	**BRAM**
FP16	112	16	1904	1920	171.3 MHz	606 $\frac{\text{GOp}}{\text{s}}$	15.1 $\frac{\text{GOp}}{\text{J}}$	956 $\frac{\text{Op}}{\text{Byte}}$	53%	24%	70%	90%
FP32	192	8	960	1632	145.7 MHz	409 $\frac{\text{GOp}}{\text{s}}$	10.9 $\frac{\text{GOp}}{\text{J}}$	302 $\frac{\text{Op}}{\text{Byte}}$	81%	46%	48%	80%
FP64	96	4	864	864	181.2 MHz	132 $\frac{\text{GOp}}{\text{s}}$	3.13 $\frac{\text{GOp}}{\text{J}}$	108 $\frac{\text{Op}}{\text{Byte}}$	38%	28%	80%	82%
uint8	132	32	1980	2176	186.5 MHz	1544 $\frac{\text{GOp}}{\text{s}}$	48.0 $\frac{\text{GOp}}{\text{J}}$	2073 $\frac{\text{Op}}{\text{Byte}}$	15%	8%	83%	51%
uint16	210	16	1680	2048	190.0 MHz	1217 $\frac{\text{GOp}}{\text{s}}$	33.1 $\frac{\text{GOp}}{\text{J}}$	923 $\frac{\text{Op}}{\text{Byte}}$	20%	11%	69%	88%
uint32	202	8	1212	1360	160.6 MHz	505 $\frac{\text{GOp}}{\text{s}}$	13.8 $\frac{\text{GOp}}{\text{J}}$	320 $\frac{\text{Op}}{\text{Byte}}$	58%	11%	84%	86%

Table 4.2: Highest performing kernels built for each data type.

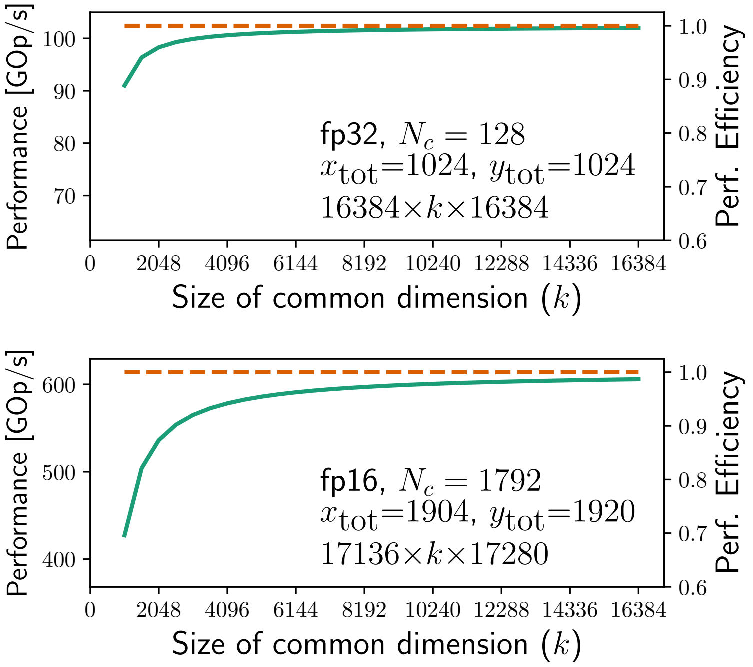


Figure 4.7: Fraction of maximum compute throughput for varying matrix size.

consumption for large compute tiles when targeting peak performance (see Sec. 4.3.3), some kernels consume less BRAM than what would otherwise be feasible to route, as increasing the memory tile by another stage of $N_{b,\text{min}}$ would exceed $N_{b,\text{max}}$.

In contrast to previous implementations, we achieve optimal usage of the on-chip memory by separating the drain phase of writing out results from the compute phase. This requires the number of computations performed per memory tile to be significantly larger than the number of cycles taken to write the tile out to memory (see Sec. 4.4.4). This effect is shown in Fig. 4.7 for small N_c (left) and large N_c (right). For large N_c, the time spent in draining the result is significant for small matrices. In either scenario, optimal computational efficiency is approached for large matrices, when there is sufficient work to do between draining each result tile.

Fig. 4.8 demonstrates the reduction in communication volume with increasing values of the outer I/O tiles (i.e., $x_t x_b \cdot y_t y_b$). We plot the arithmetic intensity, corresponding to 2× the computational intensity in Eq. 4.3 (1 addition and 1 multiplication), and verify that the communication volume reported by the runtime is verified to match the analytical value

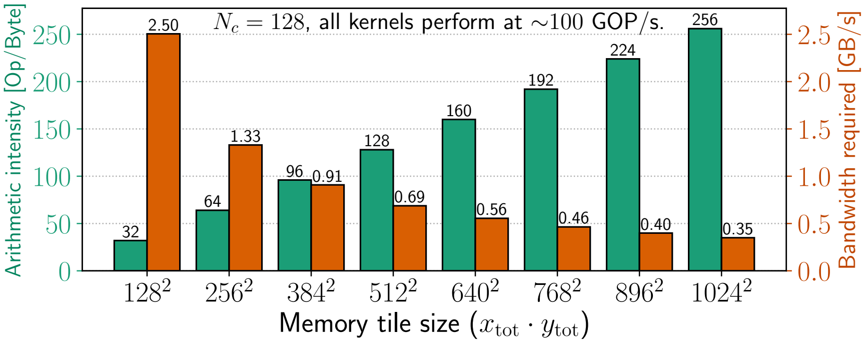


Figure 4.8: FP32 arithmetic intensity with memory tile size.

computed with Eq. 4.5. We also report the average bandwidth requirement needed to run each kernel (in practice, the bandwidth consumption is not constant during runtime, as memory accesses are done as bursts each time the row and column for a new outer product is loaded). There is a slight performance benefit from increasing memory tile size, as larger tiles increase the ratio of cycles spent in the compute phase to cycles spent writing back results, approaching perfect compute/DSP efficiency for large matrices. For the largest tile size, the kernel consumes 350 MByte/s at 100 GOp/s, which corresponds to $\frac{350}{19200} = 1.8\%$ of the maximum bandwidth of a single DDR4 module. Even at the highest measured single precision performance (Tab. 4.2) of 409 GOp/s, the kernel requires 1.35 GByte/s. This brings the I/O of matrix multiplication down to a level where nearly the full bandwidth is left available.

4.6 Related Work

Much of previous work focuses on the low level implementation for performance [82], explores high-level optimizations [47], or implements MMM in the context of neural networks [61, 109]. To the best of our knowledge, this is the first matrix multiplication accelerator to minimize I/O on FPGA *in terms of hardware constants*, and the first open source implementation to benefit of the community. We relate our work to the most relevant works below.

Tab. 4.3 shows a hybrid qualitative/quantitative comparison to previously published MMM implementations on FPGA. Cells are left empty when numbers are not reported by the authors, or when the given operation is not supported. Most previous work does not

	Year	Device	% Logic util.	Freq. [MHz]	Perf. FP16 $[\frac{\text{GOp}}{\text{s}}]$	Perf. FP32 $[\frac{\text{GOp}}{\text{s}}]$	Perf. FP64 $[\frac{\text{GOp}}{\text{s}}]$	Energy eff. FP32 $[\frac{\text{GOp}}{\text{J}}]$	Multiple data types	Lang. (Portable)	Open source	I/O model
Zhuo [161]	2004	Virtex-II Pro	98	128	-	2	2	-	👎	HDL (👎)	👎	👎
Dou [49]	2005	Virtex-II Pro	99	177	-	-	39	-	👎	HDL (👎)	👎	👎
Kumar [88]	2009	Virtex-5	61	$373^{\dagger}$	-	-	$30^{\dagger}$	-	👎	HDL (👎)	👎	👍
Jovanović [82]	2012	Virtex-6	100	403	-	203	-	-	👎	HDL (👎)	👎	👎
D'Hollander [47]	2016	Zynq-7000	99	100	-	5	-	-	👎	HLS (👍)	👎	👎
Guan [61]	2017	Stratix V	95	150	-	100	-	2.92	👍	HDL/HLS (👎)	👎	👎
Moss [109]	2018	HARPv2	99	313	-	800	-	22.0	👍	HDL (👎)	👎	👎
Ours	2019	VCU1525	69–90	146–190	606	409	122	10.9	👍	HLS (👍)	👍	👍
Vitis BLAS [155]	2021	U250	$15^{\dagger}$	242	-	346	-	-	👍	HLS (👍)	👍	👎

Table 4.3: Comparison to previous FPGA implementations. *Simulation only. † Utilization reported by the authors appears to be incompatible with the reported performance.

publish their code, and we resort to report the performance given by the authors and the respective FPGA that it was executed on. The closest comparison can be drawn to the implementation in Xilinx' own Vitis BLAS library, which is reported to achieve 85% of the performance of our 32-bit floating point kernel, on an FPGA that has 167% the number of LUTs and 180% the number of DSPs relative to the VCU1525 board used for our experiments.

Zhuo and Prasanna [161] discuss two matrix multiplication implementations on FPGA, include routing in their considerations, and support multiple floating point precisions. The authors suggest two algorithms, where both require a number of PEs proportional to the matrix size. While these only require loading each matrix once, they do not support matrices of arbitrary size, and thus do not scale without additional CPU orchestration.

Dou et al. [49] design a linear array of processing elements, implementing 64-bit floating point matrix multiplication – no support is offered for other data types, as the work emphasizes the low-level implementation of the floating point units. The authors derive the required off-chip bandwidth and buffer space required to achieve peak performance on the target device, but do not model or optimize I/O in terms of their buffer space usage, and do not report their tile sizes or how they were chosen. Furthermore, the authors double-buffer the output tile, reducing the maximum achievable computational intensity by a factor $\sqrt{2}$ (see Sec. 4.4.4).

A customizable matrix multiplication implementation for deep neural network applications on the Intel HARPv2 hybrid CPU/FPGA platform is presented by Moss et al. [109], targeting single precision floating point (FP32), and fixed point/integer types. The authors exploit native floating point DSPs on an Arria 10 device to perform accumulation, and do not consider data types that cannot be natively accumulated on their chip, such as half or double precision. The I/O characteristics of the approach is not reported quantitatively. Wu et al. [153] present a highly specialized architecture for maximizing DSP usage and frequency of 16 bit integer matrix multiplication for DNN acceleration on two Xilinx UltraScale chips, showing how peak DSP utilization and frequency can be reached, at the expense of generality, as the approach relies on low-level details of the chips' architecture, and as no other data types are supported.

Kumar et al. [88] provide an analysis of the trade-off between I/O bandwidth and on-chip memory for their implementation of 64-bit matrix multiplication. The authors arrive at a square output tile when deriving the constraints for overlapping I/O, although the derived computational intensity is reduced by a factor $\sqrt{2}$ as above from double buffering. In our model, the fast memory utilization is captured explicitly, and is maximized in terms of on-chip memory characteristics of the target FPGA, allowing tile sizes that optimize both computational performance and computational intensity to be derived directly. Lin and Leong [102] model sparse MMM, with dense MMM as a special case, and project that even dense matrix multiplication may become I/O bound in future FPGA generations. Our model guides how to maximize utilization in terms of available on-chip memory to mitigate this, by capturing their characteristics in the tiling hierarchy.

Finally, these works were implemented in hardware description languages, and *do not disclose the source code allowing their findings to be reproduced or ported to other FPGAs.* For the results presented here, we provide a high-level open source implementation, to encourage reusability and portability of FPGA codes.

Designing I/O minimizing algorithms has been an active field of research for more than 40 years. Starting with register allocation problems [25], through single processor, two-level memory system [80], distributed systems with fixed [75] and variable memory size [134]. Although most of the work focuses on linear algebra [134, 75, 56] due to its regular access pattern and powerful techniques like polyhedral modeling, the implication of these optimizations far exceeds this domain. Gholami et al. [57] studied model and data parallelism of DNN in the context of minimizing I/O for matrix multiplication routines. Demmel and Dinh [43] analyzed I/O optimal tiling strategies for convolutional layers of NN.

4.7 Summary

In this chapter we presented a high-performance, open source, flexible, portable, and scalable matrix-matrix multiplication implementation on FPGA, which simultaneously maximizes performance and minimizes off-chip data movement. By starting from a general model for computation, I/O, and resource consumption, we create a hardware architecture using the transformations described in Chapter 2 that is optimized to the resources available on a target device, and is thus not tied to specific hardware. We evaluate our implementation on a wide variety of data types and configurations, showing 409 GOp/s 32-bit floating point performance, and 1.5 TOp/s 8-bit integer performance, utilizing >80% of hardware resources. We show that our model-driven I/O optimal design is robust and high-performant in practice, yielding better or comparable performance to HDL-based implementations, and conserving bandwidth to off-chip memory, while being easy to configure, maintain and modify through the high-level HLS source code.

Since being published on GitHub, the open source repository (link on page 57) has received significant interest: as of writing, the repository has 168 stars, 23 forks, and has inspired or been used as a component of several other FPGA projects.

Chapter 5

Data-Centric Design of Spatial Architectures

The DaCe framework[1] is a long-running, massively collaborative effort, with contributions from many researchers and students. I co-wrote the core implementation of DaCe and co-authored the original paper on the SDFG representation, which was led by Tal Ben-Nun and published at SC'19 [16]. The SDFG representation was conceived by a team of researchers in our lab consisting of Torsten Hoefler, Tal Ben-Nun, Timo Schneider, Alexandros Nikolaos Ziogas and myself. For a full list of contributors to the DaCe source code, I refer to the continuously updated author list[2] hosted in the GitHub repository. I designed, developed, and now maintain the FPGA backend of DaCe, which has received numerous contributions since its inception, in particular from Tiziano De Matteis, who is now co-maintaining this aspect of the framework, and conducts our Intel FPGA experiments. I also designed and implemented the concept of Library Nodes that enables multi-level design in DaCe. FPGA results are included from Ziogas et al. [164], where the Xilinx benchmarks were collected by me, and the Intel FPGA benchmarks were collected by Tiziano De Matteis, using the full DaCe stack with contributions from all authors.

5.1 Data-Centric Parallel Programming

Long ago, we passed the threshold where the full spectrum of optimization techniques required to effectively target modern hardware can be known by any single person working in the field of HPC, let alone a scientist who writes code to solve a problem within their domain of natural sciences, but has no formal education in HPC. Scientists find themselves juggling everything from single-core optimizations – like tiling for caches and vectorization – through multi-core optimizations, implementing shared-memory parallelism via technologies like OpenMP – through targeting multi-node systems ranging from small clusters to supercomputers with MPI – to exploiting accelerators using techniques like CUDA, OpenMP 4.0, OpenACC, OpenCL, or HLS. It is clear that this kind of development is

---

[1] https://github.com/spcl/dace

[2] https://github.com/spcl/dace/blob/master/AUTHORS

no longer a single person job, and will more often than not require people from multiple different backgrounds to carry it out successfully.

Once the scientific software has been optimized, the next issue arises. After tiling, vectorizing, parallelizing, distributing, and/or accelerating the application, the code is unrecognizable. Scattered with communication primitives, strip-mined and reordered loops, pragmas, annotations, kernel launches, accelerator kernels that are often written in a different language/technology, and/or thread synchronization primitives, the code has effectively become unmaintainable – including by the authors, but especially by any new engineer joining the project, because the optimizations are so invasive to the source code.

We propose the *Data-Centric parallel programming* (DaCe) framework as a solution to these issues. DaCe is built on the fundamental concept of *separation of concerns*. We view the development process of an HPC application as a collaboration between at least two roles: the *domain scientist*, who is an expert of the scientific domain in which the application is trying to solve a program; and the *performance engineer*, who is an expert in performance optimization. Although the roles do not necessarily need to refer to different people (one person can be an expert in both), they concern distinct aspects of the development process, and should interfere with each other as little as possible.

To enable separation of concerns, DaCe employs the *Stateful DataFlow multiGraph (SDFG)* [16] intermediate representation. SDFGs isolate performance optimization in a data-centric graph-based representation, where all data movement is explicit, thereby becoming analyzable and malleable by a skilled performance engineer. Promising work has been done on auto-optimization using SDFGs [164], but the core philosophy is to enable the performance engineer to perform *guided* optimization in a productive manner, using transformations that employ graph-rewriting to reorganize the data movement of the program. The domain scientist uses one or more frontends, such as NumPy or PyTorch, to productively express the application that they wish to evaluate, which are then emitted as SDFGs, ripe for optimization. Because optimizing the SDFG does not touch the input program, it does not interfere with the maintainability of the scientific code, as optimizing transformations can merely be re-applied if changes are made to the input.

Why data-centric? Since computing hit the memory wall [154] (as described in Sec. 1.1), programming HPC applications has been almost exclusively about *data movement*. Whether it be between memory and processor, between cache and processor, between processor cores, between nodes, or between host and accelerator, most HPC optimizations strive to improve temporal and spatial locality [141]. SDFGs exploit this by exposing all this data movement directly in the representation, explicitly annotating all data movement for the performance engineer to manipulate.

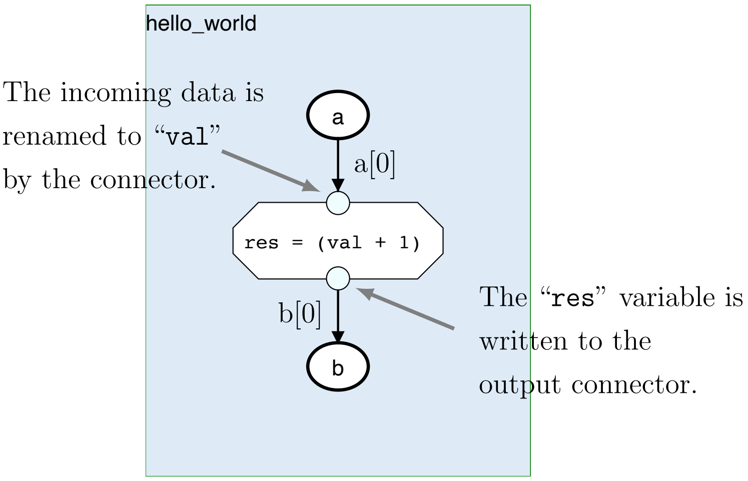
Modern compiler techniques, such as polyhedral optimization, also allow sophisticated automatic optimization of low-level IR by detecting and transforming loop constructs [22], but are restricted to the space of transformations that can be proven to be “safe”. Rather than relying on fully automated optimization to exploit all available opportunities, DaCe exposes powerful performance analysis capabilities and optimization tools that enable knowledgeable performance engineers to perform *guided* optimization of programs. Transformations are done via graph rewriting on the SDFG representation, expressed in terms of the general dataflow and control flow of the program, thus facilitating knowledge exchange between programs and domains.

When targeting a specific application domain, domain-specific languages (DSLs) can enable additional optimizations by restricting the input domain, allowing additional assumptions to be made on the program’s behavior, but typically offer limited exchange of knowledge and engineering effort with other domains. To this end, the “Library Node” extension to SDFGs enables a multi-level design methodology that exposes the best of both worlds within the same framework, enabling the application of both domain-specific and general purpose optimizations to SDFGs, described in Sec. 5.5.

Since their inception, SDFGs have been proven effective in various domains, ranging from linear algebra kernels and graph algorithms [16], through large scale machine learning [76] and numerical weather prediction (Chapter 6), to supercomputer-scale quantum transport simulations [162]. When SDFGs are manually authored or generated from specialized domain-specific languages, their performance is on a par with or outperforms state-of-the-art implementations and libraries. In the following, after giving an overview of the SDFG represention and its benefits in a general setting, we will focus on employing SDFGs and the DaCe framework as a next generation environment for developing FPGA programs.

5.2 A Primer on SDFGs

In this section, we give an overview of the core concepts and semantics of the graph-based SDFG representation, using pictures from the graphical user interface for illustration. We will describe the semantics of edges and different types of nodes in the graph in a general setting, before covering the opportunities for optimizations that this unlocks in Sec. 5.3. The background information in this section describes work done by the DaCe collaboration



The incoming data is renamed to "`val`" by the connector.

The "`res`" variable is written to the output connector.

Figure 5.1: A trivial SDFG that reads, modifies, and writes a single data element.

as a whole.

In the following we will include some simple examples of SDFGs. From the domain scientist's point of view, the SDFG can be thought of as a program, which can be invoked from a Python or C++ code as any other function with input and output arguments. Arguments are typically one or more NumPy arrays (in the case of Python) or pointers to data (in the case of C++), along with any scalar variables required by the program, such as array sizes. From the performance engineer's point of view, the host program can be ignored, and all attention focused on the SDFG itself, which appears as a Python object that can be stored and loaded as `.sdfg` files. The SDFG can be manipulated via built-in or self-defined transformations that rewrite the graph, by directly manipulating objects on the graph programmatically, or interactively using a graphical user interface.

Fig. 5.1 shows the simplest possible SDFG, where a single element from `a` is modified and written to `b`. We will describe each of the primitives used in the following.

5.2.1 Tasklets

Tasklets represent the *smallest granularity of computation*, and typically operate on one or more operands a single point in the iteration space. Because all data movement in the program must be explicit, Tasklets can only access data that is explicitly connected to them via dataflow edges. These edges are connected to ports on the Tasklet called "connectors", which gives the loaded or to-be-stored data a variable name for use in the code that described the computation. The actual computation is described by source code operating on the input/output data, which as of writing can be provided in either Python or C++. Tasklets are considered black boxes, and do not need to be analyzable, as long as they provide code that successfully compiles.

The SDFG in Fig. 5.1 contains a single Tasklet called "`hello_world`", which adds one to the input and writes the result to the output. It has one input connector on the top named "`val`", which exposes the incoming data from the `a` to the variable used in the computation as the variable `val`, and one output connector on the bottom called "`res`", which writes the variable `res` in the computation out to `b`.

5.2.2 Access Nodes

Access nodes (or data nodes) point to data containers that have been defined on the SDFG. When connected with dataflow edges, they indicate which data container is being accessed, while the edge indicates which *subset* of the container is being accessed. Data containers can be arrays, scalars, streams (FIFO queues, which we will use extensively for FPGA programs), arrays of streams, or views into arrays. Containers are either "transient"/local (i.e., temporary), meaning that they only exist in the scope in which they are used, or non-transient/global, meaning that they are persistent outside the current scope (e.g., because they refer to NumPy arrays in a calling program).

The SDFG in Fig. 5.1 contains two access nodes. `a` is an input to the program, and `b` is an output from the program, so they are both global containers. `a` is accessed read-only, while `b` is accessed write-only.

5.2.3 Dataflow Edges

Edges between nodes in Dataflow States represent data movement, and are annotated with the subset of data being accessed. When connecting two access nodes, Dataflow Edges imply a memory copy, and a subset must be specified for both the source and destination. In this case, it must be possible to copy between the source and destination location (e.g., we can copy between CPU heap memory and FPGA global memory, but not between FPGA local memory and GPU local memory). When connecting an access node to a connector on a Tasklet, Dataflow Edges represent making the data *available* (i.e., not necessarily copying the data unless it is actually used) to read or write to/from the computational primitive (see Sec. 5.2.1).

The example in Fig. 5.1 contains two Dataflow Edges: one between `a` and the Tasklet, and

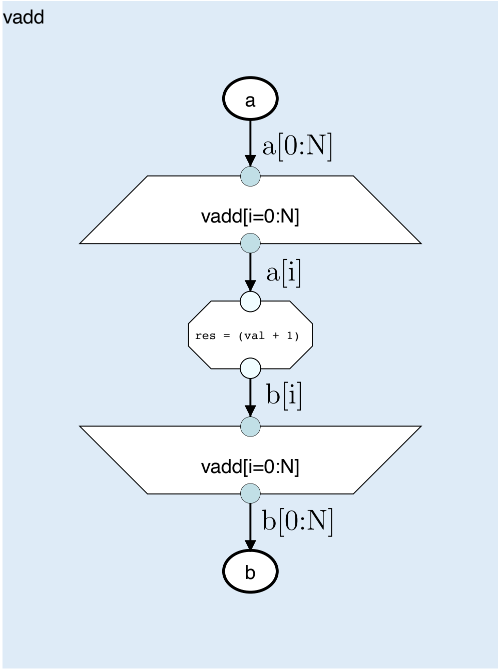


Figure 5.2: A simple vector addition example, using the map primitive.

one between the Tasklet and `b`. Both are annotated with a trivial subset of "0", as they just read and write a single element, respectively.

To support reduction-like patterns within Dataflow States, dataflow edges also support "conflict resolution", where writes than can potentially race with each other can resolved atomically, using various techniques to achieve atomicity depending on the storage location of the destination memory. These edges must specify how the resolution is done, by providing a lambda function that takes the existing and proposed new value as inputs, and outputs the value that should be written.

5.2.4 Maps

To do anything useful with SDFGs, we need to express parametric parallelism. A marginally more interesting SDFG is shown in Fig. 5.2, where an arithmetic operation is applied across all N elements of an array `a` and written to the array `b`.

Maps are the engine of parallelism in SDFGs. They appear as a pair of nodes in a Dataflow State: the first node "opens" a parallel scope (called a "Map Entry"), and the second node "closes" it (called a "Map Exit"). The Map is annotated with an iteration space of arbitrary dimensions. Any subgraph that appears between the entry and exit nodes are semantically "repeated" for every iteration in this iteration space. This can map to any kind of parallelism depending on the *schedule* defined on the Map, including vectorization, multithreading, pipelining, unrolling, GPU threads etc., or even just sequential for-loops. There is no restriction on what can appear within a Map scope, including other Maps, or even complete SDFGs with their own nested control flow and dataflow. When connected to/from a Map Entry/Exit node, the Dataflow Edges forms a "path" across the scope node, where the outer edge indicates the union of all accesses carried out by the inner edge across the iteration space of the Map.

The Map scope in Fig. 5.2 can be used to target any kind of parallelism, but will by default parallelize the computation using OpenMP multithreading.

5.2.5 Dataflow States

Because we are primarily interested in optimizing data movement, the core environment of SDFGs are pure Dataflow *States*. States are graphs, where nodes represent data accesses, schedules, and computations, and edges between nodes represent data movement. Multiple accesses to the same data container can coexist within the same Dataflow State, but will ignore any read-after-write, write-after-write, or write-after-read dependencies (these are handled via coarse-grained control flow or fine-grained conflict resolution, described in the context of control flow edges and dataflow edges, respectively). Data containers have their storage location annotated on them (e.g., CPU heap, FPGA global memory, GPU local memory), which will determine how data is moved across edges connected to them.

States are visualized as the light blue rectangles surrounding the dataflow nodes in Fig. 5.1 and Fig. 5.2. These examples only include a single Dataflow State, but States are themselves embedded as nodes in a control flow graph formed by the containing SDFG, which can again be embedded within Dataflow States, forming a nested control flow/dataflow representation. Fig. 5.3 shows an example SDFG with multiple Dataflow States connected by coarse-grained control flow between them. The Dataflow State `init` is used to initialize the array `tmp`, before entering into a sequential loop, represented by Control Flow Edges in the SDFG, described below.

5.2.6 Control Flow Edges

When an aspect of a program cannot be expressed by pure dataflow, we must resort to control flow. As a general philosophy, coarse-grained control flow is only used or generated

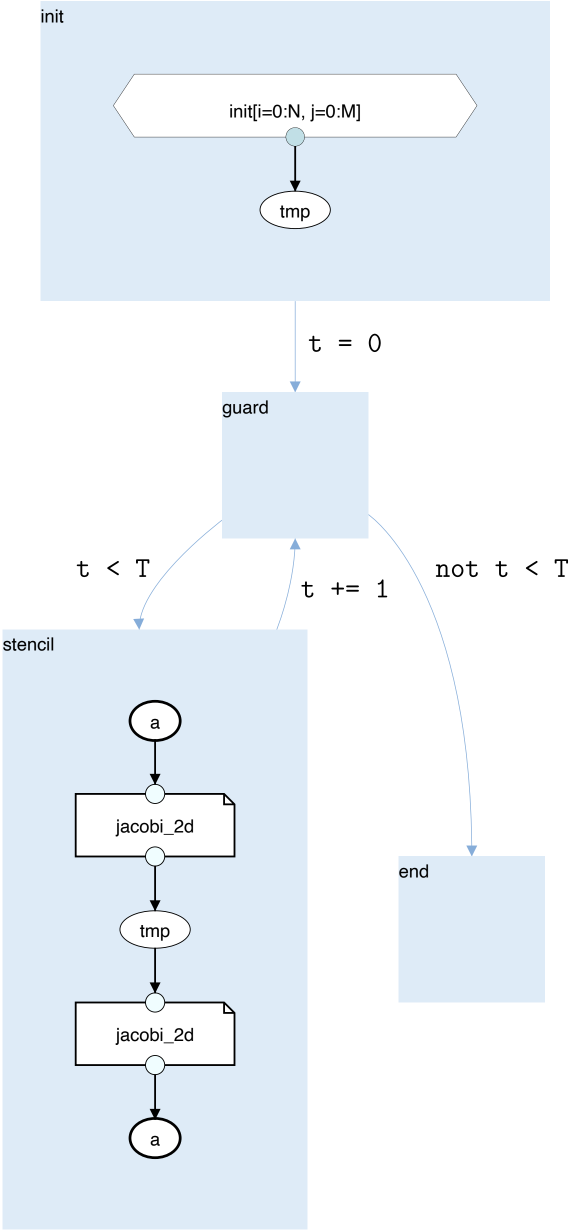


Figure 5.3: An SDFG containing multiple dataflow states with coarse-grained control flow.

in DaCe when strictly required by the program semantics. In the SDFG, Dataflow States are connected to each other via control flow edges (called "interstate edges" in the code base). If a State has multiple outgoing edges, the edges must be annotated with (mutually exclusive) conditions that determines which State will be executed next. If a State only has a single outgoing edge, no condition will be annotated on the edge. Control flow between States are typically used to express patterns such as iterating until a certain condition is met, or to resolve read-after-write, write-after-write, or write-after-read dependencies on data containers that appear multiple times. To help implement conditional edges and iterative patterns, control flow edges can also define and modify scalar variables, which are exposed to (but are immutable by) the dataflow within States.

The SDFG in Fig. 5.3 contains four states, connected by control flow edges. As the only state with no incoming edges, the `init` State will be executed first, where it initializes the temporary array `tmp` with zeros. An outgoing edge leads to the State `guard`, which acts as a loop guard for the loop body defined by the State `stencil`. On the edge between `init` and `guard`, the symbol "`t`" is defined and set to zero. This checked on the outgoing edges of `guard`, transitioning either to the loop body state, or the `end` state. On the back-edge from the body State to `guard`, the loop iterator `t` is incremented.

5.2.7 Library Nodes

Finally, appearing as another Tasklet-like computational node in Dataflow States, similar to nested SDFGs, are *Library Nodes*. Library Nodes are configurable meta-nodes describing *what* should be computed, as opposed to *how* it should be computed, thereby representing abstract behavior in the graph. Fig. 5.3 shows two library nodes, drawn as a rectangle with a "folded" corner, representing the Jacobi 2D stencil operator being applied to the input. Before compiling, these nodes will be "expanded" into lower-level primitives implementing the operation. Library Nodes enable a multi-level design approach to kernel development, and will be covered in more detail in Sec. 5.5.

