THÈSE

Pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ GRENOBLE ALPES

École doctorale : MSTII - Mathématiques, Sciences et technologies de l'information, Informatique
Spécialité : Informatique
Unité de recherche : Laboratoire d'Informatique de Grenoble

Génération automatique de modèles de performance pour architectures hétérogènes

Automatic generation of performance model for heterogeneous architectures

Présentée par :

Nicolas DERUMIGNY

Direction de thèse :

Fabrice RASTELLO DIRECTEUR DE RECHERCHE, INRIA CENTRE GRENOBLE-RHONE-ALPES	Directeur de thèse
Louis-Noël POUCHET ASSOCIATE PROFESSOR, COLORADO STATE UNIVERSITY	Co-directeur de thèse

Rapporteurs :

SEBASTIAN HACK
FULL PROFESSOR, UNIVERSITÄT DES SAARLANDES
ERVEN ROHOU
DIRECTEUR DE RECHERCHE, INRIA CENTRE RENNES-BRETAGNE ATLANTIQUE

Thèse soutenue publiquement le **13 décembre 2023**, devant le jury composé de :

FREDERIC PETROT, PROFESSEUR DES UNIVERSITES, GRENOBLE INP	Président
FABRICE RASTELLO, DIRECTEUR DE RECHERCHE, INRIA CENTRE GRENOBLE-RHONE-ALPES	Directeur de thèse
LOUIS-NOËL POUCHET, ASSOCIATE PROFESSOR, COLORADO STATE UNIVERSITY	Co-directeur de thèse
SEBASTIAN HACK, FULL PROFESSOR, UNIVERSITÄT DES SAARLANDES	Rapporteur
ERVEN ROHOU, DIRECTEUR DE RECHERCHE, INRIA CENTRE RENNES-BRETAGNE ATLANTIQUE	Rapporteur
ZAKS AYAL, INGENIEUR, MOBILEYE	Examinateur

Invités :

FRANCISCO ORTEGA
ASSISTANT PROFESSOR, COLORADO STATE UNIVERSITY
JAMES WILSON
FULL PROFESSOR, COLORADO STATE UNIVERSITY

Titre : Techniques d'Optimisation du Débit pour Architectures Hétérogènes

Mots clés : CPU, architecture, FPGA, synthèse haut-niveau, débit

Résumé :

Alors que les processeurs deviennent de plus en plus complexes et nombreux, leur optimisation manuelle est un processus coûteux et propice à l'erreur. Ce manuscript vise à guider les programmeurs et designers d'accélérateurs via une étude parallèle des impératifs logiciels et matériels qui y sont liés.

Dans la première partie, nous présentons un programme capable de déterminer automatiquement un modèle de performances décrivant le comportement d'un processeur. Dans la seconde partie, nous couvrons l'optimisation de design d'accélérateurs dédiés dans le cadre de la synthèse haut-niveau, sous l'aspect du partage de resources.

Title: Throughput Optimization Techniques for Heterogeneous Architectures

Keywords: CPU, architecture, FPGA, HLS, throughput

Abstract:

While processors are becoming more and more complex, their manual optimization is a costly and tedious process. This manuscript aims at guiding programmers and hardware designers by proposing a two-sided study of both the software and hardware constraints associated with high-performance accelerator usage.

In the first part, we present a framework able to automatically built a performance model describing the behavior of a CPU. In the second part, we cover the optimization process of dedicated hardware accelerator in the context of high-level synthesis under the angle of resource sharing.

ABSTRACT

THROUGHPUT OPTIMIZATION TECHNIQUES FOR HETEROGENEOUS ARCHITECTURES

Moore's Law has allowed during the past 40 years to exponentially increase transistor density of integrated circuits. As a result, computing devices ranging from general-purpose processors to dedicated accelerators have become more and more complex due to the specialization and the multiplication of their compute units. Therefore, both low-level program optimization (e.g. assembly-level programming and generation) and accelerator design must solve the issue of efficiently mapping the input program computations to the various chip capabilities. However, real-world chip blueprints are not openly accessible in practice, and their documentation is often incomplete. Given the diversity of CPUs available (Intel's / AMD's / Arm's microarchitectures), we tackle in this manuscript the problem of automatically inferring a performance model applicable to fine-grain throughput optimization of regular programs. Furthermore, when order of magnitude of performance gain over generic accelerators are needed, domain-specific accelerators must be considered; which raises the same question of the number of dedicated units as well as their functionality. To remedy this issue, we present two complementary approaches: on one hand, the study of single-application specialized accelerators with an emphasis on hardware reuse, and, on the other hand, the generation of semi-specialized designs suited for a user-defined set of applications.

Tout au long des 40 dernières années, la loi de Moore a permis d'augmenter de façon exponentielle la densité des transistors des circuits intégrés. En conséquence, les appareils informatiques – allant des processeurs centraux aux accélérateurs dédiés, sont devenus de plus en plus complexes du fait de la multiplicité croissante de leurs unités de calcul. Par conséquent, à la fois le design de puces et l'optimisation logicielle (qu'elle soit manuelle, en assembleur, ou effectuée par un compilateur) doivent résoudre le problème de l'association efficace des calculs variés du programmes aux unités présentes sur le matériel. Or, les caractéristiques de ces unités ne sont pas toujours disponibles. Devant la diversité des CPU du commerce (Intel, AMD, Arm ayant chacun leurs microarchitectures), nous nous attaquons ici au problème de la génération automatique de modèles de performance, applicables lors de l'optimisation à grain fin de programmes réguliers. De plus, dans les cas où des gains de multiples ordre de grandeur sont désirés, des accélérateurs spécifiques doivent être utilisés, ce qui pose une question similaire au niveau de l'organisation de la puce. Pour faire face à ces questions, nous proposons deux approches complémentaires : d'une part, l'étude d'accélérateurs de calcul haute performance dédiés à une unique application et, d'autre part, la génération automatique d'architectures semi-spécialisées à une famille d'applications.

ACKNOWLEDGEMENTS

Thought I would lie by telling that I have not been warned, the process of writing this manuscript revealed to be harder that anticipated. First, the initial goal of compiling the work realized in two radically different universities was completed by a worldwide pandemic and my inclination to launch myself in too many projects; which already made the task challenging – but I was still not expecting *that*.

Studying, researching is a never-ending quest of discoveries and knowledge, but also of disappointments and "walls" that sometimes cannot be overcome. In this path, I was helped by many fabulous people whom I dedicated this manuscript to. First, my two advisors Louis-Noël Pouchet and Fabrice Rastello. Louis-Noël, you taught me what to expect as a researcher, the challenges that come with it but also the will to do *good science*. Thank you for your never-ending goal of letting people do what makes them feel happy when they go to sleep, and sorry not to have baked more brioche while you were in Fort Collins – I am sure other occasions will come. Fabrice, you taught me to trust my ideas, to collaborate with other students, to stay impartial writing my work, and what a good coffee taste like. Thanks for your guidance and your patience, both of which should have been put quite to the test with me. Sorry again for the lack of *numbers*! Without both of you, I would not have the same understanding of academia.

I would also not have done this work without Corentin, whom I thank from the heart. I owe you a lot, and I apologies to have relied too much on you during deadlines – you are one of the kindest person I know for that. Thank also to my colleagues Chistophe, Stéphane, Théophile, Nicolas and Théo whose discussions where always very insightful.

However, academic work is not the only reflection of one's activities, and part of the work on CPU would have not been possible without the support of my colleagues at *Hardware & Co*. Thank you Eric for your experience, Thibaut for your character and your common sense, Matthieu for your model of regularity and organization, and all the others I forget: Thomas, David, Adrien, Kéta, and of course Guillaume H., whose timing was more than perfect.

A special thank is also more than deserved by you, Guillaume L. for your support, your passion and you time for both professional and personal matters. I owe you probably more than you think. One thing leading to another, I also want to reluctantly thank Pascal. You introduced me to fantastic people and taught me the hard way how to have the courage to stand up and change everything. For better or worse, this manuscript would been deeply modified if I had not met you.

Thanks also to my family for leaving me enough independence and letting me trace my own path while still being there when needed: mom, dad, and my sister Heloïse who is always so glad to see me coming back. Alexis, I know that I should have called you more often. You were of precious advice, thank you to let me catch glimpses of a professor's life, and for all the laughs we shared!

I also do not forget my colleague at AMD, Benoit Pradelle, Florian, Stefan and Thibault, who make me realize that a dynamic team is sufficient to go to work with a smile on one's lips.

Last but not least, I want to thank you Marie for your support, you presence, your frankness and your constant reminder that life is not just about work and side projects. You made me travel all around the world for you... literally! Thanks for the constant connection to France – and India. I once again apologize for the 2 years spent so far away from you. This is now over.

TABLE OF CONTENTS

ABSTRACT ii
ACKNOWLEDGEMENTS iv
LIST OF TABLES x
LIST OF FIGURES xii

Chapter 1 Introduction 1

Chapter 2 Background 6
2.1 Abstract Resource Model 7
2.1.1 Resources 7
2.1.2 Pipelining and Resource Sharing 8
2.1.3 Scheduling 10
2.2 General CPU Architecture 13
2.2.1 Out-of-order Execution 13
2.2.2 Front-end 14
2.2.3 Instruction Set Extensions 15

I Throughput Optimization for Superscalar Architectures 16

Chapter 3 Throughput Performance Models for Superscalar Architectures . . . 17
3.1 State of the Art 18
3.1.1 Performance Counters Derived Models 18
3.1.2 Proprietary and Ad-Hoc Tools 19
3.1.3 Comparison with the Abstract Resource Model 19
3.2 Conjunctive and Disjunctive Resource Mapping 21
3.2.1 Tripartite and Bipartite Port Mapping 21
3.2.2 Dual Representation of a Disjunctive Mapping 22
3.3 Code Generation for Accurate Throughput Measurements 25
3.3.1 Padding Microkernels of Multiple Instructions 25
3.3.2 Generation of Basic Blocks with no Dependencies 26
3.3.3 Limitations of the Code Generator 28

Chapter 4 PALMED: Efficient Automated Characterisation of Throughput in Superscalar Architetures 30
4.1 Complete Flow of PALMED 31
4.2 Selection Heuristics 33
4.2.1 How *not to* Benchmark the Whole ISA 33
4.2.2 Max Clique: Very Basic Instructions 34
4.2.3 Min Order: Most Greedier Instructions 35
4.3 Finding the Exact Core Mapping 37
4.3.1 Determine Hazardous Instructions 37
4.3.2 Bipartite Weight Problem (BWP) 38
4.3.3 Characterize Resources ($\texttt{LP}_1$) 38
4.3.4 Find Saturating Kernels ($\texttt{LP}_2$) 39

4.4 Faster approximation of the Complete Resource Mapping Problem 41
4.4.1 Finding the Shape of the Mapping: ($\texttt{ILP}_1$) . . . 41
4.4.2 Find Saturating Kernels ($\texttt{LP}_2$) . . . 42
4.5 Complete Mapping ($\texttt{LP}_{\texttt{AUX}}$) . . . 44
4.6 Evaluation: Basic Blocks Throughput Prediction without Dependencies . . . 45
4.6.1 Exact Port Mapping Recovery . . . 45
4.6.2 Real-world CPU Throughput Prediction . . . 46

Chapter 5 Formal Proofs of Convergence of PALMED . . . 52
5.1 Equivalence of the Abstract Resource Model and the Port Mapping Model . . . 53
5.1.1 Primary Definitions . . . 53
5.1.2 Equivalence between Disjunctive and Conjunctive formulations 55
5.2 Selection of Basic Instructions . . . 58
5.2.1 Primary Definitions . . . 58
5.2.2 Max Clique: Selection of Independent Instructions . . . 60
5.2.3 Min Order: Selection of the Instructions using Resources of High Throughput . . . 60
5.3 Convergence to the Complete Mapping . . . 62

II Generation of Throughput-Efficient Accelerators 66

Chapter 6 High-Level Synthesis . . . 67
6.1 Introduction . . . 68
6.1.1 Overview . . . 68
6.1.2 Annotation of the Source Code with `pragma` . . . 71
6.1.3 Controlling Resource Usage . . . 77
6.1.4 Handling Off-chip Communications . . . 78
6.1.5 Selection of the Design Frequency . . . 79
6.2 Generic Toolchain's Limitations . . . 81
6.2.1 Parallel Operations . . . 81
6.2.2 Frequency Domains . . . 81
6.2.3 DSP Primitives . . . 82
6.3 Toolchain's Limitations for Resource-Shared Design Generation . 83
6.3.1 Expressing Shared Modules . . . 83
6.3.2 Overhead of Operation Clustering . . . 83
6.3.3 Fine-grain Execution Pipeline Generation . . . 85
6.3.4 Accurate Measuring of the Execution Time . . . 85

Chapter 7 Towards a General Formulation of the Resource Sharing Problem . 87
7.1 Existing Resource Sharing Techniques . . . 88
7.1.1 Basic Blocks-Level Resource Sharing . . . 88
7.1.2 Loop-based Resource Sharing for Throughput-based Optimization 90
7.2 Compute Unit: Definition . . . 94
7.3 Resource Estimation of Compute Units . . . 96
7.3.1 DSP Estimation . . . 96

7.3.2 LUT Estimation . . . 96
7.3.3 Storage Units Estimation . . . 97
7.3.4 Microbenchmarking CDAGs . . . 98
7.3.5 Combination of Sequential and Parallel Models . . . 98
7.3.6 Evaluation . . . 100
7.4 Latency Estimation of Compute Units . . . 103
7.4.1 Formula and Micro-benchmarking . . . 103
7.4.2 Evaluation . . . 103
7.5 Naive Convex Encoding of the Resource Sharing Problem . . . 105
7.5.1 Variables . . . 105
7.5.2 Objective Function . . . 106
7.5.3 Constraints . . . 106
7.6 Real-life Implementation, Heuristics . . . 110
7.6.1 Exact Implementation: Scaling . . . 110
7.6.2 A Faster Greedy Approximation . . . 111

Chapter 8 Automated Generation of Semi-generic Throughput-oriented Accelerators . . . 116
8.1 Illustrative Examples . . . 117
8.1.1 Data Centering . . . 117
8.1.2 Center, Correlation and Multi-purpose Acceleration . . . 118
8.1.3 Accelerator Creation and Usage Workflow . . . 120
8.2 Kernel Merging for Multi-Functionalities . . . 122
8.2.1 Polyhedral Kernel Representation . . . 122
8.2.2 Decomposition of Applications into Kernels . . . 123
8.2.3 Kernel Set and Workloads . . . 125
8.2.4 Kernel Merging . . . 128
8.2.5 Profitability Criteria . . . 129
8.3 Accelerator Implementation . . . 132
8.3.1 Structure of the Accelerator . . . 132
8.3.2 Iteration Vector Generator (IVG) . . . 132
8.3.3 Functional Units . . . 132
8.3.4 Loop Bound Generator (LBG) . . . 134
8.3.5 Loop Control Logic . . . 134
8.3.6 Off-Chip Communications . . . 134
8.3.7 Access to the Local Buffer . . . 134
8.4 Experimental Results . . . 135
8.4.1 Linear Algebra . . . 136
8.4.2 Correlation . . . 139
8.4.3 Scaling and Comparison . . . 140
8.5 Limitations . . . 143
8.5.1 Routing between FUs and Buffers . . . 143
8.5.2 Merging of Kernels with Different Iteration Space . . . 143
8.5.3 Data Reuse: Optimizing Buffer Communication . . . 143
8.5.4 No Control Flow Instructions . . . 143
8.5.5 Vectorization of the FUs . . . 144
8.6 Related Work . . . 145
8.6.1 Generic Resource-shared Designs . . . 145

8.6.2 DSP-dedicated Resource Sharing on Overlay Architectures . . 147

Chapter 9 Conclusion and Future Research Directions 150

LIST OF TABLES

2.1 Example of resource for different hardware targets 8

4.1 Summary of key features of PALMED vs. related work 30
4.2 Number of detected equivalent classes and resources for an ideal CPU simulated from uops.info's mapping 46
4.3 Main features of the obtained mappings 49

7.1 Resource, latency and throughput estimates of two architectures for an accelerator executing 10 successive instances of the program from Fig. 7.3c 95
7.2 Resource predictor accuracy 100
7.3 Resource predictor coefficients for sequential composition of additions (a) and multiplications (b) 101
7.4 Resource predictor coefficients for parallel composition of additions (a) and multiplications (b) 101
7.5 Resource predictor accuracy on FP16 multipliers and adders 101
7.6 Latency predictor coefficients on FP16 multipliers and adders 103
7.7 Greedy scheduling-placement solving time and number of operations for one Nx16 DWT iteration. 113

8.1 Performance and area metric for coarse-grained pipeline (CGP) vs coarse grained replication (CGR) of CENTER accelerator (matrices of size 64×64, FP16 data type) 119
8.2 Performance and area metric for coarse-grain pipelined correlation, sum accelerator and dedicated accelerator 120
8.3 Performance per area metric for coarse-grain pipelined correlation, sum accelerator and dedicated accelerator 120
8.4 Configuration of the LA-GA accelerator and the Correlation accelerator . . . 135
8.5 Supported kernels list, by either the Correlation or the Linear Algebra accelerator 136
8.6 Execution time (a) and performance-per-area (b) of a custom IP optimized for Max Sharing (MS) and Max Throughput (MT) and the Generic Accelerator, both with pure HLS FUs (LA-GA-HLS) and non-HLS FUs (LA-GA) for several linear algebra benchmarks 137
8.7 Execution time (a) and performance-per-area (b) of a custom IP optimized for Max Sharing (MS) and Max Throughput (MT) and the Generic Accelerator, both with pure HLS FUs (LA-GA-HLS) and non-HLS FUs (LA-GA) for batched linear algebra benchmarks 138
8.8 Execution time (a) and performance-per-area (b) of a custom IP optimized for Max Sharing (MS) and Max Throughput (MT) and the Generic Accelerator, both with pure HLS FUs (CORR-GA-HLS) and non-HLS FUs (CORR-GA) for correlation subexpressions 138
8.9 Execution time (a) and performance-per-area (b) of a custom IP optimized for Max Sharing (MS) and Max Throughput (MT) and the Generic Accelerator, both with pure HLS FUs (CORR-GA-HLS) and non-HLS FUs (CORR-GA) for batched correlation subexpressions 139

8.10 Scaling properties of the LA-GA accelerator 141
8.11 Maximum frequency achieved for each design 141
8.12 Performance per area comparison with data extracted from other published accelerators 142

LIST OF FIGURES

2.1 Non-pipelined (a) versus pipelined (b) designs 9
2.2 Resource shared (a) and non resource shared (b) architectures 10
2.3 Abstract resource model: resource usage (a) and corresponding loads/schedules 10
2.4 Schematic view of a CPU and life cycle of an instruction 13
2.5 Schematic view of Intel's Skylake front-end 14

3.1 Mappings computed for a few SKL-SP instructions. 22
3.2 Disjunctive port assignment examples 23
3.3 Example of assembly kernels and resulting dependencies produced by the old (a) and the new (b) allocator for four `ROL` (quadword or register/memory) and one `ORQ` (register rotation left of a constant factor 1) 27
3.4 μOPs generated for SSE vector add (VADDPD) with one operand from memory (M128) and two register operands (XMM) and associated latency 28

4.1 High-level view of the algorithms of Palmed 31
4.2 Example of Max Clique instructions selection with instruction and port usage from Intel's Skylake microarchitecture. Blue instructions are selected instructions, edges symbolize instructions with additive IPC. 35
4.3 Example of Min Order instructions selection with instruction and port usage from Intel's Skylake microarchitecture. Edges symbolize the pre-order relation between instructions. 36
4.4 Accuracy of IPC predictions compared to native execution of Palmed versus uops.info, PMEvo, IACA and llvm-mca on SPEC CPU2017 and PolyBench/C 4.2 50

5.1 Conjunctive resource mapping (a) and its extended form (b); both normalized 59
5.2 Saturating benchmarks S and instructions to analyse I: individual uses (5.2b and 5.2a), and benchmark $S^{4\cdot\overline{S}}I^{\overline{I}}$ (5.2c) 62

6.1 Generic architecture of an FPGA 68
6.2 HLS complete design flow for FPGA 70
6.3 Cost of chip design for several lithographic technologies [1], in millions of dollars 72
6.4 `stp`, `flp` and `frp` pipelining types 73
6.5 Non-rewinded (a) vs rewinded (b) loops 74
6.6 Array partition types 75
6.7 Loops, dataflow and concurrent execution: code (a) and corresponding execution times with /without dataflow (b) 76
6.8 Example of resource control via the `HLS allocation` pragma on operations (a) and functions (b) for `half` (FP16) data type 78
6.9 ASAP scheduling in Vitis: operators (a) and functions (b) 81
6.10 Inlining (a) and operator reuse (b) 83
6.11 Single-operation (a) and function-based (b) explicit resource binding 84
6.12 Erroneous (a) and correct (b) HLS execution time measurement using an on-chip counter 86

7.1 Example of design exposing shareable opportunities 89

7.2 Squeletton code (a), fully pipelined (c) and non-fully pipelined (b) accelerators for DWT. Red squares indicated compute units, bright if occupied, dark if idle. Despite being slower in terms of latency, the non-fully pipelined version achieves more problems solved (15) than the fully pipelined one (12) with a similar resource and time budget. 92
7.3 Two FPGA accelerator architecture, using fully pipelined (a) and non-fully pipelined (b) CUs, for the program (c) 95
7.4 Resource predictor (red line) versus pre-P&R (blue bar) for several FPGA resources for reductions of additions. $half_a_b$ denotes a reduction using a operation with an II of b on FP16 data type 99
7.5 Resource predictor (red line) versus post-P&R values (blue bar) for FF predictions on mixes of 3 and 4 operators CU 102
7.6 Latency predictor (red line) versus HLS ground truth (blue bar). $[ops]_a$ denotes a CDAG linearised as in the list `[ops]` replicated a times 103
7.7 Scaling of the ILP formulation with the number of node in the target AST for three loop bodies. Missing points time out (more than 10 hours of solving). . 111
7.8 LUT and FF used as percentage of the chip size for N by 16 DWT designs generated using our CU greedy placement heuristic (a) and a pure HLS version (b) 113
7.9 Performance metrics (latency and II) for a the vanilla resource-constraint HLS design (non-pipelined) and the greedily scheduled one. 114

8.1 CENTER naive implementation 117
8.2 CENTER coarse-grain pipelined implementation 118
8.3 Workflow of the creation of the accelerator 121
8.4 Workflow of the usage of the accelerator 121
8.5 Example: General Matrix Multiplication, split in two kernels 122
8.6 Example of programs with trivially kernelisable (a) and non-trivially kernelisable (b) 123
8.7 Example of loop fission: corresponding MatMult code before (a) and after (b) transformation 124
8.8 Example of a 4 instructions input program 127
8.9 Example code structure of a two merged kernels 130
8.10 Layout of the Generic Accelerator 132
8.11 Anatomy of an HLS-generated FU: control path in light blue, multiplier path in red and sub/adder path in dark blue 133
8.12 Anatomy of the DSP48E1 as described by the vendor documentation [2] . . 148

Chapter 1
Introduction

Heterogeneous Architectures

While Moore's Law – doubling of the transistor density every two years – has slightly faded in the past 10 years, its applicability on single-thread workloads has at the same time greatly reduced. Because of thermal limitations, the computing frequency of general-purpose accelerators reached a hard limit in the early 2000s by stabilizing at the gigahertz scale – a barrier still in place today. To overcome this limitation, processors became multicore: coarse-grain replication of a general-purpose unit called core was implemented as the best cost-effective technique to use available silicon with a significant impact on performance while stabilizing power consumption generation over generation. This split performance measurement of chips in two main workloads: single-threaded, where only one core can actively compute, and multithreaded, where all integrated cores are used in parallel.

Given the lack of frequency gain, one could deduce that single-threaded performance has stagnated over the past fifteen years, as the simple solution to accelerated computation ("run the chip faster") was not doable in practice. This was not the case. Single-threaded performance has continued to improve, though at a slower pace. This increase has been reached part by the slight frequency progression allowed by lithographic advances (a 2005 Intel Pentium 4 reached at most 4 GHz, whereas state-of-the-art processors break the 6 GHz barrier), but mostly because of internal chips layout improvements. Architecturally speaking, Moore Law's transistors are used to build specialized logic, deriving from the one-size-fits-all initial paradigm to a set of small heterogeneous units orchestrated by a scheduler. Indeed, nowadays' CPUs integrate a variety of accelerators such as vector units, artificial neural networks or cryptographic accelerators, etc; but also multiple simple ALUs and address generation units to be able to exploit efficiently non-specific workloads. In parallel, memory subsystems have also greatly benefited from this greater density to increase the available amount of memory directly accessible by the processor by building intricate cache hierarchy and steadily improving cache size over generations.

Latest innovations (ARM's big.LITTLE technology, Intel's Alder Lake architecture) even showed the pertinence of heterogeneity at the *core* level. In these chips, several types of cores realizing different performance/size compromises are integrated: high-performance, large cores are present in order to improve execution time of single-threaded workloads, whereas multithreaded ones benefit from the numerous efficient cores, whose higher count was made possible by their smaller silicon space requirement.

Note that hardware specialization also applied to memory's logic: data paths and memory hierarchy also became more and more intricate, trading chip space for performances on chosen applications and memory access patterns. This complex hierarchy has

also proved to be challenging to exploit at its complete potential, but such research topics fall out of the scope of this manuscript.

Similarly, GPUs now integrate hardware media encoders, ray tracing cores, systolic matrix multiplication units and optical flow accelerator to better suit the needs of their users, also shifting from their single instruction multiple thread paradigm to a composition of parallel application-specific accelerators on the same chip. While being less flexible than GPUs, dedicated accelerators' architecture are also more heterogeneous: depending on their target, designs must achieve a unique balance between high-performance compute units for critical workloads (often replicated to achieve a target throughput), and slower units for less frequent thus required operations.

This diversification and specialization of dedicated hardware units lead to more and more complex architectures, whichever the type of the chip. As a consequence, the peak theoretical performance is usually achievable only on a very limited, often non-representative set of benchmarks.

Optimization and Heterogeneous Architecture

Optimization in the topic of software programming designates the capacity of achieving improvement in one metric, usually one of the followings:

- The energy consumed during one solving of a problem.
- The throughput defined as the number of instances of a problem solved in average per unit of time.
- The latency, defined as the total amount of time elapsed between the start of the computation and its termination for one instance of a problem.

When the target workload is fixed, the mean usage over time of the compute units of an accelerator, called *occupancy*, is often used as a measure of the exploited potential of the architecture. The relevance of this metric is based on the underlying assumption that expressing part of the computation under a form that is executable by dedicated hardware will lead to a lower execution time, which is true when the runtime cost of transformation of the computation does not exceed the speedup granted by the dedicated unit.

Therefore, automated optimization of an application to a heterogeneous chip requires an accurate performance model for each of its units. As some components may be shared between them (for example, a matrix unit on a CPU may share some components with its vector unit, so that some combinations of concurrent matrix and vector operations are slower or even not possible), the task can be more tedious than it may seem. Moreover, the number of elementary operations performed in the target workload must also be accurately estimated, as the key problem relies in the mapping of chains of these atomic operations to the on-chip computing resources.

In this manuscript, this aspect of the optimization framework is solved by restricting to classes of programs for which the number and the kind of all operations is *statically computable*, i.e. does not depend on any value of the input data. Though we acknowledge that this restriction does not cover the breadth of high-performance computing applications, such programs still cover workloads from a variety of domains such as linear algebra, machine learning, computer vision and simulation.

The question of optimization applied to heterogeneous architecture is two-sided, depending on what is considered fixed and what can be modified: from a software programmer's point of view, a fixed architecture has to be exploited as much as possible to ensure minimal execution time. From a chip designer's, the goal is to calibrate an accelerator to best suit the target workload, which translates to deciding the best compromise between flexibility, performances and power consumption.

In this manuscript, we discuss both points of view and present a complete approach to low-level optimization on heterogeneous chips, paving the way for future hardware-software co-designs.

Contributions

Performance Model Generation

Automated optimization is vain without an accurate estimation of the effect of one hardware/software transformation on the overall execution time. Therefore, we present in Chap. 4 a complete reverse-engineering framework of CPU resources resulting in an instruction-level cost model. We evaluate its accuracy on basic blocks with no dependencies extracted from SPECInt 2017 and achieve a mean error rate of 7.8 % compared to native execution, outperforming state-of-the-art tools such as llvm-mca (20.1 %) and Intel IACA (8.7 %).

A similar resource model of 3-operation FPGA compute units is also computed in Chap. 7, achieving a mean error rate of 0% on DSP, 7.4 % on LUT and 14.4 % of FF compared to post-netlist predicted values. Latencies are also predicted for very simple sequences of operations, leading to an accuracy of 3.57 % over 28 designs using 3 or 4 operations.

Formal Proofs of Software / Hardware Mappings

While several optimization techniques rely on inexact heuristics, we present in Chap. 5 and Chap. 7 the formal proofs for both our CPU reverse engineering framework and our FPGA design generator. Whereas the former converges to a unique, existing performance model matching the usual hardware-deduced state of the art, the second one presents a formulation of an optimization problem whose solution is an accelerator architecture achieving minimal execution time of the input program, given a fixed area budget for compute units.

Efficient Resource Usage

Depending on the context, on-chip resources can be considered either as a constraint or as a measure of the quality of an implementation. For example, saturating one of the always-used resources such as a memory bus is considered as the stopping point for both assembly-level program optimization and design generation, as gains in the computing parts will not translate further in actual performances due to stalls. However, in the general case, unused resources such as idle compute units are an indicator of under-utilization of the chip. In Chap. 8, we show how to create a balanced semi-generic accelerator given a set of applications to be accelerated. We evaluate the performances of the generated design on two distinct families of computation, Correlation and Linear Algebra, and show that our generic accelerator does not degrade either performance or usage by more than one order of magnitude on our tested applications.

Outline

In this manuscript, we present several approaches to optimize applications on two classes of chips: general-purpose processors (CPUs) and dedicated accelerators (FPGAs / ASICs). After a first introduction of the common concepts in Chap. 2, we tackle in Part I the issue of the detection of the units physically implemented in superscalar CPUs. While Chap. 4 offers a CPU-agnostic algorithm that reverse-engineer superscalar CPUs to infer a throughput-based performance model, Chap. 5 offers formal guarantees on its convergence and links the abstract resource representation used with the traditional hardware-based model. Part II follows the complementary approach: designing an efficient accelerator given the characteristics of one or more applications to be executed on the dedicated logic. Chap. 6 presents a complete overview of the ecosystem used to generate designs: the High-Level Synthesis framework, which is then used in Chap. 7 to build resource-efficient accelerators. However, due to scalability issues, we propose in Chap. 8 another approach with a semi-generic accelerator sharing its compute units to be able to accelerate a panel of user-defined applications while staying competitive in terms of performance-per-area with dedicated solutions. Finally, Chap. 9 concludes and presents possible research directions that emerged from the works presented throughout this manuscript.

Chapter 2
Background

Though the topic of hardware architectures covers a variety of research fields, ranging from high-performance computing and power-hungry accelerators to embedded systems and design focused on energy efficiency, the core components of complex designs remains similar. Memory, ALU, compute units: though offering radically different area / performance / power consumption trade-offs, all silicon-based designs rely on the same families of building blocks that are generically gathered under the notion of *resource*. Both specialized hardware such as FPGAs and ASICs and general-purpose CPUs have clocked submodules responsible of specific tasks, on which the programmer aims to balance the load in order to maximize the global usage of the chip.

This chapter covers the basic notions common to nearly any hardware accelerator that this manuscript will use as building blocks in the following chapters: resource, pipelining, scheduling, as well as compute-specific components of modern CPUs.

2.1 Abstract Resource Model

On both CPU and FPGA, the notion of resource is at the heart of optimization technics. The holy grail of optimization is the complete usage of a (generic or specialized) compute accelerator, combined with the justification that each task cannot be expressed in a simpler manner. In that regard, a resource is either a bottleneck (when the operation it performs is not executed fast enough to "feed" the other resources), or an opportunity to increase the processing speed (when it is idle). In this section, we cover the fundamental characteristics of a resource as used in the next chapter: either as an abstract resource of a CPU performance model in Chap. 3 and 4, or as a compute unit to be synthesized and shared in Chap. 7, and 8.

2.1.1 Resources

A resource is defined as a clocked component whose task is necessary for the correct execution of a subpart of a program, referred as *job* in this section, typically one instruction when dealing with CPUs. A resource is characterized by its *latency*, its *throughput* and its *critical path*:

- The *latency* is the amount of time elapsed between the start of the resource's job and its completion.
- The *throughput* is the maximum number of jobs that can be executed per amount of time by the resource. Often, this metric does not vary with time, which is why the throughput is sometimes defined as the average number of jobs executed per unit of time on a steady state when treating an infinite number of jobs.
- The *critical path* or *CP* represents the time taken by some data to traverse the longest path between two clock cycles. Due to physical constraints, if CP_{res} is the critical path of a resource res, then the maximum frequency of a design containing a resource cannot be greater that $\frac{1}{CP_{res}}$. While CPs are fixed on a CPU, as its design cannot be dynamically changed (as well as its maximum operating frequency); techniques exist on FPGAs and ASICs to minimize the critical path and increase the final design frequency [3, 4, 5, 6].

  In the case of resources computing or transforming data, the throughput corresponds to its *processing power*, while the throughput of resources storing data is called *bandwidth*.

Furthermore, as jobs consume resources, we introduce the following notations:

- $\rho_{j,r}$ is the amount of usage by the job j of the resource r, For example, a single addition will use half of a resource corresponding to a 2-operation wide ALU.
- $load(r)$ or $r.load$ is the load of the resource, equals to $\sum_j \rho_{j,r}$: the total combined usage of the resource r. By definition of the abstract resource model, a program cannot be faster than the load it puts on any resource.

Target	Resource	Role
CPU	Front-end	Decompose instructions into machine commands
CPU	ALU	Compute arithmetic and logical operations
CPU	Load/Store Units	Manage access to the memory subsystem
CPU	Cache	Store recently used values to speed up accesses
CPU	Scheduler	Dispatch operations on available execution units
FPGA	Compute Unit	Execution of a set of operations
FPGA	BRAM	Elementary on-chip memory storage location
FPGA	Off-chip bus	Access off-chip data

Table 2.1: Example of resource for different hardware targets

In the next chapters, resources are treated as an abstraction in order to decompose the behavior of compute accelerators, either for performance prediction or bottleneck analysis.

Therefore, different sorts of hardware units are represented as abstract resources, as described in Tbl. 2.1, ranging from traditional compute units whose performance is measured in FLOPS to front-end or scheduler which are mandatory for the correct execution of an instruction, but may become bottlenecks when too many of them need to be treated at the same time. Due to the versatile nature of FPGAs, resources are limited to elementary components that are part of any designs: Compute Units, on-chip memory access units and off-chip memory access mechanisms. Contrary to CPUs', the performance of FPGA resources is variable depending on the designer's needs, which explains the difference in metrics used to express performances in both worlds. For CPU, the *Instruction per Cycle* rate, defined as $\frac{\#\{instructionsexecuted\}}{time}$ is a measurement of the *occupancy* of the (fixed) hardware units, with the idea that maximizing the global resource usage leads to optimality, while FPGA express performance as a throughput or latency versus area compromise.

2.1.2 Pipelining and Resource Sharing

To increase the performance of a resource, the simplest solution relies in its mere replication [6, 7, 8]. However, that may lead to the unnecessary implementations of not performance-critical components. To limit the increase of area linked with duplication, other approaches have been developed based on the resource sharing, the most common of which being *pipelining*.

Pipelining

Pipelining refers to the decomposition of a resource into smaller blocks on the time dimension [7]. Followingly, all blocks may be used in parallel, increasing throughput without the need of new hardware units, as illustrated in Fig. 2.1. However, due to the need of synchronization of the blocks with the hardware clock, a pipelined design has often a latency higher than a non-pipelined one, thought this effect can be mitigated

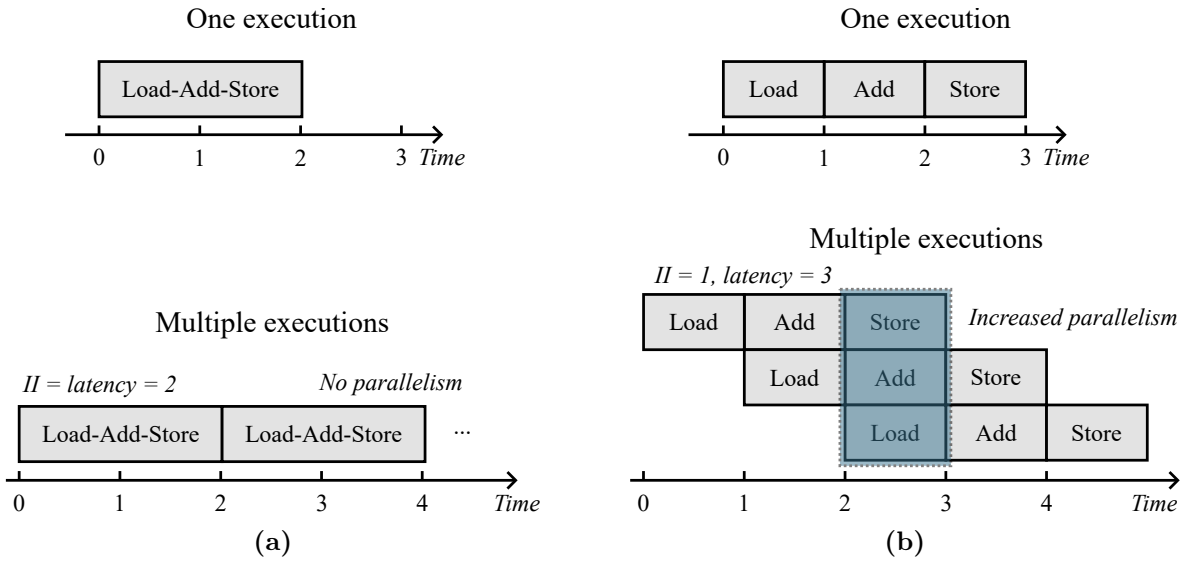


Figure 2.1: Non-pipelined (a) versus pipelined (b) designs

with a higher operating frequency. Indeed, as the data paths inside the resource are split into several submodules, so is the longest path; hence a generally lower CP on pipelined designs, translating into higher a operating frequency.

We define the *initiation interval* (or II) of a pipeline resource the amount of time (expressed as clock cycles) elapsed between the beginning of the treatment of two successive jobs. When the latency of the pipeline is fixed, then $II = \frac{1}{throughput}$.

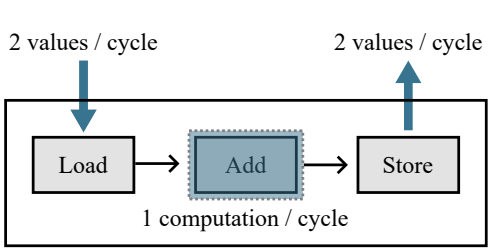
Other Resource Sharing Techniques

Traditional non-pipelining resource sharing techniques [8, 9] consist in replicating low-area / high-usage part of the design while leaving in common other components untouched. This increases overall occupancy of the available units by reusing non-performance critical units (thus increasing overall latency), hence leading to better performance per area.

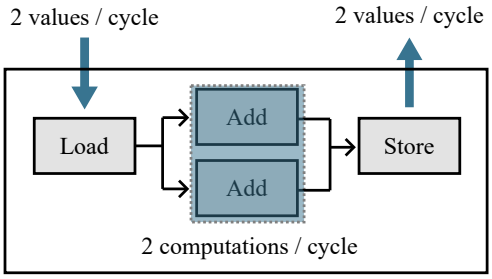
As an illustration, in the former example, duplicating the "add" stage allows to compute more tasks without replication of the load / store units in cases where the available bandwidth is already sufficient, as shown in Fig. 2.2. General-purpose chips such as CPU and GPU uses a mixture of pipelined units such as ALUs or memory access units, and shared components such as caches or instructions schedulers in order to offer the best performance-per-area given a fixed silicon budget.

On FPGAs, modification of clock domains can also be used to increase the processing speed of compute units and share them between different data paths of the design [10].

Full Pipelining A resource is said to be *fully pipelined* when its II is equal to 1, which means that the resource is able to function on new data every cycle.



(a) No resource-shared design, resulting in 1 computation per cycle

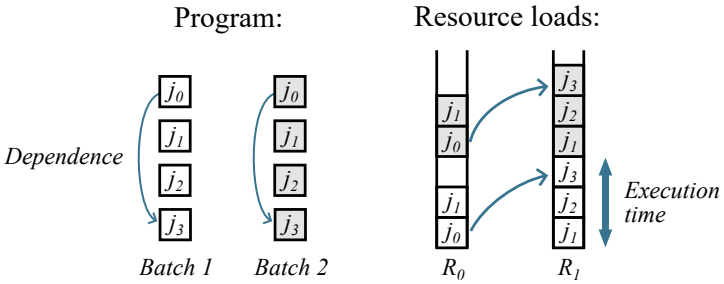


(b) Resource-shared design, resulting in 2 computations per cycle

Figure 2.2: Resource shared (a) and non resource shared (b) architectures

Job	Resource used
j_0	R_0
j_1	R_0 and R_1
j_2	R_1
j_3	R_1

(a)



(b)

Figure 2.3: Abstract resource model: resource usage (a) and corresponding loads/schedules

2.1.3 Scheduling

The *schedule* of a program $\mathcal{P}$ is a function σ returning for each of its elementary job $(j_i)_{i\in[0,n-1]}$ its the timestamp of execution, while the *placement* refers to the affectation of a resource for each couple of jobs and timestamp. A placement is *legal* when no resource is used more than its maximum throughput at any cycle, and a schedule is *legal* when there exists a legal placement respecting it, and it respect job dependencies, i.e. if there is a dependence between j_1 and j_2, then $\sigma(j_1) + lat(j_1) \leq j_2$ where lat is the latency of each job.

Scheduling can be *offline* if the full list of jobs is known before the scheduling decision, or *online* when the scheduler decision is taken before the arrival of new jobs.

Classical offline scheduling policies include:

- *ASAP* or *As Soon As Possible*: executes the jobs in the order they came in, affecting to each job to the first available compatible unit.
- *ALAP* or *As Late As Possible*: recursively builds a schedule from the finishing task by assigning to each jobs the latest timestamp without breaking dependencies.

Scheduling using the abstract resource model From the abstract resource model, we can deduce a schedule for each sub-task, which correspond to one of the ideal schedules (i.e. a schedule resulting in the highest throughput) for batched computation of independent executions.

```
1 Function Execution_ Time(P)
2 |  for j ∈ P do
3 |  |  for r ∈ R do
4 |  |  |  load[r] += ρ_{j,r};
5 |  |  end
6 |  end
7 |  return max(load)
8 end
```

Algorithm 1: Naive scheduling using abstract resources

Lemma 2.1.1. *Let us consider a program $\mathcal{P}$ composed of n jobs $j_0, ..., j_{n-1}$, each using a set of resource $\rho(j) \subseteq \mathcal{R}$. Then, the naive algorithm detailed in Alg. 1, illustrated on Fig. 2.3 outputs the optimal average throughput of independent batched executions of $\mathcal{P}$.*

Proof. By contradiction.

$\Rightarrow$ Let σ be a schedule whose execution time τ is strictly lower than the output of Alg. 1. Consider the resource $r \in \mathcal{R}$ of maximum load under Alg. 1 and τ' its associated execution time. By definition of the abstract resource mapping, $\sum_{j \in \mathcal{P}} \sigma_j \rho_{j,r} \leq \tau$, but as $\sum_{j \in \mathcal{P}} \sigma_j \rho_{j,r} = \tau'$, $\tau \leq \tau'$ which contradicts our hypothesis.

$\Leftarrow$ Let τ be the maximum load over all resources after naive scheduling. Let us σ_a be the ASAP schedule under resource constraints ignoring dependencies. Trivially, its execution time is τ as only resources constraints the execution. Let us define σ recursively such that:

- $\sigma(j_0) = \sigma_a(j_0)$ $(= 0)$
- If there exists $(j, j') \in \mathcal{P}^2$ such that j depends from j' (we note $j' \rightarrow j$) and $\sigma(j') + lat(j') > \sigma_a(j)$ (with $lat(j')$ a constant depending on the resource usage of j'), then we define $\sigma(j) = k \cdot \tau \cdot \sigma_a(j)$ with $k \in \mathbb{N}$ such that the equality is verified.
- Pipelined execution starts with an II of τ.

By construction, σ is legal: i) it respects dependence and ii) no resource is used twice at the same timestamp, because the schedule modulo τ is the ASAP schedule, thus respecting placement.

Now, let us take a steady pipelined execution of $\mathcal{P}$ using σ. As the execution is in a steady state, the resource usage is the same as the ASAP schedule, with the exception that some jobs are shifted by k. Therefore, the jobs executed during a time window of size τ are exactly the ones composing one execution of $\mathcal{P}$, so the average throughput is $1/\tau$, which demonstrate the existence of a schedule reaching the throughput predicted by Alg. 1.

□

Assuming that the abstract resources cover a subset of the actual resources of the modeled chip, then the execution time output by the naive scheduling is a *lower bound* on the actual execution time, as the reserve part of the proof becomes false. However, in the case when the execution time matches the actual execution time, then the abstract resource model also pinpoints a possible bottleneck: the resource with the highest load.

Note that this schedule relies on the fact that amount of usage of each resources by the chip and in the abstract resource model is the same. When this is not the case, such as on superscalar CPUs, where an instruction *may* use some resource but not others, the equivalence is less straightforward. Therefore, a complete proof of equivalence between the abstract resource model and the usual CPU port model where resource usage is defined with disjunctions ("or") instead of conjunctions ("and") is detailed in Chap. 5.1.

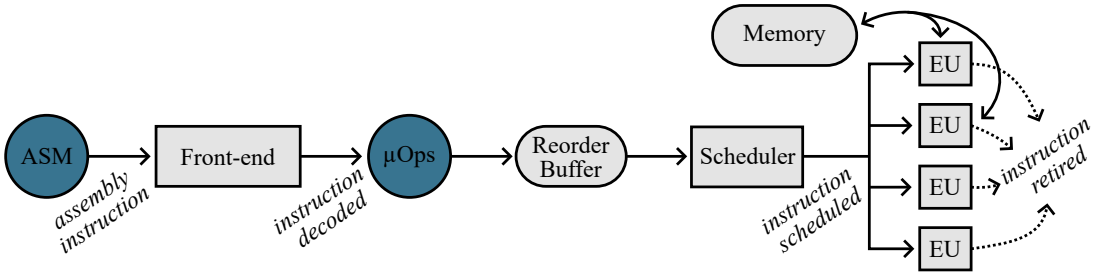


Figure 2.4: Schematic view of a CPU and life cycle of an instruction

2.2 General CPU Architecture

Even though the general principles of CPU design have not changed since the first 8086 – executing a stream of instruction as fast as possible – several implementation techniques have been developed to decrease overall execution time of programs, especially on high-performance chips for which area and power consumption are less restricted: cache, buffer, out-of-order execution, pipelining, prefetching, ... In this section, we will focus on the resources used in the steady state execution of low memory footprint kernels, i.e. ignoring the memory subsystem.

An instruction's path in the CPU can be decomposed into three phases, illustrated in Fig. 2.4: it is first *decoded* by the front-end and decomposed into simpler, machine-specific μOPs. An instruction coming out of the decoder is also called *issued*. Then, it is *scheduled* to an execution unit, before being *retired*, that is, evicted from the execution pipeline.

2.2.1 Out-of-order Execution

Modern high-performance CPU architecture are *out-of-order*, which means that the internal order of execution of the instruction may be different from the order in which instructions are submitted. This allows the extraction of *instruction-level parallelism*: depending on the dependence pattern, some instructions may be executed in parallel, even though they are not present in a consecutive order in the program.

As a consequence, out-of-order CPUs must integrate supplementary components compared to *in-order* ones:

- The *Reorder Buffer*, or ROB, is a buffer containing instructions that have been decoded and that can potentially be executed.
- The *Scheduler* is a component that selects instructions whose dependencies are satisfied from the ROB and assigns them to an *execution port*.
- *Execution Ports* are a hardware unit controlling several *execution units*, each capable of one arithmetic/logic or memory operation such as loading, storing, branching or computing. Thus, a port is usually capable of performing a set of different operations, usually in a fully pipelined fashion.

