Acceleration for the many, not the few

Jackson Woodruff

Doctor of Philosophy

Institute of Computing Systems Architecture
School of Informatics
University of Edinburgh

2024

Abstract

Although specialized hardware promises orders of magnitude performance gains, their uptake has been limited by how challenging it is to program them. Hardware accelerators present challenges programmers are not used to, exposing details of the hardware that are often hidden and requiring new programming styles to use them effectively.

Existing programming models often involve learning complex and hardware-specific APIs, using Domain Specific Languages (DSLs), or programming in customized assembly languages. These programming models for hardware accelerators present a significant challenge to uptake: a steep, unforgiving, and untransferable learning curve. However, programming hardware accelerators using traditional programming models presents a challenge: mapping code not written with hardware accelerators in mind to accelerators with restricted behaviour.

This thesis presents these challenges in the context of the *acceleration equation*, and it presents solutions to it in three different contexts: for regular expression accelerators, for API-programmable accelerators (with Fourier Transforms as a key case-study) and for heterogeneous coarse-grained reconfigurable arrays (CGRAs). This thesis shows that automatically morphing software written in traditional manners to fit hardware accelerators is possible with no programmer effort and that huge potential speedups are available.

Acknowledgements

I would like to thank my advisor, Michael O'Boyle for his endless support and guidance throughout my PhD. I would also like to thank my assistant advisor, Sam Ainsworth for his detailed and on-the-money feedback and advice. In addition, I need to thank my co-authors, Jordi, Thomas, Alex, Chris and Michel, without whom the papers comprising this thesis would be incomplete.

Thanks to my various pre- and post-covid officemates, Maurice, Celeste, Martynas and David for making my time here fun. I'd doubly like to thank David for making the office smell more than me. I would like to thank my running and cycling buddies, particularly Kai, Patrick and Tana among many others. Tana would like to use this space to point out that "I am uninterested in your work, but interested in you as a person".

Finally, I'd like to thank my parents who have provided many hours of support over the phone.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified. Some of the material used in this thesis has been published in the following papers and preprints:

1. Jackson Woodruff, Michael F.P. O'Boyle, New Regular Expressions on Old Accelerators. 58th Design Automation Conference, 2021.

2. Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, Michael F.P. O'Boyle, Bind the Gap: Compiling Real Software to Hardware FFT Accelerators. PLDI 2022.

3. Jackson Woodruff, Thomas Kœhler, Alexander Brauckmann, Chris Cummins, Sam Ainsworth, Michael F.P. O'Boyle, Rewriting History: Repurposing Domain-Specific CGRAs. CoRR, 2023. Available at `https://arxiv.org/pdf/2309.09112.pdf`.

Table of Contents

List of Figures

Lay Summary

Computer hardware has changed dramatically over the past 15 years, and the software we use to program this hardware has not kept pace. With changing hardware, computer programmers spend increasing amounts of time and energy programming complex computer hardware to keep their software running at the highest performance possible. However, this workload is becoming unsustainable, with each hardware company providing different ways to program their hardware.

This thesis introduces a mathematical framework to address this problem, called the accelerator equation, and solves it in three unique cases. The accelerator equation generalizes the idea that hardware designers and programmers have different ideas about what their programs do, and provides a practical way to make sure that advanced hardware can do what programmers want it to, without requiring the hours and hours of research and understanding typically required for this advanced hardware.

This thesis looks at three key case studies. It first looks at hardware designed for pattern-matching in text, solving the accelerator equation in this case. It subsequently looks at solutions to the accelerator equation for Fourier Transforms, a mathematical construct frequently used in signal processing. Finally, it sets the groundwork to apply the accelerator equation for an entire class of advanced hardware called CGRAs (Coarse-Grained Reconfigurable Accelerators).

In summary, this thesis sets out to address the challenges that modern hardware poses to programmers. It outlines the difference in how hardware designers and software programmers think about and express problems as a challenge to be overcome. It introduces a mathematical framework for addressing this problem and demonstrates that this framework is practical with a number of case studies.

Chapter 1

Introduction

Running software on hardware accelerators promises more performance than *generations* of software optimization [184]. With increased transistor densities [29] and more challenges powering these transistors [171], hardware accelerators are set to become even more prominent [474].

However, programming these accelerators is challenging [143]. Programs often require manual modification to run on hardware accelerators [511], which defeats one of the biggest benefits of compilers: portability. Existing approaches to hardware accelerator support largely focus on support from DSLs [10, 402] or pattern-matching [189]. However, DSL approaches do not easily support all kinds of accelerators, and pattern-matching is very brittle [127]. Figure 1.1 shows where current techniques fall short — a gap this thesis fills.

1.1 Limits of Existing Strategies

Traditional compilation techniques (Section 3.3) are a common solution; but for accelerators whose behaviour is not reprogrammable in a fine-grained manner, these are challenging to develop [190] and frequently fail [530] for these types of hardware accelerator. Lifting strategies [249, 341] are popular, but are again limited when code and accelerator do not overlap in behaviour exactly.

The lack of an easy, portable and automated way of programming hardware accelerators has restricted their development and potential. The recent explosion of machine-learning accelerators is testament to this. Tensor programs, written using libraries such as Torch can be compiled to hardware accelerators using traditional compilation strategies that break programs into smaller, more suitable-for-hardware

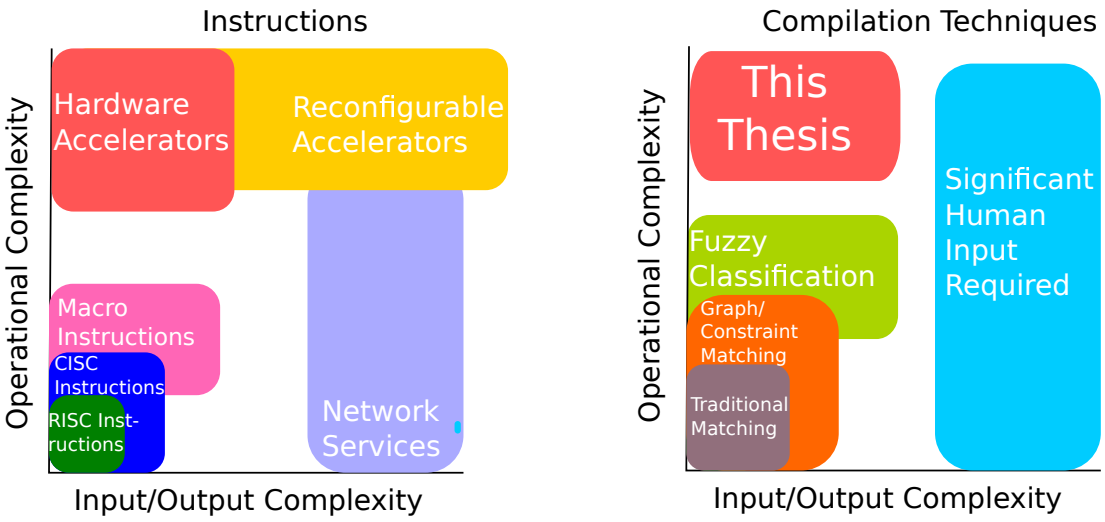


Figure 1.1: The space of accelerators (left) and compiler strategies (right) categorized by the number of steps in each step (Operational Complexity) and how difficult is to communicate with the accelerator (Input/Output Complexity). The smallest step and simplest to communicate with instructions are supported by traditional compilation strategies. Compiler support for big-step accelerators that perform a large number of operations but present simple interfaces is limited, despite their potential to provide speedup. This thesis addresses the lack of compiler techniques in this space.

components. However, for legacy programs, and for technologies without these standardised programming interfaces designing hardware accelerators leaves a key usability question.

1.2 Effects of Lack of Compiler Strategies

This has effects we see in practice. Thousands of papers [387, 408, 409] and entire conferences are dedicated to the development of specialized hardware accelerators, yet the vast majority of these are not further developed — in no small part due to concerns that they will not be used enough to be viable. A key reason for this concern is that hardware accelerators and software behaviour often do not match exactly, creating a situation where the volume of software a hardware accelerator can accelerate is not well-understood, and defeating existing compilation strategies [530].

1.3 Contributions

This thesis introduces a number of techniques to enable easy use of hardware accelerators. This thesis first introduces the *acceleration* equation (Chapter 2.1), a set of equations that, once solved, enable wider accelerator use. Solutions to these equations capture the diversity of solutions that programmers come up with when presented with the flexibility of programming languages, while using the available hardware accelerators to present the flexibility of high-level software and performance of specialized hardware.

We look at three key case studies of these equations, looking at regular expression accelerators in Chapter 4, FFT accelerators in Chapter 5 and domain-specific CGRAs in Chapter 6.

This thesis makes the following key contributions:

- It introduces the accelerator equation (Chapter 2) that can be solved to enable the use of hardware accelerators.
- It introduces three toolchains (Chapters 4, 5, 6) that demonstrate the practical applicability of these equations in enabling the use of hardware accelerators.

Aside from these key contributions, this thesis introduces key benchmark suites and a novel manner of benchmarking these changes, focused on compiler performance across a wide range of kernels rather than finely tuning the compiler for a few kernels. To explore FFTs, we introduce a benchmark suite of 24 different FFT implementations: critical in enabling evaluations of our compiler. Similarly, to evaluate FlexC, we introduce a benchmark suite of more than 2,000 loops to enable a broad analysis of our compiler. These benchmark suites are released as part of this thesis.

The challenges that this dissertation addresses open up a new direction of compilers for hardware accelerators: using compiler technology to make the next generation of fast hardware as flexible as the programmers require.

Chapter 2

The Acceleration Equation

Hardware accelerators and application behaviour often do not exactly line up. When hardware accelerators are designed with a single application in mind, it is easy to ensure that the behaviours are identical: but when accelerators are pre-designed, changes in the computation requirements result in accelerators that no longer match the programmer's code exactly. Figure 2.1 shows an example of this: a specialized bit of programmer code for reverse-sorting byte lists cannot be directly replaced with a more generic sorting accelerator.

This chapter will introduce two concepts central to understanding the problem: a theoretical framework (the acceleration equation) and an overview of existing programming models.

2.1 The Acceleration Equation Definition

To capture differences between accelerator behaviour and code behaviour in a constructive manner, we introduce the *accelerator equation*:

$$U = g \circ A \circ h \tag{2.1}$$

In this equation, A is a function representing the accelerator, U is the function performed by the user code, and g and h are the functions that enable solutions to this

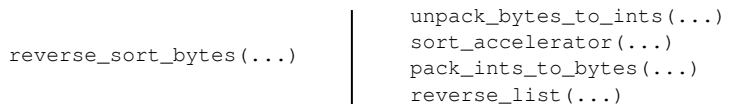


Figure 2.1: Example solution to the accelerator equation: a sorting accelerator for integers used to reverse sort bytes.

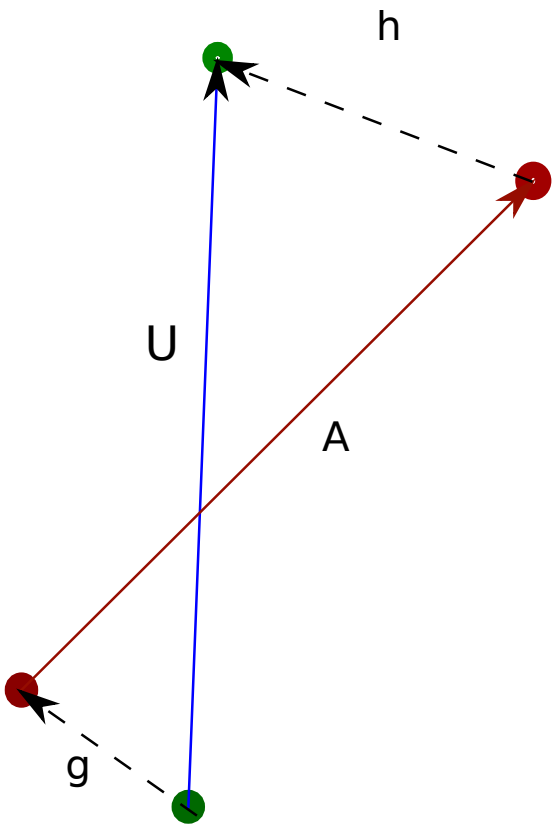


Figure 2.2: Supporting user code (blue line) that performs a different function to an accelerator, A (red line) using the accelerator equation. The functions g and h are used to wrap around the accelerator function, making it possible to go between the same two points that the original function U computed by using the fast accelerated function A instead. In this diagram, the green circles are ordered sets of values, and each function is an ordered mapping between those sets, mapping the n^{th} element to each other. In general, to solve the acceleration equation, we are given some user code U and some accelerator function A and need to find the f and g.

equation. Solutions to this equation are constructive because we can simply run g and h, enabling the use of the hardware accelerator to run the function U. Figure 2.2 shows an example, with g and h represented by dashed lines.

To find a solution to the accelerator equation, given some user code U and some accelerator function A, we must find the functions f and g so that equation 2.1 holds. Finding a solution to this equation enables the replacement of user code with faster code that relies on an accelerator instead.

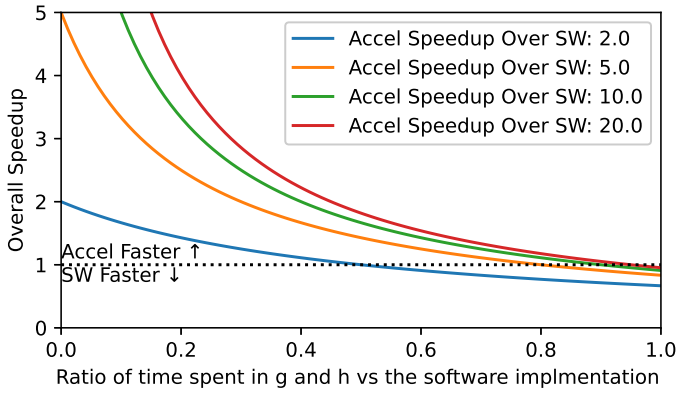


Figure 2.3: Speedup considering the overhead of g and h against the original software implementation. Each line represents an accelerator where g and h have different relative performance to user code. We can see that even if g and h take 50% as much time as the *entire* original program, if the accelerator provides 5x speedup, we can pass on a ˜1.5x speedup to the programmer.

2.1.1 Overheads of the Acceleration Equation

These functions, f and g, add overheads to running accelerators. However, provided that the accelerator provides significant speedup, these approaches can still pass on large speedups to programmers. Figure 2.3 shows the speedups available over user code. Each line represents an accelerator with different relative performance to user code. We can see that even if g and h take 50% as much time as the *entire* original program, if the accelerator provides 5x speedup, we can pass on a ˜1.5x speedup to the programmer. This indicates that we can still find significant speedup for the programmer, even if the functions required to adapt the accelerator appropriately are long and complex.

2.2 Conclusion

This chapter introduces the *acceleration* equation, an equation generalizing the challenges of compiling for specialized hardware accelerators. We extend this equation to the reconfigurable accelerator equation, which allows us to capture the challenges of compiling to reconfigurable hardware accelerators. The rest of this thesis will explore solutions to these accelerator equations.

Chapter 3

Related Work

This chapter covers the related work to this thesis. In Section 3.2, we cover a range of hardware accelerators, with particular focuses on the types of accelerators used as case studies in this thesis. In Section 3.3, we cover the existing compilation techniques used for these hardware accelerators, along with approaches that are further developed in this thesis.

3.1 More Moore

Despite being in an era of heterogeneous hardware [217], the levels of dark silicon promised by decade-old techniques [170, 250, 474] have not emerged. Considering only power consumption would predict 80% dark silicon at existing process nodes [170]. However, heat dissipation is the key limiting metric here, and there are three key ways to increase heat dissipation: increasing spacing between components, using DVFS (Dynamic Voltage and Frequency Scaling) and running non-compute components (e.g. memories) at lower frequencies.

These techniques clearly have their limits. For example, increasing spacing between high-frequency components increase communication costs between these elements. Likewise, despite DVFS becoming extremely fine-grained [196], it can lead to difficult-to-track performance problems [280].

Online articles [353] discuss dark silicon under the term "white space" in which interviews with semiconductor companies reveal 75% utilization targets. With these 75% utilization targets, Moyer's interviewees seem to be filling their white space with verification logic.

3.1.1 Why More Dark Silicon is on the way

There are two key reasons why we see more dark silicon on the way. The first is the continuation of Moore's law, and the second is the introduction of stacked chips.

Despite frequently voiced concerns over the health of Moore's law, at least for the foreseeable future, semiconductor company technology road maps[1] continue to predict reducing technology feature sizes. Further, post-FinFET transistor designs are already being discussed [426], and such technologies give rise to the possibility of continued transistor scaling. The IRDS roadmap for 2020 [29] is simply subtitled "More Moore", and contains estimates for transistor density increasing using GAA (gate all-around) transistors and 3D stacking up until 2034.

The Challenges of Manufacturing Dark Silicon Adding dark silicon makes chips larger, which adds significant technical challenges:

1. Enabling large chips to be used within existing lithgraphy tools.
2. Solving the yield problem without vastly over-provisioning redundancy.
3. Solving packaging issues.

Fortunately for accelerators, such problems are being addressed by existing chip designers [295].

3.1.2 On Accelerators without Dark Silicon

Even in a regime in which we are faced with decreasing processor fixed-function hardware, accelerators provide a significant step forward. For example, the Analog Devices FFTA [19] operates at half frequency of the core while still out-performing it by a factor of five [23] enabling both power-saving and performance gains.

It seems that predictions by Esmaeilzadeh et al. [170] have not come true due to a number of architectural techniques. However, with 3D-stacked-fabrication techniques in the IRDS pipeline, and no specified solutions to the enhanced heat-dissipation problems this will entail [29], it seems that dark silicon is still likely.

3.2 Hardware Accelerators

Large-scale accelerators are often designed with a particular application in mind. Many are wrapped as "xPUs" [537], such as TPUs for tensors [246], IPUs for images [429], HPUs for holographics [343] and EPUs for emotions [168]. Generally, these xPUs are

[1] `https://www.tsmc.com/english/dedicatedFoundry/technology/future_rd`

marketing terminology and lack publicly available details implementations — but they do demonstrate a wide potential for hardware accelerators.

Hardware accelerators are commonplace in large companies, with Meta [299], Google [421] and Microsoft [343] all developing their own accelerators, and hardware companies such as Intel [352] and Xilinx [538] developing a wide variety of hardware accelerators for tasks from machine learning to key-value stores.

These hardware accelerators exist at a variety of scales, from H.264 accelerators that exist in large blocks, to Xilinx's Vitis libraries that are intended to be used as small accelerated components and reduce programming complexity [185] e.g. implementing BLAS [541] or matrix algorithms [539]. Nvidia [375] and Intel [236] both provide image processing accelerators. Even these large-scale image processing accelerators have the potential to provide reusable blocks [248]. Intel [237] and Texas Instruments [235] provide network function accelerators, and a large number of DSP companies provide accelerator for the FFT-family of functions [15, 125].

A vast number of startups are/have recently been designing hardware accelerators from regex acceleration [157, 202, 480] Blockchain [435], Cryptography [441], video codecs [98, 99, 438–440, 443], image processing [97, 101, 137] network function offloading [201, 203, 442], networking [100, 436, 437], compression [102], query processing [77, 204, 205], finance [451, 465], automotive industries [307] and bioinformatics [151, 391] among other algorithms [58].

3.2.1 Generating Accelerators

Generating customized accelerators from source code is a common way to design an accelerator for a particular application. Researchers have explored manually finding accelerators [407] within the SHOC benchmark suite [144]. But the most common approach is to profile and then extract [287, 288, 559]. Function merging [87] and architectural optimizations [423] can be used to make the accelerators more profitable. Conservation cores [493] take the automated approaches to the extreme, accelerating 95% of the Android system using 43,000 static instructions [197].

Pragma/OpenMP-style offload is also a popular strategy to offload high-level code onto FPGAs [334]. Compilers such as LegUp [94] and ROCCC [496] take similar pragma-based approaches to enabling FPGA compilation.

3.2.2 Fixed Function Accelerators

Fixed-function accelerators tend to provide the best energy-efficiencies [300], and have been explored for checksums [61, 479], network stacks [283], sorting [399], database query acceleration [210, 247, 533] and regular expressions [194, 535], among many other types of accelerator. Many of the explored academic accelerators do not have the profitability constraints required by real hardware accelerators, making them interesting compilation targets, but practially infeasible [264].

3.2.3 Reconfigurable Accelerators

Reconfigurable accelerators differ from programmable accelerators because there is a single reconfiguration phase that happens before the accelerator starts running. However, there are a number of examples of coarse-grained reconfigurable accelerators, most notably Plasticine [397]. There are many such academic examples of coarse-grained reconfigure architectures [32, 338, 339, 406, 452, 463, 473, 526] Reconfigurable cores for certain tasks have also been explored. LAC [388] is a linear-algebra core designed to support linear-algebra implementations.

Xilinx are moving towards this domain with their Adaptive Compute Acceleration Platform (ACAP) [540]. A number of tightly coupled reconfigurable accelerators have been available for some time. Processors such as Tensilica's Xtensa [30], and Arm's customizable processors [119] offer such acceleration. Compiler work to enable use of these (re)configuable processors typically relies on profiling [123], similar to the FPGA work explored earlier.

FPGAs in the Cloud FPGAs are the most common type of reconfigurable accelerator, and are common in both the Azure [104] and AWS [49] clouds. Research surrounding these accelerators focuses on the interconnects [335], access methods [556] and costs [567] among the development of many FPGA-specific accelerators discussed above.

3.2.4 CGRAs

Research on CGRAs has been extensive [314, 329]. Older CGRAs [198, 215, 339] tend to provide a homogeneous grid of PEs (Processing Elements) with a programmable interconnect. However, the design-space of CGRAs is immense, including heterogeneous on-chip networks [35, 255, 325, 519, 566], decouplings of memory and com-

pute [516, 520], unifying memories [141], and various techniques to specialize PEs [55, 90, 192, 340, 383, 455, 523, 565]. To handle these proposals, toolchains have been developed to enable the development of CGRAs [116, 207, 309, 395, 444, 467, 468, 521] and to aid design decisions [66, 158, 515], meaning it is relatively easy to design and build a domain-specific CGRA.

Domain-Specific CGRAs Domain-specific CGRAs have been designed for a wide range of applications including neural networks [22, 56, 188, 298], scientific kernels [90, 108, 152, 164, 398], approximate computing [39, 508] and streaming applications [306]. Domain-specific CGRAs have also been explored for stencil computations [481], HPC [325], signal processing [313] and multimedia [337, 364, 551].

Industrial CGRAs Xilinx's ACAP provides a CGRA-like model of computation [542] and Xilinx have developed an MLIR-based toolchain for this engine [543]. Wave Computing [368] develop a very high frequency CGRA using a clustered approach. SambaNova Systems have designed a high-performance CGRA [22] targeting ML-workloads. Samsung have designed the SLP-URP [265] for low-power medical use-cases. Recore Systems have developed a low power CGRA designed to extend battery life of embedded systems [219]. Beyond these existing industrial CGRAs, various large-scale research projects have been funded with the intent of producing real CGRA hardware [195, 378].

3.2.5 Regular Expression Accelerators

We see a number of accelerators for regular expressions (or, more generally, Automata). Several commercial IP models exist, such as the accelerators Grovf [202] and Titan IC [480] (now purchased by Mellanox). Micron also manufacture an automata processor (AP) [157]. There also exist a large number of academic accelerators designed to be run as ASICs, such as HARE [194], CICERO [384], or HAWK [470] which is not designed for high speed regex matching, but does support it. A number of ASIC accelerators are designed to couple closely with DRAM or SRAM banks [53, 175, 282, 414].

There are a range of academic works focusing on FPGA-based automata accelerators [344,382,432,550,551]. REAPR [535] is a key player in this space, having spawned various adaptations to provide more configurability [83], better output reporting [498], more complete ecosystem [403], extended debugging capabilities [96] and more efficient compilation strategies [529]. LAP [534] presents an ADFA, a model more suited to small buffer sizes that can be made fast in hardware. Karakchi *et al.* [254] implement

an FPGA overlay — a topic that also generates interest in P4-enabled devices [240].

Acceleration on CPUs is also a common topic, with application-targeted accelerators for text matching introduced in Intel's Nehalem architecture [479]. Intel's Hyperscan [507] is a CPU-based approach, capable of parallelising to achieve 100Gbps [564] given sufficient power and CPU resources. Nourian *et al.* [370] compare CPU-based models to FPGA and GPU models and conclude that FPGA models are the most efficient — although we note that this comparison was completed before Hyperscan. Veanes *et al.* [490] explore the use of Brozowzky derivatives to make the execution of regular expressions more deterministic on a CPU.

As with many accelerators, a common suggestion for overcoming the memory-wall involves in-memory computation. REACT [239] is one such proposal, using in-memory processing to achieve 6 GB/s throughput. A number of similar proposals use CAMs to achieve high energy efficiency [148, 230].

Reconfigurable Regular Expression Accelerators There is a variety of other work that looks at the idea of supporting multiple regular expressions with a single accelerator. Bo [83] looks at reprogramming the edge transition labels. Becher *et al.* [71] explores the concept of generating smaller accelerators for regular expressions that filter almost all of the data before checking the remainder on the CPU.

Our work in chapter 4 utilizes these concepts, but focuses on automating the process, rather than relying on the user for the bulk of the work to input unintuitive automata. Moussalli *et al.* [351] and Teubner *et al.* [476] explore overlays that are for database applications, and Ng *et al.* [363] explore overlays for bioinformatics applications using read alignment.

3.2.6 FFT Accelerators

Hundreds of research implementations [73, 187] and commercial implementations [3, 13, 14, 16, 17] of FFT accelerators exist intended both as stand-alone accelerators, and to be integrated in larger accelerators [482]. Work on supporting FFT acceleration exists for FPGAs [348], GPUs [318] and specialized architectures from linear algebra cores [389] to CGRAs [222, 319], machine-learning accelerators [162, 301, 450], optical computers [290] and sonic computers [386].

FFT accelerator performance is largely memory-bandwidth limited [160, 477], a problem exacerbated by access patterns that make poor use of DRAM buffers [40, 75]. Much work has been focused on reducing memory demands. Computing twiddle

factors on-chip has been explored [111, 113, 212] and applied in industry [336]. In-memory FFT accelerators have also been proposed to reduce this communication overhead [122, 302, 553] along with 3D-stacked memory accelerators [206].

3.2.7 Data-Structure Accelerators

Data-structure accelerators focus largely on data-structures that:

- Have high operational complexities (fairly uncommon)
- Require high throughputs
- Require huge volumes of memory

Accelerators have been designed for hash tables [296, 549], queues [274] and B-trees [289]. Research into distributed hash tables [80, 231] shows that these accelerators provide speedups even over the network.

3.2.8 Machine Learning Accelerators

A huge number of machine learning accelerators exist. FPGA-based accelerators frequently rely on reshaping the hardware to fit the FPGA [154, 503]. The latest generation of these accelerators tend to support sparse matrix multiplicaiton [377, 489], and they are typically programmed using APIs [226], compilation frameworks from generic APIs [112] and more generally, can be programmed using polyhedral techniques [76], or by learning programs to run on them [271].