5.2.8 Nested SDFGs

Dataflow States are nested within a control flow graph in a single SDFG. To also allow nesting control flow within dataflow, we support nesting SDFGs within Dataflow States. Such a nested SDFGs appears in the same way as a Tasklet: as a single node with input and output connectors that are linked to data edges in the Dataflow State that it is contained in, such that it can only access data from the parent State through these connectors.

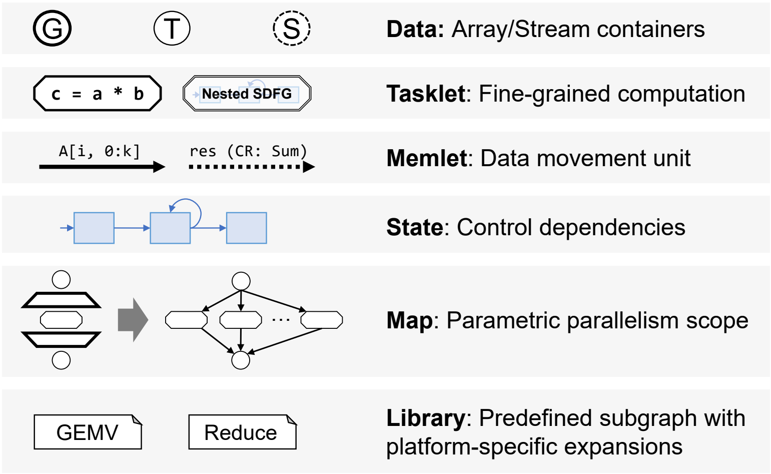

G
T
S
Data: Array/Stream containers
c = a * b
Nested SDFG
Tasklet: Fine-grained computation
A[i, 0:k]
res (CR: Sum)
Memlet: Data movement unit
State: Control dependencies
...
Map: Parametric parallelism scope
GEMV
Reduce
Library: Predefined subgraph with platform-specific expansions

Figure 5.4: Glossary of nodes and edges in the SDFG representation.

An example of a nested SDFG appearing in a Dataflow State in shown in Fig. 5.5, showing a subsection of the SDFG from Fig. 5.3 after it has been expanded to implement the stencil operator with a method that targets shift registers on Intel FPGAs. The outer state `stencil` contains a parallel Map scope, where each iteration executes the nested SDFG that contains three sequential states to shift, update, and compute at the given index.

A glossary of all graph primitives included in the SDFG representation is shown in Fig. 5.4. Combining all these primitives above, SDFGs can represent arbitrary programs in a data-centric format. In the following, we will cover the advantages that this brings for performance optimization.

5.3 Transforming the SDFG

The transformation methodology of DaCe was conceived by all authors of the SDFG paper [16], and the engine was implemented by Tal Ben-Nun, myself, and others. Automatic inference of data volumes along dataflow paths was implemented by Tal Ben-Nun, Alexandros Ziogas, and Philipp Schaad. This section contains multiple transformation developed by other DaCe authors, explicitly attributed in each subsection.

Because all data movement is explicitly annotated on the dataflow edges in the graph, we can directly read the data volume being moved between subgraphs from the representation.

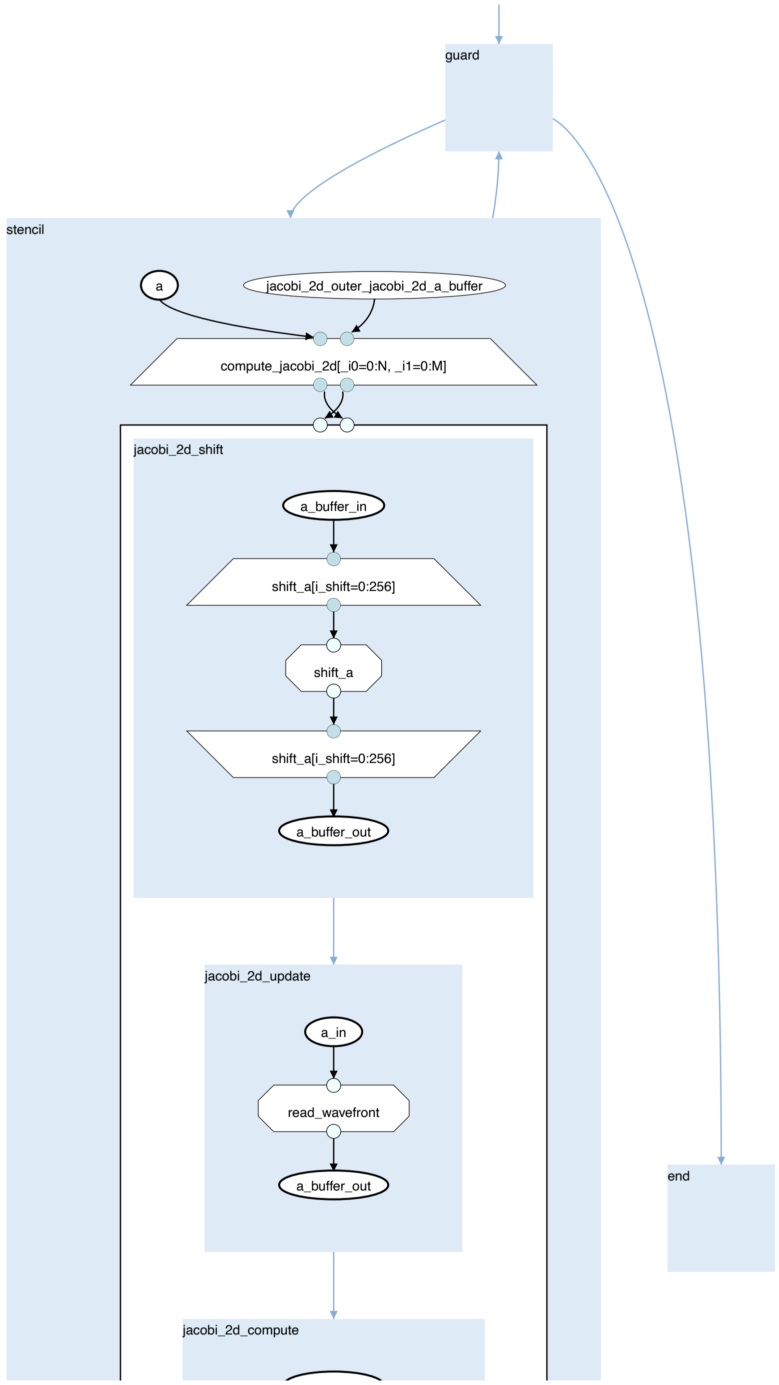


Figure 5.5: Coarse grained control flow nested in dataflow using a nested SDFG.

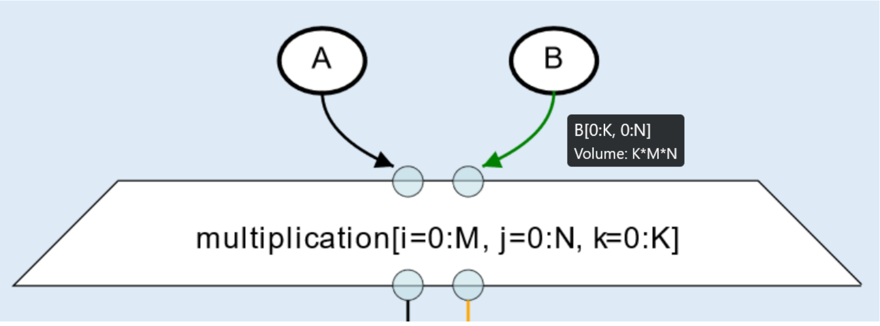


Figure 5.6: Data movement is captured and annotated on dataflow edges.

An example is shown in Fig. 5.6, where each iteration of a three-dimensional Map reads a single value from the array `B`, which will be automatically annotated on the outer edge as having a data movement volume of $M{\cdot}N{\cdot}K$, based on the iteration space of the Map.

Data volumes are automatically computed based on the accesses performed by computations and the parallel scopes that they exist in, and do thus not need to be provided by the engineer implementing the application. When the number of accesses cannot be determined exactly due to runtime-dependent accesses in the code, DaCe will state that the access is dynamic and give an upper bound.

Understanding the data movement of the program lets the performance engineer easily identify bottlenecks to make informed decisions on where optimizations should be applied. For the example in Fig. 5.6, the performance engineer could perform tiling on the parallel Map scope and/or insert local buffers, progressively using the data volume annotations to guide tile and buffer sizes to match the hardware being targeted (e.g., number of registers, or size of L1, L2, or L3 caches).

The primary engine for optimizing programs using the SDFG representation are *transformations*, which are graph-rewriting rules that manipulate a subgraph of a Dataflow State, a subgraph of the SDFG (in the control flow setting), or a combination of both. These transformations can read and mutate both the structure of the graph itself and the annotations on the nodes and edges, manipulating the dataflow or control flow, for example by rearranging schedules, inserting buffers, changing where compuations are executed, or changing where data is stored.

Transformations exist as classes in the DaCe framework, which can either be applied as is, or via a central transformation engine that will scan the nested graphs for opportunities to apply them. To determine whether a transformation can be applied, each transformation

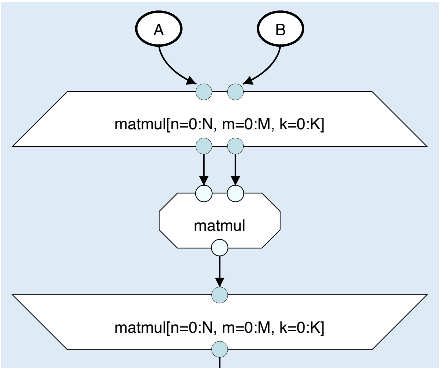


Figure 5.7: A simple three-dimensional map.

class must implement a method `can_be_applied`, which takes a matched subgraph, and inspects any number of nodes, edges, or general properties to determine whether the transformation would be valid. To take effect, the class must implement a method `apply`, which based on a matched subgraph will manipulate the representation in the desired way, by adding or removing edges and nodes, or changing properties on edges and nodes.

As of writing, the base repository includes 71 transformations (not counting specialized transformations that are part of domain-specific extensions to DaCe, such as StencilFlow (Chapter 6) or the DaCeML [1] repository for PyTorch and ONNX support). Below, we give four examples of general purpose transformations that can be used to optimize SDFGs.

5.3.1 Example: Vectorization

The "`Vectorization`" transformation was implemented by Tal Ben-Nun.

Vectorization is one of the most common optimizations applied in HPC, used to increase spatial locality and to exploit vector units to increase computational throughput. In procedural code, vectorization is typically done by relying on an auto-vectorizing compiler, by using parallel loops, or by using explicit vector types.

Fig. 5.7 shows a simple three-dimensional Map scope computing products between arrays `A` and `B` as part of a matrix-matrix multiplication implementation. To vectorize this computation using the standard `Vectorization` transformation part of the DaCe toolbox, we can use the following lines of code on an `sdfg` object:

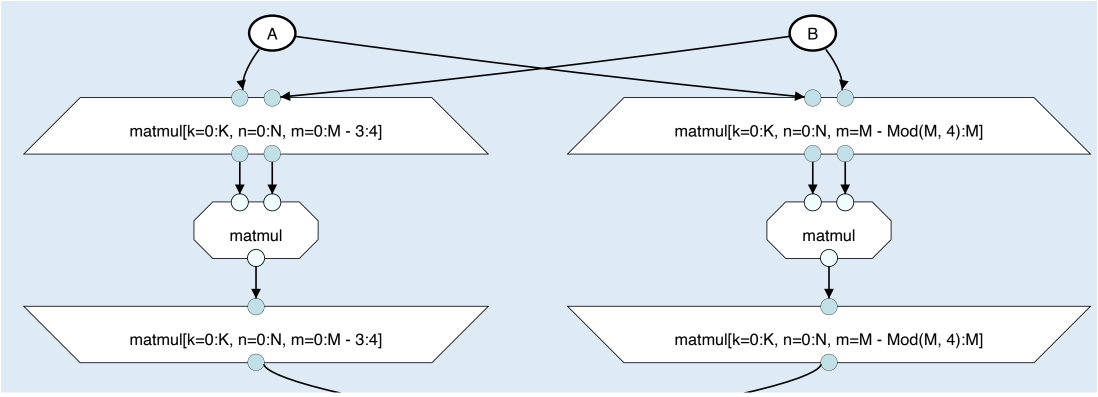

A
B
matmul[k=0:K, n=0:N, m=0:M - 3:4]
matmul[k=0:K, n=0:N, m=M - Mod(M, 4):M]
matmul
matmul
matmul[k=0:K, n=0:N, m=0:M - 3:4]
matmul[k=0:K, n=0:N, m=M - Mod(M, 4):M]

Figure 5.8: The subgraph from Fig. 5.7 after applying the vectorization transformation on the three-dimensional Map scope. The computation is split into a Map with 4-way vectorized accesses, and a Map that computes leftover iterations if $M \bmod 4 > 0$.

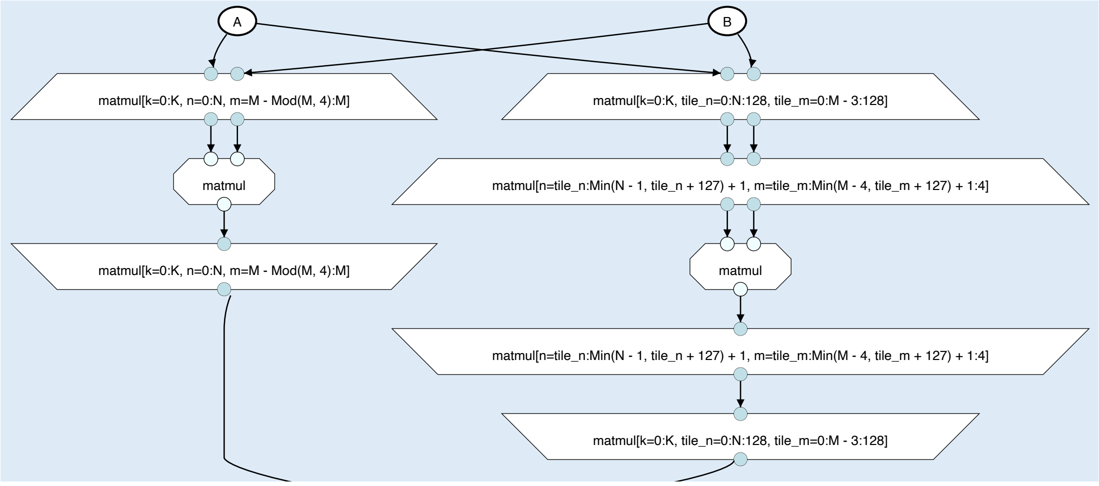

A
B
matmul[k=0:K, n=0:N, m=M - Mod(M, 4):M]
matmul[k=0:K, tile_n=0:N:128, tile_m=0:M - 3:128]
matmul[n=tile_n:Min(N - 1, tile_n + 127) + 1, m=tile_m:Min(M - 4, tile_m + 127) + 1:4]
matmul
matmul
matmul[k=0:K, n=0:N, m=M - Mod(M, 4):M]
matmul[n=tile_n:Min(N - 1, tile_n + 127) + 1, m=tile_m:Min(M - 4, tile_m + 127) + 1:4]
matmul[k=0:K, tile_n=0:N:128, tile_m=0:M - 3:128]

Figure 5.9: The subgraph from Fig. 5.8 after applying the Map tiling transformation on the `n` and `m` iterators in the vectorized Map.

```python
from dace.transformation.dataflow import MapDimShuffle, Vectorization
sdfg.apply_transformations(MapDimShuffle, options={"parameters": ("k", "n", "m")})
sdfg.apply_transformations(Vectorization)  # Defaults to 4-way vectorization
```

Before we vectorize, we reorder the iteration space such that accesses into all three matrices have a stride of 1. This is done using the `MapDimShuffle` transformation, where we specify the desired order of iterators. Applying these transformations results in the subgraph shown in Fig. 5.8. The Map is split into a vectorized subgraph, and a subgraph handling the "tail" of iterations left over if the vectorization size of 4 is not a multiple of the matrix dimension `M`. If `M` was a constant, or if an extra option is passed to the `Vectorization` transformation, the tail computation can be omitted (this might be important on spatial architectures, where the extra branch would produce dedicated hardware).

5.3.2 Example: Tiling

The tiling transformation "`MapTiling`" was implemented and is being maintained by Tal Ben-Nun, Alexandros Ziogas, and others.

Tiling is one of the most common optimizations applied in HPC. It is typically used to manipulate the access pattern into one or more memories to promote temporal locality on CPU and GPU architectures, by reducing the reuse distance between accesses so they are not flushed from the cache between them. In a procedural code, this typically involves strip-mining a loop into the desired tile size, and reordering it with the existing loops.

In an SDFG State, iteration spaces are represented by the Map primitive. By applying the `MapTiling` transformation from the standard DaCe toolbox, we can tile a Map into the desired tile size(s). Fig. 5.9 shows the subgraph from Fig. 5.8 after applying the `MapTiling` transformation, done with the following lines of code:

```python
from dace.transformation.dataflow import MapTiling
# Tile the N and M dimensions with tile size 128. Don't tile the K-dimension
sdfg.apply_transformations(MapTiling, options={"tile_sizes": (1, 128, 128)})
```

If multiple Maps were present, we would either use an API that returns the possible applications and lets us choose from them, or an API that takes the specific nodes that we wish to transform as inputs. The `MapTiling` is an example of nesting transformations to ease development and avoid duplication of functionality, as it internally uses another transformation, `StripMining`, to split indices of a single Map into two Maps, before reordering

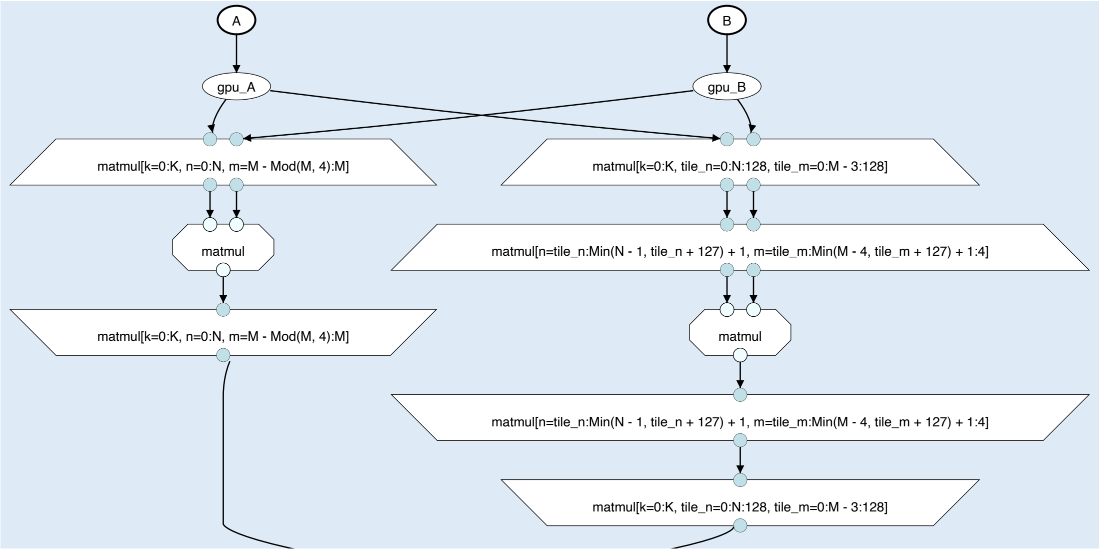

A
B
gpu_A
gpu_B
matmul[k=0:K, n=0:N, m=M - Mod(M, 4):M]
matmul[k=0:K, tile_n=0:N:128, tile_m=0:M - 3:128]
matmul
matmul[n=tile_n:Min(N - 1, tile_n + 127) + 1, m=tile_m:Min(M - 4, tile_m + 127) + 1:4]
matmul[k=0:K, n=0:N, m=M - Mod(M, 4):M]
matmul
matmul[n=tile_n:Min(N - 1, tile_n + 127) + 1, m=tile_m:Min(M - 4, tile_m + 127) + 1:4]
matmul[k=0:K, tile_n=0:N:128, tile_m=0:M - 3:128]

Figure 5.10: The subgraph from Fig. 5.9 after appling `GPUTransformSDFG` to offload the program to the GPU.

and adjusting the iteration space in the desired manner.

5.3.3 Example: GPU Offloading

The GPU offloading transformation, "`GPUTransformSDFG`" was implemented and is being maintained by Tal Ben-Nun, and uses internal components developed by Alexandros Ziogas, Tiziano De Matteis, myself, and others.

As of writing, GPUs are the most popular accelerator in HPC due to the massive vector parallelism and bandwidth that they offer. Targeting GPUs typically involves rewriting performance critical kernels in CUDA/HIP/OpenCL, or using a library like cuBLAS or TensorFlow to automatically offload a fixed set of supported kernels.

To offload an SDFG to the GPU, a user can simply use the `GPUTransformSDFG` transformation to offload an entire SDFG and all its content, or `GPUTransformMap` to offload just a single Map. This will generate all the appropriate kernel code (CUDA or HIP), host/device memory copies, and kernel launches when the SDFG is compiled. Fig. 5.10 shows the subgraph from Fig. 5.9 after applying the `GPUTransformSDFG` transformation, achieved with the following lines of code:

```
from dace.transformation.interstate import GPUTransformSDFG
sdfg.apply_transformations(GPUTransformSDFG)
```

This has created GPU memory containers, and inserted accesses to them before the computation, such that memory is copied to the device before a kernel is launched to implement the iteration space. While the particular example of matrix multiplication might seem like a futile effort considering the performance of libraries like MKL, cuBLAS, CUTLASS, DaCe has been shown (with additional transformations) to reach 99%, 70%, and 90% of MKL, cuBLAS, and CUTLASS performance on matrix multiplication [16], respectively, and was even shown to significantly outperform both cuBLAS and CUTLASS on strided, batched matrix multiplications on GPU [162].

5.3.4 Example: Control Flow Loops to Parallel Maps

The "`LoopToMap`" transformation described below was implemented by myself, Tal Ben-Nun, and Alexandros Ziogas.

While the previous three example transformations focused on optimizing existing parallel scopes, the DaCe repository also contains numerous transformations aimed at extracting parallelism and dataflow from control flow or suboptimally expressed programs (this can be thought of as a form of canonicalization towards a form with maximal dataflow sections and minimal control flow).

The code in Lst. 22 shows an implementation of sparse matrix-vector multiplication (SpMV) operating on a compressed sparse row (CSR) format, implemented in the Python frontend of DaCe. The iterations over rows `M` are implemented as a Python for-loop, which is a fundamental construct. We use the following code to parse the program into an SDFG:

```
sdfg = spmv.to_sdfg()
sdfg.apply_strict_transformations()
```

Calling the `apply_strict_transformations` method will exhaustively apply a suite of "safe" transformations that are guaranteed to *not decrease* the performance of the SDFG. Examples of this include fusing consecutive States that will not result in a race condition (eliminating unnecessary control flow), or removing redundant data copies (when data is copied, but the original data source can be used instead of the copy). The SDFG for this program will look like Fig. 5.11, where the Python loop is implemented as State transitions

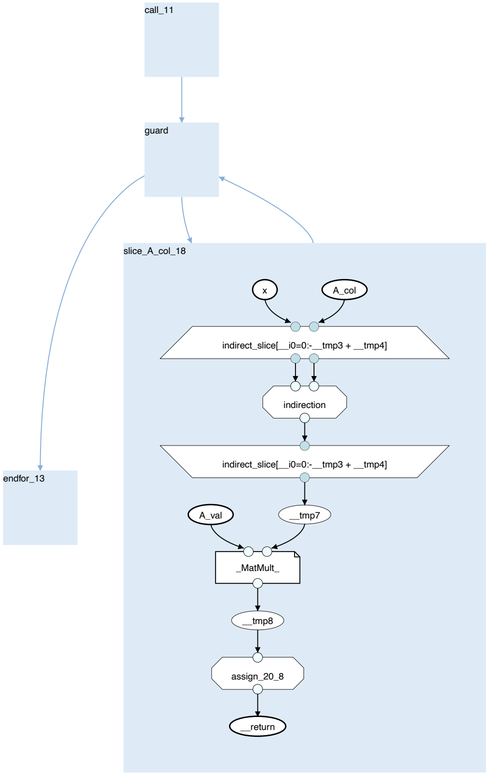


Figure 5.11: The SDFG generated from Lst. 22 after basic canonicalization.

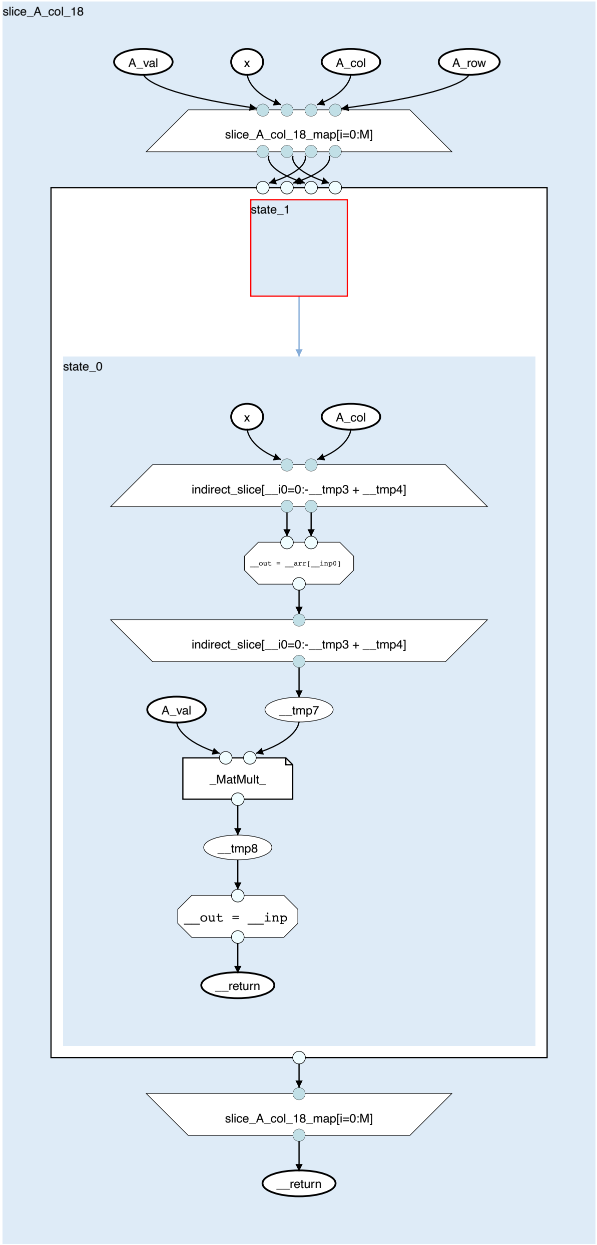

slice_A_col_18
A_val
x
A_col
A_row
slice_A_col_18_map[i=0:M]
state_1
state_0
x
A_col
indirect_slice[__i0=0:-__tmp3 + __tmp4]
__out = __arr[__inp0]
indirect_slice[__i0=0:-__tmp3 + __tmp4]
A_val
__tmp7
MatMult
__tmp8
__out = __inp
__return
slice_A_col_18_map[i=0:M]
__return

Figure 5.12: The SDFG from Fig. 5.11 after converting the control flow loop into a parallel Map.

```python
import dace as dc

M, N, nnz = (dc.symbol(s, dtype=dc.int64) for s in ("M", "N", "nnz"))

@dc.program
def spmv(A_row: dc.uint32[M + 1], A_col: dc.uint32[nnz],
         A_val: dc.float64[nnz], x: dc.float64[N]):
    y = np.empty(M, A_val.dtype)

    for i in range(M):
        start = dc.define_local_scalar(dc.uint32)
        stop = dc.define_local_scalar(dc.uint32)
        start = A_row[i]
        stop = A_row[i + 1]
        cols = A_col[start:stop]
        vals = A_val[start:stop]
        y[i] = vals @ x[cols]

    return y
```

Listing 22: A CSR-based sparse matrix-vector multiplication (SpMV) written in the Python frontend of DaCe, adapted from the NPBench [163] repository.

in the coarse-grained control flow graph.

By employing the `LoopToMap` transformation from the standard DaCe suite, we can detect control flow loops that are safe to convert into parallel Map scopes. This is possible on an SDFG, because all memory accesses are explicit in the representation, allowing us to detect if this conversion would result in a race condition. Because every write to the non-local data container `__return` happens at as distinct index `i` in Fig. 5.11, we can safely transform it with the following code:

```python
from dace.transformation.interstate import LoopToMap
sdfg.apply_transformations(LoopToMap)
sdfg.apply_strict_transformations()  # Clean up
```

This will result in the SDFG shown in Fig. 5.12, where the loop over rows is now a parallel Map scope, which can be distributed on multiple cores, or transformed further to exploit this newly extracted parallelism.

The transformations described above are a small sample of the zoo of transformations available in the standard DaCe repository, provided as tools for the performance engineer to perform guided optimization of programs. With the exception of the GPU transformation, all the above transformations are also relevant in the context of optimizing spatial architectures, which will be the focus of Sec. 5.6 and the remainder of this chapter. We will also describe more FPGA-oriented transformations in that context.

5.4 The DaCe Framework

In the following, we give a high-level overview of the capabilities of the DaCe framework, in addition to the transformations described in Sec. 5.3. All these aspects are developed in a highly collaborative manner, so each subsection will contain explicit attribution to the main contributors, ordered by degree of contribution on a best effort basis.

5.4.1 Frontends

The Python-based frontend was built by Tal Ben-Nun and Alexandros Ziogas, and the NumPy extensions were built by Alexandros Ziogas. DaCeML was built by Oliver Rausch and Tal Ben-Nun. The SDFG graph API was built by Tal Ben-Nun, myself, and others.

While the SDFG is the proposed language for the performance engineer, the domain scientist is expected to use one or more productive high-level frontends to produce the SDFGs in the first place. We will not emphasize the general-purpose frontend languages in the context of this dissertation (rather, we focus on the domain-specific language implemented for StencilFlow in Chapter 6), but mention some of them briefly here.

The NumPy-focused Python frontend provides a highly versatile way for scientists familiar with NumPy-based codes to create SDFGs. This was shown for the SpMV example code in Lst. 22, and supports matrix and vector operations (such as all the linear algebra multiplications implied by the Python operator “`@`”), element-wise operations, and general Python control flow, such as conditional statements and loops. More details, comprehensive benchmarks, and comparisons to competing frameworks for this frontend can be found in our SC’21 paper [164].

To cater to the machine learning crowd, the DaCeML [1] extension provides support for ONNX and PyTorch, in additional to the TensorFlow support included in the main DaCe repository. DaCeML uses Library Nodes (Sec. 5.5) to target CPU, GPU, and FPGA, and includes custom machine-learning oriented transformations.

For domain scientists that intersect with the performance engineering role, SDFGs can also be authored “manually” by using a graph API to define the nodes and edges that describe the computation. Many convenience methods are defined on the various SDFG objects to facilitate this. DaCe provides a graphical user interface, now maintained in the form of a Visual Studio Code plugin[3] developed by Philipp Schaad, which also allows developers to see and manipulate the graphical representation of the SDFG directly.

All the above frontends produce SDFG objects, which can be serialized and stored into JSON-based `.sdfg` objects, to be manipulated (if desired) by the performance engineer.

5.4.2 Code Generation

The central DaCe code generator was implemented by Tal Ben-Nun, myself, Alexandros Ziogas, and others. GPU code generation was implemented by Tal Ben-Nun and others. FPGA code generation was implemented by myself, Tiziano De Matteis, and others.