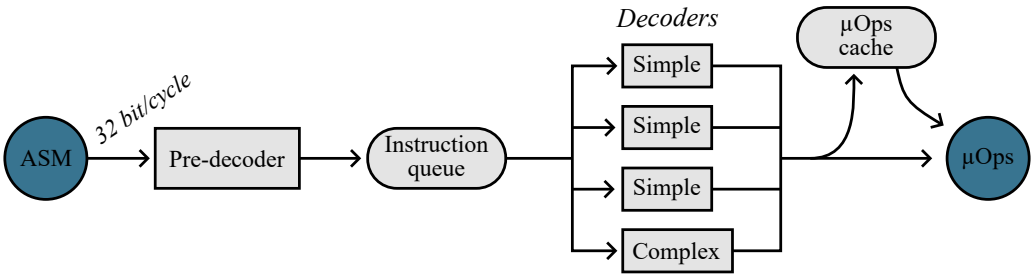


Figure 2.5: Schematic view of Intel's Skylake front-end

Usually, the peak performance of a superscalar pipeline is express as its *Instructions Per Cycle* (IPC) rate, corresponding to the maximum number of instructions that may be executed per clock cycle in a steady state. It is often limited by the size of internal buses and not the number of ports: for example, the Intel Skylake architecture has 8 ports, but can only retire a maximum of 4 instructions per cycle.

As an example, AMD's Zen3 architecture [11] integrates a ROB that stores up to 256 entries. A maximum of 6 instructions can be fetched by the schedulers, which dispatch them to ports amongst the 14 integrated ones (8 for the integer / logical part, and 6 for the floating point subsystem).

Note that even historically low-power chips such as Arm Cortex series have started to converge to port-based architectures for their top-of-the line designs. For example, the Cortex-A72, integrated in the Raspberry Pi 4, integrates an out-of-order pipeline with a maximum throughput of 3 instructions per cycle [12].

2.2.2 Front-end

Decoding Stage

The decoding stage is the step that fetches data from global memory and extract the instructions in the first *predecode* step. Then, the instructions are translated into smaller, atomic, micro-instructions (also referred as μOPs), understandable by the CPU's execution units. The complexity of the front-end depends on the target ISA and the performance objective of the processor: while simple microcontrollers operating on fixed-size ISA have straightforward front-end implementations, modern instruction sets such as Armv8a [13] or x86 [14] requires intricate logic.

For example, as illustrated in Fig. 2.5, Intel's Skylake front-end is able to pre-decode only aligned blocks of 32 bits [15], and decode at most five instructions per cycle. The translation is then done by 4 decoders: 3 "simple" that are only able to output one μOP, and 1 "complex" that can also decode multi-μOPs instructions. Note that the complex decoder always decodes the first instructions in given by the predecoder, leading to potential stalls on complex assembly code.

To avoid multiple decoding of the same instructions, CPU designers have implemented a micro-instruction cache called μOP-cache or DSB (Decoded Stream Buffer), subject to multiple restrictions [16].

Branch Prediction

One of the major sources of potential stalls in modern microprocessors is conditional branch instruction. Indeed, in this case, the next code section is unknown and a simple CPU must wait for it to be computed before actually starting its execution.

As branches represents in average around 20% of the total instruction mix of programs [17], CPU architects have developed *branch prediction* mechanisms to guess the branch destination before its actual computation, and *speculate* the rest of the execution until validation of the branch target address. If the guess is validated, then instructions are committed at no cost and execution can continue. In the other case, a *misprediction* occurs and speculated instructions' side effects have to be reverted, triggering a performance penalty whose amount typically ranges around 10 to 20 cycles in modern superscalar CPUs. Until 2018, speculative treatment of data was considered safe, but it was shown with the vulnerabilities Meltdown and Spectre [18, 19] that this is not the case.

With CPUs now being able to execute more than 4 instructions in parallel thanks to their out-of-order paradigm, and ROB capacity now reaching 512 entries (in Intel Alder Lake [20]), this means that branch predictors have to correctly predict around one hundred branches in order not to be a bottleneck. Due to this reason, branch prediction remains a hot research topic [21, 22, 23], even though this manuscript concentrates on throughput measurements and ensure perfect operation of the branch predictor in most of its experiments.

2.2.3 Instruction Set Extensions

To improve CPU efficiency on domain-specific computing, *instruction set extensions* have been developed: new instructions that do not interfere with the base ISA, but which triggers custom logic to improve computing speed. Examples include vector extensions of the x86 ISA, SSE and AVX, now a *de facto* standard of the x86_64 instruction set; but also less known variations such as Intel's GFNI [24] to improve Gallois fields computations or TSX for transactional memory support.

ISA extensions may or may not introduce new resources depending on their implementation. For example, AVX instructions are advised by Intel not to be mixed with the older SSE instructions it supersedes, as this causes further latency [25]. On the opposite, VNNI [26], an x86 extension dedicated to machine learning, builds on the top of the already existing AVX-512 extension set, thus requiring no supplementary resources.

Part I

Throughput Optimization for Superscalar Architectures

Chapter 3
Throughput Performance Models for Superscalar Architectures

With CPUs becoming more and more complex, cycle-accurate simulators have stopped being able to accurately reproduce the execution path of programs on high-performance architectures. While RTL simulation (wire-level modeling of all chip components) can theoretically achieve such objective, its prohibitive time overhead limits in practice its usage to small microcontrollers or internal CPU subparts.

However, the need of fast yet precise performance models is real, especially in the context of *performance debugging*, where the objective is to pinpoint the code region responsible for slowdowns and, ideally, offer optimization advice. This chapter covers in Sec. 3.1 various techniques aiming at constructing automatically performance models for superscalar CPUs by explicitly recovering per-instruction port usage. Then, Sec. 3.2 presents the theoretical foundations of our abstract resource decomposition built upon the model presented in Sec. 2.1, as a *conjunctive* resource mapping instead of the usual *disjunctive* one. This way, abstract mapping can supersede former works thanks to an easy expression of microbenchmarks' throughput as well as a flexible representation of existing bottlenecks. Finally, Sec. 3.3 presents the modifications done to the PIPEDREAM library [27] in order to obtain an accurate microbenchmark-based throughput measurement infrastructure.

3.1 State of the Art

The most straightforward idea when building a performance model of a CPU is to stick to *some level* of simulation by predicting the behavior of some hardware units while ignoring or approximating others. As a consequence, these approaches often require out-of-the-box information about CPU's architecture and rely on hardware-specific tools such as performance counters.

3.1.1 Performance Counters Derived Models

Performance counters are particularly useful for microarchitectural performance modeling as they provide an on-chip infrastructure for micro-architectural event measurements. Initially built for chip debuging and testing, performance counters are now usable for performance optimisation in tools such as Perf [28] or PAPI [29]. On Intel CPUs, dedicated counters exist to provide the number of μOPs that used each port in one section of the program, which can be used to characterize the port usage of every instruction.

First attempts to measure the latency and throughput of x86 instructions where led by Agner Fog [16] and Granlund [30] using hand-written microbenchmarks. Each benchmark is composed of the dependence-free repetition of one or several instructions, and performance counters to measure cycles and port usage of each type of instruction as well as their variation. This method proved to be successful for the majority of consumer-available x86 CPU form Intel, AMD and VIA. Fog's mappings are considered by the community to be quite accurate. For example, the machine model of the x86 back-end of the the LLVM compiler framework [31] is partially based on them [32].

However, Fog's and Granlund's approach using microbenchmarks and manual analysis is tedious and error-prone, since modern CPU instruction sets have thousands of different instructions with complicated interactions. Abel and Reineke [33, 34] have tackled this problem by designing an automatic microbenchmark generator detecting instructions that saturate individual ports, and using them to deduce overall resource usage thanks to hardware counters. They recently also started providing data on the newest generations of AMD CPUs, but since those do not have the required hardware counters, Abel and Reineke only publish instruction latencies, throughputs and port usage derived from the documentation.

This work has been reused in uiCA [15], an instruction-level simulator relying on an accurate port mapping, but also making extensive use of microbenchmarks [35, 36] to deduce information about Intel-specific microarchitectural details such as behavior of the predecoder, activation condition of the μOP cache or implementation of the ROB.

Another direction of research was explored by PMEvo [37]: a tool that automatically generates semi-randomly a set of benchmarks that is uses to build a port mapping. Thought PMEvo does not require hardware performance counters as it only relies on runtime measurements of its benchmarks, its goal still lies in the automated generation of an architecture-derived port mapping.

3.1.2 Proprietary and Ad-Hoc Tools

Intel has developed a static analyzer named IACA [25] which uses its internal mapping based on proprietary information. However, the project is closed-source and has been deprecated since April 2019. Even though some latencies are given directly in the documentation [14], they are known to contain errors and approximations, in addition to being incomplete.

OSACA [38], is an open source alternative to IACA that offers a similar static throughput and latency estimator. It relies on automated benchmarks manually linked with publicly available documentation to infer the port mapping and the latencies of the instructions. The tool Kerncraft [39] focuses on hot loop bodies from HPC applications while also modeling caches; its mapping comes from automated benchmarks generated through Likwid [40] and hardware counters measurements. CQA [41], a static loop analyzer integrated into the MAQAO framework [42], takes a similar path while also supporting OpenMP routines. It combines dependency analysis, microbenchmarks, and a port mapping and previous manual results to offer various types of optimization advice to the user, such as vectorisation, or how to avoid port saturation. Both Kerncraft and CQA use a hardcoded port mapping based on Fog's work and official Intel and AMD documentation.

As the LLVM compiler needs internally a performance model in order to evaluate its output quality, it now ships with a derived stand-alone tool, llvm-mca [43], dedicated to assembly-level execution time prediction. Because of the constraints that comes with a compiler infrastructure, llvm-mca relies on a simple CPU model including a decoding stage, a queue and an execution stage based on ad-hoc ports mapping (either taken from other academic work or from vendor documentation).

Besides the classic port mappings, machine learning based approaches have also been used to approximate the throughput of basic blocks with good accuracy. Ithermal [44] uses a deep neural network based on LSTM as a "black box" to predict the execution time of basic blocks, trading understanding of the model for accuracy. Followingly, the resulting model is completely opaque and cannot be analyzed or used for any other purpose than the prediction of basic block throughputs. For example, Ithemal does not output instruction-based feedbacks, i.e. detail reports of the influence of each instruction, which is critical for manual assembly optimization.

However, Ithemal has been reused as a differentiable surrogate in DiffTune [45] for the optimization of llvm-mca [43]. The neural network, enriched with llvm-mca's internal parameters, is trained on simulated measurements through convex optimization; then the trained network is used as a black box predictor to fine-tune the microarchitectural parameters.

3.1.3 Comparison with the Abstract Resource Model

Models derived from the port mapping topology, while able to accurately predict the execution of pipelined instructions bottlenecked only on the execution ports, cannot

represent other bottlenecks like the reorder buffer, or the non-pipelined instructions like division. To remedy these issues, port mapping are used as the backbone of both simulators, but enriched with other components in order to simulate the complete behavior of CPUs. We claim that a simpler approach is possible by the use of abstract resources when only dealing with instructions throughput. We detail further this approach in Chap. 4.

The other techniques relies on black-box models, either because of proprietary information or machine learning. While being competitive from the view of pure accuracy, the fact that they do not output hints about bottleneck nor potential improvements made them impractical for real-world performance optimization. Moreover, CPU documentation and vendor tools are known to contain errors and bugs [15, 16, 30], both because of human factors and deliberate obfuscation to protect designers' intellectual properties. This calls for automated, benchmark-driven performance models in order to keep up with the growing number of CPU microarchitectures available on the High Performance Computing domain.

While the idea of using performance counter may seem well-founded, they reveal to be in practice not reliable enough to build performance models. Indeed, as they were build initially for CPU debugging, they are known to be buggy, ill-documented, and specific to one vendor, if not one architecture [14]. For example, AMD's implementation of performance counters does not match Intel's one, and Arm's ecosystem is even less developed – in the case where they exist. This calls for a performance model build on the simplest form of measurement that is integrated in CPUs: time, and nothing else.

Of all the models presented in Sec. 3.1.1 and Sec. 3.1.2, PMEvo is the one that resemble the work presented in this part the most, as it satisfies both constraint of not being automated and only relying on time measurements. However, one of the most significant flaw of PMEvo's approach lies in its handling of a large set of instructions for the mapping (i.e., all available), which may lead to quickly explode the number of microbenchmarks as they are selected by evolutionary algorithms. Due to the simplicity of the expression of throughput with the abstract resource model (see Sec. 2.1.3), an approach based solely on this model is more likely to scale well with respect to the number of instructions in the ISA.

3.2 Conjunctive and Disjunctive Resource Mapping

To characterize the throughput of each individual instruction, a description of the available resources and the way they are shared is needed. As seen in Sec.3.1, the most natural way to express this sharing is through a port mapping, that is, a tripartite graph that describes how instructions decompose to μOPs and assigns μOPs to execution ports (see Fig. 3.1a). The goal of existing work has been to reverse engineer such a port mapping for different CPU architectures.

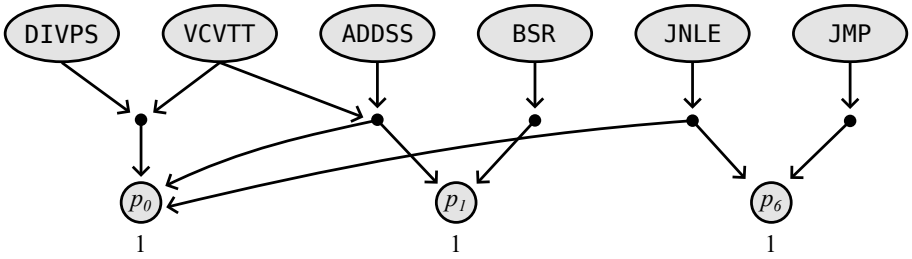
The first level of this mapping (from instructions to μOPs) is conjunctive, i.e., a given instruction decomposes into one or more of *each* of the μOPs it maps to. However, the second level of this mapping, on the other hand, is disjunctive, *i.e.* a μOP can choose to execute on *any* one of the ports it maps to. Even with hardware counters that provide the number of μOPs executed per cycle and the usage of each individual port, creating such a mapping is challenging and requires a lot of manual effort with ad hoc solutions to handle all the cases specific to each architecture [46, 16, 30, 33].

In this section, we present an intuition of the main advantage of the conjunctive abstract resource representation: its ability to predict straightforwardly throughputs equivalent to usual disjunctive mapping, with a simple correspondence in the case of port mapping representations.

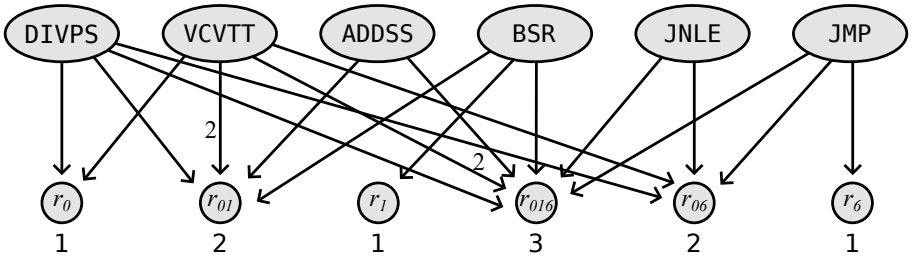
3.2.1 Tripartite and Bipartite Port Mapping

Such hardware-derived approaches, while powerful and allowing a semi-automatic characterization of basic-block throughput, suffer from several limitations. First, they often assume that the architecture provides the required hardware counters. Second, they only allow modeling the throughput bottlenecks associated with port usage, and neglect other resources, such as the front-end or reorder buffer. Thus, it provides a performance model of an ideal architecture that does not necessarily fully match reality.

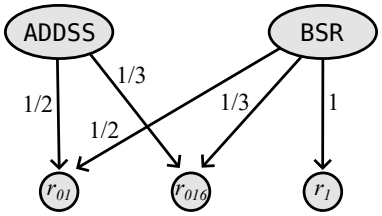
To overcome these limitations, we restrict ourselves to only using cycle measurements when building our performance model. Not relying on specialized hardware performance counters may complicate the initial model construction, but in exchange our approach is able to model resources not covered by hardware counters with relative ease. This also paves the way to significantly ease the development of modeling techniques for new CPU architectures. One of the main challenges is to generate a set of micro-benchmarks that allows the detection of all the possible resource sharing. Unfortunately, to be exhaustive, and in the absence of structural properties, this set is combinatorial: all possible mixes of instructions need to be evaluated. A simple way to reduce the set of micro-benchmarks required is to reduce the set of modeled instructions to those that are emitted by compilers [44, 37]. Another natural strategy followed by Ithemal [44] is to build micro-benchmarks from the "most executed" basic-blocks of some representative benchmarks. A third strategy, used by PMEvo [37], is to restrict micro-benchmarks to contain repetitions of only two different instructions.



(a) Port mapping (disjunctive form) and maximum throughput of each port.



(b) Abstract resource mapping (conjunctive form) and maximum throughput of each resource.



(c) Normalized conjunctive form for `ADDSS` and `BSR`.

Figure 3.1: Mappings computed for a few SKL-SP instructions.

The second main challenge addressed by PMEvo is to build an interpretable model, that is, a resource-mapping that can be used by a compiler or a performance debugging tool, instead of a black-box only able to predict the throughput of a microkernel. One issue with the standard port-mapping approach, as used in [33, 38, 43], is that computing the throughput of a set of instructions requires the resolution of a flow problem; that is, given a set of micro-benchmarks, finding a mapping of μOPs to ports that best expresses the corresponding observed performances requires solving a multi-resolution linear optimization problem. This linear problem also does not scale to larger sets of benchmarks, even when restricting the micro-benchmarks to only contain up to two different instructions. PMEvo addressed this issue by using a evolutionary algorithm that approximates the result.

3.2.2 Dual Representation of a Disjunctive Mapping

Our approach is based on a crucial observation: a dual representation exists for which computing the throughput is not a linear problem, but a simple formula instead. While

p_0	p_1
ADDSS	BSR
ADDSS	BSR
ADDSS	ADDSS

(a) ADDSS^2 BSR

p_0	p_1
ADDSS	BSR
$\emptyset$	BSR

(b) ADDSS BSR^2

Figure 3.2: Disjunctive port assignment examples

it takes several hours to solve the original disjunctive-port-mapping formulation, only a few minutes suffice for the corresponding conjunctive-resource-mapping formulation.

For the sake of illustration only let us consider the Skylake instructions restricted to those that only use ports 0, 1, or 6 (denoted as p_0, p_1, and p_6). Fig. 3.1a shows the port mapping for six such instructions. In this example: the μOP of BSR has a single port p_1 on which it can be issued; as for instruction ADDSS, its μOP can be issued on either p_0 or p_1. Hence, BSR has a throughput of one, that is, only one instruction can be issued per cycle, whereas ADDSS has a throughput of two: two different instances of ADDSS may be executed in parallel by p_0 and p_1. The throughput of the multiset $K = \{\text{ADDSS}^2, \text{BSR}\}$, more compactly denoted by ADDSS^{2}BSR, is therefore determined by the combined throughput of resources p_0 and p_1. Indeed, in a steady state mode, the execution can saturate both resources by repeating the pattern represented in Fig 3.2a. In this case, there clearly does not exist any better scheduling, and the corresponding execution time for K is 3 cycles for every 6 instructions, that is, an Instruction Per Cycle (IPC) of 2. Now, if we consider the set ADDSS BSR^2, its throughput is limited by p_1. Indeed, the optimal schedule in that case would repeat the pattern represented in Fig 3.2b, which requires 2 cycles for 3 instructions, that is, an IPC of 1.5. More generally, the maximum throughput of a multiset on a tripartite port-mapping can be solved by expressing the minimal scheduling problem as a flow problem.

The *dual representation*, advocated in this section, corresponds to a *conjunctive* bipartite resource mapping as illustrated in Fig. 3.1b. In this dual mapping, an instruction such as ADDSS which uses one out of two possible ports p_0 and p_1 only uses the abstract resource r_{01} representing the combined load on both ports, and uses neither r_0 nor r_1. Therefore, the maximum throughput of r_{01} is the sum of the throughput of p_0 and p_1, that is, 2 uses per cycle. Instructions that may only be computed on p_0 then uses r_0 *and* r_{01}, along with all other resources combining the use of p_0 with other ports such as r_{06} and r_{016}. It follows that the average execution time of a microkernel is computed as the maximum load over all abstract resources, that is, their number of uses divided by their throughput (as demonstrated in Sec. 2.1.3). We prove in Sec. 5.1.2 the strict equivalence between the two representations *without the need for any combinatorial explosion in the number of combined resources*. Because of this property, the trade-off offered by the conjunctive formulation (more resources for a simpler throughput computation) offers better overhaul solving complexity that former disjunctive-based approaches for real processors, hence the better scalability of PALMED. Indeed, in practice, some combined resources

are not needed (e.g. r_{16} in our example) as their usage is already perfectly described by the usage of individual resources (here, r_1 and r_6).

A key contribution of this chapter is to provide a less intricate two-level view, that can be constructed quicker than previous works. Instead of representing the execution flow as the traditional three-level "instructions decomposed as micro-operations (micro-ops) executed by ports" model, we opt for a direct "instructions use abstract resources" model. Whereas an instruction is transformed into several micro-ops which in turn *may* be executed by different compute units; our bipartite model *strictly uses* every resource mapped to the instructions. In other words, the *or* in the mapping graph are replaced with *and*, which greatly simplifies throughput estimation. This representation may also represent other bottlenecks such as the instruction decoder or the reorder buffer as other abstract resources. Note that this corresponds to the user view, where the micro-ops and their execution paths are kept hidden inside the processor.

3.3 Code Generation for Accurate Throughput Measurements

All performance numbers presented in the next chapters results from the usage of the PIPEDREAM [27] benchmarking library realized by Fabian Grüber. Initially, its goal was to generate assembly microbenchmark (dependence-free sequence of assembly instructions) to be characterized using a set of performance counters, and eventually to reverse engineer port mappings. We modified it to even facilitate its usage for throughput measurement, with the following changes:

- Support of the Armv8a instruction set.
- Support of only one performance counter: the one measuring CPU cycles, widely available on any high-performance architecture, either using PAPI [29] or Linux' Perf [28].
- Limitation of the effects of the pre-decoder bottleneck on recent Intel microarchitectures (Sec. 3.3.1).
- Limitation of the dependences between different instructions (Sec. 3.3.2).

Though still containing limitations, discussed in Sec. 3.3.3, the modified PIPEDREAM benchmarking library has proven to be mature enough to be the back-end of our reverse-engineering framework detailed in Chap. 4.

3.3.1 Padding Microkernels of Multiple Instructions

The predecoder is the hardware component in charge of extracting instructions from the program's binary once mapped into memory. On recent Intel microarchitectures [15], this pre-decode stage is operating on *aligned* chunks of 16 bytes, with a maximum rate of 5 instructions per cycles. When more than 5 instructions are present in the chunk, only the first 5 are decoded (in one cycle), and the next (up to 5) instructions *of the same block* are then decoded. This may create bottlenecks on codes where 6 instructions are present on a 16-byte segment, such as `XOR_R32_R32`. This instruction performs an bitwise exclusive or of two registers, storing the result in on of the operands, often used to zero out registers. In assembly, it requires 3 bytes, leading to a theoretical maximum of 5.33 instructions per 16-byte boundaries, thus saturating the pre-decoding stage. However, this will result in actual performance degradation, as this instruction translates to a single μOP: due to the retired queue bottleneck, its final IPC is reduced to 4.

To avoid biased measurements by PIPEDREAM, we pad our micro-kernels with `NO-OP`: instructions of variable size without any port usage, in order to maintain a 16-byte alignment of the benchmarked instruction. However, these `NO-OP` still take room at the predecoder level: therefore, its systematic usage results in worse performances than the non-padded version.

Therefore, we only pad benchmarks consisting in the repetition of a "large" number of instructions, for example 10 times `ADDPD` and 11 times `LEA`, for which the alignment of

the kernel is crucial for performance. Indeed, the total number of instructions generated by PIPEDREAM in its main execution loop is small to avoid hitting the instruction cache bandwidth (in practice, $\simeq$200 instructions), which induce a small number of effective repetitions of the measured kernels; hence worsening the effect of misalignment on the performance.

Experimentally, we added padding to benchmark that consists in the repetition of more than 20 instructions in order to align each repetition of the kernel in order to mitigate this effect.

3.3.2 Generation of Basic Blocks with no Dependencies

We found that the default register allocator of PIPEDREAM was not well-suited for throughput measurements when instructions of different throughput were used together.

By default, the `Minimize_Deps_Register_Allocator` used for throughput measurements has a straightforward approach: one register is used for all register read operands, one is used if needed as the base address for memory loads / stores, and the other available registers are used in a round-robin fashion in order to maximize *assembly-level* reuse distance. However, when two different instructions are benchmarked together with different resource usage profile, situations may happen where this behavior creates dependencies between each instruction type, as illustrated in Fig. 3.3. In this case, the measured IPC does not reflect only back-end resource usage of the instructions, but is polluted by dependencies.

We mitigate this effect with the `Minimize_Deps_Pooled_Register_Allocator`: for each instruction type and each written operand, a pool of registers is attributed, with size proportional to its relative number in the measured microkernel (with a minimum of 1). A single register is still used for operands that are only read, but registers used as destinations for writes are taken *from the instruction's pool*, thus removing any dependence between instructions of different types. The idea here is to be resilient to any value of the instructions' IPC: no write-after-write dependency syntactically exists between the different instructions. Between instructions of the same type, write-after-write dependencies do not result in any constraints, as all instructions are executed at the same speed: when the size of the pool exceeds the number of register written per cycle, there is no reason for the CPU to stall to enforce coherency of the register file[1]. This is especially useful in cases where read and writes happens to memory, as one register is also reserved for the base memory address[2], decreasing the number of register available for the destinations.

Fig 3.3 shows the effect of the allocation on a benchmark of the form $S^{4\overline{S}}I^{\overline{I}}$, used in PALMED to detect the resource used by I (here `ROL`). While no WaW dependencies exists

[1]While modern high-performance CPUs integrate register renaming in order to mitigate stall induces by dependencies such as WaW, this is not the case for all processors, especially on power-efficient ones where the number of available registers is reduced to save area.

[2]The former implementation used two registers: one for memory reads, the other for memory writes, whereas our improved version relies on integer offsets to access different memory cells when possible.

```
# ----- unroll  0
  orq           (%rsi), %rdx
  orq           64(%rsi), %rcx
  orq           128(%rsi),
    %rbx
  orq           192(%rsi),
    %rax
  rol           $1, %r15
# ----- unroll  1
  orq           256(%rsi),
    %r14
  orq           320(%rsi),
    %r13
  orq           384(%rsi),
    %r12
  orq           448(%rsi),
    %r11
  rol           $1, %r10
# ----- unroll  2
  orq           512(%rsi), %r9
  orq           576(%rsi), %r8
  orq           640(%rsi),
    %rdx
  orq           704(%rsi),
    %rcx
  rol           $1, %rbx
# ----- unroll  3
  orq           768(%rsi),
    %rax
  orq           832(%rsi),
    %r15
  orq           896(%rsi),
    %r14
  orq           960(%rsi),
    %r13
  rol           $1, %r12
# ----- unroll  4
  orq           1024(%rsi),
    %r11
  orq           1088(%rsi),
    %r10
  orq           1152(%rsi),
    %r9
  orq           1216(%rsi),
    %r8
  rol           $1, %rdx
```

(a)

```
# ----- unroll  0
  orq           (%rsi), %rdx
  orq           64(%rsi), %rcx
  orq           128(%rsi),
    %rax
  orq           192(%rsi),
    %r10
  rol           $1, %rbx
# ----- unroll  1
  orq           256(%rsi), %r9
  orq           320(%rsi), %r8
  orq           384(%rsi),
    %rdx
  orq           448(%rsi),
    %rcx
  rol           $1, %r15
# ----- unroll  2
  orq           512(%rsi),
    %rax
  orq           576(%rsi),
    %r10
  orq           640(%rsi), %r9
  orq           704(%rsi), %r8
  rol           $1, %r14
# ----- unroll  3
  orq           768(%rsi),
    %rdx
  orq           832(%rsi),
    %rcx
  orq           896(%rsi),
    %rax
  orq           960(%rsi),
    %r10
  rol           $1, %r13
# ----- unroll  4
  orq           1024(%rsi),
    %r9
  orq           1088(%rsi),
    %r8
  orq           1152(%rsi),
    %rdx
  orq           1216(%rsi),
    %rcx
  rol           $1, %r12
```

(b)

Figure 3.3: Example of assembly kernels and resulting dependencies produced by the old (a) and the new (b) allocator for four `ROL` (quadword or register/memory) and one `ORQ` (register rotation left of a constant factor 1)

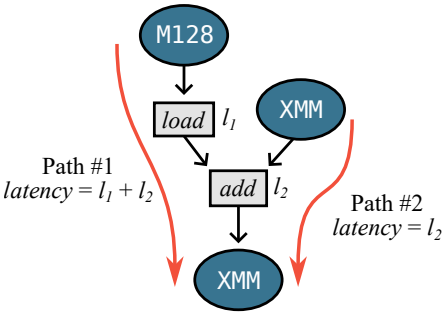


Figure 3.4: μOPs generated for SSE vector add (VADDPD) with one operand from memory (M128) and two register operands (XMM) and associated latency

between `ROL` and `ORQ` in Fig. 3.3b (i.e., register writes are guaranteed to happen in-order), the register `rbx` is used in Fig. 3.3a on line 4 by `ORQ` and line 18 by `ROL`. Therefore, cases where the latency of `ORQ` latency is higher than the number of cycles needed to start the execution of the 11 in-between instructions will trigger a WaW dependency, therefore triggering additional mechanisms inside the CPU, potentially adding noise to our measurement.

3.3.3 Limitations of the Code Generator

Complex Addressing Patterns Due to software architecture decision, some variation of instructions are not supported, such as:

- x86 Instructions using a register as scale operand of a memory location, e.g. `ADD rax, [rbx + 4*rcx]`. As they are used in practice by compilers to access attributes in arrays of struct or multi-dimensional arrays, this problem needs to be solved for real-world usage of PIPEDREAM.
- Instructions using the instruction pointer as operand.
- Instruction with side-effect on the control flow, i.e. branches.

Latency measurements As PIPEDREAM is build as throughput benchmark generation library, the notion of latency is not easily measurable as it definition is not as clear as it may seem. Indeed, while latency is often considered as an instruction-bound constant, this is not true in practice. Depending on the number of μOPs an instruction generates and the order of availability of its source operands, an instruction can have different latency profile, as described in [33]. Indeed, the latency of an instruction is defined as the minimum number of cycles between the availability of all its source operand and the availability of its results. In the case of an instruction being decomposed into several micro-ops, as illustrated in Fig. 3.4, the latency can vary between being the sequence of two μOPs (typically a load and a computation), or only one μOP (the computation) if the μOP fetching value from memory is not on the critical path of the program, depending on the sequence of μOPs considered.

As of now, PIPEDREAM is not able to generate microbenchmarks maximizing operand-specific critical path, and can only maximize register-to-register or memory-to-memory dependencies by reusing the same register / memory location in all instructions. This also means that register-to-memory and memory-to-register dependencies cannot be taken into account at all, due to the lack of automated detection of low resource usage instruction converting register to memory.

Chapter 4
PALMED: Efficient Automated Characterisation of Throughput in Superscalar Architetures

In this chapter, we present PALMED: a CPU back-end characterisation tool, built upon the theoretical foundations described in Chap. 3 and the abstract model defined in Sec. 2.1. Its goal is to propose a simple, microbenchmark-driven CPU resource model, as described in Tbl. 4.1.

This chapter starts with a complete overview of the PALMED framework in Sec. 4.1, then details each of the major steps of the algorithm: first in Sec. 4.2, how we narrow the ISA to select a few representative instructions. Then, two flavors of PALMED are presented: an exact formulation, which does not converge sufficiently fast in practice (Sec. 4.3), and a fast but inexact one (Sec. 4.4). Finally, both versions are evaluated in Sec. 4.6 on a simulated, ideal machine and on real ones.

Table 4.1: Summary of key features of PALMED vs. related work

	no HW counters	no manual expertise	interpretable	general
llvm-mca [43]	✗	✗	✓	✓
Ithemal [44]	✓	✗	✗	✗
IACA [25]	N/A	✗	✓	✓
uop.info [33]	✗	✗	✓	✓
PMEvo [37]	✓	✓	✓	✗
PALMED	✓	✓	✓	✓

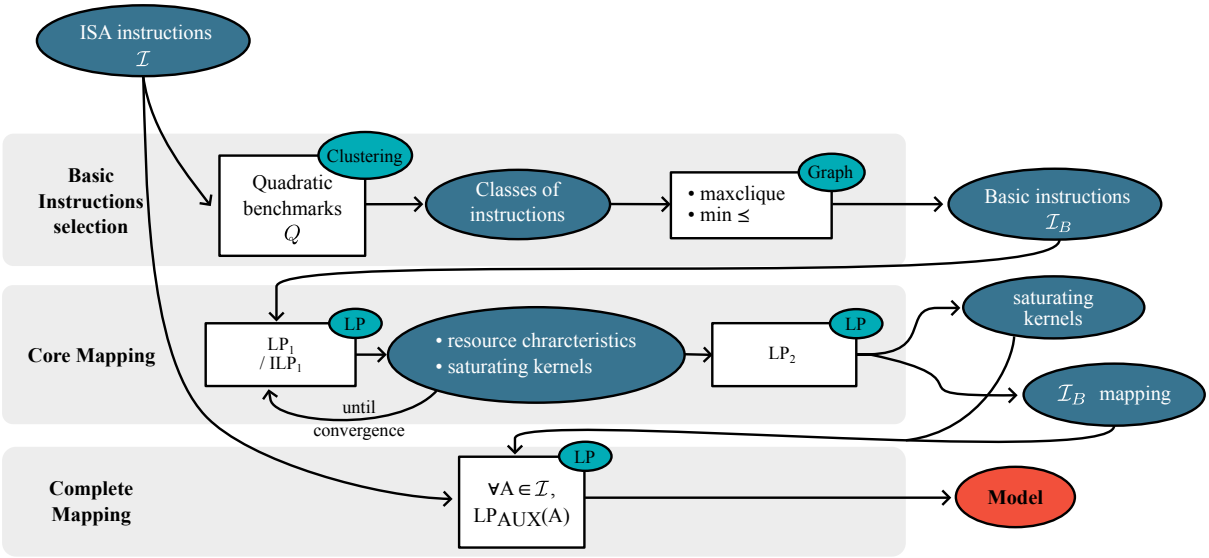


Figure 4.1: High-level view of the algorithms of PALMED

4.1 Complete Flow of PALMED

Fig. 4.1 overviews the major steps of PALMED, which are extensively described in the following sections. Our algorithm follows an approach similar to the one developed by uops.info: its principle is to first find a set of *basic instructions* producing only one μOP and bound to one port.

This first step can be done on Intel CPUs by measuring the μOP per cycle on each port for each instruction through performance counters. This approach will not be tackled here, as it contradicts our goal of non-dependence toward CPU vendors.

Those basic instructions are then used to characterize the port mapping of any general instruction by artificially saturating one-by-one each individual port and measuring the effect on the usage of the other ports. The challenge addressed by PALMED is to find a mapping, even for architectures that do not have such hardware counters.

This translates in two major hardships: firstly, in our case, there is no predefined resources; secondly, there even is no simple technique to find the number of μOPs an instruction decomposes into. As illustrated by Fig. 4.1 the algorithm of PALMED is composed of three steps: 1. Find basic instructions; 2. Characterize a set of abstract resources (expressed as a *core mapping*) and an associated set of saturating microkernels (a single instruction might not be enough to saturate a resource); 3. Compute the resource usage of each other instruction with respect to the core mapping.

As an example, let us consider x86 instructions using only p_0, p_1, or p_6 on Intel's Skylake microarchitecture; there exists 754 of such benchmarkable instructions. Quadratic benchmarking – that is, measuring the execution time of one benchmark per pair of instruction, leading to a quadratic number of measures (567762) – allows us to regroup those of similar behavior together, leading to only 9 classes of instructions. For each class, a single instruction is used as a representative. Among those instructions, two heuristics

(described in sec 4.2) select the set of basic instructions, outputting `DIVPS`, `BSR`, `JMP`, `JNLE`, and `ADDSS`.

Fig. 3.1b shows the output of the *Core Mapping* stage in Fig. 4.1. In practice, abstract resources are internally named R_0, ..., R_5. For convenience we renamed them to the hardware execution ports they correspond to: for example, the abstract resource r_{01} corresponds to the combined use of port p_0 and p_1 for an optimal schedule.

The core mapping also computes a set of saturating micro-benchmarks that individually saturate each of the individual abstract resource. Here, each basic instruction will constitute by itself a saturating micro-benchmark: `DIVPS` will saturate r_0, `BSR` will saturate r_1, `JMP` will saturate r_6, `ADDSS` will saturate r_{01}, and `JNLE` will saturate r_{06}. Note that this is not the case in general: we possibly need to combine several basic instructions together to saturate a resource. Here, the saturating micro-benchmark for resource r_{016} is composed of two basic instructions: `ADDSS` and `JNLE`. The last phase of our algorithm will, for each of the 742 remaining instructions, build a set of micro-benchmarks that combine the saturating kernels with the instruction, and compute its mapping.

4.2 Selection Heuristics

The first step of our algorithm trims the instruction set to extract a minimal set for which the mapping will be computed using a linear programming solver. As this (core) mapping will be reused later, we need enough instructions to detect all resources, but the more instructions we have, the longer the resolution of the linear problem this mapping will take. We thus first apply three filters that reduce the number of basic instructions, as depicted in Algo. 2.

4.2.1 How *not to* Benchmark the Whole ISA

Keeping High IPC instructions

If $\overline{a} < 1$ (measured with a microbenchmark repeating only a), then a is not consider as a candidate for basic instructions. Assuming every physical resource to have a throughput of 1, such instructions use one resource more than once. However, these low-IPC instructions are still mapped at the very last step of PALMED (see Sec. 4.5).

Then, we compute, for every remaining pair of instruction (a, b), the throughput of the microkernel $a^{\overline{a}}b^{\overline{b}}$. This set of benchmarks is called *quadratic benchmarks* (see Fig. 4.1) as their number is quadratic with respect to the number of instructions. These quadratic benchmarks are later reused in the following heuristics.

To speed up measurements, we parallelize the measurements of the quadratic benchmarks on all available cores of our tested CPU (simultaneous multi-threading disabled) using one process per core. Indeed, as only single core performance matters and no share data is used between benchmark, this technique allow full usage of the available processing power without disadvantages. While the CPU may throttle under high load due to power consumption or thermal limits, our measurements are not significantly modified by such behavior as we report time as clock cycles for instructions that do not tamper neither with memory nor with uncore units. Therefore, throttling – which imply decrease of the operating frequency – do not modify these measurements, as they are only linked to internal microarchitectural parameters.

Equivalence Classes

If $\forall p,\ \overline{a^{\overline{a}}p^{\overline{p}}} = \overline{b^{\overline{b}}p^{\overline{p}}}$ then it is useless to keep both a and b. Therefore, this second filter removes duplicates, that is, if two instructions behave similarly with regard to the evaluation used for our basic instruction selection, then one of them can be ignored. Obviously, on a real machine, despite all the crucial efforts to remove execution hazards, the measured IPCs never perfectly match and the correct criteria for selecting a representative instruction for duplicates should approximate the equality test $\forall p,\ \overline{a^{\overline{a}}p^{\overline{p}}} \approx \overline{b^{\overline{b}}p^{\overline{p}}}$. The construction of equivalence classes and associated representative instructions in this context uses hierarchical clustering [47]. We cluster all remaining instructions in multiple hierarchical clustering by varying the number of clusters while keeping a reasonable size (between 50 and 200 in our experiences), and retain the cluster of maximal silhouette.

Function *Select_basic_insts*$(\mathcal{I}, n)$
  $\mathcal{I}_F := \mathcal{I}$;
  `// Remove low-IPC; compute eq. classes`
  foreach $a \in \mathcal{I}_F$ **do**
    if $\overline{a} \leq 1 - \epsilon$ **then** $\mathcal{I}_F := \mathcal{I}_F - \{a\}$;
    if $\exists b \in \mathcal{I}_F,\ \forall p \in \mathcal{I},\ \overline{a^{\overline{a}} p^{\overline{p}}} = \overline{b^{\overline{b}} p^{\overline{p}}}$ **then**
      $\mathcal{I}_F := \mathcal{I}_F - \{a\}$
    end
  end
  `// Select very basic instructions`
  foreach $a \in \mathcal{I}_F$ **do**
    $Dj[a] := \left\{ b \in \mathcal{I}_F, a^{\overline{a}} b^{\overline{b}} = \overline{a} + \overline{b} \right\}$
  end
  let $a <_{VB} b \Leftrightarrow$
    $$(|Dj[a]| > |Dj[b]|) \vee \left(|Dj[a]| = |Dj[b]| \wedge \overline{a} > \overline{b}\right);$$
  $\mathcal{I}_{VB} := \emptyset$;
  for $a \in \mathcal{I}_F$ *in* $<_{VB}$ *order* **do**
    if $\mathcal{I}_{VB} \subset Dj[a]$ **then** $\mathcal{I}_{VB} := \mathcal{I}_{VB} \cup \{a\}$;
    if $|\mathcal{I}_{VB}| = n$ **then return** $\mathcal{I}_B := \mathcal{I}_{VB}$;
  end
  `// Select most greedier instructions`
  $\mathcal{I}_{MG} := \emptyset$;
  for $a \in \mathcal{I}_F$ *in* $\preccurlyeq_{greedier}$ *order* **do**
    $\mathcal{I}_{MG} := \mathcal{I}_{MG} \cup \{a\}$;
    if $|\mathcal{I}_{VB} \cup \mathcal{I}_{MG}| = n$ **then return** $\mathcal{I}_B := \mathcal{I}_{VB} \cup \mathcal{I}_{MG}$;
  end
  return $\mathcal{I}_B := \mathcal{I}_{VB} \cup \mathcal{I}_{MG}$;
end

Algorithm 2: Set of basic instructions $\mathcal{I}_B$

Finally, we select the most representative instruction by taking the one closest to the centroid of each instruction class.

As seen in Sec. 2.2.3, the benchmarked ISA may be broken down into a *initial* set of instructions, complemented by *extensions*, for example vector instructions, that add further compute capabilities to the CPU. As conflicts may happen while mixing instructions of different extensions (such as AVX and SSE on some Intel CPUs), we apply our equivalence class filter independently to each set of extension as well as the original ISA.

Once low IPC instruction duplicates have been removed, the selection relies on two criteria (as detailed in Algo. 2): *max clique* and *min order*.

4.2.2 Max Clique: Very Basic Instructions

Instructions a and b are considered *disjoint* if $\overline{a^{\overline{a}} b^{\overline{b}}} = \overline{a} + \overline{b}$; case in which their IPC is said to be *additive*. The set of *very basic instructions* is defined as a maximal clique of disjoint instructions, as illustrated in Fig. 4.2. This captures instructions that maps to a

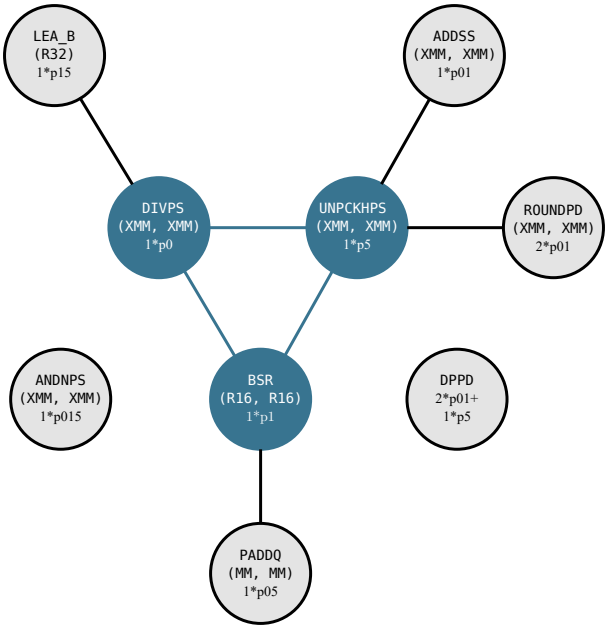


Figure 4.2: Example of Max Clique instructions selection with instruction and port usage from Intel's Skylake microarchitecture. Blue instructions are selected instructions, edges symbolize instructions with additive IPC.

single resource. Indeed, two instructions that do not share any resource will have their IPC additive, thus belonging to the maximum clique of our graph. This selection technique is formally proved to converge (assuming the existence of single-resource saturating instruction) in Sec. 5.2.2.

Note that, in practice, max clique is applied only on instructions belonging to the base ISA, as, on all tested architectures, extensions were not using new execution ports, but only reuse existing ones. This decision makes sense from a high-level point of view, as the instructions of extensions are supposed to be used in *some* parts of programs. Using dedicated executions paths for these instructions to allow concurrent execution with any other instruction is a costly design decision, which would be beneficial only if the new instruction were to be used in *nearly all* parts of any program, which is not the case in practice.

4.2.3 Min Order: Most Greedier Instructions

Instruction a is considered more greedier than b (we note $a \preccurlyeq_{greedier} b$) if $\forall p, \overline{a^{\bar{a}} p^{\bar{p}}} \geq \overline{b^{\bar{b}} p^{\bar{p}}}$. This relation defines a pre-order, and we select the n most greedier instructions (the bigger is n the more complete is the core mapping but also the more complex is the linear program) as output of this filter.

This selection filter is formally proved to converge to a set of instruction saturating a single resource in Sec. 5.2.3, under condition of existence of these instructions.

In practice, we apply the Min Order filter to every instruction set extension separately, as the distribution of execution units for new instructions is unknown. For example, on Skylake, ports `p0`, `p1`, `p5` and `p6` may perform arithmetic operation on integers, which

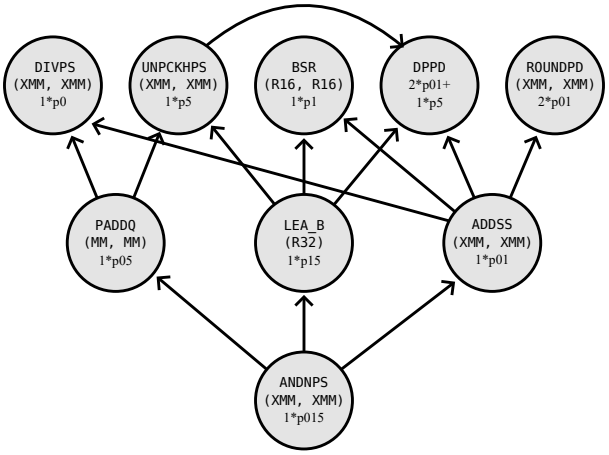


Figure 4.3: Example of Min Order instructions selection with instruction and port usage from Intel's Skylake microarchitecture. Edges symbolize the pre-order relation between instructions.

translates to a single resource `R0156` on the abstract resource model. However, AVX-512 integer instructions of the Skylake-SP microarchitecture only use a subset of these ports (`p0` and `p5`, linked to the `R05` abstract resource), and they are the only one to do so in the whole ISA. Therefore, separate treatment of extensions allow to find these `R05`-saturating instructions while avoiding the selection of other instructions. Without this trick, a larger n value would be required, polluting the selected instructions with other, non-necessary ones.

4.3 Finding the Exact Core Mapping

After having selected the *basic instructions* from the Max Clique and the Min Order filters, the challenge is now to find the complete mapping for these instructions, named *Core Mapping*. In that goal, we propose a two-step approach, based on a Linear Programming (LP) formulation. The first step determines the number of resources present in the CPU with an iterative process that tries to saturate each resource. The second step determines the edge values, that is, the amount of use of each resource by each instruction. Finally, we select for each instruction a saturating benchmark, that we use on every remaining instruction of the ISA in order to determine their resource usage, as detailed in Sec. 4.5.

We developed two variations of the Core Mapping computation. The first one, detailed in this section, is guaranteed to converge to the correct abstract resource mapping that the CPU follows (see Chap. 5 for the complete details), but is based on a high dimension LP that does not converge fast enough in practice with real measures due to complexity of the error rate formulation. The second one, detailed in the next section, is based on smart rules of thumb in order to deduce an approximation of the shape of the mapping, which is fixed later when computing the edge values.

The exact core mapping phase adds one more preliminary step to the Core Mapping computation: the calibration of the error rate, for each pair of instructions. This phase sets the maximum error we allow when computing a resource mapping that best models the observed performance over a set of micro-benchmarks. On perfect architectures (that is, architectures whose behavior can be perfectly described using the abstract resource model), all error rates would be equal to zero.

4.3.1 Determine Hazardous Instructions

Our proved infrastructure relies on our ability to measure the throughput of any multi-set of instructions without being polluted by other execution bottlenecks such as alignment issues of the front-end, that cannot be modeled by the resource mapping formalism. In theory, assuming an ideal machine that matches the port-mapping performance model, for any two instructions a and b, then three resources should be enough to model any combination $\{(i,j) \in \mathbb{N},\ a^i b^j\}$. Real-world experiments show that this is not the case, hence the presence of a modelling error on the IPC of $a^i b^j$-class microbenchmarks. However, some of the instructions show more hazards than others: a first pre-processing step considers all simple instruction pairs (a, b) and evaluates the minimal error $\epsilon(K)$ required to map those two instructions to no more than three resources. This error is extended to every benchmark K of more than two instructions with a fixed constant value.