ASICs ASICs provide almost 100x more performance than CPUs in the context of ML [85, 374] and so have been a large research focus. Beyond generic ML accelerators [312], ASIC-based ML accelerators exist for important applications [322, 475] and numerous tools exist to aid the design of these networks [31, 404, 492, 560].

In-Memory ML Accelerators Since deep learning is often memory-bound [117], a huge number of PIM (Processing-in-memory) accelerators have been proposed for GANs [328], GCNs [548], sparse matrix multiplications [536] and hyperdimensional neural networks [163]. Architectural techniques using analog operations [244] and overcoming granularity/performance tradeoffs have also been explored [563].

3.2.9 In-Memory Accelerators

More generally, in-memory accelerators exist using DRAM operations and embedded logic [50, 51, 103] and tightly coupled general purpose processors [147, 234]. Samsung

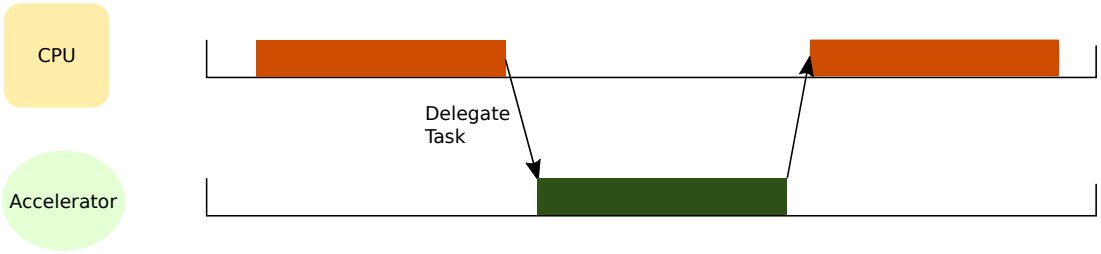


Figure 3.1: An offloading model between a CPU and an accelerator. In this example, there is only one accelerator, and there is nothing else running on the CPU, but with suitable parallelization (e.g. as explored in POAS [330]), multiple accelerators, and longer-running CPU tasks can be supported.

have developed PIM accelerators [267], and design-tools for power estimation have been developed [167].

Algorithm-Specific Accelerators Beyond these generic architectures, domain-specific PIM architectures have been developed for sorting [570], bioinformatics [120], NTT algorithms [302]. Sky-Sorter [570] introduces a PIM architecture designed for sorting.

3.2.10 Compilers for PIM Accelerators

PIM accelerators can be programmed with compiler directives [179] or traditional instructions [419]. Work on exploiting these hardware accelerators has explored the challenges of detecting regions where speedups are likely [153], compilation of DSLs [433], instruction issue costs [38] and instruction selection [37].

3.3 Accelerator Programming Models

To understand how we can automate compilation to hardware accelerators, we should understand how we can interact with these accelerators. Here we explore a model in which a chunk of the program is offloaded to, and run on, an accelerator as shown in Figure 3.1. Below we cover the three main categories of programming models for accelerators. Cãscaval et al. [95] discuss two of these, but the development of DSL-programmed accelerators has changed the options for accelerator programming. There are commonalities between these types of accelerators; the simplest DSLs are similar to APIs, and the most complex DSLs are close to C-level programmability.

3.3.1 API-Programmable Accelerators

API-programmable accelerators are good for algorithm-scale applications, where the task to accelerate forms a task that can be easily documented and easily represented using a function name.

They key benefit of an API-programmable accelerator is that it abstracts from the actual implementation — which can then be in software, hardware, or over the network [360].

3.3.2 DSL-Programmable Accelerators

DSL-programmable accelerators tend to be far more flexible. Particularly prominent accelerators that can be programmed with DSLs include Barefoot's Tofino switches [] (programmed using P4), DSPs (which can be programmed with DSLs such as Halide [402]) and GPUs (programmed using DSLs such as CUDA or OpenCL).

These DSLs serve different purposes, overcoming various issues with traditional programming languages, which are designed for CPUs:

- Explicit parallelism (e.g. CUDA/OpenCL), used to overcome challenges with auto-parallelization in languages designed for CPUs.
- Explicit structure (e.g. P4), used to overcome challenges with fitting general high-level code to structured pipelines.
- Separation of algorithm and scheduling decisions (e.g. Halide), used to overcome challenges of compiling highly optimized code to new architectures.

DSLs are common choices for more flexible accelerators because they can overcome the relevant limitations of high-level programming languages for accelerators. These DSLs tend to allow for a wide range of algorithms to be implemented on accelerators: critically, programmers can implement algorithms that accelerator designers may not have considered to be relevant when they designed the accelerators.

However, the learning curve for these kinds of accelerators is high. DSLs must be integrated into existing systems (as they often do not support the entire program that must be written); moreover, they require that programmers must learn entire languages, which may not be useful for the next-generation of accelerators. Further, these DSLs require significantly more work on the side of the company designing the hardware accelerator, requiring the development of entire compiler toolchains and significantly complex abstractions over hardware.

3.3.3 High-Level Programming Languages

Most high-level programming languages have been designed for single- or multi-core CPUs. As a result, they often encounter limitations when compiled to hardware accelerators, either in extracting parallelism or matching the structure of the hardware available appropriately. Further, in a similar manner to DSLs, code written immaculately for one kind of accelerator may not work on another kind of accelerator (e.g., we can use intrinsics to make a loop fast on AVX-512, but these make it *more* challenging to compile this code to run on a GPU).

Despite these limitations — which constitute a significant limit on the potential of accelerators using this programming method, cf. [397] and [339] — the potential for high-level language programmable accelerators is huge. They provide a low barrier to entry, as existing programming languages can simply be recompiled to target new accelerators.

Largely — but not exclusively — accelerators supporting C-language reprogrammability support the widest range of programs.

Compiling for hardware accelerators requires the development of several key techniques. First, there are *identification* techniques, that identify whether regions of code should be offloaded, and second, there are *compilation* techniques, that take these regions and offload them appropriately.

Any truly automated compiler will do both of these steps. Either is optional: pragmas can be used to overcome lack of identification techniques, and APIs can be used to mask a lack of compiler techniques. The techniques developed in this thesis automate both of these tasks, but this chapter covers existing researching into identification and compilation techniques for hardware accelerators.

3.4 Compiling to Hardware Accelerators

As discussed above, there are three key models for compiling to hardware accelerators:

- Compiling high-level software
- Compiling a DSL
- Compiling with APIs

Each of these techniques applies well to different kinds of accelerator, so all are relevant, and all require the development of different techniques for automation.

A fundamental challenge of hardware accelerators is that many they are *big-step*:

they perform a big step of computation in one action, making them difficult to compile to from traditional programming languages, which are designed for CPUs, *small-step* architectures.

3.4.1 Compiling to API Programmable Accelerators

A large number of accelerators are API programmable. These accelerators tend to compute full algorithms on each call, which provides a great deal of architectural flexibility. However, these accelerators can see limited use if the algorithms provided are not the ones the programmer is interested in using. Usability of APIs has been an important research topic. Work on effective API design and debug methodologies has been extensive [177, 569], but does not address the question of automation that compilers need to tackle.

Automatically programming API-programmable accelerators has been an active research topic. Huang et al. [227] explore replacing APIs with formal specifications, which allows for EGraphs-based rewriting for these targets. Exocompilation [232] explores externalizing optimization techniques from an API using a Halide-like scheduling language.

Constraint-based approaches [81, 146, 191] take a formal description of the API, in the form of constraints on the code. However, writing these constraints is challenging [190] and constraints are brittle, rarely scaling beyond the implementation they were designed for [146]. For accelerators that accelerate polyhedral problems, polyhedral matching approaches can be used to address some of the challenges of the matching problem [76, 469].

Compiling for API programmable accelerators has been explored using equality saturation [228] and program synthesis [530]. Equality saturation has also been used to optimize tensor programs for tensor accelerators [447].

3.4.2 Compiling to DSL-Based Accelerators

DSLs can be split into two categories: architecture-focused DSLs and problem-focused DSLs. Architecture-focused DSLs are designed to overcome the challenges of compiling high-level code to particular hardware accelerators, while problem-focused DSLs simplify the programmer's job. We will focus almost entirely on architecture-focused DSLs, but there are countless problem-focused DSLs (e.g. HTML, XML) that are in popular use.

Architecture-focused DSLs are largely designed around extracting extra parallelism [34, 68, 124, 166, 169, 273] or removing programming language features that make compilation to particular targets challenging [326, 346, 401, 424]. DSLs have been designed for a vast range of targets, including DSPs [401], linear algebra [68, 166], simulations [74, 124], FPGAs [2, 5, 273, 303, 326, 424, 428] cluster programming [34, 169] and embedded programming [346, 454].

The popularity and importance of these features in DSLs has exploded beyond just DSLs, with compiler frameworks like MLIR [293] and TVM [458] adopting similar ideas and programming languages like Julia adopting similar programming styles [173].

3.4.3 Compiling to DSLs

The challenges of writing DSLs beg the question: can we automatically generate DSLs? Compiler writers have been trending towards answering this question, with compiler-frameworks like MLIR [293] bringing architecture-specific DSLs into vogue and prompting automatic translation techniques from high-level languages to a number of these DSLs [110, 172, 349]. Lifting into traditional DSLs has also been a fruitful research topic. Program synthesis is a common tactic to achieve translations into DSLs: Halide has been targeted from C [36] and x86 [341], and Smith and Albargouthi [446] explore compilation to MapReduce accelerators. Schuts et al. [425] explore translation between DSLs for program specification. Large-language models have also been leveraged for this task, with BabelTower [514] enabling C to CUDA compilation.

3.4.4 Compiling to CGRAs

Research on compiling to CGRAs has been particularly extensive. Numerous authors address compiling branches [64, 193, 257, 365, 528, 558], nested loops [256, 410, 528], scheduling of large loops [165, 211, 275, 278, 376] and irregular memory accesses [193]. Compilation time is relevant in many fields [297, 525] and has been addressed both with faster algorithms [297] and hardware acceleration [93, 495].

CGRA scheduling can be done with a number of algorithms: binary decision diagrams [72], the polyhedral model [310, 325], SAT solvers [345, 372] and ILP models [109, 354, 502, 555]. Various heuristic approaches have also been explored leveraging information from failed placements [65, 564], rewrite rules to simply routing [268], sharing information between placement and routing phases [145] and integration of hardware features within the compiler model [277, 547]. Machine-learning has been

used to automate a number of these approaches [106, 252, 278, 311, 568]. Idiomatic compilation [190] has been used to target closely-related spatial accelerators [517]. This incredibly broad range of algorithms all focus on the task of placing operations so that loops execute quickly, but none address the challenges introduced by domain-specific CGRAs.

DSL Programming A number of CGRAs are tightly coupled to DSLs that can program them. Plasticine [397]/Spatial [273] are a key example, but other DSL compilers have been explored [62, 260, 316, 562]. However, achieving high-performance DSL support is non-trivial [178] and DSLs do not solve the problem of supporting otherwise-unsupported software. API interfaces have been used for CGRAs [319, 405], but these make the compilation task harder.

3.4.5 Compiling to Regular Expression Accelerators

A number of previous works focus specifically on compilation of automata. Bo [83] and Becher *et al.* [71] are discussed above, but they are not the only examples. Industrially, Grovf have explored compilation of unsupported regex features to supported regex features on their automaton accelerator, GRegex [263]. Likewise, the Automata Processor explores post-processing in software to support PCRE features not compatible with NFAs [157]. Liu and Torng explore the benefits of partial conversion from NFA to DFA for hardware accelerators [308]. APmap [557] is a compiler targeting Micron's AP among other traditional automata accelerators, aiming at achieving efficient resource utilization within an accelerator. A number of architectures aim to reduce reconfiguration time, such as Pantarelli et al. [380], who use a similar grouping concept similar to what we will explore in Chapter 4 to enable hardware sharing.

Several compilation-centric approaches have focused on reducing compile time for FPGA accelerators. Generalized overlays such as NAPOLY [254] and HARE [194] and others [130, 501] achieve this aim, but at the cost of large reductions in throughput, increases in area, or both. Bo [82] addresses compilation time using an improved compilation workflow, but this does not bring compilation time down to acceptable levels for real-time applications. Similarly, Park *et al.* [381] introduce a partial reconfiguration approach, but their approach is aimed at debugging applications so has a far higher overhead (30%) and does not reduce compile time to the extend RXPSC does, as expensive place and route steps must still be executed. More generally, full reconfiguration approaches are often vendor-specific, require extremely high power

utilization and largely targets reduction in reconfiguration time rather than reduction in compilation time [497].

Previous Work on Regular Expression Compression As part of the development of the Cache Automaton, Subrymanian *et al.* [457] explore prefix merging to reduce the required space. ENREM extends this by exploring both prefix and infix sharing [220]. Impala [413] explore compression of lookup tables to enable multi-stride automata without lookup-table blowup. As work that treats the underlying automata structure as a black box, our approach is compatible with these approaches.

Numerous techniques exist for compressing DFAs, with D^2FA [286] a particularly prominent one. Kong *et al.* [276] and Becchi and Crowley [70] explore the use of Alphabet Compression Tables to translate the input stream and reduce the number of characters that need to be considered. More recently, Nourian *et al.* [370] apply the same concept to FPGAs.

RXPSC can also be used to compress NFAs, where it achieves an approximately 50% reduction in regexes required across ANMLZoo. Our work focuses on NFAs, for which reduction is known to be a PSPACE-complete problem [200]. Algorithms exist that approximate this compression of individual NFAs [233], and we note that RXPSC is compatible with these approaches.

A number of works focus on generating regular expressions, from IO examples [69], natural language text [291] or both [304].

Approximate Regular Expressions There is also a body of work on approximate regular expression computation. Becher *et al.* [71] is one such example. Ceska *et al.* [105] explore approximate nondeterministic automata in the context of network intrusion detection — they do not propagate any errors to the CPU, instead using simpler matches in series with more complex matches to enable greater parallelism for less area. Sabet *et al.* [412] explore a theoretical framework for examining how errors in DFA execution propagate into DFA outputs. Grachev *et al.* [199] run experiments exploring the similarity between input/output-based measurements of similarity and structural measures of similarity between automata.

3.4.6 Compiling to FFT Accelerators

DSP optimizations [180, 262] can aid FFT performance, but do not come close to accelerator performance [23]. DSL approaches get closer [400, 401, 454, 487], and work to extract such DSLs from source code has been developed [36, 52, 249, 341], but

these approaches rely on programmable small-step accelerators and do not generalize to big-step accelerators such as the FFT accelerators we explore.

Constraint matching [81, 146, 189] provides a way of matching and extracting interfaces from high-level code. Unfortunately, these approaches are brittle [146] — they do not scale beyond a single implementation/accelerator pair, and constraint patterns are extremely hard to write [128,129,190]. Rewrite-rule based compilers can be used to target accelerators [284], but these still rely on initial matching using constraints or similar. For affine algorithms, approaches using polyhedral analysis have also been attempted [76, 315, 458, 469] — but these are inapplicable to non-affine or highly-optimized implementations. Other authors focus on ensuring that the presented API retains easy programmability [320], aiming to help programmers program accelerators directly.

A large amount of work has been done on API migration [127, 366, 367, 369, 392],the task of migrating code using one API to use a new API. Likewise, a number of API-recommendation tools [229, 545] have been developed, although these do not tell the programmer how to integrate the API. Another common approach is a backend-independent API [347, 471] allowing for migrations to happen under the hood. Samak et al. [417,418] approach a similar problem in the object-oriented space, using embeddings and synthesis to generate adapter classes for drop-in replacement classes in Java. The tools, MASK and CLASSFINDER, use symbolic execution to prove equivalence, a technique which does not scale to FFT-sized algorithms. Work applying program synthesis to take advantage of its syntax-independence has been applied in software optimization [135, 393, 422].

3.4.7 Equality Saturation

Equality saturation [472,524] has been used for a range of tasks, including: optimization and translation validation of Java bytecode and LLVM programs [453], improving accuracy of floating point expressions [379], synthesizing structured CAD models [359], optimizing linear algebra expressions [509], tensor graph superoptimization [552], vectorization for digital signal processors [488], optimizing integer multiplication on FPGAs [484], hardware datapath optimization [136].

A proposed DSLs to Accelerators (D2A) methodology [228, 447] uses equality saturation for optimization and hardware mapping of DSLs.

3.5 Identifying Code to Accelerate

Code identification techniques form the second part of existing accelerator compilation techniques. Identification strategies are critical for all accelerators: either to identify which code *can* fit, or which code *should* fit — as the lifting and extraction techniques developed in this the as lifting and extraction techniques developed in this thesis are often slow.

Code classification is a broad subject, often requiring holistic understanding of wide and varied contexts [494]. There have been a large number of attempts to classify code [79]. Classification has been done on many scales; from application-wide classifications [225, 245, 460] to basic-block-level classification [513] and for a variety of reasons ranging from classification for the sake of classification [362] to classification for bug detection [460]. We set out to answer the question, what existing program classification tools apply well to classification of code for accelerators?

3.5.1 Size Matters: How big are the regions we should be classifying?

The size of region of code that can be accelerated depends on the accelerator in question: some accelerators (e.g. DSPs) can be used to accelerate whole programs, while some accelerators (e.g. the x86 CRC32 acceleration instructions) only implement a single step in a function. Appropriate classification techniques also differ across scales.

Program-wide Classification Program-wide classification focuses on classifying whole-programs by microachitectural characteristics [245], or more frequently by making performance predictions [225, 411, 460, 554]. These classifications can be used to guide accelerator selection [510].

Function Classifiers Function classifiers are designed for numerous tasks, from document generation [33,149] to algorithm labelling [46,47], function naming [43], code clone detection [350,512,561,571] and as refactoring tools [238,569]. Embeddings [46], and more recently, transformers [571] have been popular ways to classify functions.

Snippet-scale classification Snippet-scale classification has been used for code-clone detection [242, 258, 456, 506]. More generically, idiom-detection techniques aim to extract common programming patterns [45, 445]. Because snippets are not one fixed size, techniques developed for snippets are often composable [241, 258]

Small-scale classification Most existing hardware accelerator works operate in this

space: the space of a few instructions that can be easily matched. Graph-matching techniques [357] for code extraction and variable name classification [44] techniques apply here.

3.5.2 Broadly applicable classification techniques

To enable classification at all these scales, a number of techniques have been developed. Clustering algorithms [361, 362], embeddings [47, 149], GNNs [305], LSTMs [466] have been proposed, and most have, or could be, applied to regions of any size.

3.5.3 Semantic Classifiers

We are after a classifier that produces some semantic interpretation of a program. A number of the embeddings discussed above do this by using program features in their feature vectors [46,47,149,156,366]. In addition to code features, identifier names [373], formal models [243, 266] and commit log messages [223] can be used to classify code by behaviour.

Vector Classifiers Before the machine-learning revolution, a number of techniques based on statistics were developed for programs [241] and executables [464]. Classifiers using "bag-of-operations", where features are reordered but structure is omitted address a similar problem [258, 415]. These vectors have simple mathematical properties, which makes them useful for comparisons across gigantic code-bases [416]. Program embeddings such as ProGraML [138] also achieve high success rates on code clone detection.

Functional Equivalence-Based Classifiers Functional equivalence is the idea that functions are equivalent if they behave the same. EqMiner [242] explores functional equivalence between different code snippets. Heuristics such as function names [461] can be used to increase the scalability of this approach, which has been applied to code-clone detection [176, 333].

3.5.4 Limitations of Classification Techniques

Existing classification techniques can be applied directly to accelerator programming [483], but do have several key limitations. Primarily, they do not specify *how* accelerator programming should be done, which is the hole that this thesis addresses. Each individual aspect of classification has limits: neither lexical similarity [115] nor

structural similarity [221, 500] are sufficient on their own, and functional similarity has challenges scaling, both in number of problems [261] and due to I/O differences [150]. Hybrid approaches [118] have been shown to be effective at overcoming some of these limitations — but there are still limitations to what similarity of code even means [251], and serious limitations to existing datasets [281].

3.5.5 Existing Datasets

All the learned measures of similarity require datasets. There are relatively few labelled datasets available in practice, with BigCloneBench [462] and OJClone [350] the two frequently used datasets — both from coding websites/courses. For unsupervised models (e.g. [238, 259]), large datasets such as Java's 50K-C [332], Anghabench (C) [139] or Exebench (C, with I/O exmaples) [57] can be used.

Multilingual datasets like CodeScope [546] also exist.

3.5.6 Applications for Hardware Accelerators

Vector-embedding techniques such as code2vec [47] can be used to identify and label algorithms in code. There are numerous techniques that use larger, code-clone specific datasets to achieve quantifiable results. Embeddings such as ProGraML [138] achieve upwards of 95% accuracy in clone detection, and a number of other machine-learning approaches using static information exist [89,174,186,350,373,512,518,561]. Dynamic runtime information can also be used for this task [504] and numerous approaches developed without machine learning exist [241, 243, 266, 416]. API-recommendation tools [216, 229] can also be used for algorithm identification. Finally, NLP has been applied to code comments to identify the algorithm [270]. Algorithmic mismatch has been explored on a number of dimensions in relation to AI accelerators [491], and with relation to different FFT API calls [471]. Identification tools such as those developed by Uhrie [483] are capable of achieving high-accuracy on the kernel matching tasks required for binding.

3.6 Summary

This section has discussed the research surrounding code classification and identification technologies, which form a critical part in enabling compilation to hardware accelerators. We will discuss more direct ways to apply these techniques in the next section.

Chapter 4

Regular Expression Accelerators

Regular expressions (regexes) play a key role in a wide range of systems including network intrusion detection. FPGA accelerators can provide power savings over CPUs by exploiting MISD (Multiple Instruction, Single Data) parallelism inherent in regex processing. However, FPGA solutions are brittle, requiring hours to reprogram when rulesets change, while real-world security threats evolve rapidly.

We present RXPSC (Regular eXPression Structural Compiler), a compiler designed to compile new regexes to existing regex accelerators. We use input-stream translation to enable fixed FPGA accelerators to accelerate new patterns with minimal overhead and little update delay. Compared to a solution where new regexes run on a CPU, RXPSC reduces CPU load by more than a factor of ten for 84% of unseen regexes in ANMLZoo benchmarks.

4.1 Introduction

Regex processing is critical in a wide range of fields, from genome processing to network intrusion detection [499]. This range of applications, and the applicability of the regular expression computational model to a MISD (multiple instruction, single data) processor model, has driven the development of a number of hardware accelerators. MISD tasks are well-suited to acceleration, as they fit traditional models of parallelism poorly (as processors must be capable of executing multiple instructions simultaneously) and have relatively low data-transfer overheads. IP companies such as Grovf [202] currently offer automata accelerators, and Micron offers a physical automata processing board [157]. FPGAs are also able to take advantage of the parallelism available in regex processors [403], offering an alternative to high ASIC design costs, latencies and

low design utilization [371]. In particular, the network intrusion detection setting has seen significant support for reconfigurable regex acceleration, from P4 programmable architectures [240] to FPGA-based accelerators [105]. However, FPGA-based automata accelerators face problems with the dynamic nature of network intrusion detection rulesets. As Xu *et al.* [544] state, "FPGAs do not support fast dynamic updates, so are not applicable in network security applications where signature rules are altered frequently".

Existing work on regex update times focuses either on better utilizing FPGA toolchains [82] which still leaves pattern update times orders of magnitude too long, or on introducing architectures that make reconfigurability easier [83, 254]. Of these architectures, generalized FPGA overlays such as NAPOLY [254] reduce compile time drastically, but have far less throughput and capacity than single-level reconfigurable designs [403]. There have been attempts at fast reprogrammability [83] but they lack compiler support, and only support regex *replacement* rather than regex *addition*.

We present a methodology for supporting acceleration of new regexes on existing accelerators using the acceleration equation introduced in chapter 2. Given some regular expression A, and some new regular expression U, we generate g a *stateless translator* (Section 4.3) and h, a post-acceleration check (Section 4.4.2.2) that when combined result in $U = g \circ A \circ h$. We provide a compiler, RXPSC which supports dynamically adding regexes. Our methodology is *independent* of the implementation of the underlying accelerator, and allows the new and old accelerator to coexist with negligible latency and no throughput penalty. We compare to an existing automatic compilation technique that can be applied to this problem, prefix merging, as discussed by Wadden *et al.* [499] and show that prefix merging is limited by its requirements for complete equality between prefixes, an assumption that often does not hold.

In our methodology, regexes are efficiently *translated* to each other using stateless translators, a hardware feature that converts the input-stream character-by-character to a new input-stream and pass the new input stream into an existing accelerator. Compared to a hybrid solution where new regexes are run on the CPU, this translation can reduce the number of bytes that must be checked on the CPU by more than a factor of ten in 84% of cases across ANMLZoo [499]. Stateless translation allows regexes to share underlying accelerators *regardless of their implementation* and provides easy scalability for higher performance. Our work exploits underutilized FPGA resources in use cases where not all regexes must be run over every input, for example anti-virus programs with different rules for different file types, protein-search operations with different

proteins or network intrusion detection systems where different rules apply to different protocols/ports. We use the accelerators that would not otherwise be in use to accelerate new patterns. Given a set of existing accelerators of the same family, RXPSC finds accelerators for 97% of ANMLZoo patterns among the regex family benchmarks, Brill, ClamAV, Dotstar, PowerEN, Protomata and Snort. We examine the network intrusion detection usecase in more detail, where RXPSC finds accelerators for 96% of patterns in the registered Snort ruleset.

We make the following contributions:

- A model for accelerator re-use that is scalable and independent of the underlying accelerator implementation.
- RXPSC, a compiler for finding structural similarities between regexes and generating stateless translators.

Our model is capable of reducing the CPU load of unseen accelerators by more than a factor of ten in 84% of cases across the ANMLZoo benchmark suite.

4.1.1 Connection to the Accelerator Equation

This chapter solves the accelerator equation for regex accelerators. These accelerators run some function A that computes a regex, and we have some user code U that computes a different regex. As this chapter will cover, we solve the accelerator equation, generating a stateless translator that is executed for function g and a check function that is executed for function h.

4.2 Background

Regular expressions (regexes) are matching rules used in numerous domains, from network intrusion detection (indicating malicious packets based on text matches) to bioinformatics (indicating genome sequences).

Regexes include character ranges (`[a-z]`), optional substrings (`a?`) and repeated substrings (`(ab)*`). As an example, the regex `ab*` matches the strings `a, ab, abb, ...`. Regex processing engines provide various syntax to simplify the writing process. In this chapter, we will use the + notation, which matches one or more of the preceeding regex, and the `[abc]` notation, which matches any of the characters in the brackets. The notation `[^abc]` is used to match any character *except* `a`, `b` or `c`. Finally, regex engines often provide character classes, such as `\s` which matches all white space characters.

PASS\s*\n	PASS\s[^\n]*%[^\n]*%	PASS\s[^\n]50

Table 4.1: Three prefix mergable patterns from the Snort ruleset. Syntax is explained in Section 4.2.

Some examples of regexes from the Snort network intrusion detection ruleset are shown in Table 4.1.