To actually execute an SDFG as a function, it must first be code generated into code that can be compiled with a general purpose compiler. In the DaCe framework, this is implemented as multiple backends that are each responsible for different targets, but reuse a significant amount of code between them, as they all ultimately emit C-like procedural code. This dissertation will focus on the FPGA backends, but even FPGA programs utilize both the CPU backend for generating host code, and general purpose C++ code generation shared across backends.

In the following, we give a brief overview of how different aspects of the code generator interpret the SDFG representation to emit compilable code.

5.4.2.1 Graph Traversal

To generate code, the DaCe framework traverses the SDFG in topological order (both in the dataflow layer and in the control flow layer), recursing into nested SDFGs in a depth-first fashion as they are encountered. When cycles occur in the control flow graph, dominator analysis is used to determine the order of traversal. When multiple weakly connected components appear within the same Dataflow State (i.e., no edges exist between two distinct sets of nodes), they are considered independent and can be executed in parallel (e.g., as software threads on the CPU, asynchronous GPU kernels, or as processing elements on the FPGA). A control flow detection procedure emits if-conditions, for-loops, or while loops, if they can be inferred from the control flow between Dataflow States.

---

[3] `https://marketplace.visualstudio.com/items?itemName=phschaad.sdfv`

5.4.2.2 Scheduling Computations

Whenever a Map primitive is encountered, the code generator will dispatch to one of many branches depending on the `schedule` property defined on the Map. By default, this will use CPU multithreading using OpenMP, but if specified, can dispatch to GPU kernels, GPU thread blocks, MPI processes, pipelined execution on the FPGA, unrolled execution on the FPGA, or just a sequential for-loop. Not every combination of nested schedules is valid: e.g., nesting a Map targeting a CPU multithreading schedule inside a GPU kernel will fail the validation pass of the SDFG. Similarly, a computation scheduled on the GPU cannot access FPGA memory. If transformed via transformations (such as the GPU offloading described in Sec. 5.3.3), these schedules will in most cases be valid, but invalid schedules can occur when manually manipulating the SDFG. Validation is only triggered before code generation or after transformations, so invalid configurations can exist intermittently during manual manipulation of the graph.

5.4.2.3 Computations

The core computations themselves are expressed with the Tasklet primitive, and can be written in C++ or Python. For Tasklets implemented in Python, the code is first parsed as a Python AST, then “un”parsed into C++, using information about which data is available to the Tasklet and how it is accessed to ensure that the data movement semantics of the SDFG are obeyed. For code implemented in C++, the code generator pastes the code more or less as-is, relying on the programmer to adhere to SDFG semantics (e.g., not exploiting pointers to access data that was not passed to the Tasklet). This procedure is common for all backends, with the exception of the Intel FPGA backend, which needs to emit more constrained code to target OpenCL (e.g., types must be deduced in DaCe rather than leaving this to the compiler).

5.4.2.4 Memory Allocation

The `storage` property of data containers defines in which type of memory it is located. When relevant, an additional `location` property is used to further qualify this (e.g., whether to allocate FPGA global memory in DDR bank 0 or HBM bank 22, or in a multi-GPU system, which GPU’s global memory should be used). These properties will also determine how the data is allocated during execution of the program (for data that is not part of the arguments to the program). By default, any data container is allocated within the most constrained scope in which it is used. For example, if a local buffer is only used within a single Map, it will be allocated immediately before this Map is scheduled, and destroyed immediately after. If instead the buffer is used across multiple computations within the same Dataflow State, it will be constructed and destroyed immediately before and after the code for this State is emitted, respectively. Exceptions to this strategy occur when the allocation strategy is manually overwritten by the performance engineer (as a property on the data container), or when the emitted language does not allow this (e.g., FPGA global memory cannot be allocated from device code, and must be allocated from the host code and passed to the kernel).

5.4.2.5 Memory Accesses

Computations can only access memory that has been explicitly passed to one of its connectors (ports) in the graph. The offset and subset of data made available is computed from the annotations on the dataflow edges originating from or arriving at data access nodes. When SDFGs are nested, dataflow edges connected to the node representing the nested SDFG are reinterpreted based on the accessed subset as a new data container within the nested SDFG. The offset used to compute final pointer – and the strides used to access such a pointer – is thus the result of all subsets passed throughout the nesting hierarchy. For example, if a single column of a matrix stored in row-major format is passed to a nested SDFG, the nested SDFG will only see and access the single column of data, but the appropriate stride corresponding to the row size of the outer matrix will be used when accessing consecutive elements in the column.

Once the C++/CUDA/OpenCL code has been emitted into a cache folder, the SDFG's configuration is passed to DaCe's build system, which will perform the system configuration, code compilation, and linking based on which platforms are targeted.

5.4.3 Build System

The DaCe build system was designed and built by myself, Tal Ben-Nun, and others.

DaCe uses a CMake-based build system to compile and link the generated code on the host system. The CMake script uses a list of files provided by the Python-based framework, determining which compilers are required based on the files provided – apart from a CPU host compiler, a GPU and/or an FPGA compiler might be required. Which specific compiler version to use is inferred from the user's `PATH` and environment variables as per the standard CMake flow, but can also be overwritten from the DaCe configuration file (default location `~/.dace.conf`). Options used for compilation and linking are also taken from the DaCe configuration file, including flags passed to the compiler/linker, and which FPGA vendor (Xilinx or Intel FPGA) and which FPGA platform to target (e.g., the `xilinx_u250_gen3x16_xdma_3_1_202020_1` shell when targeting an Alveo U250 board), if FPGA kernels are present in the SDFG. If the build system is triggered for an SDFG that has previously compiled with no changes, the code will not be recompiled, automatically implementing ahead-of-time compilation even when not explicitly requested.

5.4.4 Instrumentation

The instrumentation engine in DaCe was implemented by Tal-Ben Nun, Philipp Schaad, and others, and the FPGA instrumentation extension was built by myself.

Some component in an SDFG are amenable to instrumentation in the DaCe framework. In particular, Map scopes, Dataflow States, individual Tasklets, and full SDFG objects can be configured with the `instrument` property, automatically collecting profiling information for their execution during runtime. DaCe will automatically instantiate the appropriate profiling code for the given primitive and schedule in which it resides, such as OpenCL events for FPGA execution, CUDA events for GPU execution, or regular timers for CPU execution. The profiling information can be retrieved directly from the SDFG object after running the program, or accessed conveniently in the Visual Studio Code extension.

5.4.5 Summary

To conclude the overview of SDFGs (Sec. 5.2), their transformations (Sec. 5.3), and the DaCe framework (Sec. 5.4), we give an executive summary of the central concepts described.

What is DaCe? DaCe is a Python-based, open source, HPC-oriented framework that is used to produce, manipulate, and compile SDFGs.

What are SDFGs? SDFGs are a graph-based intermediate representation used by DaCe. Optimizations are performed as graph transformations on the SDFG. They are compiled into binaries that can be executed from Python or C++.

How are SDFGs made? Typically SDFGs are emitted by a frontend language. However, they can also be written and manipulated directly, either programmatically or through a graphical user interface.

How are SDFGs executed? They are code generated into a C-based language and compiled using off-the-shelf CPU, GPU, and/or FPGA compilers.

Is understanding SDFGs necessary to use DaCe? No. SDFGs are a useful tool for developers to view and understand the dataflow in their program to address bottlenecks, but can also be treated as a transparent intermediate step.

Who is DaCe for? Anyone interested in high-performance computing on CPUs, GPUs, FPGAs, or any future supported platform. This includes software developers, hardware developers, software developers interested in hardware, and pure performance engineers looking to optimize programs. We distinguish between the **domain scientist**, who is concerned with the output of the scientific application, and the **performance engineer**, who is concerned with improving the runtime of the scientific application. These two roles can be filled out by one, two, or more developers with the necessary expertise.

5.5 Multi-Level Design with Library Nodes

From this section and for the remainder of this chapter, I am the primary author of all conceptual and engineering contributions described unless explicitly stated otherwise, with Tiziano De Matteis and Tal Ben-Nun as significant contributors.

The data-centric view exposed in the SDFGs representation allows the performance engineer to perform a myriad of general purpose transformations to optimize the data movement of the application, such as the examples shown in Sec. 5.3. Some optimizations, however, arise from knowledge about the underlying application domain (for example, algebraic identities), which are difficult or impossible to express generally without encoding domain-specific knowledge into the representation. Furthermore, in some scenarios, it might be more desirable to call external libraries, such as MKL or cuBLAS, rather than implementing operators manually, or to reuse existing subgraphs of fine-grained dataflow in DaCe across different applications.

To accommodate reuse of optimized subgraphs, external library support, and domain-specific optimizations, we introduce the concept of **Library Nodes**, embedded in the SDFG representation. These nodes represent an *abstract behavior* (the "what") on the incoming and outgoing dataflow, as opposed to a concrete implementation of this behavior, (the "how"). Library Nodes are "expanded" by replacing them with a subgraph, progres-

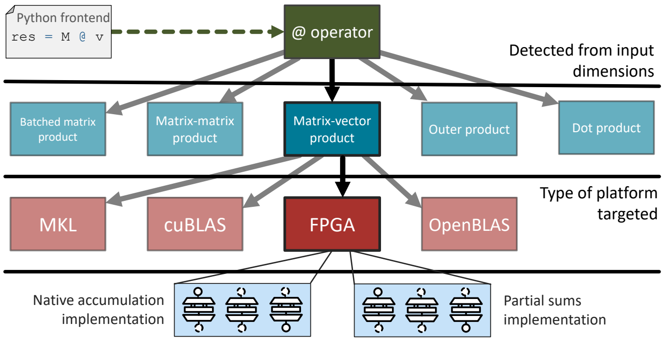


Figure 5.13: Multiple levels of nested Library Node expansions.

```python
@dace.program
def gemv(M: dace.float32[N, N], v: dace.float32[N], res: dace.float32[N]):
    res[:] = M @ v
```

Listing 23: Simple DaCe program using the `@` operator.

sively "lowering" them towards a concrete implementation of their behavior, inspired by the MLIR [97] ecosystem. During this expansion, Library Nodes can inspect their context using the surrounding nodes and edges, which may change the structure of the expanded subgraph, e.g., by checking if inputs or outputs are streams or if they use vector types.

Because Library Nodes only represent *what* is computed before they are expanded into *how* it is computed, all Library Nodes must be fully expanded before code can be generated from the SDFG representation. However, they can go through several levels of expansions (a.k.a. progressive lowering [97]) before reaching a fully expanded State.

Example. An example of progressive lowering using Library Nodes is illustrated in Fig. 5.13, based on the simple DaCe program shown in Lst. 23 using the matrix multiplication operator "`@`". In the generated SDFG, this will be represented as a generic matrix multiplication Library Node, shown in Fig. 5.14. As we lower this operator, we will mark the corresponding subgraph between consecutive stages with a dashed red outline in each figure. We go through the following phases of lowering and transformation to end up at a tiled FPGA implementation of the input computation:

1. Based on the number of dimensions of the two input operands, the Library Node is lowered to the corresponding BLAS-like operand: inner vector product (`DOT`); outer vector product (`GER`); matrix-vector multiplication (`GEMV`); matrix-matrix multipli-

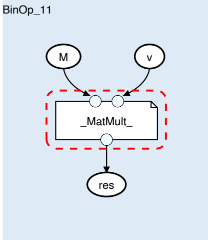


Figure 5.14: ❶ Initial SDFG.

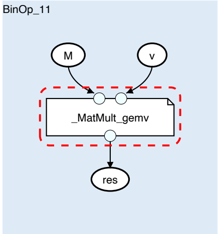


Figure 5.15: ❷ Inferred as GEMV.

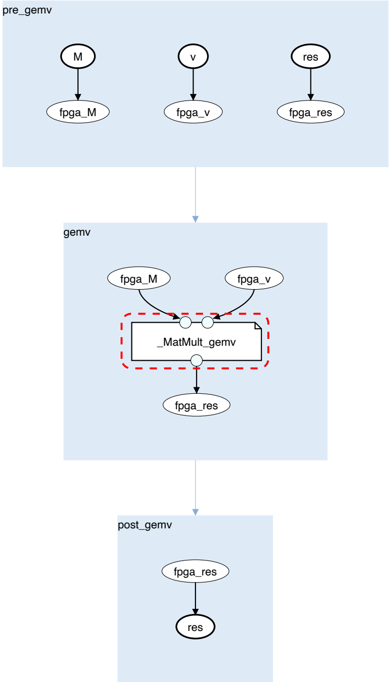


Figure 5.16: ❸ Offloaded to FPGA.

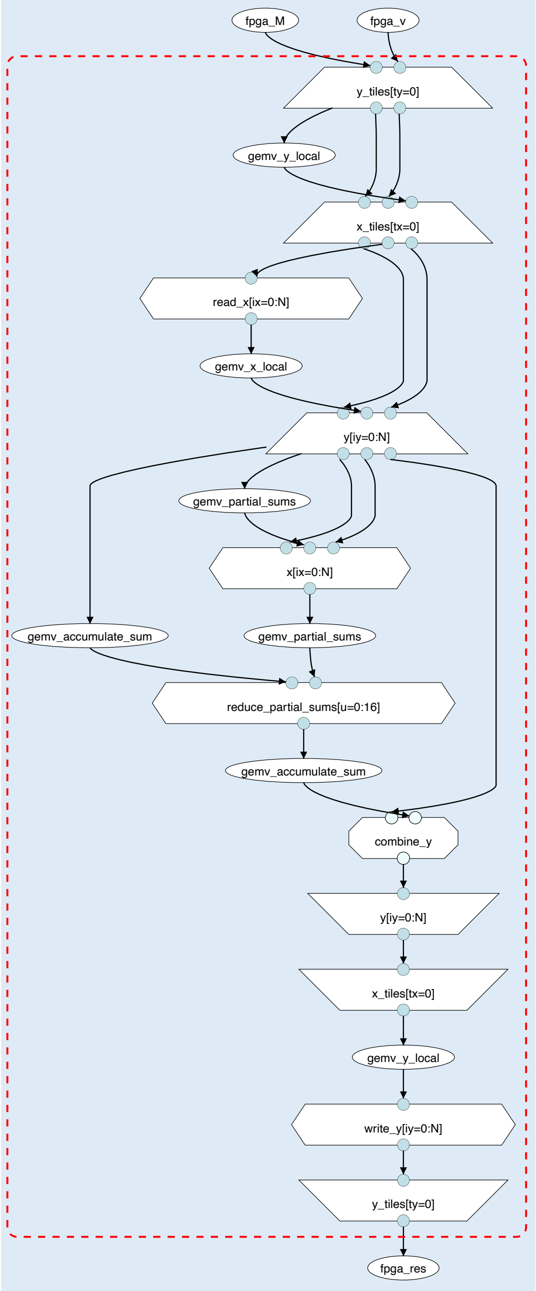


Figure 5.17: ❹ Lowered to FPGA implementation.

cation (`GEMM`); or batched matrix-matrix multiplication. Each of these operations are represented by distinct type of Library Node. For the example code, the operator will be inferred as a `GEMV` operation, shown in Fig. 5.15. In Fig. 5.16, we have applied the transformation to offload the computation to the FPGA, without expanding the abstract `GEMV` operation.

2. Depending on which type of platform the user wants to target, the BLAS-like operation is then expanded either to a library call (e.g., MKL or cuBLAS), or to a different subgraph that implements the operation. When targeting FPGA, there can be multiple implementations per operator depending on the desired tiling pattern [41], each represented by a distinct type of Library Node. In Fig. 5.17, we have expanded the `GEMV` computation to a tiled FPGA implementation. Some Map scopes have been collapsed for readability (drawn as hexagons in the figure).

3. Once the implementation has been chosen, some subgraphs might have additional specialization to the FPGA vendor. For example, the reduction in `GEMV` can be implemented with a single-cycle accumulator for 32-bit floating point on Stratix 10 and Arria 10 architectures, but requires a custom accumulation circuit on Xilinx FPGAs, where this is not natively supported.

Library Nodes are automatically emitted when the **domain scientist** uses a frontend that exposes higher level concepts, such as NumPy, and directly translate into a fast implementation on the target platform. They expose additional domain knowledge, and can be used by the **performance engineer** to target different backends, reuse fast implementations, and apply domain-specific transformations. A power user can write new Library Nodes to build new libraries and form new abstractions, which can be harnessed in new domain-specific transformations.

5.5.1 Domain-Specific Optimizations

By embedding the high-level behavior of operations into the SDFG, we unlock access to domain-specific transformations that exploit the abstract behavior of Library Nodes. We will see an example of this in Chapter 6, where we can fuse stencil operations based on their abstract representation, independent of the final platform that the program will be compiled for. This can done using the standard transformation framework of DaCe without any further extensions, by matching on patterns that include one or more Library Nodes of the desired type(s) and their properties.

5.5.2 Targeting Vendor Libraries

A typical use-case of Library Nodes is to hook into external libraries that already provide fast implementations of the desired high-level operation. Linear algebra operations, such as the one shown above, is a common use-case, where users may wish to simply call MKL or cuBLAS without having to manually optimize this aspect of their application. Other examples include FFT (calling FFTW or cuFFT), sparse linear algebra, solvers, and neural network operators. Embedding these library calls into DaCe has multiple benefits:

- By embedding high-level operators from one of the productive frontends to DaCe, the performance engineer can immediately target a range of vendor libraries without having to modify the code. DaCe libraries support setting a default Library Node expansion that will be used when the graph is compiled without any further intervention (e.g., setting `MKL` as the default BLAS expansion will automatically compile all supported linear algebra calls to call MKL routines).
- Swapping out a library for a different one becomes a trivial change: changing between OpenBLAS and MKL, or between cuBLAS and an AMD equivalent, becomes a single line of code. Even swapping between a CPU and GPU backend is trivial, although additional transformations would be recommended to avoid unnecessary data transfers between host and device for multiple consecutive calls.
- The performance engineer can simultaneously benefit from applying DaCe optimizations to the aspects of the SDFG implemented as fine-grained dataflow, and from offloading other aspects to vendor libraries while keeping their coarse-grained data movement within the representation. DaCe thus becomes a one stop shop for both fine-grained and coarse-grained optimizations.
- After quickly reaching a performant implementation by calling external libraries, the performance engineer can proceed to experiment with using fine-grained expansions instead for selected Library Nodes, which might expose cross-operation optimization opportunities (e.g., fusing maps schedules across operators, which was shown to be greatly beneficial for batched matrix multiplications in OMEN [162]).

While targeting FPGA libraries would have the same benefits as for CPU and GPU programs, the set of available FPGA libraries is very limited, and fine-grained implementations in DaCe can be as good or better than what is provided by the vendors (e.g., the matrix multiplication accelerator presented in Chapter 4 as of writing outperforms libraries provided by Xilinx). Furthermore, one of the most important optimizations when targeting FPGAs is streaming data between operators, which requires operators to be compiled into the same bitstream. DaCe has support for embedding IP cores into SDFGs for the Xilinx backend, but for external libraries the source code would be required. We explore the topic of FPGA libraries in more detail in our work on FBLAS [41].

5.5.3 A Note on MLIR

As previously noted, the multi-level design methodology implemented with the Library Node abstraction in DaCe was inspired by the concept of lowering in the MLIR [97] framework. As the DaCe project predates MLIR, the SDFG transformation machinery was developed earlier. SDFGs have since been implemented as an experimental MLIR dialect[4], which would allow data-centric transformations to be expressed within this framework instead. While MLIR offers appealing integration with various frontends and backends, the visual and interactive approach taken by the DaCe optimization flow – as opposed to automatic compiler passes – remains a central component of the DaCe philosophy, and remains the primary approach to optimizing SDFGs.

For the remainder of this dissertation, we will focus on the capabilities of the SDFG representation and the DaCe framework applied to spatial computing systems – specifically, Xilinx and Intel FPGA platforms.

5.6 Data-Centric Programming for FPGAs

After a decades long transition, HLS languages have become an accepted and widespread programming model in the reconfigurable computing community [105], complementing – albeit not fully replacing – behavioral register transfer level (RTL) design. In the domain of high-performance computing in particular, HLS has allowed programmers from a wider range of backgrounds to start using FPGAs as an alternative accelerator architecture, by interfacing with software using familiar abstractions such as OpenCL [32].

While reconfigurable computing was transitioning from RTL to C-based languages, the HPC community has begun to transition from C++ to Python as the preferred language for application development [164] following the trend in the general software community [58],

---

[4] https://github.com/spcl/mlir-dace

owing to Python's high productivity and extensive library support. The FPGA community is following this trend: the API for programming Xilinx FPGAs is exposed in Python bindings through the PYNQ package; and domain-specific frameworks targeting FPGAs typically provide their frontend interface as Python bindings [140, 38]. While some work exists to bring the core kernel development of FPGA kernels into Python [93], the majority of such development is still done in HLS and RTL languages. It is thus an ongoing effort to develop the abstractions and tooling necessary to lift FPGA kernel development into Python in a way that simultaneously preserves Python's productivity and the quality-of-result of lower-level approaches.

SDFGs allow representing programs by their dataflow and control flow independent of the chosen FPGA backend, enable compatibility across FPGA vendors through code generation, and are amenable to optimizing transformations performed on the graph representation, instead of intrusive optimizations to the source code (hardware development suffers from the same issue of making optimized code unreadable as software, perhaps even more due to the low-level nature of hardware code).

5.7 From SDFG to Spatial Architecture

In the following, we provide an overview of how SDFGs represent FPGA programs, and how the key concepts described throughout Sec. 5.2 are translated into fast HLS code for the two major vendors: Xilinx, through the Vivado HLS [160] C++-based compiler; and Intel, through the Intel OpenCL SDK for FPGA [32].

5.7.1 Code Generating SDFGs

DaCe follows the guiding principle that as many optimization opportunities as possible should be kept part of the representation — where they can be manipulated by the performance engineer — rather than happening as "magic" during code generation. Nevertheless, emitting functional and efficient code from SDFGs poses a significant design and engineering challenge, with numerous kinks and subtleties arising from moving into the hardware domain. The code generator must translate the final representation into structured HLS code that is easily digestible by the compiler, faithfully follows the functional semantics of the SDFG, and successfully achieves all parallelism implied by the representation.

The FPGA backend of DaCe is modularized into a generic part, which orchestrates the

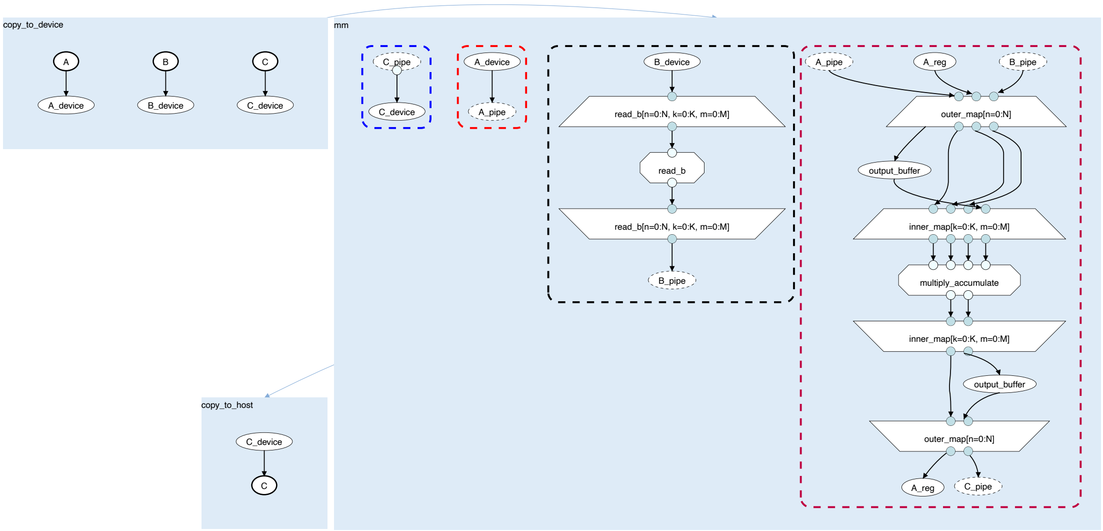


Figure 5.18: Kernel State with four processing elements (right), with pre- and post-States (left) copying memory between host and device.

traversal of the SDFG, and two lower level specialized backends for Xilinx and Intel, which are responsible for emitting vendor specific code for Vivado HLS (C++) and the Intel OpenCL compiler, respectively. The generic backend contains the most sophistication in terms of interpreting the representation and delegating code generation tasks, whereas the two specialized components are primarily concerned with vendor-specific semantics (e.g., how processing elements and memory interfaces are expressed) and syntax (e.g., vector data types and stream objects). In particular, the highly restricted syntax supported by OpenCL requires more verbose syntax to be emitted than for backends that support C++.

5.7.2 Parallelism, Pipelining, and Unrolling with Maps

Representation. Parallel sections in SDFGs are expressed via the Map construct. In software, these scopes can target multi-core and SIMD parallelism for both CPUs and GPUs. In hardware, we can exploit the parallelism implied by Maps in two ways: with *pipelining*, where iterations are executed in sequence, but exploit pipeline parallelism in the mapped computation; or with *unrolling*, representing parametrically replicated hardware, such as systolic arrays (see Sec. 5.7.6) or SIMD-style vectorization. Unrolling is annotated as a property on the Map object, while all non-unrolled Maps are pipelined. The purple box in Fig. 5.18 contains an inner Map, which will be generated as a pipelined inner loop, and an outer Map over tiles, orchestrating the buffering behavior.

Code generation. During code generation, the graph is traversed from outermost to innermost nesting to detect the innermost Map *that is not unrolled*, which will be pipelined by injecting a pragma at the generated loop. Furthermore, loop coalescing pragmas are automatically injected whenever loops generated from Maps are perfectly nested, and when necessary, pragmas to ignore false dependencies (see Sec. 5.7.7). Maps designated as being unrolled will annotate the generated loops with the vendor-specific unroll pragma, or be manually unrolled in the code if necessary.

5.7.3 Representing FPGA Kernels

Representation. The pure dataflow representation of SDFG **States** is a natural fit for mapping to streaming dataflow kernels on FPGA. When traversing the SDFG, the framework detects States *that only access memory situated on the FPGA*, designating these as FPGA kernels. Although FPGA kernels are always inferred from pure Dataflow States, coarse-grained control flow is still achievable within the kernel by embedding nested SDFGs as nodes.

Moving data between the host and device is represented as memory copies in the representation. Data nodes are annotated with a data location via the `storage` attribute: an enumeration that includes `FPGA_Global` (off-chip memory, such as DRAM or HBM), and `FPGA_Local`, representing on-chip memory. When connected by direct data-to-data edges in a Dataflow State, this will result in the appropriate copy operation depending on the source and destination storage. Streaming transfers can be natively represented using stream data nodes, but due to the OpenCL abstraction adopted by both the Xilinx and Intel toolflows, the backend currently only supports bulk transfers. Host/device streaming will be introduced once either backend exposes sufficient support to end-users (e.g., using the QDMA [156] subsystem for Xilinx FPGAs).

Code generation. When the code generation traversal encounters a subgraph that is detected as an FPGA kernel (when all data in the State resides on the FPGA), the dataflow section is dispatched to the FPGA backend. Before continuing the traversal to generate the hardware itself, the kernel “boundary” is generated by inferring the necessary arguments that must be passed to the resulting OpenCL kernel launch(es). Interaction with the OpenCL API is wrapped in the interface provided by hlslib (see Chapter 3), as shown in Lst. 25.

5.7.4 Processing Elements

Representation. The notion of partitioning the functionality of a kernel into multiple independently scheduled modules, commonly referred to as *processing elements* (PEs), is central to designing large FPGA architectures (see Sec. 2.3.2). Native support for this concept is thus a core consideration in the SDFG representation. At the same time, this should not introduce new FPGA-specific concepts to the representation.

SDFG States imply pure dataflow by representing data movement and data dependencies (e.g., everything contained in a blue rectangle Fig. 5.18 or Fig. 5.19), the latter of which must be respected by the code generating backend. When a dataflow graph contains more than one *weakly connected component* (i.e., at least two subgraphs G_0 and G_1 with no dataflow edge (u, v) connecting any node $u{\in}G_0$ with any node $v{\in}G_1$), the backend has the liberty to *schedule each weakly connected component in parallel.* For software backends, this can enable launching multiple concurrent GPU kernels, or running different concurrent tasks on multiple CPU threads. When appearing within an FPGA kernel, these are also scheduled as independent "tasks", exposing the concept of processing elements to the programmer. In the example in Fig. 5.18, each of the four connected components represent an independent processing element scheduled in parallel: the components in the red and the black box are memory reader/prefetcher modules, which read from arrays (solid borders) in off-chip memory into data streams (dashed borders). The red box is a simple copy of the full array dimensions, implemented by a single dataflow edge, where the black box repeats multiple reads of the array, using a Map to generate the desired access pattern. The blue box inversely writes from a stream back to memory.