Function *Mapping*($\mathcal{K}, \mathcal{G}$)
  Solve *Bipartite Weight Problem*
    $\mathcal{I} :=$ instructions($\mathcal{K}$);
    $(\rho_{i,r})_{\mathcal{I},\mathcal{P}} :=$ edges($\mathcal{G}$);
    $\forall (i,r) \in \mathcal{I} \times \mathcal{P},\ 0 \leq \rho_{i,r} \in [0,1]$;
    $\forall (K,r) \in \mathcal{K} \times \mathcal{P}$,
      $\rho_{K,r} = \left(\sum_{i \in \mathcal{I}} \sigma_{K,i}\rho_{i,r}\right) \times \overline{K} / \left(\sum_{i \in \mathcal{I}} \sigma_{K,i}\right)$;
    $\forall (K,r) \in \mathcal{K} \times \mathcal{P},\ \rho_{K,r} \leq 1$;
    $\forall K \in \mathcal{K},\ S_K = \max_{r \in \mathcal{P}} \rho_{K,r} \geq 1 - \epsilon(K)$;
    `// With` $\epsilon(K)$ `the error rate computed in the hazardous instructions step (Sec. 4.3.1)`
    Minimize $\sum_{i \in \mathcal{I}} \rho_{i,r}$;
  return $(\mathcal{I}, \mathcal{P}, \{\rho_{i,r}\})$;
end

Algorithm 3: Bipartite Weight Problem (BWP), as used in the exact-$\texttt{LP}_1$, $\texttt{LP}_2$ and $\texttt{LP}_{\texttt{AUX}}$

4.3.2 Bipartite Weight Problem (BWP)

The proved variations of PALMED is based on a single LP formulation, the Bipartite Weight Problem, or BWP, which is first applied to a constant seed of microbenchmarks, then enriched to grant completeness of the output mapping.

The BWP is formalized in Alg. 3, and aims at finding both the shape and the correct values of the edges of a Core Mapping at the same time. We use the following notations: $\rho_{i,r} \in \mathbb{Q}^+$ expresses the proportional usage of the resource r by instruction i, and $\overline{K}$ the average number of instructions executed each cycle when K is executed by the CPU. The proportion of a resource r that is used is thus $\rho_{K,r} = \overline{K} \cdot \left(\sum_{i \in \mathcal{I}} \sigma_{K,i}\rho_{i,r}\right) / \left(\sum_{i \in \mathcal{I}} \sigma_{K,i}\right)$, bounded by its throughput ($\rho_{K,r} \leq \rho_r = 1$). One of the resources must be the limiting factor, that is, $\exists r,\ \rho_{K,r} = 1$. However, we authorize sub-saturation of the resources, acknowledging our model does not predict accurately every microkernel, and we note $S_K = \max_r \rho_{K,r} \leq 1$. These constraints form our linear problem minimizing the sum of the edges values in order to decrease the complexity of the mapping, that is:

$$\min \sum_{i \in \mathcal{I}} \rho_{i,r}$$

4.3.3 Characterize Resources ($\texttt{LP}_1$)

The aim of the first LP step is to guarantee that the core mapping is as complete as possible: in particular, it should not miss any a resource. Therefore, $\texttt{LP}_1$ is an iterative process solving at each step a new instance of the BWP, aiming at finding all existing resources starting from an initial seed of benchmarks. The $\texttt{LP}_1$ then stops when a stabilization of the number of resources is reached.

The initial set of microbenchmarks, which needs to be as "representative" as possible, is derived from the simple instructions with the following rules:

- $a \in \mathcal{I}$ alone
- $a^{\overline{a}}b^{\overline{b}}$, as this benchmark has the following property: If a and b are independent, that is the set of resources used by a and b are disjoint, or have a cumulated usage that does not exceed $\frac{1}{\overline{a}+\overline{b}}$, then $\overline{a^{\overline{a}}b^{\overline{b}}} = \overline{a} + \overline{b}$
- $a^M b$ (with $M = 4$ in practice) to avoid the convergence of the solver to a simpler solution with fewer resources and edges representing only the conflicting case $a^{\overline{a}}b^{\overline{b}}$

The iterative process that follows is:

1. Starting from the initial set, find a mapping by solving the BWP.
2. For each abstract resource, construct a news microbenchmark composed of every instruction that are using it.
3. Solve this news BWP instance. If the new solution leads to the same number of resources than the former iteration, stop. Else, return to step 2.

The enrichment (step 2.) is done as follows: for each resource found, we add a benchmark composed of every instruction using it (with a fixed minimal threshold ϵ) with a multiplicity of their IPC, forcing the split of resources in case of undesired merges. Once convergence has been reached, we expect all existing resources to be discovered. Then, PALMED must ensure that every existing instruction-to-resource edge is represented.

4.3.4 Find Saturating Kernels ($\texttt{LP}_2$)

Once all resources have been found, we now have to find the correct values of each edge. For this purpose, we extract for each resource r a saturating kernel $sat[r]$ that we combine with every instruction i to build a new microbenchmark, and add it to our iterative microbenchmark set.

$$K_{sat}(i, r) = i^1(sat[r])^N$$

where N is chosen bigger than $4 \times \overline{sat[r]}/\overline{i}$. Then, we solve one last BWP instance. The proof of the completeness of the mapping, given a set of saturated benchmarks with a low usage of any other resource, is detailed in Sec. 5.

The saturating kernel $sat[r]$ is chosen among all saturating microbenchmarks of the last $\texttt{LP}_2$ as the benchmark K that has minimum consumption:

$$cons(K) = \sum_{i \in \mathcal{I},\ r \in \mathcal{P}} \rho_{i,r}$$

The $\texttt{LP}_2$ finally returns the Core Mapping as well as a new, final set of saturating benchmarks, that are used to find the port mapping of the complete ISA as described in Sec. 4.5. The complete flow is recapitulated in Alg. 4.

```
1  Function Core_mapping(I_B)
       // Determine hazardous instructions
2      K := ⋃_{(a,b)∈I_B^2, a≠b} { a, a^ā b^b̄, a^M b };
       // Characterize resources
3      do
4          G := Mapping(K) ;                                   // LP_1
5          K_new := ⋃_{r∈P} { Π_{i∈I_B, ρ_{i,r}≥ε} i^ī } − K;
6          K := K ∪ K_new;
7      until K_new = ∅;
       // Find saturating kernels
8      foreach r ∈ R do
9          sat[r] := K ∈ K s.t. ρ_{K,r} = 1 that minimizes cons(K);
10         for i ∈ I_B s.t. i ∉ sat[r] do
11             K := K ∪ {K_sat(i, r)};
12         end
13     end
14     G := Mapping(K) ;                                       // LP_2
       // Find final saturating kernels
15     foreach r ∈ R do
16         sat[r] := K ∈ K s.t. ρ_{K,r} = 1 that minimizes cons(K);
17     end
18     return K, sat, G;
19 end
```

Algorithm 4: Find core mapping and saturating kernels (exact version)

4.4 Faster approximation of the Complete Resource Mapping Problem

While the approach detailed in the former section is formally proven, this does not mean that it is satisfactory in practice. Indeed, the proof (see Sec. 5) assume existence of single-resource instructions, which is not granted true for all architectures (especially for combined resource), and assumes that all benchmarks are perfect, that is, without measurement noise nor resource that does not follow the abstract resource model.

Therefore, when adding error rates to benchmarks (and minimizing them as a secondary target), the solver times out on all of our experiences (see Sec. 4.6) and only offers approximated solutions. We thus present another approach, based on simple and inexact rules of thumb to determine the number of resources and then minimizing directly the error rate, but which reveals to be both faster and more accurate in practice.

4.4.1 Finding the Shape of the Mapping: ($\texttt{ILP}_1$)

For this variation, the goal of the first step is to find the *shape* of the resource mapping, that is, the number of resources needed, but also the possible edges from core instructions to resources. For this, PALMED solves the following Integer Linear Programming (ILP) problem, formalized in Alg. 5, repeated until no new benchmark is added[3] (note that the *hazardous instructions* step becomes superfluous in this fast solving version):

- **Objective function:** Minimize the number of resources.
- **Constraints:** From the same seed of microkernels as Sec. 4.3.3, we derive the following constraints (in the order of Alg. 5):
  - Each very basic instruction as defined in Sec. 4.2.2 is linked to at least one resource unused by other very basic instructions (line 4).
  - For each greedier instruction i as defined in Sec. 4.2.3, there exists at least one resource common to i and to all other instructions a for which $\overline{i^{\bar{i}}a^{\bar{a}}} \neq \bar{i}+\bar{a}$ (line 5). This relation corresponds to the negation of the *disjoint* relation defined in Sec. 4.2.2, that we note $\not\asymp$.
  - For all other microkernels:
    1. Every instruction identified as saturating (that is, instructions for which the execution time of the microkernel is equal to its execution time alone) maps to at least a resource unused by other instructions of the microkernel (line 7).
    2. If no saturating instruction is found, then there exists a resource shared by every instruction of the benchmark (line 10).

[3]This corresponds to an approximation of the $\texttt{LP}_1$ described in Sec. 4.3.3.

Function $Shape_mapping(\mathcal{K}, \mathcal{I}_{VB}, \mathcal{I}_{MG})$
  Solve
    $\forall (i,r) \in \mathcal{I} \times \mathcal{P},\ \rho_{i,r} \in \{0,1\}$;
    $\forall i \in \mathcal{I}_{VB}, \min_{r\in\mathcal{P}} 1 - \rho_{i,r} + \sum_{j\in\mathcal{I}_{VB}\setminus\{i\}} \rho_{j,r} = 0$;
    $\forall i \in \mathcal{I}_{MG}, \max_{r\in\mathcal{P}} \rho_{i,r} + \sum_{j\gtrdot i} \rho_{j,r} = 1 + |\{j \gtrdot i\}|$;
    foreach $K \in \mathcal{K}$ *s.t.* $\{i^\alpha \in K \text{ } s.t. \text{ } cycles(i^\alpha) = cycles(K)\} = \emptyset$ **do**
      $\max_{r\in\mathcal{P}} \sum_{i\in K} \rho_{i,r} \geq |\{i \in K\}|$;
    end
    foreach $K \in \mathcal{K}$ *s.t.* $\{i^\alpha \in K \text{ } s.t. \text{ } cycles(i^\alpha) = cycles(K)\} \neq \emptyset$ **do**
      $\forall i^\alpha \in K$ s.t. $\text{cycles}(i^\alpha) = \text{cycles}(K)$
        $\min_{r\in\mathcal{P}} 1 - \rho_{i,r} + \sum_{j\in K, j\neq i} \rho_{j,r} = 0$;
    end
    Minimize $\sum_{i\in\mathcal{I}_B} \max_{r\in\mathcal{P}} \rho_{i,r}$;
  return $(\mathcal{I}, \mathcal{P}, \{\rho_{i,r}\})$;
end

Algorithm 5: $\texttt{ILP}_1$: Approximation of the shape of core mapping

The enrichment is kept the same as $\texttt{LP}_1$: for each resource found, we add a benchmark composed of every instruction using it with a multiplicity of their IPC, with the same goal of splitting resources in case of undesired merges. Once convergence has been reached, we also expect *most* of existing resources and edges to be discovered.

4.4.2 Find Saturating Kernels ($\texttt{LP}_2$)

After having found the shape of the mapping consisting of the possible resources and instructions-to-resource edges; the $\texttt{LP}_2$ must find the value of the edges. Contrary to Sec. 4.3.4, the BWP is modified by forcing the number of resource and the non-zero edges so that only edges and resources discovered by the $\texttt{ILP}_1$ exist. Moreover, no further benchmark is added compared to the this last $\texttt{ILP}_1$ run.

Then, instead of relying on a costly, per-benchmark error rate, the objective function is modified to minimized the sum of *one-sided error rates*. Indeed, we consider that they may exists resources not detected by the $\texttt{ILP}_1$, but that a benchmark still cannot exceed its predicted throughput. Therefore, the objective function is changed to become:

$$\text{Minimize} \sum_{K\in\mathcal{K}} err(K) = S_K - 1$$

Finally, and similarly to the proved version, the set of saturating kernels is deduced from the last $\texttt{LP}_2$ step and returned, as well as the resource profile of the Basic Instructions. The complete approximate version of the $\texttt{LP}_2$ is detailed in Alg. 6.

Function *Core_ mapping($\mathcal{I}_B$)*

`// Characterize resources`

$\mathcal{K} := \bigcup_{(a,b)\in\mathcal{I}_B^2,\ a\neq b} \left\{a,\ a^{\bar{a}}b^{\bar{b}},\ a^M b\right\}$;

do

$\mathcal{G} :=$ Shape_Mapping$(\mathcal{K}, \mathcal{I}_{VB}, \mathcal{I}_{MG})$; `// ILP`$_1$

$\mathcal{K}_{new} := \bigcup_{r\in\mathcal{P}} \left\{\Pi_{i\in\mathcal{I}_B,\ \rho_{i,r}\geq\epsilon} i^{\bar{i}}\right\} - \mathcal{K}$;

$\mathcal{K} := \mathcal{K} \cup \mathcal{K}_{new}$;

until $\mathcal{K}_{new} = \emptyset$;

$\mathcal{G} :=$ Mapping$(\mathcal{K}, \mathcal{G})$; `// LP`$_2$

`// Find final saturating kernels`

foreach $r \in \mathcal{R}$ **do**

$sat[r] := K \in \mathcal{K}$ s.t. $\rho_{K,r} = 1$ minimizing $cons(K)$;

end

return $\mathcal{K}, sat, \mathcal{G}$;

end

Algorithm 6: Core mapping and saturating kernels (approximated version)

$\mathcal{I}_B :=$ select_basic_insts$(\mathcal{I}, n)$;
$\mathcal{K}, sat, \mathcal{G} :=$ Core_mapping$(\mathcal{I}_B)$;
foreach $inst \in \mathcal{I}$ **do**
    $\mathcal{K} := \bigcup_{r \in \mathcal{P}} K_{sat}(inst, r)$;
    $\mathcal{I} := \mathcal{I}_B \cup \{inst\}$;
    Solve *Find a solution to the following problem*
        $\forall r \in \mathcal{P},\ 0 \leq \rho_{inst,r}$;
        $\forall (K, r) \in \mathcal{K} \times \mathcal{P},\ \rho_{K,r} = \left(\sum_{i \in \mathcal{I}} \sigma_{K,i} \rho_{i,r}\right) \times \overline{K} / \left(\sum_{i \in \mathcal{I}} \sigma_{K,i}\right)$;
        $\forall (K, r) \in \mathcal{K} \times \mathcal{P},\ \rho_{K,r} \leq 1$;
        $\forall K \in \mathcal{K},\ S_K = \max_{r \in \mathcal{P}} \rho_{K,r}$;
        Minimize $\sum_{K \in \mathcal{K}} (1 - S_K)$;
end

Algorithm 7: $\texttt{LP}_{\texttt{AUX}}$: Complete resource mapping

4.5 Complete Mapping ($\texttt{LP}_{\texttt{AUX}}$)

Whichever flavor of the Core Mapping is used, the last step remains identical, corresponding to Algo. 7: solving an optimization problem for each remaining instruction. Its formulation is once again based on a modified BWP, except that the resources and the edges of the core mapping computed previously are frozen. The presence or absence of an edge from the to-be-mapped instruction i to a resource r is constrained by using $K_{sat}(i, r) = i^{\bar{i}} sat[r]^{L * \overline{sat[r]}}$ in the set of microbenchmarks, with $L = 4$ in practice (identical to the enrichment presented in Sec. 4.3.3, based on the proof from Sec. 5.3). The idea is to force the saturation of r by charging it with $sat[r]$, hence expressing the usage of r by i.

Note that we first realize all the benchmarks, then collapse the instructions into equivalence classes of *exact* same IPC and behavior with respect to the saturating benchmarks, then solve one $\texttt{LP}_{\texttt{AUX}}$ per equivalence class. The final output of PALMED is then the result of these LPs: the resource usage of each instruction in the targeted ISA.

4.6 Evaluation: Basic Blocks Throughput Prediction without Dependencies

While PALMED can in theory recover any resource mapping, things are not that simple in practice. Indeed, real CPUs cannot be described only by the abstract resource model, and benchmark measurements are always subject to noise (either from the operating system, the environment or the interference of other components such as branch prediction on long microbenchmarks). Therefore, this section is divided into two parts: the first one (Sec. 4.6.1) shows that PALMED's exact version is able to find perfectly the resources derived from a usual 3-level port mapping taken from uops.info [33]. The second one (Sec. 4.6.2) evaluates both version of PALMED on two real-world CPU microarchitecture: Intel's Skylake-SP and AMD's first generation Zen.

4.6.1 Exact Port Mapping Recovery

Here, our goal is to show that PALMED is able to find a correct disjunctive mapping, given (i) an execution model matching the dual representation of uops.info's mapping, (ii) ideal microbenchmarking results, i.e. without any constraint on the total number of instructions per microbenchmark and without rounding error. Because of the idealized nature of the simulations, the error rate given to the ILP solver was extremely tight: we set the maximum relative error between a microbenchmark simulation (by the abstract model) and the benchmark IPC (computed from an ideal representation) to 10^{-7}, corresponding to rounding errors made internally by the solver. We test every Intel microarchitecture supported up to Cannon Lake, and report the results in Table 4.2. Solving is achieved using Gurobi [48] version 9 on an Intel i9-7940X running Arch Linux (kernel version 5.8.1).

Silent resources On some architectures, certain ports cannot be detected by PALMED, as they are "hidden" under another resource. More generally, a *silent port* is a port p_s for which every instruction that uses this port also uses another fixed port p_m, which masks it.

Given this matter of fact, p_m will always be saturated before p_s, so *hidden ports are never bottlenecks of the execution*. It follows that their representation is not required for performance modeling, so we do not classify their absence as errors of the mapping.

Our algorithm successfully computes a disjunctive mapping corresponding exactly to uops.info's dual mapping on all tested architectures. Silent resources are not expressed by these mappings: for example, up to the Westmere architecture, port 3 was dedicated to the generation of memory addresses for stores only, whereas port 4 handles the memory access itself, hence we detect only abstract resources derived from port 4 for those microarchitectures.

Architecture codename	Nb. of eq. classes	Silent ports	Nb. of found res.
Conroe	161	p3 / p4	8
Wolfdale	157	p3 / p4	8
Nehalem	147	p3 / p4	8
Westmere	156	p3 / p4	8
Sandy Bridge	186	None	9
Ivy Bridge	184	None	9
Haswell	218	p7	12
Broadwell	222	p7	12
Skylake	217	p7	14
Skylake-X	288	p7	14
Kaby Lake	205	p7	14
Coffee Lake	210	p7	14
Cannon Lake	242	p7	14

Table 4.2: Number of detected equivalent classes and resources for an ideal CPU simulated from uops.info's mapping

4.6.2 Real-world CPU Throughput Prediction

This section compares throughput accuracy on assembly microkernels extracted from two benchmarking suites: the SPECrate version of SPECint2017 [49] and Polybench [50].

We compare both version of PALMED (*exact* and *approximate*) against the native execution, along with the predictions of four existing tools: IACA [25], PMEvo [37], llvm-mca [43] and the port mapping deduced from uops.info's work [33].

Our evaluation is performed on two architectures: the SKL-SP is an Intel Xeon Silver 4114 CPU at 2.20 GHz, using Debian, Linux kernel 4.19 and PAPI 6.0.0.1 to collect the execution time in cycles for each microbenchmarks, restraining to non-AVX-512 instructions due to PIPEDREAM limitations. The ZEN is an AMD EPYC 7401P CPU at 2 GHz with a similar software setup. Solving is achieved using Gurobi [48] version 10, installed on each machine, set with a timeout of 30 minutes. For each of these two architectures, the number of generated microbenchmarks, resources found and mapped instructions are gathered in Table 4.3.

Calibration of the Model

The port mapping is computed using the algorithm presented in Sec. 4.5 using a list of x86 instructions extracted from Intel's XED [51]. We discard instructions which cannot be instrumented in practice, such as instruction modifying the control flow (as our microbenchmark generator cannot handle non-trivial control flow in the instrumented instructions), privileged instructions, along with instructions whose IPC is lower than 0.05, as they do not present any interest for performance prediction of throughput-limited microkernels. While benchmarking memory instructions, we ensure that every access hits the L1 cache to avoid cache-related bottlenecks, which are out of PALMED's scope. Due

to the complexity of the x86 instruction set, we separate the SSE and AVX instructions from the "base ISA": we apply separately the heuristics of Sec. 4.2 before gathering all selected instructions in a single combined *basic instructions* set as described in Fig. 4.1.

A stated in Sec. 4.2, we also forbid benchmarks combining different extensions (e.g. SSE+AVX). Indeed, combinations of several vector extensions of different width are known to cause extra latency, that is, a sort of dependency from one instruction to the other (two consecutive SSE instructions would not be penalized, whereas one SSE and one AVX will). This violates our assumption that the relative order of instruction does not matter, and in practice we observed a significant degradation of the mapping without this mitigation.

Because of variations in the real-world measurements, we constrain the error rate to 0.05 for the micro-benchmark coefficients, meaning that the number of repetitions of an instruction inside its microkernel nay differ by at most 5% from what the algorithm requires. For example, a benchmark $a^{\overline{a}}b^{\overline{b}}$ with $\overline{a} = 0.06$ and $\overline{b} = 1$ will be rounded to a^1b^{20}. Note that in the BWP defined in Alg. 4.3.2, we use the rounded coefficients and not the ideal ones; and the IPC is also rounded accordingly. Note that our microbenchmark generator is pre-constrained with these limitations; therefore we did not evaluate PALMED with another measurement back-end – although we expect similar results as we ensured to have reproducible execution times.

Throughput Estimations

To evaluate PALMED, the same microkernel is run:

1. natively on each CPU, with the IPC measured with the `CPU_CLK_UNHALTED` performance counter
2. using our mapping with abstract resources corresponding to the actual machine, as described in Sec. 4.6.2, both using the exact and the approximation variants
3. using Abel's work (uops.info) [33], by running a conjunctive mapping with exact compatibility and approximating the execution time by the port with the highest usage
4. using PMEvo [37], ignoring any instruction not supported by its provided mapping
5. using IACA [25], by inserting assembly markers around the kernel and generated by our back-end and running the tool with this assembly
6. using llvm-mca [43], also by inserting markers in the assembly code generated by our back-end and running the tool

Unlike PMEvo and llvm-mca, UOPS and IACA do not support the ZEN architecture, hence the absence of data.

The microkernels are extracted from two reputed benchmark suites: SPECInt2017 [49] and Polybench [50]. For Polybench, we used QEMU [52] to gather the translation blocks executed at runtime along with their number of executions. For SPEC, we used static

binary analysis tools to extract the basic blocks along with performance counters statistics in order to recover the performance-critical section of the code, as the cost of running an emulator was too high to reproduce Polybench's setup. Overall these two benchmark suites generate thousands of basic blocks, and for each we use the various methods above to display the predicted performance of a microkernel made of the same instruction mix that is occurring in that basic block. This evaluation approach allows to generate a high variety of realistic instruction mixes (e.g., combining SIMD and address calculations for numerical kernels like in Polybench).

Fig. 4.4 synthesizes our results in two pieces. First, Fig. 4.4a displays the results as a heatmap for each basic block, comparing the predicted throughput with the measured one. A dark area at coordinate (x, y) means that the selected tool has a prediction accuracy of y for a significant number of microkernels with a real IPC of x.

Then, Tbl. 4.4b and Tbl. 4.4c synthesize, for each tool, its error rate, aggregated over all the basic blocks of the test suite using a *Root-Mean-Square* method:

$$\mathrm{Err}_{\mathrm{RMS,\ tool}} = \sqrt{\sum_i \frac{\mathrm{weight}_i}{\sum_j \mathrm{weight}_j} \left(\frac{\mathrm{IPC}_{i,\mathrm{tool}} - \mathrm{IPC}_{i,\mathrm{native}}}{\mathrm{IPC}_{i,\mathrm{native}}} \right)^2}$$

We also provide Kendall's τ coefficient [53], a measure of the rank correlation of a predictor – that is, for each pair of basic blocks, whether a predictor predicted correctly which block had the higher IPC. The coefficient varies between -1 (full anti-correlation) and 1 (full correlation).

The same table also provides a *coverage* metric, *with respect to* PALMED. This metric characterizes the proportion of basic blocks supported by PALMED that the tool was able to process. Note that the ability to process a basic block varies from tool to tool: some work in degraded mode when meeting instructions they cannot handle, some will crash on the basic block. For PMEvo, we ignored any instruction not supported by their mapping – degrading the quality of the result; hence, a plain error is a basic block in which *no single instruction was supported*. Although it would be fairer to other tools to measure absolute coverage – that is, the proportion of basic blocks supported by the tool, regardless of what PALMED supports –, technical limitations prevented us from doing so: running the various tools requires our back-end to generate assembly code, which can only be done for the instructions it supports.

We compare the number of instructions supported by PALMED with the ones supported by uops.info as a baseline. As uops supports only partially AMD's architecture (providing only throughput and latencies, but no usable port mapping), less than half the instructions supported by our tool are present for this target. Note the slight difference in the number of instructions supported by both versions of PALMED: this is explained by patches on the PIPEDREAM library, whose revision is different between the two PALMED version. Contrarily, on SKL-SP, uops supports the AVX-512 extension, therefore leading to a more complete set of supported instructions. PMEvo's mapping behaves poorly in

Machine	SKL-SP	ZEN
Processor	2x Intel Xeon Silver 4114	AMD EPYC 7401P
Cores	20	24
Benchmarking time	8h	6h
LP solving time (Exact Method)	4h (timeout)	4h (timeout)
LP solving time (Approximation)	2h	2h
Gen. microbenchmarks	$\sim$ 1,000,000	$\sim$ 1,000,000
Resources found (Exact Method)	10	5
Resources found (Approximation)	17	17
uops' inst. supported	3313	1104
Instructions mapped (Exact Method)	2598	2592
Instructions mapped (Approximation)	2586	2596

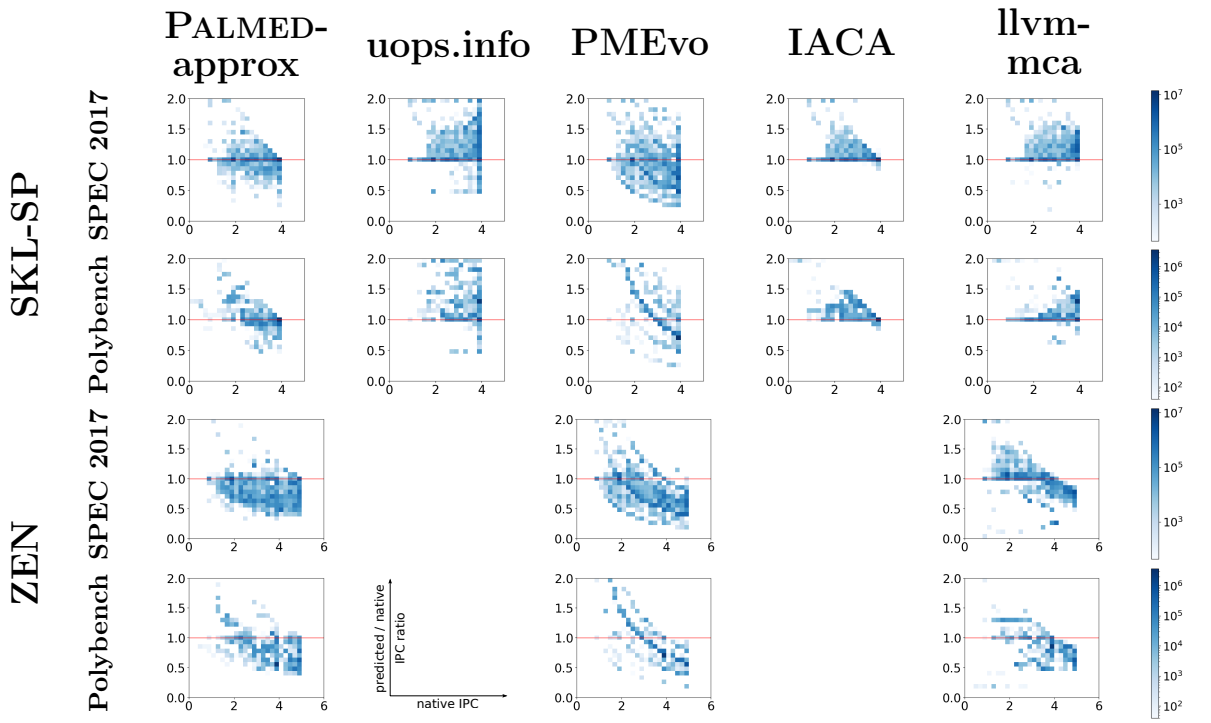
Table 4.3: Main features of the obtained mappings

terms of coverage (see Fig. 4.4), failing to support all instructions in more than 25 % of the basic blocks on any benchmark and processor tested. This behavior is due to our different compilation options, as PMEvo's supported instructions are directly collected from their SPEC2017 binaries. As a consequence, both MSE and Kendall's tau values are lower than other tools as those unsupported instructions are treated as if they took no resource at all on our IPC estimates.

Moreover, PALMED-approx requires 2h of solving time (see Tbl. 4.3) to map about 2500 instructions. This is between one half (SKL-SP) and one eighth (Zen) of PMEvo's solving time [33], demonstrating the scalability of PALMED with respect to the number of instructions. Contrarily, PALMED-exact times out on both tested architectures, while outputting mapping of lower quality. This is mainly due to the number of resource found by the $\texttt{LP}_1$, lower on the exact version than on the approximated one. Indeed, disambiguation of resources requires an accurate measurements of the IPC of benchmark with multiple instructions, which enters in contradiction with the presence of an error rate. This leads to a Core Mapping using fewer resource than necessary (10 on SKL-SP versus 17 found by PALMED-approx), hurting the accuracy of the complete mapping. The approximate version forces a minimal number of resources due to the constraints derived from the Max Clique filter, hence mitigating this effect.

In Fig. 4.4, we observe that PALMED-approx reaches a hight accuracy than uops.info and PMEvo on both platforms. On Skylake, it outperforms all other tested tools in terms of Kendall's tau, and compares well with IACA and LLVM-MCA, archiving sub-10 % mean square error rate on SPEC2017. However, those two last tools use manual expertise and are tailored for a platform, whereas our tool is fully automated and generic.

On ZEN, both versions of PALMED are comparable to LLVM-MCA, but show a greater error rate than on Intel. This is due to the internal organization of the ZEN microarchitecture, which uses a separated pipeline for integer/control flow and floating point/vector operations. This layout is more complex than Intel's, for which execution units for general



(a) IPC prediction profile heatmaps – predictions closer to the red line are more accurate. Predicted IPC ratio (y axis) against native IPC (x axis)

	PALMED-exact		PALMED-approx			uops.info			PMEvo			IACA			llvm-mca		
Unit	Cov. (%)	Err. (%)	Cov. (%)	Err. (%)	τ_K (1)	Cov. (%)	Err. (%)	τ_K (1)	Cov. (%)	Err. (%)	τ_K (1)	Cov. (%)	Err. (%)	τ_K (1)	Cov. (%)	Err. (%)	τ_K (1)
SPEC2017	N/A	19.2	N/A	7.8	0.90	99.9	40.3	0.71	71.3	28.1	0.47	100.0	8.7	0.80	96.8	20.1	0.73
Polybench	N/A	27.4	N/A	24.4	0.78	100.0	68.1	0.29	66.8	46.7	0.14	100.0	15.1	0.67	99.5	15.3	0.65

(b) Translation block coverage (Cov.), root-mean-square error on IPC predictions (Err.) and Kendall's tau correlation coefficient (τ_K) compared to native execution for SKL-SP architecture

	PALMED-exact		PALMED-approx			uops.info			PMEvo			IACA			llvm-mca		
Unit	Cov. (%)	Err. (%)	Cov. (%)	Err. (%)	τ_K (1)	Cov. (%)	Err. (%)	τ_K (1)	Cov. (%)	Err. (%)	τ_K (1)	Cov. (%)	Err. (%)	τ_K (1)	Cov. (%)	Err. (%)	τ_K (1)
SPEC2017	N/A	18.9	N/A	29.9	0.68	N/A	N/A	N/A	71.3	36.5	0.43	N/A	N/A	N/A	96.8	33.4	0.75
Polybench	N/A	32.2	N/A	32.6	0.46	N/A	N/A	N/A	66.8	38.5	0.11	N/A	N/A	N/A	99.5	28.6	0.40

(c) Translation block coverage (Cov.), root-mean-square error on IPC predictions (Err.) and Kendall's tau correlation coefficient (τ_K) compared to native execution for Zen architecture

Figure 4.4: Accuracy of IPC predictions compared to native execution of PALMED versus uops.info, PMEvo, IACA and llvm-mca on SPEC CPU2017 and PolyBench/C 4.2

computation and vector operations share the same ports. This translates to a greater number of abstract resources for the ZEN CPU and more complex interactions than what the abstract model is able to represent – which adds to the fact that both version of PALMED seek to minimize the number of resources. Moreover, we suppose that the hypothesis of a perfect scheduling may not be true for ZEN, as most of the benchmark's IPC are lower than the one predicted, as seen on Fig. 4.4a.

More generally, IACA, uops.info and LLVM-MCA tend to over-estimate the IPC, which is due to their port-based approach: bottlenecks coming from neither ports nor

front-end limitations are not taken into account, leading to higher IPC estimations for microkernels where other resources are bottlenecking. Contrarily, benchmarking-based approaches (Palmed and PMEvo) present both under and over approximations as they are based on real-life execution, where all bottlenecks are present. Note that Palmed, IACA, LLVM-MCA (ZEN only) and PMEvo (ZEN only) also express the front-end bottleneck: the limit on the maximal number of instructions being decoding in one cycle (no over-approximation of microkernels with high IPC), that is, a maximal IPC of 4 on SKL-SP and 5 for ZEN. Therefore, we expect Palmed (and PMEvo) to have maximal error rate on benchmarks with few instructions, case in which some undetected / wrongly detected common resources will have higher importance, whereas LLVM-MCA, uops.info and IACA will tend to be more fragile on long microkernels with possible non-port related resources – especially buffer ones.

Chapter 5
Formal Proofs of Convergence of PALMED

While Sec. 4.3 offers an "exact" solution in the sense that it perfectly recover simulated port mappings (as shown in Sec. 4.6.1), this empirical evidence is not sufficient to claim completeness and generality of PALMED.

This section remedies this issue by providing formal proofs of convergence of PALMED flow of work, under the assumption of existence of instructions using “few” resources in the abstract resource model.

In Sec. 5.1, a proof of equivalence between the well-known port mapping problem, where one instruction can be executed by one port among a set of compatible ones, is proposed. Then, Sec. 5.2 and Sec. 5.3 builds over this result to show first that PALMED's selection mechanism succeed in selecting instructions with a propriety called 1/4-exclusive saturation, and then that they are used in our sequence of LP in a way that detects all possible resources with correct usage.

5.1 Equivalence of the Abstract Resource Model and the Port Mapping Model

5.1.1 Primary Definitions

Definition 5.1.1 (Microkernel)**.** *A microkernel K is an* infinite loop *made up of a finite multiset of instructions, $K = I_1^{\sigma_{K,1}} I_2^{\sigma_{K,2}} \cdots I_m^{\sigma_{K,m}}$ without dependencies between instructions. The number of instructions executed during one loop iteration is $|K| = \sum_i \sigma_{K,i}$.*

Definition 5.1.2 (Disjunctive port mapping)**.** *A disjunctive port mapping is a bipartite graph $(V, \mathcal{P}, E)$ where: V represents the set of μOPs; $\mathcal{P}$ represents the set of resources (corresponding to execution ports in a real-world CPU); $E \subset V \times \mathcal{P}$ expresses the possible mappings from μOPs to ports. In this original form each port $r \in \mathcal{P}$ has a throughput $\rho(r)$ of 1.*

Let $K = I_1^{\sigma_{K,1}} I_2^{\sigma_{K,2}} \cdots I_m^{\sigma_{K,m}}$ be a microkernel where each instruction is composed of a single μOP v_i.

A valid assignment represents the choice of which resources to associate with a given instance of an instruction. However, this choice might change between iterations. Thus, we represent the valid assignment as a mapping $p : \mathcal{I} \times \mathcal{P} \mapsto [0; 1]$ where $p_{i,r}$ corresponds to the frequency a given resource is chosen. We also define $R_i(p) = \{r,\ p_{i,r} \neq 0\}$. This assignment is valid if:

$$\forall I_i \in K, \forall r \in R_i(p),\ (v_i, r) \in E$$

$$\forall I_i \in K,\ \sum_{r \in R_i(p)} p_{i,r} = 1$$

The execution time *of an assignment $(p_{i,r})_{i,r}$, is:*

$$t_{end} = \max_{r \in \mathcal{P}} \sum_{i \in K} \sigma_{K,i} \cdot p_{i,r}$$

The minimal execution time over all valid assignments is denoted $t(K)$ (obtained using an optimal assignment*).*

Definition 5.1.3 (Conjunctive port mapping)**.** *A conjunctive port mapping is a bipartite* weighted *graph $(I, \mathcal{P}, E,\ \rho_{I,\mathcal{P}})$ where: I represents the set of instructions; $\mathcal{P}$ represents the set of* abstract resources*; $E \subset I \times \mathcal{P}$ expresses the required use of abstract resources for each instruction;*

Each abstract resource $r \in \mathcal{P}$ has a (normalized) throughput of 1; an instruction i that uses a resource r ($(i, r) \in E$) always uses the same proportion (number of cycles, possibly lower/greater than 1) $\rho_{i,r} \in \mathbb{Q}^+$; if i does not use r, then $\rho_{i,r} = 0$.

Let $K = I_1^{\sigma_{K,I_1}} I_2^{\sigma_{K,I_2}} \cdots I_m^{\sigma_{K,I_m}}$ be a microkernel. In a steady state execution of K, for each loop iteration, instruction i must use resource r $(\sigma_{K,i} \cdot \rho_{i,r})$ cycles.

The number of cycles required to execute one loop iteration is:

$$t(K) = \max_{r \in \mathcal{P}} \left(\sum_{i \in K} \sigma_{K,i} \cdot \rho_{i,r} \right)$$

One should observe that Def. 5.1.3 defines formally a *normalized* version of the graph used in the illustrative example of Sec. 3.2.2; where throughputs of abstract resources are set to 1. For the sake of clarity, we used non-normalized throughputs (that is, different than 1) in Fig. 3.1b with the following notations: *use* stands for the non-normalized usage, and *load* for the normalized $\rho_{i,r}$, equal to $\frac{\#use_i}{throughput(r)}$. For example, `VCVTT` uses 2 times r_{01}, which has a throughput of 2: its load $\rho_{\texttt{VCVTT},r_{01}}$ is equal to 1. Similarly, $\rho_{\texttt{ADDSS},r_{016}} = 1/3$.

Definition 5.1.4 (Throughput)**.** *The throughput $\overline{K}$ of a microkernel K is its instruction per cycle rate (IPC), defined as:*

- $$\overline{K} = \frac{|K|}{t(K)} = \max_{valid\ assignment\ p} \left(\frac{\sum_{i \in K} \sigma_{K,i}}{\max_{r \in \mathcal{P}} \sum_{i \in K} \sigma_{K,i} \cdot p_{i,r}} \right)$$
  for a disjunctive port mapping.
- $$\overline{K} = \frac{|K|}{t(K)} = \frac{\sum_{i \in K} \sigma_{K,i}}{\max_{r \in \mathcal{P}} \sum_{i \in K} \sigma_{K,i} \cdot \rho_{i,r}}$$
  for a conjunctive port mapping

Example Given a and b two instructions, $a^{\overline{a}} b^{\overline{b}}$ represents a microkernel repeating a and b as many times as their respective IPC $\overline{a}$ and $\overline{b}$. We note its throughput $\overline{a^{\overline{a}} b^{\overline{b}}}$.

Definition 5.1.5 (∇-dual conjunctive port mapping)**.** *Let $(V, \mathcal{P}, E)$ be a disjunctive port mapping. Let ∇ be a non-empty set of subsets of $\mathcal{P}$. We define its ∇-dual, a conjunctive port mapping, as $(V, \overline{\mathcal{P}}, \overline{E})$ such that:*

$$\overline{\mathcal{P}} = \{\overline{r}_J,\ J \in \nabla\}$$
$$\overline{E} = \{(v, \overline{r}_J)\ s.t.\ \{r, (v, r) \in E\} \subseteq J\}$$
$$\rho(\overline{r}_J) = \sum_{r_j \in J} \rho(r_j) = |J|$$

Then, we can normalize this graph by adding weights to edges, and update the resource throughput, noted ρ^N.

$$\rho^N_{i,\overline{r}_J} = \begin{cases} 1/\rho(\overline{r}_J) & \text{if } (i, \overline{r}_J) \in \overline{E} \\ 0 & \text{else} \end{cases}$$
$$\rho^N(\overline{r}_J) = 1$$

5.1.2 Equivalence between Disjunctive and Conjunctive formulations

This section provides the main intuition to understand the equivalence between the disjunctive and the conjunctive form.

Definition 5.1.6 (Saturated port set)**.** *Consider a microkernel K. Let $(p_{i,r})_{i,r}$ be a valid assignment of K for a disjunctive port mapping $(V, \mathcal{P}, E)$. Its saturated port set $\mathcal{S}$ is defined as follows:*

$$\mathcal{S} = \left\{ r_s \text{ such that } t_{end} = \sum_{i \in K} \sigma_{K,i} \cdot p_{i,r_s} \right\}$$

That is, the set of resources r_S for which their loads $\sum_{i \in K} \sigma_{K,i} \cdot p_{i,r_s}$ correspond to $|K|/\overline{K}$, the steady state execution time of K.

Lemma 5.1.1 (Saturated set assumption)**.** *Let $(p_{i,r})_{i,r}$ be a valid assignment for a microkernel K in a disjunctive port mapping $(V, \mathcal{P}, E)$ and $\mathcal{S}$ its saturated set. Let r_s and r_t be two resources such that $(v, r_s) \in E$ and $(v, r_t) \in E$, we assume $r_s \in \mathcal{S}$ and $r_t \notin \mathcal{S}$.*

Then, either there exists a faster valid assignment for which both resources r_s and r_t are saturated, or there exists a valid assignment whose saturated set is strictly smaller than p.

A direct consequence of this lemma is:

Corollary 5.1.1 (Saturating assignment)**.** *Let us consider an optimal assignment $(p_{i,r})_{i,r}$ of a list of μOPs K on a disjunctive port mapping $(V, \mathcal{P}, E)$, such that the size of its saturated set $\mathcal{S}$ is minimal. For all $v \in V$ such that there are $(r_x, r_y) \in \mathcal{P}^2$ connected to v (i.e. $(v, r_x) \in E$ and $(v, r_y) \in E$): if $r_x \in \mathcal{S}$, then $r_y \in \mathcal{S}$.*

Thus: $\forall i \in \mathcal{I},\ [R_i(p) \subset \mathcal{S} \Leftrightarrow \{r,\ (v_i, r) \in E\} \subset \mathcal{S}]$

Theorem 5.1.1 (Equivalence of ∇-duality)**.** *Let K be a microkernel. Let $(V, \mathcal{P}, E)$ (with the set of resources $\mathcal{P} = \{r_j\}_j$), ∇ a set of subsets of $\mathcal{P}$, and $(V, \overline{\mathcal{P}}, \overline{E})$ (with the set of resources $\overline{\mathcal{P}}$ also denoted $\{\overline{r}_J\}_{J \in \nabla}$) be its ∇-dual.*

(i) Let $(p_{i,r})_{i,r}$ be a valid optimal assignment (i.e. of minimal execution time and minimal saturated set size) of K with regard to $(V, \mathcal{P}, E)$. This assignment can be translated into its ∇-dual, with no change to its execution time. In other words, $\overline{t}(K) \leq t(K)$.

(ii) If ∇ is the set of all *subsets of $\mathcal{P}$ then $\overline{t}(K) = t(K)$.*

Theorem 5.1.2 (Equivalence)**.** *Let K be a microkernel and $(V, \mathcal{P}, E)$ (with the associated throughput function t) be a disjunctive port mapping. Then, there exists ∇ a set of subsets of $\mathcal{P}$, and $(V, \overline{\mathcal{P}}, \overline{E})$ a conjunctive port mapping called the* dual *(with the associated throughput $\overline{t}$) whose set of resources $\overline{\mathcal{P}}$ is indexed by ∇ such that:*

(i) Every $(p_{i,r})_{i,r}$ optimal assignment (i.e. of minimal execution time and minimal saturated set size) of K with regard to $(V, \mathcal{P}, E)$ can be translated into $(V, \overline{\mathcal{P}}, \overline{E})$, with no change of its execution time. In other words, $\overline{t}(K) \leq t(K)$.

(ii) If ∇ is the set of all *subsets of $\mathcal{P}$ then $\overline{t}(K) = t(K)$.*

Proof. (i) Let p be an optimal valid assignment for a list of μOPs K on a disjunctive port mapping $(V, \mathcal{P}, E)$, which minimizes its saturated set of size $|\mathcal{S}|$.

From the definition of an execution time, we have:

$$\forall r \in \mathcal{P},\ \sum_{i \in K} \sigma_{K,i} \cdot p_{i,r} \leq t(K)$$

Hence, for any subset of resources $J \subset \mathcal{P}$,

$$\sum_{r \in J} \sum_{i \in K} \sigma_{K,i} \cdot p_{i,r} \leq t(K) \cdot |J|$$

Thus,

$$\forall J \subset \mathcal{P},\ \frac{\sum_{r \in J} \sum_{i \in K} \sigma_{K,i} \cdot p_{i,r}}{|J|} \leq t(K) \tag{5.1}$$

Notice that this is an equality when J is a subset of a saturated set $\mathcal{S}$ of any optimal placement $(p_{i,r})_i$. Indeed, by the definition of the saturated set (definition 5.1.6) we have that for each $r_s \in \mathcal{S}$, $t(K) = \sum_{i \in K} \sigma_{K,i} \cdot p_{i,r}$. We will now prove that $\bar{t}(K) \leq t(K)$.

Consider any combined port $\overline{r}_J \in \overline{\mathcal{P}}$ whose throughput is $\rho(\overline{r}_J) = |J|$. For any $\overline{r}_J$, by the definition of the dual:

$$\begin{aligned}\sum_{i \in K} \sigma_{K,i} \cdot \rho^N_{i,\overline{r}_J} &= \sum_{i \in K} \sigma_{K,i} \cdot \frac{\delta_{(v_i, \overline{r}_J) \in \overline{E}}}{|J|} \\ &= \sum_{i \in K} \sigma_{K,i} \cdot \frac{\delta_{\{r, (v_i, r) \in E\} \subseteq J}}{|J|}\end{aligned}$$

where $\delta_{stat} = 1$ if the statement $stat$ is true, and $\delta_{stat} = 0$ if it is false.

Now, let us show that for any assignment $(p_{i,r})_{i,r}$ in the disjunctive graph, we have:

$$\delta_{\{r, (v_i, r) \in E\} \subseteq J} \leq \sum_{r \in J} p_{i,r} \tag{5.2}$$

In order to prove this statement, we consider the two cases on the value of δ:

- If $\{r, (v_i, r) \in E\} \not\subseteq J$, then the $\delta = 0$. Because $p_{i,r}$ are positive values by definition, this inequality is trivially satisfied.
- If $\{r, (v_i, r) \in E\} \subseteq J$, then all the neighbors of r in the disjunctive graph are in J. Thus, $R_i(p) \subseteq \{r, (v_i, r) \in E\} \subseteq J$. So, $\sum_{r \in J} p_{i,r} = 1$. Notice that in this case, we have an equality.

Therefore, for any $\overline{r}_J \in \overline{\mathcal{P}}$:

$$\begin{aligned}\sum_{i\in K} \sigma_{K,i} \cdot \rho^N_{i,\overline{r}_J} &\leq \sum_{i\in K} \sigma_{K,i} \cdot \frac{\sum_{r\in J} p_{i,r}}{|J|} \\ &\leq \frac{\sum_{r\in J}\sum_{i\in K} \sigma_{K,i} \cdot p_{i,r}}{|J|}\end{aligned}$$

By using equation (5.1), we have $\overline{t}(K) \leq t(K)$.

(ii) Now, assuming that ∇ is not limited to a few subsets of $\mathcal{P}$, let us prove that $\overline{t}(K) = t(K)$.

For a given optimal assignment $(p_{i,r})_{i,r}$, let us pick $J = \mathcal{S}$, his saturated set of minimal size. Let us show that for this particular J, the previously considered inequalities are equalities.