Regexes are usually implemented using either Deterministic Finite Automata (DFAs) or Non-deterministic Finite Automata (NFAs). Grapefruit [403] implements regexes using NFAs, which take advantage of the parallelism available on an FPGA.

Regexes are useful in numerous contexts, as they provide a high-level portable description that can be used across different execution contexts [54]. Although regexes are frequently used as one-off string matching, for example for input validation [431], there are numerous applications where large sets of regular expressions must be executed over the same inputs, for example Snort [478], a network intrusion detection ruleset, where rules are written in a DSL specifying a number of parameters often including a regex. For other applications such as NLP tag matching, the rules are written to find common sequences of word classes. These approaches result in large rulesets, all of which must be accelerated at the same time, presenting a large MISD (multiple instruction single data) problem that FPGAs accelerate effectively. In this programming model, it is difficult to add new expressions to an FPGA accelerator as FPGA compile times are excessively long. RXPSC is evaluated on this style of regular expression application, but there is no barrier to using RXPSC to accelerate the bespoke pattern matching regexes that regularly appear in high-level code, which would enable these patterns to use fixed regex hardware accelerators.

4.2.1 Regular Expression Accelerator Architectures

FPGA accelerators are designed to take advantage of the MISD model that regular expression modelling presents. This makes them excellent offload opportunities as the size of the input data does not increase with the number of regular expressions: there is one stream of input data (single data) and each regular expression is run over that stream. Architectures typically convert regular expressions to non-deterministic finite automata (NFAs) for execution. In a non-deterministic finite automaton, multiple states may be active at any one point in time. This fits the capabilities of FPGAs very well as many active states may exist on-chip at the same time, taking advantage of the inherent

parallelism. In CPU models, keeping track of multiple active states at the same time is much more difficult. By comparison to CPUs and GPUs, FPGAs provide a much better performance-per-watt [370].

4.2.2 Why Use Accelerators?

Since CPUs are general purpose, they have lower power-efficiency per-watt than FPGAs when running regexes. Further, the generality of CPUs means that they can be used for any task, and as discussed the MISD parallelism of regexes makes them a good offloading target. Thus, a key goal of accelerators for regexes is to reduce the workload of the CPU, and to free it to execute other, less-acceleratable tasks.

4.2.3 Summary of Existing Architectures

Numerous FPGA acceleration strategies exist for regexes, all with different benefits. Grapefruit [403] converts regexes to NFAs, then distributes NFA states through the FPGA fabric. States are either activated by a centralized memory for each accelerator that looks up which states can become active on each input character (BRAM-mode), or by a distributed symbol check (LUT-mode). Applications with large numbers of automata perform better in a BRAM mode, while applications with smaller numbers of automata perform better in a LUT mode due to the challenges of wiring optimization [83]. Impala [413] changes the structure of the underlying automata to read multiple characters per cycle without incurring the cost of exponential lookup table blowup that would normally come from reading multiple symbols per cycle by reducing the size of each input symbol to be considered. Ceska *et al.* [105] introduce an approach for very high throughput automata by using partial matching. These partial accelerators are small, so can be parallelised efficiently, but still eliminate almost all text sequences. This enables throughputs that can keep up with the demands of modern network intrusion detection workloads. RXPSC is designed to work with all of these approaches, as it does not rely on the implementation of the accelerator. This is a further advantage over related work on the topic such as symbol-only-reconfiguration [82] that relies on a particular accelerator design.

4.2.4 Regular Expression Use Cases

Regular expressions (regexes) are used in many fields from network intrusion detection, where rules are used to detect suspicious patterns in packets to bioinformatics, where rules can be used to detect common genome sequences. ANMLZoo [499] is a popular benchmark suite containing patterns for these applications. There are two categories of application in ANMLZoo: those that are regular expression-based, and those that are finite-automaton based. These two are representations of the same concept, but some problems are easier to express in one framework than the other. For example, finite automata can be used to calculate the number of mismatches between strings: it is easy to design automata for this task, but designing regular expressions for this is much more challenging.

This work focuses on regex-based tasks in ANMLZoo. These are summarized below (reproduced from [499]):

Snort are regular expressions from the Snort network intrusion detection ruleset, and often used to benchmark regular expression processing engines.

Dotstar is a synthetically-generated set of regular expressions used by Becchi et al. [70] for their evaluation.

ClamAV is a set of regexes for identifying viruses in files.

PowerEN is a set of synthetic regexes from IBM [59].

Brill is a set of rules for part-of-speech tagging in NLP.

Protomata is a set of mixed real and synthetic protein motif signatures.

The benchmarks we use are sized to fit on FPGA accelerators, so are roughly 2,000 regexes large each. In real usecases, the number of expressions can be much larger, with Snort's full ruleset exceeding 30,000 rules. Similarly, applications such as representing bioinformatics motifs can easily become enormous. In general, the techniques in this section scale well to larger benchmark suites, since they rely on similarities that are easier to find with larger benchmark suites. However, it is worth bearing in mind that more development of FPGA technology, and work to select smaller parts of these applications to be accelerated for them to be applicable as real-world applications.

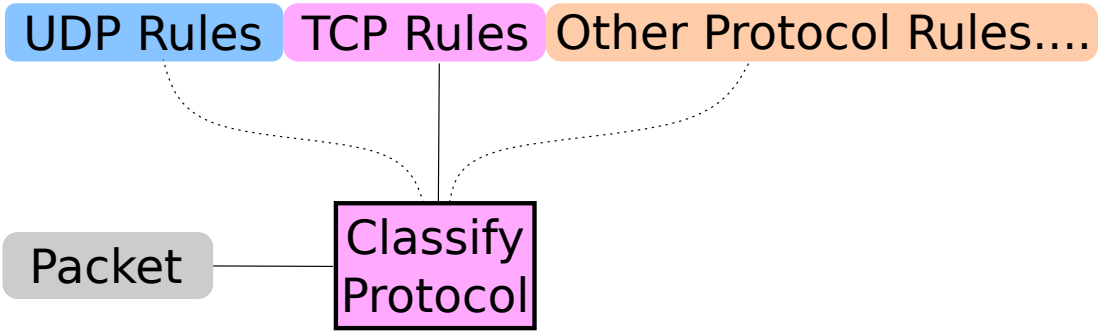


Figure 4.1: Not all rules have to be run for every byte of input data in all streams.

4.2.4.1 Groups of Regular Expressions

A recurring theme that occurs in this chapter is the idea of a "group" of regular expressions. This is a common theme across many of these tasks: not all data must be run through every regex. For example, in Snort, rules are associated with particular protocols, so we can devise acceleration techniques that share hardware as long as the time required to switch between rulesets is small. An example of this is shown in Figure 4.1.

4.3 Input Stream Translation

The model we present for translating input streams is *stateless-translation*. We use a character lookup that applied to every input character. We present this model because it is easy to implement as a lookup on FPGA, *easy to parallelize* for higher throughput as there are no stateful dependencies, *easy to enable* or disable in a fine-grained manner and *independent of the underlying accelerator* implementation.

A diagram of how this integrates into a Grapefruit [403] accelerator is shown in Figure 4.2. Input data is streamed into the FPGA, which distributes it between multiple regex accelerators. These report to the CPU when they match. Our translator sits between the regex and the input, and can optionally be enabled for certain types of input. It behaves as a lookup table, translating characters byte-by-byte. The CPU can be used to inject new patterns on to the accelerator in milliseconds by updating the stateless translators without a full recompilation which takes hours.

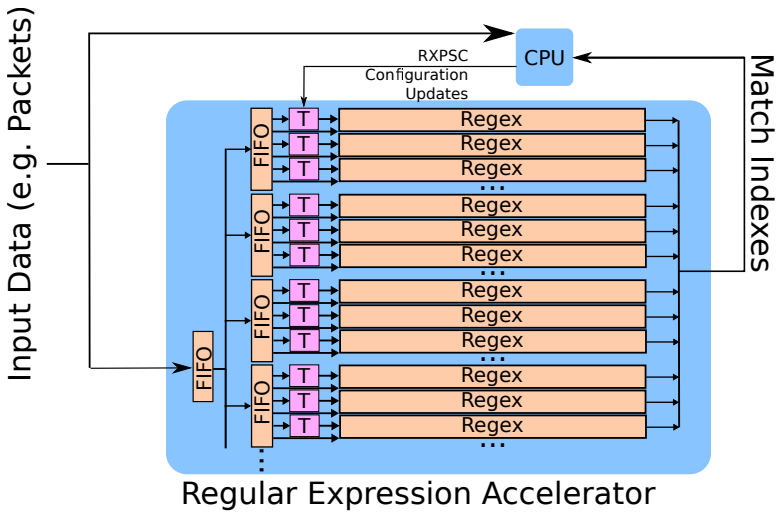


Figure 4.2: Data is read into the accelerator, which sends match indexes to the CPU. Translators (T) are used to change the behaviour of accelerators to enable acceleration of new regexes. After matching, the indexes are sent to the CPU to be checked. These functions correspond to g and h in the accelerator equation with $g =$ T and $h =$ Checking on CPU.

4.3.1 Motivating Example

RXPSC is designed to allow for fast addition of new regular expressions using old accelerators. In this section, we briefly describe how prefix merging can be used to solve the same problem, and demonstrate where it falls apart.

4.3.1.1 Prefix Merging

Prefix merging is a well-known automata compression technique [499]. As an example, suppose we have accelerators for the regexes `abx` and `aby`. These share a common prefix of `ab`, which can be extracted at compile time and accelerated. If we add a regex, `abz`, this common prefix can be used to reduce the computational load of accelerating this third regex without recompiling the FPGA accelerator. An example architecture supporting this is shown in Figure 4.3.

Prefix merging scales well as an acceleration technique as the number of rules increases, as each additional rule increases the number of common prefixes. However, it scales very poorly with rule length, as prefixes become much less likely as the length of rules increases. In fact, as we will see below, many rules do not share common prefixes despite sharing significant structure.

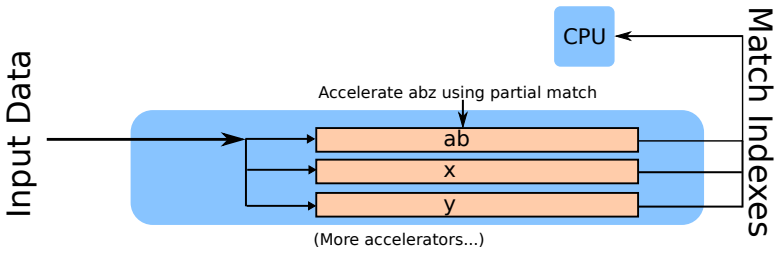


Figure 4.3: Example showing how we can partially accelerate the expression `abz` using prefix merging. Although matching just `ab` does not guarantee that `abz` was found, it significantly reduces the workload to check for the regex `abz`.

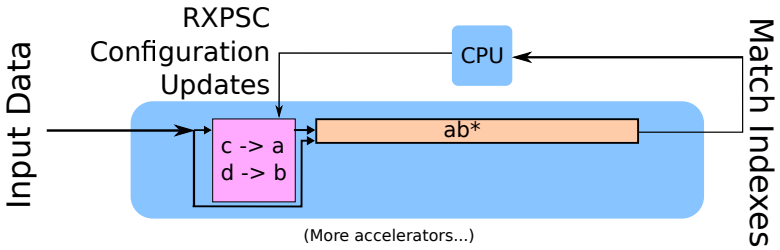


Figure 4.4: Example showing how RXPSC can be used to accelerate the expression `cd*` given an accelerator for `ab*`.

4.3.1.2 Limits of Prefix Merging

However, prefix merging fails in many cases. Suppose we have an accelerator for the regex `ab*`, and we wish to accelerate the regex `cd*`. Despite significant similarity between the regexes, prefix merging fails to provide even a partial accelerator, as the two regexes do not share a prefix. As a result, we would have to run the regex `cd*` on the CPU despite is sharing a great deal of structural similarity with our existing accelerator.

Stateless translation can generate the character lookup table shown in Figure 4.5 and the use of this table is shown in Figure 4.6. If we translate the input stream through the stateless translation table, a match for the regex `ab*` means that the (pretranslation) input stream contained the string `cd*`. We can see what this accelerator looks like in figure 4.4. We will explore how our algorithm generates this translation as a running

Input	Output	Input	Output	Input	Output
c	a	d	b	*	x

Figure 4.5: A stateless translator converting `cd*`to `ab*`. `x` is an arbitrarily selected character to avoid false-positives. An example of this table in use is shown in figure 4.6

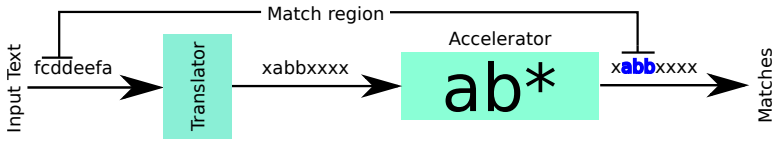


Figure 4.6: How a stateless translator (Table 4.5) is used to accelerate the expression `cd*` using an accelerator for `ab*`.

Input	Output	Input	Output	Input	Output
F	U	:	‘ ’	\n	\r
r	S	\r	‘ ’	‘ ’	%
o	E	;	’	<	\r
m	R	,	’	"	\r

Table 4.2: A stateless translation table to convert from `From: +[^\r\n"<]*[;',]` to `USER *[^\r]+%'`.

example in Section 4.4.

In general, regexes are more complex, and deriving translators is challenging as stateless translators must satisfy larger sets of constraints. To be compatible, regular expressions must have some degree of *structural similarity*, meaning that they share similar constructs and *symbol similarity*, meaning that the symbols are translatable. These are discussed in more detail in Section 4.5.

4.3.1.3 Examples from Snort

The Snort network intrusion detection ruleset [478] is a set of network rules that are used to identify security threats. A large number of these rules have regexes that must be applied to incoming packets — a task that makes sense to accelerate since it is easy to offload to a NIC (network interconnect card) without additional data transfers. This is a convenient place to run network intrusion detection, as all network traffic passes through the NIC before reaching the computer. FPGA accelerators fit this domain well, as many existing NICs now feature FPGAs [104]. Further, Snort updates are issued frequently — every few days — so requiring FPGA recompilation is time consuming and slow.

A real world example from the Snort ruleset supported by prefix merging is shown in Table 4.1. Here, we can see that `PASS` is a prefix of all the rules. With any two of these rules already accelerated and the prefix extracted, we can reduce the CPU load required by using the prefix. However, using stateless translation, we can support more

rules and further reduce the CPU load of supporting prefix-merged rules. Table 4.2 shows one such real-world example of a translator for the Snort ruleset, in a case where prefix matching would fail.

4.3.1.4 Why not Stateful Translators?

In this section we have covered stateless translators, the technique RXPSC uses to map one regex to another accelerator. Generally, stateless translators (i.e. those that require only a single state to do the translation) require fewer resources that stateful translators. However, more generally, stateful translators are rarely useful in practice.

The intuition here is that each state of a translator maps certain states of the regex to states in the accelerator. Having multiple states can reduce the number of symbol collisions (which we will cover in more detail in Section 4.5.3), but only when the translator can always be in a certain state for a certain state in the accelerator. In generally, this is not possible, as common features like symbol looks (e.g. with `a*`) make it impossible for the translator to stay in sync with the accelerator.

As a result, using a stateless translator is the best choice, as it has low resource usage, but covers the same cases in practice.

4.3.2 Groups

RXPSC exploits groupings of regexes that do not have to evaluated at the same time. A group contains a set of regexes that are all evaluated at the same time: it means that the hardware used to accelerate other groups is unutilized during the operation of any particular group. An example is shown in Figure 4.1. Groups are application-dependent; for example, network intrusion detection provides groups in the form of protocol, port numbers and IP address ranges. These can be distinguished rapidly in hardware [60, 380], to activate a group of accelerators. RXPSC takes advantage of the otherwise unutilized accelerators. This enables RXPSC to *add new* regexes to the existing accelerators, rather than just replacing old regexes. Figure 4.7 shows an example of a TCP rule being multiplexed onto hardware designed for UDP rules.

4.4 Compilation Overview

RXPSC takes as input a set of regexes that already have accelerator implementations, and a number of new regexes to be accelerated. It then translates each new regex to an

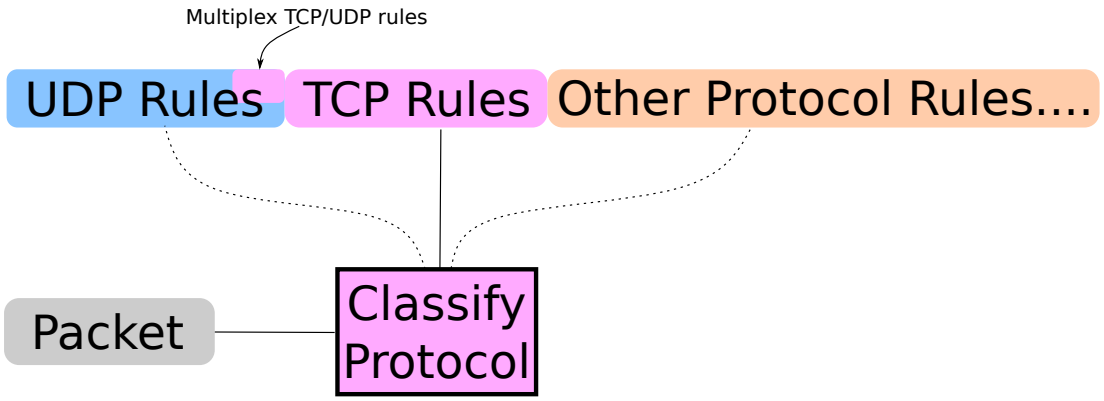


Figure 4.7: Since UDP hardware is unused when a TCP packet arrives, we can use it to accelerate new TCP regexes, provided that we can quickly multiplex between the new TCP pattern and the original UDP pattern.

accelerator in a different group, enabling acceleration of the new regex. The difference between these two phases is shown in Figure 4.8.

4.4.1 Accelerator Assignment

When adding a regex, RXPSC will calculate the similarity (Section 4.5) of that regex to *all* of the existing accelerators. Once RXPSC has all the potential translations, it picks the best, operating in a greedy manner. Existing accelerators may not accelerate the whole regex, so RXPSC identifies and configures accelerators that can accelerate the remainder. The set of accelerators covering the new regular expression, and corresponding stateless translators, is the output of RXPSC. The core technique of determining similarity is described in the next section.

4.4.2 Striking a Balance for Stateless Translation

Stateless translation tries to strike a balance between hardware overheads, which are smaller if expressions are larger and regex compatibility, which is better if expressions are smaller. As discussed in Section 4.6, the architectural overheads are fairly small. In this section, we discuss the techniques we use to ensure better regex compatibility.

4.4.2.1 Prefix Splitting

Smaller expressions are more likely to generate regex matches. Although we cannot control the size of the expressions that need to be accelerated, we can split them into

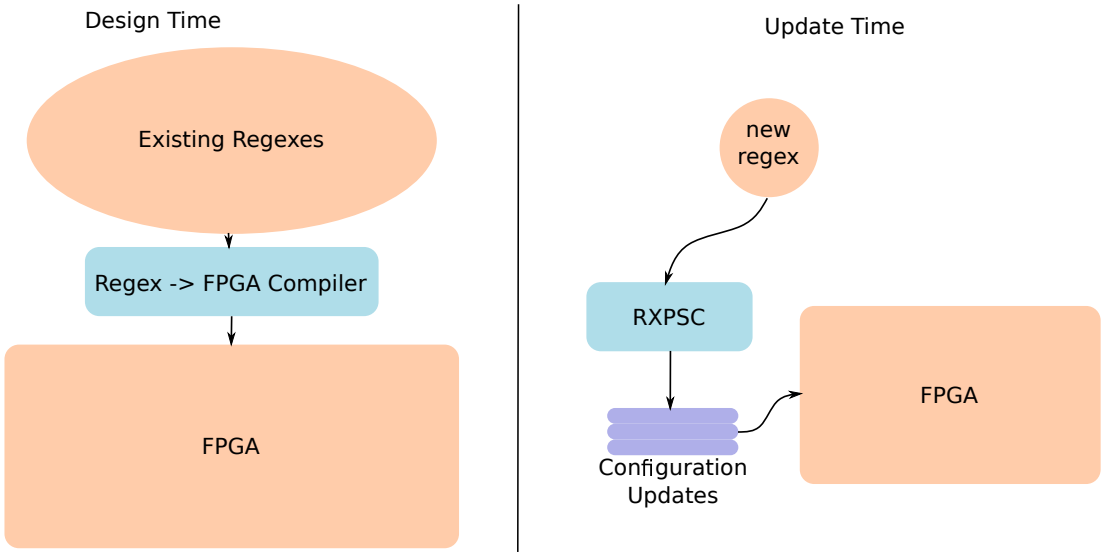


Figure 4.8: RXPSC fits into the update pipeline of FPGA accelerators, and the initial design can be done by any regex to FPGA tool. Grapefruit [403] is a good example of such a tool.

distinct translation regions. We do this using a technique we call prefix splitting.

In prefix splitting, we inspect the set of expressions to be accelerated. At the initial design time, we split common prefixes (longer than five regex terms[1]) into individual accelerators. This results in a large number of small accelerators that represent the common patterns that expressions that must be added at short notice will often share.

When RXPSC adds support for a new expression, it can use these prefixes to find matches. For example, if we have accelerators for the expressions `ab*c*` and `ab*d`, these have a prefix `ab*`. If a new expression `cd*d` is introduced, we can achieve partial acceleration using the prefix `ab*` using the stateless translator discussed before, shown in Table 4.5. Had the prefix not been extracted, we could not simply use the first portion of `ab*c*`, as `b*` does not report as an accepted string.

4.4.2.2 Overapproximation

To find accelerators for a wider range of regexes, RXPSC can find overapproximations, meaning that they always accept strings that the original regex would have accepted, but they also accept additional strings. These additional matches can be filtered on the CPU — but since almost all potential matches are eliminated, the workload is drastically

[1]This number was selected empirically to be a good tradeoff between too many false-positives (which we would get with shorter prefixes) and omitting too many possible prefix matches (which we would get with a longer length requirement).

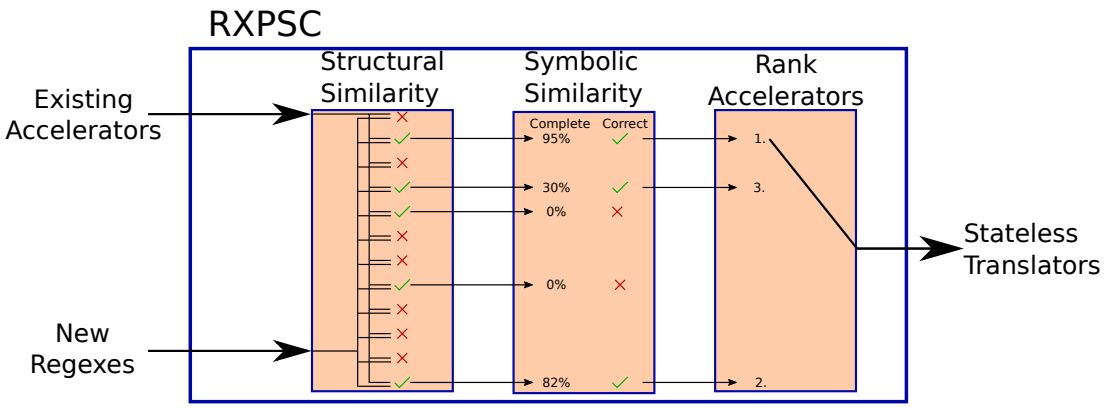


Figure 4.9: How the compilation passes in RXPSC fit together.

reduced. To distinguish between accelerators that overapproximate by various degrees internally, we use an overapproximation factor, which is the fraction of states in the accelerator that are spuriously activated by input symbols. This is a simplification of more complex error representations [412], but allows RXPSC to make informed decisions about which accelerators are likely to be useful. Overapproximation is not a required property — RXPSC can still find compatible regexes without overapproximation. However, as regexes get longer, the probability of a perfect match is reduced. Overapproximation is required to support many real-world usecases. The result of this overapproximation is discussed in terms of the extra bytes it involves sending to the CPU in Section 4.7.

In the above example, where we accelerate the expression `cd*d` using `ab*` by translating `c -> a, d -> b` demonstrates overacceptance. For example, the string `c` would be accepted, even though it does not match the original expression. As a result, matches must be filtered on the CPU.

4.5 Regular Expression Similarity

RXPSC breaks compilation down into structural similarity (Section 4.5.1) and symbol similarity (Section 4.5.3). Structural similarity determines whether regular expressions use similar constructs and symbol similarity determines whether symbols in expressions can be translated to each other. How these operations fit together in RXPSC is shown in Figure 4.9

4.5.1 Structural Similarity

Structural similarity matches a new regex to parts of existing accelerators, while abstracting away any symbol. Symbol similarity (Section 4.5.3) subsequently determines whether a translator is feasible once symbols are taken into account. We approach structural similarity by compiling regexes to an intermediate language, the accepting path algebra, that abstracts structure from each regex. We then use this algebra to determine which accelerators can structurally support a regex.

For example, `ab*`, `cd*` and `aa*` are structurally similar, but `ab*` and `b*` are not structurally similar.

4.5.1.1 Accepting Path Algebra

We propose the accepting path algebra, which can be created from regexes. The accepting path algebra enables comparison between different regexes to determine which regexes are similar to each other. The elements of the algebra are:

$n \in \{0,1\}$ This means n symbols are read from the input.

a This means there is an accept.

e Means that this branch of the regex continues no further.

$x+y$ Where x and y are accepting path algebra terms. This means y follows x. It is not commutative.

We collapse long sums for readability (e.g. writing $1+1+1$ as 3), but our algorithms work on unary digits.

$x*$ Where x is an accepting path algebra term. This means that there are loops, one represented by x.

$\{x_0,\ldots,x_n\}$ This means that there is a branch and each arm of the branch is one element within the set. Each of the x_i are accepting path algebra terms.

(x) Where x is an accepting path algebra term. This is used to indicate order of operations, frequently in conjunction with $*$.

This representation is based on the structure of regexes, but abstracts away from the symbols in the regex. This differs from a closed semiring in that the multiplication operation (here $+$) is not annihilated by the identity of the addition operation (here $\{,\}$, with identity `0`).

Regex	Accepting Path Algebra
`ab*`	$1+(1)*+a+e$
`cd*`	$1+(1)*+a+e$
`a(b|c)de`	$1+\{1,1\}+2+a+e$

Table 4.3: Accepting path algebras for some example regexes. The accepting path algebras are representative structure of the corresponding regexes, abstracting away symbol-specific parts of these.

4.5.1.2 Example

In practice, we generate the accepting path algebra from NFAs. We will use regexes in these examples for simplicity. The accepting path algebras for various regexes are shown in Table 4.3.

4.5.1.3 Determining Structural Support

We describe the algorithm used to determine whether an accelerator can structurally support a different regex. We define the notation $x \leq y$ (read as y structurally supports x) by cases on the structure of the accepting path algebra in Algorithm 1. This algorithm shows simplified boolean version of this algorithm — our implementation uses heuristics to reduce the size of the search space. Given two accepting path algebras A and B, we apply this algorithm with a recursive walk through the algebra structures.