Code generation. In the Vivado HLS toolflow, processing elements are expressed by annotating a scope in the C++ code with the `DATAFLOW` pragma, resulting in every loop and function call in the scope to be scheduled as a distinct processing element. This requires a top-level "entry" function that contains the processing elements, and is annotated with additional pragmas that designate the hardware interfaces used by the kernel to interact with the FPGA shell, and instantiates the on-chip streams (i.e., FIFOs) that facilitate inter-PE communication, shown for an example in Lst. 24. The Xilinx backend uses the simulation extensions from hlslib (see Sec. 3.2.3), providing the `HLSLIB_DATAFLOW_FUNCTION` macro wrappers and the thread-safe and bounded `hlslib::Stream` class (see Sec. 3.3.1), to achieve *actually* concurrent simulation of parallel processing elements, including support for feedback/back-edges in the dataflow. The Intel OpenCL flow takes a different approach: rather than being contained in a top-level function, every processing element

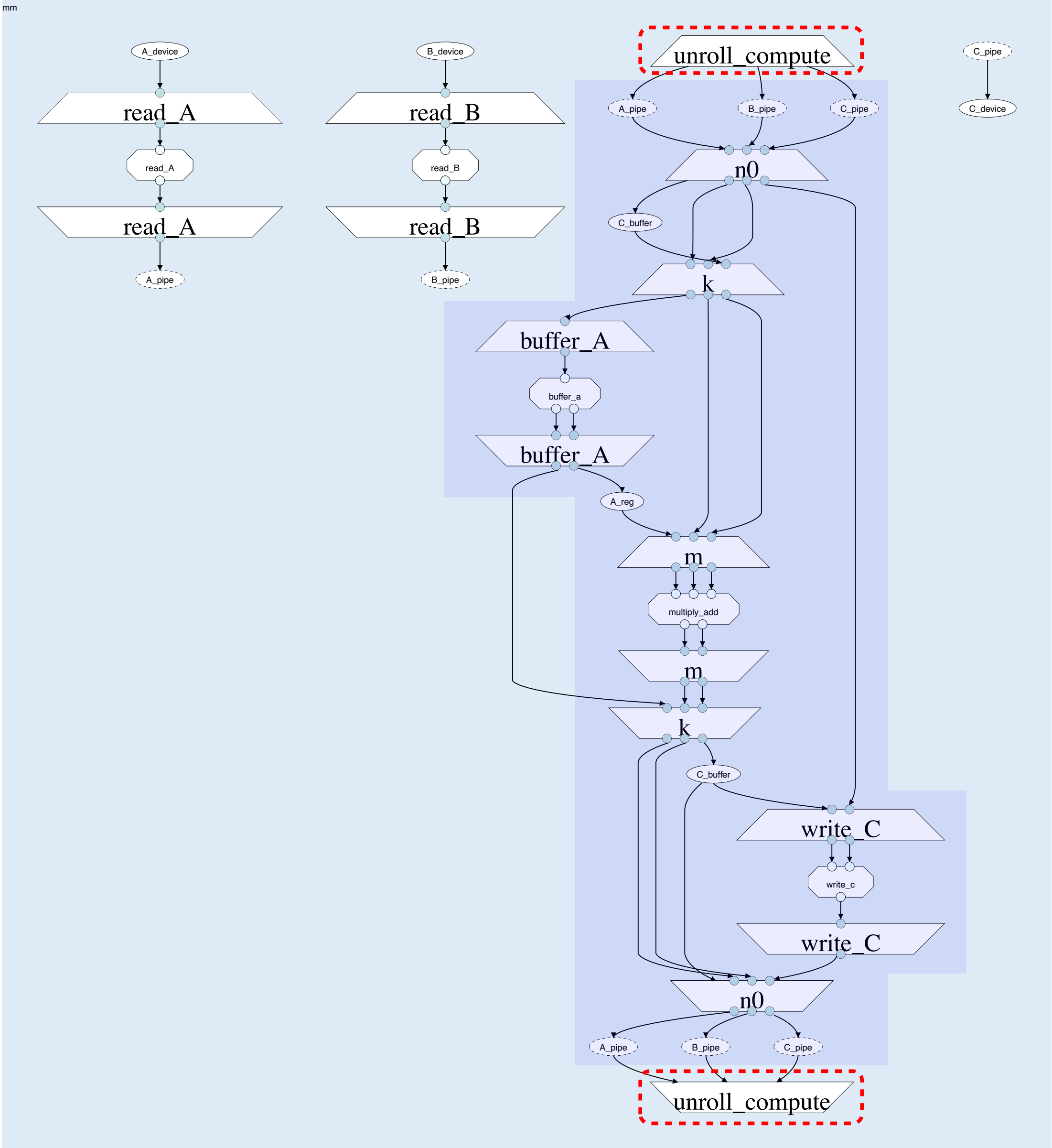


Figure 5.19: Multiple parallel processing elements, including a parametrically sized systolic array in the shaded area.

```
void mm(float *A, float *B, float *C, int n) {
  // ...interface pragmas omitted...
  #pragma HLS DATAFLOW
  HLSLIB_DATAFLOW_INIT();
  dace::FIFO<float, 1, 4> A_pipe[P + 1];
  dace::FIFO<float, 1, 1> B_pipe[P + 1];
  dace::FIFO<float, 1, 1> C_pipe[P + 1];
  HLSLIB_DATAFLOW_FUNCTION(read_A, A, A_pipe, n);
  HLSLIB_DATAFLOW_FUNCTION(read_B, B, B_pipe, n);
  for (size_t p = 0; p < P; p += 1) {
    #pragma HLS UNROLL
    HLSLIB_DATAFLOW_FUNCTION(compute, p, A_pipe, B_pipe,
                             C_pipe, n);
  }
  HLSLIB_DATAFLOW_FUNCTION(write_C, C, C_pipe, n);
  HLSLIB_DATAFLOW_FINALIZE();
}
```

Listing 24: Processing elements are implemented as top-level function calls in the Vitis HLS ecosystem. Only `mm` is invoked from the host code.

```
hlslib::ocl::Kernel kernels[] = {
    program.MakeKernel("read_A", A, n),
    program.MakeKernel("read_B", B, n),
    program.MakeKernel("compute", n),
    program.MakeKernel("compute_1", n),
    program.MakeKernel("compute_2", n),
    program.MakeKernel("compute_3", n),
    program.MakeKernel("write_C", C, n)};
std::vector<cl::Event> events = {
    kernels[0].ExecuteTaskFork(),
    kernels[1].ExecuteTaskFork(),
    kernels[2].ExecuteTaskFork(),
    kernels[3].ExecuteTaskFork(),
    kernels[4].ExecuteTaskFork(),
    kernels[5].ExecuteTaskFork(),
    kernels[6].ExecuteTaskFork()};
cl::Event::waitForEvents(events);
```

Listing 25: Processing elements are launched as kernels from the host code in the Intel OpenCL ecosystem.

must be expressed as a separate OpenCL kernel in the top-level scope, where they are connected using global `channel` objects. Launching each processing element is thus done from the host code, shown in Lst. 25. These two methods of expressing kernels thus affect both the host code (which kernels are generated and launched) and the kernel code (one vs. multiple top-level kernels, global channel objects vs. local stream objects). If a generated OpenCL kernel has *no* arguments, it will be generated as an "`autorun`" kernel, which is always active and will run whenever data is available on the connected channels, and thus does not need to be invoked from the host code.

5.7.5 Channels/Streams/FIFOs

Representation. Streams are a native data container construct in the SDFG representation, representing first-in, first-out queues, that can be used to communicate between subgraphs in dataflow sections. In CPU and GPU codes, these are employed as single- or multi-producer queues: for example, a breadth-first search kernel can produce tasks to a queue that is consumed by multiple workers, dynamically distributing work. Stream semantics are the same in FPGA kernels, but with additional constraints due to the underlying hardware implementation that they imply: streams cannot be unbounded, and must be single-producer, single-consumer. Streams facilitate communication between process-

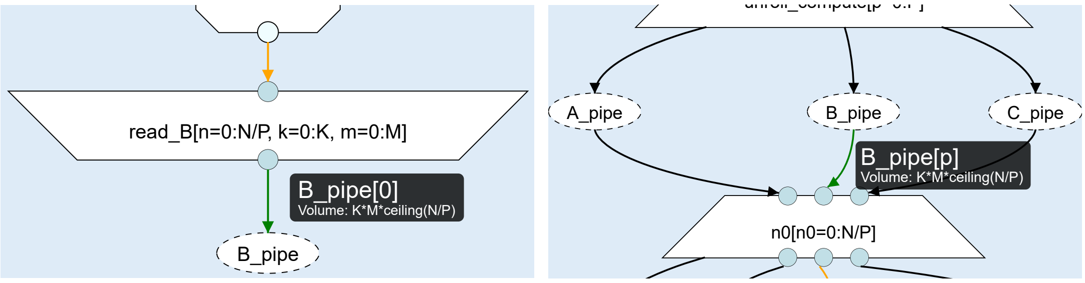


Figure 5.20: Data movement annotations on producer (left) and consumer (right) of a stream, used to verify correctness of the program.

ing elements, while simultaneously acting as *synchronization* primitives between kernels through the producer-consumer relationship. Even though the components in Fig. 5.18 do not have dataflow edges between them, they synchronize by pushing/popping the same stream data container.

Because all data movement is explicitly captured in the SDFG, programmers can benefit from the information annotated on dataflow edges to verify the correctness of producer/-consumer relationships, which are automatically inferred by the tool based on the access pattern expressed by Map scopes in the graph. Fig. 5.20 shows the annotation of a dataflow edge written by the processing element prefetching the matrix `B` from Fig. 5.19 into the stream object `B_pipe`, and the corresponding read from `B_pipe` within the processing element. The matrix of size $K \times M$ is read N/P times, where P is the tile size introduced by the systolic array (see Sec. 5.7.6), resulting in a data volume of $K \cdot M \cdot \frac{N}{P}$ annotated on the dataflow edge/memlet.

Code generation. Due to the distinct methods of expressing processing elements, the semantics of allocating streams varies significantly between the Xilinx and Intel backends. When generating Xilinx code, streams are emitted in the top-level kernel function as local objects, where they must be passed as arguments to the producer and consumer accessing them (see Lst. 24). For Intel OpenCL codes, they must be emitted to the global kernel scope, where the appropriate producer and consumer will read them directly (i.e., rather than receiving them as arguments).

5.7.6 Parametric Processing Elements: Systolic Arrays

Representation. Systolic arrays [89] are a powerful pattern to express parametric parallelism through deep pipelines, and are the most potent source of parallelism on modern

FPGAs (see Sec. 2.3.2 and Sec. 2.3.3) when applicable. SDFGs implement this pattern with unrolled Map scopes in the FPGA Dataflow State, with a parametric — but compile-time specialized — number of iterations, coupled with *arrays* of stream objects. When such a Map is unrolled, each instance semantically becomes a weakly connected component in the State, resulting in them being instantiated as separate processing elements according to the semantics in Sec. 5.7.4. This is equivalent to any other Map construct in the SDFG: namely, they represent independently executable replications of the contained subgraph (unrolled Maps can occur at any level of nesting in the program), but are recognized as a special case by the code generator when appearing in the top-level scope of an FPGA kernel.

An SDFG implementing a one-dimensional systolic array for matrix multiplication ($\mathbf{C} = \mathbf{A}\times\mathbf{B}$) is shown in Fig. 5.19, where the Map nodes annotated by red borders instantiate the systolic array. Each element implements the same content (highlighted), but reads from a distinct index in three arrays of stream objects ("pipes") for `A`, `B`, and `C`, respectively. Since every processing element is only connected to the previous and the next, they must pass data along the chain from the head towards the tail (see Sec. 4.4.1). The processing elements implement a simply buffering scheme where each element stores one element of `A` in a local buffer, then streams over the full `B` matrix, before writing back a complete output tile of `C`. This simple SDFG already yields 364 and 188 GOp/s on $8k\times 8k$ matrices when compiled for an Intel Stratix 10 and a Xilinx Alveo U250 board, respectively, with much potential for additional optimization.

Code generation. Systolic array code generation varies between vendors due to the different ways of expressing processing elements. In Xilinx codes, it is sufficient to unroll a loop in the C++ kernel code with bounds known at compile time, letting constant propagation fix all the indices in each instantiation to lay out the systolic array, as shown in Lst. 24. For Intel, the OpenCL kernel itself is replicated and specialized directly in the code generator (see Lst. 25).

5.7.7 Memory Hierarchy

Representation. Not all data movement is born equal: dataflow can have significantly different performance impact depending on the location and storage type of the source and destination, even if the number of bytes moved are the same. The FPGA backend exposes *global* memory, which represents data present in off-chip, memory-mapped storage such as DDR or HBM; *local* memory, representing any on-chip memory implementation such as registers, BRAM/M20K, LUTRAM, or UltraRAM (left up to the HLS compiler), i.e., memory that is physically local to the computational logic; *registers*, which is a subset of local memory, but forces the HLS compiler to allow fully parallel read/write access to every entry of the container; and experimental support for *shift registers*, implementing cyclic buffering patterns with multiple access points (natively supported by Intel OpenCL). Combining these allows implementing highly specialized memory hierarchies, as well as host/device interaction, in a way that is compatible with both Xilinx and Intel devices.

Code generation. Local memories can be emitted as regular C arrays directly in the kernel code, while off-chip memory is allocated with API calls in the host code and passed to the kernel arguments. The FPGA backend gives the underlying HLS compilers additional scheduling freedom by generating a distinct pointer argument for every access to the same DRAM memory container present in the kernel and marking them as `restrict` in OpenCL, such that every read and write can be performed independently, which is safe due to SDFG semantics.

Whenever both reads and writes are emitted to local memory, and the write is not marked as a potential conflict, the generated code is annotated with pragmas to instruct the compiler to ignore dependencies (`HLS DEPENDENCE` for Xilinx and `ivdep` for Intel). This is implied by SDFG semantics, where these accesses are either in dataflow sections (where conflicts must be annotated), or in control flow scopes, that are inherently sequentialized.

With the above concepts, we have covered how SDFGs are interpreted as a spatial architecture and code generated for the two major FPGA vendors. We will now take a step back, and introduce higher-level abstractions into the SDFG representation in the form of Library Nodes, built on top of these concepts to constitute the last important tool for accelerating productivity and promoting portability within the framework.

5.8 Optimizations for Spatial Architectures

Apart from the general purpose transformations described in Sec. 5.3, we can design and apply spatial computing-oriented transformations, that target the hardware optimization goals that we described in the context of HLS optimizations in Chapter 2.

We will use the input code shown in Lst. 26 as a running example, which implements `AXPYDOT`, a small composite BLAS kernel that sums two input vectors, then takes the dot product with the resulting vector and a third input vector. The SDFG generated by the

```
n = dace.symbol("n")
a = dace.symbol("a", dace.float32)

@dace.program
def axpydot(x: dace.float32[n], y: dace.float32[n],
            w: dace.float32[n], result: dace.float32[1]):
    z = np.ndarray([n], x.dace.float32)
    blas.Axpy(a, x, y, z)
    blas.Dot(z, w, result)
```

Listing 26: Implementation of AXPYDOT using the standard DaCe Python frontend and BLAS library calls.

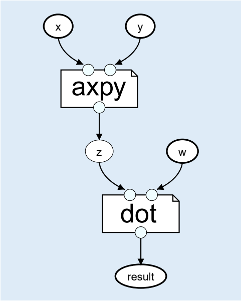


Figure 5.21: SDFG generated from Lst. 26.

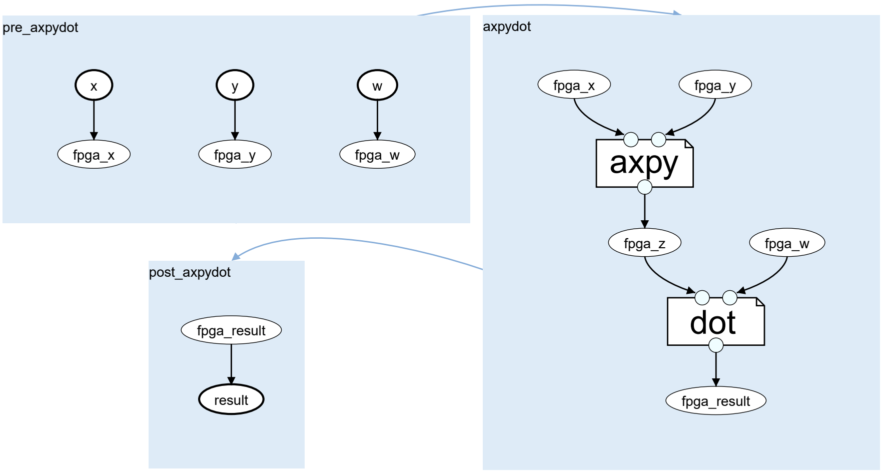


Figure 5.22: The SDFG from Fig. 5.21 automatically transformed for FPGA execution.

frontend for this code is shown in Fig. 5.21. The BLAS operators are instantiated as the `axpy` and `dot` Library Nodes, reading and writing from array inputs/outputs. The two kernels exchange data through the array `z`, which will be first written by `axpy` and then read by `dot`, in sequence. Kernels that are composed of BLAS level 1 and 2 routines, such as `AXPYDOT`, are fully memory bound, but expose a promising opportunity for streaming computation by pipelining temporaries directly between subroutines [41] on the FPGA, which we will exploit in the following.

5.8.1 Transformation: Offloading to the FPGA

As a first step, we offload the full SDFG for FPGA execution using the `FPGATransformSDFG` transformation shown in Sec. 5.9 (and following a similar approach to Sec. 5.3.3). This detects all DRAM accesses in the graph, then create additional pre- and post-states performing memory transfers between host and device. The memories accessed by the transformed subgraph are replaced with their FPGA equivalents.

Fig. 5.22 shows the `AXPYDOT` example from Fig. 5.21 after applying the `FPGATransformSDFG` transformation. Occurrences of the DRAM memories `x`, `y`, and `w` and replaced with corresponding FPGA memories `fpga_x`, `fpga_y`, and `fpga_w` in the kernel graph, the memories are copied to the FPGA before the kernel is executed in the state `pre_axpy`, and the output array `result` is copied back in the state `post_axpy`. This program can already be generated and compiled for both Xilinx and Intel boards using appropriate expansions of the two Library Nodes `axpy` and `dot`.

5.8.2 Architecture Specialization

For computations that need to perform accumulation, such as the `DOT` operator used in Fig. 5.21, it is beneficial to specialize the computation based on whether the underlying architecture supports accumulation on the given data type. Intel Arria 10 and Stratix 10 architectures supports native 32-bit floating point accumulation, which allows a stream of floats to be directly summed into an output register. Contemporary Xilinx FPGAs, such as the Alveo U250, do not have native 32-bit floating point units, and cannot directly accumulate floating point numbers into a single register, as this results in a loop-carried dependency induced by the multiple-cycle latency of the addition operation. For 64-bit floating point, neither Xilinx nor Intel support accumulation, and both must address the issue of loop-carried dependencies.

To avoid the loop-carried dependency for `DOT`, we can use the single-loop *accumulation interleaving* transformation from Sec. 2.2.1.3. We sum up the incoming data into a number of *partial* sums, stored in a buffer of a size larger than the latency of the addition operation.

In Fig. 5.23 and Fig. 5.24, the `AXPY` and `DOT` operators have both been expanded for Xilinx and Intel, respectively. `AXPY` uses a generic implementation (identical to the CPU implementation), while `DOT` is implemented using specialized expansions depending on the target architecture. The Xilinx-targeted expansion included in Fig. 5.23 uses a partial sum strategy to resolve the loop-carried dependency, using two unrolled maps. The first (“`unroll`”) sums up all entries of the vector containing the product of contributions from x and y using a fully unrolled circuit (i.e., $W-1$ adders, where W is the vectorization width), resulting in a single element contribution. This contribution is added into the partial sum buffer, accessed with a cyclic index. The second unrolled map (“`reduce`”) is

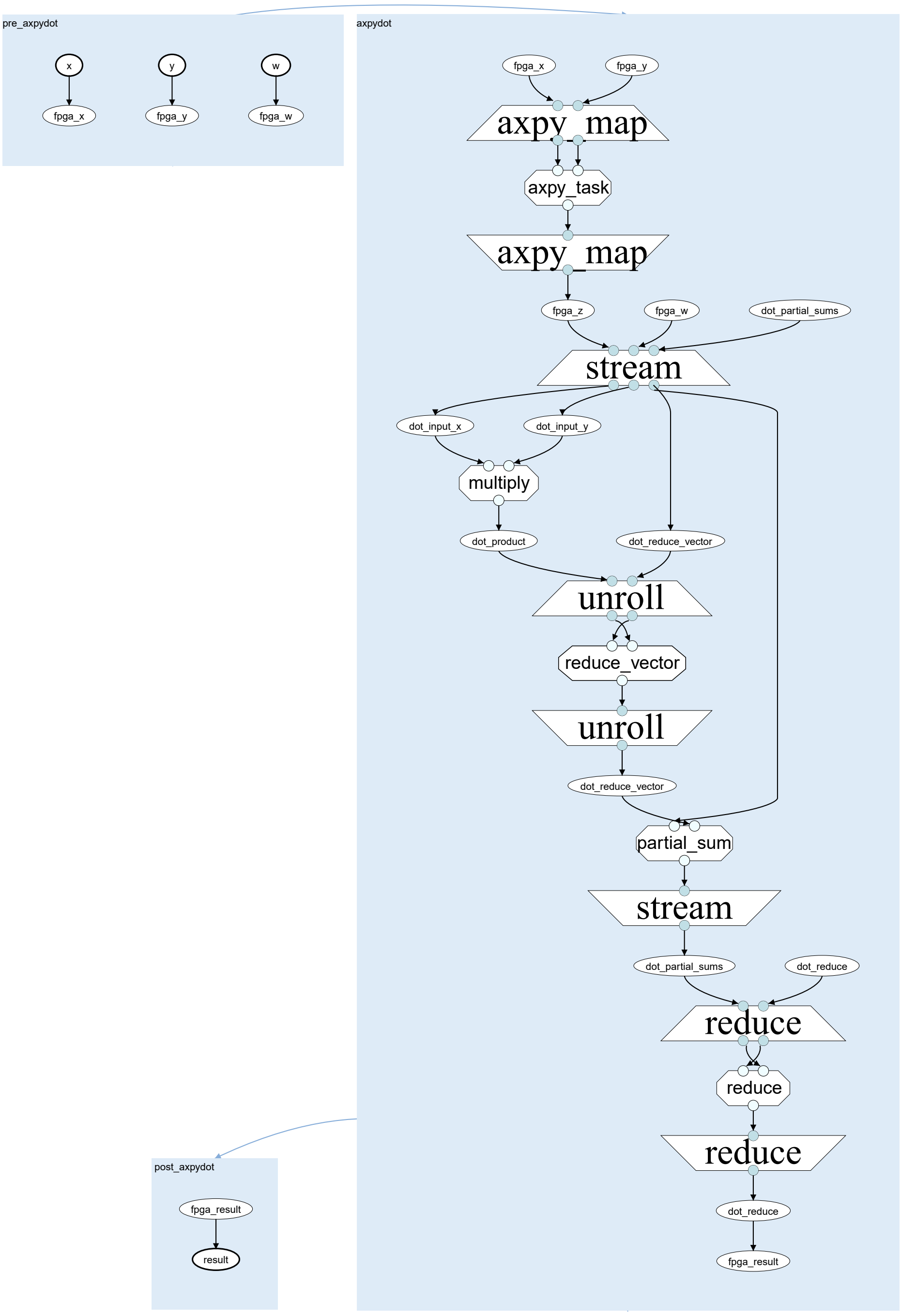


Figure 5.23: `AXPYDOT` expanded for Xilinx, using a partial sum and reduce phase.

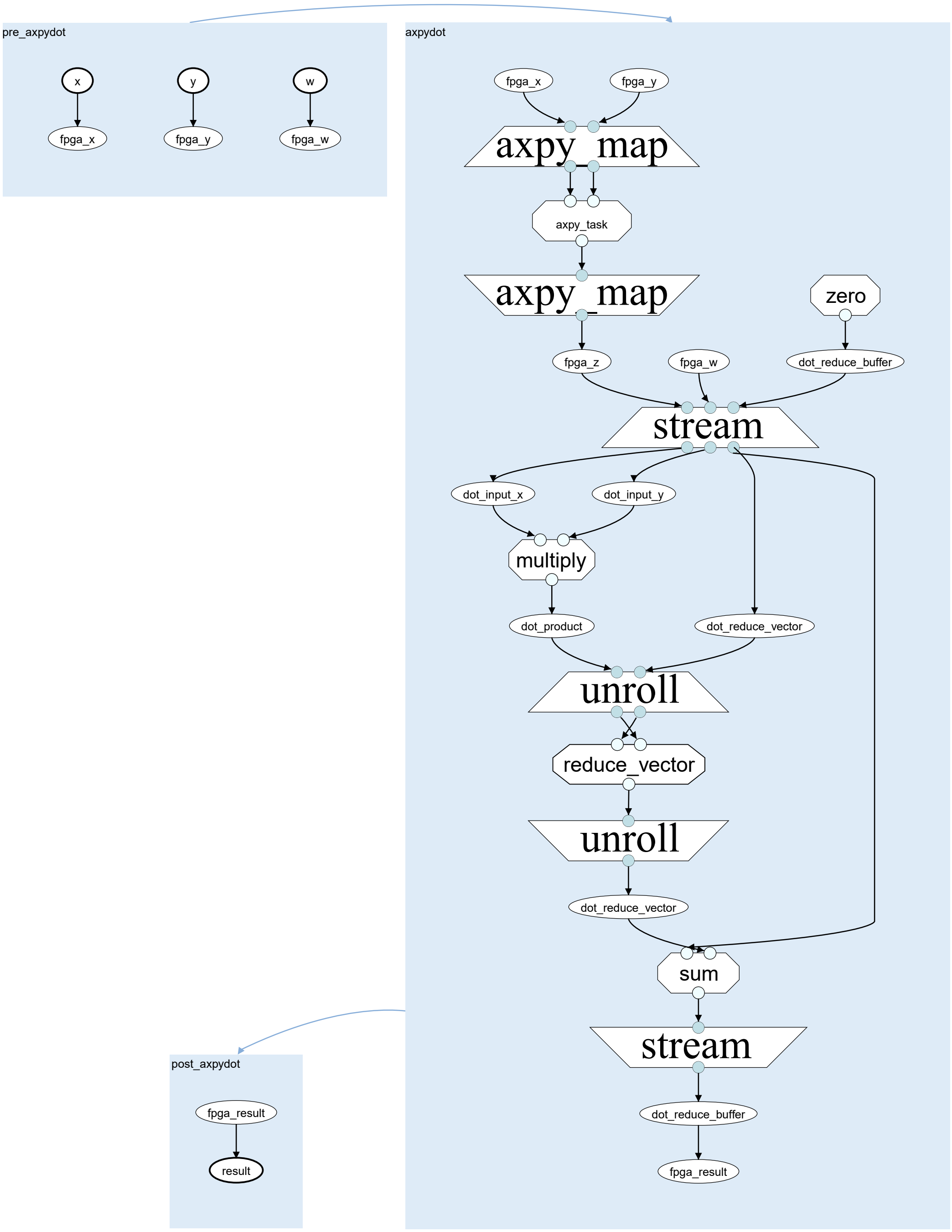


Figure 5.24: `AXPYDOT` expanded for Intel, accumulating into a single register.

performed after the main streaming phase, and sums up all values in the partial sums buffer into a single output, which is written to the output (again consuming $W-1$ adders. In a resource-constrained scenario, this could be reduced to a single adder without impacting the asymptotic runtime). The Intel specialization included in Fig. 5.24 instead accumulates into a single register, saving the partial buffer and additional reduction.

5.8.3 Transformation: Memory Access Extraction

The `StreamingMemory` transformation was implemented by Tal Ben-Nun, based on concepts developed in Chapter 2 and for FBLAS [41] by myself and Tiziano De Matteis.

When the memory access pattern of a certain computation is known ahead of time, it is often beneficial to "prefetch" data from memory and stream the data into the FPGA processing elements, as described in Sec. 2.4.1. Creating streaming accessors has many benefits, including exploiting burst accesses in memory controllers, doing in-memory reordering of data streams (Sec. 4.4.3), broadcasting off-chip memory to multiple processing elements, or implementing custom caching mechanisms. On the writing side, having a dedicated writer can prevent backpressure from flushing writes to memory.

DaCe provides the "`StreamingMemory`" transformation, which extracts the access pattern from an existing memory read or write into a separate reader/writer module. The transformation looks for all recurring access patterns of unique symbolic expressions. If the range accessed consists of one scalar or vector element, the transformation can be applied. It then extracts the read/write out of the computation by introducing another component that accesses the memory in the same order as the computation, and communicates between them via a Stream object. The corresponding outgoing/incoming dataflow edges are replaced by edges that access the stream instead.

If more than one PE uses the same memory access order, the transformation generates a single streaming component that connects one array node to multiple streams. In order to avoid deadlocks, the transformation also detects dependencies by computing reachability from the destinations/sources of the memlet paths (inherently given by the construction of the SDFG). If accesses are dependent, separate components are created, even for the same access pattern.

5.8.4 Transformation: Pipeline Fusion

The `StreamingComposition` *transformation was implemented by Tal Ben-Nun, based on concepts developed for FBLAS [41] by Tiziano De Matteis and myself.*

As convered extensively by FBLAS [41], fusing consecutive pipelines is a critical optimization for spatial architectures to transition from a centralized memory design accessing DRAM to on-chip dataflow, where data is moved directly across the chip between computations. As described in Sec. 2.2.4, fusing two consecutive pipelines reduces the minimum number of cycles required to evaluate them by $N + N$ to just N, assuming that they both run for N iterations and are fully pipelined with $I = 1$.

In the SDFG for `AXPYDOT` shown in Fig. 5.22 after expanding the two Library Nodes with FPGA implementations, the intermediate memory `z` is stored in off-chip memory, resulting in a round-trip through global memory between the `axpy` and `dot` operators.

The `StreamingComposition` can be used to convert a temporary buffer in off-chip memory into streaming communication directly between the two endpoints. This transformation is similar in structure to the memory access extraction described in the previous section. The current implementation (as of writing) checks for array nodes with in-degree and out-degree of one, which are read/written with equivalent access patterns that can be composed. To do so, the transformation traces the dataflow through Map scopes and nested SDFGs, canonicalizing the expressions found on the edges. If the read and write subsets match exactly, the result of the first computation can be streamed into the second. Similarly to `StreamingMemory`, we replace the memory access nodes and neighboring memlets with streams, converting global memory arrays into local streams. This procedure can eventually be extended to recognize more complex patterns, such as having multiple consumers connected to a single producer.

In Fig. 5.25 we show the `AXPYDOT` example after applying both Memory Access Extraction and Pipeline Fusion to the SDFG from Fig. 5.22. The `StreamingMemory` transformation is applied on all three input/output vector accesses, namely x, y, and w, and `StreamingComposition` is used to stream the `z`-vector directly between the `AXPY` and `DOT` components, avoiding accesses to off-chip memory and letting the two pipelines run in parallel.

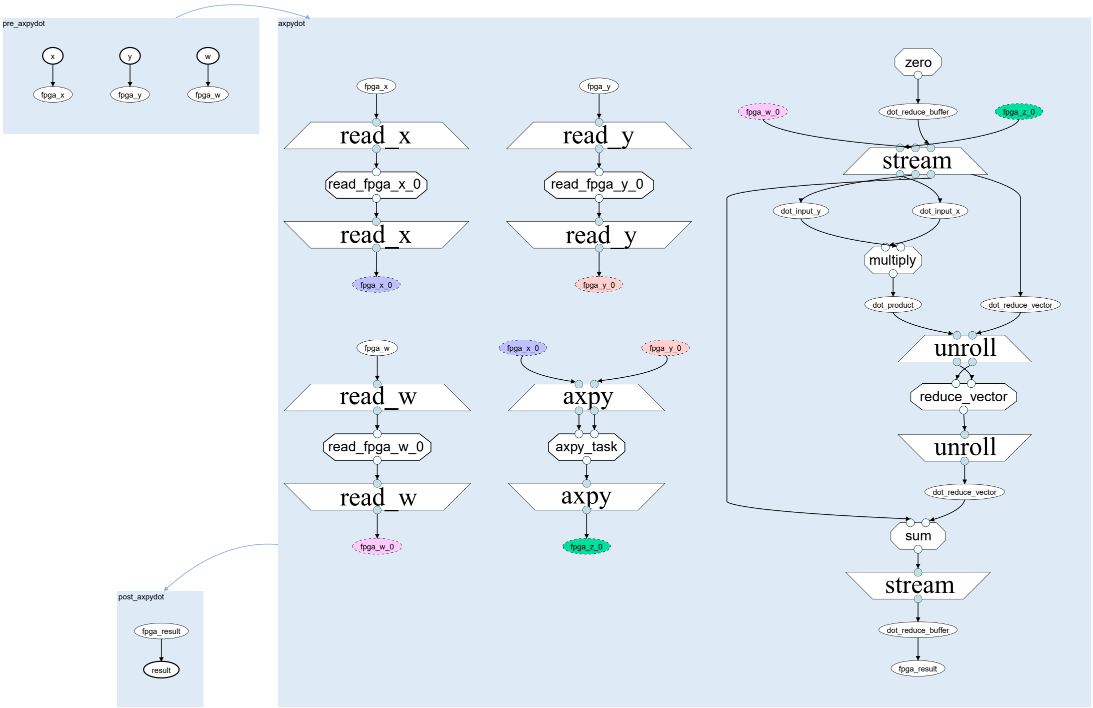


Figure 5.25: The `AXPYDOT` program after automatically extracting memory accesses into processing elements and streaming between operators. Streams are color-coded by name for pipeline visualization.