As mentioned previously, equation (5.1) is an equality when J is a subset of the saturated set $\mathcal{S}$. Thus, we only need to show that the inequality (5.2) is an equality for this J.

Notice that for any instruction v_i, if we have an edge $(v_i, r) \in E$ when $r \in J = \mathcal{S}$, then by Corollary 5.1.1, we have $\{r, (v_i, r) \in E\} \subseteq J$. Thus, given a v_i we have two situations:

- Either there are no edge from v_i to any $r \in J$, then $\delta_{\{r,(v_i,r)\in E\}\subseteq J} = 0$, and $\sum_{r\in J} p_{i,r} = 0$. Thus, we have equality.
- Or there are an edge from v_i to a saturated resource $r \in J$. Thus, as mentioned before, $\{r, (v_i, r) \in E\} \subseteq J$ and $\delta_{\{r,(v_i,r)\in E\}\subseteq J} = 1 = \sum_{r\in J} p_{i,r}$. Thus, we also have an equality.

Therefore, the whole chain of inequality linking $\overline{t}(K)$ to $t(K)$ are equalities. Thus, $\overline{t}(K) = t(K)$. □

We have an equality if ∇ is the set of all subsets of $\mathcal{P}$, whose size is exponential in the number of resources. However, the proof shows that we can restrict ourselves to the saturated set $\mathcal{S}$ of an optimal assignment.

In practice, we build ∇ by considering the abstract resources that directly correspond to the set of resources that a given μOP can be mapped to in the disjunctive mapping. Then, we recursively apply this rule: if two abstract resources have a non-empty intersection, we then add their union as a new abstract resource. Intuitively, this new abstract resource introduces a new constraint on the valid assignment in the dual, corresponding to a potential saturation of these resources. Given the fact that CPU resources reflects design choices, the in-practice complexity of the conjunctive mapping is not translated into an exponential number of resources.

5.2 Selection of Basic Instructions

In this section, we build upon the definition of a bipartite conjunctive mapping as introduced in Definition 5.1.3 to prove the convergence of the two instruction selection algorithms, Max Clique and Min Order, to a reduced set of instructions. We show that the resulting set presents strong properties in terms of resource usage, under condition of existence of these instructions, thus satisfying crucial hypothesis for our end-to-end proof of PALMED.

5.2.1 Primary Definitions

Definition 5.2.1 (Extended bipartite conjunctive mapping)**.** *A bipartite conjunctive port mapping is equivalent to a unique extended form that decouples the use of combined resources as either a consequence of the use of simpler resources, or as the sole use of the combined resource.*

Let $(V, \mathcal{P}, E)$ be a bipartite conjunctive port mapping. Its extended form is a graph $(V, \mathcal{P}, E' \cup B)$ with B the set of back edges *defined by:*

- $(r, r') \in B \subset \mathcal{P}^2 \Leftrightarrow \forall i,\ \rho_{i,r'} \geq \rho_{i,r} \wedge \rho(r') > \rho(r)$.
  Then, $\rho_{r,r'} = 1$, and $(r, r') \in B$ is said to be a back edge.
- $\forall (v, r), (v, r, \rho'_{v,r}) \in E' \Leftrightarrow (v, r, \rho_{v,r}) \in E$ *with weight* $\rho'_{v,r} = \rho_{v,r} - \sum_{r' \neq r} \rho_{v,r} \cdot \rho_{r,r'} > 0$.
  If $\rho'_{v,r}$ is reduced to 0, the edge is not in E'.

An illustrative example is given in figure 5.1.

Definition 5.2.2 (Resource usage)**.** *Given a bipartite conjunctive port mapping and a microkernel K, we note $\rho_{K,r}$ the use of the resource r during the execution of K, i.e.*

$$\rho_{K,r} = \sum_{v_i \in K} \rho_{v_i,r}$$

Definition 5.2.3 (Load of a resource)**.** *Given the conjunctive port mapping $(V, \mathcal{P}, E)$ under its extended form and a microkernel K, we note* load(r) *the normalized use of the resource r during the execution of the microkernel of execution time t_{end}, i.e.*

$$\text{load}(r) = \frac{\sum_{v \in K} \rho_{v,r} + \sum_{r' \in \mathcal{P}} \rho_{K,r'} \rho_{r',r}}{t_{end}}$$

Definition 5.2.4 (Normalized resource mapping)**.** *The normalized version $(V, \mathcal{P}, E')$ of a conjunctive (or disjunctive) resource mapping is the semantically equivalent resource mapping $(V, \mathcal{P}, E)$ where the throughput of every resource has been normalized to 1, thus decreasing the value of the edges:*

$$w^{E'}_{v,r} = \frac{\rho^{E}_{v,r}}{\rho(r)}$$



Figure 5.1: Conjunctive resource mapping (a) and its extended form (b); both normalized

In the extended resource mapping form, the values of the back-edges become:

$$\rho^{B'}_{r,r'} = \frac{\rho^{B}_{r,r'}}{\rho(r')}$$

Lemma 5.2.1 (Bounds of the normalized back edges)**.** *On a normalized bipartite resource mapping under its extended form:*

$$0 \leq \rho_{r,r'} \leq \frac{1}{2}$$

Definition 5.2.5 (k-exclusive saturation)**.** *A microkernel S is said to be a k-exclusive saturation with $k \in [0,1]$ of a resource r when the maximum of the uses of every resource but r is bounded by k (excluding back edge coming from r), and when S never uses a resource that has a back edge with r, i.e. when S verifies*

$$\max_{\substack{r' \neq r \\ \rho_{r,r'}=0}} \rho_{S,r'} \leq \frac{1-k}{\overline{S}}$$

And

$$\forall r', \rho_{r,r'} \neq 0 \Rightarrow \rho_{S,r'} = 0$$

We call S an exclusive saturation of r when S is a 1-exclusive saturation of r, which means that S only uses the resource r. By extension, we call $i \in \mathcal{I}$ a k-exclusive saturation if the benchmark composed of the instruction alone forms a k-exclusive saturation.

Definition 5.2.6 (Basic Instructions and Covering Set of Instructions)**.** *Let $(\mathcal{I}, \mathcal{P}, E)$ be a bipartite conjunctive mapping under extended form. We call k-*basic instructions *and denote $\mathcal{I}^{(k)}_{\mathcal{B}} \subset \mathcal{I}$ the set of instructions realizing a k-exclusive saturation of any resource $r \in \mathcal{P}$.*

*If, for every resource r, an instruction realizing a k-exclusive saturation of r belongs to $\mathcal{J} \subseteq \mathcal{I}$, then $\mathcal{J}$ is said to be a k-*covering *of $\mathcal{P}$. Similarly, we call $\mathcal{J}$ a covering of $\mathcal{P}$ if $\mathcal{J}$ is a 1-covering of $\mathcal{P}$.*

5.2.2 Max Clique: Selection of Independent Instructions

Theorem 5.2.1 (Very Basic Instruction Selection)**.** *Let $(\mathcal{I}, \mathcal{P}, E)$ be a tripartite conjunctive mapping under extended form, and let $\mathcal{G} = (\mathcal{I}, E)$ be a graph with*

$$E = \{(a, b) \in \mathcal{I}^2, \overline{a^{\overline{a}} b^{\overline{b}}} = \overline{a} + \overline{b}\}$$

Let us assume that $\mathcal{I}$ is a $\frac{1}{2}$-cover of $\mathcal{P}$

Then a complete subgraph of maximum size of $\mathcal{G}$ belongs to $\mathcal{I}_{\mathcal{B}}^{(1/2)}$ and is a minimal $\frac{1}{2}$-covering set of $\{r \mid \nexists r', \rho_{r',r} \neq 0\}$.

Proof. Trivially, the set of instructions realizing a $\frac{1}{2}$-exclusive saturation of $\{r \in \mathcal{P}, \forall r', (r', r) \notin B\}$ is a complete subgraph of $\mathcal{G}$, and its cardinal is minimal.

By contradiction: Let us prove that I belonging to the complete subgraph of maximum size cannot use r such that $\rho_{I,r} > \frac{1}{2} \cdot \frac{1}{\overline{I}}$ and $\exists r', \rho_{r',r} \neq 0$. By definition, there exist at least two such resources, r_1 and r_2 with a back edge directed toward r (else r and r_1 can be merged to a single one), and no back edge directed to them. Moreover, by hypothesis, $\mathcal{I}$ is a $\frac{1}{2}$-covering set of instructions, so there exists I_1 and I_2 $\frac{1}{2}$-exclusive saturation of r_1 and r_2. Then the benchmark $I_1^{\overline{I_1}} I^{\overline{I}}$ does not verify $\overline{I_1^{\overline{I_1}} I^{\overline{I}}} = \overline{I_1} + \overline{I_2}$

Now, let us prove that $I \in \mathcal{G}_M$ cannot use several resources: let $I \in \mathcal{G}_M$ and r, r' such that $r \neq r'$ with $\nexists r'', \rho_{r'',r} \neq 0$ (respectively for r'); let us suppose that $\rho_{I,r} \neq 0$ and $\rho_{I,r'} \neq 0$. We assume that r is its limiting resource, i.e. $\rho_{I,r} = \frac{1}{\overline{I}}$. By hypothesis, $\mathcal{I}$ is a $\frac{1}{2}$-cover of $\mathcal{P}$, so $\exists I_r$, an $\frac{1}{2}$-exclusive saturation of r. By definition, $\overline{I^{\overline{I}} I_r^{\overline{I_r}}} \neq \overline{I} + \overline{I_r}$ as they share a common saturating resource r, thus $I_r \notin \mathcal{G}_M$. But, by definition, $\overline{I^{\overline{I}} I_{r'}^{\overline{I_{r'}}}} \neq \overline{I} + \overline{I_{r'}}$ as I uses r', saturating resource of $I_{r'}$, so $I_{r'} \notin \mathcal{G}_M$. But I_r and $I_{r'}$ uses fewer resources than I, so $\forall a, \overline{I^{\overline{I}} a^{\overline{a}}} = \overline{I} + \overline{a} \Rightarrow \overline{I_r^{\overline{I_r}} a^{\overline{a}}} = \overline{I_r} + \overline{a}$, which means that if I has an edge with a, then I_r also has an edge with a (reciprocally with r'). So $\mathcal{G}_M \setminus I \cup \{I_r, I_{r'}\}$ is also a complete subgraph, and its size is bigger than $\mathcal{G}$, hence the contradiction. □

5.2.3 Min Order: Selection of the Instructions using Resources of High Throughput

While the algorithm presented in Sec. 4.2.3 is efficient in practice to detect instructions mapping to resources of highest throughput (while Max Clique gathers resources of lowest throughput), we prove in this section a slightly different one, detailed in Alg. 8. As the latter relies on exact values of IPC, the non-exact one is preferred in practice because of its resilience to experimental noise (e.g. resources that do not follow exactly the bipartite model) and the freedom it grant to select more instructions than theoretically needed.

Theorem 5.2.2 (Most Greedier Instructions Selection)**.** *Let $(\mathcal{I}, \mathcal{P}, E)$ be a bipartite conjunctive mapping under normalized non-extended form. We assume that $\mathcal{I}$ covers $\mathcal{P}$, and that covering instructions verify $\rho_{i,r} = \dfrac{1}{\rho(r)}$, i.e. their usage of r is atomic.*

$\mathcal{I}_{MG} = \emptyset$;
for $i = max_I PC(\mathcal{I})$ *to 1* **do**
  $\mathcal{I}' = \{a \in \mathcal{I} \text{ such that } \overline{a} \leq i\}$;
  $\mathcal{I}_{MG} = \mathcal{I}_{MG} \cup \{a \in \mathcal{I} \text{ such that } a \in \min_{\preccurlyeq} \mathcal{I}' \text{ and } \overline{a} = i\}$;
end

Algorithm 8: Exact version of the Most Greedier instruction selection

Let $\mathcal{G}$ be the DAG associated with the order relation

$$a \preccurlyeq b \Leftrightarrow \forall p, \overline{a^{\overline{a}} p^{\overline{p}}} \geq \overline{b^{\overline{b}} p^{\overline{p}}}$$

Then, for $i \in \mathbb{N}^*$, the set of instructions of IPC equals to i that are minima of $\preccurlyeq$ over $\{a \in \mathcal{I}$ such that $\overline{a} \leq i\}$ is exactly the covering set of instructions for $\{r \in \mathcal{P}$ such that $\min_v \in \mathcal{I}(\rho_{v,r}) = \frac{1}{i}\}$ (resources of throughput equal to i).

Proof. Existence: Let r be a resource with throughput i. Then a, an instruction realizing an exclusive saturation of r exists by hypothesis. As its usage of r is $\rho_{i,r} = \frac{1}{\rho(r)}$, then its throughput is i, proving its existence.

Unicity: Let r be a resource with throughput i, and b an instruction that uses r. If b verifies $\overline{b} = i$ and b is not an exclusive saturation of r, then there exists $r_2 \neq r$ used by b. As r is the resource used by b with maximum usage, $\rho_{b,r} = \frac{1}{i}$. By hypothesis, there also exists a an exclusive saturation of r with $\overline{a} = i$, and $\rho_{a,r} = \frac{1}{i}$.

Then, a trivially verifies $\forall p, \overline{a^{\overline{a}} p^{\overline{p}}} \geq \overline{b^{\overline{b}} p^{\overline{p}}}$, as a uses r with the same amount than b and no other resource. Therefore, $a \preccurlyeq b$ so either $a = b$ or b is not a minimum for $\preccurlyeq$. Therefore, only covering instructions can be minima of $\preccurlyeq$, proving their unicity,

□

Corollary 5.2.1 (Most Greedier Instructions Selection)**.** *Alg. 8 selects exactly a covering set of instructions of $\mathcal{I}$, that is $\mathcal{I}_{\mathcal{B}}^{(1)}$.*

Therefore, the combined use of Theorem 5.2.1 and 5.2.2 outputs a $1/2$-covering of $\mathcal{I}$, one by selecting instruction mapping to instruction of low throughput, the other selecting iteratively a covering set of instructions of high throughput.

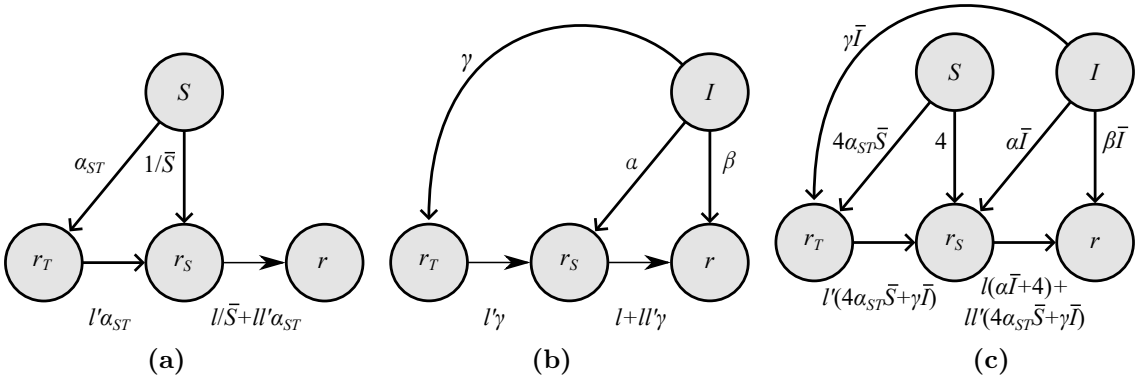


Figure 5.2: Saturating benchmarks S and instructions to analyse I: individual uses (5.2b and 5.2a), and benchmark $S^{4\cdot\overline{S}}I^{\overline{I}}$ (5.2c)

5.3 Convergence to the Complete Mapping

In this section, we build upon the previous result (selection of a set of covering instructions) to prove convergence of the LP encoded proposed in the previous chapter. In particular, we justify here the choice of $S^{4\cdot\overline{S}}I^{\overline{I}}$ as a widget allowing the characterization of instruction-to-resource edges in the final mapping.

In this section, we assume that the solver finds an edge (I, r) when we provide a microkernel containing at least once I that saturates r. This means that the solver is powerful enough to detect that an instruction which uses a resource (and its amount) as long as we provide a benchmark using the instruction limited by it.

Theorem 5.3.1 (Completeness of the output mapping)**.** *Let r, r_{ST} and r_S be three resources, and S and I two microkernels represented as a single vertex combining the use of all their instructions, forming a normalized conjunctive bipartite mapping under its extended form (see definition 5.2.1).*

Let us assume that S verifies the following properties:

- *S realize a $\frac{1}{4}$-exclusive saturation of r_S*
- *S uses another resource r_T with a coefficient α_{ST} (noted ρ_{S,r_T} with the former notation)*

Then the benchmark $S^{4\cdot\overline{S}}I^{\overline{I}}$ saturates r_S, allowing the solver to find the link $I \to r_S$.

Proof. For the sake of simplicity, we will use greek letters instead of multiple indexes of ρ in this proof:

- ℓ, ℓ' (possibly 0) are the back edge from r_S (resp. r_{ST}) to r (resp. r_S), which corresponds to $\rho_{r_S,r}$ and ρ_{r_T,r_S}, respectively
- α, β, and γ are used to denote $\rho_{I,r_{ST}}$,, ρ_{I,r_S}, and $\rho_{I,r}$.

The graph representing the resource usage for one execution of the benchmark S is represented in figure 5.2a and figure 5.2b for I. As S realizes a $\frac{1}{4}$-exclusive saturation of r_S, then $\rho_{S,r_S} = 1/\overline{S}$, so that $\overline{S}$ repetitions of S are needed to load resource r_S with the value 1.

Let us consider the benchmark $S^{4\cdot\overline{S}}I^{\overline{I}}$, illustrated in figure 5.2c. By definition of a k-exclusive saturation, S does not use r apart from the contribution from r_{ST} and r_T.

Then, the load of r_T is:

$$\text{load}(r_T) = 4\alpha_{ST}\overline{S} + \gamma\overline{I}$$

The load of r_S is:

$$\begin{aligned}\text{load}(r_S) &= \ell' \cdot \text{load}(r_T) + \alpha\overline{I} + 4 \cdot \frac{\overline{S}}{\overline{S}} \\ &= \ell' \cdot (4\alpha_{ST}\overline{S} + \gamma\overline{I}) + \alpha\overline{I} + 4 \\ &= 4\ell'\alpha_{ST}\overline{S} + \ell'\gamma\overline{I} + \alpha\overline{I} + 4\end{aligned}$$

And the load of R is:

$$\begin{aligned}\text{load}(R) &\leq \ell \cdot \text{load}(r_S) + \beta\overline{I} \\ &= 4\ell'\ell\alpha_{ST}\overline{S} + \ell'\ell\gamma\overline{I} + \ell\alpha\overline{I} + 4\ell + \beta\overline{I}\end{aligned}$$

By lemma 5.2.1, $\ell \leq \dfrac{1}{2}$ and $\ell' \leq \dfrac{1}{2}$.

Then

$$\text{load}(r) \leq 4\alpha_{ST}\frac{\overline{S}}{4} + \gamma\frac{\overline{I}}{4} + \alpha\frac{\overline{I}}{2} + \frac{4}{2} + \beta\overline{I}$$

But, by definition of the IPC, $\max(\alpha, \beta, \gamma) = \dfrac{1}{\overline{I}}$, so

$$\begin{aligned}\text{load}(r) &\leq \alpha_{ST}\overline{S} + \frac{1}{4} + \frac{1}{2} + \frac{4}{2} + \frac{1}{2} \\ &\leq \alpha_{ST}\overline{S} + \frac{13}{4}\end{aligned}$$

As S realizes a $\frac{1}{4}$-exclusive saturation of r_S, then $\alpha_{ST} \leq \frac{3}{4\cdot\overline{S}}$, so

$$\begin{aligned}\text{load}(r) &\leq \frac{3 \cdot \overline{S}}{4 \cdot \overline{S}} + \frac{13}{4} \\ &\leq 4\end{aligned}$$

Similarly,

$$\begin{aligned}\text{load}(r_T) &\leq 4\alpha_{ST}\overline{S} + \gamma\overline{I} \leq 3 + 1 \\ &\leq 4\end{aligned}$$

To obtain a lower bound on $\text{load}(r_S)$, we use similar bounds: $\alpha \geq 0$ and $\ell' \geq 0$, so

$$\begin{aligned}\text{load}(r_S) &= \ell' \cdot (4\alpha_{ST}\overline{S} + \gamma\overline{I}) + \alpha\overline{I} + 4 \\ &\geq 4 + \overline{I} \cdot (\ell'\gamma + \alpha)\end{aligned}$$

So, when r_S is used by S, either indirectly by $\gamma > 0$ and $\ell' > 0$ or directly when $\alpha > 0$, then $\text{load}(r_S) > \text{load}(r_T)$ and $\text{load}(r_S) > \text{load}(r)$. So r_S is the bottleneck, and the solver will find the edge $I \to r_S$. □

Note that this proof still stands when S and I use several resources that indirectly contribute to r_S, as it is be equivalent to a bigger value of γ.

Conclusion

Given an arbitrary three-level port mapping, we know thanks to Sec. 5.1 that there exist an equivalent conjunctive resource mapping for throughput modeling.

In Sec. 5.2, we prove that, assuming that the target ISA contains a covering set of instructions for the resources present in the microarchitecture, Max Clique and Min Order output a 1/2-covering from these instructions. Max Clique selects the minimal 1/2-covering of non-combined resources, while Min Order selects a covering set of instructions for $\mathcal{P}$. Combined, this forms an overall 1/2-covering of $\mathcal{P}$. This allows PALMED's sequence of LP to build saturating benchmarks for each resource, detecting correctly all edges of these Basic Instructions.

Then, the benchmarks described in Sec. 5.3 allow the characterization of any usage edge given a 1/4-exclusive saturation of its resource. As the 1/2-Basic Instructions are a covering set, then there exist a 1/2-exclusive saturation of it, that *a fortiori* is a 1/4-exclusive saturation. Therefore, PALMED is able to find the only corresponding disjunctive mapping of an arbitrary three-level port mapping, assuming that the ISA contains a covering set of instructions with atomic usage of its resources.

Part II

Generation of Throughput-Efficient Accelerators

Chapter 6
High-Level Synthesis

HLS designates the process of creating either FPGA or ASIC designs from a high-level language, usually C, C++ or SystemC, instead of the usual RTL specification, usually VHDL or Verilog.

Current HLS toolchains were heavily influenced by the academic project AutoPilot [54], now part of *Xilinx Vitis Development Suite* [55] since 2012 following its acquisition in 2011. Indeed, C-based HLS was shown to significantly decrease the total cost of development of dedicated accelerator designs, compared to manual RTL designs [56]. Xilinx's concurrent, Altera (part of Intel since late 2018) launched their HLS tool targeting OpenCL source code in 2013, followed two years later by a C++ front-end, forming [57].

Since then, several optimization techniques have been applied throughout the whole toolchain, concerning domains ranging from data transfer optimizations [58] to algorithm structure modifications inherited from CPU or GPU compiler designs [59], or even generation of custom architecture from more restricted DSLs [60].

This chapter presents the usual HLS workflow and the motivation of using HLS as a designing technique concurrent to the usual RTL flow in Sec. 6.1. In this section, we recapitulate the most useful HLS features for high-performance design generation: source code annotations for performance targets, resource control and communications as well as control of the design target frequency. Then, Sec. 6.2 discusses the main HLS limitations and weaknesses compared to manual RTL code, as well as possible workarounds using pure-HLS techniques. Finally, Sec. 6.3 goes through more intricate limitations for high-performance, resource-shared designs as well as their consequence in terms of design size and performance.

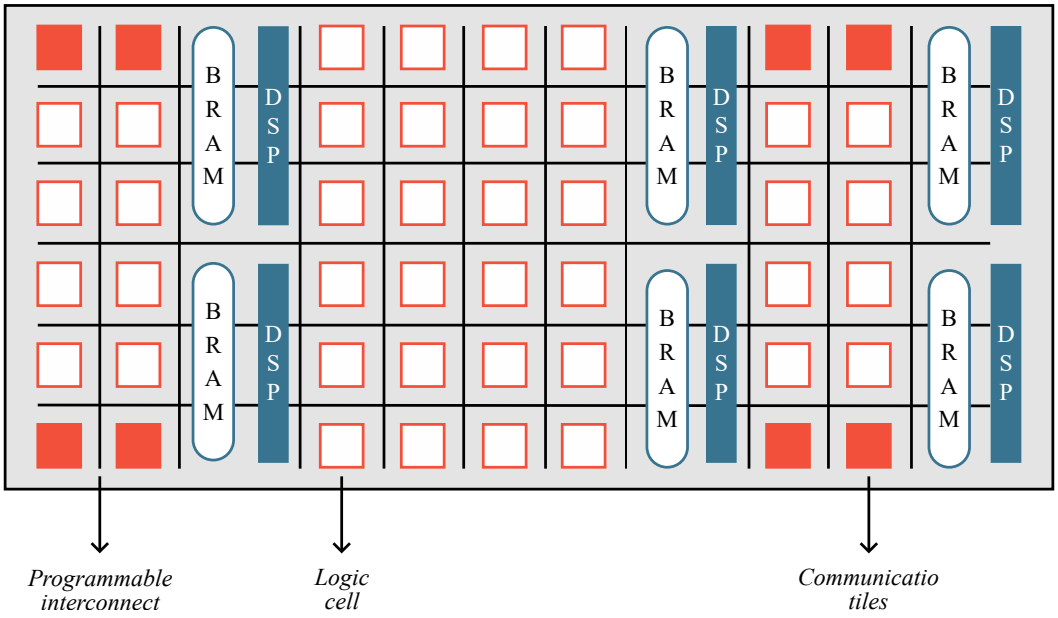


Figure 6.1: Generic architecture of an FPGA

6.1 Introduction

This section is dedicated to non-HLS experts or beginners, and covers the fundamental notions of FPGA architecture and HLS design. Basic concepts are defined in Sec. 6.1.1, then technical explanations of HLS-specific primitives are discussed in Sec. 6.1.2, with a specific focus on resource sharing in Sec. 6.1.3. Then, HLS specification of data transfers is explained in Sec. 6.1.4, as well as decisions regarding the implementation of the design's operating frequency in Sec. 6.1.5.

6.1.1 Overview

Generic FPGA architecture

A FPGA is a reconfigurable array of elements called *logic cells*, which contains several hardware primitives [61], as illustrated in Fig. 6.1:

- FF: *Flip-Flops* are the basic storage hardware primitives. One FF holds one bit of data, and has no restriction on the number of accesses per cycle apart due to routing issues.
- LUT: *Look-Up Table* are the basic logic primitives. They may be used either as a truth table for computing, as small chunks of memory (LUTRAM) or as routing resources (multiplexers).
- SLR: *Shift Registers* are another storage primitive, implemented internally using LUTs. Contrary to FFs that hold data until modification, shift registers delay the transmission of their value by a custom number of clock cycles, which is useful for synchronization of inputs between pipeline stages.

Furthermore, FPGAs are enriched with non-standard cells dedicated to specific purposes whose location is fixed on the chip:

- BRAM / URAM: *Block RAM* and *Ultra RAM* are two memory storage primitives presenting higher density compared to FFs, to the cost of a reduced number of possible accesses per clock cycle: one per *memory port*. Usually, only two ports per BRAM unit are available, which leads to *contention* when data has to be accessed in parallel.
- DSP: *Digital Signal Processing* units are programmable hardware accelerators, efficient for specific types of computation such as floating-point additions and multiplications. The usual design technique [62] for HPC accelerators aims at optimizing the DSP usage for floating-point computations as they are mostly the limiting resource on compute-limited applications.
- Communication tiles: *Tiles* that are connected to out-of-chip components. This can vary from high-performance interfaces directly connected to RAM to transceivers dedicated to out-of-board communications, as well as classical interfaces such as USB, HDMI or even LED control for debugging purposes.

The principle of FPGA design is to activate and combine some of these elementary blocks together to form bigger units such as multi-bit adders, multipliers, multiplexers and memories. These elementary design units are also hierarchically assembled to create hardware components such as execution units, buffers, scratchpads and, ultimately, a complete *design*. We designate in this manuscript by the term *FPGA architecture* the type and the layout of all units integrated in a functional design.

Due to the variety of hardware primitives available on-chip, FPGA accelerator design fundamentally differs from traditional low-level CPU tuning such as techniques seen in Part I. The goal shifts from *ensuring maximal usage of the available resources through instruction selection* to *deciding the best software-to-hardware mapping* to enforce maximum occupancy of the on-chip resources. For example, FPGA designs try to achieve a balance between memory and computation characteristics of the program to accelerate in order to limit stalls.

HLS Workflow versus Traditionnal RTL Workflow

HLS aims at producing FPGA designs from a software-inherited language, usually C/C++ language. Internally, the "high-level"[4] input code is first translated to a *Register Transfer Level* (RTL)[5] language in the *HLS synthesis* phase to produce a reusable IP (Intellectual Property [of a sub-design]) that can be used as a building block in a larger

[4]Considering C as "High Level" gives an intuition of the complexity of RTL, that HLS aims at replacing!

[5]Note that *RTL* specify the *abstraction level* of the architectural specification, whereas *HDL* (Hardware Description Language) is used to designate the *language itself*, usually Verilog or VHDL. In this manuscript, we will only use the term "RTL" for both meaning for consistency.

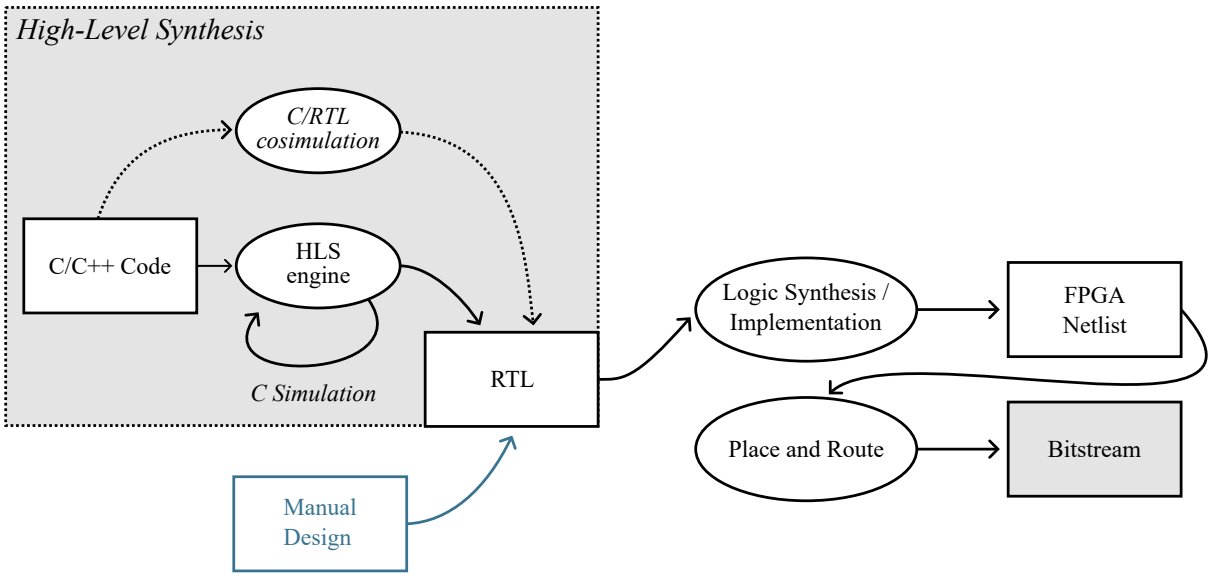


Figure 6.2: HLS complete design flow for FPGA

design. In this framework, code behavior can be checked, either with C simulation, which is the testing directly on the C program, or with co-simulation that also performs a simulation of the generated RTL. The next steps are then common with RTL-based workflow, as illustrated in Fig. 6.2. First, the designs are processed by the tool to generate a *netlist*, that is, the list of all required cells and their input/output mapping. This pass is either called *implementation* or *design synthesis* depending on the toolchain used. Then, the netlist is mapped following the specifications of an existing chip (for FPGA design) or lithographic process library (for ASICs), so that each input-output link is compliant with its corresponding constraints, in a step called *place and route* (P&R). These constraints can concern:

Placement On a FPGA, the location of small compute accelerators called *Digital Signal Processing units* (DSP) and memory storage banks called *Block RAM* (BRAMs) or *UltraRAMs* (URAMs) is fixed. Similarly, I/O pins always have a non-movable location, as well as PLLs or off-memory links: the RTL-defined sub-designs using these blocks/pins must then be placed in a way that allowing access to their physical location.

Timing The interconnect between logic blocks should be functional at the target frequency, which constrains the type of link and its maximum length on the chip.

On FPGA design, P&R results in a *bitstream*, that is, a binary file that may be flashed to the programmable logic to configure it. Note that the place and route step also gives an estimate of the slowest clocked path between two storage locations, called *critical path* (noted CP). The more complex operations are and the routing distance is between two logic blocks, the longer the critical path is. Optimizing it is a key step in accelerator

design as the maximum achievable stable frequency f_{max} is given by:

$$f_{max} = \frac{1}{\text{CP}}$$

Several techniques are known to lower CP length, such as retiming, pipelining, trimming, buffer insertion, deportation of the computation to dedicated units [5]; or adaptation of the interfaces to better suits the timing constraints. On the other hand, complex combinatorial patterns, high occupation of the FPGA and designs with high pressure on routing resources are susceptible to exhibit longer CP, and thus low operating frequency.

Advantages and Downsides

HLS shifts the description from an *architectural* one to a *semantic-based* one, that is *by design* not suited for this use. Therefore, C/C++-based HLS comes with several structural challenges [63], boiling down to the fact that one (C/C++) specification of a program can be executed on a multitude of different architectures exposing different performance/area trade-offs and different bottlenecks. This, combined with the (relative) early stage of the toolchains, is why the source code has to be annotated with `pragma` (see Sec. 6.1.2 for more details) to guide the synthesis tool in producing optimized designs. As a consequence, the output design quality is heavily dependent of the syntactic structure of the synthesized code.

Though these annotations require intricate knowledge about chip architecture – a point that HLS aims at lightening –, they also allow faster design space exploration (DSE) than traditional RTL methods. Changing the data width, replicating compute units, pipelining are typical examples of design changes that would lead to tedious and error-prone glue writing in RTL, but are very simple to express in HLS.

Another advantage of HLS lies in its simplification of the design verification check. In real-world designs, verification outcosts designing (see Fig. 6.3); HLS softens this cost by allowing direct *behavioral* specification of the architecture at the earliest stage: the source code. However, this means that most of the complexity of the verification now lies in the HLS tool, like in the software world, where both compiler and source code must be examined to ensure a validated specification. Similarly, the output RTL code consumes usually more area than manual, optimized ones due to the use of necessary generic interfaces and limitations of the source language, especially in the expression of parallel operations and cycle-accurate communications (see Sec. 6.2 for more details).

6.1.2 Annotation of the Source Code with `pragma`

For the following sections, we limit ourselves to HLS synthesis from Xilinx's Vitis 2022.1 toolchain [55]. Though most of the annotations are limited to this infrastructure, we believe that their underlying weaknesses are common to other concurrent solutions such as Intel HLS Compiler *or* Mentor Catapult*.*

Unless stated otherwise, all information concerning annotation behavior is documented in the Vitis High-Level Synthesis User Guide [64].

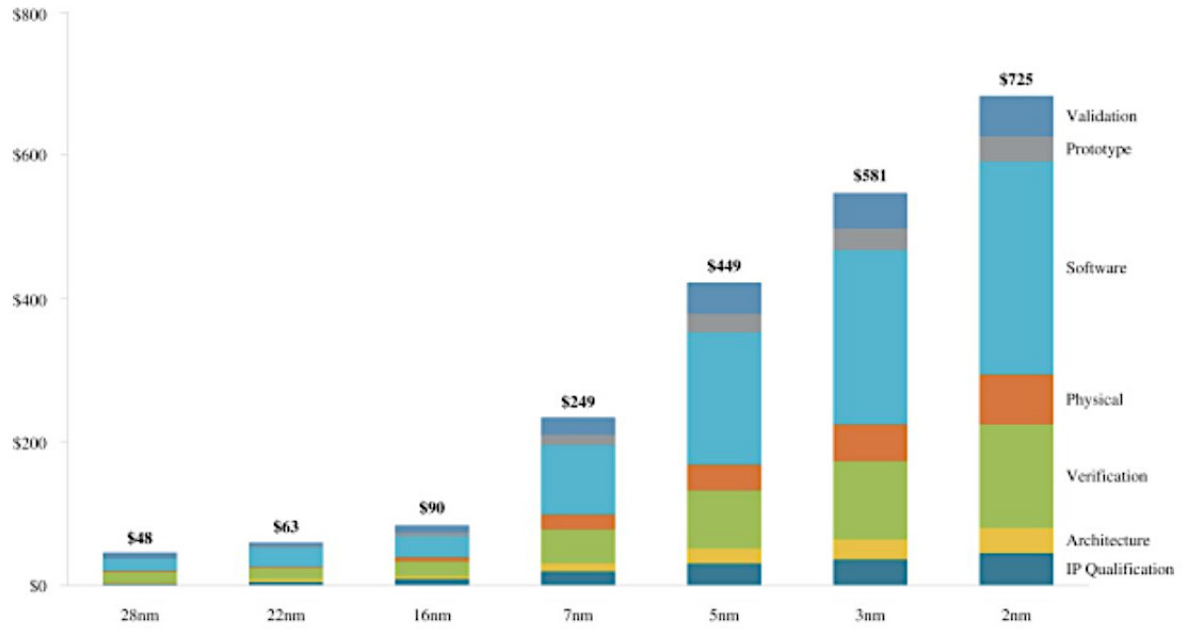


Figure 6.3: Cost of chip design for several lithographic technologies [1], in millions of dollars

Pipelining

Pipelining a design corresponds to splitting it into stages that can be executed concurrently, leading to an improved throughput with limited resource overhead (see Sec. 2.1.2 for a detailed explanation).

In the HLS context, pipelining is obtained by annotating the pipelined section with `#pragma HLS pipeline ii=xx`, with `xx` the pipeline initiation interval. The higher the initiation interval is, the lower the throughput will be, but the more resource reuse opportunities may appear. Syntactically, Vitis has two ways of splitting a main design into smaller, simpler sub-designs that are candidates to pipelining: loops and functions.

Both can use the `pipeline` pragma, which is effective for a function only when it is used in an innermost loop body. Note that the top-level module cannot be pipelined, as no fixed specification exists for interface that "loops" over a module.

Though pipelining with minimal II has become the golden rule of HLS design, as it leads to an asymptotic 100 % occupancy of the compute units (as long as the pipeline is fed and after the first initialization cycles, all stages are active at all timesteps), some works suggest that this design rule is not the most efficient in all cases in terms of throughput per area, especially in designed focused on resource reuse [62].

Stalls To control latency in cases where an input is not available, several types of pipeline can be generated via the option `style=xxx`. Possible designs are (see Fig. 6.4 for a graphical illustration):

- `stp`: *Stalled pipeline*, the default one. If an input is not available, then all stages of the pipelined are frozen, and the execution resumes when new data become available.

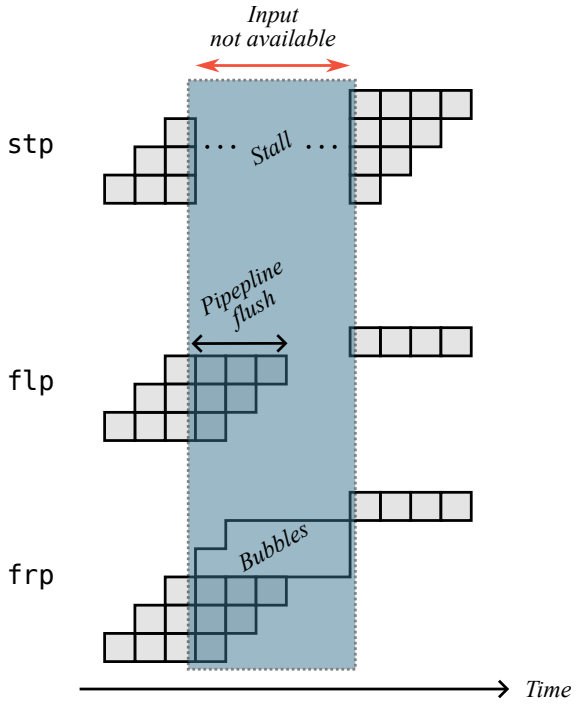


Figure 6.4: `stp`, `flp` and `frp` pipelining types

- `flp`: *Flushable pipeline*, that allows data to be processed even without input thanks to dedicated flushing logic. Flushable pipelines come with a slight resource increase due to the additional control of each pipeline stage.
- `frp`: *Free-running / flushable pipeline*, which is always running even when no input is provided by inserting a "bubble", that is, an operation whose output is discarded in the pipeline. It may only be synthesized in datataflow regions (see Sec. 6.1.2), adds buffers to the design to store outputs and requires the use a blockable I/O interface (e.g. `AXIStream`, see Sec. 6.1.4). However, it may also decrease routing resources by eliminating the need of a cross-stage interconnect, which may result in higher frequency as seen in Sec. 6.1.1.

Rewinding The `rewind` option avoids stalling when the pipelined region is encapsulated in another loop, by allowing cross outer loop iteration overlapping. This optimization, illustrated in Fig. 6.5 is especially efficient for loops with small trip counts.

Another optimization that ca be used to limit stalls is specified by `#pragma HLS loop_flatten`, that will generate control logic for perfectly or semi-perfectly nested loops nest as if only one loop was syntactically specified. This is equivalent to rewriting the code in a way that merges the loops together, without changing the scheduling of each operation. Thus, pipelines generated from flattened loops will only stall at each termination of one outer loop iteration (unless rewind), limiting resource spillage with virtually no additional logic needed.

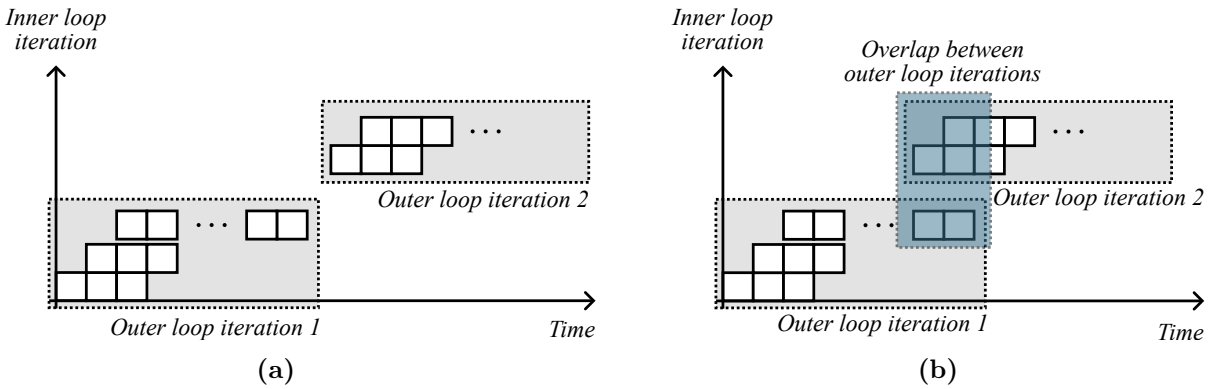


Figure 6.5: Non-rewinded (a) vs rewinded (b) loops

Unrolling

Unrolling a loop means, in HLS, coarse grain replication of the subdesign defined by the loop body, creating as many replicas as the unroll factor. The corresponding annotation is `#pragma HLS unroll factor=xxx`.

Following a toolchain-dependent dependence analysis, the iterations of the loops are distributed evenly between replicas. If the analysis concluded that iterations are parallelizable and if the tool succeeds in scheduling the loading and storing of required data in parallel, then their execution is scheduled at the same timestamp, leading to a speedup equal to the number of replicas. In the other case, the benefit of unrolling is voided by the dependencies, and replicas are scheduled one after the other, leading to no speedup, but a significant increase in resource usage.

One of the major drawbacks of this technique lies in the underperformance of the dependency analysis pass. Far from widely spread, exact techniques such as ADA [65], Vitis' dependence analysis is based on syntactic check on the variables used, thus requiring further annotations with the `#pragma HLS dependence var=xxx type=inter false` to override its conservative deductions, with `xxx` the variable where a non-existing dependence is detected.

Moreover, a high replication factor also leads to a design with complex routing path, leading to a decrease of operating frequency. This effect is essentially present on full unrolling of compute-intensive loops, for which other techniques such as pipelining or coarse-grain replication may be better suited.

Array Storage Type

By default, C-arrays are mapped by Vitis HLS to either flip-flops or BRAMs depending on their size and their number of concurrent accesses, limited by the number of read/write ports in the case of BRAMs. This behavior is configurable through the `#pragma HLS bind_storage variable=xxx type=yyy impl=zzz` with `xxx` the target array, `zzz` the physical block used to implement memory (BRAM, URAM if available, LUTRAM or SRL, with or without ECC when the target supports it) and `yyy` the memory type:

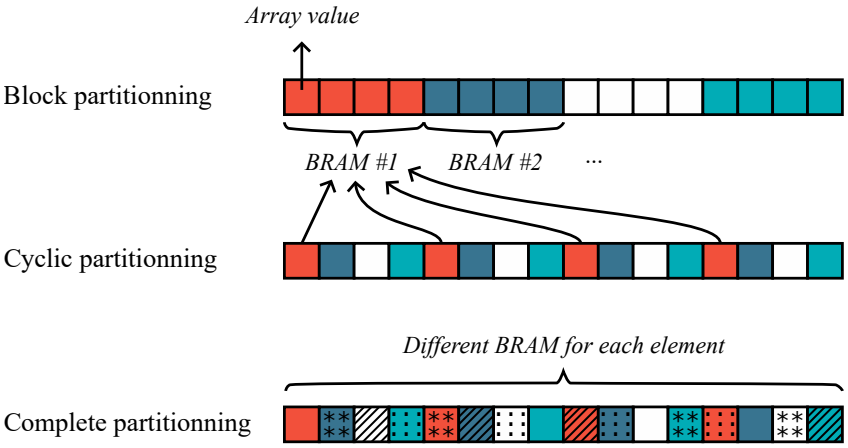


Figure 6.6: Array partition types

- `FIFO` to implement the storage as a FIFO stack (with an unlimited number of readers).
- `RAM_1P`, `RAM_1PWNR`, `RAM_2P`, `RAM_S1P` , `RAM_T2P` to implement the storage as addressable memory, with different configurations of simultaneous accesses: 1 read-/write port, 1 write port/N read ports (which leads to BRAM replications in the final hardware), 2 ports with 1 read/write and 1 read, 2 ports with 1 read and 1 write, or 2 read/write ports.
- `ROM_1P`, `ROM_2P`, `ROM_NP` addressable, read-only memory, with support for 1, 2 or N read ports.

Array Partitioning

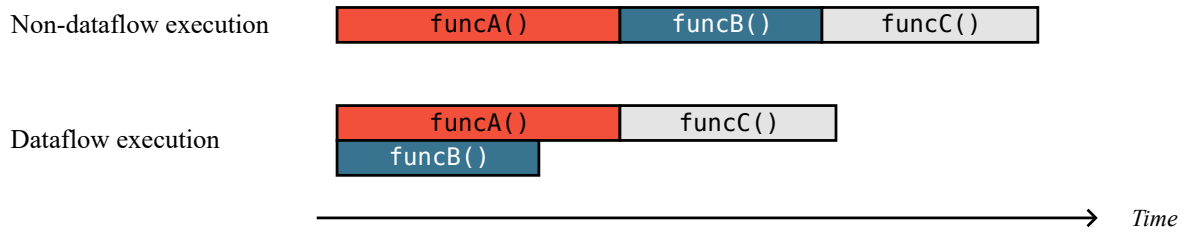
The simplest way to map a C-like array onto an FPGA is to store linearly its data in on-chip memory, following its original semantic. However, physical BRAM[6] units have a limited amount of access ports (usually two), that are used to either read or store data. Following pipelining and unrolling transformations, an array may be accessed more than twice per cycle, which means that data have to be distributed on several BRAMs to allow concurrent access, or data have to be replicated. This data distribution is called *partitioning* and is controlled by the `#pragma HLS array_partition type=xxx factor=yyy dim=zzz` that splits array in sub-arrays placed in different BRAMs, with:

- `dim` controlling the target dimension of the array
- `factor` number of sub-arrays to be stored in different BRAMs
- `type` the partitioned algorithm illustrated in Fig. 6.6, either:
  - `complete`: one BRAM is used for each value
  - `block`: blocks of size$(array)/FACTOR$ consecutive values are stored in the same BRAM

[6]All the content of this section is also valid for URAMs as well.

```
1 void top(int in1[N], int in2[N], int out[N]) {
2     int tmp1[N], tmp2[N];
3     funcA(in1, tmp1);          // read in1, write tmp1
4     funcB(in2, tmp2);          // read in2, writes tmp2
5     // dependency on tmp1 and tmp2
6     funcC(tmp1, tmp2, out);    // read tmp1 and tmp2, write out
7 }
```

(a)



(b)

Figure 6.7: Loops, dataflow and concurrent execution: code (a) and corresponding execution times with /without dataflow (b)

- `cyclic`: values are distributed on BRAM following a modulo placement, so that consecutive values are never assigned to the same BRAM

Depending of the access pattern and the pipelining/unrolling annotations, `block` or `cyclic` partitioning may result in a minimum BRAM usage.