4.5.1.4 Walk-Through Algorithm

We will walk-through the example shown in Figure 4.10. In this figure, we apply Algorithm 1 to two identical accepting path algebras, showing that $1+(1)*+a+e \leq 1+(1*)+a+e$. To show this, we take the following steps:

1. Apply the **plus** rule. This application is heuristic guided, and described more in Section 4.5.2. In this case, we find the following groups: $\text{XGroup} = [1,(1)*,a,e]$ and $\text{YGroup} = [1,(1)*,a,e]$. We proceed to show that: $\forall m.\text{XGroup}[m] \leq \text{YGroup}[m]$.
   (a) $1 \leq 1$: by **inteq**
   (b) $(1)* \leq (1)*$: We apply rule **muleq** and need to show that $1 \leq 1$.
      i. $1 \leq 1$: by **inteq**
   (c) $a \leq a$: by **accepteq**

Algorithm 1 Simplified structural support algorithm, defined by structural recursion over terms in the accepting path algebra (Section 4.5.1.1). The implementation of the two most abstract rules (**plus** and **set**) is discussed in Section 4.5.2.

procedure A $\leq$ B
  $e \leq x$: if $x \neq a + ...$ **return** True ▷ **(trim)**
  $a \leq a + x$: **return** True ▷ **(dropadd)**
  $0 \leq y*$: **return** True ▷ **(dropmul)**
  $m \leq n$ $(m, n \in \mathbb{N})$ **return** $m == n$ ▷ **(inteq)**
  $a \leq a$: **return** True ▷ **(accepteq)**
  $e \leq e$: **return** True ▷ **(endeq)**
  $x* \leq y*$: **return** $x \leq y$ ▷ **(muleq)**
  $x_0 + \cdots + x_n \leq y_0 + \cdots + y_m$: **return** True if: ▷ **(plus)**
    $\exists i$. with $x_0 + \cdots + x_i \leq y_0 + \cdots + y_m$
    Or, $\exists .i$ with $x_0 + \cdots + x_n \leq y_0 + \cdots + y_i$
    Or, $\exists$
      XGroup = $[x_0 + \cdots + x_i, x_{i+1} + \cdots + x_j, \ldots]$
      YGroup = $[y_0 + \cdots + y_a, y_{a+1} + \cdots y_b, \ldots]$.
      Such that: $\forall m$. XGroup[m] $\leq$ YGroup[m]
      (Covered in Detail in Section 4.5.2)
  $\{x_0, \ldots, x_n\} \leq \{y_0, \ldots, y_m\}$: **return** True if: ▷ **(set)**
    $\forall x_i . \exists y_j . x_i \leq y_j$
  Otherwise: **return** False
end procedure

(d) $e \leq e$: by **endeq**

This algorithm produces a number of equivalencies that must hold to be able to replace one accelerator with the other. These are discussed in Section 4.5.3.

4.5.1.5 Example

Consider the regexes `ab*` and `cd*`. As we can see in Table 4.3, these both have accepting path algebras $1+(1)*+a+e$. That structural support exists here is clear since the algebras are the same, and is shown in Figure 4.10. A non-trivial example of structural support is shown in Figure 4.11, showing that the regex `h(i|jk|l)m*` structurally supports the regex `a(b|c)`.

4.5.1.5.1 Algorithm Walk-Through The regex `g(i|jk|l)m*` has accepting path algebra $1+\{1,2,1\}+(1)*+a+e$ and the regex `a(b|c)` has accepting path algebra $1+\{1,1\}+a+e$ as shown in Table 4.3. We wish to show that $1+\{1,1\}+a+e \leq 1+\{1,2,1\}+(1)*+a+e$. Applying Algorithm 1, we take the following steps:

1. Apply the **plus** rule. The exploration of which sub-case to use is heuristic guided (see Section 4.5.2), and in this case, we find the following groups: XGroup $=[1,\{1,1\},a,e]$ and YGroup $=[1,\{1,2,1\}+(1)*,a,e]$. To show that this rule holds, we must show that $\forall m.\text{XGroup}[m] \leq \text{YGroup}[m]$ and we proceed with one application of $\leq$ for each case:

   (a) $1 \leq 1$: This holds by rule **inteq**.

   (b) $\{1,1\} \leq \{1,2,1\}+(1)*$: We apply the **plus** rule, rewriting $\{1,1\}$ to the equivalent $\{1,1\}+0$. We find: XGroup $=[\{1,1\},0]$ and YGroup $=[\{1,2,1\},(1)*]$. Again, we must show that $\forall m.\text{XGroup}[m] \leq \text{YGroup}[m]$ and we proceed with one application of $\leq$ for each case:

      i. $\{1,1\} \leq \{1,2,1\}$: We apply rule **set**. For each element x of $\{1,1\}$, we must find an element y of $\{1,2,1\}$ such that $x \leq y$.

         A. $1 \leq 1$: By **inteq**

         B. $1 \leq 1$: By **inteq**

      ii. $0 \leq (1)*$: By **dropmul**

   (c) $a \leq a$: By **accepteq**

   (d) $e \leq e$: By **endeq**

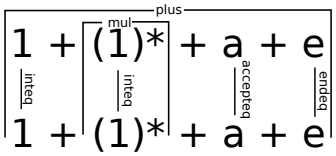


Figure 4.10: Rules from Algorithm 1 applied to show how the accepting path algebra for `ab*` structurally supports the algebra for `cd*`.

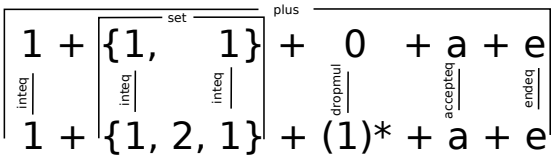


Figure 4.11: Rules from Algorithm 1 applied to show how the accepting path algebra for `h(i|jk|l)m*` structurally supports the algebra for `a(b|c)`.

We can see that the algorithm produces a true answer. Figure 4.11 shows a diagram of the result of this algorithm application. Each case of **inteq** results in a mapping that is used for symbol-unification in Section 4.5.3, while each application of a **dropX**-rule results in a symbol that should be disabled (again this will be covered in Section 4.5.3).

4.5.2 Implementation of Structural Support

Algorithm 1 has an exponential runtime in the number of algebra terms (or the length of the regex). As a result, RXPSC makes a number of heuristic decisions to reduce the runtime. There are two cases where the structural support algorithm can produce excessive runtimes:

- The **(set)** case
- The **(plus)** case

(set) In the set case, we must try every combination of branches — this produces a

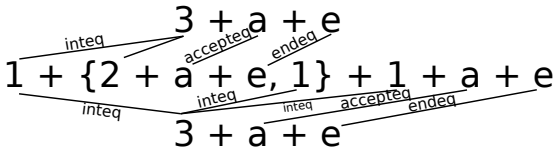


Figure 4.12: Base-case rules from Algorithm 1 applied to show how the accepting path algebra for `a(bc|dd)` structurally supports the accepting path algebra for `xyy` in two different ways. Only one way has symbol similarity. For the sake of clarity, only the base-case rules are shown.

huge number of combinations that must be tried. To avoid dealing with large numbers of unlikely combinations, we use heuristics to select likely branch assignments first. For example, a naïve permutations-based approach results in many failed assignments, even though it is the true implementation of the structural similarity algorithm. In practice, we find it to be exceedingly infrequent that two different branches of a regex can be unified to the same set of states in an accelerator — using combinations rather than permutations solves the bulk of the problem with the number of unifications in the **(set)** case.

The other problem with the **(set)** rule is that we often encounter a large number of false positives — that is structural matches without sufficient symbol similarity. We resolve this problem by partially executing the symbolic similarity algorithm (Section 4.5.3 while determining structural similarity. The reasoning behind this issue and mitigation are discussed more in Section 4.5.2.2.

(plus) Implemented naïvely, the **(plus)** case can result in exponential behaviour, as each "splitting" of the respective pluses results in an exponential number of subterms in the number of terms. In the appendix, we show a lower bound of $2^{\min(m,k)}$ where m,k are the lengths of each term respectively. It is not uncommon to find regexes with sums hundreds of terms long — a perfect implementation is clearly not feasible.

We address this problem with the one-at-a-time heuristic. In the vast majority of cases, we can reduce the **(plus)** by reducing the size of *one of* the algebras by one. For example, $1+1+1$ has structural similarity to $2,1+1$ using the first two terms of the first sum, and the first term of the second sum. Instead of checking all possible splits of the two sums, we can try to match all sums to the first term of each algebra, and pick the split that finds the most structural similarity. This produces a heuristic that reduces the problem from an exponential-sized problem to an $O(n^3)$ problem in the length of the algebras.

$O(n^3)$ is still no efficient execution time, so in addition to the one-at-a-time heuristics, several other heuristics feature to avoid the quadratic behaviour that the one-at-a-time heuristic results in. If both sums start with $1+x$ and $1+y$ respectively, there is structural similarity between the sums iff there is structural similarity between x and y. Similarly, if the accelerator regex sum starts with a product term but the regex to be accelerated starts with a non-product, then we can try disabling the term for the product and continuing finding structural similarity in the tails.

4.5.2.1 Dropping Rules

The **dropadd** and **dropmul** rules operate on the basis that we can represent a smaller accelerator with a larger accelerator. It is correct to drop unneeded algebra components, as the corresponding regexes can be disabled.

For example, suppose we have an accelerator for the regex `ab*c`. If we wish to accelerate the regex `ac`, we could use stateless translation from `b` to `x` (where `x` is a fresh variable, not otherwise used in the accelerator), so that the term `b*` is never activated. Using the **dropmul** rule, we show structural support of `ac`. The **dropadd** rule captures a similar idea.

Structural support is a necessary, but not sufficient, condition for the existence of a stateless translator. Finding an accelerator that structurally supports a new regex shows that acceleration is plausible; but to be useful, we must have sufficient symbol set similarity.

4.5.2.2 Costs of Algebraic Abstraction

The accepting path algebra does not come without costs. Abstracting the symbol-set from the structure means that local structural choices can have negative effects of global symbol similarity and similarly that equally good matches from a structural perspective may not be equally good matches from a symbolic perspective. An example of this is shown in figure 4.12. In this example, although the two shown matches are both *structurally similar*, only the upper match will result in a complete translator — the lower match will require translation of `y` to `b` and also `y` to `c` which is impossible.

To address the potential for uninformed decisions to effect the probability of finding a match, we:

1. Maintain a *set of sets* of assignments from one symbol to another rather than a single set.
2. To avoid an exponential blowup in the number of sets of assignments, we maintain only the top 20[2]. Discarding elements from this set is based on a quick symbolic-collision detection algorithm, a partial evaluation of the algorithm described in Section 4.5.3.

[2]A number determined experimentally to be a good trade-off between speed and performance.

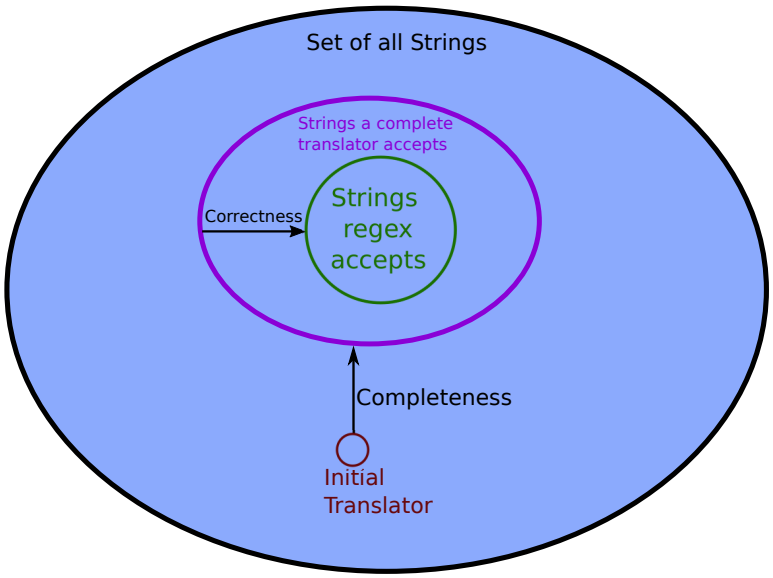


Figure 4.13: Effects of completeness and correctness on the set of strings accepted by a generated translator.

4.5.3 Symbol Similarity

Symbol similarity receives, as input, pairs of symbol sets and produces a stateless translation. One input set is a "virtual" set of symbols that corresponds to the symbols of the expression we *wish* to accelerate, while the other set is a "physical" symbol set which corresponds to the symbols used on the accelerator. Symbol similarity operates in two steps: symbol completeness then symbol correctness. **Symbol completeness** means our translator does not miss any regexes, and **symbol correctness** means our translator does not accept any patterns that should not be accepted. These steps can fail, in which case we either reject the translation, or accept it as an overapproximation. The different aims of the two steps are shown in figure 4.13

4.5.4 Terminology

Symbol Set A range of symbols, e.g. {`a`} or {`[0-9]`}.

Symbol Set Pairs The pairings produced by the structural support algorithm, for example in Figure 4.10.

Virtual Symbols Symbols in the regex to accelerate.

Physical Symbols Symbols in the existing accelerator.

Algebra	Symbol Set (`ab*`)	Algebra	Symbol Set (`cd*`)
1	{a}	1	{c}
(1)*	{b}	(1)*	{d}
a	{}	a	{}
e	{}	e	{}

Table 4.4: The symbol sets related to each algebra term.

Disable Symbols Symbols that activate terms in the accelerator that should not be activated. For example, the term `jk` in Figure 4.11.

Algorithm 2 Algorithm for generating symbol complete translators.

```
1: procedure SymbolComplete(SymbolSetPairs)
2:     ∀x. Translator(x) = U                          ▷ Universal set
3:     for (VirtSymbs, AccSymbs) in SymbolSetPairs do
4:         for Each symbol in VirtSymbs, v do
5:             Translator(v) = Translator(v)∩AccSymbs
6:         end for
7:     end for
8: end procedure
```

4.5.4.1 Symbol Complete Translator

Symbol completeness (Algorithm 2) begins with the pairwise assignment of accepting path algebra terms produced by the structural support algorithm. It produces a translator that accepts all strings the added regex accepts. The output of this step is passed to symbol-correctness to produce a stateless translator.

4.5.4.2 Example

We take a simple example to demonstrate this algorithm. Suppose we are computing a symbol-complete translator for expressions `ab*` and `cd*`. We saw in Section 4.5.1.2 that these have algebras $1+(1)*+a+e$. We know that each term has the sets shown in Table 4.4.

Using Algorithm 2, we generate the complete translator shown in Figure 4.14.

$$
\begin{aligned}
\mathit{Translator}(\texttt{a}) &= \{\texttt{a...z}\} \\
\mathit{Translator}(\texttt{b}) &= \{\texttt{a...z}\} \\
\mathit{Translator}(\texttt{c}) &= \{\texttt{a}\} \\
\mathit{Translator}(\texttt{d}) &= \{\texttt{b}\} \\
\mathit{Translator}(\texttt{e}) &= \{\texttt{a...z}\} \\
&\ldots
\end{aligned}
$$

Figure 4.14: An example symbol-complete translator. This translator translates symbols to sets of symbols, any of which would achieve completeness. An example specializing this symbol-complete translator into a usable translator is shown in Figure 4.15.

Of course, this translator cannot be applied, as it translates each character to a set of characters rather than to an individual character. To generate an overapproximating accelerator from this translator, we can select one element arbitrarily from each set. Suppose we arbitrarily select the first element of each set, resulting in the translator shown in figure 4.15.

This complete translator results in the expression `[abce-z][d]*`, as all characters except `d` are translated to `a`. This may be a partially useful accelerator, as it can help us detect some patterns that do not match the pattern we wish to accelerate. It accepts all valid strings (i.e. `c, cd, cdd, ...`) but also accepts strings such as `ad, zd, xdd`. Thus, matches for this expression must be checked as it can no longer be trusted. Symbol-correctness addresses this issue, arriving at a complete translator that does not overapproximate.

4.5.4.3 Symbol Correct Translation

Symbol-correctness (Algorithm 3) begins with a symbol-complete translator. The process for arriving at a symbol-correct translator ensures that terms are not spuriously activated in the accelerator, which would cause acceptance of strings that should be rejected. For example, suppose we have the accelerator `ab*` and we wish to accelerate the regex `cd*`. We ensure that `a` and `b` are translated to characters that are neither `a` or `b` to avoid incorrectly accepting strings such as `ab, abb, ...`. We often omit or partially execute the correctness phase of stateless translator generation. This results in complete accelerators that overapproximate and leave some extra computation for

$$
\begin{aligned}
\mathit{Translator}(\texttt{a}) &= \texttt{a} \\
\mathit{Translator}(\texttt{b}) &= \texttt{a} \\
\mathit{Translator}(\texttt{c}) &= \texttt{a} \\
\mathit{Translator}(\texttt{d}) &= \texttt{b} \\
\mathit{Translator}(\texttt{e}) &= \texttt{a} \\
&\cdots \\
\mathit{Translator}(\texttt{z}) &= \texttt{a}
\end{aligned}
$$

Figure 4.15: An example symbol-complete translator derived from the sets shown in Figure 4.14. To find this symbol-complete translator, we arbitrarily select the first element of each set which determines the possible targets.

Algorithm 3 Algorithm for generating correct translators from a regex R to an accelerator A. X^c denotes the set compliment.

procedure SYMBOLCORRECT(CompleteTranslator, ToDisable)
  $\forall x$. ActiveSet$(x) = \{T \mid T$ is a term in A with x in its symbol set$\}$
  $\forall x$. MustBeActive$(x) = \{T \mid T$ is a term in A paired to T' in R where x is in the symbol set of T' $\}$
  for Each Symbol, x **do**
    ApproxSymbs = Active$(x) \cap$MustBeActive$(x)^c$
    Trans(x) = CompleteTranslator $\cap$ ApproxSymbs^c
  end for
  InactiveTerms = $\{x \mid$ ActiveSet$(x) = \emptyset\}$
  ToDisable = ToDisable $\cap$ InactiveTerms
  if ToDisable $= \emptyset$, fail.
  if Translator$(x) = \emptyset$ for any x, fail
  Select an arbitrary element from each Translator(x).
end procedure

$$
\begin{aligned}
\mathit{Translator}(\texttt{a}) &= \texttt{x} \\
\mathit{Translator}(\texttt{b}) &= \texttt{x} \\
\mathit{Translator}(\texttt{c}) &= \texttt{a} \\
\mathit{Translator}(\texttt{d}) &= \texttt{b} \\
\mathit{Translator}(\texttt{e}) &= \texttt{x} \\
&\ldots
\end{aligned}
$$

Figure 4.16: A symbol-correct translator derived from the symbol-complete translator in Figure 4.14.

the CPU. We discuss this overapproximation more in Section 4.4.2.2 and evaluate it in Section 4.7.2.

Disabling Terms In addition to ensuring that symbols do not activate the wrong terms, symbol correctness must disable certain terms completely. This enables regexes with different structures to compile to each other rather than requiring exactly the same structure between different regexes. For example, consider the structural similarity shown in figure 4.11 from the regex `a(b|c)` to the regex `h(i|jk|l)m*`. We can see that the term `jk` (represented by 2 in the accepting path algebra) and the term `m*` must not be triggered in order to maintain correctness of the underlying accelerator. We can achieve this using stateless translation by ensuring that our translator *never* translates any character to the symbol `j` or to the symbol `m`. Notice in this example that we may still translate to the symbol `k`, as if we never activate the term `j`, activations of the term `k` will have no impact.

4.5.4.4 Example

Suppose we are given the symbol-complete translator we derived in Section 4.5.4.2. We wish to ensure that no terms are spuriously activated. The intuition here is that we require that the symbols `a` and `b` activate no terms as we do not wish our stateless translator and accelerator pair to accept strings such as `abbbb`, even though the underlying accelerator would respond to such characters by default.

Executing the correctness phase, we compute the translator shown in Figure 4.16 which is the same table shown in Figure 4.5.

4.5.4.5 Heuristics to Reduce Overapproximation

There are two ways that an accelerator may overapproximate:

1. Accelerated prefix is too short (e.g. accelerating `ab*` when the whole expression is `ab*c*`)
2. Translator activates too many terms

When accelerators overapproximate, they produce matches for regions that should not be matched: the host CPU must check the intended regular expression actually matches. Given a large set of potential accelerators, RXPSC must chose the one that is likely to require the least CPU checking. In some sense this is a domain-specific problem, as some patterns that have near-zero probability in a random string may occur very frequently in particular domains (for example, the device IP address in network intrusion detection). However, we find that a domain-agnostic ranking of accelerators is sufficient to determine which accelerators are likely to perform best.

We give each accelerator two values, one based on accelerator size, the prefix cost, and one based on translator overapproximation, the overapproximation cost. These represent a measure of the frequency with which the accelerator will overapproximate. Each is calculated as follows:

Prefix Cost Measures the fraction of strings that the selected prefix will accept — it does not take into account the fraction of strings that should be accepted by a particular regex as that is likely to be (a) low and (b) is constant across all potential accelerators, so does not need to be accounted for.

$$\text{Prefix Cost} = \prod_{t \in PrefixTerms} \frac{Symbols(t)}{SymbolCount}$$

Here, *PrefixTerms* refers to all the accepting path algebra terms, *Symbols(t)* refers to the symbols that activate the term t and *SymbolCount* refers to the total number of available symbols.

Overapproximation Cost Measures the fraction of strings that are spuriously accepted due to the translator translating them to a valid string.

$$\text{Overapproximation Cost} = \prod_{t \in Terms} \frac{\text{Symbols Activating Term } t \text{ After Translation}}{\text{Symbols Activating Term } t \text{ Before Translation}}$$

We then rank all translators by the likelyhood of overapproximation using the combined metric given by $1 - \max(\text{Prefix Cost}, \text{Overapproximation Cost})$. These two costs represent the probability that, given a uniformly distributed input of characters from the set of possible symbols, that there is overapproximation due to prefixing of the new expression, and the probability that there is overapproximation due to an imcomplete translator. This ranking of translators is critical in achieving consistent performance, as for many patterns, we find the existence of many accelerators that are too short or overapproximate too much to be effective.

4.5.4.6 Time Complexity and Speed Optimizations

The symbol completeness algorithm, shown in Algorithm 2 has a time complexity of $O\left(\text{AcceleratorSize} \times \text{SymbolSetSize}^2\right)$, where AcceleratorSize is the number of accepting path algebra terms required to represent the accelerator. The symbol correctness algorithm (Algorithm 3) has time complexity $O\left(\text{SymbolSetSize}^2 + \text{AcceleratorSize}\right)$.

In the ANMLZoo benchmarks, which we evaluate in more detail later, symbol set size is always 256. The accelerator size varies from very small (we set a threshold size of 2) to hundreds of terms large. Although the time complexities of these algorithms are not excessive, we find that without careful management of set operations which are frequent in these two algorithms, the implementations can easily become very slow, particularly as we often compute multiple translators for each new regex to be accelerated in order to enable us to pick the translator with the most potential for acceleration.

4.6 Architectural Overheads

The introduced overheads of our technique are minor for the instant programmability measures they enable. Before each accelerator, we require a stateless translator, which requires 256 B of memory per regular expression. Due to signal splitting, we require twice as many FIFO IP blocks within the design to keep the branching factor low (and thus the clock frequency high). Including the duplicated FIFOs, we estimate an increase in resource usage of 10% for a typical Xilinx FPGA.

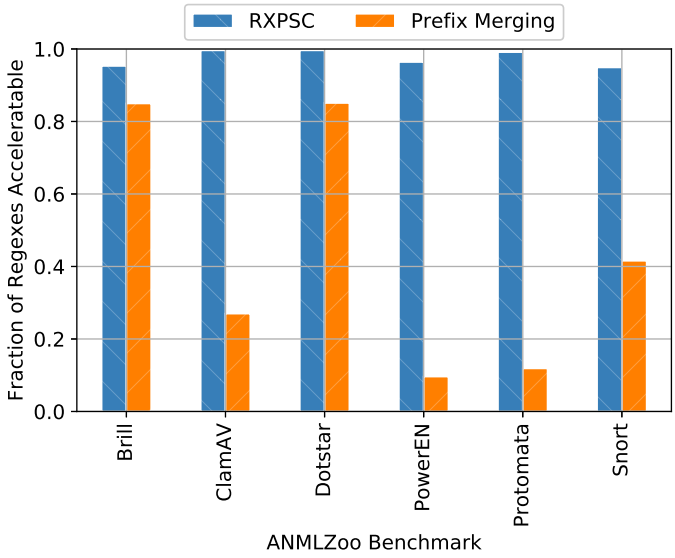


Figure 4.17: Fraction of patterns that can run on the accelerators for the other patterns in each benchmark.

4.7 Evaluation

We examine RXPSC's performance on the ANMLZoo [499] benchmarks in the regex family: Brill, Snort, Protomata, PowerEN, Dotstar and ClamAV. We also consider a network intrusion detection setting.

4.7.1 Setup

Experiments are run by taking each benchmark set, and removing a regex. The remaining regexes are compiled. We then add the new regex, by computing whether some prefix of the new regex can be represented using the other regexes. In each experiment, we generate simulators for each converted regex and run the 1 MB ANMLZoo inputs into each simulator. We compare these to the unmodified expression to compute the overapproximation rate — the rate at which the accelerator generates spurious accepts. Using the average accepting length as a proxy for the number of bytes that the CPU must then check, we can compute the fraction reduction in bytes that the CPU must scan. We run these experiments for every regex within each benchmark (˜2,500 regexes each).

RXPSC is implemented within the REAPR framework [535]. This is a python framework for generating regex accelerators on FPGAs, which we have modified to compile to stateless translators. In this section, we only provide simulation results, as

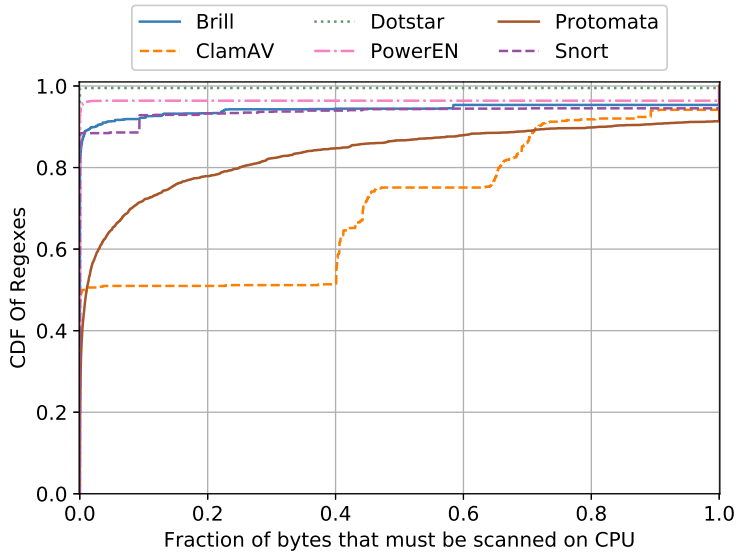


Figure 4.18: Fraction of bytes executed on the CPU using RXPSC.