Table 5.1: Effect of optimizing `AXPYDOT` for FPGA execution with $N = 16 \cdot 10^6$.

	Runtime	**Effective bandwidth**	**Speedup**
Alveo U250 (non-streaming)	12.0 ms	21 GByte/s	-
Alveo U250 (streaming)	4.9 ms	52 GByte/s	2.42×
Stratix 10 (non-streaming)	11.4 ms	23 GByte/s	-
Stratix 10 (streaming)	3.8 ms	71 GByte/s	3.11×

5.8.5 `AXPYDOT` Evaluation

The following results are based on Library Node implementations that I have implemented, while the benchmarks were designed and measured by Tiziano De Matteis.

For the following experiment, we target the Alveo U250 for Xilinx execution and the BittWare 520N housing a Stratix 10 for Intel execution. Both FPGAs are equipped with 4× DDR4 memory banks, with a maximum total bandwidth of 77 GByte/s and 86 GByte/s for the Xilinx and Intel board, respectively.

We evaluate the performance of the composed `AXPYDOT` kernel before and after applying the two spatial transformations (extraction and fusion), but after applying the FPGA offloading and expanding the Library Nodes to the respective efficient FPGA implementations, in Tab. 5.1. The kernels are executed for single-precision floating point with 16-way vectorization to saturate the memory inferfaces, and are configured to access all four DDR4 banks. The effective bandwidth is computed as $4N$ times the operand size (we use 4 Bytes for single-precision floating point), which does *not* include the extra $2N$ memory accesses required to write `z` back to memory from `AXPY` and then reading it from `DOT`. The non-streamed implementations thus in practice achieve 33 GByte/s and 34 GByte/s for the Alveo U250 and the Stratix 10, but where $2N$ accesses are unnecessary.

Although we are only fusing two consecutive pipelines of size N, the speedup after transformations is greater than 2, because the total data volume accessed from off-chip memory becomes smaller by eliminating `w` from memory, reducing the runtime of the memory bound application; and because more DRAM banks can be used in parallel: 4 in the streaming version (accessing `x`, `y`, `w`, and `result` in parallel) versus 3 in the non-streaming version (accessing either `x`, `y`, and `z` in parallel in the first sequential phase, or `z`, `w`, and `result` in parallel in the second phase). Finally, moving the memory accesses out of the core computation allows memory accesses and computations to run asynchronously.

Fast FPGA implementations are provided for many BLAS Library Nodes in the DaCe repository, which can be fused into fully pipelined kernels using the methodology described above when their access patterns agree. By adding transformations specialized to target properties of spatial computing systems to the arsenal of DaCe, we further boost the productivity of the performance engineer, allowing him to transform input programs into efficient, streaming FPGA implementations.

5.9 Automatically Compiling NumPy to FPGAs

This section uses results from our SC'21 [164] paper led by Alexandros Ziogas, where the Xilinx results were collected by myself, and the Intel FPGA results were collected by Tiziano De Matteis, using elements of the DaCe frontend, transformations, and code generation that have received contributions from all authors.

In the following, we show how DaCe can be leveraged to target FPGAs directly from NumPy-based kernels. We offload programs using the DaCe transformation shown for `GEMV` in Sec. 5.5, and where relevant, the streaming transformations described in Sec. 5.8, then compile them for both Xilinx and Intel FPGAs. By embedding high-level operations from NumPy in the SDFG, such as the example shown in Sec. 5.5, we can preserve their semantics in order to emit pre-optimized library implementations specialized for FPGA execution, with no additional effort on the user's behalf.

To compile a Python function operating on NumPy arrays for FPGA using DaCe, such as the one shown in Fig. 5.26, it simply needs to be annotated with the `@dc.program` decorator. Optionally, in order to enable ahead-of-time (AoT) compilation – which is typically desired for FPGA programs – the data types of the function arguments must be specified. If this annotated function is called as-is, DaCe will transparently generate the SDFG, code generate C++ from the SDFG, compile, and run the resulting binary. The program can be run as is, or additional optimization opportunities exposed by transforming into the data-centric representation can be exploited by further transforming the SDFG.

By default, the IR generated from Fig. 5.26 will target CPU execution. Applications must first be transformed to be executed on the FPGA. This is achieved by a graph transformation that can be applied directly to the program:

```python
import numpy as np
import dace as dc

N = dc.symbol('N', dtype=dc.int64)

@dc.program
def gesummv(alpha: dc.float32, beta: dc.float32,
            A: dc.float32[N, N], B: dc.float32[N, N],
            x: dc.float32[N]):

    return alpha * A @ x + beta * B @ x
```

Figure 5.26: A simple Python program operating on NumPy arrays, annotated with a decorator and type hints on the arguments to be compiled with DaCe.

```python
1 sdfg = gesummv.to_sdfg()
2 sdfg.apply_transformation(FPGATransformSDFG)
3 sdfg(alpha, beta, A, B, x)
```

The first line produces the SDFG from the program `gesummv`, and the second line applies the transformation in-place to offload the computation to the FPGA. The transformation will instantiate all memory used in the computation as FPGA arrays, and insert host-to-device and device-to-host copies before and after the computation, respectively. The computation itself will be annotated for FPGA execution, and can be compiled for either FPGA vendor using the relevant backend. Executing the transformed program is no different from the initial program, and is done by calling the object as a function and providing the necessary arguments in the form of NumPy arrays and scalars, as shown on the third line. If the kernel was not compiled ahead of time, it will be compiled implicitly before executing the program, but will be cached for future use.

To demonstrate the coverage and versatility the DaCe framework, we consider **all 30 kernels** from the PolyBench [118] benchmark suite from their NumPy implementations in the NPBench [163] repository, building bitstreams for a Xilinx Alveo U250 board using Vitis 2020.2 and targeting the `xilinx_u250_xdma_201830_2` shell, and for the BittWare 520N accelerator housing an Intel Stratix 10 FPGA, using Quartus and the Intel OpenCL SDK for FPGAs version 20.3 and targeting the `p520_max_sg280h` shell. To the best of our knowledge, **no previous work has successfully built and run the full PolyBench suite, or indeed allowed FPGA kernels to be built and run directly from NumPy code**. The results included below build on the results published in the original

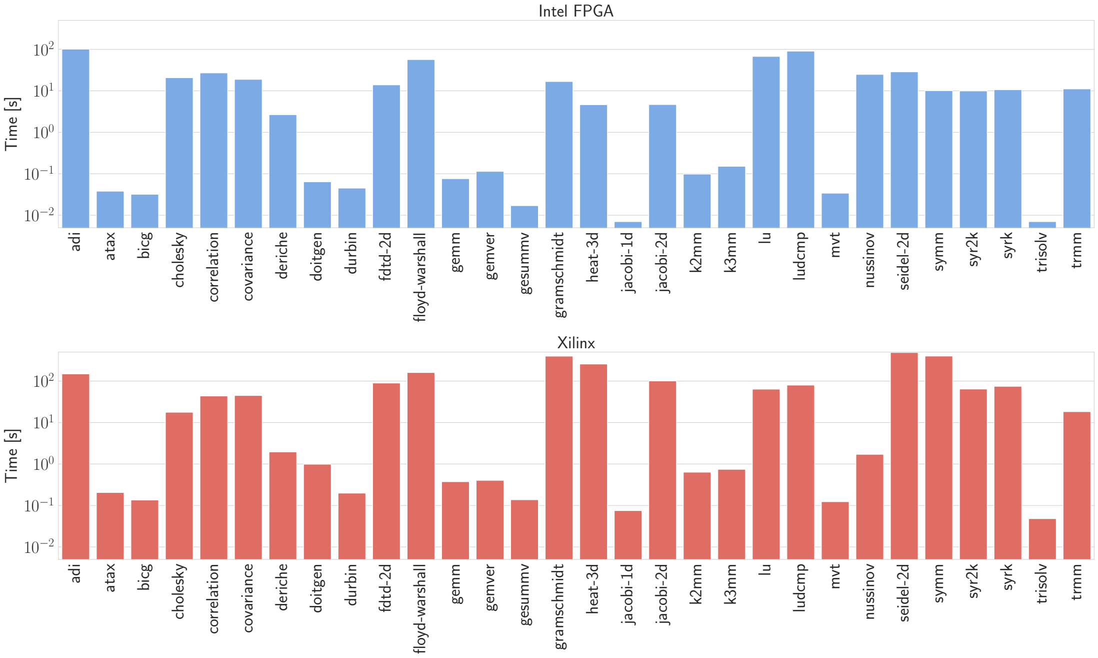


Figure 5.27: All 30 kernels from the PolyBench test suite, compiled from their NumPy implementation in the NPBench [163] project, to be executed on both Xilinx and Intel FPGAs, using the `LARGE` dataset size.

paper proposing SDFG representation [16], where all NumPy kernels were shown, but were compiled from manually authored SDFGs.

Results are shown in Fig. 5.27. The best results are obtained for benchmarks that rely on capturing high-level operations from the NumPy formulation, which are translated into Library Nodes, which can be then lowered into specialized hardware implementations. This is the case of benchmarks relying on BLAS-like routines, namely `atax`, `bicg`, `gemm`, `k2mm`, and `k3mm`. Here, matrix multiplications are instantiated as a high-throughput systolic array, and other linear algebra operations use efficient tiled implementations, exploiting device-specific features such as the presence of hardened floating-point units on the Stratix 10. In stencil-like applications (e.g., `jacobi1d`, `jacobi2d`, `heat3d`), Intel FPGA outperforms Xilinx, due to stencil pattern detection in the Intel compiler. Other benchmarks, such as `adi`, `floyd-warshall`, and `lu` suffer from heavy control flow and/or complicated dependencies that do not map well to dataflow on the device out of the box. The productivity benefit of Python as the input language over manual HLS designs can be quantified by the difference in verbosity of the code: The SLOC of the input Python codes in NPBench is an average of 28× and 29× less than the corresponding generated code targeting Intel FPGA and Xilinx, respectively, with the former requiring no boilerplate code for host/device interaction.

With this, we have demonstrated the versatility of the DaCe framework when targeting FPGAs, by building a large set of applications for execution on FPGAs from both vendors.

5.10 Summary

We introduced the data-centric SDFG representation, which exposes pure dataflow in the form of States where all data movement of a program is explicitly defined. This allows a skilled performance engineer to apply graph-based transformations, such as tiling or vectorization, to optimize the data movement of the program. Computations can even be offloaded to accelerators by simply changing the data storage location and schedule of computations on the graph nodes. SDFGs are obtained from one of several productive high-level frontends, and can be optimized independent of the frontend, resulting in a separation of concerns between program definition and its optimization.

The data-centric nature of SDFGs makes them a natural fit for mapping to spatial architectures, mapping data movement in the graph to data movement on the spatial device. SDFGs can implement streaming computation across fast on-chip memory, providing automatic memory management, portability between FPGA vendors, and seamless integration with host programs. By expressing hardware programs as SDFGs, performance engineers also gain access to the rich suite of data-centric optimizations in the DaCe toolbox, enabling knowledge transfer and code reuse between software and hardware optimization.

We show how we can compile a large benchmarks suite for execution on both Xilinx and Intel FPGAs directly from a NumPy representation, demonstrating the coverage and flexibility of the code generating backends and our ability to transform arbitrary computations to be offloaded to the FPGA. Along with the general purpose optimizations of DaCe, we also provide transformations specialized for spatial architectures, allowing the performance engineer to fuse consecutive pipelines to greatly boost the performance of kernels that can be composed in this way.

DaCe is published as an open source repository on GitHub (see page 83). As of writing, the repository has 192 stars, 50 forks, received contributions from 36 contributors, and has fueled a large body of research into data-centric optimization. Ziogas et al. [162] even scaled a large DaCe application to a full supercomputer, winning the 2019 Gordon Bell award. As the project develops, we hope that it continues to inspire great research and become the technical foundation of many more applications systems.

In the next chapter, we will exploit all the tools described here to further raise the level of abstraction to a domain-specific frontend, utilizing the multi-level design enabled by Library Nodes to embed domain information into the SDFG, using DaCe's powerful code generator to target both Xilinx and Intel FPGA platforms.

Chapter 6

StencilFlow: Domain-Specific Dataflow

This chapter is based on work published at CGO'21 [38], which was extended from work developed with Andreas Kuster for his bachelor thesis. Tal Ben-Nun and Dominic Hofer worked on extracting and canonicalizing stencil programs from the COSMO model for use in StencilFlow. Tiziano De Matteis developed the extension to DaCe required to run distributed experiments on Intel FPGAs, and ran the experiments on University of Paderborn's Noctua cluster. I co-authored the paper on SMI [40], which is used for distributed computations in StencilFlow. The StencilFlow stack is published as an open source repository[1] built on top of the DaCe framework, with support for both Xilinx and Intel FPGAs.

The temporal locality in iterative or dependent stencil computations is challenging to exploit on load/store architectures, as they require complex tiling schemes [106] and selective fusion of code segments [64, 65]. In contrast, exploiting this reuse via dataflow is intuitive, as consecutive stages can be pipelined and synchronized via their fine-grained dependencies [124]. Implementations of stencils achieving high performance on reconfigurable hardware often assume idealized iterative stencils, as this enables temporal blocking of consecutive timesteps [166, 27], which maps naturally to pipelined architectures.

In this chapter, we consider the challenging case of arbitrary stencil DAGs, motivated by their existence in numerical climate and weather prediction, where each node is a (potentially complex) stencil operation reading from one or more input memories, and writing its output to one or more consumers. As a motivating case study, we target an application from the Consortium for Small-scale Modeling (COSMO). The consortium consists of eight national weather services which aim to develop, improve and maintain a non-hydrostatic local area atmospheric model. The COSMO model is used for both operational [15, 150] and research [96, 117] applications by the members of the consortium and many universities worldwide. The stencils used in these simulations are dominated by series of *heterogeneous* stencil computations. Unlike the uniform codes often evaluated in high-performance computing research, these programs run many different stencil opera-

---

[1] https://github.com/spcl/stencilflow

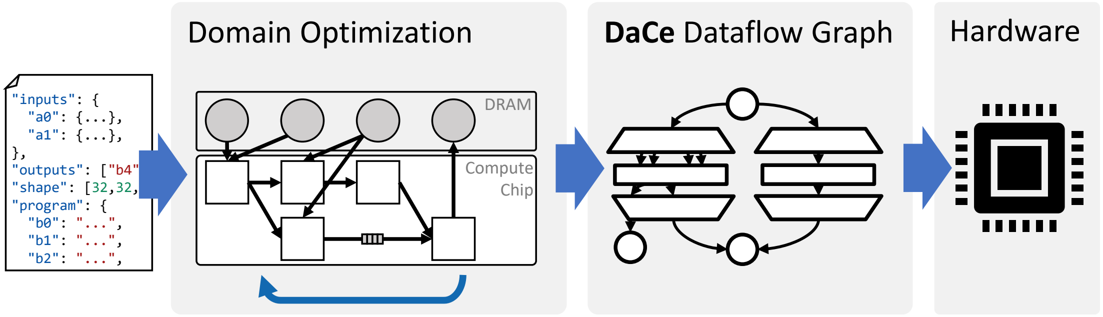


Figure 6.1: Overview of the StencilFlow end-to-end system.

tions on many different inputs of varying dimensionality, and exhibit complex dependency patterns between them.

We present a full-stack solution, from a high-level stencil DSL to low-level spatial program definitions, that are code generated for hardware execution, summarized in Fig. 6.1. We introduce a method that maps stencil programs to spatial architectures by using dataflow principles to form compositions that are deadlock free and maximize the number of active pipelines, based on an analysis of iteration patterns and the computational source code. *Fully code-generated* architectures emitted by StencilFlow evaluated on an Intel FPGA testbed reach 1.32 TOp/s and 4.18 TOp/s in single-device and multi-device experiments, respectively, which to the best of our knowledge is the *highest performance recorded for stencil programs executed on FPGA hardware to date*, and 765 GOp/s on a Xilinx accelerator platform, which to the best of our knowledge is the highest performance recorded for stencil programs executed on a Xilinx FPGA to date. The full source code is available on GitHub (see page 135), exposing productive high-level Python interfaces, while compiling to highly efficient hardware through the code-generating backend.

Clear separation of concerns at multiple levels of the stack is a key concept in our approach, as provided by SDFGs and the Library Node abstraction. An input program is formulated as a high-level DSL, constraining the program to an analyzable and optimizable form. Input programs are first optimized on a domain-specific level, where we can perform specialized transformations, such as fusing consecutive stencil nodes. Then, programs are lowered to standard DaCe dataflow nodes, where we can control and optimize for data movement. The dataflow representation is then specialized for the targeted architecture, and finally code-generated to be compiled and synthesized for hardware.

6.1 Definition of a Stencil Program

As the input format of StencilFlow, we define a “stencil program” as a *directed acyclic graph of stencil operations on a structured grid* (an example is shown in Fig. 6.2), where each node is either a stencil operation performed on the full output domain or a memory container, and edges are dependencies between stencils and memories: i.e., outputs produced by one stencil that are consumed by one or more other stencils, and/or are read from/written to memory. Each stencil takes one or more inputs, that are sourced either from off-chip memory, or fed by a previous stencil evaluation, and produces exactly one output. To support a broader class of computations present in weather models considered, we furthermore allow stencils to read from lower-dimensional inputs: e.g., a 3D stencil can read from a 2D, 1D, or even “0D” (scalar) arrays using subsets of its indices. A stencil node is defined by:

- A definition of each logical input that is read, which we refer to as “fields”, with a corresponding data type, and a sequence of offsets relative to the center (“field accesses”).
- A code segment describing the computation at each point in the iteration space, where only the specified input accesses (including 0D constants) can be used in computations. Since it is important to know the latency of computations, the code is restricted to be *analyzable* (i.e., no external data structures or external functions, with the exception of standard math functions). However, ternary functions/conditionals are allowed, **including data-dependent branches**.
- A series of boundary conditions, defining how out-of-bounds accesses should be handled.

Currently supported boundary conditions include: **constant**, where out of bounds accesses are replaced with a given constant value; **copy**, where out of bounds accesses are placed by the value at offset 0 in all dimensions (the “center” value); and **shrink**, where all computed values that read out of bounds values are simply ignored in the output. The former two are specified per input, whereas shrink is specified on the output.

To facilitate productive definition of stencil programs, we define a simple JSON-based input format, which only requires the minimum amount of information necessary to instantiate the stencil DAG to be specified explicitly. An example is shown in Lst. 27. In practice, the definition must additionally provide data sources for each input field. Stencil programs can have 1, 2, or 3 dimensions, but assume all stencils iterate over the same iteration space (although they can have variable constant offsets into the output field).

```
{ "inputs": {"a0": {"dtype": "float32",
                    "dims": ["i","j","k"]},
             "a1": {"dtype": "float32",
                    "dims": ["i","j","k"]},
             "a2": {"dtype": "float32",
                    "dims": ["i","k"]} },
  "outputs": ["b4"], "shape": [32, 32, 32],
  "program": {
    "b0": {"code": "a0[i,j,k] + a1[i,j,k]",
           "boundary_condition": {
               "a0": {"type": "constant",
                      "value": 1},
               "a1": {"type": "copy"} } },
    "b1": {"code": "0.5*(b0[i,j,k] + a2[i,k])",
           "boundary_condition": "shrink"},
    "b2": {"code": "0.5*(b0[i,j,k] - a2[i,k])",
           "boundary_condition": "shrink"},
    "b3": {"code": "b1[i-1,j,k] + b1[i+1,j k]",
           "boundary_condition": "shrink"},
    "b4": {"code": "b2[i,j,k] + b3[i,j,k]",
           "boundary_condition": "shrink"} } }
```

Listing 27: Program description.

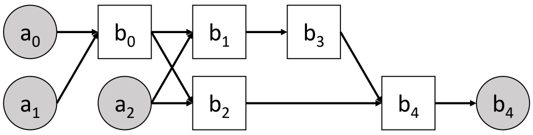


Figure 6.2: Corresponding DAG.

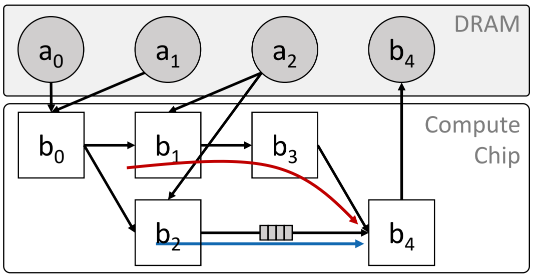


Figure 6.3: Hardware mapping.

6.2 Mapping to Distributed Hardware

There is a substantial body of previous work on mapping single stencil operations to reconfigurable hardware [166, 124, 53, 27], where high performance is achieved by chaining many consecutive timesteps together as a rich source of temporal locality. Some of this methodology carries over to the more general scenario we consider here, but we must additionally consider forks and joins in the stencil program, inputs and outputs shared by multiple producers and consumers, heterogeneity and complexity in stencil computations, and mapping the graph to multiple devices.

6.2.1 Mapping to Hardware

For our hardware mapping, *we work from the base assumption that every stencil operation in the dependency graph is mapped to simultaneous dedicated logic* (stencil *units*/operators), even if this requires the design to span multiple devices. All stencil operations are scheduled simultaneously, operating in a **fully pipeline parallel manner**. In this scenario, production and consumption rates are identical across the dataflow graph, allowing the runtime to be modeled as a single, deep pipeline (described in Sec. 6.7.1).

Each stencil unit executes a pipeline, which processes a number of cells equal to the product of the input dimensions, where logic required to handle out-of-bound accesses is predicated into the pipeline. The next cell is evaluated as soon as all inputs required *for that cell* are ready. This way, all dependencies between stencils become fine-grained on a per-cell level. This spatial computing view is distinct from the load/store view, as we default to *perfect data reuse* (i.e., we exploit all available temporal locality). In contrast, the efficiency of computations on load/store architectures relies on maintaining a task granularity suitable for the architecture (large number of identical threads on GPU, small number of large tasks on CPU). Kernel fusion is thus a critical optimization to achieve the right task granularity [65] for performance through spatial locality, whereas *StencilFlow programs are executed in a fully "fused" schedule*, but are instead concerned with satisfying the off-chip and on-chip memory constraints (fusing stencil operators takes a different meaning, described in Sec. 6.4.1).

Inputs are provided to each stencil unit through on-chip channels with a *compile-time fixed size*, where the producer can either be another stencil unit (i.e., a dependency), or a memory unit reading directly from off-chip memory. If one or more inputs are not ready, the pipeline must stall while waiting for the remaining inputs to arrive. For any DAG that is not a multi-tree, this can result in deadlocks if channel capacities are insufficient to buffer inputs ready early until inputs ready later arrive, due to the circular dependency implied by each data exchange requiring the receiver and sender to not be *full* and not be *empty*, respectively. We must thus take all paths through the DAG into account when deciding the size of buffers between dependencies.

In the example shown in Fig. 6.4, the stencil unit computing `C` requires data from both stencil units `A` and `B` to begin streaming. The results streamed out of `A` are also required by `B`. On the left hand side, `C` is waiting for data from `B` (i.e., for the data stream to not be empty), `B` is waiting for additional data from `A`, and `A` is waiting for `C` to accept the data (i.e., for the data stream to not be full), thus forming a circular dependency. Without additional buffering, this results in a deadlock. By adding an appropriate buffer between `A` and `C` (right hand side), we can inject sufficient credits to tolerate the delay induced by the path through `B`. We describe how StencilFlow computes the buffer depths required to prevent deadlocks and ensure continuous streaming operation in Sec. 6.3.2.

6.2.2 Mapping to the Distributed Setting

To scale beyond the off-chip memory bandwidth, on-chip memory capacity, and logic resources available on a single chip, we let designs scale to multiple devices. For modeling

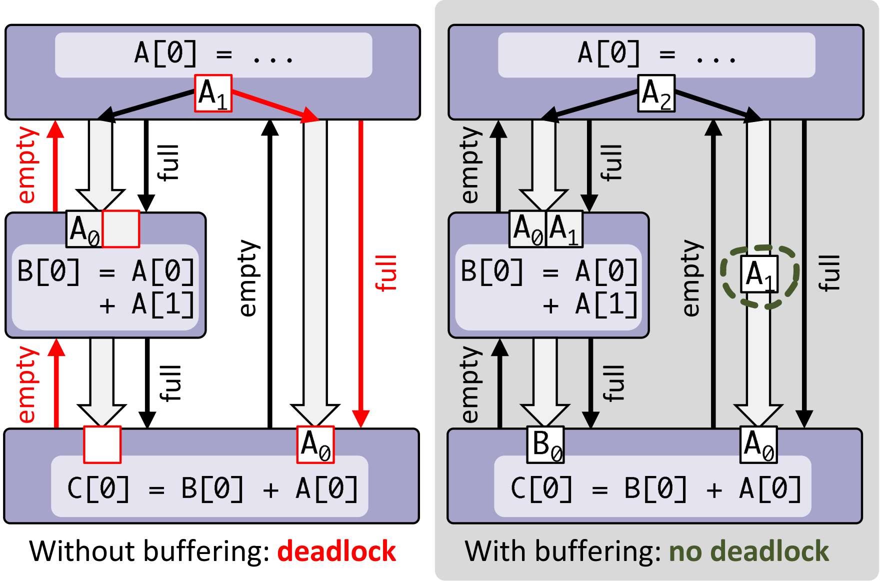


Figure 6.4: Preventing deadlocks by injecting buffers.

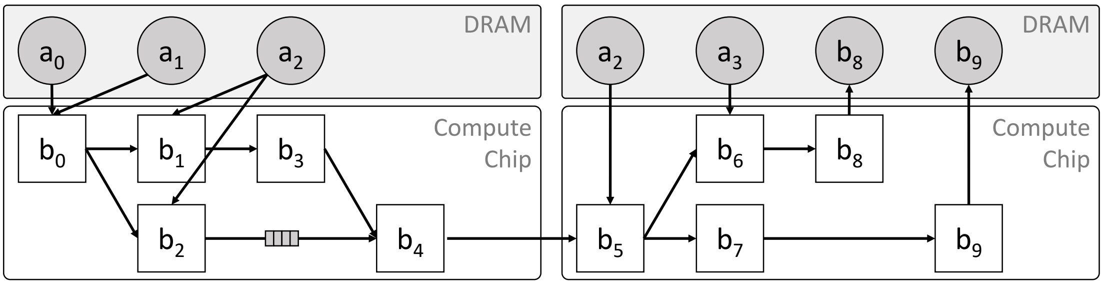


Figure 6.5: Stencil program spanning two devices.

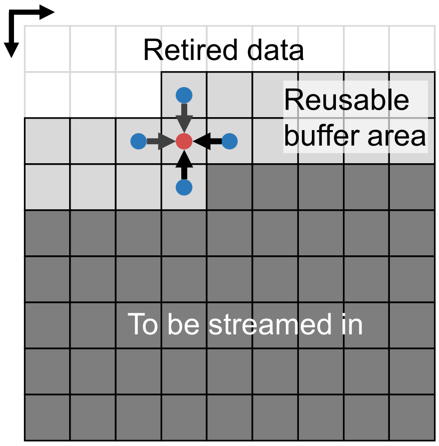


Figure 6.6: Internal data reuse buffer.

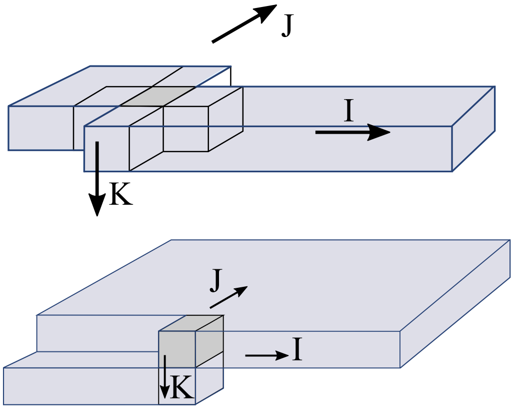


Figure 6.7: Varying buffer shapes.

and code generation, this means that certain inter-stencil connections will cross devices, and thus imply communication across the network. Furthermore, data located in off-chip memory must be present on any device that accesses it, implying potential replication to multiple devices that require it. In the example shown in Fig. 6.5, a_2 is accessed by stencils on either device, requiring it to exist in both DRAM memories.

To implement inter-node communication in practice, we leverage the Streaming Message Interface [40] (SMI), which exposes communication as channels with FIFO semantics, resulting in inter-node communication being nearly identical to intra-device communication between stencils in the generated code. With the target in mind, the following will describe the program analysis, and the central components of the StencilFlow stack, required to build these spatial architectures.

6.3 From DAG to Dataflow

The StencilFlow framework analyzes the stencil DAG, and uses this to construct a dataflow graph that maps to efficient hardware. Data reuse happens both internally in each stencil, facilitated by “internal buffers”, and on the edges between stencil nodes, referred to as “delay buffers”.

6.3.1 Internal Buffers for Intra-Stencil Reuse

The most straightforward source of temporal locality comes from within each stencil operation, where the same input field is often accessed at multiple offsets relative to the center, illustrated in Fig. 6.6 for accesses $\{[-1,0],[0,-1],[0,1],[1,0]\}$ in a 2D iteration space. Furthermore, in the global dataflow setting, the core assumption of StencilFlow is that data should only be loaded once, streaming directly between kernels without going through off-chip memory.

A stencil node has 0 or 1 internal buffers per field accessed, depending on whether there are multiple accesses to the given field within the stencil. The size of each buffer is determined by the *largest distance between any two offsets in memory order, plus one* (or plus the vector width, in the case of vectorized kernels) in the stencil iteration space: e.g., in a 3D iteration space of shape $\{K,J,I\}$, two accesses `a[0, 1, 0]` and `a[0, -1, 0]` require buffering two 1D rows ($2I+W$ elements, where W is the vector width), while two accesses `b[0, 0, 0]` and `b[1, 0, 0]` require buffering a 2D slice ($2IJ+W$), shown in Fig. 6.7 top and bottom, respectively. In general, buffers sizes can be up to a constant number of $(D-1)$-dimensional slices for a D-dimensional stencil.

StencilFlow computes the internal buffer size for each field, for each stencil, independently. However, the schedule for when the pipeline starts writing each buffer is dependent on the other fields accessed. For example, if a stencil reads multiple fields with internal buffer sizes $\{B_1,\ldots,B_F\}$, each internal buffer can be only start to be filled after the first $B_i - \max\{B_1,\ldots,B_F\}$ iterations (the largest buffer(s) will always start reading immediately), so it is synchronized with the other fields. Additional accesses *in between* the "highest" and "lowest" offset in memory order do not affect the total buffer size, although they can affect the buffer implementation in practice by adding more parallel accesses into the buffer.

Filling the internal buffers also affects the latency, and transitively the runtime, of the stencil program. A stencil node cannot begin computations before all operands are available, which only happens once all internal buffers have been filled. As the size of buffers is exactly the distance between the lowest and highest accessed index in order of the stencil iteration space, the *initialization phase* of a stencil is given by $\max\{B_1,\ldots,B_F\}$, which is crucial to the delay buffer calculation described in the following.