Task-level Pipelining

One of the major drawbacks of C/C++-based HLS lies in its implicit expression of parallelism, as detailed in Sec. 6.2.1. For example, the toolchain is often not able to overlap the execution of two functions[7], especially in cases where no dependence exists between them, as illustrated in Fig. 6.7. The annotation `#pragma HLS dataflow` aims at correcting this behavior by allowing task-level concurrent execution: all functions in a pragma dataflow-annotated region are synthesized as independent sub-designs running as soon as their inputs are available. Thus, all modules are able to run in parallel, to the cost of either FIFO or ping-pong buffers to implement the storage of their input and output data. The schedule is then dynamic depending on the availability if the input, leading to both task-level parallelism (space-multiplexing of the units) and task-level pipelining (time-multiplexing of the compute units) when the I/O signals allow it. Both usages are illustrated on Fig. 6.7.

However, Vitis can only apply its dataflow optimization to code sections following a pattern allowing simple dependence analysis [66]: only single-producer / multiple-consumer is currently supported; potentially leading to code transformations increasing memory footprint.

[7]Even when applied, this behavior is currently legacy and unmaintained by Vitis HLS.

Performance

The annotation `#pragma HLS performance ti=xxx` aims at autodiscovering pipelining, inlining, array partitioning and unrolling parameters, so that a loop with known trip count achieves the given *transaction interval* (TI), that is, the number of cycles elapsed between two successive executions of the loop. It may be defined two-fold:

$$TI = \frac{II}{TC} = \frac{F}{OPS}$$

With:

- II the initiation interval of the loop
- TC its trip count
- F the frequency of the design (in Hz)
- OPS the number of executions of one loop body per second

Though being the current objective in terms of user interface, the performance annotation is still limited to code simple to analyze and uses extensively array partitioning to conservatively reach the target TI.

6.1.3 Controlling Resource Usage

Resource usage cannot be explicitly controlled, in the sense that the user cannot fix a hard limit over the number of DSP/FF/LUT/BRAM that a design can consume. However, annotations can be used to guide the synthesis tool toward a reduced usage of certain resources.

BRAM

BRAM usage can be controlled by the `HLS array_partition` pragma. Reducing the partition factor will lead to a lower BRAM usage, to the cost of lower performance due to port contention.

Logic

Similarly to BRAM usage, overall compute resources can be reduced by specifying lower unroll factors, once again to the cost of lower performances. Fine-grain resource usage can be controlled with the `#pragma HLS allocation function/operation instances=xxx limit=yyy` with `xxx` the name of the function and `yyy` the maximum number of instantiations of the component (see Fig. 6.8 for more details).

Depending on the code structure and the dependence pattern between data, the `HLS allocation` pragma may increase efficiency (measured by throughput/area) by forcing reuse of the targeted component.

```
1 void foo(half a, half b,
      half &c) {
2 #pragma HLS allocation
      operation instances=hadd
      limit=1
3      // Reuse the same
      hardware unit for both
      additions
4      c = a + b + c;
5 }
```

(a)

```
1 void foo(half a, half b,
      half &c) {
2 #pragma HLS allocation
      function instances=bar
      limit=3
3      // Instantiate 3
      different hardware units
4      bar(a);
5      bar(b);
6      bar(c);
7 }
```

(b)

Figure 6.8: Example of resource control via the `HLS allocation` pragma on operations (a) and functions (b) for `half` (FP16) data type

6.1.4 Handling Off-chip Communications

Interfaces are one of the weak points of HLS. The user is left with two choices: either using built-in interfaces, that are compatible with other off-the-shelf IPs due to their standard communication protocol[8], or implementing their own, which is more flexible but time-consuming and error-prone.

Standard Communication Protocols

The top-level function, which defines the communication interfaces with other IPs, may be annotated with the `#pragma HLS interface` [67] to specify communication protocol with other IPs (e.g. for off-chip data communication). Possible choices are:

- `AXI4`: High-Performance communication interface linking two designs in a slave-master hierarchy. Off-chip memory accessed via AXI is configured on MPSoC by memory-mapped registers, but the AXI-to-memory IP has direct access to the off-chip RAM (DMA). The AXI protocol handles handshake, bursts (high-throughput access of consecutive data by allowing multiple reads on one request), and is composed internally by 5 mandatory channels: Read Address, Read Data, Write Address, Write Data, Write Response, for a total of 45 signals.
- `AXIStream`: A simpler version of AXI, which only allows streaming of data, that is, no request can be made by the receiving end other than stopping or continuing the transfer. This is equivalent to an AXI interface handling only bursts with no specification of the target address. Compared to full-AXI, AXIStream only requires 5 mandatory signals to be implemented, hence being and ideal choice for streaming accelerators treating on-line data.
- `AXI4-Lite`: Low-performance, low-resource communication interface base on the AXI protocol. On MPSoC such as Zynq-7000, AXILite-to-memory IP directly

[8] At least *theoretically* compatible...

maps data in memory (MMIO), as the specification only allows 32 or 64 bits data accesses, corresponding to a register-like structure. As a consequence, only 19 signals are required to implement an AXI4-Lite link, while still being compatible with AXI4 targets.

Custom Communications

Input variables defined as `volatile` are generated as signals rather than being stored in a buffer at the start of the execution of an HLS-generated IP[9]. This means that the user can read or change the value of the signal by reading or writing the targeted variable. Synchronization of the signal with the clock is achieved by using the `#pragma HLS protocol` annotation: the signals separated by `ap_wait()` statements (defined in `ap_utils.h`) inside a protocol-annotated blocks are sent through the corresponding wire at the same cycle.

Another less used method consists in enclosing the state machine defining the signal in a `for` loop pipelined with an II of 1. Though not being officially documented, this technique can also be used to generate fine-grain pipelines, tackled later in this manuscript in Sec. 6.3.3.

6.1.5 Selection of the Design Frequency

The selection of the operating frequency is equivalent to setting a maximal target of the critical path. This operation is three-fold: first, in HLS, the tool must know its target CP in order to limit combinatorial logic that may lead to an invalid design, or split it if possible. After HLS synthesis, the toolchain outputs an estimate of the maximum CP. As HLS-generated designs are supposed to be functional in most cases, this estimate is often conservative, and previous work has shown that overclocking the IP still leads to valid designs [68].

The implementation step further refines the estimate, as the output netlist is optimized for a specific cell library, for which the timings of compute paths can be computed from the specifications. However, the most accurate CP estimation is obtained after P&R, as larger designs' CP may lie in the routing between two modules of the final design.

Note that, on some FPGAs such as Xilinx Zync MPSoC, clock generators are not integrated in the programmable logic, but out-of-chip. Therefore, the final bitstream does *not* configure them to the target frequency (even though the complete toolchain is aware of it!). Instead, the user has to manually configure the clock generator to the desired frequency[10], typically through memory-mapped registers setting the base clock and the multiplier. The tool `setfclk`[11], written in C, was derived from the PYNQ's

[9] This behavior may be enforced with `#pragma HLS interface mode=ap_none port=xxx register=off`, which is (surprisingly) not conflicting with annotations specifying the communication standard *on the same variable*.

[10] Which is why electronicians refer to these clocks as *constant* and not *static*.

[11] https://gitlab.inria.fr/CORSE/setfclk

Python SDK to allow easy configuration of the fabric clock for the tested FPGAs in this manuscript.

```
1 void foo(half a, half b,
      half &c) {
2     half t1, t2;
3     // Scheduled in parallel
4     t1 = a + b;
5     t2 = a * b;
6
7     // Scheduled after t1
      and t2
8     c = t1 + t2;
9
10 }
```

(a)

```
1 void foo(half a1[N], half
      a2[N], half c1[N], half
      c2[N]) {
2     // Scheduled in parallel
3     bar1(a, c1);
4     bar2(b, c2);
5 }
```

(b)

Figure 6.9: ASAP scheduling in Vitis: operators (a) and functions (b)

6.2 Generic Toolchain's Limitations

Due to the inherent complexity of hardware designs, current toolchains are limited in the feasible design space, either due to lack of primitives or lack of syntactic structure for their expression. This section covers some of these limitations with a focus on compute-limited designs: the lack of control over parallelism in Sec. 6.2.1, the lack of control over frequency domains in Sec. 6.2.2, and the inefficiencies in DSP usage in Sec. 6.2.3.

6.2.1 Parallel Operations

As discussed in 6.1.2, Vitis has a way of specifying task-level parallelism, but to the cost of increased resource usage and heavy limitations on the source code pattern. However, Vitis' scheduling policy is to generate an ASAP schedule: all operations that may be executed are executed as soon as possible, leading *de facto* to a parallel design to some extent. This scheduling choice is applied to operators, that is, statements performing computations, but also to functions[12], as illustrated in Fig. 6.9, when the dependence analysis succeeds in deducing that parallelization is legal, which boils down to aliasing of the function arguments.

6.2.2 Frequency Domains

Frequency selection is a key research topic in both CPU, GPU and FPGA/ASIC design, as it has deep consequences on the power/performance ratio of end-user chips. However, C-based HLS tools cannot handle multiple clock domains, as all C-defined operations are by definition synchronous, thus scheduled on a single, module-wide clock. Another, more programming-language view of this issue is that C lacks the expression of parallelism (as seen in Sec. 6.2.1), and thus lacks a way to express different execution speeds of the parallel sections. Nevertheless, an HLS source file may be broken down into several submodules that may be clocked differently; but the responsibility of handling

[12]However, this behavior is being considered as legacy and unmaintained.

synchronization (that is, managing glue logic in RTL) is left to the user, voiding the interest of HLS as a *semantic* specification of the design.

6.2.3 DSP Primitives

By leveling up the abstraction level from RTL to C, the user loses part of the low-level access to FPGA components. The best example of this loss lies in the DSP usage. Whereas, in RTL, DSP usage has virtually no limitation apart from their frequency limit, in HLS, the latter are limited to hardcoded primitives such as adders, multipliers, cascade operations, etc. This limits the user in multiple ways:

No User-Specified II of DSP primitives

The DSP primitives given to the user all achieve an II of 1, in the sense that they are all available for a new computation every cycle. While these designs are the most widely spread and lead to high-performance modules, their behavior restricts the possibilities of the user in terms of fine grain control of the resources. For example, an FP32 adder takes 4 DSP for a PYNQ-Z1 FPGA. There is currently no way to specify to the toolchain to generate a (slower) 1 DSP FP32 adder, even though this can lead to smaller designs, or even more efficient ones in cases where the adder is not a bottleneck of the final module.

No Cross-operator DSP reuse

As DSP primitives use a fixed, preallocated amount of DSPs, no cross-operator sharing of the DSPs is possible. For example, a design that needs N additions followed by N multiplication cannot reuse the DSPs between these two primitives, even though this is technically possible (to the cost of more control and storage logic) with RTL designs.

No Time Mutiplexing of DSPs

As discussed in Sec. 6.2.2, HLS cannot specify different clock domains for different modules. However, overclocking the DSP is a well-known technique used to increase throughput and resource efficiency of FPGA designs [69]. Note that this may technically be achieved by synthesizing separate HLS modules encapsulating DSP, targeted to the overclocked frequency, then linking these accelerators to a glue HLS module. However, this approach contradicts the spirit of HLS as the synthesis tool loses its semantic view of the execution (unknown modules and communications being treated as black boxes).

```
void add(half a, half b,
    half &c) {
#pragma HLS inline
    c = a + b;
}
```

(a)

```
void foo(half a, half b,
    half &c) {
#pragma HLS allocation
    operation
        instances=hadd
    limit=1
    // Functions are inlined:
    // Only one adder
    // is instantiated
    add(a, c1);
    add(b, c2);
}
```

(b)

Figure 6.10: Inlining (a) and operator reuse (b)

6.3 Toolchain's Limitations for Resource-Shared Design Generation

Though HLS tools are mature enough to allow fast design space exploration [70, 71], several weaknesses in design specification limit the range of possibilities of source-to-source approaches. Contrary to the former sections that provide low-level details of non-reachable designs, mostly due to the lack of dedicated annotations or primitives, this section tackles restrictions on the syntactic formulation of C codes on the angle of resource sharing. Expression oof sharing at the module scope is discussed in Sec. 6.3.1, the associated cost in resources in Sec. 6.3.2 and Sec. 6.3.3 proposes two opposed approaches for high-performance, custom accelerator specification.

6.3.1 Expressing Shared Modules

As discussed in Sec. 6.1.3, Vitis HLS has no explicit annotation for resource sharing. As a consequence, cross-function sharing is impossible without the use of `#pragma HLS inline`, that explicit copy of the function code at its call site (inlining). This way, single operator units can be shared with other surrounding operations, possibly coming from other inlined functions as well, authorizing cross-function resource sharing *syntactically*. However, *semantically*, the inlined function has lost its propriety of being an atomic reusable unit, often voiding its interest in terms of hardware design.

The combined use of the `inline` and `allocation` annotation is illustrated in Fig. 6.10. Only one FP16 adder instance will be synthesized per `foo` replica. However, calls to `add` outer `foo` body will not reuse any of the `foo` units.

Similarly, cross-loop operator reuse is not enforceable by any annotation, though the `resource` pragma on different loop nest may lead to compute unit reuse by the HLS tool. However, in any cases, the affectation of operators to units stays implicit.

6.3.2 Overhead of Operation Clustering

```
void foo(half *in, half
    *out) {
     ...
     half a, a1, b, b1, out,
    out1;
     for (int i=0; i<NB_IT;
    i++) {
#pragma HLS pipeline II=XXX
         switch
    (computation_id) {
             case 0: {
                 a = ...;
                 b = ...;
                 a1 = ...;
                 b1 = ...;
                 break;
             }
             ...
         }
         out = a + b;
         out1 = a1 + b1;
         switch
    (computation_id) {
             case 0: {
                 ... = out;
                 ... = out1;
                 break;
             }
         }
     }
     ...
}
```

(a)

```
void add1(half a, half b,
    half &c) {
#pragma HLS inline off
    c = a + b;
}

void add2(half a, half b,
    half &c) {
#pragma HLS inline off
    c = a + b;
}

void foo(half *in, half
    *out) {
#pragma HLS allocation
    function
        instances=add1
    limit=1
#pragma HLS allocation
    function
        instances=add2
    limit=1
#pragma HLS pipeline II=XXX

    ...
    add1(a, b, c);
    add2(a1, b1, c1);
    add1(a2, b2, c2);
    ...
}
```

(b)

Figure 6.11: Single-operation (a) and function-based (b) explicit resource binding

To allow explicit compute unit binding, the programmer has two efficient choices:

- Syntactically relying on one operation, and rely on `switch` to route the input / outputs to the rest of the program (Fig 6.11a)
- Cluster operations into functions, limit the number of instances of each function to 1 and reuse the functions as much as needed (Fig 6.11b).

While the first case is constraining the syntactic structure of the program by forcing a main loop structure iterating over all usages of the operator, the second one is more resource-hungry due to the additional interfaces needed by the (non-inlined) functions. However, more complex sequences of operations can be shared through function-level replication, which is impossible in the other case due to the `instance` annotation only applicable to toolchain-specific operations.

6.3.3 Fine-grain Execution Pipeline Generation

In the case of pipelined IPs, that is, HLS-crafted IP in which a custom pipeline is needed with control over the resource used and its iteration interval, the programmer is left with two solutions depending on the expression of the shared operations seen in Sec. 6.3.2.

Explicit Scheduling If the approach of single-operation reuse is taken, the pipeline is composed of a single loop with user-defined iteration intervals as illustrated in Fig. 6.11a, with `switch` statements selecting input and output of the operators depending on the *user-defined* schedule. While this approach is the most flexible in terms of control of the schedule/placement of the operations, the lack of intricate dependence analysis of the HLS tool may detect memory conflicts between iterations of the loop, and then relax the II to meet its conservative constraints. As a consequence, a module capable of sustaining the desired memory access workload is required to bypass this analysis.

Implicit Scheduling If the approach of function-level reuse is taken, then the programmer may schedule explicitly each call, or opt for an implicit scheduling. The latter is expressed in Fig. 6.11b, with the subtlety of requiring a function-level pipelining annotation (even with a target latency higher than the module's) to activate Vitis' HLS resource sharing optimization. In this case, inner pipeline computations may be reordered: as a consequence, the overall latency may be higher than the explicit scheduling's, as the synthesis tool can interpolate bubbles (idle cycles) in the compute unit's schedule to ensure consistency of the dependency analysis.

6.3.4 Accurate Measuring of the Execution Time

The most accurate way to measure time in HLS relies in the use of an on-chip counter, that is, an external IP placed in the usual RTL hardware block design editor, which value is polled by the HLS IP before and after the measured computation. However, due to dead code elimination by the HLS tool, not all HLS-based computations of the execution time will result in a correct value. For example, the code illustrated in Fig. 6.12a will result in a `nb_cycles` of always zero as the compiler can reorder statements and execute both `counter` reads in the same cycle.

Instead, one way to avoid undesired simplification of the cycle measurements is both the usage of the `#pragma HLS protocol` that specifies the order in which operations are scheduled (as seen in Sec. 6.1.4) and the direct transfer of the values of the counter instead of performing the subtraction (potentially simplified) on-chip. In this case, the programmer must place `ap_wait()` calls around the measuring statement to correctly specify the scheduling of the `protocol` section. The corresponding code is reported Fig. 6.12b.

```
1 void foo(...,
2     volatile unsigned
    &counter,
3     unsigned &nb_cycles) {
4     unsigned before =
    counter;
5     ... // compute
6     nb_cycles = counter -
    before;
7 }
```

(a)

```
1 void foo(...,
2     volatile unsigned
    &counter,
3     unsigned
    &nb_cycles_before,
4     unsigned
    &nb_cycles_after) {
5 # pragma HLS INTERFACE
    mode=s_axilite
    port=nb_cycles_before
6 # pragma HLS INTERFACE
    mode=ap_none
    port=nb_cycles_before
    register
7 # pragma HLS INTERFACE
    mode=s_axilite
    port=nb_cycles_after
8 # pragma HLS INTERFACE
    mode=ap_none
    port=nb_cycles_after
    register
9 # pragma HLS INTERFACE
    mode=ap_none port=counter
    register=off
10     start: {
11 #pragma HLS protocol
    mode=fixed
12         ap_wait();
13         nb_cycles_before =
    counter;
14         ap_wait();
15     }
16     ... // compute
17     end: {
18 #pragma HLS protocol
    mode=fixed
19         ap_wait();
20         nb_cycles_after =
    counter;
21         ap_wait();
22     }
23 }
```

(b)

Figure 6.12: Erroneous (a) and correct (b) HLS execution time measurement using an on-chip counter

Chapter 7
Towards a General Formulation of the Resource Sharing Problem

Former limitations of HLS described in Sec. 6.2 concern mainly back-end, that is, families of designs that the synthesis tool cannot generate due to the lack of standard specification for their use in HLS C/C++ (Sec. 6.2.1), or its lacks of support for specific primitives (Sec. 6.2.3), or However, even given these restrictions on the design space, current tools are far from implementing automatically Paretto-optimal designs for a fixed latency/throughput or resource budget.

This section proposes an approach to remedy this issue under the resource sharing aspect, that is, reuse of functional units throughout the program to ensure their high occupancy. This chapter paves the way for an automatized framework that, given in input i) a C specification of a program with statically analyzable control flow and ii) an area budget, generates both an architecture capable of executing the program and a mapping from the program's operations to the architecture compute units. First, Sec. 7.1 overviews former approaches. Then, Sec. 7.3 and 7.4 tackles the issue of resource and latency estimates of the compute units, needed for the tool to estimate total resource usage. Sec. 7.5 proposes a naive convex encoding of the resource sharing problem; and Sec. 7.6 discusses the limitations of this approach as well as possible heuristics for real-life, fast generation of non-provably optimal accelerator designs.

7.1 Existing Resource Sharing Techniques

Resource sharing techniques on FPGAs are by no means new. The idea of reducing the size of designs by time-multiplexing their execution units was already around when the first FPGAs were commercialized [72], and has evolved in par with High-Level Synthesis techniques [73, 74] that started from ad-hoc languages to the modern subset of C/C++ [55, 57, 75]. Classical techniques rely on optimization of loop structures [76] in order to generate efficient pipelines through sharing of code-derived execution units, and remain the current direction of research today [59, 77]. Another direction concerns synthesis tools, where the granularity of shared element is smaller [78, 79], often limited to a few DSPs or load/store units, as detailed in Sec. 6.1.1.

This section analyses two resource sharing approaches in a bottom-up way: first, sharing at Basic Block (BB) level is studied in Sec. 7.1.1, based on the research work from Josipovic et al. [78]. Then, Sec. 7.1.2 details the technique used by Li et al. [59] to detect shareable components across loops on statically scheduled circuits, and thus to generate efficient designs.

7.1.1 Basic Blocks-Level Resource Sharing

As current HLS tools operate from high-level C/C++ source code, scheduling information must be automatically inferred during synthesis while at the same time keeping track of resources. Therefore, the tool has chose which compromise to implement between replication of units, that performs better but uses more area; and sharing, that involves more complex routing – while ensuring correctness of the generated circuit. To simplify the problem, some tools [55, 80] rely on *static* scheduling: the timestamp of execution of each operation is fixed. However, in the case of *dynamic* scheduling, also referred as *dataflow* circuits, timestamp of execution of operations is not decided at compile time, hence a more challenging task as wrong scheduling decisions may lead to deadlocks, that is, indefinite waiting before the execution of some operations.

The approach taken by Josipovic et al. [78], described in this sub-section, is integrated in the Dynamatic HLS tool [75] and targets sharing at the granularity of dynamically-scheduled operations on high-performance sections, defined as loops in the original program. It relies on a occupancy metric in order to quantify the shareable operations in dataflow circuits and uses synchronizing components to assign them to pipelined execution units without possibility of deadlocks. The main idea of this technique is detailed in the next section.

Compared to Xilinx's commercial tool, this approach uses exactly as many DSPs when evaluated on a 10 synthetic benchmarks. However, it requires between 3.56 and 7.02 times more routing resources (LUTs) and its execution time/critical path also suffers from non-negligible overhead due to dynamic scheduling that is not profitable in all applications.

```
for (i = 0; i < N; i++) {
    a[i*x] = i*y;
}
```

Figure 7.1: Example of design exposing shareable opportunities

Technical Solution

Motivating Example Let us illustrate the non-triviality of legal resource sharing in dataflow circuit with a simple example: the synthesis of an accelerator respecting the specification expressed in Fig. 7.1.

Assuming the data type of `i`, `x` and `y` are the same, then the multiplier operation can be shared using two multiplexers to select the corresponding `x` or `y` value.

However, this design fails to provide a deadlock-free implementation of the specification no matter the input order of operands – a requirement for safe dataflow circuits –, as starvation may occur if two $\mathtt{y}_1, \mathtt{y}_2$ values corresponding to two different executions arrives consecutively. In that case, the storing stage will wait for the result of $\mathtt{i} * \mathtt{x}_1$ only to find $\mathtt{i} * \mathtt{y}_2$, hence an incorrect execution.

Resource Sharing in the Absence of Input Ordering The principle of resource sharing is to route data from multiple possible input paths, treat it, then route the corresponding result to the correct next unit. Therefore, the difficulty when designing resource shared circuits in the dataflow paradigm is to avoid starvation by ensuring correct synchronization of inputs. As deadlocks can emerge from situations in which the arrival ordering of the inputs does not follow the one deduced from the control flow of the program, the safest way to ensure coherency of the execution is to limit the ordering of the inputs and operations to the one specified by the control flow. This can be enforced by an in-order implementation of the sequence of operations, for example through a dataless token that orchestrates the data path: on FPGA architectures, this translates into buffers that store the intermediate results, *forks* that distribute intermediate values and *joins* for the stalling logic. Such glue guarding elements can be naively implemented before each shared operations to provide maximal safety to the cost of higher resource usage, degrading both latency and throughput.

One solution proposed to mitigate this overhead is to implement the ordering of the input at the *basic block* level, that is, in regions delimited by two branches. Note that further application-specific optimizations can also be realized when the control flow of preceding units allows the derivation of timing-related rules, but this is not the case in general.

In order to maximize the range of applicability of this technique, Josipovic et al. have proposed a criteria of profitability based on the occupancy of units to decided whether its operations are worth sharing: the sum of the token occupancy of a pair of operation must lower or equal than the total unit latency.

Limitations

First, this approach is limited to operation-level sharing, similarly to [59] and to the approach presented in this chapter. On the other hand, overlay-based techniques such as [81] and the Generic Accelerator, described in Chap. 8, leverage the on-the-fly configuration of Xilinx' DSP48E1 units to select at runtime which operation to execute (amongst a fixed, supported set). Supporting such kind of flexible execution units would allow more aggressive operator sharing, reducing DSP usage on low-throughput workloads to the cost of additional routing and synchronization primitives – as well as latency.

Moreover, the decomposition of dataflow circuits into basic blocks forbids sharing across them *by design*. As this approach focuses on throughput per area in a steady execution state, this decision is designed not to harm performances, but this is often not the case in practice. Once again, more aggressive sharing can be achieve by breaking basic blocks boundaries, which would also increase the complexity of routing resources as well as requiring associated multiplexing components.

Finally, this approach proposes an on/off optimization framework, without decision margin from the designer. However, real-life IPs are often required to be customizable in terms of size and performance in order to globally balance each individual component of a VLSI design. Contrarily, the Linear Programming formulation, advocated in this section, aims at formalizing a globally optimal solution to the resource sharing problem *under area budget*, similarly to [59] where the optimization target accounts for the available on-chip area.

7.1.2 Loop-based Resource Sharing for Throughput-based Optimization

While the approach in 7.1.1 is dedicated to loop-level optimization in order to generate efficient designs for *dynamically scheduled* applications, it cannot perform *cross-loop* resource sharing, nor does it focus on globally optimal architectures.

Indeed, designs can be optimized for *global throughput* instead of latency: the relevant metric is then the average number of independent executions of the workload per unit of time. In this context, a *local* optima in terms of throughput-per-area, i.e. one design with best throughput for *one* execution of the application, may not be the best *globally*, i.e. when dealing with batched repetitions.

This observation was made by Li et al. [59], who proposed an approach based on per-loop resource vector usage (detailed in the next section) in order to determine components worth sharing, and achieve a globally optimal design. Compared to the locally-optimized version on 8 double-precision synthetic benchmarks, this approach reaches a mean speedup of 31 %, demonstrating experimentally its interest.

Motivating Example

In applications presented as a sequence of loop nests (e.g. stencils), operator imbalance across inner loop nests can create idling situations for compute units, thus leaving

opportunities for resource sharing. Let us illustrate with the skeleton of DWT as an example, illustrated in Fig. 7.2a. It is composed of 4 loop nests: 2 with twice many additions as multiplications (`L1`, `L2`), and 2 with as many additions as multiplications (`L3`, `L4`).

Let us compare the performance of the following hardware accelerator topologies:

1. Fully Pipelined: 2 adders, 2 multipliers
2. Non Fully-Pipelined: 2 adders, 1 multiplier

While loops `L1` and `L2` are both executable with an II of 1 cycle on both architecture, `L3` and `L4` must be executed with an II of at least 2 cycles on (2) due to the presence of only one multiplier. Contrarily, the architecture (1) still achieves an II of 1 on `L3` and `L4`.

The usual designing technique consist in maximizing local throughput, hence selecting the architecture (1), then performing coarse-grain replication in order to maximize utilization of the targeted FPGA accelerator. However, a 64-bit floating point implementation of (1) uses 28 DSPs, while (2) only requires 17. When looking at the performance on a 512x512 image size, (1) takes 59 kilo-cycles, whereas (2) takes 85 kilo-cycles. We can then compute the efficiency of each design, defined as the *throughput per DSP* (or inverse latency per DSP, in this context): 6.1 10^{-7} $\text{cycle}^{-1}.\text{DSP}^{-1}$ for (1), whereas (2) reaches 6.9 10^{-7} $\text{cycle}^{-1}.\text{DSP}^{-1}$. Therefore, *the non-fully pipelined implementation achieves proportionally more performance per area that the fully pipelined version.*

When measuring only throughput, this means that the architecture (2) is preferable, as more replicas can be instantiated than (1) for the same chip area usage, leading to a *global* gain in terms of throughput.

Technical Solution

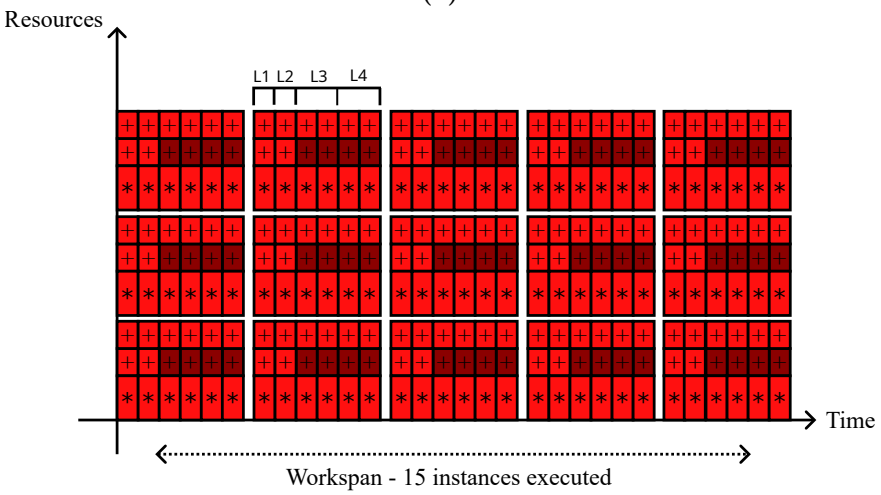
This counter-intuitive remark can be generalized and formalized; in that regard, Li et al. define the *Loop Shareable Load Vector*: the vector whose coordinates denote the number of operations of each type that take more resource than a 32-bit 8-to-1 multiplexer[13] and a fixed, per-operation additive resource model. The objective is then to find the *Loop Resource Allocation Vector* corresponding to the repartition of compute units in the design space which maximizes the global performance (expressed as $\#replica * perf_per_replica$).

However, external constraints complicate the formalization of the loop-based, throughput focused resource sharing problem. First, the estimated performance of the design has to take dependencies into account without over-approximating, and secondly the overall resource usage must fit within the FPGA's hardware limitations. The best solution is then deduced either by manual enumeration of all valid values of the resource allocation vectore, or using an Integer Linear Programming solver such as [48], thanks to the proposed formulation.

[13]This condition ensures that the cost of the interconnect, i.e. multiplexers inserted at the inputs of the units, remains negligible compared to the gains provided through resource sharing

```
L1: for i = 1 to M-3 step 2
    // two additions, one multiplications
L2: for i = 2 to M-1 step 2
    // two additions, one multiplications
L3: for i = 1 to M-3 step 2
    // two additions, two multiplications
L4: for i = 2 to M-1 step 2
    // two additions, two multiplications
```

(a)



(b)



(c)

Figure 7.2: Squeletton code (a), fully pipelined (c) and non-fully pipelined (b) accelerators for DWT. Red squares indicated compute units, bright if occupied, dark if idle. Despite being slower in terms of latency, the non-fully pipelined version achieves more problems solved (15) than the fully pipelined one (12) with a similar resource and time budget.

Limitations

While this approach tackles a counter-intuitive result, its scope is not as wide as what could be expected. Indeed, while the targeted optimization is the global throughput, the metric used to quantify it is the average performance, computed as $\#replica * perf_per_replica$. This corresponds to batched execution of the targeted workload, i.e.

repeated solving of independent instances of each problem. However, the proposed solution does not guarantee an occupancy of 100 % of the integrated compute units due to inherent operator imbalance in the loops of the tested benchmarks. Theoretically, there could exist a solution achieving better throughput, if all units were to be used at each timestep during a steady state execution profile. Such designs exist in practices thanks to *Coarse-Grain Pipelining* (CGP), resulting in a bigger design with higher latency, but superior performance per area. The GA detailed in Sec. 8 is entirely built around the idea of CGP by allowing parallel execution of kernels (seen as subparts of programs) that offers optimal occupancy on batched workloads when resource requirements are met.

Nevertheless, as the coarse-grained pipelined version of the accelerators detailed in Li's work can be significantly bigger, this also means that pure replication of this design may not achieve maximal performances in extremely resource-constrained conditions. In particular, cases where only one instance of a CGP design may fit on-chip can still benefit from the approach described in this subsection. Indeed, it produces smaller designs, which can maximize *chip* occupancy by leaving fewer resources unused after maximal replication. As the latter are physically integrated on-chip on FPGAs, there is little interest in letting them out of the design.

Another weakness of this approach lies in the fact that achieving a reduction of the design surface without hurting the performance to the same extent requires major resource usage difference between operators. In the current Vitis HLS toolchain, this is only valid for double precision operations, where an adder uses 3 DSP and a multiplier 11. On lower bitwidth, both units use the same amount of DSP, voiding any interest to local slowdowns.

7.2 Compute Unit: Definition

In this chapter, a *Compute Unit* (CU) is defined as an atomic subdesign of an accelerator, that is able to compute only a fixed graph of operations from register-like inputs. A CU is entirely defined by:

1. The list of its (scalar) inputs / outputs
2. The data type of its inputs / outputs
3. The compute graph (CDAG) of its output
4. Its II, that is, its peak throughput

Whereas points 1, 2 and 3 describe the functional behavior of the CU, point 4 is heavily linked to its resource usage. Indeed, as seen in Sec. 2.1.2 fully pipelined CUs exhibit no reuse of single-operator units as all of them are already fully time-shared. On the other hand, CUs with an II strictly greater than 1 allow inter-CU reuse of those units, which may seem to increase efficiency. While this fact may be true for some designs, intra-CU reuse can lead to under-utilization of its units, voiding from its conception the target of high occupancy. Moreover, reusing single-operators units implies that the CDAG of the CU is composed of several operations, thus specializing the CU to a specific pattern. This leads to less potential call sites in the target program, thus translating into a potentially a lesser efficient design when assembled in the complete accelerator.

Example Let us consider the accelerator architectures and the program described in Fig. 7.3. Pre-P&R resource, latency and throughput measurements of both architecture are reported in Table 7.1 for a Xilinx ZCU104 development board.

Though exploiting reuse of the "+" operator directly in the CU thanks to its II of 2, the design specified in Fig. 7.3b achieves strictly worse throughput-per-area than the design composed only of single-operation CU (Fig. 7.3a). This is due to the non-reuse of the multiplier in `cu_2add_1mul` across the program, leading to the need of another instance of a multiplier (in `cu_mul`). On the other hand, latency was improved because of the reduction of the interconnect between CUs.

In other words, using resource sharing directly in non operator-balanced CUs limits its applicability to specific programs and performance targets where the loss of occupancy at the single-operator level does not matter. This is due to the fact that, as seen in Sec. 6.2.3, the current toolchain is not able to generate primitives for single-operator CU with an II higher than 1: it will instead use the one with an II of 1, leading to no resource reduction or cross-operator DSP sharing, contrarily to what a non-expert user could expect.

In the following sections, we limit ourselves to single-datatype CUs, but we believe that only minimal changes are needed in the resource and latency estimates to adapt this work for type-heterogeneous CUs.

```
void cu_add(half i1, half
    i2, half &o1) {
#pragma HLS pipeline ii=1
    o1 = i1 + i2;
}
```

```
void cu_mul(half i1, half
    i2, half &o1) {
#pragma HLS pipeline ii=1
    o1 = i1 * i2;
}
```

(a)

```
void cu_2add_1mul(half
    i1, half i2, half i3,
    half i4, half &o) {
#pragma HLS pipeline ii=2
    o1 = (i1+i2) *
    (i3+i4);
}
```

```
void cu_mul(half i1, half
    i2, half &o1) {
#pragma HLS pipeline ii=2
    o1 = i1*i2;
}
```

(b)

```
void foo(half i1, half i2, half i3, half i4, half &o1) {
    half tmp;
    tmp = i3*i3;
    &o = (i1+i2) * (tmp+i4);
}
```

(c)

Figure 7.3: Two FPGA accelerator architecture, using fully pipelined (a) and non-fully pipelined (b) CUs, for the program (c)

Metric	LUT	FF	DSP	Throughput	Latency
Archi (a)	195	167	4	0.5 pb/cycle	8 cycles/pb
Archi (b)	220	227	6	0.5 pb/cycle	7 cycles/pb

Table 7.1: Resource, latency and throughput estimates of two architectures for an accelerator executing 10 successive instances of the program from Fig. 7.3c

7.3 Resource Estimation of Compute Units

Our estimation of the resources taken by the CU considers the HLS toolchain as a black-box and follows a physicist-like approach. Starting from plausible terms, we infer for each operator their resource consumption behavior for both parallel and sequential repetitions thanks to microbenchmarking, similar in its spirit to the work presented in Part. I. Models for each resource are detailed in the following sections: DSP in 7.3.1, LUT in Sec. 7.3.2, FF and SRL in Sec. 7.3.3. Then, all the resource models are calibrated and evaluated in Sec. 7.3.4. Finally, we combine both estimators into a final one that operates on arbitrary CDAG in Sec. 7.3.5, and evaluate its accuracy in Sec. 7.3.6.

7.3.1 DSP Estimation

We propose the following model for the estimation of the number of DSPs inside a CU:

$$N_{DSP} = K_{DSP}^{op} \cdot \left\lceil \frac{n}{II} \right\rceil$$

With:

- N_{DSP} the number of DSP
- K_{DSP}^{op} a constant specific to the operation type (such as "addition/FP32")
- n the number of operations in the CU (either fully parallel or fully sequential)
- II the initiation interval (i.e. max throughput) of the CU

As floating-points operations are by default mapped to DSPs due to their efficiency compared to LUT-based computation, and as DSPs are only computation units (nor storage nor routing), their estimate is only composed of one term. $\left\lceil \frac{n}{II} \right\rceil$ corresponds to the minimal number of units performing the operation to ensure a throughput of one computation every ii cycles, which is multiplied by the number of DSP per operator K_{DSP}. In practice, such a simple model is sufficient in practice for a perfect DSP estimator (see Sec. 7.3.5) on our test CUs, because the HLS tool failed to exploit sharing opportunities at the DSP level, as described in Sec. 6.2.3.

7.3.2 LUT Estimation

We propose the following model for the estimation of the number of LUT inside a CU:

$$N_{LUT} = K_{LUT} \cdot bitwidth \cdot \left\lceil \frac{n}{II} \right\rceil + nb_in \cdot bitwidth \cdot \left(K_{LUT}^{'} \cdot II - \underbrace{K_{LUT}^{''} \cdot (II-2)}_{\text{if } II>2} \right)$$

With (keeping the same notations as the previous sections):

- $bitwidth$ the number of bits used to represent input data
- nb_in the number of inputs of the CU

- K_{LUT}, K'_{LUT} and K''_{LUT} three positive constants specific to the operation type

As LUTs represent both compute and routing components, we could have expect their number to be quadratic with respect to the size of the inputs, as lower-weight bits may interact with higher-weight one, typically on multipliers. While this is true for integer-based computation, on floating-point ones, the computations (and these non-linear interactions) are handled by the DSPs. As a consequence, LUTs are only used as routing components, thus scaling linearly with the number of single-operator units in the CU.

Qualitatively,

- $K_{LUT} \cdot bitwidth \left\lceil \frac{n}{II} \right\rceil$ represents the number of LUTs used by the actual single-operation units, i.e. DSPs
- $nb_in \cdot bitwidth \cdot K'_{LUT} \cdot II$ represents the additional LUTs needed to route a slower pipeline
- $nb_in \cdot bitwidth \cdot K''_{LUT} \cdot (II - 2)$ is a correction term expressing LUT-based resource sharing on pipelines with a high initiation interval

7.3.3 Storage Units Estimation

SRL

We propose the following model for the estimation of the number of SLRs inside a CU:

$$N_{SRL} = \max\left(0, \begin{cases} bitwidth \cdot \mathbb{C}_{SRL}(nb_ops, II) & \text{if } \mathbb{C}'(nb_ops, II) > 0 \\ 0 & \text{else} \end{cases}\right)$$

With (keeping the same notations as the previous sections):

- N_{SRL} the number of SRL
- $\mathbb{C}$ and $\mathbb{C}'$ linear combinations of nb_ops and II with integer coefficients

Experimentally, SLRs are mostly used in CUs with a low initiation interval, due to the synthesis tool's inner cost model that prefers SLR for high-throughput units in order to synchronize input arrival for computation on pipepline stages that are not the first operations. This may be caused by the fact that SRLs do not require extra routing logic to select their value depending on some inner state of the CU (typically representing the state of the execution pipeline for CUs that have an II higher than 1). As no clear behavior emerged from the toolchain estimate, we opted for an ad-hoc linear model with a cut-off, which experimentally provides satisfactory accuracy, as evauated in Sec. 7.3.6.

FF

FF number is estimated by (reusing the previous notations, with N_{FF} and K_{FF} the number of FFs and the FF-specific coefficients):

$$N_{FF} = K_{FF} \cdot bitwidth \cdot \left\lceil \frac{n}{II} \right\rceil + nb_in \cdot bitwidth \cdot \left(K'_{FF} \cdot II - \underbrace{K''_{FF} \cdot (II - 2)}_{\text{if } II>2} \right) - N_{SRL}$$

In the worst case, storage units are needed at each step of the compute pipeline (i.e. before and after each operator-specific combination of DSP), which is why no quadratic term appears in the estimator. Therefore, the idea behind each term is exactly the same as the one for LUTs: FF are either linked to the number of single-operation units, or due to pipelining, or saved by resource sharing on high-II pipelines. As FF and SRL are both storage units, our generic estimator formula (used for LUTs) estimates their sum. Then, as the synthesis of SRL follows a non-generic scaling, we estimate them separately with a the former's section formula, then subtracts them from our FF estimator in order to keep their sum unchanged.

7.3.4 Microbenchmarking CDAGs

The calibration of the estimator is done by manually exploring the space of possible variables and finding, for each operator and each combination (sequential or parallel), the values minimizing the mean average error of the prediction versus the post-P&R value over CUs composed of 1 to 8 repetitions of a unique operator, while keeping a meaning from a high-level point of view. Technically, this mean that the constants must be under the form $\frac{a}{2^b}$ with b lower than 4. Predictor Mean Average Error rates are reported in Table 7.2a for half-precision adders and in Table 7.2b for half-precision multipliers, corresponding to aggregated results of the measures presented in Fig. 7.4.

MAE of SRL prediction is always zero for parallel composition of operations, as the toochain never instantiates any SRL for these tasks. Indeed, operations are fully parallel, sp there is no need of input synchronization across pipeline stages. On reductions, SRL predictions match exactly the resource usage for CUs with II of 1 and 2, but failed to detect any SRLs on designs with II of 3 and 4, due to the cut-off (condition using $\mathbb{C}'$). This can be solved by completing the predictor with higher-order terms, but we have chosen not to implement them to avoid overfitting.

7.3.5 Combination of Sequential and Parallel Models

As our estimator is able to predict with reasonable accuracy the area of both sequential and parallel CDAGs, we propose a combination of both models to support predictions on arbitrary CDAG topologies.

Starting from the roots of the CDAG of a CU, that is, its output, we recursively apply either the sequential or the parallel model depending on the operation type of the parent: when the current node and the parent node uses the same operation, then the

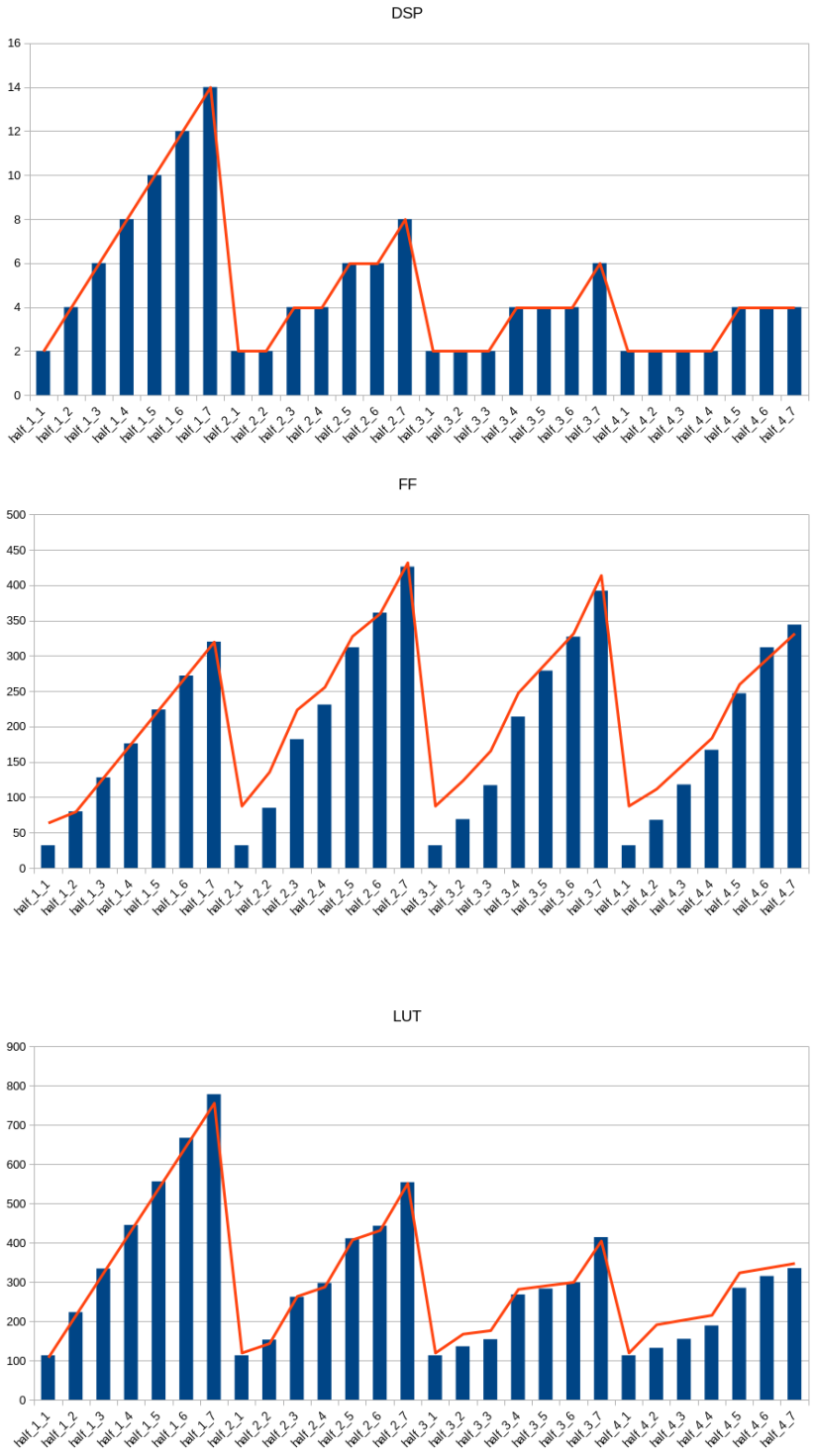


Figure 7.4: Resource predictor (red line) versus pre-P&R (blue bar) for several FPGA resources for reductions of additions. $half_a_b$ denotes a reduction using a operation with an II of b on FP16 data type

Resource	MAE (reductions)	MAE (vector)
DSP	0 %	0 %
LUT	7.8 %	4.2 %
FF	35.3 %	7.5 %
SRL	10.7 %	0 %

(a) Predictor accuracy on sequence and parallel (vector) instances of FP16 adders

Resource	MAE (reductions)	MAE (vector)
DSP	0 %	0 %
LUT	16.6 %	7.9 %
FF	35.3 %	7.5 %
SRL	10.7 %	0 %

(b) Predictor accuracy on sequence and parallel (vector) instances of FP16 multipliers

Table 7.2: Resource predictor accuracy

```
   Input: C the CDAG of a CU
 1 Function estimate_area(C, parent_node)
 2     node = root(C);
 3     if parent_node = nil then
 4         return seq_estimate (node);
 5     else
 6         left = estimate_area(node→left_child, node);
 7         right = estimate_area(node→right_child, node);
 8         if node.type = parent_node.type then
 9             return left + right + seq_estimate (node);
10         else
11             return left + right + par_estimate (node);
12         end
13     end
14 end
```

Algorithm 9: Composition of the sequential and parallel CU area models

sequential model is applied. In the other case and for the base case, the parallel one is used, as illustrated in Alg. 9. Note that all arithmetic operations considered have only two children, hence our restriction to this case.