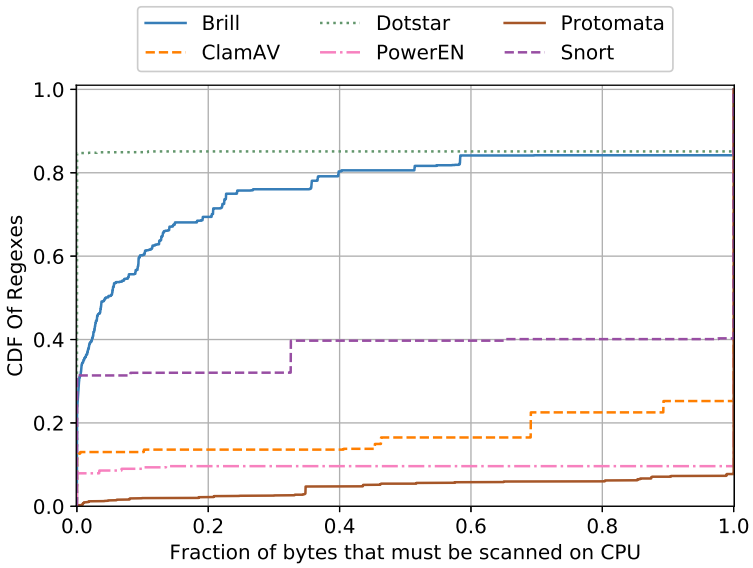


Figure 4.19: Fraction of bytes executed on the CPU using prefix merging.

we were unable to reproduce the full end-to-end FPGA compilation within REAPR.

RXPSC is particularly useful within a network intrusion detection setting as it provides easy-to-form groups of regexes and requires pattern updates without excessive compile time. We consider Snort [478], a set of network intrusion detection rules from Cisco and look at two different rule sets, the unregistered rule set (1,497 regexes). We split rules by their protocol, port numbers and IP addresses. This results in a number of different groups of regexes, that can each be run on different packets and creates a less general, but more realistic, situation than that used for ANMLZoo. For each generated accelerator, we compute the fraction of bytes that must be scanned on the CPU. We compute this by pessimistically assuming that each false-positive must scan on average the same number of bytes in an accept for that accelerator. We compare this to the number of bytes that would have to be scanned without RXPSC in place.

For the Snort experiment, we remove one *class* of regexes from the set of all regexes. A class of regexes is some set of regexes that must be run for a particular protocol/port/IP address combination. We then compile each of these regexes to the accelerators presented by the remaining regexes (of different groups).

4.7.2 Results

Figure 4.17 shows what fraction of unseen regexes RXPSC and prefix merging can find *any* accelerator for. We can see that RXPSC finds accelerators for 97% of supplied regexes on average, performing particularly well on ClamAV, Dotstar and PowerEN in this metric, where RXPSC finds accelerators for more than 99% of regexes. In contrast, prefix merging only finds accelerators for 43% of regexes.

As discussed above, finding an accelerator does not tell the full story, as for some regexes RXPSC only achieves partial offload from the CPU. We show the number of additional bytes that must be scanned on a CPU as a fraction of the total data in Figure 4.18 for RXPSC, and Figure 4.19 for prefix merging.

We see that for some benchmarks, PowerEN, Dotstar, Snort, and Brill, RXPSC is able to almost entirely remove the need for the CPU with new patterns. For others, ClamAV and Protomata, we see that the generated accelerators are only able to reduce the CPU in a smaller fraction of cases, reducing CPU by more than a factor of ten in 51% and 72% of cases respectively. We can also see that RXPSC outperforms prefix merging by a significant margin on all benchmarks, where only Brill and Dotstar can be run without complete reliance on the CPU for more than 50% of regexes.

RXPSC generates assignments reducing the quantity of data that must be scanned by the CPU by more than a factor of ten in 84% of cases, performing particularly well on Dotstar, PowerEN and Snort where RXPSC achieves this benchmark in 99.5%, 96% and 93% of cases respectively. Prefix merging reaches this threshold for 85%, 9% and 32% respectively.

The differences between the accelerator generation graph (Figure 4.17) and the bytes requiring CPU graph (Figure 4.18) are down to two key features: first, we do not require an entire regex match; and second, translators often overapproximate. For some benchmarks, such as ClamAV, regexes often begin with large series of single-character symbol sets (e.g. `\x00\x00\x00`). RXPSC is capable of finding accelerator matches for these, which under our own heuristics perform well, but these do not distinguish test data significantly. Similarly, for Protomata, a very reduced dictionary of 16 characters is used, again resulting in poor choices of accelerator to use. We expect both benchmarks could see improved performance with more appropriate heuristics for their particularities.

4.7.2.1 Network Intrusion Detection

Figure 4.20 shows that 96% of regexes can be accelerated using existing accelerators in the registered ruleset and 93% of regexes can be accelerated in the unregistered ruleset. The difference is down to ruleset size — the larger ruleset provides more accelerators to choose from. Prefix merging finds alternatives for only 49% and 25% of each ruleset, again showing the benefit of having more accelerators to choose from. In this real-world situation where we must load a new pattern onto an accelerator quickly, RXPSC does so in the vast majority of cases, reducing the volume of data the CPU must process, and freeing CPU cycles.

4.7.3 Compile Time

RXPSC is capable of compiling new regexes to existing accelerators in a number of seconds. In this experiment, we explore compile times for additional regexes using RXPSC, and compare them to the reported compile times for the ANMLZoo benchmark suite [82] on REAPR [535] (a prior version of Grapefruit), showing both the default compile times and compile times improved with compilation toolchain optimizations. Figure 4.21. RXPSC's compile times that are almost all a factor of 10,000 less than those reported for REAPR. This is because RXPSC does not require re-compilation

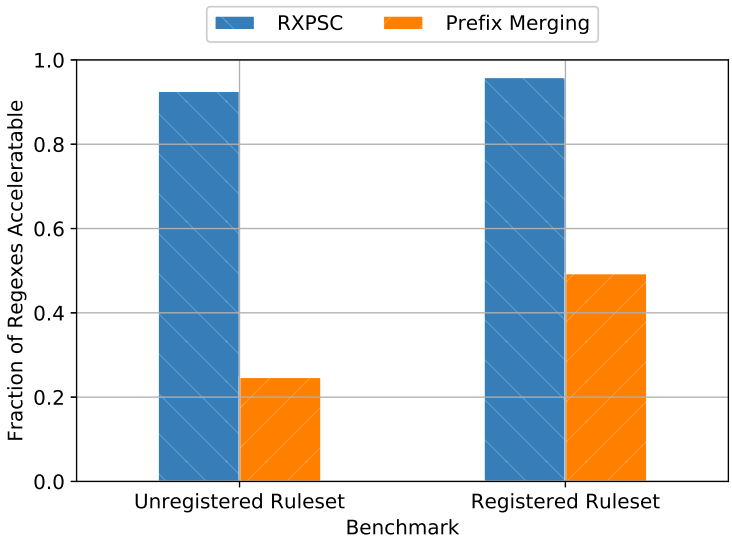


Figure 4.20: Fraction of regexes that can be supported by existing accelerators.

Benchmark	Number of States [499]	Number of Regexes	REAPR [82]	Optimized REAPR [82]	RXPSC	Prefix Merging
Brill	26,364	2050	21168 s	15864 s	1.8 s	0.2 s
ClamAV	42,543	515	17100 s	13020 s	0.9 s	0.1 s
Dotstar	38,951	3000	Unreported	Unreported	0.5 s	0.1 s
PowerEN	34,495	2860	Unreported	Unreported	0.7 s	0.1 s
Protomata	38,251	2340	23388 s	17130 s	12.4 s	0.1 s
Snort	34,480	3379	25020 s	25020 s	0.9 s	0.03 s

Figure 4.21: Time required to add additional regexes. We compare to REAPR [535], a prior version of Grapefruit, as numbers for Grapefruit are unreported.

through the FPGA tool-chain.

RXPSC's compilation time is dominated by how quickly the structural-similarity algorithm can determine which accelerators are valid and which are not. Protomata has a particularly slow compile time here because the expressions have significant structural similarity but often fail to have symbolic similarity. The other benchmarks have more structural differences between the expressions and raise fewer symbolic similarity issues, which means RXPSC is more quickly able to identify suitable accelerators.

4.8 Conclusion

We present RXPSC, a regex compilation tool capable of compiling new regexes to existing accelerators. RXPSC finds structural similarity between regexes and generates stateless translators that allow in-place accelerator updates without recompiling an FPGA accelerator. We find that we can reduce CPU workload by more than a factor of ten for 84% of unseen regexes across the ANMLZoo benchmarks, outperforming prefix merging, which only reaches this benchmark for 34% of unseen regexes in ANMLZoo. We demonstrate a usecase in network intrusion detection, where new rules must be implemented quickly in response to new threats, and again show that RXPSC achieves significant improvement over prefix merging.

This chapter explores the acceleration equation (Chapter 2) in the context of regular expressions, solving it in the context of regular expression accelerators, by setting g to be the stateless translator, and h to be the post-match check.

Chapter 5

Fourier Transform Accelerators

Specialized hardware accelerators continue to be a source of performance improvement. However, such specialization comes at a programming price. The fundamental issue is that of a *mismatch* between the diversity of user code and the functionality of fixed hardware, limiting its wider uptake.

Here we focus on a particular set of accelerators: those for Fast Fourier Transforms. We present FACC (Fourier ACcelerator Compiler), a novel approach to automatically map legacy code to Fourier Transform accelerators. It automatically generates drop-in replacement adapters using Input-Output (IO)-based program synthesis that bridge the gap between user code and accelerators. We apply FACC to unmodified GitHub C programs of varying complexity and compare against two existing approaches. We target FACC to a high-performance library, FFTW, and two hardware accelerators, the NXP PowerQuad and the Analog Devices FFTA, and demonstrate mean speedups of 9x, 17x and 27x respectively.

5.1 Introduction

Specialized accelerators deliver significant performance improvements [474]. However, specialization is in direct tension with programmability [140]. The more specialized the accelerator, the greater its potential performance [515], but the less likely it is to be used [371].

Fast Fourier Transform (FFT) acceleration is a good example of this. While there are hundreds of commercial accelerator designs [3, 13, 14, 16, 17, 73], the API calls used to program them lack the portability and flexibility of software libraries [21, 182] making offloading the domain of experts [396]. Manually migrating to new software

APIs is complex and time-consuming [155, 253], and made more challenging by the inability for APIs to hide the complex eccentricities exposed by real hardware [78, 80].

Ideally, we would like hardware to be as specialized and idiosyncratic as needed for performance. We also want existing code to automatically morph to new accelerators with no user involvement [88, 159]. Unfortunately, "*most applications require modifications to achieve high speedup on domain-specific accelerators*" [143]. Here we focus on FFT acceleration as a real-world example of this problem. We demonstrate that automatic modification *is* possible and achieve significant speedups on GitHub legacy C programs[1].

Attempts at replacing application code with accelerator library calls [191] are brittle and do not scale to real-world code or algorithms complex enough to justify acceleration [48]. The fundamental issue is *mismatch*. As the complexity of accelerator functionality increases, the likelihood that it exactly matches a user's application becomes vanishingly small [333, 531].

Mismatch occurs at a variety of levels. The most basic form is *code mismatch* where the number of different ways of writing the same algorithm defeats approaches based on code-shape. Significant mismatch also occurs at a data-representation level, *data mismatch*. Here the code and accelerator may have different types or values — for instance using a custom definition of a complex type. Further still, *domain mismatch* is common, with many accelerators only supporting powers-of-2-sized FFTs [19] or limiting the size of inputs [14]. Finally, there may be *behavioral mismatch*. For example, accelerator output values or user code may be bit reversed or un-normalized. We tackle this fundamental issue in targeting accelerators: the mismatch between user code and accelerator functionality using a novel input-output behavioral scheme using generate-and-test over fuzzing samples to find unique solutions.

5.1.1 Current Schemes

Programming accelerators typically involves rewriting code in an language or with a new API [511] but this is time-consuming and requires expert knowledge [155, 253]. Recently, work trying to automatically match and replace existing code with accelerator libraries for simple operations has used constraint matching of code to an API description [81, 126, 146, 191]. However these schemes are brittle and fail with minor code variations, and constraints are challenging to write [190]. Exact matching

[1] All code available at [24]

techniques [358, 430] fail once the code scales beyond an order of magnitude of ten instructions, and FFTs scale up to thousands. The near duplicate codes these techniques require is highly unlikely, even when implementations are copy-and-pasted [527].

There is a different stream of research aimed at code-clone detection and algorithm classification [47, 138]. Rather than focusing on code structure via constraint solving, it uses machine-learning-based embeddings of code. Codes with similar behavior will have similar embeddings. These have been successful at *labelling* sections of code [114] and we leverage this as a novel filter to our IO program synthesis. While these code-embedding schemes can identify relevant sections of code, they cannot reason, transform code or compile to accelerators. We evaluate constraint and code embedding approaches in Section 5.10.

5.1.2 Our Approach

We present FACC (Fourier ACcelerator Compiler), a compiler that maps user code to Fourier transform accelerators. FACC builds a neural classifier [47, 138] to isolate procedures within user code as candidates for potential acceleration. It then explores a space of possible bindings from user variables to accelerator parameters. Next, FACC uses input-output behavioral synthesis to generate accelerator wrappers that bridge the mismatch between user code and accelerator. This allows us to match user code as small as 12 lines of code scaling up to procedures with more than 2000.

We take two accelerators: the Analog Devices FFTA [19] and the NXP PowerQuad [18], and an optimized software library, FFTW [182], and automatically match them to unmodified GitHub code, showing large performance improvement: 27x, 19x and 9x over the original software respectively.

This chapter makes the following contributions:

- We introduce four key mismatches that must be overcome for source-code to accelerator compilation.
- We implement a synthesis-based IO-matching solution to overcome these mismatches for FFT accelerators.
- We evaluate on real-world code and show significant speedups of automatically compiling to hardware accelerators and optimized libraries.

5.2 Motivation

Compiling software to specialized hardware accelerators faces the challenge of mismatch between user code and accelerator behavior. Fourier transforms are an excellent example of this problem: they are one of the most widely used transforms in DSP [459], and offer a number of performance/flexibility tradeoffs [485]. This results in a large amount of legacy C code implemented in drastically different styles and optimized for different input sizes. Current-generation hardware accelerators for FFT can out-perform even the most optimized software implementations [23] provided we can bridge the gap between software and hardware.

Consider Figure 5.1: there is a section of existing legacy C code that performs a Fourier transform. We would like to cut it out and replace it with a call to an accelerator API. Unfortunately, there is a mismatch between the user code and the accelerator API that prevents this. FACC automatically generates an adapter that acts as a mediator between user code and the accelerator API. User code is now replaced with a call to the adapter enabling acceleration.

5.2.1 Mismatch Example

Fourier transforms can be implemented in any number of different ways [161]. Figure 5.2 shows a number of Fourier transforms that are not trivially acceleratable due to mismatches between user codes and accelerators.

The first, left-most column shows a *code mismatch*. The user code is a recursive FFT implementation, but the optimized library provides an iterative implementation. This kind of difference is extremely common in real-world code, but cannot be handled by pattern-matching solutions, which struggle to match copy-pasted code [527]. Indeed, the core FFT in the code we evaluate on ranges from 12 to over 2000 lines, representing a huge diversity in code despite performing the same task.

The second column shows a *data mismatch*. The user code uses a custom complex type different from the complex representation used by the accelerator. To run this code on the accelerator, the types must be de-constructed and mapped to the inputs to the accelerator.

The third column shows a *domain mismatch*. The user code implements a mixed-radix FFT, which allows for inputs of many different lengths, but the Analog Devices FFTA only supports inputs that are powers of two. As a result, only some inputs to the original user code can be accelerated and dynamic or static checks must be added.

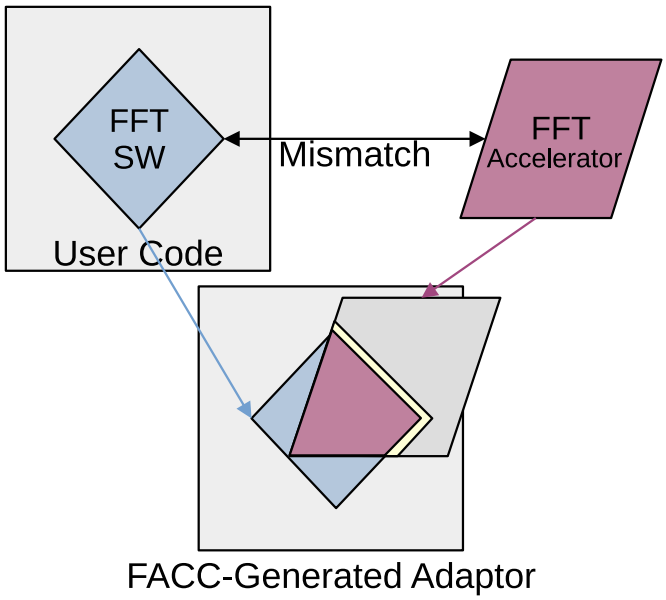


Figure 5.1: FACC takes user code and an accelerator interface as input and produces an adapter that appears identical to the user code, but uses the accelerator. FACC identifies target regions in the user code that implement FFTs (highlighted in blue), then automatically finds and replaces compatible parts of the FFT with their hardware equivalent where functionality overlaps, while falling back to the software for other operations. Code is synthesised to bridge the gap (in yellow) in implementation and data structures.

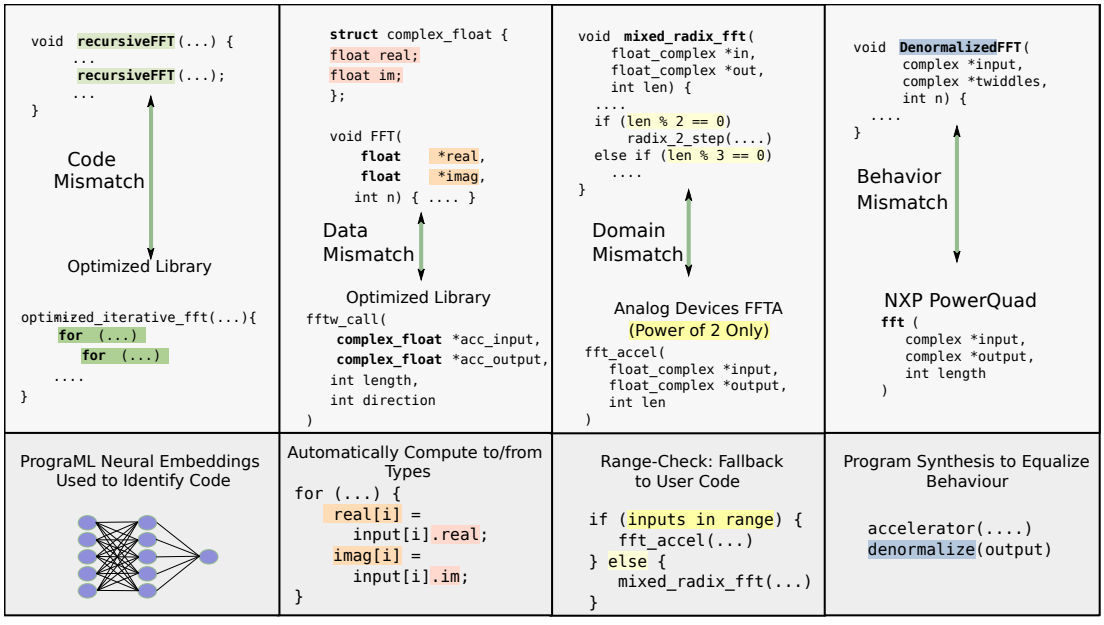


Figure 5.2: Examples of common mismatches between source code and accelerators, with FACC's resolution below them.

Finally, the right-most column shows a *behavioral mismatch*. The user code implements an FFT but does not normalize the result. To undo the normalization the accelerator does perform, an adapter that denormalizes the output from the accelerator must be used.

Despite these mismatches, this code is acceleratable provided the code, data, domain and behavioral gaps are bridged.

5.2.2 The General Challenges of Generic FFT Support

There are potentially an unbounded number of differences between functionally equivalent FFTs. Mismatches of code, data domain and behavior can all be handled by FACC's combination of code detection, program synthesis, and generate-and-test IO equivalence.

5.2.2.1 Code Mismatch

Different programmers use different strategies for solving the same problem. This results in incidental differences between implementations which defeat constraint-based approaches. FACC achieves independence of coding style by first using code neuro-embedding to find candidate regions in user code. Once it has synthesised candidate adapters for these regions, FACC uses IO examples to test whether the adapter and original candidate code are behaviorally equivalent.

5.2.2.2 Data Mismatch

Different implementations of the same algorithm can use different representations of the same data. FACC explores the space of possible mappings between user code and accelerator API variable types via binding synthesis. It uses constraints on data types and variable ranges to reduce the space of possible mappings which are then later evaluated for input-output (IO) behavioral equivalence.

5.2.2.3 Domain Mismatch

A valid input to user code may not be a valid input to an accelerator and vice-versa, and this causes complex constraints on functional equivalence between accelerator and code. For example, the Analog Devices FFTA [17] only supports inputs that are powers of two of size greater than 64 and less than 2048 in small mode, and 65536 in large

mode. There are two issues to deal with here. The accelerators may not support the full range of inputs that the user code supports. This is a task that requires the generation of a static or dynamic range check. The user code may also not support the full range of *its own* inputs, either throwing errors or resulting in undefined or arbitrary behavior when fed with unintended random inputs, which can make equivalence testing difficult. FACC uses value profiling [92] and range analysis [214] to address this problem.

5.2.2.4 Behavioral Mismatch

Accelerators may not implement the same functionality as user code. To make code match, we either specialize or generalize.

Behavioral specialization is where accelerator input/configuration parameters are assigned constant values. For example an accelerator may support both FFT and IFFT algorithms, but user code may only implement FFT and so the accelerator should be specialized to match the user code.

Behavioral generalization is where software performs some function that the accelerator does not. For example, the user code may compute un-normalized results, while a hardware accelerator may return normalized results. A software function should be used to generalize the accelerator to produce compatible results.

5.2.3 Correctness

Implementations of FFTs vary between tens of lines of code and thousands (see Section 5.10.1) and handle arrays of floating-point numbers. Proving traditional correctness is impossible, as different implementations have different error properties [317, 342]. Correctness is further complicated by a lack of formal models available for commercial hardware accelerators, whose designs are often corporate secrets [12]. Even if these issues are overcome, modern floating-point theorem provers are not capable of proving equivalence of such large-scale floating-point algorithms.

Instead of relying on formal proofs of equivalence, we use a pragmatic approach based on fuzzing via input/output examples to determine behavioral equivalence in a number of test cases. Once FACC has confidence that the code can be replaced, it is the developer's role to sign off the source-code replacement via code output in the source language they understand (e.g. Figure 5.3). False positives are very rare without malicious input designed to disguise itself as an FFT. During our evaluation,

we encountered no false positives.

5.3 Fourier Transforms

The Fast Fourier Transform (FFT) was (re)discovered by Cooley and Turkey in 1964 [132], and has since revolutionized signal processing. In this section, we provide an overview of the FFT.

5.4 Math

The Discrete Fourier Transform (DFT) is the following sequence:

$$DFT[i] = \sum_{n=0}^{N-1} x[n] \times e^{-j\frac{2\pi nk}{N}}$$

Given inputs x, the DFT produces the frequencies that make up these inputs. It can be used for a wide range of techniques, from image compression [327] to processing analogue signals. However, computed in the manner above, the DFT requires $O(N^2)$ operations in the length of the input x. The FFT improves on this, using a butterfly of multiplications and twiddle factors. A typical radix-2 FFT only works on arrays that are sized by powers of two — a modified algorithm is used for all other array lengths. However, the vast majority of implementations focus on lengths that are powers of two. A typical approach that enables the use of these implementations for non-power of two inputs is to pad input data with zeroes until it is a power of two. For most usecases, this is a suitable tradeoff, however this means that FFTs loose their orthogonality traits [485] and the cost of padding to the next power of two is can be too high [356].

5.4.1 Bit-Reversal

The result of a typical FFT provides the results out of order. There is a *bit-reversal* step that must be computed. The bit-reversal step reverses the bits in any individual index (e.g. 0001 becomes 1000). The bit-reversal step can be done on the inputs, the outputs or implicitly within the FFT algorithm, but must be done somewhere.

5.4.2 Decimation in time (DIT) vs decimation in frequency (DIF)

There are two key ways to structure the butterfly computation. Either, we can do the multiplication with the twiddle factors before the corresponding data crossover or

afterwards. Both are valid and result in different orderings of the FFT operations.

There are a large number of ways to arrange the butterflies depending on whether you want to do bit-reversal on the inputs or the outputs, and whether you want to do the small FFT operations first or last [390].

5.4.3 Efficiently Managing Twiddle Factors

Twiddle factors have a large effect on algorithm-design. The computation of twiddle factors on-demand reduces memory-bandwidth requirements in bandwidth-limited solutions [477]. Despite efforts to reduce the overhead of computing twiddle factors [394], computing twiddle factors in the general case is expensive [532].

Due to this overhead, many implementations choose to store precomputed twiddle factors in memory. Loading twiddle-factors from memory can be cache-unfriendly due to large strides between required indexes. Thus, many implementations arrange the twiddle factors based on their use order. However, this use order is often dependent on the exact algorithm used, and so generic twiddle factors are not always possible [121].

5.4.3.1 Twiddle-Factor Free FFTs

Some FFT algorithms are twiddle-factor free. These algorithms have the desireable property of removing many otherwise required `sin` and `cos` operations.

Twiddle-factor-free algorithms are based on the idea of using smaller-length DFTs for lengths that are co-prime factors of the original FFT. These are then combined to produce the longer FFT arrays [161]. Good's algorithm is a good example of this.

5.4.4 Radix-N FFT

In the FFT, the radix refers to the number of signals that go into each butterfly flap. The most commonly used radix is radix-2, but using a larger radix can be worthwhile as it reduces the number of multiplications needed, an effect magnified for larger FFTs. The downside of using a larger radix is the restricted range of viable input sizes. For a general radix-N FFT, the inputs must be of size N^r.

5.4.5 What Other Algorithms Could We Accelerate with an FFT Accelerator?

The DCT (Discrete Cosine Transform) is one example of a function that can be computed with an FFT, but often is not. A convolution can also be accelerated using a combination of an FFT and an inverse FFT. We can also explore the multi-dimensional FFT, which can be broken down into multiple single-dimensional FFTs.

Less drastic, but possibly more useful is to accelerate FFTs of different sizes than the ones supported by the underlying accelerators. We can use the existing accelerator to support behaviour like this using algorithms like the ones discussed above.

5.5 System Overview

FACC uses Input-Output (IO)-based program synthesis to generate an adapter that is a drop-in replacement for the original user code, matching the output behavior for all inputs even though the implementation is different. Given some accelerator performing a function A and some user code performing a function U, FACC finds adapter functions g, h such that $\forall x.U(x) = g(A(h(x)))$, where x represents test input. Crucially, this test for IO equivalence is invariant of the exact structure of the code, which in FFTs can vary from tens to thousands of lines, and so can match any code which given the same inputs produces the same outputs. Adapters are created via a generate-and-test approach, by generating many plausible candidates, filtered first using known constraints and heuristics, before all but one option is eliminated using fuzzing. Finally, the synthesised adapter is presented to the user for verification.