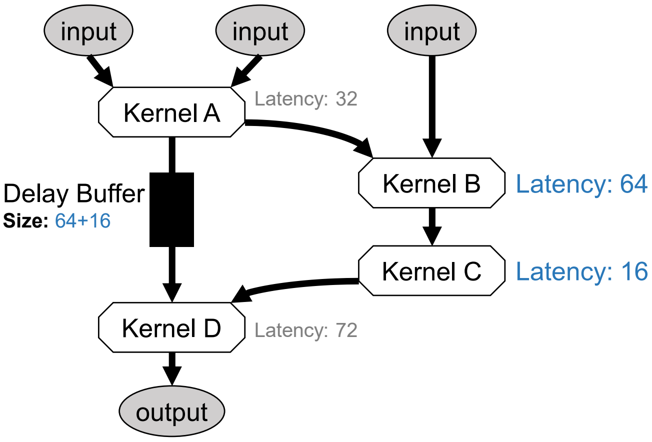


Figure 6.8: Delay buffers on edges enable reuse and deadlock freedom.

6.3.2 Delay Buffers for Inter-Stencil Reuse

Edges between stencils in the DAG enable data reuse by replacing expensive round-trips to off-chip memory with direct dataflow. Furthermore, if multiple stencils require data from the same input field, it is sufficient to read it from memory once, and stream the data to all stencils requiring it. StencilFlow exploits all such opportunities, while preventing the deadlock scenario illustrated in Fig. 6.4. This requires synchronizing inputs to consumers by adding buffers that delay the data (i.e., inject sufficient credits) until all inputs are ready without blocking the producer(s). We annotate these delay buffers on edges in the dataflow graph, corresponding to FIFO channel depths.

There are two factors that determine delays in the DAG. First, the AST formed by computation of a stencil operation forms another DAG, whose critical path adds a delay between a sequence of inputs entering and exiting the pipeline. Computing the critical path requires latency information for each operation performed, which is both type and architecture dependent. As a result, these latencies can be provided as configuration to the framework, and default to conservative values to account for the worst case scenario. We note that these delays are typically small ($<$100 cycles), and do not contribute significantly to the overall fast memory usage, even when conservatively overestimated. More importantly, delays occur in the initialization phases within each stencil, where internal buffers are being filled before enough data is available to start computations. Each stencil node in the stencil program will contribute $\max\{B_1, \ldots, B_F\}$ elements to this delay, where $\{B_1, \ldots, B_F\}$ is the set of F internal buffer sizes for the given stencil.

To determine the size of delay buffers on the edges arriving at a given node, we traverse the DAG backwards from the node, computing the latency contributions along all possible paths, from all possible source nodes, and for each edge, *including the contribution of the initialization phase of the node itself*, recording the highest delay encountered per edge. The buffer size on each edge is then the highest delay found for that edge, subtracted from the highest delay found across *all* edges (it follows that each node will have at least one incoming edge with delay size zero). Similar to internal buffers, the maximum size of delay buffers is proportional to the size of a $(D-1)$-dimensional slice of the iteration space. An example of annotated delay buffers in shown in Fig. 6.8.

For scenarios where there is sufficient reuse to not be memory bound, there could be a potential trade-off between off-chip memory bandwidth spent on doing round-trips to memory, and the resource consumption of delay buffers required to move the access into fast memory: We could choose one or the other for each connection. We do not consider this trade-off for two reasons. First, the fast memory consumption can ultimately be bounded using tiling, making it an attractive choice even when the capacity requirements would be large. Second, the weather simulation kernels we consider in this work are predominantly memory bound, consistently making moving communication into on-chip delay buffers the superior choice.

6.3.3 Vectorization

When insufficient reuse is present in a target program, we can employ vectorization to increase parallelism and memory bandwidth utilization, in order to approach a compute logic or memory bandwidth bound. To this end, we allow StencilFlow input programs to specify a vectorization factor, which will not only affect the generated hardware, but also the dataflow analysis. Vectorizing by a factor of W reduces the number of iterations in the inner loop of all stencils in the program by a factor of W, which affects the size of initialization phases, and transitively the delay buffers in the system. In addition to directly increasing the bandwidth requirement and parallelism in the program, vectorization can also have the subtler effect of coarsening stencil nodes, increasing the ratio of "useful" compute logic to overhead logic. We can thus also use vectorization in time tiling-like scenarios to coarsen simple stencils and increase the achievable performance.

Once the stencil program has been enriched with the appropriate internal buffer and delay buffer sizes, the resulting graph is emitted to the data-centric backend for domain-specific and low-level optimizations.

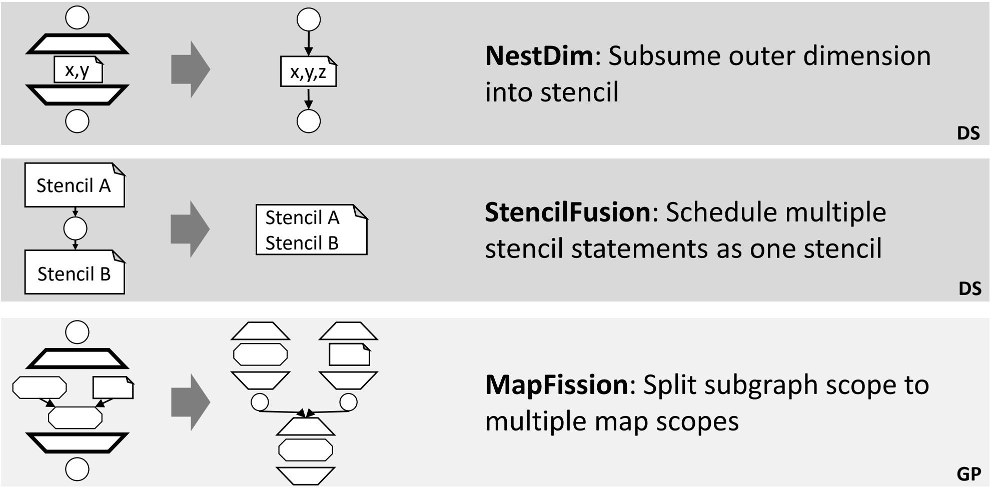

x,y
x,y,z
NestDim: Subsume outer dimension into stencil
DS
Stencil A
Stencil B
Stencil A
Stencil B
StencilFusion: Schedule multiple stencil statements as one stencil
DS
MapFission: Split subgraph scope to multiple map scopes
GP

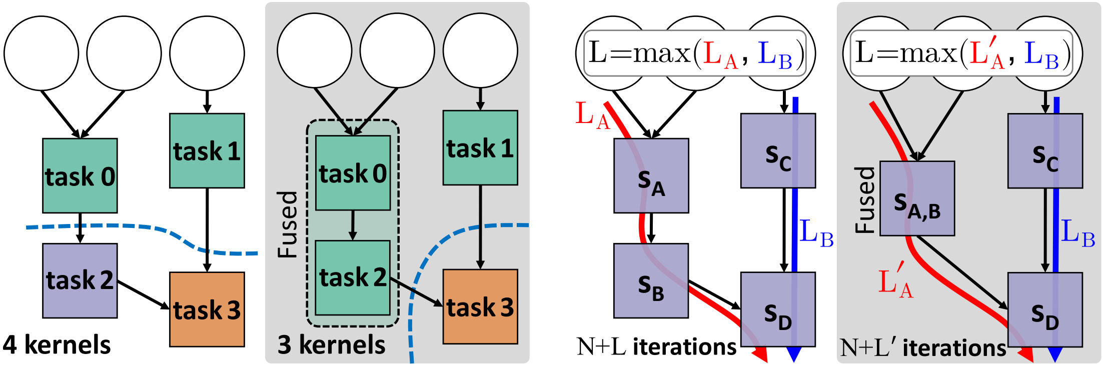
Figure 6.9: Transformations used.
(DS: Domain-Specific, GP: General-Purpose).

6.4 Data-Centric Abstract Representation

To support the functionality required by StencilFlow, we introduce a Library Node representing stencil computations, provide three new transformations, and extend the Map scope concept with specialized Pipeline scopes that provide a convenient abstraction for initialization/draining phases in pipelined computations. The StencilFlow-specific Library Node `Stencil` was developed as the primary computational driver for StencilFlow, and will be used extensively throughout the following. Since the high-level semantics of Library Node types are known, they allow performance engineers to develop domain-specific transformations, such as algebraic contractions (e.g., double transposition). With Library Node expansions potentially containing other Library Nodes, multi-level coarsening and transformations are thus enabled in SDFGs, inspired by the MLIR [97] stack (see Sec. 5.5).

As a useful shorthand for pipelined iteration spaces, we introduce the *Pipeline Scope*, augmented with information on initialization and draining phases, to easily allow the programmer to inject specialized behavior during initialization, streaming, and draining phases. For StencilFlow, this allows encoding the internal buffer initialization phase, and draining phases where results are still being computed only using data present in local buffers, thus omitting reads from inputs.

With the domain-specific concepts enabled by Library Nodes, we are able to develop trans-



(a) Load/store stencil fusion. (b) Stencil fusion in StencilFlow.

Figure 6.10: Unlike load/store fusion, spatial fusion only reduces latency.

formations for stencil programs on reconfigurable hardware. We develop both domain-specific and a general-purpose transformation, summarized in Fig. 6.9. `NestDim` reschedules stencil computations by taking multiple, parametrically-parallel stencils and creating one stencil, which can be mapped into different schedules on hardware. `StencilFusion` schedules multiple dependent stencils as one stencil with multiple statements, differing from standard map fusion by taking boundary conditions and redundancy into account. For general-purpose transformations, we add the `MapFission` transformation, which splits a parallel subgraph into multiple parallel subgraphs (which can in turn be rescheduled), introducing temporary storage between the subgraph components. The `NestDim` and `MapFission` transformations are used as a tool to extract stencil programs from existing SDFGs to analyze them in StencilFlow, while `StencilFusion` is an optimization for both load/store and spatial architectures, described in the context of StencilFlow below.

6.4.1 Spatial Stencil Fusion

On load/store architectures, fusing consecutive stencils is used to increase performance by improving data locality, reducing write/read roundtrips from off-chip memory, and reducing context scheduling overhead [72]. When applying the transformation on StencilFlow dataflow graphs, the effect is somewhat different, as the schedule of the spatial architecture is already fully “fused” into a global pipeline. Instead, fusing stencils has the following effects:

- The critical path through the program can be reduced by combining the initialization phases (see Sec. 6.3.1) of two consecutive stencils.

- Internal buffers for the same input field are combined into a single internal buffer.
- Multiple smaller delay buffers can be combined into fewer, larger buffers, which affects hardware utilization, depending on the granularity of on-chip memory on the target platform.
- Combined code sections increase the opportunity for common subexpression elimination by the optimizing compiler.
- Coarser stencil nodes increase the ratio of "useful" logic to the number of pipelines instantiated, which can affect spatial resource overhead.

The difference between fusing tasks on load/store architectures and the spatial fusion performed here is illustrated in Fig. 6.10. On load/store architectures (Fig. 6.10a), the total number of scheduled kernels is reduced when fusing task 0 and task 2 into a single kernel. In Fig. 6.10b, all operators are already scheduled in parallel, but the initialization latency can be reduced *if the fused nodes* s_A *and* s_B *are on the critical path.*

When canonicalizing an input program obtained from the COSMO suite, we define a collection of heuristics for fusing two stencils so that these effects are observed. Firstly, the necessary conditions for fusion are checked, namely that the two stencils operate on the same data shape (correlating to iteration space) and that they have the same StencilFlow boundary condition definitions. Then, we only consider stencils that are connected by one data container node u with $\deg(u)=2$, in order to ensure that all stencils (fused or otherwise) have a single output. Finally, we ensure no other instances of u exist in other states, so that it can be completely removed from the graph without adding an extra write to off-chip memory.

For the experiments in the following, we perform aggressive stencil fusion of input programs, as this is observed to reduce overall logic through the coarsening of stencil nodes, and slightly reduces runtime by pruning initialization latencies.

6.5 Architecture Generation

StencilFlow relies on DaCe backends to generate the final kernel code, which is passed to an optimizing compiler. For the experiments performed in the following, we primarily target the Intel FPGA SDK for OpenCL backend [32], but also include results collected on a Xilinx Alveo accelerator card. DaCe automatically performs necessary annotations for pipelining, unrolling, and coalescing loops emitted from parametric maps in the dataflow graph, splits parallel sections into processing elements (i.e., OpenCL kernels), annotates buffer depth properties for channels, declares kernels as `autorun` when possible, inlines constants, and performs conversions between vectorized and non-vectorized data types. Host code necessary to interface with the kernel and the necessary memory copies are generated, and the final program can be called by using the high-level Python interface. By using DaCe for code generation and by using the Library Node abstraction for stencil computations, we can support both Xilinx and Intel FPGAs. To eventually emit RTL code directly, or to target other spatial systems entirely, will only require adapting the stencil Library Node expansion, provided that support for the desired architecture is present in/added to the DaCe framework.

When targeting the FPGA backends, delay buffers are represented as DaCe streams with a given buffer size, which are mapped to the Intel OpenCL `channel` abstraction and Vitis HLS stream objects, respectively, that are in turn mapped to FIFOs in hardware. The computation itself is represented by stencil Library Nodes, which will be expanded to different subgraphs depending on whether Intel or Xilinx is targeted.

This full graph returned by either Library Node expansion eill be wrapped in a parametric scope that defines the iteration space of the stencil program, which is fully pipelined, such that all three phases are executed in a pipeline parallel manner. The input and output streams (drawn with dashed borders) are connected to the appropriate producers and consumers in the global dataflow graph. Source nodes are instantiated as dedicated prefetchers that can read ahead of computations, and dedicated writers are instantiated at sink nodes that can buffer data while waiting for DRAM writes.

6.5.1 Intel Stencil Node Expansion

We target the shift register pattern in Intel's OpenCL compiler to efficiently implement internal buffers within each stencil node. To achieve this in DaCe, a data container spanning the full width of each internal buffer is created, injecting and shifting elements every cycle. "Tap" points (constant offset accesses) into the array are then connected to the stencil, where offsets are generated from the distance between accesses flattened into a 1D iteration space.

The processing done per cell in a stencil Library Node expanded for Intel FPGA is shown in Fig. 6.11. The graph contains three consecutive components: a **shift** phase, containing a fully unrolled Map scope where a Tasklet shifts each entry of the shift register memory by the vectorization width to $i{+}W$; an **update** phase, where new values are read from the

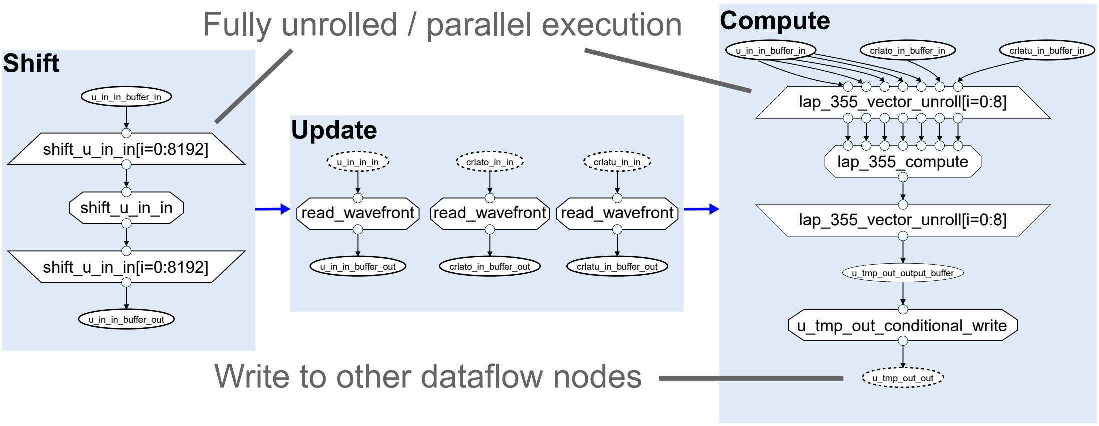


Figure 6.11: Stencil Library Node expansion targeting Intel FPGAs.

input channel into the front of each shift register by a tasklet; and a **compute** phase, where the buffers are accessed at all tap points and fed to the main computation tasklet, which is parametrically unrolled to treat each element in the vector with potentially different boundary conditions, and passes through another tasklet that conditionally writes the output stream if the stencil is not in the initialization phase.

6.5.2 Xilinx Stencil Node Expansion

Xilinx does not expose a scalable shift register abstraction. Instead, the buffers between each stencil access must be deduced, instantiated, and accessed explicitly, which is challenging due to vectorization resulting in non-aligned accesses into the vector strides. Fig. 6.12 shows a Xilinx expansion, implementing the same pattern as Fig. 6.11, but without the aid of shift registers. In this example, a 4-point stencil has four access points, which with 4-way vectorization requires 14 unique offset. Offsets are computed as the distance from the "earliest access" relative to the iteration pattern. These offsets are translated into major indices (which buffer is accessed, according to the vector stride) and minor indices (indices into each accessed vector). The major indices become the "access points" into the vectorized buffers, resulting in 4 buffers for this example. At each iteration, the buffers are read at a cyclic index along with the value from the wavefront. Because the kernel is vectorized, a full vector must be read from each buffer first, after which the individual scalar elements can be extracted by the kernel (implemented by the 14 dataflow edges between the buffers and the computation `b0`). Finally, each buffer is updated with the value from the *following* access point, and the front access point (i.e., highest flattened

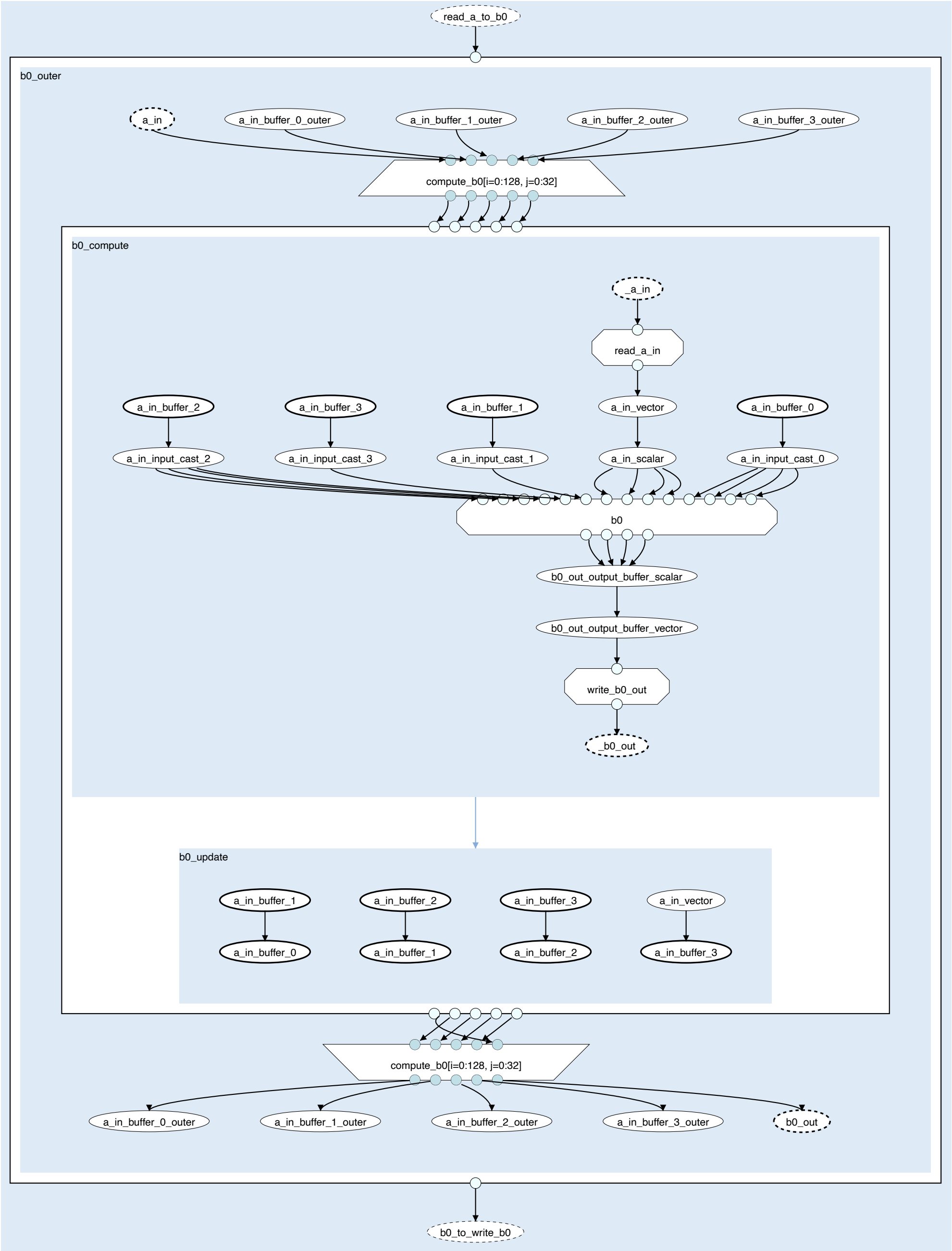


Figure 6.12: Stencil Library Node expansion targeting Xilinx FPGAs.

index) is updated from the wavefront.

6.5.3 Reference Expansion

While exploring CPU and GPU performance is out of the scope of this work, we exploit the Library Node abstraction to also generate naive CPU-executed graphs for reference, where stencil evaluations are executed sequentially in topological order (i.e, no fusion or parallelism between stencil evaluations), which we can use to verify the generated hardware kernels.

6.5.4 Generating Distributed Programs

To code-generate distributed implementations, we integrate an OpenCL implementation of the *Streaming Message Interface* [40] (SMI) into the DaCe backend. SMI extends HLS with a distributed memory programming model for reconfigurable hardware that unifies message passing with pipelined stream-based communication of data, such that cross-chip communication is expressed the same way as on-chip communication.

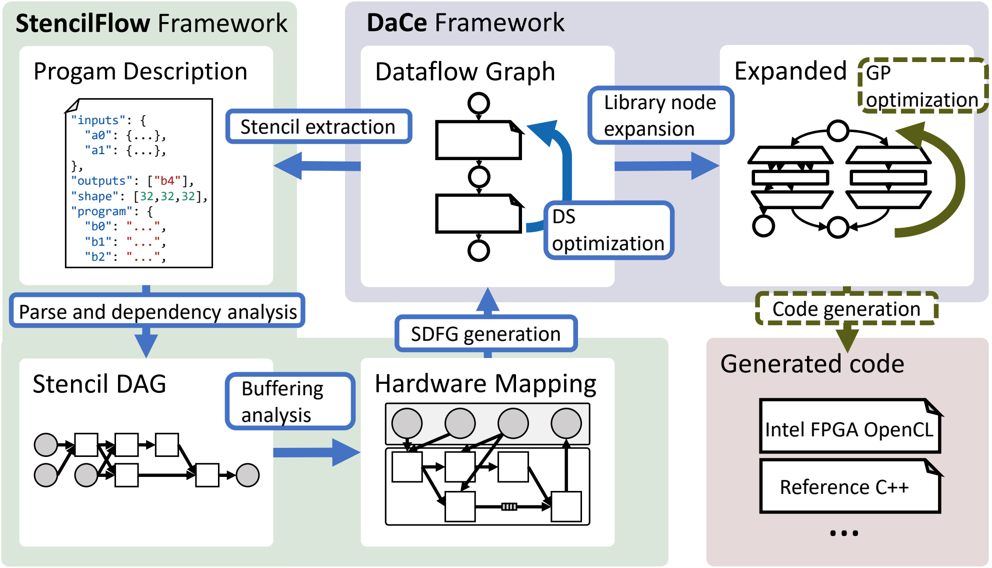
When a stencil program spans multiple devices, the computation running on each device is represented by a separate DaCe program, as it will compile to separate bitstreams that must be configured to each device in the sequence. Devices communicate via *remote streams*, which are DaCe streams annotated with having a source/destination located on a different device, which will trigger the SMI backend to code generate the relevant networking code and emit streaming message communication.

If multiple network connections are present between two endpoints, SMI can split a communication stream into two or more substreams following different channels across the network, and recombine them at the other end, allowing for a multiplicative increase in achievable bandwidth. In StencilFlow, we exploit this to increase the vectorization width and number of channels spanning across devices.

6.6 Workflow and Artifacts

To summarize the stack described throughout the above, an overview of the StencilFlow workflow is shown in Fig. 6.13. The StencilFlow framework is a pure Python code (≈5300 SLOC at the time of writing).

The input program to StencilFlow can either be given as the JSON-based program description described in Sec. 6.1, or as a DaCe dataflow graph containing domain-specific



```
"inputs": {
  "a0": {...},
  "a1": {...},
},
"outputs": ["b4"],
"shape": [32,32,32],
"program": {
  "b0": "...",
  "b1": "...",
  "b2": "...",
```

Figure 6.13: Workflow overview, with code artifacts annotated on arrows. Dashed outline indicates an existing feature that was extended.

stencil nodes. In the latter case, we developed software that performs canonicalization passes to the DaCe graph, before extracting the stencil pattern to the standard program description format. This allows us to read in external programs, which will be required for the case study in Sec. 6.8.

StencilFlow can directly run the stencil program from the input description, transparently executing parsing, dependency analysis, buffering analysis, SDFG generation, domain-specific optimization, Library Node expansion, general purpose optimization, code generation, compilation of the host code, compilation of the kernel (requiring the full synthesis, placement and routing flow if FPGAs are targeted), execution of the program, and validation of results.

6.7 Benchmarks

We benchmark the architectures emitted by StencilFlow to establish the highest achievable performance and bandwidth on a testbed platform, which we can use to analyze the characteristics required to push performance of stencil applications.

6.7.1 Computing Expected Runtime

We annotate benchmarks with the "expected" runtime, given by the lower bound on number of cycles required to evaluate the program, assuming all data is available at the earliest possible cycle. Because the full stencil DAG is executed in a pipeline parallel manner, we can model the runtime as a single, global pipeline. It is generally true for a pipelined circuit that the number of cycles required to process N inputs follows Eq. 2.1. All architectures emitted by StencilFlow are fully pipelined, so we fix $I{=}1\,\frac{\text{cycles}}{\text{operand}}$. N is the product of the domain dimensions (number of iterations in the iteration space), divided by the vectorization width W when applicable. L is computed from the circuit latency and initialization delay described in Sec. 6.3.2. N and L compose differently: N covers the streaming section where stencils can operate in a pipeline parallel fashion, whereas L covers the initialization phase where stencil units are not feeding downstream consumers. The depth of the DAG thus adversely affects the performance upper bound, while the size of the domain affects it favorably, increasing the relatively number of "useful" cycles to cycles spent in initialization. Since L is only proportional to $D{-}1$ or fewer dimensions (see Sec. 6.3.2), it becomes negligible when the domain is large relative to the depth of the stencil DAG. However, we include it when computing expected runtime for completeness.

6.7.2 Experimental Platforms

To evaluate the efficiency of dataflow architectures laid out by StencilFlow, we map them to state-of-the-art FPGA platforms from both major vendors. Our benchmarks focus on 32-bit precision, as this is used in production by our motivating weather simulation example, and because this precision is supported natively on the Stratix 10. However, all parts of the StencilFlow stack support any data type recognized by the underlying compiler, including double precision floating point and integer types.

For Intel FPGA execution, we target the BittWare 520N PCI-e attached board, with an Intel GX 2800 Stratix 10 processor, 4 DDR4 memory banks with a combined peak bandwidth of 76.8 GByte/s, and four network-attached QSFP ports rated at 40 Gbit/s. The annotated OpenCL code generated from DaCe is compiled with version 19.1.0 of the Intel FPGA OpenCL SDK and Quartus compiler, targeting the `p520_max_sg280l` shell offered by BittWare. This shell supports networking via OpenCL channels, which we target using the SMI library (Sec. 6.5.4). The FPGAs are installed in the Noctua cluster at the Paderborn Center for Parallel Computing, which exposes a programmable, fully connected optical switch, allowing us to chain FPGAs together in sequence with two

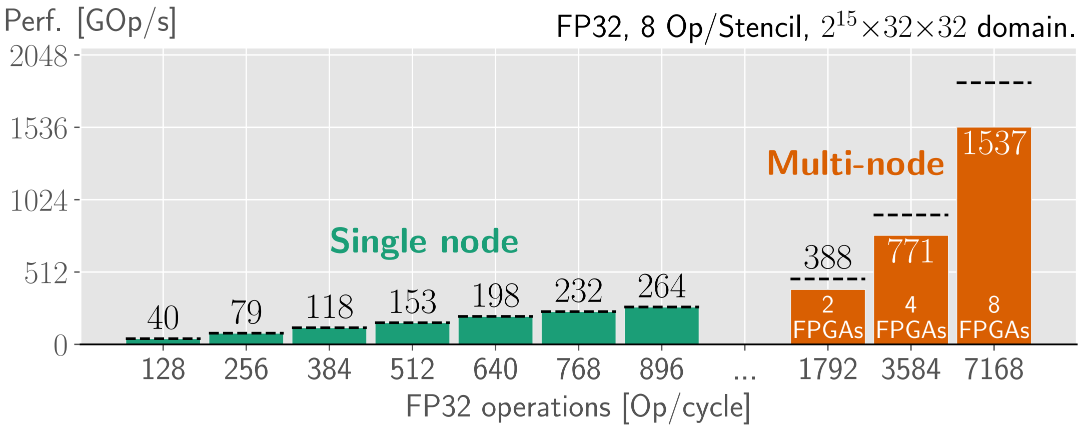


Figure 6.14: Performance scaling for single and multi-node.

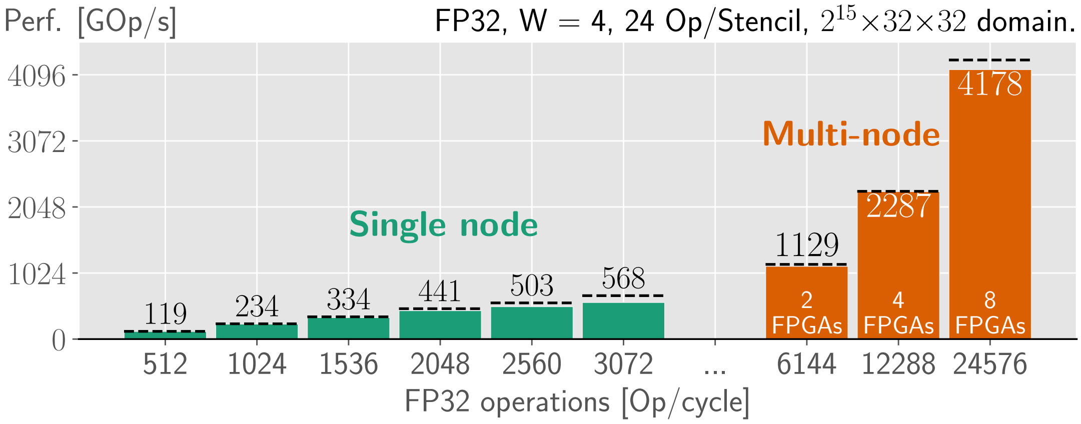


Figure 6.15: Performance scaling with 4-way vectorization.

40 Gbit/s links between each consecutive device to explore multi-device scaling.

For Xilinx, we target the same Alveo U250 accelerator board used in Sec. 2.6, housing a 14 nm Xilinx UltraScale+ XCU250-FIGD2104-2L-E FPGA and four 2400 MT/s DDR4 banks, built with a four chiplet architecture. We use Vitis 2020.2 to target the `xilinx_u250_xdma_201830_2` shell. The board is hosted at the Xilinx Adaptive Compute Cluster (XACC) at ETH Zurich. We do not evaluate multi-node stencils on Xilinx devices, as this was not supported by the SMI reference implementation at time of writing.