7.3.6 Evaluation

We evaluate the complete CU resource estimator against the systematic combination of the purely sequential model on a random set of 7 CUs types mixing sequence and parallel composition of 3 or 4 multipliers and adders. We vary the number of repetitions of the CDAG inside the CU (from 1 to 4) as well as its II (also from 1 to 4) and report our estimates compared to the post-P&R resource usage in Tbl. 7.5. As expected, combination of the sequential and parallel model is needed to achieve satisfying (i.e. sub-10% MAE) accuracy. Coefficients for each model are reported in Fig. 7.3 (sequential combinations) and in Fig. 7.4 (parallel combinations). We use the Xilinx ZCU104 board with a target frequency of 100 MHz as target, and Xilinx Vitis High Level Synthesis design suite 2021.2 for the synthesis tool.

Resource	$\mathcal{K}_{res}$	$\mathcal{K}'_{res}$	$\mathcal{K}''_{res}$
DSP	2	0	0
LUT	2	0.625	-0.5
FF	2.5	1.5	-1.875
SRL	1	0	0

(a)

Resource	$\mathcal{K}_{res}$	$\mathcal{K}'_{res}$	$\mathcal{K}''_{res}$
DSP	2	0	0
LUT	6	0.75	-0.5625
FF	2.5	1.5	-1.875
SRL	1	0	0

(b)

Table 7.3: Resource predictor coefficients for sequential composition of additions (a) and multiplications (b)

Resource	$\mathcal{K}_{res}$	$\mathcal{K}'_{res}$	$\mathcal{K}''_{res}$
DSP	2	0	0
LUT	7	0.125	0
FF	1	0.5	-0.35
SRL	0	0	0

(a)

Resource	$\mathcal{K}_{res}$	$\mathcal{K}'_{res}$	$\mathcal{K}''_{res}$
DSP	2	0	0
LUT	2.5	0.125	-0.0625
FF	1	0.5	-0.35
SRL	0	0	0

(b)

Table 7.4: Resource predictor coefficients for parallel composition of additions (a) and multiplications (b)

Resource	MAE (Sequential)	MAE (Composition)
DSP	0 %	0 %
LUT	11.4 %	7.4 %
FF	37.2 %	14.4%
SRL	29.5 %	7.6 %

Table 7.5: Resource predictor accuracy on FP16 multipliers and adders

As DSP are explicitly instantiated by the HLS tool when required by arithmetical operations – and nowhere else in our simple PU as they carry no loop nor data-dependent control flow –, we are able to perfectly predict their number. LUT and SRL are also predicted within a 10 % error margin as they handle the routing of data inside the CU. However, the number of FF is harder to predict as they indicates the amount of elementary storage element used in the design. Because of resource reuse happening directly inside the CU (one register may be reused at several operator stages) the predicted number of FFs is bigger than the actual number of integrated FFs on CUs with 4 operators, as seen in Fig. 7.5; hence a higher MAE of 14.4 % for this metric.

For the same reasons, the number of DSPs is proportional to the number of required operations: from a pure resource usage point of view, there is no interest in combining operations into CUs. This is especially true for CUs with an II greater than one, where the synthesis tool is not able to reuse DSPs across different operators. This sharing technique will be tackled in Chap. 8, where aggressive resource sharing of DSPs is performed through

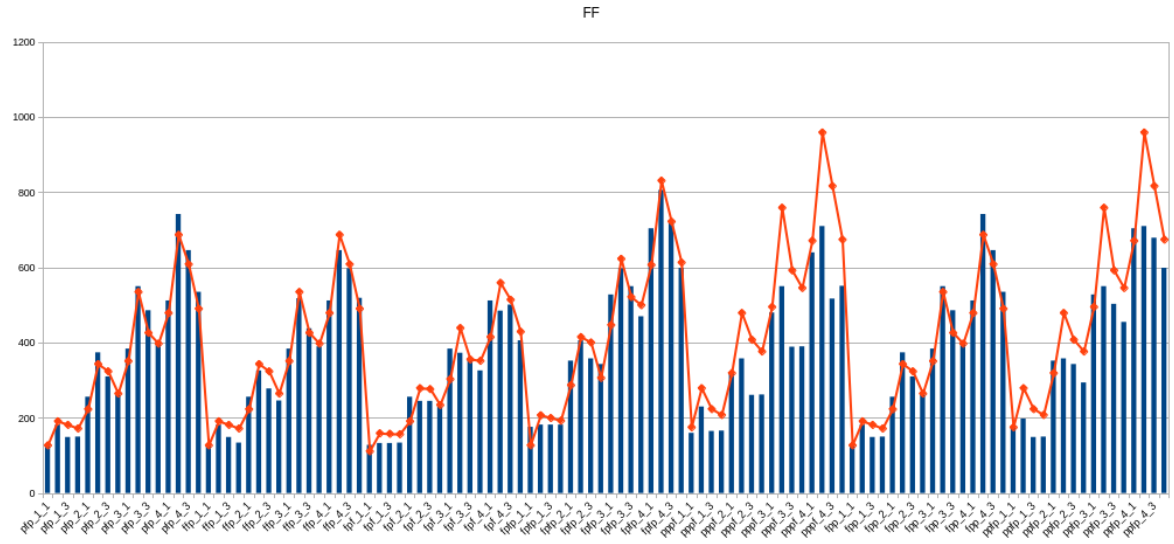


Figure 7.5: Resource predictor (red line) versus post-P&R values (blue bar) for FF predictions on mixes of 3 and 4 operators CU

cross-operator reuse. Moreover, the effect of clustering operations in CUs is even worse on FFs and LUTs: multiplying the size of a CU by replicating it (e.g. creating a vector CU) leads to no gain, neither does the increase of the CU's II on the performance-per-resource ratio. Indeed, $\mathcal{K}_{res} = 0$, $K'_{res} > 0$ and $2K'_{res} * 2 + K''_{res}$, for both LUTs and FFs, showing that the number of LUTs and FFs *increases* with the II.

Contrary to what could be expected, decreasing the speed of CUs results in no resource sharing on both routing (LUTs) and storage (FFs) elements, as data pathes become more complex when the II increases. On our tested designs, the number of instantiated SRL is also not significant enough to change the overall resource usage, with value being non-zero only on CUs with an II of 1, typically in less that 400 units, whereas FF and LUT number ranges from ∼300 to ∼3000.

Operator	Overhead Latency	Initial Latency
+	2	1
*	1	1

Table 7.6: Latency predictor coefficients on FP16 multipliers and adders

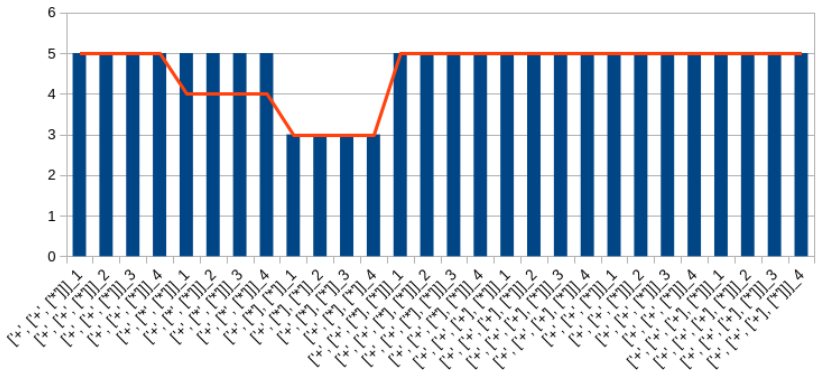


Figure 7.6: Latency predictor (red line) versus HLS ground truth (blue bar). [*ops*]_*a* denotes a CDAG linearised as in the list `[ops]` replicated *a* times

7.4 Latency Estimation of Compute Units

Along with the resource usage (estimated in the previous section) and the II (specified using source code annotations), the latency is a crucial metric for design optimization, as it specifies the number of cycles elapsed between the start of the computation of a CU and its results being available. In this section, we present a simple model for latency prediction of a CDAG, when mapped to a single Compute Unit.

7.4.1 Formula and Micro-benchmarking

The latency estimate follows a simple idea: if two operations happen in sequence, then the total latency is the sum of the individual latency of each path; if they occur in parallel, then the latency is the maximum of the two paths. This correspond the notion of *critical path*: the longest possible path combination of data inside the CU.

Therefore, we use a simple model that takes into account both the latency due to computations and the latency due to data routing. For each supported operator, we measure their latency when enclosed alone in a CU, that we call *initial latency*. Then, we measure sequences of operator and deduce the *overhead latency*, that is, the supplementary time for each new operator in sequence. Corresponding coefficients are reported in Tbl. 7.6.

7.4.2 Evaluation

We evaluate the estimator of latency defined in the previous section with the same experimental setup than Sec. 7.3.4, restricted to fully pipelined CUs as we saw in Sec. 7.3.6 that reducing the II did not provide resource reduction.

On every tested CU, the estimate was at most 1 cycle off, leading to an average error of 3.57 %, as illustrated in Fig. 7.6. These artifacts are due to clock synchronization: while we count latency as a number of cycles, it in fact represents a duration, which is over-approximated to fit into cycles by the synthesis tool. However, composition of operators may shortcut some unnecessary stall cycles to ensure correct propagation of the signal to the next operator, leading to one-off errors on the tested CDAGs.

7.5 Naive Convex Encoding of the Resource Sharing Problem

Given an estimator of the area of CUs and their latency, we can formulate the combined architecture generation and resource mapping/scheduling problem as a convex optimization one in order to solve it using state-of-the-art Integer Linear Programming (ILP) solvers such as Gurobi [48] or CPLEX [82]. This section presents such a formulation in a naive way, using one variable per solution of the problem and explicitly affecting a timestamp to each operation of the application, without concern about regularity of the original program.

7.5.1 Variables

We define a program as the weighted CDAG $(\mathcal{O}, \mathcal{D})$ representing the fully unrolled graph of computations executed at runtime, with $\mathcal{O}$ the set of operations and $\mathcal{D}$ the set of dependencies, weighted by the latency of each operator.

Our goal is to find the optimal partitioning of $\mathcal{O}$ into CUs, i.e. a set of hardware units and a legal affectation of the CDAG operations to the units such that the total execution time of the CDAG is minimal.

We use the following notations:

Sets

- $\mathcal{O}$ is the set of all operations in the program, we note its elements $o \in \mathcal{O}$.
- $\mathcal{D}$ is the set of all dependencies, modeled by a triplet (o_1, o_2, i) with $(o_1, o_2) \in \mathcal{O}^2$ and i the type of dependency (i.e. place of o_1 as argument of o_2). We note $o_1 \rightarrow_i o_2$, or $o \rightarrow o'$ for any type of dependency.
- $\mathcal{P}$ is the set of all CUs, we note $p \in \mathcal{P}$ its elements.
- $\mathcal{T}$ is the set of all operation types, in our case restricted to $\{+, \times\}$.
- $\mathcal{C} = [\![0, n]\!]$ is the set of all possible timestamps of operations in the global program. We note $c \in \mathcal{C}$ its elements, standing for *cycles*.
- $\mathcal{C}_p = [\![0, n_p]\!]$ is the set op all timestamps of operations inside a CU. We note $s \in \mathcal{C}_p$ its elements.
- $\mathcal{S}$ is the set of all operation slots of a PE, corresponding to parallel execution paths at a given timestamp. We note $s \in \mathcal{S}$ its elements.

Variables

- $\delta_{p,c}^{o,s} \in \{0, 1\}$ indicates that op o is scheduled at the (CU-local) timestamp c on CU p slot s. For convenience, we note $\delta_p^o \in \{0, 1\}$ which indicates that op o is mapped to PE p, regardless of its timestamp and its slot. For convenience, we also define $\delta_p^o \in \{0, 1\} = \sum_{s \in \mathcal{S}} \sum_{c \in \mathcal{C}_p} \delta_{p,c}^{o,s}$ and $\delta_{p,c}^o$ which indicates that o is mapped to CU p cycle c, regardless of the slot, with $\delta_{p,c}^o \in \{0, 1\} = \sum_{s \in \mathcal{S}} \delta_{p,c}^{o,s}$.

- $\sigma_c^o \in \{0,1\}$ indicates that op o is scheduled at timestamp c in *the global design*. For convenience, we note $\sigma^o \in \mathbb{N} = \sum_{c \in C} c \cdot \sigma_c^o$ the starting timestamp of o.
- $\gamma_c^o \in \{0,1\}$ indicates that the CU containing o is scheduled in the global design to start at timestamp c. For convenience, we note: $\gamma^o \in \mathbb{N} = \sum_{c \in C} c \cdot \gamma_c^o$ the starting timestamp of the PE containing o.
- $\iota_{o_1,o_2} \in \{0,1\}$ indicates that the CU containing o_1 also contains o_2 and that o_1 and o_2 are mapped to the same PE call.
- $area_p \in \mathbb{N}$ is the area taken by PE p, and $area_{p,c} \in \mathbb{N}$ is the area taken by PE p restraint to its $c \in \mathcal{C}_P$ first timestamps.
- $lat_p \in \mathbb{N}$ is the latency of CU p.

Other constants

- $type : \mathcal{O} \to \mathcal{T}$ the function giving the type of an operation, we note $o.type$ for $type(o)$.
- $lat : \mathcal{T} \to \mathbb{N}_+^*$ the function giving the latency of an operation, we note $o.lat$ for $lat(o)$.
- $area_{pat}^t$ is the area taken by an op of type t with pattern pat being either par (in parallel) of seq (in sequence). From 7.3, our model does not require neither the slot nor the timestamp of the operation.
- K_{area} an upper bound of the area, defined by the user as a constraint over the size of the output IP. This is also the constant that varies when generating a family of design of various area/performance ratio.
- K_{cycle} an upper bound of $\mathcal{C}$, used in the constraints to specify if-conditions.

7.5.2 Objective Function

As our goal is to minimize the total execution time of the IP while keeping the best efficiency in terms of performance-per-area, our objective function is:

$$\min_{lex}(lat, area)$$

7.5.3 Constraints

To ensure correctness of the output design and schedule, *legality* constraints are given to the ILP solver:

Legality constraints

- Definition of the global area as the sum of CU areas, neglecting the interconnect cost:

$$area = \sum_{p \in \mathcal{P}} area_p$$

- Definition of the CU areas as the maximum of the area of the CU restricted to the first c cycles:

$$\forall c \in \mathcal{C}, area_p \geq area_{p,c}$$

- Definition of the individual CU area, from the previous section's modeling:

$$\begin{aligned}
&\forall p \in \mathcal{P}, \forall c \in \mathcal{C}_p, area_{p,c} = \\
&area_{p,c-1} + \\
&\sum_{s \in \mathcal{S}} \max \Bigg[\max_{\substack{(o,o',i) \in \mathcal{D} \\ o.type = o'.type}} \left(\iota_{o,o'} \cdot area_{seq}^{o'.type} + (1 - \iota_{o,o'}) \cdot area_{par}^{o'.type} - K_{area} \cdot (1 - \delta_{p,c}^{o',s}) \right), \\
&\qquad \max_{\substack{o' \in \mathcal{O} \\ \nexists (o,o',i) \in \mathcal{D} \\ \text{s.t. } o.type = o'.type}} \left(\delta_{p,c}^{o',s} \cdot area_{par}^{o'.type} \right) \Bigg]
\end{aligned}$$

- Definition of the global latency as the maximum of the final timestamp of each PE, that is, the starting time of the CU summed with its latency:

$$\forall o \in \mathcal{O}, \forall p \in \mathcal{P}, \gamma^o + lat_p \leq lat + K_{cycle} \cdot (1 - \delta_p^o)$$

- Definition of the CU latency as the maximum timestamp of an operation inside a CU, summed with its latency:

$$\forall p \in \mathcal{P}, lat_p = \max_{o \in \mathcal{O}} \left(\sigma^o - \gamma^o + o.lat - K_{cycle} \cdot (1 - \delta_p^o) \right)$$

- Definition of the common appurtenance to the same CU:

$$\forall (o_1, o_2) \in \mathcal{O}^2, \iota_{o_1,o_2} = \max_{\substack{p \in \mathcal{P} \\ c \in \mathcal{C}}} \left(\gamma_c^{o_1} \wedge \gamma_c^{o_2} \wedge \delta_p^{o_1} \wedge \delta_p^{o_2} \right)$$

- One operation is mapped to only a unique CU and a unique (CU-local) timestamp:

$$\forall o \in \mathcal{O}, \sum_{p \in \mathcal{P}} \delta_p^o = 1$$

- One operation has a single global timestamp:

$$\forall o \in \mathcal{O}, \sum_{c \in \mathcal{C}} \sigma_c^o = 1$$

- A CU has at most one operation per slot and per timestamp, though it can have "holes":

$$\forall p \in \mathcal{P}, \forall s \in \mathcal{S}, \forall c \in \mathcal{C}, \forall c' \in \mathcal{C}_p \sum_{o \in \mathcal{O}} \left(\sigma_c^o \wedge \delta_{p,c'}^{o,s} \right) \leq 1$$

- Homogeneity of type in CU slots – if any two operations are mapped to the same timestamp, same slot, then they must be of the same type:

$$\forall p \in \mathcal{P}, \forall s \in \mathcal{S}, \forall c \in \mathcal{C}_p, \forall (o_1, o_2) \in \mathcal{O}^2 \text{ s.t. } o_1.type \neq o_2.type, \delta_{p,c}^{o_1,s} + \delta_{p,c}^{o_2,s} \leq 1$$

- Definition of starting time c of the CU call containing an operation as the timestamp where there exist c' for which the operation in scheduled locally in the CU at c', and globally at $c + c'$:

$$\forall o \in \mathcal{O}, \forall c \in \mathcal{C}, \gamma_c^o = \max_{\substack{c' \in \mathcal{C}_p \\ c+c' \in \mathcal{C} \\ p \in \mathcal{P}}} \left(\sigma_{c+c'}^o + \delta_{p,c'}^o - 1 \right)$$

- Relationship between the scheduled time in a CU and in the global program – the global timestamp of an operation is defined such that there exist c' for which the operation in scheduled locally in the CU at c', and the CU is scheduled (globally) at $c - c'$:

$$\forall o \in \mathcal{O}, \forall c \in \mathcal{C}, \sigma_c^o = \max_{\substack{c' \in \mathcal{C} \\ c-c' \in \mathcal{C}_p \\ p \in \mathcal{P}}} \left(\gamma_{c'}^o + \delta_{p,c-c'}^o - 1 \right)$$

Program-specific constraints

The former constraints ensure that the output CU-covering of the original program is well-formed, but they are not enough to guarantee the semantic equivalence between the original program and the generated one. Indeed, dependencies must be respected, which is has not been encoded into ILP constraints. Therefore, we add the following constraints:

- Latency constraint on the global schedule – an operation cannot be scheduled before the availability of its source operands:

$$\forall (o, o', i) \in \mathcal{D}, \sigma^o + o.lat \leq \sigma^{o'}$$

  S

- Data produced by a CU are not available right at the end of $igna_o + o.lat$, but have to wait for the end of the CU execution before being transmitted again to another CU:

$$\forall (o_1, o_2, i) \in \mathcal{D}, \forall p \in \mathcal{P}, \gamma^{o_1} + lat_p \leq \gamma^{o_2} + K_{cycle} \cdot (1 - \delta_p^{o_2}) + K_{cycle} \cdot (1 - \iota_{o_1,o_2})$$

- A CU must be atomic, i.e. must not depend both for input and output on another CU:

$$\forall (o_1, o_2, i) \in \mathcal{D}, \forall o_3 \in \mathcal{O} \text{ s.t. } \exists (o_2, o_3, i') \in \mathcal{D}, \iota_{o_1,o_3} \leq \iota_{o_2,o_3}$$

- Unicity of the DAG of computation for each CU:

$$\forall (o_1, o_2, o_3, o_4) \in \mathcal{O}^4, \text{ if } o_1.type = o_3.type \text{ and } o_2.type = o_4.type, \text{ then}$$

  - If $\exists i, o_1 \rightarrow'_i o_2$ and $o_3 \rightarrow_i o_4$, then no constraint is added.
  - Else:

$$\forall p \in \mathcal{P}, \forall (s_1, s_2) \in \mathcal{S}^2, \forall (c_1, c_2) \in \mathcal{C}_p^2 \text{ s.t. } c_1 \neq c_2, \left(\delta_{p,c_1}^{o_1,s_1} + \delta_{p,c_2}^{o_2,s_2} + \delta_{p,c_1}^{o_3,s_1} + \delta_{p,c_2}^{o_4,s_2}\right) \leq 3$$

Constraints to speed up solving time

Though this set of constraints provides *valid* solutions of the general resource sharing problem, they do not guarantee unicity of the representation of the solution, in the sense that one CU may be represented under different encodings because of the degree of freedom provided by slots and CU identifiers. Therefore, we add supplementary constraints in order to prune the solution space and keep a minimal number of encodings per CU.

- Order the CU operation slots, so that the first one are filled first:

$$\forall p \in \mathcal{P}, \forall s \in \mathcal{S} \setminus \{s_{last}\}, \forall c \in \mathcal{C}_p, \delta_{p,c}^{o,s} \geq \delta_{p,c}^{o,s+1}$$

- Order the CUs, so that the one of minimal identifiers are used first:

$$\forall p \in \mathcal{P} \setminus \{p_{minus}\}, \sum_{o \in \mathcal{O}} \delta_p^o \leq \sum_{o \in \mathcal{O}} \delta_{p+1}^o$$

7.6 Real-life Implementation, Heuristics

Though the presented ILP is a correct encoding of the combined architecture design and scheduling problem, it still leave a lot to be desired. From the expressiveness point of view, the instantiated CUs are of fixed CDAG and cannot be programmed even for small variation of control flow. This lack of flexibility may cause the solution to either add extra stall cycles (waiting for another CU to be ready) or extra CUs when dealing with repetition of similar compute pattern. Therefore, the LP tends to converge to solutions with CUs containing few operations in order to maximize operator reuse instead of discovering regular compute pattern. This fact lead to the production of a Generic Accelerator where small CUs are reused along with a customizable structure in order to accelerate a family of applications, as described later in Chap. 8.

This formulation also completely eludes the cost of the interconnect, which is a classical issue with resource sharing [83]. Here, designs with irregular reuse patterns will result in generating, for each CU, a multiplexer of high fan-in selecting one input per active cycle. A solution to this problem is also presented in Chap. 8, where the considerable amount of glue logic is acknowledged and used to allow reprogramming of the design with regeneration of a news bitstream.

Finally, this formulation is useless in practice due to its complexity, as illustrated in this section. Indeed, the number of boolean variables is of the order of $O(|\mathcal{O}|\cdot|\mathcal{S}|\cdot|\mathcal{C}_p|\cdot|\mathcal{T}|+|\mathcal{O}|^2)$ (number of $\gamma_{p,c}^{o,s}$ and ι_{o_1,o_2}), with $O\left(|\mathcal{P}|\cdot|\mathcal{C}_p|\cdot|\mathcal{S}|\cdot(|\mathcal{O}|^2+|\mathcal{C}|)\right)$ constraints (number of sub-expression in the computation of the CU areas and unicity of the CU/slot/local timestamp of each operation). Therefore, the number of possible solutions grows exponentially with respect to the size of the input application, dooming any hope of scaling on real-life problems of several thousands of nodes.

7.6.1 Exact Implementation: Scaling

Even though the size of the space is exponential, a smart convex formulation of the problem could reveal to be enough on small problem sizes. To determine whether this is the case for the combined placement-schedule problem, on FPGA, we encoded the ILP described in the previous section using Gurobi 10 [48], and collect solving time on several toy AST on an Arch Linux machine running Gurobi 10 and Linux 6.2.11, equipped with an AMD Ryzen 2700U and 32 GiB of RAM, with CU sizes restricted to only 1 operation.

Results are reported in Fig. 7.7 (note the log y axis). All applications output ILPs that times out (more than 10 hours solving time) above 12 nodes to place/schedule, even though *this problem does not require any design space exploration from the solver*, as there is only one valid affectation of operations to possible CUs. Given the time reported for such trivial CU mappings, we conclude that this formulation is not usable on real-life problems sizes due to scalability issues.

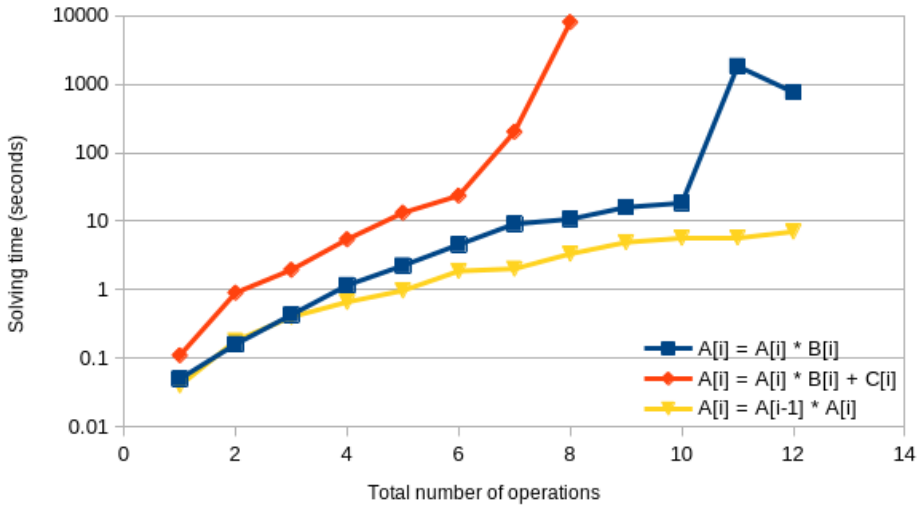


Figure 7.7: Scaling of the ILP formulation with the number of node in the target AST for three loop bodies. Missing points time out (more than 10 hours of solving).

7.6.2 A Faster Greedy Approximation

A classic solution to overcome the complexity issue is to fall back to approximations, trading optimality in favor of execution time, as seen in Part. I. Nevertheless, the goal still lies in maximizing accuracy and speed gains, but reduced to cases that are the most common in real-life, while more exotic corner cases may present degraded behavior.

Implementation

Algorithm To avoid the complexity resulting from the evaluation of all possible mappings, we reduce down the problem to the classic scheduling under resource constraint problem. Given a fixed architecture represented as a list of PE with multiplicity, we aims at maximizing reuse of available components, which by proxy minimizes the execution time.

A classic approximation for this family of problems is greedy algorithms, in which the output solution results from a series of locally optimal choices. In our case, the choice is made during scheduling of the operations, operating on the fully unrolled CDAG. More precisely, we use a priority list on the CUs to map the first compatible unit to the first matching CDAG pattern, detailed in Alg. 10.

Code generation While solving time reveals to be an issue for optimal resource-shared designs, actual implementation of and end-to-end design generation chain raises further issues. Indeed, while the combined placement-schedule problem optimizes reuse of low-level operations such as adders / multipliers, it does not explicitly specify the interconnect size and topology. Our implementation uses one intermediate register for each CU input and output, and relies on the HLS tool to simply redundant ones. Moreover, to avoid redundant computations, we run a quadratic constant subexpression elimination pass

Input: $\mathcal{C}$ the unrolled CDAG of a program, $\mathcal{A}$ an array of CU representing the architecture, sorted by priority
Output: σ the schedule of the operations, δ its schedule

```
1  WorkList = {c ∈ C with no dependence};
2  t = 0;
3  RS = [[False]×size(A)] × size(C)]; // Reservation Station
4  while WorkList is not empty do
5      CurRS = RS[t];
6      for i ∈ [0, size(A) − 1] such that CurrRS[i] = False do
7          for o ∈ WorkList do
8              if A[i].CDAG matches o.tree then
9                  for op ∈ o.tree matched by A[i].CDAG do
10                     σ(op) = t + latency_offset(op, match);
11                     δ(op) = A[i];
12                     op.available = t + A[i].lat;
13                 end
14                 for ti ∈ [0, A[i].II − 1] do
15                     RS[t+ti][i] = True;
16                 end
17                 WorkList.remove(o);
18             end
19         end
20         for o ∈ A without placement/schedule do
21             if ∀op ∈ dep(o), op.avail ≤ t then
22                 WorkList.add(op);
23             end
24         end
25     end
26     t = t+1;
27 end
```

Algorithm 10: Greedy placement-scheduling of CDAG under resource constraints

in order to generate only one execution path for each possible CU source value. These intermediate results are stored in one global buffer, implemented as BRAMs in the final design.

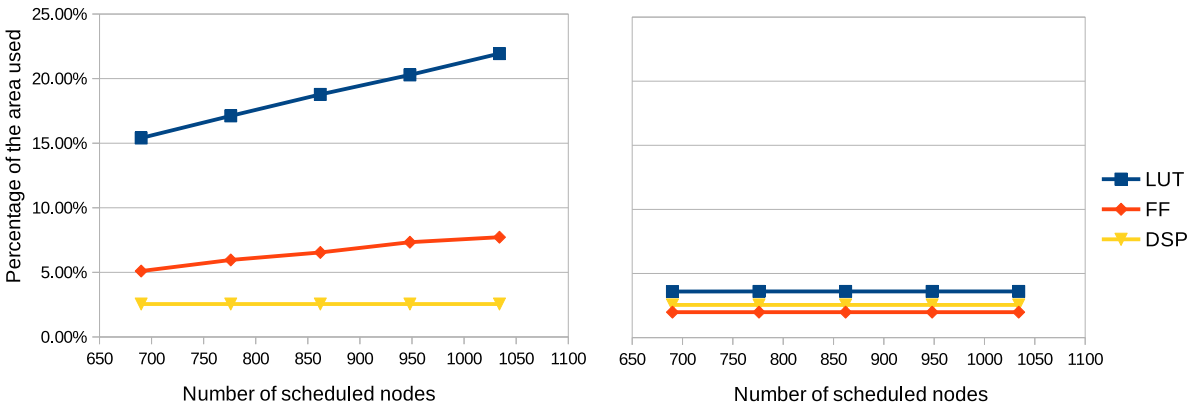
Experimental Results

We implemented the greedy scheduling heuristic as a pass inside the PoCC [84] compiler, and report in this paragraph resources and execution times after after place and route of an out-of-context implementation of the designs using Vivado Design Suite and Vitis HLS 2022.2. The target FPGA was set to a Zynq Ultrascale+ XCZU7EV MPSoC (252k LUT, 504k FFs, 1728 DSPs) configured to a 100 MHz frequency goal.

Execution time of our placement-schedule implementation is detailed in Tbl. 7.7, as well as number of operations for our problem sizes. Fig. 7.8a shows the evolution of

N	Number of ops	Schedule-mapping time
8	690	2.4s
9	776	3.5s
10	862	4.5s
11	948	5.8s
12	1034	7.3s

Table 7.7: Greedy scheduling-placement solving time and number of operations for one Nx16 DWT iteration.



(a) **(b)**

Figure 7.8: LUT and FF used as percentage of the chip size for N by 16 DWT designs generated using our CU greedy placement heuristic (a) and a pure HLS version (b)

the resource usage as a function of the number of scheduled nodes when computing one iteration of the Discrete Wavelet Transform (DWT) benchmark on a Nx16 image using double precision data type (the most resource-hungry in terms of compute unit size) for a fixed architecture of 8 CUs: 4 adders and 4 multipliers.

As a baseline, the resources used by same benchmark synthesized using pure HLS annotated with `pragma HLS allocation operation` in order to limit synthesis to the same number of hardware units is reported in Fig. 7.8b. Performances metrics of the design produced by both techniques are presented in Fig. 7.9. As the pure HLS version is not pipelined, only its latency (i.e. equals to its II) is reported.

As the generated hardware is derived from the loop structure of the code, the size of the problem is only represented in the final design as the number of iterations of the loop (i.e. number of execution of the subdesign representing the loop body), which has little effect on the required resources. This explains the reason for the designs not to change significantly its resource usage with the size of the input data, as the topology of the loop body (only linked to the number of CUs) remains unchanged.

Due to the use of intermediate registers to store partial computations in our greedy scheduling implementation, the output designs are pipelined, allowing small overlap of executions during final stage of copying output data from the shared buffer to the top

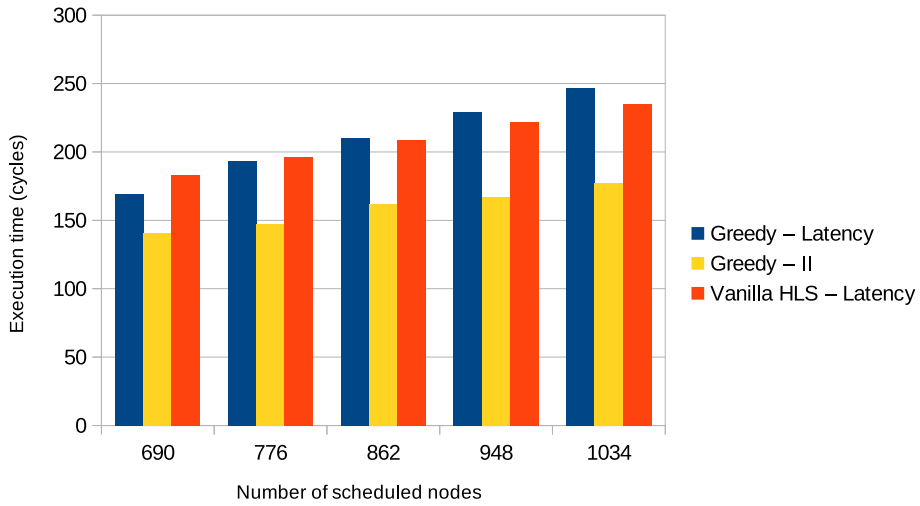


Figure 7.9: Performance metrics (latency and II) for a the vanilla resource-constraint HLS design (non-pipelined) and the greedily scheduled one.

level I/O interfaces. On the other hand, the pure HLS version does not provide any pipelining, which explain partially its efficiency in terms of used resources.

As seen in Sec. 7.1.2, due do dependencies on the syntactic structure of the loop, all CUs are not used in the first two loop nests of the benchmark, while the greedy approach uses all possible parallelism opportunities, hence resulting in a 25 % performance gain in terms of II. However, the time needed to copy back the data void this gains when looking at the latency, where the greedy version offers similar performances than the pure HLS one.

While the main goal of the greedy heuristic – reducing the execution time to acceptable level, less than 10 seconds on a problem of more thanks a thousand nodes to place/schedule –, another issue occurs: the usage of routing resource skyrockets, reaching 22% of the chip for an 8-CU design on DWT 12x16 and linearly growing with the size of the data. As a comparison, the architecture only requires 44 DSP (no matter the size of the input), which accounts for 2.5 % of the total number of DSPs on the chip; and a 4-pragma annotated HLS version only requires an area usage of 3.6 % of the available LUTs and 2.8 % of the FFs for the same DSP usage.

The reason for this resource-hungriness is rather straightforward: in the absence of a clear reuse pattern of the data across CUs[14], the synthesis tool uses one multiplexer per compute unit input, with a fan-in at least equals to its number of calls inside the design. As the number of CU is kept fixed for this experiments, the multiplexers becomes proportionally bigger with the size of the input data.

[14]Contrarily to vanilla HLS that leverages the loop-base structure of the code use fixed, area-efficient data selection units.

Worse, this behavior would apply to both the greedy approximation's solution and the solutions output by the solver, as a metric of the regularity of the access pattern is never being optimized for, nor even expressed in our approaches.

We thus conclude that the notion of optimality when only considering placement / schedule at the compute unit level is not a relevant metric on real-world applications, as the cost of the interconnect becomes preponderant. This calls either for a tool dedicated to the optimization of the interconnect by finding regularity in their reuse pattern – which falls out of scope of this manuscript – or a more flexible approach that would replace the costly multiplexers with a more flexible interconnect whose benefits overpass mere pipelining. This second approach is tackled in the next chapter of this manuscript with the Generic Accelerator, a method dedicated to the generation of DSP-efficient designs for multi-benchmark acceleration.

Chapter 8
Automated Generation of Semi-generic Throughput-oriented Accelerators

As seen in Chap. 6, the accessibility of designing accelerators has significantly increased thanks to the recent improvements in performances and user interface of HLS tools [55, 57], as well as supplementary hardware/software design stacks (e.g., with compilers for High-Level Synthesis such as the Xilinx Merlin compiler [85, 86]). Designers can now quickly generate customized designs for a particular application, or possibly (a set of) kernels within it which are candidates for profitable acceleration [87, 77]. However, re-targeting an existing implementation to a new application is often time-consuming if not impossible: at best, it requires generating and updating a new bitstream on the FPGA, and at worst it is not possible at all for ASIC-based designs if the new workload has not been taken into account at design time.

However, as seen in Chap. 7, the issue of efficient hardware generation is complex, and automated generation of optimal designs in the sense of maximal performance/area ratio is far from being generally answered. Flexible accelerators, e.g., using overlays [88] or VTA [89], ignores this issue by taking the opposite approach: finding a unique design that maximize occupancy of the chip, while mapping on a second step programs to the overlay topology. These attempts try to bring the best of both worlds: (most of) the performance benefits of hardware specialization, while maintaining some generality of computations that can be accelerated; to the cost of a supplementary compilation step when mapping applications to the (one-size-fit-all) design. In this chapter, whose results have been presented in [90], we develop an approach called *kernel merging* to create multi-functionality accelerators from a collection of to-be-accelerated applications, and present the algorithms needed for end-to-end automation. We start by a simple yet pragmatic observation: *it is possible to easily build a semi-Generic Accelerator (GA) by restricting the functionalities addressed to those amenable to polyhedral modeling*, that is, cases where each functionality supported (e.g., GEMM, AXPY, etc.) by the accelerator can be exactly modeled as a polyhedral program, where the loop bounds and array access functions are affine expressions made of the surrounding loop iterators. From then, we derived polyhedral analysis to split the input benchmarks into polyhedral kernels defining the atomic primitives to be accelerated on-chip, an generate a corresponding resource-efficient accelerator based on a template architecture. This chapter is organized as follow: first, Sec. 8.1 illustrates our approach on two simple examples, while Sec. 8.2 details the polyhedral kernel decomposition and merging algorithms as well as possible code generation approaches. Sec 8.3 details the hardware implementation of the accelerator, and Sec 8.4 evaluates its performance and area consumption compared to dedicated designs. Sec. 8.5 goes through the limitations of our approach as well as possible improvement directions. Finally, Sec. 8.6 presents former approaches to semi-specific accelerator design and compare them with the one presented in this chapter.

```
1  L1: for (j = 0; j < N ; j++)
2        mean[j] = 0.0;
3  L2: for (i = 0; i < N ; i++)
4        for (j = 0; j < N ; j++)
5          mean[j] += data[i][j];
6  L3: for (j = 0; j < N ; j++)
7        mean[j] /= N;
8  L4: for (i = 0; i < N ; i++)
9        for (j = 0; j < N; j++)
10         data[i][j] -= mean[j];
```

Figure 8.1: CENTER naive implementation

8.1 Illustrative Examples

We illustrate the gains of a semi-generic accelerator on a workload composed of 3 independent correlation matrix (CORR) computations, a widely used data science calculus. First, in Sec. 8.1.1, we show how coarse grain pipelining may help speed up batched computation of CENTER, a sub-problem of CORR. Then, we show in Sec. 8.1.2 the design choices at stake when crafting a semi-generic accelerator capable of efficiently executing both problems. Finally, we detailed the flow of the accelerator generation and usage in Sec. 8.1.3.

8.1.1 Data Centering

Let us consider the program realizing the following matrix transformation, corresponding to data centering:

$$X^C_{ij} = X_{ij} - (\sum_{i'} X_{i'j})/n$$

One naive implementation of this computation is given in Fig. 8.1. It uses four loop nests with different operators:

L1: Initialization of the mean vector (no operator)

L2: Column-wise accumulation of the matrix coefficients ($+$)

L3: Division of the previous accumulated values by N ($/$)

L4: Column-wise subtraction of the mean to the input matrix ($-$)

These loops form what we call *functionalities* or *kernels*, which are defined as affine subparts of the input program, represented using a single loop nest. Under the resource sharing point of view, some of this functionalities can rely on the same physical compute unit, that may or may not be shared across kernels. For example, operator sharing can happen between the addition and subtraction part, as it boils down to a preprocessing of a single bitflip on FPGA per FP16-encoded data. In the rest of the paper, we note this operator $\pm$, that can also perform absolute value with the same trick.

The dispatch of kernels over functional units that execute them is fundamental for the generation of efficient accelerators. For example, the usual coarse-grain replication [62, 91]

```
for(id=0; id<BATCH_SIZE+4; id++)
  for (i = 0; i < N ; i++)
    for (j = 0; j < N ; j++) {
      if (id < BATCH_SIZE and i==0)
        mean[id][j] = 0.0;
      if (id < BATCH_SIZE+1 and id>=1)
        mean[id-1][j] += data[id-1][i][j];
      if (id < BATCH_SIZE+2 and id>=2 and i==0)
        mean[id-2][j] /= N;
      if (id >= 3 and i==0)
        data[id-3][i][j] -= mean[id-3][j];
    }
```

Figure 8.2: CENTER coarse-grain pipelined implementation

of a single high-performance design will fail to provide the best throughput-per-area on a sequence of CENTER. However, deeper resource sharing can be achieved through retiming [92] of the kernels: by spreading problems across time, we avoid simultaneous usage of the / operator. The transformed code corresponding to retimed CENTER is reported on Fig. 8.2. In the HLS framework [64], this retiming must be followed by a loop merging transformation to ensure both operator and control structure reuse; which forms the structure of a coarse-grain pipeline [93]. At each step of the `id` loop, each pipeline stage executes one kernel instance.

However, this merging is not trivial when it comes to iteration spaces: L1 and L3 iterate over a space of size N, while L2 and L3 iterate over a space of size N^2, hence the need of conditions on the loop iterator (here `i`) to ensure a correct number of executions of the loops bodies of smaller iteration spaces. As a downside, this means that the divider unit is idle at least $(N-1)/N$ fraction of the time during the whole computation. We can reduce this idle time with batching: by executing several independent instances of the same problem, low usage compute units can be reused without heavy impact on the overall execution time, as illustrated in Tbl. 8.1. Here, we report execution time and DSP usage of a coarse-grained pipelined design realizing 10 batched executions of CENTER, compared to a dedicated design either replicated 10 times (CENTERx10) or 10 successive calls to the same IP (10xCENTER); CGP-CENTER-inf denoting the maximum achievable throughput, corresponding to an infinite number of successive independent CENTER instances. As expected, high batching factors favor performances as the pipeline stages are proportionally less idle on this workload.

8.1.2 Center, Correlation and Multi-purpose Acceleration

Even though CENTER transformation is a part of the Correlation (denoted CORR in the rest of this chapter) computation, a dedicated CORR accelerator cannot be used for the sole purpose of CENTER computations, as it lacks communication logic for this intermediary result. However, designers may want this capability for small post-processing, semi-specific IPs. This raises the following question: “What is the area of a programmable

Benchmark	Cycles/Pb	Operators	DSP
CENTER	8343	1±, 1/	2
CENTERx10	834	10±, 10/	20
10xCENTER	8343	1±, 1/	2
CGP-CENTERx10	5744	2±, 1/	4
CGP-CENTER-inf	4096	2±, 1/	4

Table 8.1: Performance and area metric for coarse-grained pipeline (CGP) vs coarse grained replication (CGR) of CENTER accelerator (matrices of size 64×64, FP16 data type)

accelerator capable of executing arbitrary subproblems of CORR?" In this chapter, we present a DSP-efficient answer to this problem.

CORR can be decomposed into several computations, corresponding to the loop nests that a programmer would write when designing an HLS accelerator:

- CENTER: $X^C_{ij} = X_{ij} - (\sum_{i'} X_{i'j})/n$
- STDDEV: $\sigma^X_j = \sqrt{\sum_i (X^C_i)^2/n}$
- CENTER-REDUCE: $X^{CR}_{ij} = \left(X_{ij} - \sum_{i'} X_{i'j}\right) / (\sigma^X_j \cdot \sqrt{n})$
- T-MATMULT: $(X^{CR})^t \cdot X^{CR}$

The naive approach consists in the juxtaposition of fixed-functions dedicated to each of the individual problems, without hardware reuse at all. Though this collection of dedicated accelerators does not need any interconnection between its sub-accelerators, its efficiency is far from being optimal. Indeed, a naive HLS-designed accelerator composed of the 4 primitives mentioned above would use 14 DSP when each accelerator is (individually) optimized for efficiency (SUM-AREA), and 36 DSP for a throughput-oriented one (SUM-THR), as illustrated in Tbl. 8.3. On the opposite, our kernel merging approach focus on heavy reuse of DSP units and clear separation of the compute units. This allow the integration of 1-DSP FMA units, superior in density to the HLS primitives that implements FMAs with 4 DSP (2 for the multiplication and 2 for the addition). Furthermore, the GA supports the computation of any CORR sub-problem using only 3 DSPs, with support of batching factor up to 3 without significant degradation of the total execution time of CORR instances. Followingly, our approach is able to outperform dedicated designs in terms of performance per area, measured by the FLOP per cycle per DSP metric, as illustrated in Tbl. 8.2 (see Sec. 8.4 for more details). As a reference, ScaleHLS [77] achieves similar DSP-efficiency, as explained in Sec. 8.4.3.

To understand these gains, let us detail the specificities of the CORR accelerator. Assuming N is the size of the input matrix, only T-MATMULT is computed in $O(N^3)$ operations, the others being computed in $O(N^2)$. Furthermore, T-MATMULT uses only additions and multiplications, which means that the majority of the time will be spent using these units on a dedicated accelerator: sharing them will only lead to marginal gains. However, CENTER, STDDEV and CENTER-REDUCE also require the use of a

Design	Cycles (CORR)	DSP	Nb of + and $*$	Nb of $\sqrt{\cdot}$ and /
SUM-AREA	291221	**14**	4 and 3	3 and 2
SUM-THR	144614	**36**	10 and 8	3 and 2
3xCORR-AREA	291221	**12**	3	3
GA-CORR	320603	**3**	3	1

Table 8.2: Performance and area metric for coarse-grain pipelined correlation, sum accelerator and dedicated accelerator

	SUM-AREA	SUM-THR	3xCORR-AREA	GA-CORR
FLOP/C/DSP	0.134	0.105	0.314	**1.256**

Table 8.3: Performance per area metric for coarse-grain pipelined correlation, sum accelerator and dedicated accelerator

division and a square root operator, which can be shared between independent batched executions of CORR. This lead to significant area gains over the traditional coarse-grain replication strategy by avoiding unnecessary replicas of low-usage units, i.e. division and square root operators, with minimal impact on overall latency.

This time, the kernel merging approach allows us to mix both sharing and replication: we replicate the MATMULT accelerator to keep minimal impact of the sharing on the overall execution time. This leads to an increase of the execution time of 10 % compared to a basic dedicated accelerator while saving the area of two compute units per replica (/ and $\sqrt{\cdot}$).

Furthermore, we enrich the accelerator with additional data routing capabilities to become more versatile: depending on a user configuration, any sub-computations of Correlation can be computed, hence our "generic" naming. Area and execution time of the GA-CORR3 (generic accelerator capable of executing a 3-batched Correlation) compared to a simple non batched, non-generic accelerator is reported in Tbl. 8.2. As expected, the generic batched accelerator is able to provide a reduction of 66 % of the number of divider and square root units, no change in terms of adders / multipliers, to the cost of a 10 % increase in execution time.

8.1.3 Accelerator Creation and Usage Workflow

The steps used to create the Generic accelerators are illustrated Fig. 8.3. The input applications, broken down into elementary kernels, are used to create Functional Units. Then, the generated FUs are selected and replicated following FPGA-specific constraints (e.g. available area and resource consumption of FUs) in the fixed accelerator structure. Finally, an HLS synthesis tool produces the output bitstream as well as C drivers for basic FPGA interaction.