5.5.1 A Generic Framework for Accelerator Support

Our key insight is that to support an accelerator performing function A, and use it to accelerate diverse user code U, we must patch the difference using functions range (r), pre-binding (b), post-binding (b'), pre-behavioral (s) and post-behavioral (s') such that

$$U = \text{ if } r \text{ then } s \circ b \circ A \circ b' \circ s' \text{ else } U$$

where each function provides the following behavior:

b, b' address the *data mismatch* problem by mapping between accelerator variables and user-code variables. b takes user-code inputs and produces conversions to

```
complex *FFT_accel(complex *x, int N) {
  // Check if valid accelerator inputs
  if (power_of_two(N) && N <= 65536) {
    // Bind user inputs to accelerator
    int len = N;
    #pragma align 64
    complex_float output[len];
    complex_float input[len];
    #pragma end
    for (int i = 0; i < len; i++) {
      input[i].re = x[i].real;
      input[i].im = x[i].imag;
    }
    // Call accelerator
    accel_cfft(input, output, len);
    // Bind accelerator outputs
    for (int j = 0; j < N; j++) {
      x[j].imag = output[j].im;
      x[j].real = output[j].re;
    }
    // De-normalize outputs
    for (int k = 0; k < N; i ++)  {
      x[k].imag *= N;
      x[k].real *= N;
    }
  } else { // Not valid input
    // Fallback to user code.
    UserFFT(x, N);
  }
}
```

Figure 5.3: A drop-in replacement for user code generated by FACC. The Analog Devices FFTA used here requires that inputs are 64-byte aligned, and is out-of-place, while the user's code is in-place. Pre-binding is highlighted in gray, post-binding in pink, post-behavior in green and range in orange — pre-behavior is empty in this case.

accelerator inputs, while b' takes the accelerator outputs and converts them to user outputs (Section 5.7.1).

r addresses the *domain mismatch* problem with *input range checking* to determine whether the inputs presented can be run on the accelerator. FACC does this with a mix of static and dynamic analysis, generating the minimal possible check with the static information available (Section 5.7.2).

s, s' address the *behavioral mismatch* problem by adding or undoing accelerator functionality to match the user code. FACC sets s to the identity function, as many pre-behavioral FFT problems have a post-behavioral equivalent s' which can be used instead (Section 5.7.3).

An example output is shown in figure 5.3. In order to match the accelerator's data format (gray) the adapter converts the user code's input to a different datatype — aligned and changed to be out-of-place. After accelerator execution, the adapter restores the in-place representation (pink). The normalization performed by the accelerator but not the user code is undone (green). If the accelerator's constraints on size and being a power of two aren't met, the user code is run instead (orange).

Generic and Domain-Specific Components The framework described above is domain-agnostic. However, to make the synthesis problem tractable, some parts are domain-specific. In particular, our solution to behavior mismatch relies on sketch-based synthesis [449] and is domain-specific to FFTs. We expect our sketches to be easily extendable to new domains. Our solutions to the data mismatch and domain mismatch problems are general and applicable to many types of accelerator.

5.5.2 Operation

FACC uses synthesis to generate an adapter that enables drop-in accelerator use. Multiple candidates are generated and tested against the user code to pick the correct one. Figure 5.4 shows the stages of the tool:

1. An API to compile to and limitations of the hardware are provided as input.
2. **Candidate detection** discovers potential targets using neural classification [138], and analyzes user code using static analysis to aid in generating a match (Section 5.6.3).
3. **Synthesis** generates candidates for the r, s, s', b, b' functions, discarding those made invalid via constraints and heuristics (Section 5.7).

4. **Generate and Test** filters the combination of all possible matches using IO tests to generate a drop-in replacement (Section 5.8).

5.6 Identifying Acceleratable Candidates

FACC bridges the gap between user code and accelerator behavior by generating adapters. Before it can do that, it employs an existing tool [138] to identify candidate acceleratable code regions. FACC then gathers information on how variables within code regions are used to drive adapter synthesis.

5.6.1 Identifying Acceleratable Regions

FACC is a binding tool, using a neural classifier based on ProGraML [138], to detect likely acceleratable FFT-based code.

Data We use the OJClone algorithm classification dataset introduced in Mou et al. [350] consisting of 105 classes, each composed of different implementations of the same task. We add FFT as an additional class, with the same FFT code snippets obtained from Github used in the rest of the article. We restrict all classes to 20 instances for a balanced dataset. Each instance is parsed and transformed into a data flow graph of its LLVM instructions with ProGraML [138], which uses the instruction characteristics as node information and dataflow and controlflow as edges. Due to the reduced size of the dataset, we implement 10-fold cross validation, such that each train split contains 80% of the dataset and the remaining 20% is left as holdout.

Model We implement a Graph Convolutional Neural Network with two graph convolutional layers followed by max-pooling and a linear layer to perform the actual classification, using PyTorch [385] and DGL [505]. We do not perform any hyperparameter search (instead, set reasonable default values), and use the Adam optimizer [269] with weight decay as regularization. All models are trained for a maximum of 100 epochs using early stopping with a patience of 10, which led to convergence in all experiments.

Identifying Invalid Regions No code detection tool is perfect, and so ProGraML may misclassify algorithms. FACC evaluates all of these as potential generate-and-test targets, and if an invalid region (i.e. one not matching the accelerator interface) is identified, FACC will fail to generate valid bindings and leave the spuriously identified

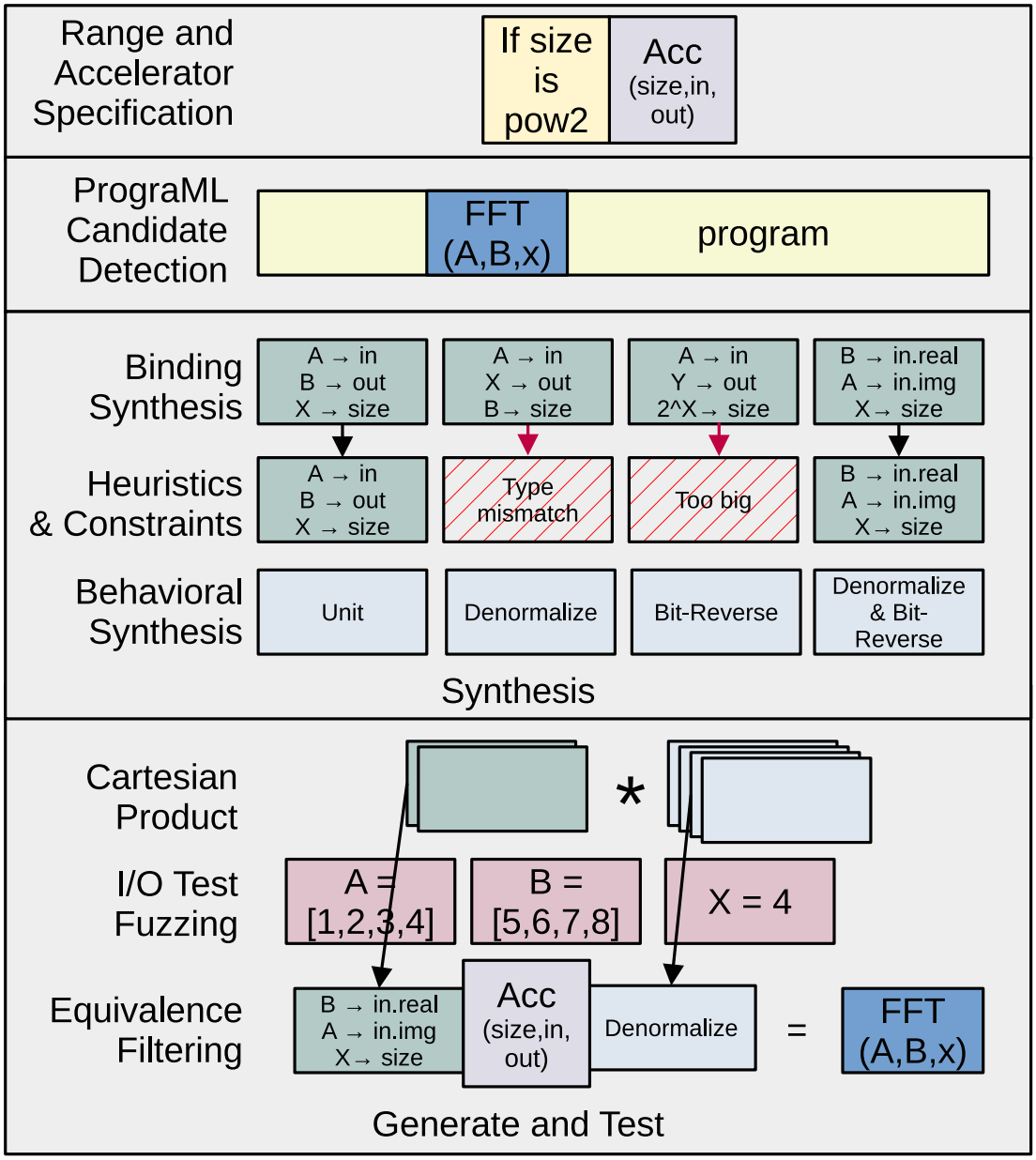


Figure 5.4: FACC takes a specification of an accelerator, and produces an equivalent version of the original program with acceleration. It uses neural embeddings to find plausible candidates for replacement, then creates a set of possible input and output bindings, filtered by constraints and heuristics. It then tries to patch the functionality of the accelerator to match that of the user code via behavioral synthesis. Finally, FACC generates all possible combinations of these mappings, and tests them for IO equivalence with the user code.

region unchanged. In this sense, the neural classifier is used to cut down the search space: rather than considering all instruction sequences of all programs as possible targets, it only tries to match those labelled by the neural classifier[2].

Code Mismatch Identifying code regions is only the first part of overcoming code mismatch. The second is that code itself is highly diverse; our evaluation set ranges from 12 lines to over 2000 for similar behavior. In Section 5.8, input-output (IO) testing is used to test whether the adapter synthesized in Section 5.7 matches the behavior of the identified code. IO-testing allows us to ignore the underlying code structure eliminating code mismatch by focusing only on the interface.

5.6.2 Identifying Input/Output Variables

FACC relies on existing liveness analysis to determine which variables are output variables and which are input variables. This allows for extraction of functions with side-effects, or extraction of sub-function regions of code. We use variable range analyses [214, 294, 323], points-to analyses [67, 213] and value-profiling [92] to reduce compilation time.

5.6.3 Type Inference

FACC expands types in two ways: by inferring the lengths of arrays, and by inferring more structured types over base types where they may be required by the accelerator.

This step takes a single type from the user code as input, and produces a number of plausible extended types to use for the remainder of the synthesis as an output. A pseudo-code type augmentation algorithm is shown in Algorithm 4.

Length Inference Arguments passed as arrays to functions often have a variable number of values. For example, a type signature that takes a single integer as argument can only take a single input, but a function that takes an array can take `N` inputs, where `N` is the length of the array. In languages like C, array lengths are implicit, not directly specified by the programmer. Although best-effort compiler passes can assist with providing this information [321], FACC is able to infer array lengths using a generate-and-test approach. Each array is assigned a number of possible length parameters, and the correct one is determined during testing.

This testing is done by selecting random values that are assumed to correspond to

[2]Code is available at [25]

Algorithm 4 Type Augmentation Algorithm. Takes a type as input, and produces all plausible types that can replace it. IsCompatible is a function encoding the heuristics of the type search, and marks different types of integer (e.g. `int32`, `int64`) as compatible, but types such as `bool` and `float32` incompatible.

procedure AUGMENTTYPES($Type_{in}$)
    Types $= \emptyset$
    if IsArray($Type_{in}$) **then** ▷ Infer Lengths
        for len $\in$ Possible Length Variables **do**
            Add $Type_{in}$#len to Types
        end for
    else
        Add $Type_{in}$ to Types ▷ Not Array so No Length
    end if
    for $T \in$ All Possible Types **do** ▷ Infer Structure
        for $T' \in$ Types **do**
            if IsCompatible(T, T') **then**
                Add T to Types
            end if
        end for
    end for
    return Types
end procedure

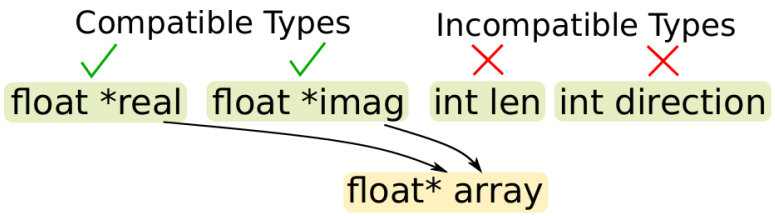


Figure 5.5: Type constraints reject two impossible bindings. The green boxes (top) are user-code variables. The yellow box (bottom) is an accelerator variable, and each arrow marks a mapping that was considered.

array length. When the wrong length parameter is picked, it will be a different value to the correct array length parameter[3]. If the value is too small, the array will not be accessed to its full extent, so some of it will remain unchanged. If the value is too large, an access to the array will segfault, triggering the address sanitizer, which we enable.

Structure Inference API designers are often encouraged to present APIs with the most syntactic information possible [218]. The user code faces no such restrictions. As a result, FACC needs to infer more syntactic information over base types. All plausible (dependent on the types in the accelerator API) inferred types are considered, and filtered via generate-and-test (see Algorithm 4).

5.7 Synthesis

Here we describe the core accelerator support problem. We address three key mismatches: data mismatch using binding synthesis, domain mismatch using range-check generation, and behavioral mismatch using behavioral synthesis.

5.7.1 Data Mismatch: Binding Synthesis

In binding synthesis, we take a set of input variables and a set of output variables from the user code. We generate every mapping that Type Inference (section 5.6.3) does not allow us to eliminate either via constraint or heuristic, between these variables and the accelerator API variables, to be evaluated using generate-and-test. Figures 5.5 and 5.6 show an example creating possible bindings for a single variable while rejecting those statically known to be impossible. Figure 5.7 shows a full candidate mapping.

[3]This is a requirement of the testing inputs: that every pair of parameters must differ in value in at least one example.

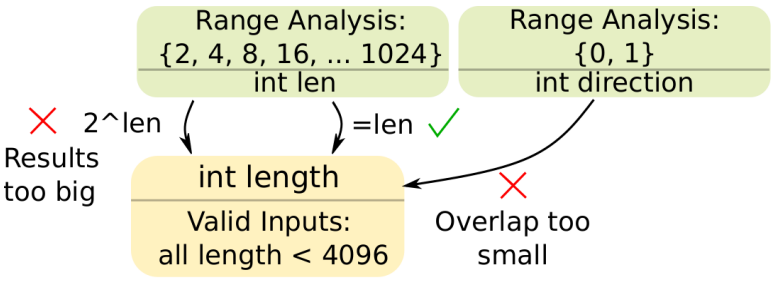


Figure 5.6: A non-trivial conversion (2^n) is considered, but ruled out due to range heuristics. The green boxes (top) are user-code variables, along with their range analyses. The yellow box (bottom) is an accelerator variable, along with the input constraints specified for that variable. Each arrow represents a mapping that was considered.

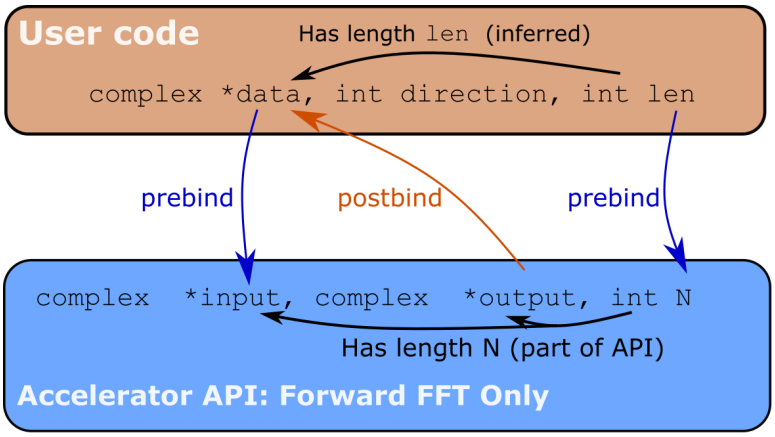


Figure 5.7: An example solution to the binding problem. To determine the correct binding, FACC tries all plausible bindings that cannot be statically determined impossible. API information, like specifying the length of an array, is provided by the Accelerator Specification, which specifies information about the API.

5.7.1.1 Non-trivial Conversions

The vast majority of accelerator parameters can be copied directly from parameters existing in user code. However, frequently, the same information is encoded in indirectly compatible ways. A typical example is using N to directly encode array length, compared to using 2^N to represent array length. Another typical example is the many different ways that a flag can be represented in C: 0 and 1, -1 and 1, 1 and 0, etc. FACC generates conversions allowing compatibility between implementations with different flag values. Variable-range information is used to vastly reduce the search-space of conversions.

5.7.1.2 Constraints

FACC applies constraints to generated bindings, limiting the search of impossible matches.

Type Conversions If a variable x is to be assigned to some variable y, then there must be a known conversion between the two types, including over distinct representations of complex numbers.

Array Assignments If any two array variables share a length variable, then the arrays that they are assigned to must also share a length variable — and those two length variables must be assigned to each other.

5.7.1.3 Heuristics

FACC also applies a number of heuristics to the bindings generated.

Range Heuristics are applied to determine whether the accelerator is likely to be useful. For example, if a variable x may take any one of 100 values (as determined by a variable range analysis), and is assigned to an accelerator API variable y, which only supports one value, the odds of successful acceleration are extremely small, so the binding is not considered likely (figure 5.6).

Single-Read Heuristics FACC assumes that user-code variables should only be read from once when assigning to accelerator variables. This heuristic greatly reduces the synthesis space by assuming a lack of unneeded redundancy in the programmer's original code.

5.7.2 Domain Mismatch: Range-Check Generation

Fixed function accelerators are often extremely specialized — significant performance is possible by making the common case fast. However, legacy C code is more general in scope.

It is important that offloaded code only operates within the valid range of the accelerator. To ensure this, we synthesize range checks, which offload to the accelerator if the inputs are valid, and fall back to the user code otherwise.

We use two sets to determine the overlap region of an accelerator and user source code.

Accelerator Specification The accelerator API is expected to specify what set of inputs the API functions on. These inputs are used both to direct testing of compatibility, and to generate input-range checks.

User-Code Analysis Inter-procedural range analysis complements the accelerator specification by allowing FACC to reduce the quantity of input checking to the intersection of the accelerator's range and the user code's range, rather than all possible FFT inputs.

5.7.3 Behavior Mismatch: Behavioral Synthesis

Behavioral synthesis introduces adapters that make accelerators transparently compatible with more user code. For example, suppose we have a user FFT function that does not normalize the results, even though the FFT accelerator available does. We use post-behavioral synthesis to generate de-normalizing code and enable accelerator use while allowing the programmer to use de-normalized results.

We implement domain-specific post-behavioral synthesis program using sketch-based synthesis [449]. For FFT functions, there are a small number of behaviors that are often omitted: normalization/denormalization and bit-reversal.

We provide a number of sketches with holes, and a procedure to fill the holes and produce all options. No infinite sketches are allowed — all sketches must be finite once holes are filled, and there must be a finite number of ways to fill each hole. Generated candidates are tested against user code.

5.8 Generate and Test

FACC is an Input-Ouput (IO)-based synthesis tool. The candidate adapters generated by synthesis are compared to the original code using fuzzing to determine equivalence. The working adapter is output in the original source language (figure 5.3) and used as a drop-in replacement in the user's code.

5.8.1 Random-Input Generation

Tests are randomly generated with a bias towards smaller examples that run more quickly. Examples are constrained to be within the computed range analyses of user code, and the valid-input range of the accelerator. As discussed in Section 5.6.3, variable-length arrays have inferred length variables and so order of generation is important. We use a topology sort to ensure that that variables are assigned in a valid order. In addition to IO-equivalence, AddressSanitizer [427] is used to detect arrays with incorrect length parameters assigned by detecting out-of-bounds accesses.

5.8.2 Challenges with Bounded Model Checking

Bounded model checking is an approach where a theorem-proving tool shows that a program cannot enter a specified error state, or provides a counter example. Given that the accelerators we support have bounded input sizes and for other sizes we call the original code, bounded model checking is sufficient. However, FFT algorithms are reliant on floating-point analyses and fall into a significantly harder category of model checking. The input to a floating-point model checker can be phrased as:

```
float *u = user_fft(...);
float *a = accel_fft(...);
float e = error(u, a);
assert (e < threshold);
```

Despite the portability of IEEE 754 floating point [20], it is designed for small-step operations, rather than full algorithms such as FFTs. Floating-point tools such as XSat [183] or Klee [91] can accept bounded model checking problems that could theoretically prove equivalence between functions within accuracy bounds. Existing techniques fall far short from being computationally efficient enough to prove the correctness of complex floating-point functions. Beyond this, a definition of the `error` function is challenging in the case of floating-point arrays as simple definitions like

mean-squared error tend to have deficiencies around very large or very small numbers, and even more so in the case of FFTs, which have inputs (exact sine-waves) for which they are unstable.

FACC requires programmer sign-off due to imprecision of hardware and software implementations, as well as the IO testing mechanism. Instead of providing these guarantees, the IO testing provides high confidence that the accelerator is equal to the original code (under various assumptions about the algorithms, which are discussed in more detail in Section 8.2).

5.8.3 Numerical Characteristics

Different implementation strategies, and by extension different accelerators, have different numerical characteristics. These numerical characteristics are typically part of the accelerator's documentation. Internally, FACC relies on mean-squared error (despite the challenges with it discussed above), and the tolerable mean-squared error (MSE) is a controllable parameter. This enables users to prohibit certain types of hardware accelerator that would frequently be outside the error they are willing to accept. Further, FACC enables testing with particular examples that are particularly relevant to the programmer, ensuring that the accelerator performs within error in those cases.

In general, it is left to the programmer to determine whether the numeric characteristics of a particular accelerator are suitable for the task they have at hand: and the support that FACC provides makes these assurances easier.

5.9 Setup

We search GitHub for "FFT", and restrict the results to C. Of the first 100 results, we have identified 24 distinct complex floating-point FFT implementations after excluding buggy code[4], code with missing dependencies, clones and implementations in different languages. We have added the FFT in MiBench [209]. We have placed these 25 implementations into a benchmark suite, and used FACC to compile from each. Where required, we have constructed a value-profiling environment, to enable FACC to compile the benchmark to the accelerator.

Implementation FACC is implemented using OCaml, with behavioral synthesis

[4]Buggy should be interpreted as "the authors were unable to make the code produce correct results to the Fourier transformation."

Table 5.1: Features of each benchmark used, representative of a wide range of implementation styles, from highly-optimized several-thousand line implementations to short, simple Discrete Fourier Transforms (DFTs).

Project	Lines of Code	Lengths Supported	Algorithm	Twiddle Factors	Imaginary Numbers	Pointer Arithmetic	Loop Structure	Optimizations
0	83	Only 64	Radix-2 FFT	Constant	Custom	No	While-True-Break	Minimal
1	278	Powers of 2 (≤ 256)	Radix-2 FFT	Constant	Custom	No	Do-While/For	Minimal
2	65	Powers of 2	Radix-2 FFT	Computed in FFT	Custom	No	For/Recursive	Minimal
3	107	Powers of 2	Radix-2 FFT	Computed in FFT	Custom	No	For	Minimal
4	934	All	Mixed-Radix FFT	Computed in FFT	Custom	No	For/Recursive	Unrolling
5	2159	All	Mixed-Radix FFT	Pre-Computed	Custom	Yes	For	Vectorized/Unrolled
6	77	Powers of 2	Radix-2 FFT	Computed in FFT	Custom	No	For	Minimal
7	237	Powers of 2	Radix-2 FFT	Pre-Computed	Custom	Yes	For	Minimal
8	101	Powers of 2	Radix-2 FFT (DIF)	Computed in FFT	C99 Complex	No	For	Minimal
9	1627	All	Mixed-Radix FFT	Pre-Computed	Custom	Yes	For/While/Recursive	Unrolling
10	75	Powers of 2	Radix-2 FFT	Pre-Computed	Custom	No	For	Minimal
11	538	All	Mixed-Radix FFT	Pre-Computed	Custom	Yes	Do-While/For	Memoization
12	367	All	Mixed-Radix + Bluestein	Computed in FFT	Custom	No	For/Recursive	Unrolling
13	101	Powers of 2	Radix-2 FFT (DIT)	Computed in FFT	C99 Complex	No	For	Minimal
14	314	Powers of 2	Radix-2 FFT	Computed in FFT	None	No	For	Minimal
15	215	All	Recursive FFT	Computed in FFT	C99 Complex	No	Recursive	Minimal
16	20	All	DFT	Unneeded	C99 Complex	No	For	None
17	12	All	DFT	Unneeded	C99 Complex	No	For	None

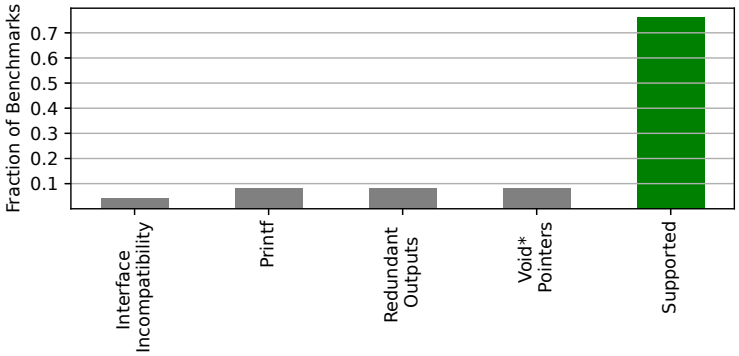


Figure 5.8: FACC success and failure classification.

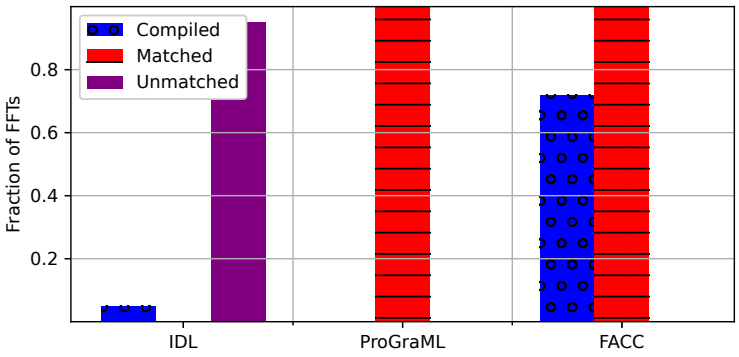


Figure 5.9: Performance of different strategies: constraint matching, neural embeddings and FACC.

libraries implemented in C. FACC currently has a C backend which is compatible with toolchains for the various backend targets. In total our implementation is 13,000 lines of OCaml, with 1,000 lines for range check generation, 1,000 lines for behavioural synthesis, 3,000 lines for binding, and 4,000 lines for backend-specific generation and the remaining 4,000 used for various utilities. All compiler and benchmark code is available at [24].