6.7.3 Iterative Stencil Performance

StencilFlow is built to handle complex stencil kernels, but is also capable of processing traditional, iterative-style stencil codes. We produce benchmarks using such kernels to establish the highest floating point performance reachable by StencilFlow, which can be compared to previous work. This is achieved by chaining together long linear sequences of stencils executed on a large input domain, analogous to time-tiled iterative stencils. To evaluate scaling behavior of an iterative stencil, we gradually increase the number of chained stencil computations until a single Stratix 10 device is fully utilized, then we continue the chain across multiple nodes by replacing accesses to on-chip FIFOs with network channels. We repeat the experiment with and without vectorization, to see the effect of coarsening stencil stages. The resulting benchmarks are shown in Fig. 6.14 and Fig. 6.15 without and with vectorization, respectively.

Without vectorization, the highest performing bitstream yields 264 GOp/s on a single Stratix 10 device, and scales up to 1.5 TOp/s across 8 FPGAs. A 4-way vectorized code reaches 568.2 GOp/s and 4.2 TOp/s on single and multi-device, respectively. Vectorization thus proves to be crucial to achieve high utilization of compute resources on the Stratix 10, as it reduces the ratio of overhead logic to computational logic. This further motivates the necessity of the stencil fusion transformation (Sec. 6.4.1) on input programs to coarsen the granularity of stencil nodes. Frequencies across all benchmarks are consistently in the range 292-317 MHz, which is factored into the upper bound calculation shown as dashed black lines, computed from Eq. 2.1 as C/f, where f is the design frequency.

We additionally measure the highest performance achievable *without* networking on a single device, as we are unable to vectorize the stencils in the distributed experiment further due to the network bandwidth bottlenecking the computation, included in Tab. 6.1. As a non-relative measure of device utilization, the table includes resource usage for the maximum performing stencil for each data type. The highest measured stencil performance of 1.3 TOp/s and 4.2 TOp/s for the Intel device, and 765 GOp/s for the Xilinx device, marks a 9.4×, 30×, and 5.5× speedup over the stencil performance reported for a single VCU1525 device in the original work on DaCe for single-device and multi-device, respectively (which in turn outperformed a state-of-the-art HLS compiler by five orders of magnitude, showing in inability of HLS compilers to yield satisfactory out-of-the-box performance).

For a more direct comparison on the Stratix 10 platform, we compare StencilFlow to a handwritten stencil implementation. Zohouri et al. [166] combine spatial and temporal

	Performance	ALM	FF	M20K	DSP
Total		103 M	3.7 M	11.7 K	5760
Avail.		692 K	2.8 M	8.9 K	4468
Jacobi 3D (Ours)	265 GOp/s	233 K 33.6%	534 K 19.3%	1495 16.7%	784 17.6%
Jacobi 3D W=8 (Ours)	921 GOp/s	437 K 63.1%	1207 K 43.6%	2285 25.5%	3072 68.8%
Diffusion 2D W=8 (Ours)	1313 GOp/s	449 K 64.8%	1329 K 48.0%	2565 28.6%	2304 51.6%
Diffusion 3D W=8 (Ours)	1152 GOp/s	567 K 81.9%	1606 K 57.9%	5357 59.8%	3072 68.8%
Diffusion 2D W=4 (Ours)	527 GOp/s	Alveo U250			
Diffusion 3D W=4 (Ours)	765 GOp/s	Alveo U250			
Diffusion 2D (Zohouri et. al. [166])	913 GOp/s	471.4 K 68.0%	1173.6 K 42.3%	2204 24.6%	3844 86.0%
Diffusion 3D (Zohouri et. al. [166])	934 GOp/s	450.5 K 65.0%	1078.2 K 38.9%	8684 97.0%	3592 80.4%
Waidyasooriya and Hariyama [145]	630 GOp/s	Arria 10 GX 1150			
SODA [27]	135 GOp/s	ADM-PCIE-KU3			
Niu et al. [112]	119 GOp/s	Virtex-6 SX475T			
Ben-Nun et al. [16]	139 GOp/s	Virtex UltraScale+ VCU1525			

Table 6.1: Highest performing kernels and their resource usage.

blocking in an HLS-based design to achieve high performance on stencil codes on an Arria 10 FPGA. We extend the authors' work by building their code[2] for Stratix 10, using the Diffusion 2D and 3D stencil codes. On advice from the authors, we configure parameters to a vectorization width of 16, run enough repetitions that the kernel runs for multiple seconds to hide initialization overhead, and disable burst interleaving. We include the resulting performance in Tab. 6.1, along with other previous results by Niu et al. [112] and Waidyasooriya and Hariyama [145], showing that StencilFlow is competitive even with hand-tuned code. We also consider frameworks emitting stencil FPGA code, including the Jacobi 3D result of SODA [27], which is the stencil backend of HeteroHalide [101] and HeteroCL [93]. For previous work we note the FPGA used for evaluation by the respective authors. We do not compare quantitatively to HeteroCL and Wang and Liang [147], as the authors do not report absolute performance numbers.

6.7.4 Off-Chip Memory Bandwidth

To measure achievable off-chip memory bandwidth by StencilFlow programs, we run two series of benchmarks on the Stratix 10 target: first, we measure the effective bandwidth utilization when scaling up *number of accesses*, but accessing only 32-bits per cycle at each access point. This stresses the routing on the device to deliver data to all endpoints every cycle. Second, we request the same total number of 32-bit operands, but at fewer, vectorized endpoints, requiring more operands per cycle per endpoint. We found the `-global-ring` and `-duplicate-ring` options to the Intel FPGA OpenCL compiler to significantly increase the number of parallel access points supported in the architecture before designs dropped in frequency. The resulting benchmarks, along with the analytically computed performance upper bound, are shown in Fig. 6.16. For the non-vectorized green bars, the x-axis corresponds to the number of access points, while the number of access points for the vectorized orange bars is the number of operands divided by the vector size of 4 (i.e., up to 12 access points are depicted).

After 24 parallel access points, we see a decrease in effective memory performance relative to peak, flattening out at 36.4 GByte/s, which is $36.4/76.8 = 47\%$ of peak bandwidth. This marks the limit of the memory controller crossbar, and of routing a large number of memory accesses across the device. The 4-way vectorized scenario allows for higher achievable bandwidth, but experiences a drop in efficiency at a lower number of access points (0.94× at 12 access points), and flattens out at 58.3 GByte/s, which is 76% of peak

---

[2] `https://github.com/zohourih/Diffusion_FPGA`, commit 96588e2.

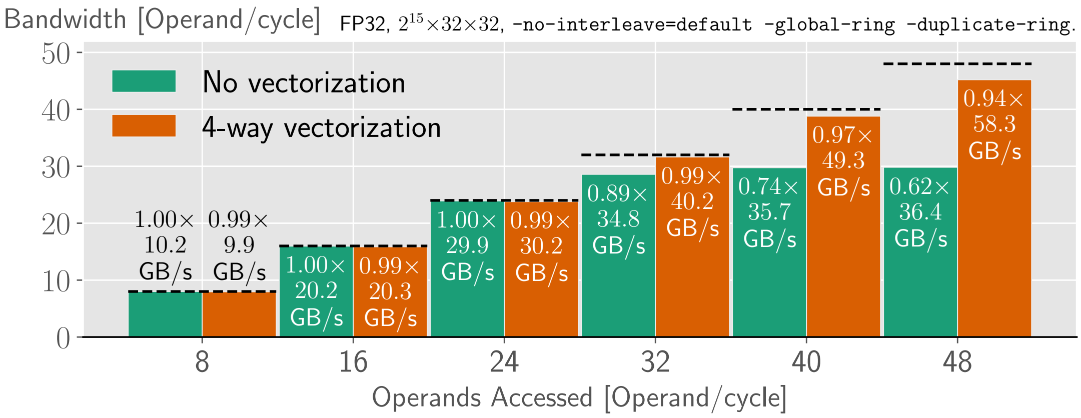
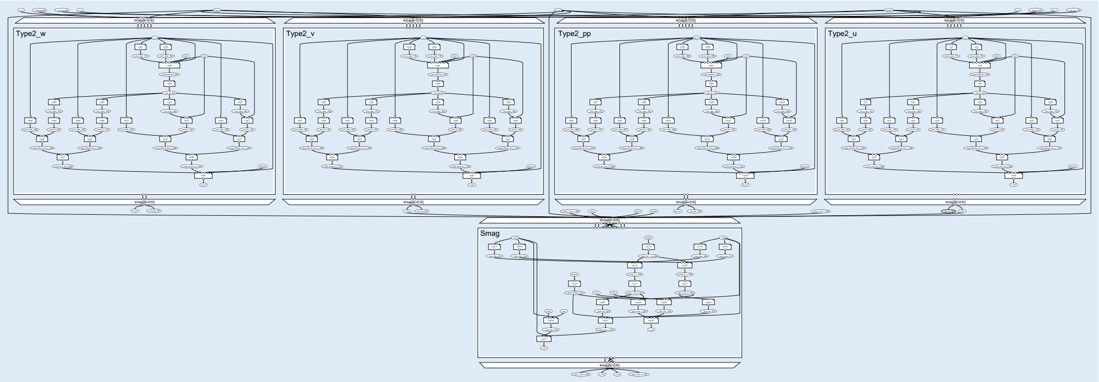


Figure 6.16: Effective bandwidth with number of operands requested per cycle (i.e., number of operands served if infinite bandwidth).

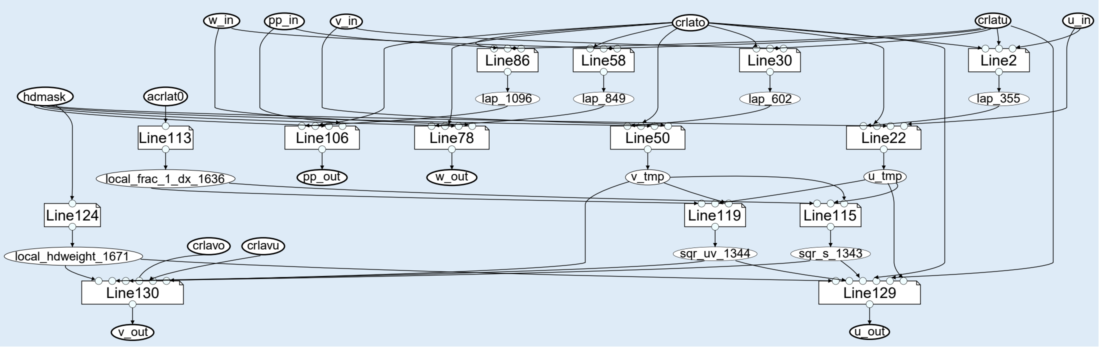
bandwidth. No further increase was seen with more access points, and 8-way vectorized programs achieve similar bandwidth.

6.8 Weather Simulation Application Study

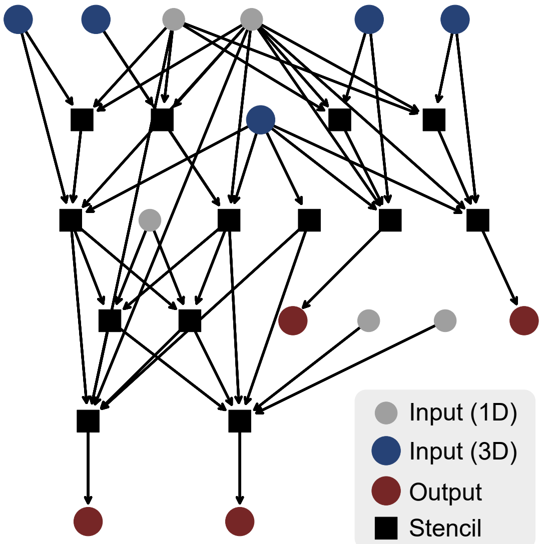
To stress the full capability of the StencilFlow stack we evaluate the *horizontal diffusion* stencil program, a large real-life weather simulation kernel from the COSMO weather model. Horizontal diffusion is a 4th order explicit method performed on a staggered latitude-longitude grid with Smagorinsky diffusion to smoothen wind velocity components [132]. We obtain the program from an input SDFGs using stencil Library Nodes, shown for horizontal diffusion in Fig. 6.17a, applying the `NestDim` and `MapFission` transformations described in Sec. 6.4, resulting in an SDFG as the one shown in Fig. 6.17b, from which the stencil program is extracted. The DAG in Fig. 6.17c is created after aggressively fusing consecutive stencils (see Sec. 6.4.1). In the fully fused program, initialization latency (L in Eq. 2.1) accounts for ∼0.7% of the total number of iterations required to evaluate the program, and is thus a negligible overhead. This program is run in production by the Swiss Federal Office of Meteorology and Climatology (MeteoSwiss), where simulations are performed with 32-bit floating point on an NVIDIA Pascal Tesla K80 cluster. We compare StencilFlow to the stronger TSMC 16 nm Tesla P100 GPU on the same architecture (comparable release window to the Stratix 10), a TSMC 12 nm Tesla V100 Volta GPU, and a 12-core Xeon CPU.



(a) Input program as SDFG, targeting software execution.



(b) Canonicalized SDFG, where only stencil computations and accesses remain.



(c) DAG of the transformed program inferred by StencilFlow.

Figure 6.17: Horizontal diffusion stencil program from the COSMO model.

6.8.1 Horizontal Diffusion Analysis

The horizontal diffusion DAG characterized by a high number of stencils reading the same input locations (28 accesses of 10 unique fields), allowing for the communication volume between them to be consolidated via delay buffers, as well as complex dependencies between stencil nodes (each non-source stencil receives data from 2−6 other stencil nodes). This requires the full complexity of an arbitrary DAG, and allows us to stress the full stack of StencilFlow.

Floating point operations in the DAG include 87 additions, 41 multiplications, and 2 square roots, in addition to 2 minimum and 2 maximum operations, and ternary operations resulting in 20 data-dependent branches. With maximum reuse of all input fields and all computed fields (i.e., perfect locality), the program reads $5IJK + 5I$ operands and writes $4IJK$ operands, for a total of $9IJK + 5I$ operands. Considering floating point arithmetic only, this implies a upper bound arithmetic intensity of (square root is counted as one operation):

$$\frac{(87+41+2)IJK\ \text{[Ops]}}{9IJK+5I\ \text{[operands]}} \approx \frac{130}{9}\ \left[\frac{\text{Ops}}{\text{operand}}\right],$$

which for 32-bit floating point corresponds to

$$\frac{130/9\ \left[\frac{\text{Ops}}{\text{operand}}\right]}{4\ \left[\frac{\text{Byte}}{\text{operand}}\right]} = \frac{65}{18}\ \left[\frac{\text{Ops}}{\text{Byte}}\right]. \tag{6.1}$$

Using the benchmark of practically achievable bandwidth presented in Sec. 6.7.4 for the Stratix 10 FPGA, the highest achievable performance in roofline model [151] terms is:

$$\frac{65}{18}\ \left[\frac{\text{Ops}}{\text{Byte}}\right] \cdot 58.3\ \left[\frac{\text{GByte}}{\text{s}}\right] = 210.5\ \left[\frac{\text{GOp}}{\text{s}}\right], \tag{6.2}$$

or 277.3 GOp/s at the peak data sheet bandwidth of 76.8 GByte/s. This is well below what is achievable by a stencil program with higher arithmetic intensity (see Sec. 6.7.3), indicating that high bandwidth is required to shine in realistic stencil applications. We compute the bandwidth required to saturate the compute performance measured in Sec. 6.7.3 for the arithmetic intensity of the studied program to be:

$$\frac{917.1\ \left[\frac{\text{GOp}}{\text{s}}\right]}{65/18\ \left[\frac{\text{Op}}{\text{Byte}}\right]} = 254.0\ \left[\frac{\text{GByte}}{\text{s}}\right]. \tag{6.3}$$

	Runtime	Performance	Peak BW.	%Roof.
Stratix 10	1178 µs	145 GOp/s	77 GByte/s	52%
Stratix 10*	332 µs	513 GOp/s	∞ GByte/s	–
Xeon 12C	5270 µs	32 GOp/s	68 GByte/s	13%
P100	810 µs	210 GOp/s	732 GByte/s	8%
V100	201 µs	849 GOp/s	900 GByte/s	26%

*Without memory bandwidth constraints.

Table 6.2: Horizontal diffusion benchmarks.

The ideal logic to bandwidth ratio is thus off from the ideal ratio by a factor of $\sim$3$-$4 on the target Stratix 10 platform. To explore the performance potential of the Stratix 10 without this memory bottleneck, we will include experiments with simulated "infinite" memory bandwidth, by replacing memory accesses with compile-time constants fed to the computational circuit (and omitting validation of functional correctness).

6.8.2 Horizontal Diffusion Benchmark

We compile the DAG in Fig. 6.17c for the Stratix 10 from the constructed dataflow graph by StencilFlow. As shown in the analysis above, the program is bandwidth-bound on this platform, which requires us to saturate the bandwidth to maximize performance. Without vectorization, the pipelined circuit requires approximately 9 operands/cycle, corresponding to 10.8 GByte/s at 300 MHz for single precision floating point. We thus vectorize the program by a factor of 8 for a maximum bandwidth of 86.4 GOp/s, in addition to building a 16-way vectorized kernel with simulated input memory to evaluate performance without the memory bottleneck. We target a 128×128×80 domain size, which is used for performance benchmarking by MeteoSwiss. Specifically, a 128×128 horizontal domain is stacked in 80 vertical layers. In addition to runtime and the effective performance, we consider peak memory bandwidth and the associated fraction of highest achievable performance *for the given arithmetic intensity* computed according to Eq. 6.1 (%Roof.). The results are listed in Tab. 6.2.

We include CPU and GPU performance as a point of comparison, using a 12-core Intel Xeon 2.60/3.50 GHz E5-2690V3 CPU, and NVIDIA Tesla P100 and V100 GPUs, compiled with CUDA v10.1 and gcc 8.3.0. The application is synthesized using the MeteoSwiss Dawn [113] stencil-optimizing compiler toolchain[3], which was also used to generate the

---

[3] `https://github.com/MeteoSwiss-APN/dawn`, commit `4ae6dc0`.

StencilFlow input program. Dawn is specifically designed to optimize weather and climate stencil programs for GPU and CPU, employing data movement optimizations, GPU kernel fusion, CPU multi-threading, vectorization, and efficient GPU boundary scheduling. The domain size of 128×128×80 is sufficient for saturating the GPU thread scheduler (i.e., larger domains do not significantly increase GPU performance). The horizontal diffusion program emitted by Dawn for CPU and GPU executes five components of horizontal diffusion as distinct kernels. We omit kernel launch overhead and report the raw kernel execution time only, included in Tab. 6.2.

The FPGA platform outperforms the CPU by 4.5× and is outperformed by either GPU, but comes closest to the upper bound (Eq. 6.2) imposed by its roofline characteristics at the given arithmetic intensity: 52% of the bandwidth upper bound (69% of the highest measured bandwidth in Sec. 6.7.4), at 26% ALMs, 27% DSPs, and 20% M20K utilization, respectively. The benchmark simulating infinite memory bandwidth shows significant headroom for pushing the performance at this arithmetic intensity with higher bandwidth off-chip memory: without the bandwidth bottleneck, the Stratix 10 would outperform the P100, but falls at 60% of the performance of the V100, at 46% ALMs, 48% DSPs, and 20% M20Ks.

6.8.3 Silicon Efficiency

The Stratix 10 is estimated to be a 700 mm^2 die [3] (half the Stratix 10M, which fuses two Stratix 10 chiplets) on Intel's 14 nm process, compared to 610 mm^2 on TSMC 16 nm and 815 mm^2 on TSMC 12 nm for the P100 and V100, respectively. Using the benchmarks from Tab. 6.2, this amounts to a silicon efficiency of 0.21 and 0.71 $\frac{\text{GOp/s}}{\text{mm}^2}$ with and without the memory bottleneck for the Stratix 10, respectively; $0.34\frac{\text{GOp/s}}{\text{mm}^2}$ for P100; and $1.04\frac{\text{GOp/s}}{\text{mm}^2}$ for the V100, when performing the horizontal diffusion experiment.

6.8.4 Spatial Tiling

We have not considered spatial tiling, as on-chip memory requirements were not a restriction for building the large weather stencil program evaluated. Both memory bandwidth and logic were bottlenecks before on-chip memory capacity, despite minimizing off-chip memory bandwidth in the program. Eventually, increasing the domain size will scale the internal buffer and delay buffer sizes beyond what is feasible to buffer in on-chip memory. Spatial tiling can be employed in this scenario, introducing redundant computation at the domain boundaries proportional to the DAG depth and the tile surface-to-volume ratio.

This is primarily a scheduling challenge, which can be efficiently solved in practice [166].

6.9 Related Work

There are numerous works on stencil accelerators on FPGAs [53, 166, 145], including for multi-device settings on up to 9 interconnected FPGAs [124], all of which we have considered throughout this chapter. Other frameworks generating stencil architectures have also been proposed [27, 93, 101, 147], which we consider in Sec. 6.7.3. Common to these works is that they treat a single stencil operation applied iteratively, allowing them to unroll the time dimension as a source of temporal locality. StencilFlow is on a par or outperforms all the above on simple iterative stencils, and treats a much wider range of input programs. Niu et al. [112] explore runtime reconfiguration of an FPGA to eliminate idle operators during program execution. Runtime reconfiguration is not beneficial for stencil programs considered by StencilFlow, as all operators are assumed to operate in the same iteration space and fully in parallel after the initialization phase.

Darkroom [70] is a framework producing spatial accelerators of image processing pipelines from a high-level input DSL. StencilFlow takes a similar approach, but accepts a wider scope of input programs: in particular arbitrary DAGs of stencils, and 3D input/output domains. Other DSLs [66, 67] do not consider spatial computing architectures.

For the application study, Singha et al. [129, 128] present a hand-tuned implementation of the horizontal diffusion application targeting an FPGA+CPU coherent system. The authors report 129.9 GOp/s on an ADM-PCIE-9V3 board with the NARMADA accelerator, and 485.4 GOp/s on an ADM-PCIE-9H7 board with the NERO accelerator, the latter owing its large increase in performance to the introduction of HBM memory, effectively eliminating the memory bottleneck described by Eq. 6.3. The fully code generated kernels emitted by StencilFlow outperform the DDR4-based accelerator when memory bound, and the HBM-based when compute bound (i.e., when high memory bandwidth is simulated).

6.10 Summary

We introduced StencilFlow, an end-to-end analysis, optimization and code-generation stack built on the DaCe framework, enabling the generation of complex high-performance stencil programs on spatial architectures from a high-level input DSL. Based on a DAG representation, StencilFlow automatically insert buffers within and between stencil operations to achieve *perfect reuse* of all data in the program. Architectures emitted by StencilFlow achieve the *highest recorded single-device performance* of 1.31 TOp/s, and the *highest recorded multi-device performance* of 4.18 TOp/s on 8 FPGAs, and the highest recorded stencil performance on Xilinx FPGAs of 765 GOp/s. We demonstrated the domain complexity supported by the framework by treating a large stencil program used in production for weather prediction, comparing the generated architecture to state-of-the-art GPU and CPU performance. We release StencilFlow as open source software, enabling reproducibility and allowing scientists to easily target spatial computing accelerators with complex stencil programs.

StencilFlow showcases how the powerful toolbox of DaCe can be used as a backend engine for a DSL like StencilFlow, by exploiting the multi-level design methodology enabled by Library Nodes to both embed domain-specific information and exploit the basic SDFG primitives to build the dataflow architectures that is finally code-generated for either FPGA vendor, directly from a Python code taking nothing but a `.json`-file as input.

To extend the applicability of the optimized stencil Library Node developed for StencilFlow, a generalized version has been merged back into the main DaCe repository, which can also read directly from memory in addition to reading from streams, and will instantiate the appropriate internal buffers when expanded for FPGAs. This can be used as a drop-in stencil computation to make targeting FPGAs with stencil computations easy and portable, and can also be expanded into a CPU implementation.

Chapter 7

Conclusion

Traditional load/store architectures spend the majority of their transistor and power budgets on general-purpose cache, register files, and control logic, to mitigate the severe memory bottleneck that has emerged from the diverging evolution of computational performance and memory bandwidth. Spatial computing architectures can provide a leap forward in performance and power efficiency by shedding these in favor of moving data directly between computational logic, rather than through centralized memory. Programmable spatial architectures, such as FPGAs, can be used to create application-specific hardware architectures. However, programming these devices is notoriously difficult, and as a result, they are not commonly used in the high-performance computing (HPC) domain.

With this dissertation, we set out to provide a rich toolbox of knowledge and systems to improve productivity when programming spatial computing systems for HPC.

In Chapter 2, we proposed a set of source-to-source transformations for high-level synthesis (HLS) languages, which target hardware-specific properties that were not covered by traditional software optimization, such as enabling pipelining of operations, streaming computations, on-chip buffering, and vertical unrolling of computations to exploit temporal reuse. With the transformations described, the provided cheat sheet, and the example reference codes, we empower **HLS developers** and **compiler engineers** to productively optimize HPC applications for hardware acceleration.

In Chapter 3, we further improved the quality of life of HLS development with the open source project hlslib, filling some of the gaps in existing HLS tools and providing useful abstractions for project configuration, simulation, and compilation. hlslib has gathered significant interest in the community, empowering **HLS developers** and **compiler engineers**, and is used as a component in all following chapters.

In Chapter 4, we covered the end-to-end optimization of a matrix multiplication HLS code, applying the proposed transformations from Chapter 2 in a concrete setting and using the tools from Chapter 3, designing the architecture from a model that simultaneously maximizes performance and minimizes I/O based on hardware constants of the target platform. The code was released as an open source repository, and like hlslib, has seen significant interest in the community, empowering **HLS developers** by offering one of the most commonly used HPC kernels as a plug-and-play component.

In Chapter 5, we proposed a new abstraction for developing FPGA kernels by introducing *Stateful DataFlow multiGraphs* (SDFG); a data-centric intermediate representation that empowers **performance engineers** by giving them a platform to productively optimize programs using graph-based transformations to manipulate their data movement. SDFGs are created, optimized, and compiled using the Data-Centric (DaCe) parallel programming framework. DaCe is based on the concept of separation of concerns, empowering **domain scientists** by letting them define programs using a productive high-level frontend such as NumPy or StencilFlow, then handing off the generated SDFG for optimization by a skilled performance engineer. We extend SDFGs with a multi-level design methodology, allowing abstract functionality to co-exist with general dataflow, which can be progressively lowered to library calls or specialized implementations, empowering **performance engineers** to seamlessly target different libraries and implementations of key operators, and empowering **compiler engineers** to build custom extensions and DSLs based on the DaCe infrastructure. The DaCe project is a massively collaborative effort that has been shown to accelerate a large and diverse set of applications, ranging from single-node programs to accelerators and distributed systems. For this dissertation, we showed how DaCe can be used to build applications for both Xilinx and Intel FPGA devices, offering portability and knowledge transfer between software and hardware optimization.

Finally, in Chapter 6, we showed how DaCe can be employed as a powerful backend engine for a domain-specific language, empowering the **domain scientist** by achieving ultimate productivity within this domain with a full end-to-end stack, emitting highly efficient stencil architectures from a simple high-level input description. By providing optimized parametric subgraph expansions for both FPGA vendors, we can reuse the remaining dataflow through the DaCe code generator to achieve seamless portability between them.

With the set of optimization techniques, open source software, abstractions, and optimization tools presented throughout this dissertation, we have shown how we can empower a wide range of users, including HLS programmers, performance engineers, domain scientists, and compiler engineer to tap into acceleration of HPC programs using application-specific hardware architectures. With this dissertation as an academic contribution, our software as engineering contributions, and our tutorial series as an eductional contribution, we hope to contribute to bridge the gap between hardware development and HPC.

7.1 Future Work

DaCe is a promising step forward in productivity when programming spatial architectures, but there are still aspects of optimization that require a more manual intervention. Forming systolic arrays by streaming between vertically unrolled computations is the most potent source of parallelism on spatial architectures (Sec. 2.3.2–2.3.3), but these currently still have to be manually inferred and implemented. Designing a dataflow transformation on SDFGs that can automatically compose such architectures from a source of reuse would allow us to productively form them, and potentially discover new opportunities for systolic arrays by matching it against the large set of applications that already exist as SDFGs.

In the domain of FPGAs, a major roadblock for achieving higher performance on existing platforms remains the placement and routing (P&R) process. By the nature of DaCe's dataflow-based representation, we could provide contextual information to perform a more guided P&R to achieve higher resource utilization at high frequencies, and potentially reducing compilation times by shrinking the design space. The first steps towards this has been taken by Carl Johnsen, who has implemented support for embedding RTL directly in DaCe, and taking more control of the build process for Xilinx devices.

In Chapter 6, we scaled stencil architectures across multiple interconnected FPGA devices. However, more general support for distributing FPGA applications in DaCe remains a challenge, partly due to the lack of consistent support by vendors. As of writing, only a very recent community-implemented project, EasyNet [69], exists for MPI-style communication in HLS on Xilinx devices. Furthermore, the number of available FPGA-based clusters with inter-node communication is extremely limited, making it challenging to test solutions beyond the ecosystem of a single deployment per vendor. Once this landscape has stabilized, however, DaCe can provide an excellent platform for portable multi-node programs, as this can be transparently abstracted by the existing stream objects in the SDFG combined with the SMI programming model for FPGA communication [40].

While FPGAs were chosen as the most viable spatial computing architecture to target in this work, we wish to test the techniques and systems described against other and potential future spatial architectures. The DaCe framework provides an excellent platform for this, as the "only" extension required would be to add a new code generating backend, while the mapping to the spatial domain and the existing infrastructure of transformations, frontends, and libraries would remain. Currently announced platforms of interest could be the Cerebras wafer-on-a-chip architecture and Xilinx' CGRA-style AI Engines.

Bibliography

[1] DaCeML – machine learning powered by data-centric parallel programming. https://github.com/spcl/daceml. Accessed August 30, 2021.

[2] Intel FPGA SDK for OpenCL Pro Edition Best Practices Guide, UG-OCL003, revision 2021.03.29. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/opencl-sdk/aocl-best-practices-guide.pdf. Accessed July 16, 2021.

[3] Intel introduces world's largest FPGA with 43.3 billion transistors. https://www.tomshardware.com/news/intel-introduces-worlds-largest-fpga-with-433-billion-transistors. Published by Tom's Hardware, November 2019. Accessed October 27, 2020.

[4] Intel's exascale dataflow engine drops x86 and von Neumann. https://www.nextplatform.com/2018/08/30/intels-exascale-dataflow-engine-drops-x86-and-von-neuman/. Published on The Next Platform, August 2018. Accessed September 13, 2021.

[5] Alok Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9), September 1988.

[6] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. Compilers, principles, techniques. *Addison Wesley*, 7(8):9, 1986.

[7] Frances E. Allen and John Cocke. *A catalogue of optimizing transformations*. 1971.

[8] John R. Allen and Ken Kennedy. Automatic loop interchange. In *Proceedings of the 1984 SIGPLAN Symposium on Compiler Construction (CC'84)*, pages 233–246, 1984.

[9] Edward Anderson, Zhaojun Bai, Christian Bischof, L Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, et al. *LAPACK Users' guide*. SIAM, 1999.