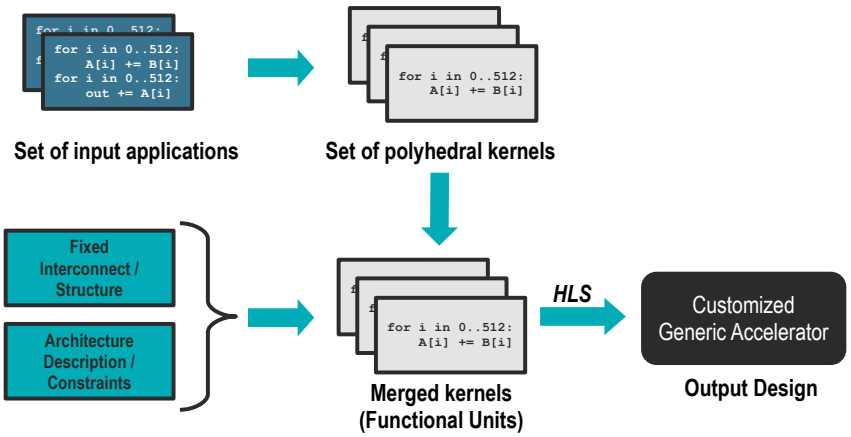


Figure 8.3: Workflow of the creation of the accelerator

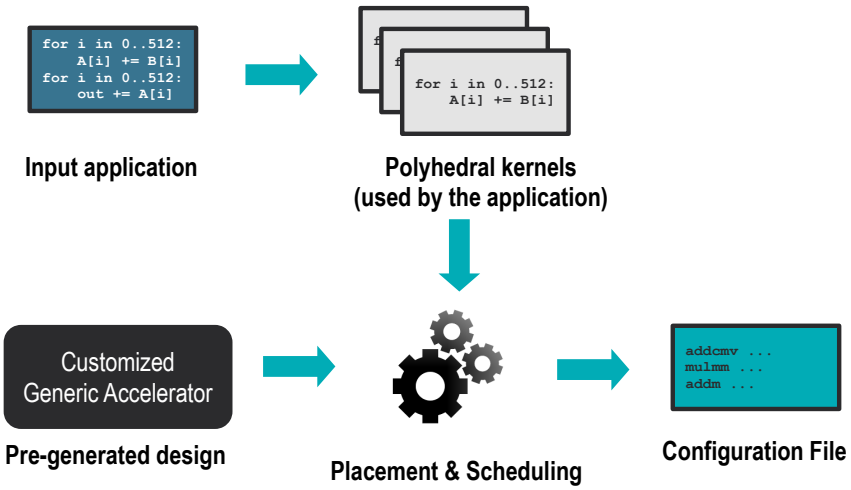


Figure 8.4: Workflow of the usage of the accelerator

To use an already generated GA for one of its compatible applications, a configuration file containing the sequence of kernels must be generated. Its generation wokflow, relying on the previously generated design, is detailed Fig. 8.4. The application is first broken down into kernels; then, these kernels are mapped and scheduled to the existing FUs given their number and supported primitives.

```
1 // Kernel 1
2 for (i = 0; i < N; ++i)
3   for (j = 0; j < N; ++j)
4       C[i][j] = beta * C[i][j]; // S1
5 // Kernel 2
6 for (i = 0; i < N; ++i)
7  for (j = 0; j < N; ++j)
8    for (k = 0; k < N; ++k)
9       C[i][j] += alpha * A[i][k] * B[k][j];
```

Figure 8.5: Example: General Matrix Multiplication, split in two kernels

8.2 Kernel Merging for Multi-Functionalities

We now present our approach of building a multi-functionality accelerator. First, we deduce from each input application a set of *polyhedral kernels*, each computing a particular functionality. Then, we apply *kernel merging* to generate a family of multi-functionality Functional Units, and we finally select and replicate some of these FUs depending on our ad-hoc profitability criteria.

8.2.1 Polyhedral Kernel Representation

In this work, a kernel is a polyhedral program; that is, a program with a static control-flow (every branch taken in the code can be exactly predicted at compile-time, independently of the value of the data computed on). In addition, polyhedral programs must be described exactly using only affine functions of the surrounding loop iterator and parametric constants. Three structures are used to describe such programs: for each statement S, we define their *iteration domain* $\mathcal{D}_S$, which describe the set of all dynamic executions of the statement, each identified by the vector of values that the surrounding loop iterators take when it executes (that is, the iteration vector $\vec{x}_S$); their *access functions* which maps every iteration vector to the specific memory location(s) accessed by that instance; and a *scheduling function* Θ_S which maps every iteration vector to multidimensional timestamp $\vec{t}$, such that in the transformed code, the iteration vectors are executed in the lexicographic order of their timestamps [94, 65]. We note $t = [t_1, t_2, ...] \in \mathcal{T}$ where $\mathcal{T}$ is the schedule space.

We illustrate with the two kernels below in Fig. 8.5, where, for the sake of illustration, we decomposed a classical GEMM kernel into two kernels.

The iteration domain of Kernel 1 (K1) is $\mathcal{D}_{K1} : \{[i,j] : 0 \leq i < N \text{ and } 0 \leq j < N\}$, and Kernel 2 (K2) is $\mathcal{D}_{K2} : \{[i,j,k] : 0 \leq i < N \text{ and } 0 \leq j < N \text{ and } 0 \leq k < N\}$. The access functions of K1 include $Read_{K1} : \{[i,j] \mapsto C[x,y] : x = i \text{ and } y = j\}$ and K2 includes $Read_{K2} : \{[i,j,k] \mapsto A[x,y] : x = i \text{ and } y = k\}$. The original schedule of K1 is $\Theta_{K1} = \{\vec{x}_{S1} = [i,j] \mapsto [t_1,t_2,t_3,t_4,t_5] : t_1 = 0 \text{ and } t_2 = i \text{ and } t_3 = 0 \text{ and } t_4 = j \text{ and } t_5 = 0\}$, that is a $2d+1$ encoding of the schedule, for a loop depth d [95, 96].

```
1 for (i = 0; i < N; ++i)
2     for (j = 0; j < N; ++j)
3         C[i][j] = 0; // S1
4     for (j = 0; j < N; ++j)
5         for (k = 0; k < N ;
    ++k)
6             C[i][j] +=
    A[i][k]  * B[k][j]; // S2
```

(a)

```
1 for (i = 0; i < N; ++i)
2     for (j = 0; j < N; ++j)
3         C[i][j] =
    (i==0)?0:C[i-1][j]; // S1
4     for (j = 0; j < N; ++j)
5         for (k = 0; k < N ;
    ++k)
6             C[i][j] +=
    A[i][k] * B[k][j]; // S2
```

(b)

Figure 8.6: Example of programs with trivially kernelisable (a) and non-trivially kernelisable (b)

More generally, for a polyhedral program $\mathcal{P}$ composed of n statements $(Si)_{1\leq i\leq n}$ we define $\Theta_{\mathcal{P}}$ the schedule of a complete program by the statement-wise collection of schedules:

$$\Theta_{\mathcal{S}} \ : \begin{cases} \cup_{Si\in\mathcal{P}}\mathcal{D}_{S_i} \to \mathcal{T} \\ x_{Si} \mapsto \Theta_{Si}(x_i) \end{cases}$$

8.2.2 Decomposition of Applications into Kernels

The Generic Accelerator operates at the *kernel* granularity, whereas all polyhedral programs are not directly written as a sequence of kernels.

Let us restrict the applications that can be mapped to a GA to programs without loop-carried dependency between statements of different loop nests, as illustrated in Fig. 8.6. For a fixed $i \neq 0$, in Fig. 8.6a, $S1$ reads no value, whereas $S1$ in Fig. 8.6b reads $\{C[i-1][j] \ : \ 0 \leq j < n \}$, which is written by $S2$. Formally, we defined the class of *trivially kernelisable* programs as follows.

Definition 8.2.1 (Trivially Kernelisable Program)**.** *Let $S1$ and $S2$ be two statements of a polyhedral program $\mathcal{P}$ belonging to two different loop nests.*

Using the former notations of a $2d+1$ encoded schedule, let c be the inner-most common scheduling dimension of $S1$ and $S2$, and t_c the corresponding value in the $2d+1$ schedule. Without loss of generality, we assume for any schedule prefix $(t_1, ..., t_c)$ that there exists a iteration vector prefix $(x_1, ..., x_{c'})$ such that:

$$\begin{cases} \forall \vec{x}_{S1} = (x_1, ..., x_{c'}, x^{S1}_{c'+1}, \ ...), \exists (t^{S1}_{c+1}, t^{S1}_{c+2}, ...) \ s.t. \ \Theta_{\mathcal{P}}(\vec{x}_{S1}) = [t_1, ..., t_c, t^{S1}_{c+1}, ...] \\ \forall \vec{x}_{S2} = (x_1, ..., x_{c'}, \ x^{S2}_{c'+1}, ...), \exists (t^{S2}_{c+1}, t^{S2}_{c+2}, ...) \ s.t. \ \Theta_{\mathcal{P}}(\vec{x}_{S2}) = [t_1, ..., t_c, t^{S2}_{c+1}, ...] \end{cases}$$

We also assume that for all $\vec{x}_{S1}$ and $\vec{x}_{S2}$ such vectors and a fixed t_c, $t^{S2}_{c+1} > t^{S1}_{c+1}$, that is, the loop nest of $S2$ is executed after the one of $S1$.

```
1 for (i = 0; i < N; ++i)
2     for (j = 0; j < N; ++j)
3         C[i][j] = 0; // S1
4     for (j = 0; j < N; ++j)
5         for (k = 0; k < N ;
    ++k)
6             C[i][j] +=
    A[i][k]  * B[k][j]; // S2
```

(a)

```
1 for (i = 0; i < N; ++i)
2     for (j = 0; j < N; ++j)
3         C[i][j] = 0; // S1
4 for (i = 0; i < N; ++i)
5     for (j = 0; j < N; ++j)
6         for (k = 0; k < N ;
    ++k)
7             C[i][j] +=
    A[i][k] * B[k][j]; // S2
```

(b)

Figure 8.7: Example of loop fission: corresponding MatMult code before (a) and after (b) transformation

$\mathcal{P}$ *is* trivially kernelisable *if:*

$$\forall S1 \neq S2, \forall t, \quad \left(Read_{S1}^{|t_c=t} \cup Write_{S1}^{|t_c=t}\right) \cap \left(\bigcup_{i=0}^{t-1} Write_{S2}^{|t_c=i}\right) = \emptyset$$

With:

- $Read_{S1}^{|t_c=t}$ *the set of values read by* $S1$ *for all* $\vec{x}_{S1} \in \mathcal{D}_{S1}$*, that is*

$$Read_S^{|t_c=t} = Read_S\left(\Theta_{\mathcal{P}}^{-1}\left(\{[t_1, ..., t_{c-1}, t, t_{c+1}, ...]\}\right)\right)$$

  Note that we implicitly extend $Read_{S1}$ *by* $Read_{S1} = \emptyset$ *if* $\Theta_{\mathcal{P}}^{-1}(\{\vec{t}_{S1}\}) \cap \mathcal{D}_{S1} = \emptyset$*.*

- *Similarly,* $Write_S^{|t_c=i}$ *is defined as*

$$Write_S^{|t_c=t} = Write_S\left(\Theta_{\mathcal{P}}^{-1}\left(\{[t_1, ..., t_{c-1}, t, t_{c+1}, ...]\}\right)\right)$$

  With a similar implicit extension of $Write_{S2}$*.*

Given a trivially kernelisable program, our goal is to break it into kernels, that is, a sequence of perfect loop nests. For that, we must apply *loop fission* (illustrated in Fig. 8.7), a transformation isolating statements of imperfectly nested loops in their own loop nest.

Definition 8.2.2 (Loop Fission)**.** Loop fission *is a transformation of the schedule of a polyhedral program breaking a loop nest into two loop nests of same iteration domain, each one enclosing a different part (statement) of the original loop body.*

Reusing the former notations, when $S1$ *and* $S2$ *are two consecutive statements* ($t_{c+1}^{S2} = t_{c+1}^{S1} + 1$)*, we define* $\Theta_{\mathcal{P}}^{f}$ *one schedule resulting of the fission of* $S1$ *and* $S2$*, such that:*

$$\Theta^f_{\mathcal{P}} = \begin{cases} \vec{x}_{S1} \mapsto \Theta_{\mathcal{P}}(\vec{x}_{S1}) = [t_1, ..., t_{c-1}, t, t^{S1}_{c+1}, ...] & \text{if } \vec{x}_{S1} \in \mathcal{D}_{S1} \\ \vec{x}_{S2} \mapsto \Theta_{\mathcal{P}}(\vec{x}_{S2}) + [1, 0, ...] = [t_1 + 1, ..., t_{c-1}, t, t^{S2}_{c+1}, ...] & \text{if } \vec{x}_{S2} \in \mathcal{D}_{S2} \\ \vec{x} \mapsto \Theta_{\mathcal{P}}(\vec{x}) + [1, 0, ...] & \text{if } x \notin \mathcal{D}_{S1} \cup \mathcal{D}_{S2} \text{ and } \Theta_{\mathcal{P}}(\vec{x}) > \min_{\vec{x}_{S1} \in \mathcal{T}} \Theta_{\mathcal{P}}(\vec{x}_{S1}) \\ \vec{x} \mapsto \Theta_{\mathcal{P}}(\vec{x}) & \text{otherwise} \end{cases}$$

In the general case, such a transformation is illegal, as it may break dependencies.

Theorem 8.2.1. *Any splitting of two different statements of a trivially kernelisable program is legal.*

Proof. Let $S1$ and $S2$ be two different statements and d be a (RaR or WaW) dependence between $S1$ and $S2$, we note $d = \vec{x}_{S1} \longrightarrow \vec{x}_{S2}$ (producer to consumer). Let x_c be the outer-most common dimension of $\vec{x}_{S1}$ and $\vec{x}_{S2}$, corresponding to t_c in the original schedule.

Without loss of generality, we assume $\Theta_{\mathcal{P}}(\vec{x}_{S2}) > \Theta_{\mathcal{P}}(\vec{x}_{S1})$. Let $\Theta^f_{\mathcal{P}}$ be the schedule after fission of $\mathcal{P}$ between Sa and Sb.

We note $\preccurlyeq$ the order relation of statements defined by $S \preccurlyeq S' \Leftrightarrow x^S_{c+1} \leq x^{S'}_{c+1}$, corresponding to the *syntactic order* of loops in the original program.

- If $S2 \preccurlyeq Sa$, then $\Theta^f_{\mathcal{P}}(\vec{x}_{S2}) = \Theta_{\mathcal{P}}(\vec{x}_{S2})$ and $\Theta^f_{\mathcal{P}}(\vec{x}_{S1}) = \Theta_{\mathcal{P}}(\vec{x}_{S1})$, so $\Theta_{\mathcal{P}}(\vec{x}_{S1}) < \Theta_{\mathcal{P}}(\vec{x}_{S2})$ as $\Theta_{\mathcal{P}}$ is a valid schedule, and d is respected.
- If $S1 \succcurlyeq Sb$, then $\Theta^f_{\mathcal{P}}(\vec{x}_{S1}) = \Theta_{\mathcal{P}}(\vec{x}_{S1}) + [1, 0, ...]$ and $\Theta^f_{\mathcal{P}}(\vec{x}_{S2}) = \Theta_{\mathcal{P}}(\vec{x}_{S2}) + [1, 0, ...]$, so $\Theta_{\mathcal{P}}(\vec{x}_{S1}) < \Theta_{\mathcal{P}}(\vec{x}_{S2})$ as $\Theta_{\mathcal{P}}$ is a valid schedule, and d is respected.
- If $S1 = Sa$ and $S2 = Sb$, then $\Theta^f_{\mathcal{P}}(\vec{x}_{S1}) = \Theta_{\mathcal{P}}(\vec{x}_{S1})$ and $\Theta^f_{\mathcal{P}}(\vec{x}_{S2} = \Theta_{\mathcal{P}}(\vec{x}_{S2}) + [1, 0, ...]$, so $\Theta_{\mathcal{P}}(\vec{x}_{S1}) < \Theta_{\mathcal{P}}(\vec{x}_{S2})$ and d is respected.

□

The complete flow of the decomposition of a polyhedral program into kernels is given in Alg. 11. Checking whether an input program is trivially kernelisable can be done using a dependence analysis software such as [97], by checking for each pair of statement $S \preccurlyeq R$, for all l loop level and p pair of references the emptiness of the dependence polyhedrons $\mathcal{D}_{S\delta R,l,p}$ as defined in [98].

If the program fails this step due to write-after-write dependencies on the input program, then a conversion to a Dynamic Single Assignment form using [99] can be performed, suppressing the former to result in a trivially kernelisable program. Then, we perform complete loop fission of the program (legal thanks to Thm. 8.2.1), and output as kernels all individual perfectly nested loops of the converted program.

8.2.3 Kernel Set and Workloads

Given a set of polyhedral kernels that are candidates to merges, we aim to execute workloads that are arbitrary compositions (in sequence or in parallel) of calls to these

```
   Input: A polyhedral program 𝒫
 1 if is_trivially_kernelisable(𝒫) then
 2 |  𝒫′ := 𝒫;
 3 else
 4 |  𝒫′ := convert_to_DSA(𝒫);
 5 |  if ¬ is_trivially_kernelisable(𝒫′) then
 6 |  |  fail;
 7 |  end
 8 end
 9 for all S statement of 𝒫′ do
10 |  𝒫′ := apply_loop_fission(𝒫′, S);
11 end
12 return the set of all loop nests of 𝒫′
```

Algorithm 11: Kernel decomposition of a trivially kernelisable program

kernels. These computations can be captured by a simple language for straight-line programs, which is then trivially amenable to compilation, to extract a a forest of directed acyclic graphs, where each node represents one kernel call. We assume each kernel represents a pure function, and summarizes its functionality as follows.

Definition 8.2.3 (Kernel representation)**.** *Given a kernel K, we define its functionality as the signature of the kernel augmented with the loop bounds, for each loop:*

$$K : input_1, ..., input_n, N_1, ..., N_m \rightarrow output_1, ..., output_p$$

We also define Ops_K the set of arithmetic operations executed by K.

For example, the complete signature of $K2$ is

$$K2 : C[N][N], A[N][N], B[N][N], alpha, N, N, N \rightarrow C[N][N]$$

where $Ops_{K2} = \{+, *, *\}$. A workload in the present work can be modeled as a straight-line program, such that (a) temporary variables are allowed; (b) there is a single kernel call per instruction; (c) type and size analysis for every input/output passed as argument to the program kernels succeeds, given the signatures of every kernel. Focusing on (dense) linear algebra, high-level expressions can be written in this simple form, which is then compiled to obtain a sequence of kernel calls implementing this program. Parallelism between kernel calls is automatically detected from the DAGs, creating "batches" of calls when possible from the input workload, simply recognizing parallelizable operations by computing the earliest schedule of each node in the DAGs.

We illustrate with the simple following program composed of 4 instructions represented Fig. 8.8, that is a valid input to our system. For clarity, `K1` is renamed to `MatScale`, and `K2` is renamed to `MatMulScaleA`. In our prototype implementation, supported vari-

```
1 TMP1[N][N] := MatScale(C1[N][N],42,N,N)
2 TMP2[N][N] := MatMulScaleA(TMP1[N][N],A[N][N],B[N][N],51,N,N,N)
3 TMP3[N][N] := MatScale(C2[N][N],43,N,N)
4 TMP4[N][N] := MatMulScaleA(TMP3[N][N],A[N][N],B[N][N],52,N,N,N)
```

Figure 8.8: Example of a 4 instructions input program

Input: $\mathcal{K}$ a set of kernels, $\mathcal{D}$ its DAG of dependencies, $\mathcal{F}$ a set of FUs
Function *make_largest_insn($\mathcal{K}$, $\mathcal{D}$, $\mathcal{F}$)*
  ins := [ ];
  scheduled := $\emptyset$;
  for $f = 0$ *to* $\#FU - 1$ **do**
    ins.append();
    for k *in* $\mathcal{K}$ **do**
      if *is_compatible(k,f) and do_not_depend(k, scheduled, $\mathcal{D}$)* **then**
        ins.append(k);
        scheduled.add(k);
        break;
      end
    end
  end
  merge_nodes(scheduled, $\mathcal{D}$);
  return *ins*;
end

Algorithm 12: Selection of the next macro-instruction

able types are scalars, 1D arrays (vectors) and 2D arrays (matrices), which should all be of the same data type.

This program may be input by the user, and is then compiled to a sequence of "instructions" to be executed by the accelerator. As described in Sec. 8.3, the accelerator executes a stream of instructions given as input, where each instruction contains the name of the kernel to invoke, which hardware unit it must be placed, and the operands/loop bound information as in the example above. The order of execution follows the order of instructions sent to the accelerator. A simple compilation step creates this sequence of instructions from the input program above.

This analysis delivers the set of calls to be executed as their earliest start time (assuming each call takes 1 time quantum), e.g. `MatScale:0,0` and `MatMulScaleA:1,1` giving explicitly the number of calls (i.e., the number of entries per kernel name) and the parallelism opportunities (i.e., all calls at the same time step can be executed in parallel). In our current implementation, we weight timesteps by their iteration latency, and a simple greedy placement of the calls on the available hardware units is implemented, illustrated in Alg. 13. Decomposition of the application into kernels is performed using a similar algorithm than Alg. 11, but also gathering dependencies between statement to create the DAG of kernel execution $\mathcal{D}$. Then, we sort the kernel list by their latency, solving ties by

Input: $\mathcal{P}$ a polyhedral program, $\mathcal{F}$ a set of FUs
Output: Insn, a sequence of macro-instructions
1 $\mathcal{K}$, $\mathcal{D}$ = decompose_kernels (A);
2 sort($\mathcal{K}$);
3 Insn := [ ];
4 **while** $\mathcal{K} \neq \emptyset$ **do**
5 | Insn.append(make_largest_insn($\mathcal{K}$, $\mathcal{D}$, $\mathcal{F}$));
6 **end**
7 sort(Insn);
8 **return** (Insn);

Algorithm 13: Placement of an application on an existing Generic Accelerator

minimizing ASAP scheduling timestamp difference (without resource limitation) between consecutive elements.

Then, we greedily build macro-instructions of the GA by packing operations of largest execution time and modify $\mathcal{D}$ to ensure legality of the dependencies (Alg. 12). Indeed, once an instruction has been generated, the corresponding nodes are merged in the DAG, creating new dependencies that prevent illegal scheduling, that is, cycles in the dependence graph of macro-instructions. Finally, we perform a topological sort to the set of macro-instructions generated in an out-of-order manner to create a legal scheduling of the complete program.

Therefore, the problems to be addressed when designing the accelerator include (a) how many *parallel instances* of each kernel should be possible? And (b) which operations (`+`,`*`, etc) may be shared between kernels?

8.2.4 Kernel Merging

We now outline our high-level method for generating a semantically correct perfectly nested loop structure, that capture the set of all functionalities to be implemented by the accelerator. We leverage polyhedral program analysis and transformations [94] to create such code structure.

Iteration domain extension

The first operation is to normalize all kernels so that every statement is represented by an iteration domain of identical, maximal dimensionality across all kernels, while preserving the semantics. This amounts to computing a maximal common loop embedding, and statement perfectization [77] is an instance of such transformation. Specifically, we first compute $maxd$ the maximal dimensionality of all iterations domains to be merged: $maxd = max_{K \in kernels} dim(\mathcal{D}_K)$. Then, for every kernel whose dimensionality is less than $maxd$, we create $\mathcal{D}_K^{ext} = Universe(maxd) \cap \mathcal{D}_K \cap oneiterdims(K)$, where $Universe(x)$ builds the infinite/unbounded polyhedron of dimensionality x, and $oneiterdims(K)$ is the lexicographic minimum of every dimension in $maxd$ that is not a dimension in $\mathcal{D}_K$. For example, we would get: $\mathcal{D}_{K1}^{ext} : \{[i,j,k] : 0 \leq i < N \ and \ 0 \leq j < N \ and \ k = 0\}$.

We then further extend the iteration domains systematically with one additional dimension: *kid*, which represents the unique ID of a kernel that is merged. For our example, assuming Kernel1 (K1) identifier is 1, and K2's is 2, we get: $\mathcal{D}^{ext}_{K1} : [K1] \rightarrow \{[kid, i, j, k] : 0 \leq i < N \; and \; 0 \leq j < N \; and \; k = 0 \; and \; kid = K1\}$.

Scheduling for fusion and pipelining

The next operation builds the union of all extended iteration domains into a single polyhedral program, by building a schedule for fusion. This schedule merges all loop levels, and only separate kernels at the inner-most loop level. For example, the short notation for Θ_{K2} is $\{[i, j, k] \rightarrow [0, i, 0, j, 0, k, 0]\}$. The schedules merging K1, then K2, are simply their original identity schedule (possibly extended to *maxd*), where we use the kernel id to compute the last schedule dimension, for every statement in each kernel. We have $\Theta_K : [K] \rightarrow \{[kid, i, ..., m] \rightarrow [0, i, 0, ..., 0, m, kid]\}$ if the kernel contains a single statement, otherwise *kid* needs to be extended to model the unique id of every statement in the kernel instead, in their order of execution, such that for every kernel and every statement *kid* is globally unique.

For example, to fuse K1 with K2 we would get $\Theta^{ext}_{K1} : [K1] \rightarrow \{[kid, i, j, k] \rightarrow [0, i, 0, j, 0, k, kid] : kid = K1\}$, and $\Theta_{K2} : [K2] \rightarrow \{[kid, i, j, k] \rightarrow [0, i, 0, j, 0, k, kid] : kid = K2\}$. However, further modifications of the schedule may be implemented: in particular, *loop permutation* may be employed to implement fine-grain parallelism when possible, as discussed below in Sec. 8.2.5, for example $\Theta_{K2} : [K2] \rightarrow \{[i, j, k] \rightarrow [0, i, 0, k, 0, j, kid] : kid = K2\}$ that permutes the k and j loops to expose a synchronization-free inner-parallel loop if possible.

Controlling separation

The final operation is to actually generate the candidate loop nest, by using polyhedral code generation [94]. Intuitively, CLooG [94] generates a code that scans the iteration domains in the lexicographic order of the timestamps computed by the Θ functions. A key aspect of performance for the generated codes is to implement *separation* along every loop dimension, that is the process of grouping iterations of the loop as a function of the specific set of statements to be executed. For example, along the k loop, at iteration 0 both K1 and K2 execute, but at iteration > 0 only $K2$ executes. In this work, we aim to push conditionals that guard the execution of a statement to the inner-most loop level, therefore we simply turn off separation in CLooG, to obtain the code illustrated in Fig. 8.9.

8.2.5 Profitability Criteria

While any set of polyhedral programs can be merged with the procedure above, not all such programs are candidate for *efficient* acceleration, and may not benefit from being merged with other kernels. However, the profitability criteria can be expressed as the result of polyhedral analyses on the set of kernels.

```
for (i = 0; i < N; ++i)
 for (k = 0; k < N; ++k)
   for (j = 0; j < N; ++j) {
      if (KER == K1 && k == 0)
         C[i][j] = beta * C[i][j]; // S1
      if (KER == K2)
         C[i][j] += alpha * A[i][k] * B[k][j];
   }
```

Figure 8.9: Example code structure of a two merged kernels

Pipelining

A central objective is to enable coarse-grain pipelining across kernels. Therefore, we model a criterion for making pipelining *possible* (otherwise no pipelining is implemented), that eventually drives the loop order: the inner-most loop should be such that either there is no loop-carried dependence (LCD) along it for the kernel, or if there is a LCD, the distance must be constant, and greater than the expected iteration latency (for one iteration of the inner-most loop). The final loop permutation for the program is computed such that we minimize dependences satisfied by the inner-most loop level in the merged program, using only loop permutations as the possible transformations. We simply compute all possible loop permutations for the merged loop nest, and for each case compute whether the inner loop is parallel. If this system has no solution, we relax it to enable LCD for the inner-most loop level if and only if the dependence distance is greater than the iteration latency for the statement.

Exposing Functional Units

A kernel can be viewed as the actual computation statement(s) associated with it, along with their iteration domain. As we generate a fused loop nest, all statements share the same unique loop nest implemented in hardware to iterate them. Therefore, two parallel instances of a kernel can be implemented by simply replicating the *statement(s)* in the inner-most loop. We call such hardware instances implementing a statement a *functional units* (FUs), and we aim to select how many instances of each functional unit should be implemented in the accelerator. We note that depending on the kernels being merged, syntactically identical statements (after variable renaming) may occur: in this case two functional units may compute exactly the same operations, albeit perhaps with different iteration domains. This can be easily detected from the kernels representations, and we merge into a single FU those computing identical operations, to facilitate solving the optimization problem below. Note in our simple compilation phase to convert the input straight-line program to a sequence of instructions, we exploit the fact that multiple kernels/functionalities may be mapped to the same FU, perhaps by adjusting the loop bounds passed as argument to the instruction, to find a compact placement and schedule.

Number of replications

The key challenge is to determine the number of replications of each kernel/functionality, given that (a) different workloads may expose vastly different amount of parallelism; and (b) the number of elementary operation(s) that can be shared between kernels is related to the number of replications of each kernel. Our objective is to optimize throughput per area, in other words, we aim to increase resource sharing without performance penalty, something typically achievable when units would anyway be otherwise idling due to sequences of dependent kernel calls.

For a particular workload summarized as the number of calls to each kernel, we aim to minimize the expected execution time under resource constraints, summarized in the following optimization problem:

$$\begin{aligned} \text{minimize} \quad & \sum_{K \in Kernels} \lceil \#calls(K)/\#FU(K) \rceil * card(\mathcal{D}_K) * IL_K \\ \text{subject to} \quad & \sum_{i \in FUs} Area(FU_i) * \#FU_i < \text{max_area} \end{aligned}$$

Where a FU, or Functional Unit, is the *hardware implementation* of the operations in a kernel K, and IL_K the iteration latency to execute one inner-most loop iteration, that is the latency of the FU to execute once. The unknown to be computed is the number of FU, for each FU type. The workload mix, given by the number of calls to each kernel/-functionality, is an input of this optimization problem. As we weight the latency of an iteration by the cardinality of its iteration domain, in case of an heterogeneous workload combining N^2 (e.g., matrix addition) and N^3 (e.g., matrix-multiplication) operations, the dominant cost driving the solution found will be the latter. $Area(FU)$ is computed by approximating the DSP consumption of a FU, itself adjusted if operations in a FU can be shared across multiple FUs: they are of the same type, and can execute in pipelined fashion. In practice, to solve this problem we simply enumerate all solutions (i.e., number of FUs of each type) and for each, we compute its latency and resources. We output the first solution that meets resource constraints with minimal latency.

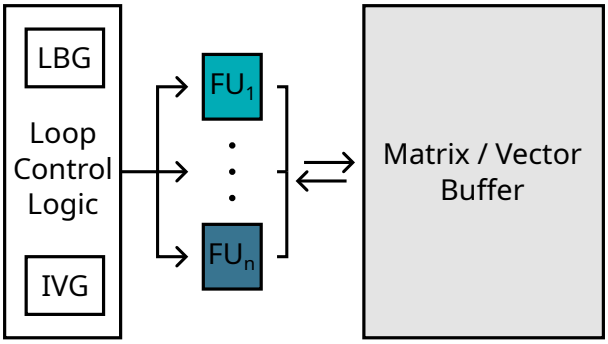


Figure 8.10: Layout of the Generic Accelerator

8.3 Accelerator Implementation

In this section, we analyze the modules that compose the accelerator and discuss their implementation.

8.3.1 Structure of the Accelerator

The accelerator layout is illustrated Fig 8.10, and is orchestrated around a single pipelined loop dispatching user-specified computing tasks to Functional Units (FUs), corresponding to the loop bodies of the merged kernels. The execution of one kernel is decomposed in four steps: computation of the read/write locations as function of the current loop iterator, loading of the data, actual computation, storage of the produced output. Each of these stages is executed by one of the three accelerator submodules: loop control logic (dispatch of the operation to the FU), FUs (execution), local buffer (data storage and access).

While our accelerator architecture is designed for the Xilinx Ultrascale+ MPSoC [100], that is, a FPGA integrated with a CPU, none of its features depends on the CPU. Therefore, some implementation details may be specific such as the communication-handling logic and the wrapping application mentioned below, but they do not limit the genericity of our approach.

8.3.2 Iteration Vector Generator (IVG)

Though merged kernels are transformed to iterate on the same space, additional logic is still needed to convert the global (scalar) loop index to the iteration vector given as input to the FU – typically indexes of accessed arrays. This computation is done by the IVG, that initializes the iteration vectors to 0_d and update them using their former value through a fixed state machine.

8.3.3 Functional Units

Functional Units (FU) are specialized, fully pipelined units capable of executing one or more operations of the input kernels such as addition, multiplication, data movement (e.g. for transposition), etc. They take as input the iteration vector, load directly values

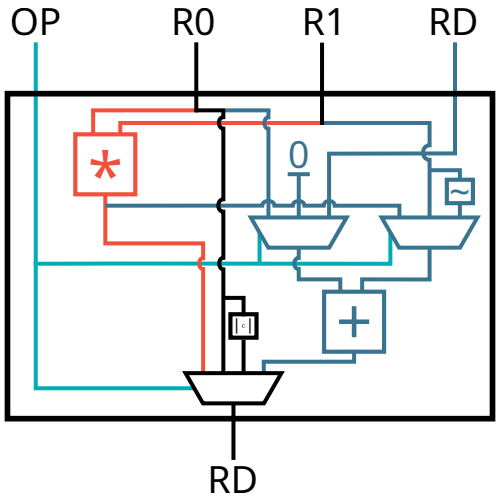


Figure 8.11: Anatomy of an HLS-generated FU: control path in light blue, multiplier path in red and sub/adder path in dark blue

from the local buffer depending on the current iteration vector value, and compute one or more outputs, which are directly stored back to the local buffer. A FU can expose several computation capabilities, e.g. matrix addition, FMA, but uses internally pre-generated compute units optimized to the target technology. We propose two implementations for FPGA FUs relying on FMA, additions, subtraction or multiplications:

- One relying on *Vivado Block Design Floating Point primitives* configured for maximal efficiency. This primitive provides an 1-DSP FP16 FMA unit that can be reused for any scalar operation (addition, multiplication or subtraction), leading to a maximum throughput per area of *2 FLOP/Cycle/DSP*. Internally, these units are connected to the accelerator using AXIStream interfaces at the Vivado block design level, which also increase the cost of the interconnect.
- One relying on *Vitis HLS Floating Point primitives*. With this technique, an 16-bit FMA is decomposed as a multiplier (2-DSP) and an adder (2-DSP) cascaded. Therefore, the maximal achievable throughput-per area is only *0.5 FLOP/Cycle/DSP*, while reducing interconnect costs by removing the need of AXIStream interfaces. Its structure is illustrated Fig. 8.11, and can be configured to be an FMA (using both an adder and a multiplier) or a single adder/multiplier to save DSPs.

Furthermore, we allow further sharing by adding logic to flip the sign bit of one of the FP16-encoded operand in order to perform $a + (-b)$ with no additional DSP cost as well as absolute value computation.

As FUs are fully pipelined, they must have no loop-carried dependence: the minimal reuse distance of read-after-write dependence has to be higher than the latency of the complete FU. In the case of reduction, we store the intermediary results in a separated scalar buffer, and stall the pipeline while waiting for the end of each operation.

8.3.4 Loop Bound Generator (LBG)

Before the execution of any kernel, the LBG scans all scheduled kernels and computes the cardinality of the minimal iteration space by maxing their iteration space sizes. This value is then used as the trip count of the FU-scheduling loop.

8.3.5 Loop Control Logic

The accelerator is organized around a single loop defined in HLS-C++ as a `for` ranging from 0 to a maximum value given by the LBG. This loop corresponds to a flattened version of the fully merged loops of all accelerated kernels, and is pipelined to achieve a maximum throughput of 1 execution of all FUs per cycle, i.e. to fully exploit all FUs.

The role of the loop control logic is twofold. First, it schedules operations on the FU, and second, it iterates over all merged kernels to ensure a correct and complete execution of the input workload.

8.3.6 Off-Chip Communications

Data in the local buffer are coalesced [7] for transfers into 64-bit packets that are sent or received together in one burst to/from the off-chip DRAM before and after execution of the accelerator. Execution time is measured by an on-chip counter wired to the main clock, whose value is fetched before the execution of the computation and right after its termination (i.e. not including communications). The local buffer communicates with off-chip memory using the AXI4 bus, connected to one high-performance communication port of the Zync MPSoC. Its setup is controlled by MMIO-mapped registers using an AXILite bus, managed by a wrapper C++ application running on CPU integrated in the ZCU104 MPSoC. This application also handles the execution flow as well as memory management from an embedded Linux OS, using libraries provided by the PYNQ framework as well as autogenerated drivers from Vitis HLS.

8.3.7 Access to the Local Buffer

Loads and stores on the local buffer are performed through one generic load-store unit, offering one store and two loads per FU per cycle on at least two different memory locations. To allow off-chip communications at a rate of 64-bit per cycle, the Local Buffer is implemented with double-port BRAM and partitioned cyclically by a factor of 2, granting a maximum of 4 simultaneous FP16 loads/stores per cycle.

	Number of operators			Nb. of	IVG supports	Local Buffer
	FMA	a/b	$\sqrt{a}$	FU	triangular loops	Size
BLAS	2	0	0	2	Yes	27 Matrices
CORR	3	1	1	4	No	27 Matrices

Table 8.4: Configuration of the LA-GA accelerator and the Correlation accelerator

8.4 Experimental Results

In this section, we will analyze the performances of two merged accelerators whose characteristics are reported in Tbl. 8.4: one optimized for dense linear algebra computation, the other for the computation of correlation matrices, as expressed in PolyBench/C [50].

All cycle measurements of the GAs are on a ZCU104 board running the PYNQ 2.6 Linux image, and all IP are generated from annotated C++ code using Xilinx Vitis 2022.1 [64] on Linux 6 on a laptop equipped with an AMD Ryzen 7 2700U @ 2.2 GHz. Custom designs as well as pure-HLS GAs are implemented in pure HLS using Vitis HLS 2022.2 (unsafe math optimizations disabled), and their resource usage is measured after out-of-context P&R, which excludes data routing between the accelerators and the integrated CPU. For non-pure HLS GA that also uses Vivado's block design primitives, resource usage are computed as a delta between the complete placed and routed GA and the SoC-GA interconnect placed and routed alone. In our experimental setup, this interconnect accounts for 3007 FFs and 2708 LUTs. Cycles measurements of the proposed accelerator are taken by an on-chip counter on the target board, whereas custom accelerators' execution times are computed from the pipeline latency given by the HLS Tool report. Unless specified, the data type used is 16-bit floating point. The total functionalities of the accelerators are summarized in Tbl. 8.5; which are integrated in two FU types:

- one capable of handling `mulmm` and all the `mul` and `add`/`sub` derivatives (customized for either Linear Algebra or Correlation with kernel-specific data routing)
- one handling `sqrt` and `div`, based on two operators: $\sqrt{\cdot}$ and /, only used for Correlation

We compare our accelerators both with HLS-based FUs and Vivado-based FUs with the Max Sharing (MS) dedicated design, where only one physical hardware unit is instantiated for each operation type, and the Max Throughput (MT) one that achieves minimal execution time while keeping all data in a local buffer of the same characteristics than the generic accelerator one, thus limiting the available parallelism. *On both MT and MS, no genericity of the design is possible*, i.e. only the selected benchmark can be executed. We evaluate our generic accelerator on two metrics: execution time (in cycle) and throughput per area, computed as $\frac{NB_FLOP}{EXEC_TIME*NB_RESOURCE}$ with NB_FLOP the number of

Kernel	Description	Op.	LA-GA	CORR-GA
`noop`	Do nothing	None	✓	✓
`mulmm`	Matrix-matrix multiplication	FMA	✓	✓
`mulmv`	Matrix-vector multiplication	FMA	✓	✓
`multrmm`	Triangular matrix-matrix multiplication	FMA	✓	
`multrmv`	Triangular matrix-vector multiplication	FMA	✓	
`mulsm`	Scalar-matrix multiplication	FMA	✓	✓
`multrsm`	Scalar-triangular matrix multiplication	FMA	✓	
`mulsv`	Scalar-vector multiplication	FMA	✓	✓
`muls`	Scalar-scalar multiplication	FMA	✓	✓
`trm`	Matrix transposition	None	✓	✓
`addm`	Matrix addition	FMA	✓	✓
`addv`	Vector addition	FMA	✓	✓
`adds`	Scalar addition	FMA	✓	✓
`addtrm`	Triangular matrix addition	FMA	✓	
`subm`	Matrix subtraction	FMA	✓	✓
`subcmv`	Column-wise matrix subtraction	FMA		✓
`subv`	Vector subtraction	FMA	✓	✓
`subs`	Scalar subtraction	FMA	✓	✓
`pmulm`	Point-wise matrix multiplication	FMA		✓
`pmulv`	Point-wise vector multiplication	FMA		✓
`oprodv`	Outer (vector) product	FMA	✓	✓
`copyv`	Vector copy	None	✓	
`dot`	Dot product	FMA	✓	
`sasum`	Vector sum of absolute value	FMA	✓	
`sqrtv`	Point-wise vector square root	$\sqrt{\cdot}$		✓
`sqrts`	Scalar square root	$\sqrt{\cdot}$		✓
`accsumcm`	Columns-wise accumulation of a matrix	FMA	✓	✓
`cutminv`	Vector round to 1 low values	None	✓	✓
`divms`	Pointwise division of matrices	FMA		✓
`divvs`	Pointwise division of vectors	FMA		✓
`divcmv`	Point-wise division with col.-wise value	/		✓
`set0m`	Initialisation of a matrix to 0	None		✓
`setidm`	Initialisation of a matrix to Id	None		✓
`setd1`	Initialisation of the diag. of a matrix to 1	None		✓

Table 8.5: Supported kernels list, by either the Correlation or the Linear Algebra accelerator

16-bit floating-point operations in the input benchmarks, and $NB_RESOURCE$ the number of DSPs or chunks of 10 000 FF / LUT in the design.

8.4.1 Linear Algebra

The accelerator for linear algebra, noted LA-GA is composed of two identical FUs supporting FMA, additions / subtractions, multiplications, absolute value computations as well as matrix transpositions. Its execution time, resource and performance per area

Bench name	Arithmetic expression	Execution Time (cycles) MS	MT	LA-GA-HLS	LA-GA
ASUM	$\alpha = \|x\|_{L1}$	70	70	227	493
DOT	$\alpha = x \cdot y$	71	71	227	493
SCALV	$A = \alpha \cdot x$	69	37	99	109
SCALM	$A = \alpha \cdot A + B$	5572	2059	8243	8258
GEMV	$y = \alpha \cdot A \cdot x + \beta \cdot y$	4553	2126	4291	4311
TRMV	$y = A \cdot x$	2304	2304	2115	2125
GER	$A = \alpha \cdot x \cdot y^t + A$	4738	2057	8323	8343
GEMM	$C = \alpha \cdot A \cdot B + \beta \cdot C$	307586	134018	270403	270423
TRMM	$C = \alpha \cdot A \cdot B$	147456	147456	137267	137282

(a)

Bench name	FLOP/C/DSP MS	MT	LA-GA HLS	LA-GA	FLOP/C/10kFF MS	MT	LA-GA HLS	LA-GA	FLOP/C/10kLUT MS	MT	LA-GA HLS	LA-GA
ASUM	0.457	0.457	0.047	0.065	3.302	3.302	0.700	0.188	5.224	5.224	0.226	0.081
DOT	0.601	0.451	0.094	0.130	6.432	6.432	1.401	0.376	9.955	10.005	0.452	0.163
SCALV	0.464	0.432	0.108	0.294	3.356	6.229	1.606	0.850	5.453	10.062	0.518	0.368
SCALM	0.368	0.497	0.166	0.496	5.080	13.593	2.468	1.436	7.722	20.466	0.796	0.622
GEMV	0.457	0.391	0.323	0.965	3.960	12.950	4.816	2.793	5.686	18.752	1.553	1.210
TRMV	0.444	0.444	0.323	0.964	6.074	6.084	4.810	2.789	9.250	9.259	1.551	1.208
GER	0.436	0.401	0.165	0.495	6.093	13.528	2.464	1.432	9.348	20.058	0.794	0.620
GEMM	0.433	0.397	0.328	0.985	5.759	12.934	4.891	2.850	8.860	19.011	1.577	1.234
TRMM	0.445	0.296	0.318	0.955	5.867	5.953	4.745	2.764	9.190	9.026	1.530	1.197

(b)

Table 8.6: Execution time (a) and performance-per-area (b) of a custom IP optimized for Max Sharing (MS) and Max Throughput (MT) and the Generic Accelerator, both with pure HLS FUs (LA-GA-HLS) and non-HLS FUs (LA-GA) for several linear algebra benchmarks

metrics are reported in Tbl. 8.6 for 9 linear algebra benchmarks, along with the description of their computations. Performances on batches of 3 independent problems are also evaluated in Tbl. 8.7.

The FU layout of the LA-GA was guided by the presence of high-density FP16 FMA primitives on our synthesis toolchain (Vivado ML Edition 2022.2): at the cost of AXIStream connections in the top-level block design, fused multiply-add units can only cost 1 DSP, which is not possible using pure HLS without heavy rewrite of the original code to explicitly schedule operation on available units. Therefore, our target board architecture does not benefit from FU integrating only additions or only multiplication, and will in some cases exhibit higher performance-per-area metric than both dedicated designs, cases in which we bold the GA FLOP/Cycle/DSP value. However, this is not the case in the general case. To measure the effect of this optimization, we also report metrics of the *pure HLS* GS (LA-GA-HLS), relying on the Vitis HLS FMA, which takes 4 DSP but have the advantage of a lower latency.

Bench name	Execution Time (cycles)				FLOP/C/DSP			
	MS	MT	LA-GA-HLS	LA-GA	MS	MT	LA-GA-HLS	LA-GA
ASUMx3	210	210	435	962	0.457	0.457	0.074	0.100
DOTx3	213	213	435	962	0.601	0.451	0.147	0.200
SCALVx3	207	111	179	194	0.464	0.432	0.179	**0.495**
SCALMx3	16716	6177	12355	12375	0.368	0.497	0.332	**0.993**
GEMVx3	13659	6378	8643	8683	0.457	0.391	0.481	**1.437**
TRMVx3	6912	6912	4211	4226	0.444	0.444	0.486	**1.454**
GERx3	14214	6171	16627	16662	0.436	0.401	0.248	**0.743**
GEMMx3	922758	402054	544899	544939	0.433	0.397	0.489	**1.466**
TRMMx3	442368	442368	274515	274540	0.445	0.296	0.478	**1.433**

(a)

Bench name	FLOP/C/10kFF				FLOP/C/10kLUT			
	MS	MT	LA-GA-HLS	LA-GA	MS	MT	LA-GA-HLS	LA-GA
ASUMx3	3.302	3.302	1.096	0.289	5.224	5.224	0.202	0.125
DOTx3	6.432	6.432	2.193	0.578	9.955	10.005	0.404	0.250
SCALVx3	3.356	6.229	2.664	1.432	5.453	10.062	0.490	0.620
SCALMx3	5.080	13.593	4.941	2.874	7.722	20.466	0.909	0.909
GEMVx3	3.960	12.950	7.173	4.160	5.686	18.752	1.320	1.802
TRMVx3	6.074	6.084	7.248	4.208	9.250	9.259	1.334	1.822
GERx3	6.093	13.528	3.700	2.151	9.348	20.058	0.681	0.932
GEMMx3	5.759	12.934	7.282	4.242	8.860	19.011	1.340	1.837
TRMMx3	5.867	5.953	7.117	4.147	9.190	9.026	1.310	1.796

(b)

Table 8.7: Execution time (a) and performance-per-area (b) of a custom IP optimized for Max Sharing (MS) and Max Throughput (MT) and the Generic Accelerator, both with pure HLS FUs (LA-GA-HLS) and non-HLS FUs (LA-GA) for batched linear algebra benchmarks

Bench name	Arithmetic expression	Execution Time (cycles)			
		MS	MT	CORR-GA-HLS	CORR-GA
CENTER	$X^C_{ij} = X_{ij} - (\sum_{i'} X_{i'j})/n$	8343	4166	12530	12535
STDDEV	$\sigma^X_j = \sqrt{\sum_i (X^C_i)^2/n}$	16691	8370	29153	29163
CTR-RED-DIV	$X^{CR}_{ij} = (X_{ij} - \sum_{i'} X_{i'j}) / (\sigma^X_j \cdot \sqrt{n})$	20935	10486	33482	33495
CORR	$(X^{CR})^t \cdot X^{CR}$	291221	144614	308054	308071

(a)

Bench name	FLOP/C/DSP				FLOP/C/10kFF				FLOP/C/10kLUT			
	MS	MT	CORR-GA HLS	CORR-GA	MS	MT	CORR-GA HLS	CORR-GA	MS MS	MT	CORR-GA HLS	CORR-GA
CENTER	0.495	0.495	0.055	0.220	10.362	19.448	0.881	0.709	9.570	15.374	0.347	0.314
STDDEV	0.247	0.247	0.047	0.189	7.796	13.991	0.758	0.610	6.148	10.148	0.299	0.270
CTR-RED-DIV	0.247	0.164	0.051	0.206	6.579	0.826	9.707	0.665	5.579	7.761	0.326	0.294
CORR	0.468	0.314	0.147	**0.590**	10.905	17.962	2.366	1.905	9.119	13.834	0.933	0.844

(b)

Table 8.8: Execution time (a) and performance-per-area (b) of a custom IP optimized for Max Sharing (MS) and Max Throughput (MT) and the Generic Accelerator, both with pure HLS FUs (CORR-GA-HLS) and non-HLS FUs (CORR-GA) for correlation subexpressions

Bench name	Execution Time (cycles)				FLOP/C/DSP			
	MS	MT	CORR-GA-HLS	CORR-GA	MS	MT	CORR-GA-HLS	CORR-GA
CENTERx3	25029	12498	12728	12735	0.495	0.495	0.162	**0.648**
STDDEVx3	50073	25110	37811	37827	0.247	0.247	0.109	**0.437**
CENTER-REDUCE-DIVx3	62805	50402	31458	50423	0.247	0.164	0.103	**0.410**
CORRx3	873663	433842	320843	320867	0.468	0.314	0.425	**1.698**

(a)

Bench name	FLOP/C/10kFF				FLOP/C/10kLUT			
	MS	MT	CORR-GA-HLS	CORR-GA	MS	MT	CORR-GA-HLS	CORR-GA
CENTERx3	10.362	19.448	2.603	2.095	9.570	15.374	1.026	0.928
STDDEVx3	7.796	13.991	1.752	1.410	6.148	10.148	0.691	0.625
CENTER-REDUCE-DIVx3	6.579	1.646	9.707	1.325	5.579	7.761	0.649	0.587
CORRx3	10.905	17.962	6.815	5.488	9.119	13.834	2.687	2.430

(b)

Table 8.9: Execution time (a) and performance-per-area (b) of a custom IP optimized for Max Sharing (MS) and Max Throughput (MT) and the Generic Accelerator, both with pure HLS FUs (CORR-GA-HLS) and non-HLS FUs (CORR-GA) for batched correlation subexpressions

Even with this optimization, on SCALV, SCALM and GER, the LA-GA is around 2 times slower than MS for non-batched workloads. This is due to the dedicated accelerator expressing in one fully pipelined loop nest the complete application, whereas the LA-GA splits it in several (fully pipelined) kernels, increasing the overall latency. A solution to avoid this issue is to increase the granularity of FUs and allow expressions such as $\alpha \cdot A + B$ as a single kernel, which is not possible in our current implementation as FUs are limited to 2 inputs / 1 output computations per cycle. However, these differences fade away when the input is batched as the LA-GA will overlap sequential kernel executions through coarse-grain pipelining and present systematically faster execution that MS except for GER, where the batching factor is not enough to benefit from coarse-grain pipelining due to a 3-stage pipeline.