Experimental Setup Codes were placed in a benchmark suite that tests them on inputs that could be accelerated by the accelerator in question. We evaluate on three platforms:

FFTW: A desktop environment running Windows Subsystem for Linux and using an Intel i9-10900X processor and the FFTW optimized library. Code is available at [24].

ADSP board (SC589/FFTA): A multicore embedded environment using the Analog Devices ADSP-SC589 Development board with an Arm Cortex A5 as a primary core, an *SC589* SHARC DSP core and an *FFTA* Fourier transform hardware accelerator. Code is available at [26].

NXP Board (Powerquad): A single core embedded environment using the NXP LPC55S69 Development board with an Arm M33 as a primary core and an NXP *PowerQuad* accelerator capable of accelerating Fourier transforms. Code is available at [28].

Competitive Approaches We evaluate IDL [191], an existing constraint based approach to identifying code sections for acceleration. We evaluate our ProGraML-based classifier's [138] speedup by offloading FFTs to an SC589 DSP core. FFTs can be offloaded to the SC589 DSP core simply by identifying them, but the semantic information required to offload to the FFTA is not inferred. Rather, we use ProGraML as a hint that the code is likely to perform better on the DSP than the CPU.

5.10 Results

We evaluate FACC along several dimensions, comparing against success rates of IDL and ProGraML (Section 5.10.2), performance of IDL and ProGraML (Section 5.10.3), performance across multiple platforms (Section 5.10.4) and properties of the compilation (Section 5.10.5).

5.10.1 Which Benchmarks Does FACC Support?

FACC compiles 18 of the 25 implementations as shown in Figure 5.8. Table 5.1 shows a summary of the code features used in the projects FACC is able to compile. We can see that implementations vary both at the level of functionality they support, with different implementations supporting different lengths of input, and in the way they implement the Fourier transform. Approaches vary between 12 and 2,159 lines of code, using iterative and recursive approaches. A number of implementations unroll loops and base cases by hand to achieve better performance, while others introduce memoization between calls and others still use hand-vectorized instructions. It is very common to use custom-defined complex types, rather than the standard C99 type.

Figure 5.8 shows why FACC cannot compile some cases. Printfs during execution results in observably different behavior than can be supported on an accelerator that does not print to stdout. Void* pointers and Integer FFTs both require more implementation work to support the appropriate type conversions required. Support for nested memory structures requires implementation of support for nested calls to malloc. The features to support these are work in progress.

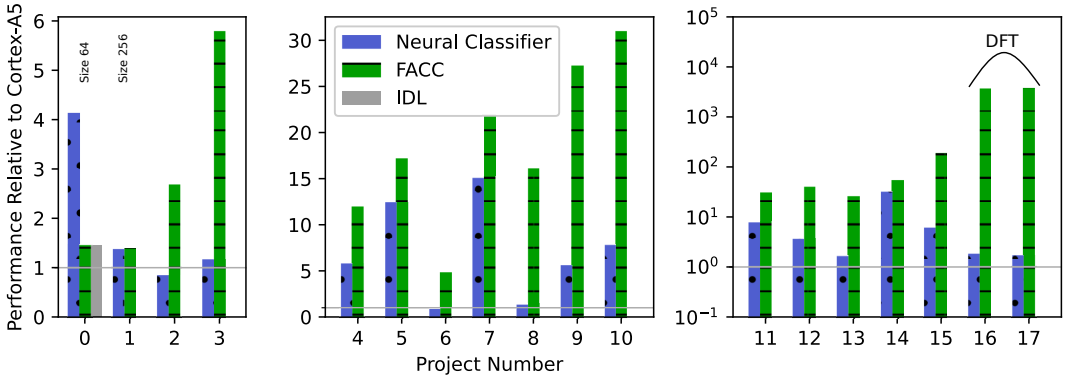


Figure 5.10: Comparing offloading techniques between on the Analog Devices ADSP-SC589 Development board. Inputs of size 1024 are used unless otherwise noted. An Arm Cortex-A5 is the master core, and can offload either to the SC-589 DSP or to the FFTA accelerator. A neural embedding is used to offload to the DSP core and achieves geometric mean speedup of 3.5x. FACC offloads to the FFTA, and achieves corresponding speedup of 27x.

5.10.2 Which Benchmarks Do IDL and ProGraML Support?

Figure 5.9 shows the performance of three different compilation techniques on our benchmark suite.

IDL For IDL [191], we design a pattern for project 0 (in Table 5.1). We can see that IDL can compile the single benchmark we hand-crafted a pattern for, but cannot generalize. Figure 5.12 shows why: from our workload set, no pattern becomes similar enough to any other past 50 lines, and most diverge much more quickly. While simple function prologue snippets are sometimes similar enough to allow us to match them between functions for a few lines, the level of code mismatch in the core FFT algorithm makes this strategy ineffective. Even if we charitably try to match the two simplest codes, 16 and 17 at 20 and 12 lines respectively, we immediately fail; they use different library functions for complex arithmetic.

ProGraML By contrast, the modified ProGraML [138] classifier is effective at detecting FFTs: Figure 5.11 shows a cross validation. Top-1 refers to classifying a code region solely by the highest predicted probability class. Top-3 refers to considering those 3 classes with the highest probability. The FFT top-3 recall reaches 100% with as few as 11 examples. Using top-3, we also find precision converges rapidly to 1. This means FACC will try binding on all code regions labelled FFT by Top-3, discarding those where there is no legal binding, to avoid false-positive code outputs — it is better to have a classifier that identifies too many regions than too few regions.

Although we use a top-3 scheme, a top-1 scheme for FFTs provides a different performance point with an F1 score of 0.8. Such classification schemes can be tuned to obtain suitable performance/coverage characteristic for the compute power available.

We also show the overall performance for predicting all classes - not just FFT. We observe that with around 8 examples per class, top-3 accuracy is consistently above 50%. Overall, the model does not overfit to the train split, and reaches useful performance with relatively few examples. This is due to the effectiveness of the ProGraML representation, the convolutional graph inductive bias, and the class separability of the dataset, especially in the case of FFT, whose data-flow graph shows clearly distinguishable patterns. Generally, we can see that neural embeddings are effective at detecting FFTs, and also have applicability for similar acceleration-identification tasks in other domains.

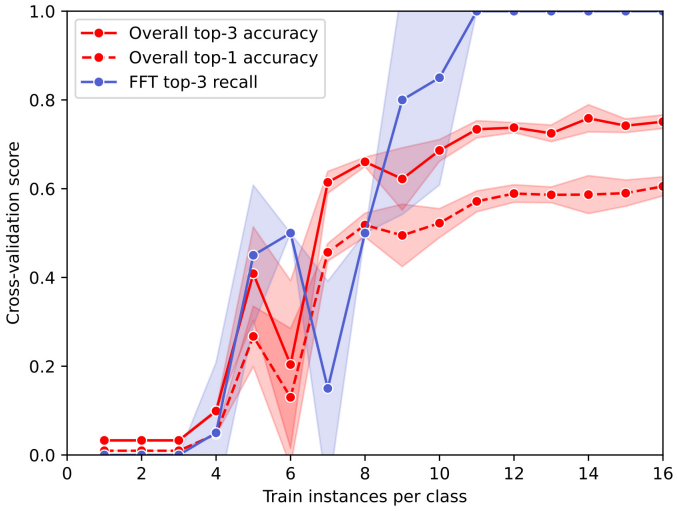


Figure 5.11: Cross-validation accuracy (mean and standard deviation) of our ProGraML-based neural classifier in terms of the number examples per class when trained using a reduced version of the OJClone dataset with FFT examples injected.

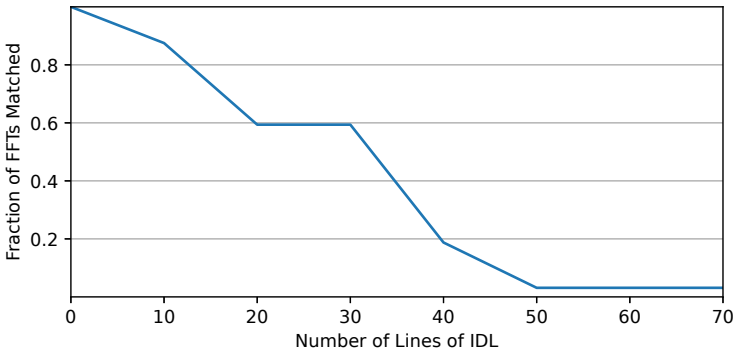


Figure 5.12: How the number of patterns matched changes with the length of the IDL pattern used. IDL patterns to match entire FFTs are thousands of lines long and do not generalize. By 50 lines we have only a single remaining match and still only cover the prologue of a single FFT function.

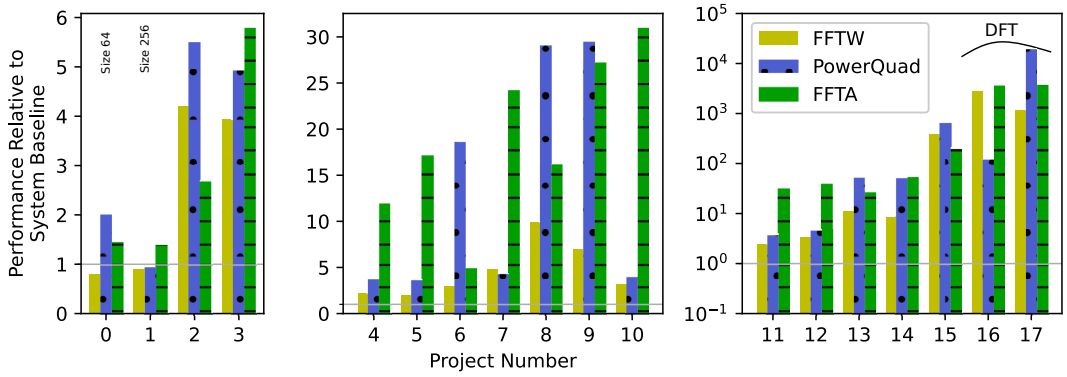


Figure 5.13: Relative performance for different FFT implementations on GitHub, comparing original software and FACC's generated accelerator call for FFTs of length 1024. FFTA results from the ADSP board are compared to software running on the Arm Cortex-A5. PowerQuad results from the NXP board are compared to software on an Arm Cortex-M33. FFTW results are compared to software on an i9-10900X desktop CPU. Geometric mean speedup is 9x for FFTW, 17x for the PowerQuad and 27x for the FFTA.

5.10.3 How Do FACC's Adapters Perform?

Figure 5.10 shows the performance of FACC on the ADSP board compared with the ProGraML Neural Classifier and IDL. FACC takes advantage of the algorithm-specific accelerator, achieving geometric mean speedups of 27x. ProGraML cannot exploit the accelerator since it is just a classifier, but achieves a 3.5x speedup by moving FFT code to the DSP. Interestingly, in one case the DSP is actually faster than the FFTA due to the small data size. IDL only detects one acceleration opportunity, achieving the same performance as FACC on benchmark 0 only.

5.10.4 How Do FACC's Adapters Perform on Different Platforms?

Figure 5.13 shows the performance improvement obtained by each implementation on the ADSP board, the NXP board and the FFTW optimized software library. Relative performance is dictated by the performance of the accelerator and the performance of the compiler used to compile the original implementation.

These differences can be seen on benchmark 8, where the original implementation is poorly optimized for the hardware on the Arm Cortex-M33, but runs much better on the Arm Cortex-A5 and Intel i9-10900X. We can also see significant differences between styles of implementations, with projects 16 and 17, which are DFTs yielding particularly large speedups (10,000x on the PowerQuad). The geometric mean speedup

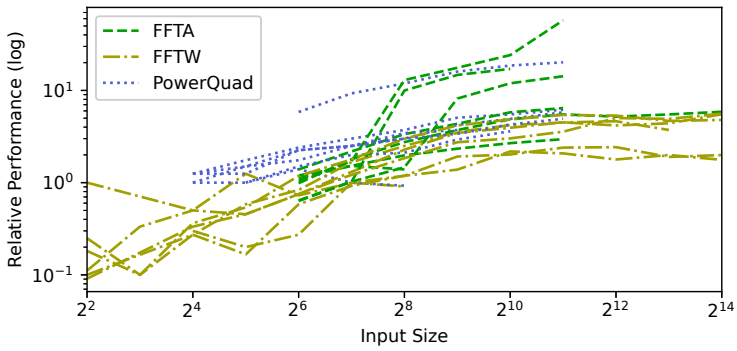


Figure 5.14: Speedup for accelerating benchmarks 1–7 on different sizes of input. Different accelerators and benchmarks have different overlap ranges, but in general, as problem size increases relative speedup increases.

for each accelerator relative to their baseline is 9x for FFTW, 17x for the PowerQuad and 27x for the FFTA.

Performance for varying sizes of input for projects 1–7 is shown in Figure 5.14. Speedups increase with data size as expected for an offloading-based accelerator model [48]. Speedups are possible using optimized software libraries, although the opportunities are more limited and may require profiling to determine viability.

5.10.5 Compilation Time

Figure 5.15 shows the compilation time taken by FACC for each benchmark. Results are gathered on a 6 core Intel i7-8700K CPU running at 3.70 GHz with 32 GB. We anticipate a number of simple parallelism-based optimizations could significantly reduce compilation time.

Figure 5.16 shows how the number of binding examples generated for each target. FFTW exposes more functionality in its interface, so requires more examples to be generated. FACC uses the same interface to access the ADSP board's FFTA and the NXP board's PowerQuad, so the number of examples is identical. The difference in compile time is due to different supported input lengths: the PowerQuad supports smaller input sizes, which are faster to test. None of these programs result in excessively large search spaces. If the search space were to grow, standard synthesis pruning techniques could be applied [84].

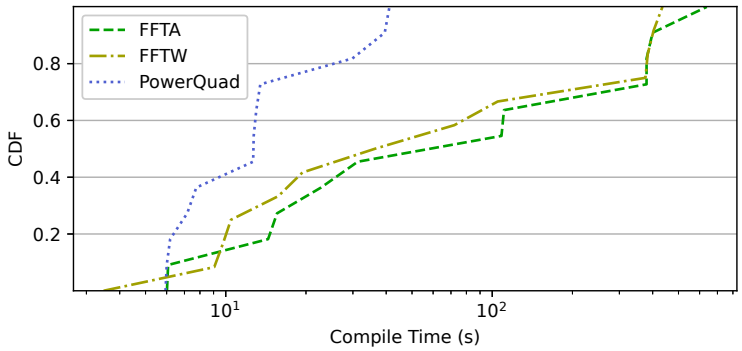


Figure 5.15: CDF of the compilation times taken by FACC for each benchmark. One distribution per target.

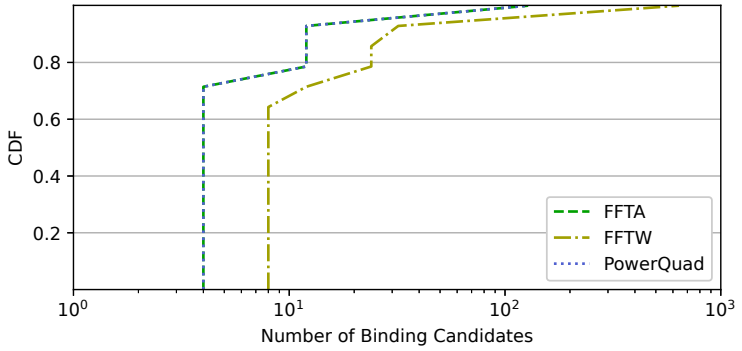


Figure 5.16: CDF of the number of candidates generated by FACC for each benchmark. One distribution per target. FFTA and PowerQuad overlap due to similarity of restrictions exposed via API from the hardware, unlike the software FFTW.

5.11 Conclusion

This chapter describes FACC, a tool for compiling user-code to Fourier-transform accelerators and optimized libraries. FACC uses IO matching and program synthesis to address the problems of code, data, domain and behavioral mismatch, allowing for easy accelerator integration into existing source code. Using FACC and real-world optimized libraries and hardware accelerators, we are able to achieve speedups averaging 9x for FFTW, 17x for the PowerQuad and 27x for the Analog Devices FFTA.

While FACC focusses on matching user code to acceleration APIs, it can also be used to match optimized libraries to emerging hardware e.g matching FFTW to FFTA. This would allow users, who have already restructured their application to use libraries, to continue to benefit from hardware evolution, while automatically handling the unusual constraints that fixed-function hardware poses.

Although, this chapter focusses on Fourier-transforms, this approach is readily applicable to other fixed-function accelerators. Fixed-function need not be the enemy of programmability and automatic targeting. Rather, we can automatically rearchitect software to build adapters that bend the accelerator to the user's will rather than vice versa.

FACC demonstrates a solution to the acceleration equation introduced in Chapter 2, using the generated adaptors to find solutions for g, h that enable the accelerator to be used across a wide diversity of code and algorithms.

Chapter 6

Rewriting for Domain-Specific CGRAs

Coarse-grained reconfigurable arrays (CGRAs) are domain-specific devices promising both the flexibility of FPGAs and the performance of ASICs. However, with restricted domains comes a danger: designing chips that cannot accelerate enough current and future software to justify the hardware cost.

We introduce *FlexC*, the first flexible CGRA compiler, which allows CGRAs to be adapted to operations they do not natively support. FlexC uses dataflow rewriting, replacing unsupported regions of code with equivalent operations that are supported by the CGRA. We use equality saturation, a technique enabling efficient exploration of a large space of rewrite rules, to effectively search through the program-space for supported programs. FlexC sets the groundwork for future applications of the acceleration equation to accelerators like CGRAs: rather than dealing with finding f and g directly, this work focuses on getting a program rewritten so it can run.

We applied FlexC to over 2,000 loop kernels and demonstrate a 2.2× increase in the number of loop kernels accelerated leading to 3× speedup on kernels that would otherwise be unsupported.

6.1 Introduction

Specialized hardware has demonstrated truly significant performance gains over general-purpose processors [184], yet despite its potential [29, 474], it faces real challenges to wider adoption [143]. The fundamental reason is that programming such accelerators is difficult [134], often requiring modification of the underlying algorithms [143]. It is user reluctance to do this [434] that brings frequency-of-use [88, 208, 371] and cost [264] concerns.

Heterogeneous CGRAs [314] (Coarse-Grained Reconfigurable Architectures) are a class of architectures that promise to solve this problem [371]. CGRAs can achieve near-ASIC level performance [306] and provide enough flexibility to run a wider class of code [371]. These heterogeneous CGRAs use processing elements specialized to various degrees [42]. While this makes them more efficient [66, 164, 486], specializing hardware introduces limitations on the software being compiled to it [224, 530, 531].

So despite aiming at flexibility, such CGRAs are hard to use beyond the scope they were designed for. They age poorly as software evolves [131] and falls out of the scope of the narrowly designed hardware: the *domain-restriction problem*.

This problem is highlighted by existing state-of-the-art CGRA compilers such as OpenCGRA [468] which frequently fail due to hardware specialization. If code contains an operation that is unsupported by a particular hardware, existing techniques simply cannot accelerate it, restricting CGRAs to a narrow software domain. This domain-restriction poses a significant challenge and is not well understood [522]. What we need is a new approach that *automatically transforms* user programs to fit heterogeneous CGRAs expanding the domain of supported software without user effort.

We introduce *FlexC*, the first *flexible* CGRA compiler that addresses the domain-restriction problem. FlexC uses a set of rewrite rules that translate operations that are unsupported into those that are supported. This compilation strategy requires a non-trivial application of rewrite rules in an attempt to find a valid transformation to an expression the CGRA can support, leading to a large search space. To explore this space efficiently, FlexC uses equality saturation [472, 524]. While a powerful technique, CGRA compilation presents a number of unique challenges to equality saturation including, crucially, transformation encoding and cost modeling. Overcoming these challenges allows us to simultaneously represent many equivalent programs, enabling efficient large space exploration.

We extract over 2,000 loop kernels from projects in domains including multimedia code and compression libraries and demonstrate that FlexC increases the code that existing proposed accelerators can support by a factor of 2.2× over existing compilers. We show that rewriting loops to run on CGRAs produces speedups averaging 3×, and that FlexC is applicable to a wide range of possible CGRA designs.

In summary, we contribute:

- FlexC, the first *flexible* CGRA toolchain designed to support operationally-specialized CGRAs.

- Four sets of rewrite rules for the purpose of *rewrite exploration*, which enables effective translation of code to run on CGRAs designed for different purposes.
- The first large-scale benchmark suite for CGRA compilers, with more than 2,000 loops from five different projects[1].
- An evaluation of these tools, demonstrating the importance of non-linear exploration techniques like equality saturation in finding working compilation sequences for real-world heterogeneous CGRAs.

6.1.1 Connection to the Accelerator Equation

This chapter sets the groundwork for more complete applications of the accelerator application to reconfigurable accelerators. It does not focus on generating f and g, but rather focuses on providing a rewriting system to find the best function A when there are many configurations to choose from. With this groundwork completed, future work can explore the generation of f and g in the cases of reconfigurable accelerators.

6.2 Motivation

CGRAs promise near-ASIC performance while retaining the reconfigurability of the interconnect of a fixed set of processing elements [152]. In this chapter, we demonstrate with FlexC that compiler technology can be used to overcome the domain-restriction problem of CGRAs and dramatically increase the amount of software that can be accelerated.

6.2.1 Coarse-Grained Reconfigurable Arrays

Coarse-Grained Reconfigurable Arrays (CGRAs) are a hardware architecture with a fixed number of processing elements (PEs) and a reconfigurable interconnect between those PEs. Traditional CGRAs have been homogeneous, with PEs supporting all operations arranged in a grid [339]. More recently, heterogeneous CGRAs have been a significant research topic, with specialized PEs and a runtime-reconfigurable interconnect [66]. Figure 6.1 shows some example CGRAs. CGRA architectures exploit loop-level parallelism in two ways: within a loop by scheduling operations in parallel, and between loop iterations via loop pipelining [339].

[1] Available at `https://github.com/j-c-w/LoopBenchmarks`

While CGRAs are less flexible than more fine-grained reconfigurable architectures such as FPGAs, they offer significant flexibility benefits over ASIC accelerators — and can often run unmodified C code [468]. Compared to FPGAs, they offer significantly faster reconfiguration times [181] and better power efficiency, but their specialized PEs support a narrower set of software, often restricted to a specific domain.

6.2.1.1 Heterogeneous CGRAs

Heterogeneous CGRAs promise to achieve better power efficiency and lower area utilization than their heterogeneous counterparts [66]. The innovations in these heterogeneous CGRAs have been critical in delivering the latest generation of low-power CGRAs, such as Snafu [192]. However, introducing this heterogeneity introduces significant compilation challenges.

6.2.2 The Software Domain-Restriction Problem

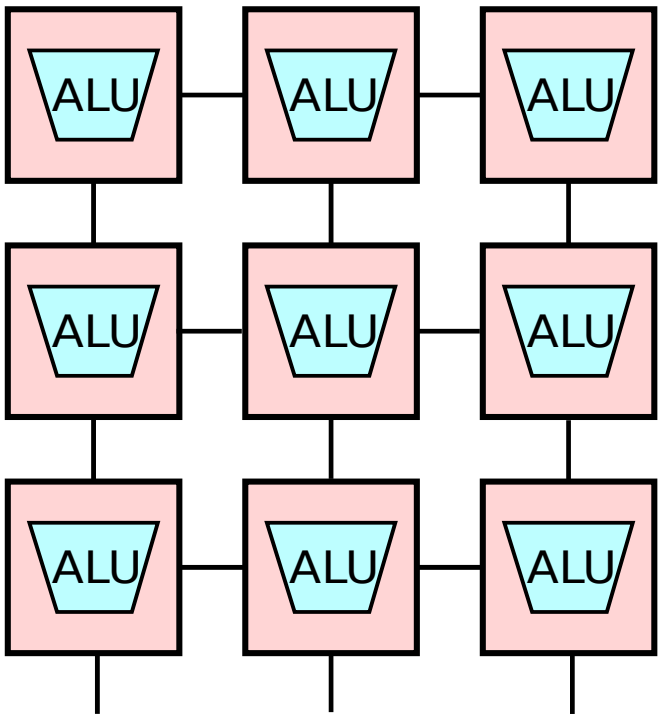
The nature of hardware specialization is that some programs are excluded. This can greatly increase the performance of the accelerator, and decrease the resources required — both to design it and to run it.

However, the cost of the specialized hardware has to be justified by frequent enough use [371] and enough demand for software benefiting from the hardware acceleration [264]. Software also evolves in various ways [324], and small changes to code can render narrow and restrictive accelerators useless, as shown in Figure 6.2. Here, the user has deleted the addition operator and replaced it with a subtract. If the CGRA does not support subtraction, then the code can no longer be executed on the CGRA accelerator shown in Figure 6.3.

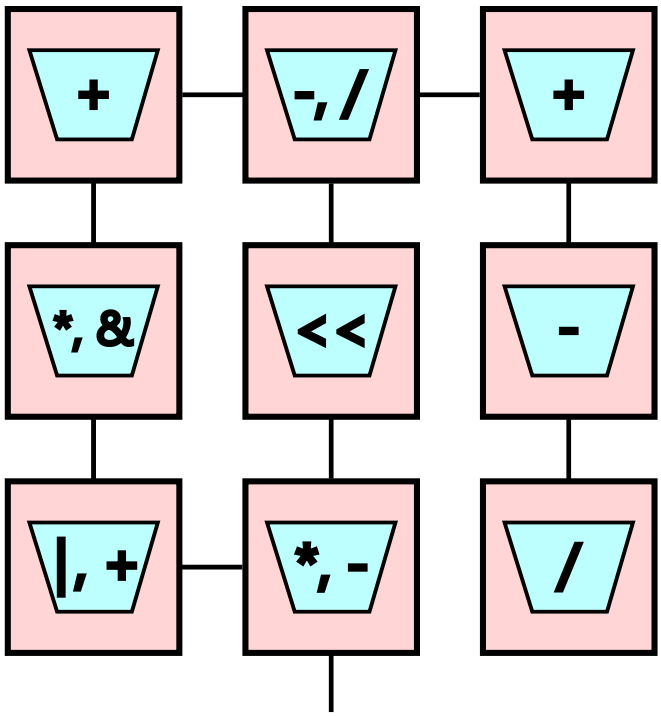
We coin this the *domain-restriction problem*, where specialized hardware faces the danger of supporting too narrow a domain, greatly restricting the supported software. This results in a hesitation to adopt specialized hardware that could otherwise greatly increase performance and efficiency [264].

6.2.3 Overcoming the Domain-Restriction Problem with FlexC

To overcome the domain-restriction problem, we need to broaden the software supported by specialized hardware, by *transforming* software that uses operations not natively supported by the hardware.