[10] Michael Anderson, Grey Ballard, James Demmel, and Kurt Keutzer. Communication-avoiding QR decomposition for GPUs. In *Proceedings of the 25th International Parallel and Distributed Processing Symposium (IPDPS'11)*, pages 48–58. IEEE, 2011.

[11] Joshua Auerbach, David F. Bacon, Ioana Burcea, Perry Cheng, Stephen J. Fink, Rodric Rabbah, and Sunil Shukla. A compiler and runtime for heterogeneous computing. In *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*, pages 271–276. ACM, 2012.

[12] 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 *Proceedings of the 49th Annual Design Automation Conference (DAC'12)*, pages 1216–1225. ACM, 2012.

[13] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys (CSUR)*, 26(4):345–420, December 1994.

[14] David F. Bacon, Rodric Rabbah, and Sunil Shukla. FPGA programming for the masses. *Communications of the ACM*, 56(4):56–63, 2013.

[15] M. Baldauf, A. Seifert, J. Förstner, D. Majewski, and M. Raschendorfer. Operational convective-scale numerical weather prediction with the COSMO model: Description and sensitivities. *Monthly Weather Review, 139:3387–3905*, 2011.

[16] Tal Ben-Nun, Johannes de Fine Licht, Alexandros N Ziogas, Timo Schneider, and Torsten Hoefler. Stateful dataflow multigraphs: A data-centric model for performance portability on heterogeneous architectures. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*, 2019.

[17] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *ACM Computing Surveys (CSUR)*, 52(4):1–43, 2019.

[18] R. Bernstein. Multiplication by integer constants. *Software: Practice and Experience*, 16(7):641–652, July 1986.

[19] Maciej Besta, Marc Fischer, Tal Ben-Nun, Johannes de Fine Licht, and Torsten Hoefler. Substream-centric maximum matchings on FPGA. In *Proceedings of the 27th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'19)*, pages 152–161, 2019.

[20] Maciej Besta, Dimitri Stanojevic, Johannes de Fine Licht, Tal Ben-Nun, and Torsten Hoefler. Graph processing on FPGAs: Taxonomy, survey, challenges. *arXiv:1903.06697*, 2019.

[21] Michaela Blott, Thomas B Preußer, Nicholas J Fraser, Giulio Gambardella, Kenneth O'brien, Yaman Umuroglu, Miriam Leeser, and Kees Vissers. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 11(3), 2018.

[22] Uday Bondhugula, Albert Hartono, Jagannathan Ramanujam, and Ponnuswamy Sadayappan. A practical automatic polyhedral parallelizer and locality optimizer. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*, pages 101–113, 2008.

[23] Andre R. Brodtkorb, Christopher Dyken, Trond R. Hagen, Jon M. Hjelmervik, and Olaf O. Storaasli. State-of-the-art in heterogeneous computing. *Scientific Programming*, 18(1):1–33, 2010.

[24] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown, and Tomasz Czajkowski. LegUp: High-level synthesis for FPGA-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'11)*, pages 33–36, 2011.

[25] Gregory J. Chaitin, Marc A. Auslander, Ashok K. Chandra, John Cocke, Martin E. Hopkins, and Peter W. Markstein. Register allocation via coloring. *Computer Languages*, 6(1):47 – 57, 1981.

[26] Xinyu Chen, Ronak Bajaj, Yao Chen, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. On-the-fly parallel data shuffling for graph processing on OpenCL-based FPGAs. In *IEEE 29th International Conference on Field Programmable Logic and Applications (FPL'19)*, pages 67–73, 2019.

[27] Yuze Chi, Jason Cong, Peng Wei, and Peipei Zhou. SODA: Stencil with optimized dataflow architecture. In *2018 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'18)*, 2018.

[28] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11):850–856, November 1977.

[29] Jason Cong, Muhuan Huang, Peichen Pan, Yuxin Wang, and Peng Zhang. Source-to-source optimization for HLS. In *FPGAs for Software Programmers*, pages 137–163. Springer, 2016.

[30] Jason Cong, Bin Liu, Stephen Neuendorffer, Juanjo Noguera, Kees Vissers, and Zhiru Zhang. High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 30(4):473–491, 2011.

[31] Matthieu Courbariaux, Itay Hubara, Daniel Soudry, Ran El-Yaniv, and Yoshua Bengio. Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or -1. *arXiv:1602.02830*, 2016.

[32] Tomasz S. Czajkowski, Utku Aydonat, Dmitry Denisenko, John Freeman, Michael Kinsner, David Neto, Jason Wong, Peter Yiannacouras, and Deshanand P. Singh. From OpenCL to high-performance hardware on FPGAs. In *IEEE 22nd International Conference on Field Programmable Logic and Applications (FPL'12)*, pages 531–534, 2012.

[33] Jeferson Santiago da Silva, François-Raymond Boyer, and J.M. Pierre Langlois. Module-per-Object: a human-driven methodology for C++-based high-level synthesis design. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'19)*, pages 218–226, 2019.

[34] Paolo D'Alberto and Alexandru Nicolau. Using recursion to boost ATLAS's performance. In *High-Performance Computing*, pages 142–151. Springer, 2008.

[35] Bita Darvish Rouhani, Daniel Lo, Ritchie Zhao, Ming Liu, Jeremy Fowers, Kalin Ovtcharov, Anna Vinogradsky, Sarah Massengill, Lita Yang, Ray Bittner, Alessandro Forin, Haishan Zhu, Taesik Na, Prerak Patel, Shuai Che, Lok Chand Koppaka, Xia Song, Subhojit Som, Kaustav Das, Saurabh Tiwary, Steve Reinhardt, Sitaram

Lanka, Eric Chung, and Doug Burger. Pushing the limits of narrow precision inferencing at cloud scale with Microsoft floating point. In *Advances in Neural Information Processing Systems (NeurIPS'20)*, volume 33, pages 10271–10281. Curran Associates, Inc., 2020.

[36] Johannes de Fine Licht, Maciej Besta, Simon Meierhans, and Torsten Hoefler. Transformations of high-level synthesis codes for high-performance computing. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 32(5):1014–1029, May 2021.

[37] Johannes de Fine Licht and Torsten Hoefler. hlslib: Software engineering for hardware design. *arXiv:1910.04436*, 2019.

[38] 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 *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO'21)*, pages 315–326, 2021.

[39] Johannes de Fine Licht, Grzegorz Kwasniewski, and Torsten Hoefler. Flexible communication avoiding matrix multiplication on FPGA with high-level synthesis. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*, pages 152–161, 2020.

[40] Tiziano De Matteis, Johannes de Fine Licht, Jakub Beránek, and Torsten Hoefler. Streaming message interface: High-performance distributed memory programming on reconfigurable hardware. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*, 2019.

[41] Tiziano De Matteis, Johannes de Fine Licht, and Torsten Hoefler. FBLAS: Streaming linear algebra kernels on FPGA. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20)*, 2020.

[42] Mauro Del Ben, Ole Schütt, Tim Wentz, Peter Messmer, Jürg Hutter, and Joost VandeVondele. Enabling simulation at the fifth rung of DFT: Large scale RPA calculations with excellent time to solution. *Computer Physics Communications*, 187:120–129, 2015.

[43] James Demmel and Grace Dinh. Communication-optimal convolutional neural nets. *arXiv preprint arXiv:1802.06905*, 2018.

[44] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-optimal parallel recursive rectangular matrix multiplication. In *Proceedings of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing (IPDPS'13)*, pages 261–272, 2013.

[45] Robert H Dennard, Fritz H Gaensslen, Hwa-Nien Yu, V Leo Rideout, Ernest Bassous, and Andre R LeBlanc. Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE Journal of Solid-State Circuits (JSSC)*, 9(5):256–268, 1974.

[46] Erik H. D'Hollander. High-level synthesis optimization for blocked floating-point matrix multiplication. *SIGARCH Computer Architecture News*, 44(4):74–79, January 2017.

[47] Erik H. D'Hollander. High-level synthesis optimization for blocked floating-point matrix multiplication. *SIGARCH Computer Architecture News*, 44(4):74–79, January 2017.

[48] Jack J. Dongarra and A. R. Hinds. Unrolling loops in Fortran. *Software: Practice and Experience*, 9(3):219–226, 1979.

[49] Yong Dou, S. Vassiliadis, G. K. Kuzmanov, and G. N. Gaydadjiev. 64-bit floating-point FPGA matrix multiplication. In *Proceedings of the 2005 ACM/SIGDA 13th International Symposium on Field-Programmable Gate Arrays (FPGA'05)*, pages 86–95, 2005.

[50] Haggai Eran, Lior Zeno, Zsolt István, and Mark Silberstein. Design patterns for code reuse in HLS packet processing pipelines. In *IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'19)*, pages 208–217, 2019.

[51] C. A. Fletcher. *Computational Techniques for Fluid Dynamics 2*. Springer-Verlag New York, Inc., 1988.

[52] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In *Proceedings of the 20th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'12)*, pages 47–56, 2012.

[53] Haohuan Fu and Robert Clapp. Eliminating the memory bottleneck: an FPGA-based solution for 3D reverse time migration. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'11)*, pages 65–74, 2011.

[54] Haohuan Fu and Robert G. Clapp. Eliminating the memory bottleneck: An FPGA-based solution for 3D reverse time migration. In *Proceedings of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'11)*, pages 65–74, 2011.

[55] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx adaptive compute acceleration platform: Versal architecture. In *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)*, pages 84–93.

[56] G. A. Geist and E. Ng. Task scheduling for parallel sparse Cholesky factorization. *International Journal of Parallel Programming*, 18(4):291–314, July 1990.

[57] Amir Gholami, Ariful Azad, Peter Jin, Kurt Keutzer, and Aydin Buluc. Integrated model, batch and domain parallelism in training neural networks. *arXiv preprint arXiv:1712.04432*, 2017.

[58] GitHub. The 2020 state of the Octoverse, 2020. https://octoverse.github.com/. Accessed August 3, 2021.

[59] Paolo Gorlani, Tobias Kenter, and Christian Plessl. OpenCL implementation of Cannon's matrix multiplication algorithm on Intel Stratix 10 FPGAs. In *IEEE International Conference on Field-Programmable Technology (ICFPT'19)*, pages 99–107, 2019.

[60] Tobias Grosser, Armin Groesslinger, and Christian Lengauer. Polly – performing polyhedral optimizations on a low-level intermediate representation. *Parallel Processing Letters (PPL)*, 22(04), 2012.

[61] Y. Guan, H. Liang, N. Xu, W. Wang, S. Shi, X. Chen, G. Sun, W. Zhang, and J. Cong. FP-DNN: An automated framework for mapping deep neural networks onto FPGAs with RTL-HLS hybrid templates. In *IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'17)*, pages 152–159, 2017.

[62] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. SPARK: a high-level synthesis framework for applying parallelizing compiler transformations. In *Proceedings of the 16th International Conference on VLSI Design (VLSID'03)*, pages 461–466, 2003.

[63] Sumit Gupta, Rajesh Kumar Gupta, Nikil D. Dutt, and Alexandru Nicolau. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 9(4):441–470, October 2004.

[64] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. MODESTO: Data-centric analytic optimization of complex stencil programs on heterogeneous architectures. In *Proceedings of the 29th ACM on International Conference on Supercomputing (ICS'15)*, pages 177–186, 2015.

[65] Tobias Gysi, Tobias Grosser, and Torsten Hoefler. Absinthe: Learning an analytical performance model to fuse and tile stencil codes in one shot. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT'19)*, pages 370–382. IEEE, 2019.

[66] Tobias Gysi, Carlos Osuna, Oliver Fuhrer, Mauro Bianco, and Thomas C Schulthess. STELLA: A domain-specific tool for structured grid methods in weather and climate models. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'15)*, 2015.

[67] Bastian Hagedorn, Larisa Stoltzfus, Michel Steuwer, Sergei Gorlatch, and Christophe Dubach. High performance stencil code generation with Lift. In *Proceedings of the 2018 International Symposium on Code Generation and Optimization (CGO'18)*, pages 100–112, 2018.

[68] Azzam Haidar, Stanimire Tomov, Jack Dongarra, and Nicholas J Higham. Harnessing GPU tensor cores for fast FP16 arithmetic to speed up mixed-precision iterative refinement solvers. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'18)*, 2018.

[69] Zhenhao He, Dario Korolija, and Gustavo Alonso. EasyNet: 100 Gbps network for HLS. In *IEEE 31st International Conference on Field-Programmable Logic and Applications (FPL'21)*, 2021.

[70] 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. *ACM Transactions on Graphics (TOG)*, 33(4):144–1, 2014.

[71] Mark Horowitz. Computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, pages 10–14, 2014.

[72] Paul Hudak and Benjamin Goldberg. Serial combinators: "optimal" grains of parallelism. In *Conference on Functional Programming Languages and Computer Architecture (FPCA'85)*, pages 382–399. Springer, 1985.

[73] Intel. Intel oneAPI Math Kernel Library (MKL), 2007. `https://software.intel.com/content/www/us/en/develop/tools/oneapi/components/onemkl.html`. Accessed July 19, 2021.

[74] Dror Irony et al. Communication lower bounds for distributed-memory matrix multiplication. *JPDC*, 64(9):1017–1026, September 2004.

[75] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication lower bounds for distributed-memory matrix multiplication. *Journal of Parallel and Distributed Computing (JPDC)*, 64(9):1017–1026, 2004.

[76] Andrei Ivanov, Nikoli Dryden, Tal Ben-Nun, Shigang Li, and Torsten Hoefler. Data movement is all you need: A case study on optimizing transformers. *Proceedings of Machine Learning and Systems (MLSys'21)*, 3, 2021.

[77] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, et al. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *IEEE/ACM International Conference on Computer-Aided Design (ICCAD'17)*, pages 209–216, 2017.

[78] Michael James, Marvin Tom, Patrick Groeneveld, and Vladimir Kibardin. Physical mapping of neural networks on a wafer-scale deep learning accelerator. In *Proceedings of the 2020 International Symposium on Physical Design (ISPD'20)*, pages 145–149.

[79] Q. Jia and H. Zhou. Tuning stencil codes in OpenCL for FPGAs. In *2016 IEEE 34th International Conference on Computer Design (ICCD'16)*, pages 249–256, 2016.

[80] Hong Jia-Wei and Hsiang-Tsung Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the 13th Annual ACM Symposium on Theory of Computing (STOC'81)*, pages 326–333, 1981.

[81] Lana Josipović, Radhika Ghosal, and Paolo Ienne. Dynamically scheduled high-level synthesis. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*, pages 127–136, 2018.

[82] Zeljko Jovanović and V Milutinovic. FPGA accelerator for floating-point matrix multiplication. *IET Computers & Digital Techniques*, 6(4):249–256, 2012.

[83] Ryan Kastner, Janarbek Matai, and Stephen Neuendorffer. Parallel programming for FPGAs. *arXiv:1805.03648*, May 2018.

[84] Tobias Kenter, Gopinath Mahale, Samer Alhaddad, Yevgen Grynko, Christian Schmitt, Ayesha Afzal, Frank Hannig, Jens Förstner, and Christian Plessl. OpenCL-based FPGA design to accelerate the nodal discontinuous Galerkin method for unstructured meshes. In *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'18)*, pages 189–196, 2018.

[85] D. Koeplinger, R. Prabhakar, Y. Zhang, C. Delimitrou, C. Kozyrakis, and K. Olukotun. Automatic generation of efficient accelerators for reconfigurable hardware. In *2016 ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA'16)*, pages 115–127, 2016.

[86] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'81)*, pages 207–218, 1981.

[87] David J. Kuck. A survey of parallel machine organization and programming. *ACM Computing Surveys (CSUR)*, 9(1):29–59, March 1977.

[88] V. B. Y. Kumar, S. Joshi, S. B. Patkar, and H. Narayanan. FPGA based high performance double-precision matrix multiplication. In *22nd International Conference on VLSI Design (VLSI'09)*, 2009.

[89] HT Kung and Charles E Leiserson. Systolic arrays (for VLSI). In *Sparse Matrix Proceedings*, volume 1, pages 256–282, 1978.

[90] Ian Kuon and Jonathan Rose. Measuring the gap between FPGAs and ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD)*, 26(2):203–215, 2007.

[91] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis (SC'19)*, 2019.

[92] Griffin Lacey, Graham W. Taylor, and Shawki Areibi. Deep learning on FPGAs: Past, present, and future. *arXiv:1602.04283*, 2016.

[93] 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 *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'19)*, pages 242–251, 2019.

[94] M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the ACM/SIGPLAN 1988 Conference on Programming Language Design and Implementation (PLDI'88)*, pages 318–328, 1988.

[95] Monica D. Lam, Edward E. Rothberg, and Michael E. Wolf. The cache performance and optimizations of blocked algorithms. 26(4):63–74, April 1991.

[96] W. Langhans, J. Schmidli, O. Fuhrer, S. Bieri, and C. Schär. Long-term simulations of thermally driven flows and orographic convection at convection-parameterizing and cloudresolving resolutions. *Journal of Applied Meteorology and Climatology, 52:1490–1510*, 2013.

[97] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. MLIR: Scaling compiler infrastructure for domain specific computation. In *Proceedings of the 19th ACM/IEEE International Symposium on Code Generation and Optimization (CGO'21)*, 2021.

[98] Chris Lavin and Alireza Kaviani. RapidWright: Enabling custom crafted implementations for FPGAs. In *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM'18)*, pages 133–140, 2018.

[99] S. Lee, J. Kim, and J. S. Vetter. OpenACC to FPGA: A framework for directive-based high-performance reconfigurable computing. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS'16)*, pages 544–554, 2016.

[100] Y. Y. Leow, C. y. Ng, and W. f. Wong. Generating hardware from OpenMP programs. In *2006 IEEE International Conference on Field Programmable Technology (FPT'06)*, pages 73–80, 2006.

[101] Jiajie Li, Yuze Chi, and Jason Cong. HeteroHalide: From image processing DSL to efficient FPGA acceleration. In *Proceedings of the 28th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*, pages 51–57, 2020.

[102] Colin Yu Lin, Hayden Kwok-Hay So, and Philip HW Leong. A model for matrix multiplication performance on FPGAs. In *IEEE 21st International Conference on Field Programmable Logic and Applications (FPL'11)*, pages 305–310, 2011.

[103] Taylor Lloyd, Artem Chikin, Erick Ochoa, Karim Ali, and Jose Nelson Amaral. A case for better integration of host and target compilation when using OpenCL for FPGAs. In *Proceedings of the 4th International Workshop on FPGAs for Software Programmers (FSP'17)*. VDE, 2017.

[104] Wang Luzhou et al. Domain-specific language and compiler for stencil computation on fpga-based systolic computational-memory array. In *Proceedings of the 8th International Conference on Reconfigurable Computing: Architectures, Tools and Applications (ARC'12)*, pages 26–39. Springer-Verlag, 2012.

[105] G. Martin and G. Smith. High-level synthesis: Past, present, and future. *IEEE Design Test of Computers*, 26(4):18–25, 2009.

[106] Kazuaki Matsumura, Hamid Reza Zohouri, Mohamed Wahib, Toshio Endo, and Satoshi Matsuoka. AN5D: automated stencil framework for high-degree temporal blocking on GPUs. In *Proceedings of the 18th ACM/IEEE International Symposium on Code Generation and Optimization (CGO'20)*, pages 199–211, 2020.

[107] Maxeler Technologies. Programming MPC systems (white paper). 2013.

[108] Gordon E Moore. Cramming more components onto integrated circuits. *Electronics*, pages 114–117, 1965.

[109] Duncan J.M Moss, Srivatsan Krishnan, Eriko Nurvitadhi, Piotr Ratuszniak, Chris Johnson, Jaewoong Sim, Asit Mishra, Debbie Marr, Suchit Subhaschandra, and Philip H.W. Leong. A customizable matrix multiplication framework for the Intel HARPv2 Xeon+FPGA platform: A deep learning case study. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*, pages 107–116, 2018.

[110] X. Niu, J. G. F. Coutinho, Y. Wang, and W. Luk. Dynamic stencil: Effective exploitation of run-time resources in reconfigurable clusters. In *2013 International Conference on Field-Programmable Technology (FPT'13)*, pages 214–221, 2013.

[111] X. Niu, Q. Jin, W. Luk, Q. Liu, and O. Pell. Exploiting run-time reconfiguration in stencil computation. In *IEEE 22nd International Conference on Field Programmable Logic and Applications (FPL'12)*, pages 173–180, 2012.

[112] Xinyu Niu, Thomas C. P. Chau, Qiwei Jin, Wayne Luk, Qiang Liu, and Oliver Pell. Automating elimination of idle functions by runtime reconfiguration. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 8(3), 2015.

[113] Carlos Osuna, Tobias Wicky, Fabian Thuering, Torsten Hoefler, and Oliver Fuhrer. Dawn: a high-level domain-specific language compiler toolchain for weather and climate applications. *Supercomputing Frontiers and Innovations (SFI)*, 7(2):79–97, 2020.

[114] A. Papakonstantinou, K. Gururaj, J. A. Stratton, D. Chen, J. Cong, and W. M. W. Hwu. FCUDA: Enabling efficient compilation of CUDA kernels onto FPGAs. In *2009 IEEE 7th Symposium on Application Specific Processors (SASP'09)*, pages 35–42, 2009.

[115] Constantine D. Polychronopoulos. Loop coalescing: A compiler transformation for parallel machines. Technical report, Illinois Univ., Urbana (USA), 1987.

[116] Constantine D. Polychronopoulos. Advanced loop optimizations for parallel computers. In *ICS*, pages 255–277, 1988.

[117] A. Possner, E. Zubler, O. Fuhrer, U. Lohmann, and C. Schär. A case study in modeling lowlying inversions and stratocumulus cloud cover in the bay of Biscay. *Weather and Forecasting, 29(2):289–304*, 2014.

[118] Louis-Noël Pouchet et al. Polybench: The polyhedral benchmark suite, 2012.

[119] Louis-Noel Pouchet, Peng Zhang, P. Sadayappan, and Jason Cong. Polyhedral-based data reuse optimization for configurable computing. In *Proceedings of the 21st ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA'13)*, pages 29–38, 2013.

[120] Jing Pu, Steven Bell, Xuan Yang, Jeff Setter, Stephen Richardson, Jonathan Ragan-Kelley, and Mark Horowitz. Programming heterogeneous systems from an image processing DSL. *ACM Transactions on Architecture and Code Optimization (TACO)*, 14(3):26, 2017.

[121] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'13)*, pages 519–530, 2013.

[122] E. Rucci, C. García, G. Botella, A. D. Giusti, M. Naiouf, and M. Prieto-Matias. Smith-Waterman protein search with OpenCL on an FPGA. In *2015 IEEE Trustcom/BigDataSE/ISPA*, volume 3, pages 208–213, 2015.

[123] Shane Ryoo, Christopher I. Rodrigues, Sara S. Baghsorkhi, Sam S. Stone, David B. Kirk, and Wen-mei W. Hwu. Optimization principles and application performance evaluation of a multithreaded GPU using CUDA. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'08)*, pages 73–82, 2008.

[124] K. Sano, Y. Hatsuda, and S. Yamamoto. Multi-FPGA accelerator for scalable stencil computation with constant memory bandwidth. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 25(3):695–705, 2014.

[125] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. LLHD: A multi-level intermediate representation for hardware description languages. In *Proceedings*

of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'20), pages 258–271, 2020.

[126] Michele Scquizzato and Francesco Silvestri. Communication lower bounds for distributed-memory computations. *arXiv:1307.1805*, 2013.

[127] Sean O. Settle. High-performance dynamic programming on FPGAs with OpenCL. In *Proceedings of the 2013 IEEE High Performance Extreme Computing Conference (HPEC'13)*, 2013.

[128] G. Singh, D. Diamantopoulos, C. Hagleitner, J. Gómez-Luna, S. Stuijk, O. Mutlu, and H. Corporaal. NERO: A near high-bandwidth memory stencil acceleratorfor weather prediction modeling. In *IEEE 30th International Conference on Field Programmable Logic and Applications (FPL'20)*, pages 263–269, 2020.

[129] G. Singh, D. Diamantopoulos, C. Hagleitner, S. Stuijk, and H. Corporaal. NARMADA: Near-memory horizontal diffusion accelerator for scalable stencil computations. In *IEEE 29th International Conference on Field Programmable Logic and Applications (FPL'19)*, pages 263–269, 2019.

[130] Udayan Sinha. Enabling impactful DSP designs on FPGAs with hardened floating-point implementation. *Altera White Paper, WP-01227-1.0 (Aug. 2014)*.

[131] Scott Sirowy and Alessandro Forin. Where's the beef? why FPGAs are so fast. *Microsoft Research, Microsoft Corp., Redmond, WA*, 2008.

[132] Joseph Smagorinsky. General circulation experiments with the primitive equations. *Monthly Weather Review, Vol. 91, No. 3*, 1963.

[133] Gordon D. Smith. *Numerical solution of partial differential equations: finite difference methods.* Oxford University Press, 1985.

[134] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *Euro-Par 2011 Parallel Processing*, pages 90–109. Springer Berlin Heidelberg, 2011.

[135] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio. A common backend for hardware acceleration on FPGA. In *2017 IEEE International Conference on Computer Design (ICCD'17)*, pages 427–430, 2017.

[136] Guy L. Steele. Arithmetic shifting considered harmful. *ACM SIGPLAN Notices*, 12(11):61–69, 1977.

[137] Volker Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, 13(4):354–356, August 1969.

[138] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized OpenCL-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'16)*, pages 16–25, 2016.

[139] Allen Taflove and Susan C. Hagness. Computational electrodynamics: The finite-difference time-domain method. *Norwood, 2nd Edition, MA: Artech House*, 1995.

[140] Yaman Umuroglu, Nicholas J. Fraser, Giulio Gambardella, Michaela Blott, Philip Leong, Magnus Jahre, and Kees Vissers. FINN: A framework for fast, scalable binarized neural network inference. In *Proceedings of the 25th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*, pages 65–74, 2017.

[141] Didem Unat, Anshu Dubey, Torsten Hoefler, John Shalf, Mark Abraham, et al. Trends in data locality abstractions for HPC systems. *IEEE Transactions on Parallel and Distributed Systems (TPDS)*, 28(10), Oct. 2017.

[142] Jessica Vandebon, Jose Gabriel De Figueiredo Coutinho, Wayne Luk, and Eriko Nurvitadhi. Enhancing high-level synthesis using a meta-programming approach. *IEEE Transactions on Computers*, 2021.

[143] Vincent Vanhoucke, Andrew Senior, and Mark Z. Mao. Improving the speed of neural networks on CPUs. In *Proc. Deep Learning and Unsupervised Feature Learning NIPS Workshop*, volume 1. Citeseer, 2011.

[144] Anshuman Verma, Ahmed E. Helal, Konstantinos Krommydas, and Wu-Chun Feng. Accelerating workloads on FPGAs via OpenCL: A case study with OpenDwarfs. Technical report, Department of Computer Science, Virginia Polytechnic Institute & State University, 2016.

[145] Hasitha Muthumala Waidyasooriya and Masanori Hariyama. Multi-FPGA accelerator architecture for stencil computation exploiting spacial and temporal scalability. *IEEE Access*, 7, 2019.

[146] Hasitha Muthumala Waidyasooriya, Yasuhiro Takei, Shunsuke Tatsumi, and Masanori Hariyama. OpenCL-based FPGA-platform for stencil computation and its optimization methodology. *TPDS*, 28(5):1390–1402, May 2017.

[147] Shuo Wang and Yun Liang. A comprehensive framework for synthesizing stencil algorithms on FPGAs using OpenCL model. In *Proceedings of the 54th Annual Design Automation Conference (DAC'17)*, 2017.

[148] Michael Weiss. Strip mining on SIMD architectures. In *Proceedings of the 5th ACM/SIGARCH International Conference on Supercomputing (ICS'91)*, pages 234–243, 1991.

[149] Dennis Weller, Fabian Oboril, Dimitar Lukarski, Juergen Becker, and Mehdi Tahoori. Energy efficient scientific computing on FPGAs using OpenCL. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*, pages 247–256, 2017.

[150] T. Weusthoff, F. Ament, M. Arpagaus, and M. W. Rotach. Assessing the benefits of convection-permitting models by neighborhood verification: Examples from map d-phase. *Monthly Weather Review, 138:3418–3433*, 2010.

[151] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Communications of the ACM*, 52(4):65–76, 2009.

[152] Michael Joseph Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, 1982.

[153] Ephrem Wu, Xiaoqian Zhang, David Berman, and Inkeun Cho. A high-throughput reconfigurable processing array for neural networks. In *IEEE 27th International Conference on Field Programmable Logic and Applications (FPL'17)*, 2017.

[154] William A. Wulf and Sally A. McKee. Hitting the memory wall: implications of the obvious. *ACM SIGARCH Computer Architecture News*, 23(1):20–24, 1995.

[155] Xilinx. L3 API GEMM benchmark. https://xilinx.github.io/Vitis_Libraries/blas/2021.2/user_guide/L3/L3_benchmark_gemm.html. Accessed December 16, 2021.

[156] Xilinx. QDMA Subsystem for PCI Express v4.0, PG302 (v4.0) April 15, 2021. https://www.xilinx.com/support/documentation/ip_documentation/qdma/v4_0/pg302-qdma.pdf. Accessed August 9, 2021.

[157] A. P. Yershov. ALPHA – an automatic programming system of high efficiency. *Journal of the ACM*, 13(1):17–24, January 1966.

[158] 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 *The 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'20)*, pages 86–96, 2020.

[159] Jialiang Zhang and Jing Li. Improving the performance of OpenCL-based FPGA accelerator for convolutional neural network. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'17)*, pages 25–34, 2017.

[160] Zhiru Zhang, Yiping Fan, Wei Jiang, Guoling Han, Changqi Yang, and Jason Cong. AutoPilot: A platform-based ESL synthesis system. In *High-Level Synthesis*, pages 99–112. Springer, 2008.

[161] L. Zhuo and V. K. Prasanna. Scalable and modular algorithms for floating-point matrix multiplication on FPGAs. In *Proceedings of the 18th International Parallel and Distributed Processing Symposium (IPDPS'04)*, 2004.

[162] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Guillermo Indalecio Fernández, Timo Schneider, Mathieu Luisier, and Torsten Hoefler. A data-centric approach to extreme-scale ab initio dissipative quantum transport simulations. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'19)*, 2019.

[163] Alexandros Nikolaos Ziogas, Tal Ben-Nun, Timo Schneider, and Torsten Hoefler. NPBench: A benchmarking suite for high-performance NumPy. In *Proceedings of the 35th ACM International Conference on Supercomputing (ICS'21)*, pages 63–74, 2021.

[164] Alexandros Nikolaos Ziogas, Timo Schneider, Tal Ben-Nun, Alexandru Calotoiu, Tiziano De Matteis, Johannes de Fine Licht, Luca Lavarini, and Torsten Hoefler. Productivity, Portability, Performance: Data-centric Python. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'21)*, 2021.

[165] Hamid Reza Zohouri, Naoya Maruyama, Aaron Smith, Motohiko Matsuda, and Satoshi Matsuoka. Evaluating and optimizing OpenCL kernels for high performance computing with FPGAs. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'16)*, 2016.

[166] Hamid Reza Zohouri, Artur Podobas, and Satoshi Matsuoka. Combined spatial and temporal blocking for high-performance stencil computation on FPGAs using OpenCL. In *Proceedings of the 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA'18)*, pages 153–162, 2018.