On ASUM and DOT, the LA-GA execution time is even worse, falling behind dedicated accelerators with a factor of 7. This is one limitation of our approach: because of the use of AXIStream interconnect with the area-efficient FMA primitives, scalar computation takes 7 cycles on the LA-GA instead of 1 on dedicated designs, and 3 on the LA-GA-HLS for which this interconnect is also not present. Therefore, the execution pipeline of the LA-GA must stall 6 cycles waiting for the former computation in the case of reductions, leading to poor execution time (and thus performance-per-area) on BLAS1 primitives with sequential dependence on their (only) loop.

8.4.2 Correlation

For data science applications such as Correlation, linear algebra primitives are not sufficient as other kernels are needed: column-wise accumulation of the matrix (a one-kernel implementation of $A^t \cdot 1_{vector}$), column-wise subtraction of a vector to a matrix and cut-off of a vector (used to avoid floating-points error when dividing by a near-zero value) as well as division and square root.

Therefore, we enriched our accelerator with one FU merging these four kernels to create the CORR-GA accelerator whose configuration is detailed in Tbl. 8.4, that integrates:

- 3 FU capable of computing `mulmm` and all derivatives (kernels relying on FMA or any of its variations);
- 1 FU capable of computing either $\sqrt{\cdot}$ or /

Reports of the execution time as well as performance-per-area metrics are summarized in Tbl. 8.8 for one execution of each kernel, while performances on 3-batched workspans are detailed in Tbl. 8.9.

The CORR-GA performs significantly worse than dedicated designs on CENTER, STDDEV and CTR-RED-DIV, with execution time degraded from 50 % to 133 %. As for the LA-GA, this is due to the granularity of the kernels that imposes additional execution stages compared to dedicated accelerators. On the complete Correlation computation, the CORR-GA is similar in execution time to Max Sharing and better in terms of performance-per-area than dedicated designs. Indeed, CORR complexity is dominated by the matrix multiplication step, for which the CORR-GA behaves similarly to the Max Sharing accelerator (one FMA per cycle), so our approach shows similar execution time and better performance per area thanks to its area-efficient FMA units.

Batching multiplies this ratio by 3 as all three available FMA-compatible FUs can be used in parallel, while lesser-used divider and square root units are shared between instances. More generally, on all of the tested batched expressions, the CORR-GA outperforms dedicated designs in terms of performance-per-DSP, showing the interest of our approach for area-efficient, throughput-based accelerator generation.

8.4.3 Scaling and Comparison

We evaluate the scalability of our approach on three different aspects: data type, number of entries of the local buffer and problem size. Area measurements are reported after P&R in Tbl. 8.10.

While switching from half precision to double precision doubles LUT due to the additional routing resources necessary to handle the supplementary data, the accelerator logic size (LUT and FF) only increases by around 20 % when quadrupling the size of the local buffer. This is due to the fact that loading and storing units are the only elements to scale with its size: the data dispatch, FU selection and iteration vector generation logic remains unchanged. Moreover, the number of DSP is multiplied by 10 when the data size quadruples, which shows the limitations of FPGAs for high precision floating-point computations.[15]

On the other hand, LUT, FF and DSP usage increases linearly with the number of FU, suggesting that our approach does not generate quadratic amount of logic with respect to its raw computation power. However, synthesis time increases significantly with the

[15]In this case, the pure HLS version would take 11 DSP (8 for the multiplier and 3 for the adder), suggesting that the DSP used in the ZCU104 are not designed for FP64 computations.

Data Type	Nb. Entries	Nb. FU	Pb. Size	LUT	FF	DSP	BRAM
FP16	27	2	64	15955	6910	2	117
FP16 (HLS)	27	2	64	12487	4026	6	117
FP64	27	2	64	33190	18795	20	441
FP16	**50**	2	64	18452	8038	2	209
FP16	**100**	2	64	19499	8274	2	409
FP16	27	2	**90**	16422	7706	2	225
FP16	27	**5**	64	32240	11170	5	117
FP16	27	**10**	64	56701	17090	10	120
FP16	27	**20**	64	HLS Synthesis time out (> 3h)			

Table 8.10: Scaling properties of the LA-GA accelerator

Accelerator	Maximum achieved frequency after P&R (MHz)
CORR-MS	222
CORR-MT	225
CORR-GA (2 FU)	**217**
CORR-GA (10 FU)	**176**

Table 8.11: Maximum frequency achieved for each design

number of FUs, reaching several hours for a GA with 10 FUs. This aligns with the results shown in Sec. 7.6.2 for our greedy mapping heuristics, with interconnect size in the order of magnitude of 10 000 LUTs and FFs for less than 50 DSPs.

As a demonstration of the influence of the GA structure on frequency compared to a full design, Tbl. 8.11 reports maximum achievable frequency according to the timing report for different implementations of the Corr accelerator[16]: the Max Sharing and Max Throughput one, and the Generic Accelerator with 2 and 10 FUs. As the Generic accelerator requires more logic to schedule dynamically its kernel on the FUs and complex memory controller units to route data from the shared buffer to the FUs, its critical path is slightly reduced compared to the dedicated designs: from 222 MHz achieved by the Max Sharing IP, the GA with 10 FU could only reach 176 MHz. However, this complexity is bound to the number of FUs, as shown by the 2 FU version, whose frequency (217 MHz) nearly matches dedicated design. This is easily explainable, as the GA complexity comes mainly from the adaptation of generic templated components to multiple FUs – and not from the FUs themselves.

[16]Note that the dedicated designs require specification of a target critical path of 5 ns at the HLS step, while the default HLS target of 10 ns is sufficient for the GA to reach 200 MHz-level frequency, for the reasons described hereafter.

Data Type	Implementation	OP/Cycle/DSP
INT32	ResNet-18 ScaleHLS [77]	1.343
INT32	ResNet-18 TVM-VTA [101]	0.344
INT32	LA-GA GEMM	0.646
FP32	GEMM ScaleHLS	0.393
FP32	LA-GA-HLS GEMM	0.212
FP32	LA-GA GEMM	**0.849**

Table 8.12: Performance per area comparison with data extracted from other published accelerators

We also provide as a indicative example in Tbl. 8.12 a comparison of our performance against two state-of-the art designs dedicated to machine learning workloads: ScaleHLS [77] and VTA [101], on GEMM, extrapolated from their FP16 (2 DSP per addition, 2 DSP per multiplication) to an FP32 projection (2 DSP per addition, 3 DSP per multiplication) and INT32 one (0 DSP per addition, 3 DSP per multiplication) from the ScaleHLS publication, and compare to ours. However, our utilization of block design-level DSP IP does not allow an apple-to-apple comparison with tools that only rely on DSP as instantiated by Vitis HLS. Therefore, we included the LA-GA-HLS GEMM implementation to measure the exact cost of kernel merging in terms of throughput-per-area when only relying on current High-Level Synthesis tools: roughly 50 %. When comparing the LA-GA optimized with single-DSP, throughput optimized FUs, we achieve comparable performance to state of the art designs, whereas ScaleHLS optimizes a single workload and is not producing a generic accelerator.

8.5 Limitations

Though the kernel merging approach for automated generation of general accelerators is promising, our implementation suffers from several flaws, both on the technical side (unused/overused FPGA resources) and on our evaluated accelerators.

8.5.1 Routing between FUs and Buffers

Our implementation allows every FU to access every memory location of the local buffer for easier customization of the generated accelerator. Indeed, a generic local buffer load/store IP is integrated for every FU, that rely on costly multiplexers and intricate code formulation for the HLS toolchain in order to generate valid RTL output. This could be avoided by specializing it to the access pattern of the FU, to the cost of an increased pressure on the final placement/scheduling compilation step.

Deeper polyhedral analysis and re-scheduling may also exhibit cross-FU reuse opportunities when the same data is used by 2 different FUs. A future research direction may be to ensure maximal merging of these data paths to avoid as best as possible redundant loads; but we expect this analysis to lead to few real-life use cases.

8.5.2 Merging of Kernels with Different Iteration Space

In all tested benchmarks, the iteration space vector can be shared among all merged kernels. However, this is not true in general: two kernels may iterate over dimensions of different size, which requires the generation of two iteration vectors by the IVG. This leads to additional LUT-based logic limiting the application of our approach on LUT-constrained designs, but should not disturb the execution time of our tested benchmarks.

8.5.3 Data Reuse: Optimizing Buffer Communication

Our implementation does not consider reuse of data inter iteration of the FU, as this may introduce loop carried dependencies and thus stall the pipeline. However, short-distance single-producer/multiple consumer data can be kept in FF-based memory to alleviate BRAM's load, diminishing pressure on ports and allowing further parallel computations on the now-loadable data.

8.5.4 No Control Flow Instructions

The major difference between a GA and a CPU lies in the incapability of the GA to perform data-dependent control flow kernels. On the architectural point of view, this translates by two unfitted components:

1. The GA has restricted access to its kernel index (its "program counter"), as it can only read it then increment it by one.
2. The bound used for the main scheduling loop is fixed, which prevent any form of early exit.

While simple transformations of the accelerator template can allow direct access to the kernel index, solving the first issue; early exit management requires deeper modifications to ensure correct handling of the `break` statements without degrading the main loop's initiation interval. Another approach relies in rewriting the main loop from a `for` to a `while`, which has not been explored yet.

8.5.5 Vectorization of the FUs

In our evaluation, we only consider FUs composed of one fused multiply-add operation, that is, scalar FUs. While this is a limitation of our current implementation, there is no technical reason to do so in the general case: the elementary data type may be switched from 16-bit floating point scalar to a wider one such as 4xFP16 to adapt our existing structure to vectorized FUs. Such an adaptation with AXIStream FU primitives timed out ($>$ 20 hours of HLS synthesis) on our test machine, probably due to the use of 20 GiB of SWAP space as the 32 GiB of RAM available were not sufficient.

8.6 Related Work

The topic of semi-specialized accelerator design has been widely studied recently [102, 103, 104], targeting a variety of subdomains such as encryption [105], graph processing [106] or machine learning [107, 108, 109, 110]. Thanks to HLS, the expertise required to design hardware architectures has lowered, which allows the focus on the high-level sharing method rather that its implementation details: for example Cong et al. [111] propose a technique to quickly generate accelerators on a template architecture, but targets single application acceleration on MPSoC, in contrast to the multi-functionality approach detailed in this chapter.

This section analysis the major research projects dedicated to generic accelerator designs, by comparing the effect of the underlying architecture onto design decisions: while Sec. 8.6.1 overviews the trade-offs offered by 2 implementations of architecture-agnostic resource-shared generic designs (DSAGEN [112] on ASICs and the Versatile Tensor Accelerator [101] on FPGAs), Sec 8.6.2 focuses on a DSP-specialized overlay from Jain et al. [81], whose goal is similar to the Generic Accelerator presented in this chapter.

8.6.1 Generic Resource-shared Designs

DSAGEN

Common data dependence patterns described in [113], composed of stream-join and alias-free indirections, lead to the development of the SPU – Sparse Processing Unit –, a generic systolic accelerator for applications presenting complex data-dependent control flow.

From this work emerges DSAGEN[112], an infrastructure that enables the generation of custom architectures for domain-specific computation (such as the SPU) through the customizable composition of modular building blocks: Processing Element, Switch, Memory Delay, Sync and Controller. This approach relies on a decoupling of the memory and computation pipeline while also exposing hardware features to the software stack through a feature-rich compiler. Optionally, the hardware architecture can be further optimized to better suite software characteristics using a local search algorithm to perform automated Design Space Exploration (DSE).

Contrary to the GA, both projects target ASIC generation, which allows more complex interconnect and achieve better overall performances compared to FPGAs because of a usually higher operating frequency and denser computing logic. This orientation was abandoned in OverGen [88], a hardware generation framework coupling DSAGEN with ChipYard [114] (a SoC generator build for agile design flow) in order to automatically design domain-specific overlays dedicated to a specific underlying architecture.

Moreover, DSAGEN's decomposition of the architecture operates as a finer grain, as some templated components of the GA appear in DSAGEN as configurable modular blocks, e.g. memory or synchronization elements.

Concerning generality of the output designs, both approaches take as input C code as a proxy for the supported features, hence the same notion of *semi-specialization*.

While the Control Core may seem analogous to our Loop Control Logic unit (See Sec. 8.3 for more details), it is in fact implemented using a Turing-complete RISC-V core, more powerful but bigger than the workload-derived ISA supported by the GA.

Finally, the spread of PE layouts supported by DSAGEN differs fundamentally from the fixed, complete memory-to-FU topology implemented in the GA. Indeed, DSAGEN rely on an Architecture Description Graph (ADG) to specify the layout of its supported building blocks, allowing the generation of CGRA tiles as well as systolic architectures. Contrarily, the fixed, templated architecture of the GA allows simpler parametrization and synthesis, but also hurt scaling to the point of timing out for designs as small as 20 FUs, and requiring extra routing logic because of the possible accesses of *all* FUs to the shared buffer.

VTA

While DSAGEN targets ASICs in order to achieve performances comparable to state-of-the-art hardware accelerators, the Versatile Tensor Accelerator [101] (subsequently referred as VTA) leverage High Level Synthesis to also target edge-class FPGA such as Xilinx' PYNQ Z1[17].

Similarly to the GA presented in this chapter, VTA relies on the concept of decomposition of programs into kernels for deported execution on an accelerator. However, its deep learning focus limits its functional units to configurable GEMM and tensor-specialized ALUs, which differs from the automatically generated FUs of our GA. Nevertheless, VTA is a fully parametrizable extensible framework in which new hardware intrinsics can be added seamlessly by design. Contrarily to the GA, these new compute capabilities must be implemented by hand in the framework, instead of relying on kernel fusion to automate this step.

VTA also features a DSE step to optimize its architectural parameters such as the shape of the internal GEMM accelerator in order to adapt it to a new chip. Such a functionality is not present in the GA, where the resource usage is only guided by the set of accelerated kernels and an upper bound corresponding to the FPGA size.

Moreover, VTA's kernels are translated from a task-level ISA to micro-code using a JIT compiler, whereas our approach directly exposes the supported operations to the user (similar to VTA's micro-ops) and allows seamless mapping of new applications to the design thanks to a static placement algorithm detailed in Sec. 8.2.3. Interestingly, both approaches are limited in the expressiveness of their internal ISA as neither of them can express branches: repetitions of the (micro-)instructions have to be used to emulate loops.

[17] Note that VTA has to degrade the original FP32 data type to a FPGA-friendlier INT8 "with negligible accuracy degradation" in order to exhibit competitive performances when compared to industry-optimized ARM libraries.

8.6.2 DSP-dedicated Resource Sharing on Overlay Architectures

While approaches mentioned in the former section focus on the synthesis of generic architectures, they function as high-level tools, either unaware of the actual architecture of the targeted FPGA or dedicated to ASICs, for which this architecture is non-existent. While this allows simpler adaptation to new chips thanks to this genericity, this also means that part of the performances may be left on the table, either due to unnecessary generic interfaces (and subsequently, glue logic), or lack of usage of all the possibilities offer by the targeted compute units. For example, HLS-based designs are limited by the panel of DSP primitives integrated in the synthesis tool, as described in Sec. 6.2.3: manually or semi-manually optimized designs would not suffer from this issue.

Jane et al. [81] project explores this direction by manually implementing their FUs to take advantage of the underlying DSP48E1 block of Xilinx' Ultrascale Zynq chips in a runtime-configurable manner. These FUs are composed following the classic island structure [8] to provide efficient yet configurable overlays on which applications are mapped using a customized compiler, as detailed in the next paragraphs.

When compared with Vitis HLS's out-of-the-box designs, the overlay architecture is 40% faster in average over a set of 24 benchmarks, while using up to 50 % of its theoretical processing power. Furthermore, benchmark-specific reconfiguration is performed more than 1000 times faster than using HLS, as no synthesis is required to map an application to the overlay once the has been generated.

Technical Solution

The architecture used in Jain et al.'s overlay relies on a classical *island* topology: a square 2D array of precessing elements called tiles, where each tile can communicate horizontally and vertically to its immediate neighbor.

Choice of the Overlay Architecture The major drawback of an island-style topology lies in the imbalance between of IO resources and the computation resources. While computing resources are integrated in all tiles (thus growing quadratically), the amount of I/O resource only increase linearly as only tiles on the edges of the array can communicate with off-chip components.

Therefore, the compromises at stake here are:

- The size of the 2D array, that defines the ratio between I/O tiles and compute tiles. The bigger the array, the more powerful the overlay is in terms of theoretical peak power, but the harder it is to fully use it.
- The number of communication channels per tile edge: more channels means more bandwidth between tiles and for off-chip communications, but also more complex interconnect resources.
- The computing power of each tile. While single-DSP tiles offers the most flexibility in term of exposition of the hardware-level features to the application, it also comes

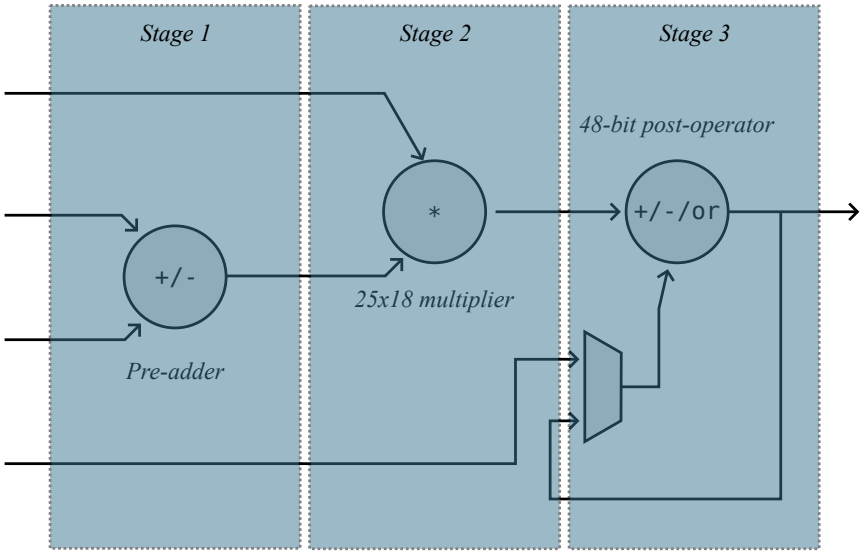


Figure 8.12: Anatomy of the DSP48E1 as described by the vendor documentation [2]

with more routing resource per DSP than multi-DSP tiles. Contrarily, multi-DSP tiles limit the CDFG pattern that can take full advantage of the DSPs due to the specialization of certain data paths.

The DSP48E1-derived FU As seen in Sec. 6.1.1, FPGAs integrate pipelined units called DSP, dedicated to the acceleration of a several fixed-bitwidth computation. Xilinx' Ultrascale Architecture integrates the *DSP48E1* [2], a 48-bit programmable hardware primitive composed of a pre-adder (optionally configurable as a subtraction), a 25x18-bit multiplier and a configurable 48-bit post-operator (either an addition, a subtraction or a logical `and`), as illustrated in Fig. 8.12. Therefore, the maximum theoretical peak performance of an FPGA, assuming all operations to be mapped to DSP, is $3 \times \#DSP$. However, usual designing methods rely on fixed data paths and usage of DSPs for one fixed sequence of computation, which considerably limits their in-practice compute power.

The FUs used in [81] aims at offering a configurable elementary compute unit while minimizing interconnect size and exposing hardware features to the software stack. Following a scalability study of the compromises detailed in the former paragraph, the final FU is composed of 2 DSP48E1 units in cascade, fully pipelined with a 4-input, 4-output structure. One of them is preceded by shift registers for synchronization purposes, which translate to a 13 cycles latency when using both DSP, or 8 cycles when shortcuting the second one. The desired operation and data path is selected by configuring a 109-bit control register, implemented as Flip-Flops.

Finally, a tile corresponds to a FU, enriched with routing elements that allow communications with the neighboring tiles.

Compilation to the Overlay To deport an application on the overlay, its complete CDFG is deduced from a set of kernels described in a high level language, similarly to the GA. Its composing nodes are then collapsed using pattern matching to form one of

the hardware-accelerated primitives if the data path allows it. This reduces the size of the graph and allows faster execution compared to non-architecture optimized designs. Then, each node is assigned to a FU of the overlay.

Optionally, the mapped kernels can be replicated several times, either as identical or different instances, to further increase occupancy of the overlay and increase overall throughput.

Finally, the synchronization between FUs is achieved by configuring the size of their shift registers in order to delay inputs and ensure coherency of the data.

This operation-level mapping contrasts with the approach taken by VTA and the GA, which both operate at the coarser granularity of kernels. In that case, targeted applications are decomposed into hardware-accelerated primitives, before being mapped to the final design.

Limitations

One of the strength of this approach relies in the tight coupling between the FU architecture and the FPGA's DSP blocks. However, this feature has a major drawback: no automation of the generation of the FUs is possible without information about the target FPGA, trading performance for customizability. This direction of improvement is explored in by our Generic Accelerator, for which loop merging is used to generate FUs that are specialized to a set of applications, as described in Sec. 8.2.

Also, the approach of the GA is limited to polyhedral codes, and its loop merging technique is in practice efficient for applications relying on multiple loops of similar cardinality of their iteration domain. This limitation is not present in Jain's approach, where the compiler targeting the overlay operates at the operation level instead of the loop level. This allows a bigger spectrum of supported applications, as well as better load balancing across FUs.

Moreover, the interconnect between overlay tiles (FUs augmented with data routing capabilities) is array-shaped [8], which improve scalability in terms of number of FUs that can be integrated on-chip but limits the in-practice throughput due the lack of parallelism exploitable by of all FUs simultaneously, translating to a sub-optimal occupancy of the tiles.

Contrarily, our GA is built around a templated interconnect that allows arbitrary communications between FUs, as data is stored in an on-chip buffer shared between all FUs. While this approach allows superior performances in terms of throughput-per-DSP, as evaluated in Sec. 8.4, it is not without compromises on its scaling, both with respect to local buffer size and the number of FUs – similarily to the approach taken by DSAGEN [112].

Chapter 9
Conclusion and Future Research Directions

With the decline of yearly frequency improvement of chips, the pressure of delivering generational performance gains now shifts toward the rest of the computation stack: software and architecture [115]. In a ideal world, both would be co-developed together in order to reduce the so-called software bloat and avoid unused chips elements. However, we are far from this situation, partly due to the multiplicity of actors and use cases of silicon chips, hence the perpetual back and forth between new architecture bringing *possibilities* and optimization of the software on *existing hardware*.

This manuscript aims at providing possible answers for both software optimization and hardware generation:

On one hand, the PALMED framework developed in Chap. 3 and 4 allows to generate automatically accurate performance models through a microbenchmark-driven approach, paving the way for simpler assembly-level optimizations, either compiler-automated or manual. In these chapters, we show how two selection filters can reduce to ISA to a limited number of instructions with different resource usage characteristics after a first profiling of the instruction set using quadratic benchmarking. We also developed an algorithm able to solve the instruction to resource mapping problem by formulating the search space as a *dual* form in which performance estimation of microkernels boils down to a max of summation, allowing faster solving. We apply this technique to two state-of-the art processors and show an accuracy similar to alternative hand-optimized tools, illustrating the soundness of this approach. Finally, Chap. 5 perfects this work by embedding it into a formal frame and proving convergence of PALMED to the unique resource mapping of any processor.

On the other hand, other ways to adapt hardware architectures to the program they run are explored in this manuscript under the compute resource sharing aspect on FPGA. Within the HLS framework, the tentative detailed in Chap. 7 of solving the combined placement-schedule using an LP solver hits a complexity wall when trying to find an exact solution, with solving time reaching more than 10 hours for mapping of compute DAGs of fewer than 50 nodes. Fall backing to an approximated approach, using a greedy heuristic shows promising results in terms of performances, but hits a second wall: the interconnect size that voids partly the resource sharing gains.

Therefore, we build in Chap. 8 a generic accelerator from the idea of developing this interconnect in order to construct a multi-purpose design, instead of trying to reduce its size. Thus, we leverage loop merging, a well-known polyhedral transformation, to build semi-specialized FUs and integrate them in a fixed canvas to create accelerators dedicated to several applications while keeping our focused on resource sharing, that is, efficiency of usage of available computing elements. This technique has been evaluated on Linear Algebra as well as Correlation IPs, and has shown to conserve DSP-efficiency

of dedicated designs, with a cost in logic elements similar to our greedy mapping try, but adding multi-functionality as a supplementary functionality.

Future Research Directions

PALMED

Precise processor models, and by extension resource mappings are particularly useful for two applications: performance predictions and bottleneck analysis. The most straightforward application of Part I of this manuscript would be to automatically plug PALMED's resource mapping into a compiler back-end such as LLVM, avoiding the ad-hoc mapping it currently relies on without degrading its performance. However, as shown in Sec.4.6, PALMED is not significantly more accurate than existing performance models, as it strength mostly relies in its automated behavior. Therefore, we think of the integration of automated performance models into compilers as a long-term research direction that offers a pragmatic way of dealing with the diversity of chips released on the market over the years.

However, bottleneck analysis has also proven to be a top-of-the-line research topic. Micro-architectural simulators such as Gem5 [116], IACA [25], OSACA [38] or GUS [27] demonstrates the demand for such tools, whose development is cumbersome when CPU microarchitectures are refreshed. Therefore, PALMED's mapping can be used as one component of a modular performance analysis tool by providing the instruction-level throughput model. To that goal, ideas and infrastructure components of PALMED/PIPEDREAM may be reused for assembly latency measurements in order to automatically characterize instruction-level performance profiles. Nevertheless, other main components of such a simulator also have to be developed: a PALMED-like estimator for the memory subsystem latency (including cache hierarchy and prefetching), a cost model for the branch prediction and a system to take data dependence into account at the instruction scheduling step.

FPGA Resource Sharing

While the principle of HLS – bringing chip design closer to their use case by embedding the process within a semantic hull – goes in the direction of hardware/software codesign, the link between the source code and generated hardware is still far from being perfect. Part II of this manuscript identifies three issues when tackling resource sharing in the HLS framework:

- *Scaling of an optimal solution and choice of the approximation.* While LP solvers and greedy scheduling under resource constraints are the two possibilities explored in this manuscript, other technical solutions must be explored. For optimal solving, non-linear solvers or SAT-solvers may exhibit easier formulation and/or solving time; while neither graph-derived scheduling algorithms nor iterative compilation techniques were explored as alternatives.
- *Interconnect cost on highly-reuse units.* Currently, the technical solution lies in implementing high fan-in multiplexers of prohibitive cost. However, control structures directly derived from the input loops show that more efficient implementations are

already known. One possible research direction lies in their efficient selection and adaptation to irregular input pattern such as those exhibited by heavy resource-shared designs.

- *General improvement of the structure of the Generic Accelerator.* Limitations cited in Sec. 8.5 raise possible ways of improvement in the creation of semi-custom accelerators. As of now, the GA is only configurable by the number and nature of FUs, as well as its memory size. Further customization of the memory controller and the scheduler should help reducing overall resource usage; while implementations with more complex FUs (e.g. vector, matrix or control-flow modifying FUs) would help pushing the spectrum of possible designs closer to general-purpose ones, offering greater flexibility to the designers.

Bibliography

[1] International Business Strategy. Cost of advances designs, 2022.

[2] Xilinx. 7 series dsp48e1 slice user guide, 2011.

[3] Sharad Sinha, Udit Dhawan, Siew Kei Lam, and Thambipillai Srikanthan. A novel binding algorithm to reduce critical path delay during high level synthesis. In *2011 IEEE Computer Society Annual Symposium on VLSI*, pages 278–283. IEEE, 2011.

[4] Sun Sik Lee, Thanh Dat Nguyen, Pramod Kumar Meher, and Sang Yoon Park. Energy-efficient high-speed asic implementation of convolutional neural network using novel reduced critical-path design. *IEEE Access*, 10:34032–34045, 2022.

[5] Alexander Marquardt, Vaughn Betz, and Jonathan Rose. Timing-driven placement for fpgas. In *Proceedings of the 2000 ACM/SIGDA eighth international symposium on Field programmable gate arrays*, pages 203–213, 2000.

[6] Khalid Javeed, Xiaojun Wang, and Mike Scott. Serial and parallel interleaved modular multipliers on fpga platform. In *2015 25th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4. IEEE, 2015.

[7] Xilinx. *Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393)*, October 2022.

[8] Abhishek Kumar Jain, Suhaib A Fahmy, and Douglas L Maskell. Efficient overlay architecture based on dsp blocks. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 25–28. IEEE, 2015.

[9] Jason Cong, Muhuan Huang, Bin Liu, Peng Zhang, and Yi Zou. Combining module selection and replication for throughput-driven streaming programs. In *2012 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1018–1023. IEEE, 2012.

[10] Andrew Canis, Jason H Anderson, and Stephen D Brown. Multi-pumping for resource reduction in fpga high-level synthesis. In *2013 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 194–197. IEEE, 2013.

[11] Zen 3 - microarchitectures - amd.

[12] Arm. Arm® cortex®-a72 mpcore processor technical reference manual, 2016.

[13] Arm Limited. Armv8-a instruction set architecture.

[14] Intel Corporation. Intel 64 and ia-32 architectures optimization reference manual.

[15] Andreas Abel and Jan Reineke. uica: Accurate throughput prediction of basic blocks on recent intel microarchitectures. In *Proceedings of the 36th ACM International Conference on Supercomputing*, pages 1–14, 2022.

[16] Agner Fog. Instruction tables: Lists of instruction latencies, through-puts and micro-operation breakdowns for intel, AMD and VIA CPUs, 2020.

[17] Amogh Akshintala, Bhushan Jain, Chia-Che Tsai, Michael Ferdman, and Donald E Porter. X86-64 instruction usage among c/c++ applications. In *Proceedings of the 12th ACM International Conference on Systems and Storage*, pages 68–79, 2019.

[18] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *40th IEEE Symposium on Security and Privacy (S&P'19)*, 2019.

[19] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *27th USENIX Security Symposium (USENIX Security 18)*, 2018.

[20] Andrei Frumusanu Ian Cutress. Intel architecture day 2021: Alder lake, golden cove, and gracemont detailed.

[21] André Seznec. Exploring branch predictability limits with the mtage+ sc predictor. In *5th JILP Workshop on Computer Architecture Competitions (JWAC-5): Championship Branch Prediction (CBP-5)*, page 4, 2016.

[22] Sparsh Mittal. A survey of techniques for dynamic branch prediction. *Concurrency and Computation: Practice and Experience*, 31(1):e4666, 2019.

[23] Ilias Vougioukas, Nikos Nikoleris, Andreas Sandberg, Stephan Diestelhorst, Bashir M Al-Hashimi, and Geoff V Merrett. Brb: Mitigating branch predictor side-channels. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 466–477. IEEE, 2019.

[24] Intel Corporation. Galois field new instructions (gfni) technology guide.

[25] Israel Hirsh and Gideon S. Intel® architecture code analyzer.

[26] Intel Corporation. Intel deep learnin boost product overview.

[27] Fabian Gruber. *Performance Debugging Toolbox for Binaries: Sensitivity Analysis and Dependence Profiling.* PhD thesis, Université Grenoble Alpes, 2019.

[28] Arnaldo Carvalho de Melo. The new linux'perf'tools. In *Slides from Linux Kongress*, volume 18, 2010.

[29] P. Mucci et al. Performance Application Programming Interface. http://icl.cs.utk.edu/papi/people/index.html. Last accessed November 2017.

[30] Torbjörn Granlund. Instruction latencies and throughput for AMD and intel x86 processors, 2017.

[31] University of Illinois. Low-Level Virtual Machine. www.llvm.org.

[32] Craig Topper. Update to the llvm scheduling model for intel sandy bridge, haswell, broadwell, and skylake processors, March 2018.

[33] Andreas Abel and Jan Reineke. uops.info: Characterizing latency, throughput, and port usage of instructions on intel microarchitectures. In Iris Bahar, Maurice Herlihy, Emmett Witchel, and Alvin R. Lebeck, editors, *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2019*, pages 673–686, New York, NY, USA, April 2019. ACM.

[34] Andreas Abel and Jan Reineke. nanoBench: A low-overhead tool for running microbenchmarks on x86 systems. *arXiv e-prints*, abs/1911.03282, 2019.

[35] Yishen Chen, Ajay Brahmakshatriya, Charith Mendis, Alex Renda, Eric Atkinson, Ondřej Sỳkora, Saman Amarasinghe, and Michael Carbin. Bhive: A benchmark suite and measurement framework for validating x86-64 basic block performance models. In *2019 IEEE International Symposium on Workload Characterization (IISWC)*, pages 167–177. IEEE, 2019.

[36] Andreas Abel and Jan Reineke. nanobench: A low-overhead tool for running microbenchmarks on x86 systems. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 34–46. IEEE, 2020.

[37] Fabian Ritter and Sebastian Hack. Pmevo: portable inference of port mappings for out-of-order processors by evolutionary optimization. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020*, pages 608–622, New York, USA, June 2020. ACM.

[38] Jan Laukemann, Julian Hammert, Johannes Hofmann, Georg Hager, and Gerhard Wellein. Automated instruction stream throughput prediction for intel and AMD microarchitectures. In *2018 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS)*, pages 121–131, Dallas, TX, USA, November 2018. IEEE Computer Society, ACM.

[39] Julian Hammer, Jan Eitzinger, Georg Hager, and Gerhard Wellein. Kerncraft: A tool for analytic performance modeling of loop kernels. In *Tools for High Performance Computing 2016*, volume abs/1702.04653, pages 1–22, Cham, 2017. Springer International Publishing.

[40] Jan Treibig, Georg Hager, and Gerhard Wellein. LIKWID: A lightweight performance-oriented tool suite for x86 multicore environments. In Wang-Chien Lee and Xin Yuan, editors, *39th International Conference on Parallel Processing (ICPP) Workshops 2010*, pages 207–216, San Diego, California, USA, September 2010. IEEE Computer Society.

[41] Andres Charif Rubial, Emmanuel Oseret, Jose Noudohouenou, William Jalby, and Ghislain Lartigue. CQA: A code quality analyzer tool at binary level. In *21st International Conference on High Performance Computing, HiPC 2014*, pages 1–10, Goa, India, December 2014. IEEE Computer Society.

[42] Lamia Djoudi, Jose Noudohouenou, and William Jalby. The design and architecture of MAQAOAdvisor: A live tuning guide. In P. Sadayappan, Manish Parashar, Ramamurthy Badrinath, and Viktor K. Prasanna, editors, *Proceedings of the 15th International Conference on High Performance Computing*, volume 5374 of *HiPC 2008*, pages 42–56, Berlin, Heidelberg, December 2008. Springer-Verlag.

[43] Sony Corporation and LLVM Project. LLVM machine code analyzer.

[44] Charith Mendis, Alex Renda, Saman P. Amarasinghe, and Michael Carbin. Ithemal: Accurate, portable and fast basic block throughput estimation using deep neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019*, volume 97 of *Proceedings of Machine Learning Research*, pages 4505–4515, Long Beach, California, USA, June 2019. PMLR.

[45] Alex Renda, Yishen Chen, Charith Mendis, and Michael Carbin. Difftune: Optimizing CPU simulator parameters with learned differentiable surrogates. In *53rd Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2020, Athens, Greece, October 17-21, 2020*, pages 442–455. IEEE, 2020.

[46] Andres Charif Rubial, Emmanuel Oseret, Jose Noudohouenou, William Jalby, and Ghislain Lartigue. CQA: A code quality analyzer tool at binary level. In *21st International Conference on High Performance Computing, HiPC 2014*, pages 1–10, Goa, India, December 2014. IEEE Computer Society.

[47] Frank Nielsen. *Hierarchical Clustering*, pages 195–211. 02 2016.

[48] Gurobi. Gurobi optimization.

[49] James Bucek, Klaus-Dieter Lange, and Jóakim von Kistowski. SPEC CPU2017: Next-generation compute benchmark. In Katinka Wolter, William J. Knottenbelt, André van Hoorn, and Manoj Nambiar, editors, *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering, ICPE 2018*, pages 41–42. ACM, April 2018.

[50] L.-N. Pouchet. PolyBench: The Polyhedral Benchmarking suite, version PolyBench/C 4.2.1. http://polybench.sf.net, 2011. Last accessed: May 2017.

[51] Intel Corporation. Intel x86 encoder decoder (intel xed).

[52] QEMU: the FAST! processor emulator. https://www.qemu.org.

[53] Maurice G Kendall. A new measure of rank correlation. *Biometrika*, 30(1/2):81–93, 1938.

[54] 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.

[55] Xilinx. Vitis unified software platform, 2022. https://www.xilinx.com/products/design-tools/vitis/vitis-platform.html.

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

[57] Intel. High level synthesis compiler, 2022. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/hls-compiler.html.

[58] Corentin Ferry, Tomofumi Yuki, Steven Derrien, and Sanjay Rajopadhye. Increasing fpga accelerators memory bandwidth with a burst-friendly memory layout. *arXiv preprint arXiv:2202.05933*, 2022.

[59] Peng Li, Louis-Noël Pouchet, and Jason Cong. Throughput optimization for high-level synthesis using resource constraints. In *Int. Workshop on Polyhedral Compilation Techniques (IMPACT'14)*, 2014.

[60] 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)*, pages 1–8. IEEE, 2018.

[61] Xilinx. *UltraScale Architectuge Configurable Logic Block User Guide (UG574)*, January 2022.

[62] Peng Li, Peng Zhang, Louis-Noel Pouchet, and Jason Cong. Resource-aware throughput optimization for high-level synthesis. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, page 200–209, New York, NY, USA, 2015. Association for Computing Machinery.

[63] Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. Fpga hls today: successes, challenges, and opportunities. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 15(4):1–42, 2022.

[64] Xilinx. *Vitis High-Level Synthesis User Guide (UG1399)*, October 2022.

[65] Paul Feautrier. Dataflow analysis of array and scalar references. *International Journal of Parallel Programming*, 20(1):23–53, 1991.

[66] Edward A Lee and David G Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[67] zipcpu. Fixing xilinx's broken axi-lite design in vhdl, May 2021.

[68] Thibaut Marty, Tomofumi Yuki, and Steven Derrien. Safe overclocking for cnn accelerators through algorithm-level error detection. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(12):4777–4790, 2020.

[69] Xiangwei Li, Abhishek Kumar Jain, Douglas L Maskell, and Suhaib A Fahmy. A time-multiplexed fpga overlay with linear interconnect. In *2018 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1075–1080. IEEE, 2018.

[70] Atefeh Sohrabizadeh, Cody Hao Yu, Min Gao, and Jason Cong. Autodse: Enabling software programmers to design efficient fpga accelerators. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 27(4):1–27, 2022.

[71] Shuangnan Liu, Francis CM Lau, and Benjamin Carrion Schafer. Accelerating fpga prototyping through predictive model-based hls design space exploration. In *2019 56th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2019.

[72] Charles Y Hitchcock and Donald E Thomas. A method of automatic data path synthesis. In *20th Design Automation Conference Proceedings*, pages 484–489. IEEE, 1983.

[73] Salil Raje and Reinaldo A Bergamaschi. Generalized resource sharing. In *Proceedings of the 1997 IEEE/ACM international conference on Computer-aided design*, pages 326–332. Citeseer, 1997.

[74] Welson Sun, Michael J Wirthlin, and Stephen Neuendorffer. Fpga pipeline synthesis design exploration using module selection and resource sharing. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 26(2):254–265, 2007.

[75] Lana Josipović, Andrea Guerrieri, and Paolo Ienne. Invited tutorial: Dynamatic: From c/c++ to dynamically scheduled circuits. In *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 1–10, 2020.

[76] B Ramakrishna Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74, 1994.

[77] Hanchen Ye, Cong Hao, Jianyi Cheng, Hyunmin Jeong, Jack Huang, Stephen Neuendorffer, and Deming Chen. Scalehls: A new scalable high-level synthesis framework on multi-level intermediate representation. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pages 741–755. IEEE, 2022.

[78] Lana Josipović, Axel Marmet, Andrea Guerrieri, and Paolo Ienne. Resource sharing in dataflow circuits. In *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–9. IEEE, 2022.

[79] Bajaj Ronak and Suhaib A Fahmy. Multipumping flexible dsp blocks for resource reduction on xilinx fpgas. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(9):1471–1482, 2016.

[80] 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*, pages 33–36, 2011.

[81] Abhishek Kumar Jain, Douglas L Maskell, and Suhaib A Fahmy. Throughput oriented fpga overlays using dsp blocks. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1628–1633. IEEE, 2016.

[82] IBM. Ibm ilog cplex optimization studio.

[83] Benjamin Carrion Schafer. Enabling high-level synthesis resource sharing design space exploration in fpgas through automatic internal bitwidth adjustments. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(1):97–105, 2016.

[84] Louis-Noël Pouchet. PoCC, the Polyhedral Compiler Collection 1.3, 2022. Ver 1.6.

[85] Jason Cong, Muhuan Huang, Peichen Pan, Di Wu, and Peng Zhang. Software infrastructure for enabling fpga-based accelerations in data centers. In *Proceedings of the 2016 International Symposium on Low Power Electronics and Design*, pages 154–155, 2016.

[86] Xilinx. The merlin compiler. https://github.com/Xilinx/merlin-compiler, 2022.

[87] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In

2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC), pages 1–6. IEEE, 2018.

[88] Sihao Liu, Jian Weng, Dylan Kupsh, Atefeh Sohrabizadeh, Zhengrong Wang, Licheng Guo, Jiuyang Liu, Maxim Zhulin, Rishabh Mani, Lucheng Zhang, et al. Overgen: Improving fpga usability through domain-specific overlay generation. In *2022 55th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 35–56. IEEE, 2022.

[89] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, et al. A hardware–software blueprint for flexible deep learning specialization. *IEEE Micro*, 39(5):8–16, 2019.

[90] Louis-Noël Pouchet Nicolas Derumigny and Fabrice Rastello. Kernel merging for throughput-oriented accelerator generation. In *Proceedings of the 13th International Workshop on Polyhedral Compilation Techniques (IMPACT)*, 2023.

[91] Andrei Hagiescu, Weng-Fai Wong, David F Bacon, and Rodric Rabbah. A computing origami: Folding streams in fpgas. In *Proceedings of the 46th Annual Design Automation Conference*, pages 282–287, 2009.

[92] Nicholas Weaver. Retiming, repipelining and c-slow retiming. *Reconfigurable Computing*, pages 383–399, 2008.

[93] Heidi Ziegler, Byoungro So, Mary Hall, and Pedro C Diniz. Coarse-grain pipelining on multiple fpga architectures. In *Proceedings. 10th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 77–86. IEEE, 2002.

[94] C. Bastoul. Code generation in the polyhedral model is easier than you think. In *13th International Conference on Parallel Architectures and Compilation Techniques, PACT*, pages 7–16, Antibes, France, 2004. IEEE.

[95] Sylvain Girbal, Nicolas Vasilache, Cédric Bastoul, Albert Cohen, David Parello, Marc Sigler, and Olivier Temam. Semi-automatic composition of loop transformations. *International Journal of Parallel Programming*, 34(3):261–317, June 2006.

[96] P. Feautrier. Some efficient solutions to the affine scheduling problem, part II: multidimensional time. *International Journal of Parallel Programming*, 21(6):389–420, 1992.

[97] C. Bastoul. Candl: Data dependence analysis tool in the polyhedral model.

[98] Cédric Bastoul. Improving data locality in static control programs. 2004.

[99] Denis Barthou, Albert Cohen, and Jean-François Collard. Maximal static expansion. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 98–106, 1998.

[100] Xilinx. *UltraScale Architecture Configuration User Guide (UG570)*, January 2022.

[101] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. A hardware–software blueprint for flexible deep learning specialization. *IEEE Micro*, 39(5):8–16, 2019.

[102] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, and Glenn Reinman. Charm: A composable heterogeneous accelerator-rich microprocessor. In *Proceedings of the 2012 ACM/IEEE International Symposium on Low Power Electronics and Design*, ISLPED '12, page 379–384, New York, NY, USA, 2012. Association for Computing Machinery.

[103] Steven Margerm, Amirali Sharifian, Apala Guha, Arrvindh Shriraman, and Gilles Pokam. Tapas: Generating parallel accelerators from parallel programs. In *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 245–257, 2018.

[104] Cheng Liu, Ho-Cheung Ng, and Hayden Kwok-Hay So. Quickdough: A rapid fpga loop accelerator design framework using soft cgra overlay. In *2015 International Conference on Field Programmable Technology (FPT)*, pages 56–63, 2015.

[105] A. Mkhinini, P. Maistri, R. Leveugle, and R. Tourki. Hls design of a hardware accelerator for homomorphic encryption. In *2017 IEEE 20th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pages 178–183, 2017.

[106] Xinyu Chen, Hongshi Tan, Yao Chen, Bingsheng He, Weng-Fai Wong, and Deming Chen. Thundergp: Hls-based graph processing framework on fpgas. In *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '21, page 69–80, New York, NY, USA, 2021. Association for Computing Machinery.

[107] Yijin Guan, Hao Liang, Ningyi Xu, Wenqiang Wang, Shaoshuai Shi, Xi Chen, Guangyu Sun, Wei Zhang, and Jason Cong. Fp-dnn: An automated framework for mapping deep neural networks onto fpgas with rtl-hls hybrid templates. In *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 152–159, 2017.

[108] Stefan Abi-Karam, Yuqi He, Rishov Sarkar, Lakshmi Sathidevi, Zihang Qiao, and Cong Hao. Gengnn: A generic FPGA framework for graph neural network acceleration. *CoRR*, abs/2201.08475, 2022.

[109] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Haichen Shen, Eddie Q Yan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. Tvm:

end-to-end optimization stack for deep learning. *arXiv preprint arXiv:1802.04799*, 11(2018):20, 2018.

[110] Xinyi Zhang, Weiwen Jiang, and Jingtong Hu. Achieving full parallelism in lstm via a unified accelerator design. In *2020 IEEE 38th International Conference on Computer Design (ICCD)*, pages 469–477, 2020.

[111] Jason Cong, Peng Wei, Cody Hao Yu, and Peng Zhang. Automated accelerator generation and optimization with composable, parallel and pipeline architecture. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pages 1–6, 2018.

[112] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. Dsagen: Synthesizing programmable spatial accelerators. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 268–281. IEEE, 2020.

[113] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, pages 924–939, 2019.

[114] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert Ou, Nathan Pemberton, et al. Chipyard: Integrated design, simulation, and implementation framework for custom socs. *IEEE Micro*, 40(4):10–21, 2020.

[115] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. There's plenty of room at the top: What will drive computer performance after moore's law? *Science*, 368(6495):eaam9744, 2020.

[116] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 simulator. *ACM SIGARCH Computer Architecture News*, 39(2):1–7, 2011.