(a) Homogeneous CGRA



(b) Heterogeneous CGRA

Figure 6.1: Homogeneous CGRAs, such as ADRES [339], connect elements uniformly, and have the same functional units within each element. Heterogeneous CGRAs, such as REVAMP [66], specialize both the computation units and datapaths, to remove wiring and logic when it is unneeded for a proposed target use case. This increases efficiency, but limits applicability when use cases change.

```
    for (...)
-       x[i] = x[i] + y;
+       x[i] = x[i] - y;
```

Figure 6.2: An example bug-fix. If the accelerator does not support `-` operations, then the post-bug-fix loop cannot be accelerated. (For example the toy CGRA shown in Figure 6.3.)

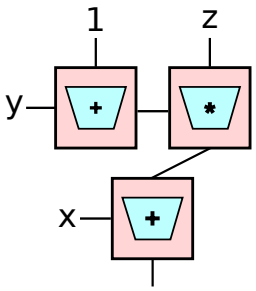
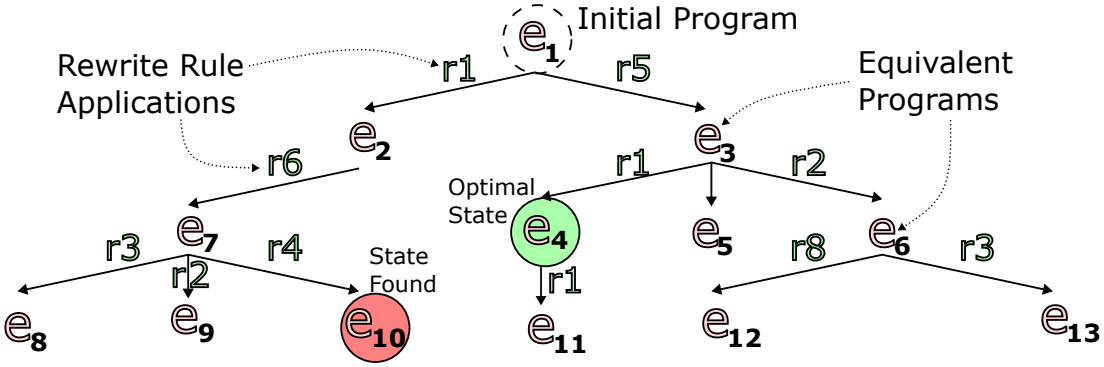


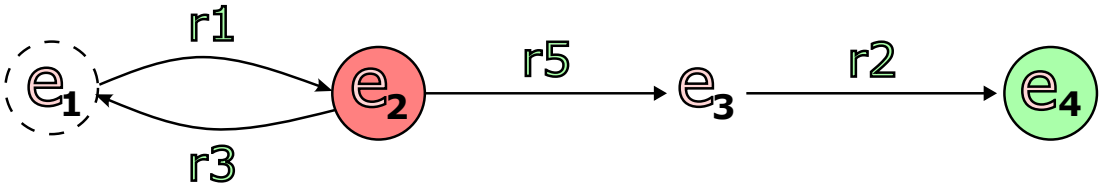
Figure 6.3: Example of a domain-specific CGRA supporting expressions like `x + ((y + 1) * z)` [66]. To support the expression `a - b - 1`, we can apply rewrite rules to transform it to the equivalent `a + ((b + 1) * -1)`.

As a simple example, Figure 6.3 shows a CGRA designed to accelerate the expression `x+((y+1)*z)`. It is incapable of natively accelerating the expression `a-b-1`, as the `-` operation is not supported. By rewriting `a-b-1` as `a+((b+1)*-1)`, the CGRA can support this new expression. Despite replacing two operators with three, and thus making it more complex, supporting this expression on a CGRA more than makes up for this in terms of power and performance. This is because on general-purpose processors, stages such as instruction fetch and decode typically require an order of magnitude more power than compute [142].

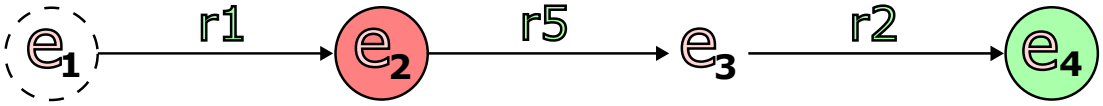
FlexC automatically adapts the software, replacing unsupported operations via dataflow rewriting. FlexC uses equality saturation, a recently popularized rewrite technique, that overcomes challenges with traditional canonicalization and greedy techniques. To motivate our choice of rewrite technique, we describe the limitations of traditional compiler techniques used for rewriting, and we discuss the considerations faced when applying equality saturation in this new domain.



(a) In the *exploration problem*, the expression in green is the optimal choice for this CGRA, but may never be reached in a greedy application of rewrite rules, which will reach the red state instead.



(b) In the *loop problem*, A greedy rewriter may get stuck in a loop due to cyclical groups of rules, preventing it from finding the optimal state.



(c) In the *cost-trap problem*, A greedy rewriter can get stuck in state e_2 as e_3 is a less valuable state.

Figure 6.4: Applying rewrite rules with a greedy rewriter results in dead-ends that equality saturation avoids.

6.2.3.1 The Limits of Canonicalization

Compiler frameworks such as LLVM, GCC and MLIR, use canonicalization passes to transform IRs into a predictable and efficient form that compiler transformations can assume simplifying their development. Canonicalization is implemented with a set of simple rewrite rules that are applied greedily. To make canonicalization feasible, rules that would allow exploring more optimization opportunities such as commutativity (flipping the order of operands) are prohibited: these do not work towards the goal of transforming the IR into a canonical form. Further, in heterogeneous CGRAs, canonicalizing does not solve the domain-restriction problem, as the rewritten expressions may not be supported.

6.2.3.2 The Limits of Greedy Dataflow Rewriting

Greedy rewriting can also be used more broadly beyond canonicalization. Figure 6.4 highlights three problems that can cause greedy rewriters to get stuck. In Figure 6.4(a), greedily applying the first available rule to expression e_1 leads to the resulting expression e_{10} which is less performant than the optimal expression e_4. In Figure 6.4(b), greedy rewriting leads to a cycle between e_1 to e_3, never reaching the solution e_4. Finally, in Figure 6.4(c), the greedy rewriter gets stuck in a local minimum due to the cost of applying further local rewrites.

In summary, greedy rewriting works well for simple rewrite problems but quickly faces limitations. We, therefore, explore the use of a more complex rewrite technique. Our results confirm that equality saturation enables FlexC to compile more software to the CGRA.

6.2.3.3 Benefits of Equality Saturation for CGRAs

Traditional rewrite techniques must decide on an order in which to apply the rules. In contrast, equality saturation [472] can apply rewrite rules all at once using an e-graph data structure.

This is important when compiling to domain-specific hardware, as the order in which rules should be applied differs between hardware targets. Figure 6.5 shows an example from the FFMpeg [7] library, part of our benchmark suite. Equality saturation is used to rewrite the top program, which does not fit on the CCA-like accelerator adapted from [516] because it contains multiplications and subtractions. Equality saturation avoids the *cost-trap problem* by not committing to specific rewrites too early,

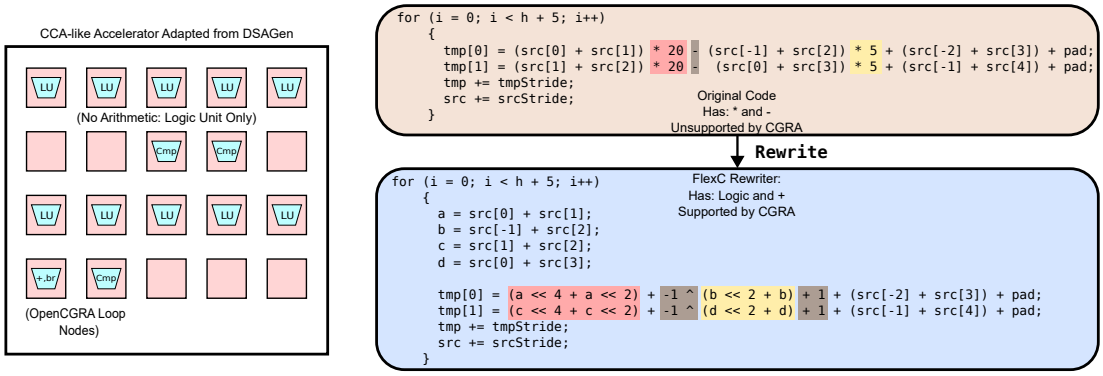


Figure 6.5: An example from the FFMpeg [7] library, which is part of our benchmark suite. FlexC rewrites the loop to run within the context of the CCA-like accelerator adapted from DSAGen [516]. Equality saturation is critical in this example to enable the conversion of `a - b` into `a + -1 ^ b + 1`, as the rewriter must traverse the `a + (-b)` state, which is no better than `a - b`. This is an example of the cost-trap problem (Figure 6.4c).

and the *looping problem* because rewrites grow a set of equivalent programs instead of transitioning between programs.

6.2.3.4 Challenges posed by CGRAs for Equality Saturation

Equality saturation is used in many compiler optimization problems [359, 488, 509, 552]. However, it has not been studied in the context of CGRAs, which pose a number of key challenges we tackle in this work. These include:

- What IR encoding to use? (Section 6.4)
- What cost model to use? (Section 6.4.1)
- When can we avoid the costs of equality saturation altogether? (Section 6.4.4)
- Which rewrite rules to use? (Section 6.5)

Effectively resolving these challenges determines the efficacy of any compiler solving the domain-restriction problem. We evaluate our decisions in Section 6.7.

6.3 System Overview

FlexC is implemented in OpenCGRA [468], a CGRA compiler intended to target heterogeneous CGRAs. Given an input DFG, FlexC explores sequences of rewrites that enable the DFG to be compiled to a specialized architecture that does not support all the operations within the code. After rewriting the DFG, FlexC uses OpenCGRA to

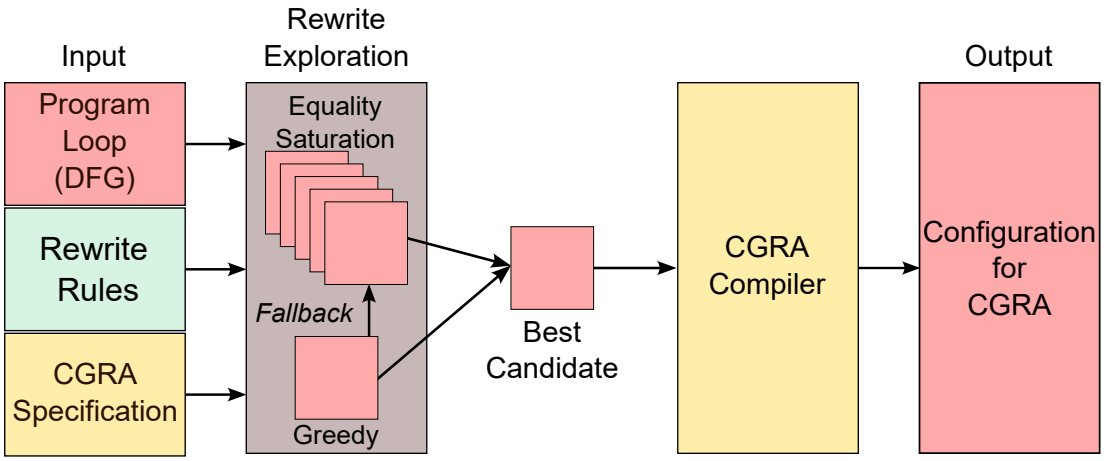


Figure 6.6: FlexC system overview. A Data-Flow Graph (DFG), set of Rewrite Rules and a CGRA specification are input. FlexC first applies a hybrid-rewrite strategy and selects the most suitable candidate to pass to the CGRA compiler, which generates the configuration.

target the hardware.

Figure 6.6 shows how FlexC compiles software for a CGRA. In a traditional CGRA compiler, a Data-Flow Graph (DFG) is used to generate a CGRA configuration. If the DFG does not match the target CGRA precisely, the code generation fails.

FlexC adds a rewrite system, using a set of rewrite rules dictated by the context and a cost function based on the target CGRA. After selecting the optimal graph according to the cost model (the most likely DFG to be compilable to the underlying CGRA), FlexC uses a traditional CGRA compiler to generate the final mapping.

FlexC can be applied in conjunction with any CGRA compiler — provided that appropriate rulesets using the right instructions can be supplied. We provide FlexC under a liberal license to allow this[2].

6.4 Graph Rewriting

FlexC translates programs to domain-specific CGRAs by generating a large set of equivalent code loops, in the aim of finding a suitable match should one exist. We first formally define our inputs: a data flow graph representing a loop, a set of rewrite rules to transform the program, and a CGRA specification to target, before describing our

[2]Available at `https://github.com/j-c-w/LoopBenchmarks`

rewriting strategy. We then define our rewrite algorithms in pseudocode. We start by attempting greedy rewriting in the hope of generating a successful result quickly (we will see in Section 6.7 that this succeeds about 40% of the time). If this fails to produce a suitable match for the accelerator, when then use equality saturation to explore the full space of rewrites.

Definition 1 *A* data flow graph *D is a finite set of nodes N corresponding to operations $op(n_1,...,n_m)$, where op is an operation symbol and $n_i \in N$ are children operands.*

D must be a directed acyclic graph*, meaning that a function $id : N \to |N|$ should exist such that:*

$$\forall n = op(n_1,...,n_m) \in N.\ \forall i.\ id(n_i) < id(n)$$

While OpenCGRA uses Control Data Flow Graphs (CDFGs), and thus can handle branches and loops, we do not attempt to rewrite across control-flow boundaries. Instead, we break all control flow before rewriting, and restore control flow after rewriting.

Definition 2 *A* rewrite rule *R is of the form $l \Rightarrow r$, where l and r are patterns. A* pattern *P is a tuple (N_P, O_P), where N_P is a data flow graph that may contain variable nodes on top of operation nodes, and $O_P \subseteq N_P$ is a list of output nodes.*

R can be applied to D when l has a match (M_l, σ) in D, where M_l is a list of nodes from D matching O_l, and σ maps variables to matching nodes from D. To produce the list of nodes M_r that should replace the M_l nodes in D, the variables are substituted in r, written as $r[\sigma] = (N'_r, M_r)$.

A rewrite rule must be semantics-preserving*, meaning that $\forall(M_l, \sigma).\ M_l = M_r$ which depends on the element-wise application of a given semantic equality. The meaning of equality in this case depends on the rules provided. We will see in Section 6.5 that this may be true equality, fuzzy equality (e.g. with floating-point manipulation rules) or even weaker definitions of equality (e.g. with stochastic computing rules 6.5.4).*

Definition 3 *In a CGRA, we have an array of processing elements, PEs (PE_i), each of which supports a particular set of operations ($op(n_1,...,n_m)$), $Supported_i$. We generate this set from the CGRA's specification.*

Given a particular DFG D, with nodes N, there may be some subset of nodes $Unsupported(N)$ that have operations without hardware support anywhere *on the CGRA. We wish to find a sequences of rewrite rules that we can apply to the DFG to produce D' with nodes N' such that $Unsupported(N') = \{\}$, as otherwise it will be*

impossible to schedule that particular code onto the CGRA. We thus define the set of operations a particular full CGRA can support as:

$$ops = \bigcup_i Supported_i$$

,

6.4.1 Rewriting Goal

The compiler takes a dataflow graph (DFG) D as input. Numerous existing techniques attempt to find a valid mapping [329], but in heterogeneous CGRAs, the operations in the DFG may not be in the supported set for *any* node.

The goal of a rewriting algorithm $A(D, Rs, ops)$ is to return D', obtained by rewriting D using the set of rules Rs, such that D' only uses operation symbols from ops.

We further define a cost function $C(D, ops)$ to minimize:

$$\sum_{op(...) \in N} 1 \text{ if } op \in ops \text{ else } 10^6$$

This incentivizes smaller programs by giving a cost of 1 to available operations, while giving a huge penalty to unavailable operations by giving them a cost of 10^6. Our CGRA specification and cost function aim to eliminate unavailable operations to successfully map the program onto the CGRA, without trying to precisely model the execution performance.

With the assumption that $|N| < 10^6$, rewriting successfully eliminates all unavailable operations if $C(D', ops) < 10^6$, and fails to do so if $C(D', ops) \geq 10^6$.

6.4.2 Greedy Rewriting

Listing 6.1 shows our greedy rewriter. Greedy rewriting is the most straightforward rewriting approach; it runs quickly but often gets stuck in local minima.

On each greedy iteration, we iterate over every rewrite rule to find matches (lines 6 to 8). If applying a rewrite for a given match leads to a cost reduction, we proceed with the rewritten program and forget about the previous program (lines 9 to 11). The local_minima variable keeps track of whether a fixed point was reached, which is the termination condition (line 3).

Listing 6.1: Greedy rewriting algorithm

```
1  def greedy(d, rs, ops):
2   local_minima = false
3   while not local_minima:
4    local_minima = true
5
6    for r in rs:
7     matches = find_matches(d, r)
8     for m in matches:
9      d2 = apply_match(d, m)
10     if C(d2, ops) < C(d, ops):
11       d = d2
12       local_minima = false
13       break
14
15  return d
```

6.4.3 Equality Saturation

Listing 6.2 shows our algorithm for rewriting via equality saturation. Equality saturation [472] is a more sophisticated rewriting approach; it avoids getting stuck in local minima but can be slow to execute. We leverage both the state-of-the-art Rust `egg` library [524] and existing work extending equality saturation to graph rewriting [552].

First, we initialize an e-graph data structure that compactly represents a space of equivalent programs by sharing equivalent sub-terms as much as possible (line 2). Then, we run the explorative phase of equality saturation using our set of rewrite rules, iteratively exploring possible rewrites in a breadth-first manner and growing the e-graph (line 3).

As visible in line 10, the explorative phase terminates when all possible rewrites have been explored (a fixed point, called saturation, is reached), or when another stopping criteria is reached (e.g. a timeout). On each explorative iteration, all rewrite-rule matches are collected (line 13) and applied in a non-destructive way, adding new equalities into the e-graph (line 15).

Finally, we extract the best program from the e-graph according to our cost function

Listing 6.2: Equality saturation algorithm

```
1  def eqsat(d, rs, ops):
2   egraph = initialize_egraph(d)
3   egg_exploration(egraph, rs)
4   return egg_lp_extraction(
5    egraph,
6    cost_for_egg(ops))
7
8  def egg_exploration(eg, rs):
9   ...
10  while not saturation_or_timeout:
11   matches = []
12   for r in rs:
13    matches += find_matches(eg, r)
14   for m in matches:
15    apply_match(eg, m)
16   ...
```

using Linear Programming (line 4). Despite our problem being amenable the bottom-up extractor, this extractor is buggy and performs poorly on this problem.

6.4.4 Hybrid Rewriting

FlexC uses hybrid rewriting (listing 6.3), which takes the best from both strategies. In hybrid rewriting, we first apply a fast greedy rewriter. If the greedy rewriter does not find a suitable candidate, FlexC falls back to the more expensive, but more likely to succeed, equality saturation.

6.5 Rewrite Rules

We explore several different rulesets in this work: some rules are always correct, while other rulesets may only be useful in certain domains, such as the stochastic-computing rewrite rules (section 6.5.4).

In a traditional application of rewrite rules, compilers look to perform strength

Listing 6.3: Hybrid rewriting algorithm

```
1 def hybrid(d, rs, ops):
2   d2 = greedy(d, rs, ops)
3   if cost(d2, ops) < 10^6:
4     return d2
5   else:
6     eqsat(d, rs, ops)
```

reduction [133], by replacing more complex expressions with simpler expressions — this is typically achieved by canonicalizing towards the simplest method of representing an expression. In a traditional compiler, a rule is typically formatted as:

$$\text{Complex Operation} \rightarrow \text{Simpler Operation}$$

A typical rewrite rule system produces a series of independent rewrites,

$$e_1 \underset{\text{rule}1}{\Longrightarrow} \cdots \underset{\text{rule}N-1}{\Longrightarrow} e_N$$

to generate the expression best suited to the architecture. In this case, the rules can be written in such a way that they chain together, as they are in existing compilers. We simply stop rewriting when there are no more rules to apply.

However, when compiling for a CGRA, replacing simpler operations with *more complex* operations can be beneficial if they enable an entire region of code to be run on faster, fixed-function hardware.

As a result, for some sequence of rules

$$e_1 \underset{\text{rule}1}{\Longrightarrow} \cdots \underset{\text{rule}i-1}{\Longrightarrow} e_i \underset{\text{rule}i}{\Longrightarrow} \cdots \underset{\text{rule}N-1}{\Longrightarrow} e_N$$

some intermediate e_i may be the best choice of expression, and further, rule application can occur bidirectionally. Rather than strength reduction, which implies a linear sequence of operations that become strictly simpler, the process for compiling for a CGRA is instead *rewrite exploration*.

6.5.1 Integer Rules

We use a set of strength-reduction and canonicalization rules representative of those in a typical compiler. An example is:

```
x - y <=> x + (-y)
x >> y <=> x / (1 << y)
x and y <=> not ((not x) or (not y))
```

Table 6.1: Some example rewrite rules that can be used to change the operations an expression requires.

```
x * y <=> x / (1.0 / y)
-1.0 * x <=> -x
```

Table 6.2: Rewrite rules enabled by reducing requirements on floating-point equality.

$$x * -1 \Rightarrow -x$$

On the left-hand side of this rule, we require a multiplication operation, and on the right-hand side of this rule, we require a negation operation. For most compilers, the right-hand side is (almost) always the better choice, so most rewriters only apply these rules forward.

FlexC applies this rule in both directions, as some CGRAs may have multiplication-supporting PEs and other CGRAs may have negation-supporting PEs. We refer to this universally applicable ruleset as the *integer ruleset*. Some examples from the set are shown in Table 6.1.

6.5.2 Floating-Point Rules

Floating-point rewrite rules are rarely bit-for-bit correct. Compilers typically use various flags to allow for different levels of correctness guarantees, enabling floating-point transformations only when the programmer is willing to forgo accuracy.

When compiling floating-point operations to CGRAs, FlexC uses these rules by default (they can be turned off). This enables more rewrites at the cost of losing bit-correctness. An example of rules enabled by this assumption are shown in Table 6.2.

6.5.3 Boolean Logical Operations

Logical operations such as `AND` (&) and `OR` (|) can take two different meanings within programs: sometimes they are used to specify bitwise operations on entire words at a time, and sometimes they are used as boolean operators (where any result other than

```
x AND y => x * y
x OR y => (x + y) > 0
x XOR y => x != y
```

Table 6.3: Rewrite rules under the assumption that binary logical operators are boolean operators.

```
x * y => x AND y
```

Table 6.4: Example rewrite rule for stochastic computing. The SC-CGRA only provides a stochastic multiplier, so the stochastic adder rule is not used.

a `0` is `true`). With a compiler flag provided by a programmer to indicate these are equivalent, we can add more rewrite rules.

For example, as boolean operations, `AND` can be rewritten using multiplication nodes, increasing the space of programs that a CGRA without logical operator support can be used for. We supply a set of rewrite operations that assume logical operations are equivalent to boolean operations. Some examples of rewrite rules in this set are shown in Table 6.3.

6.5.4 Stochastic Computing

Stochastic computing is a computing paradigm aimed at achieving better energy efficiency than traditional computing by trading off accuracy [41]. In particular, stochastic computing allows multipliers to be replaced by logical `and` operators, and `add` operations to be replaced by muxes [63]. Table 6.4 shows an example of these rules.

These rules are used as an example of rules for specific architectures that can be combined with existing rewrite rules. In this particular case, many of the rewritten kernels must be checked by the programmer to ensure that the operations that are stochastic are suitable.

6.6 Results

We implement FlexC above OpenCGRA, which is written in C++. We use the `egg` Rust library [524] to implement our rewriters. For equality saturation, we use an iteration limit of 10 with a node limit of 100,000 to prevent the e-graphs from growing too large.

Domain	Project	Samples
Compression	Bzip2 [4]	13
Multimedia	FFmpeg [7]	1852
	FreeImage [8]	223
Scientific Computing	DarkNet [6]	77
	LivermoreC [9]	26
Sum		2188

Table 6.5: Quantities of unique loops in benchmark suite.

FlexC is integrated into the LLVM framework and is invoked using the `opt` tool using LLVM IR as input.

FlexC relies on OpenCGRA to find the loop to accelerate. OpenCGRA looks for the first loop in each provided function. We implement the architecture specification in JSON, adding a mapping from each PE to the sets of operations it will be able to support.

6.6.1 Benchmarks

We have collected a benchmark suite of real-world open-source code loops. As shown in Table 6.5, we compose it of projects from the domains of multimedia, compression, and simulation and extract a total of 2188 loops. Typically, CGRA compilers are evaluated on benchmark suites of a few tens of loops. However, such small benchmark suites do not capture the wide spectrum of loops that programmers write, and are easy to hand optimize [11]. Our benchmark suite captures a wide range of loops, without the overheads of running whole programs [27]. These loops allow us demonstrate FlexC works on a wide range of architectures and programs.

We extract loops suitable for CGRA scheduling from the projects shown in Table 6.5. Each extracted loop is:

- The innermost loop
- Has no internal branches or function calls (unless inlined)
- Contains at least one array access

These properties have been selected to make our benchmarks applicable to many different CGRAs (with or without heterogeneous restriction) and different compilation

techniques.

We build a custom Clang-based tool that identifies loop structures and detects required type definitions. Clang is run using the build-system rules for each project. Each loop is placed into a function skeleton so that it can be compiled by CGRA compilers.

6.6.2 Alternative approaches

We compare FlexC against three alternative approaches: OpenCGRA [468], the LLVM [292] Rewriter and our own Greedy Rewriter. OpenCGRA is the default scheme that simply maps operations to function units without any rewriting. The LLVM rewriter employs the rewrite rules within the LLVM compiler infrastructure (using the `-inst-combine` and `-aggressive-instcombine` passes) to search through a space of equivalent programs. The greedy rewriter is FlexC without the equality saturation fallback: greedy search over the FlexC rewrite ruleset using hardware-aware rule selection.

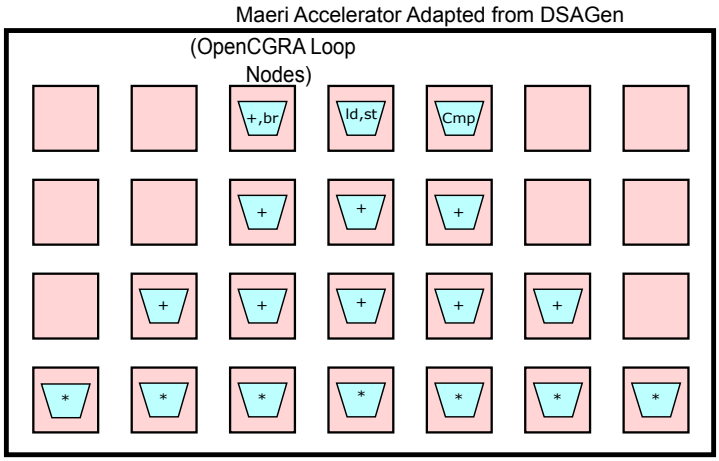
6.6.3 Existing Domain-Specific Accelerators

We evaluate domain-specific architectures from three prior works. We consider one domain-specific CGRA work (REVAMP [66]), one more general domain-specific *accelerator* work (DSAGen [516]), and one stochastic computing CGRA (SC-CGRA [508]).

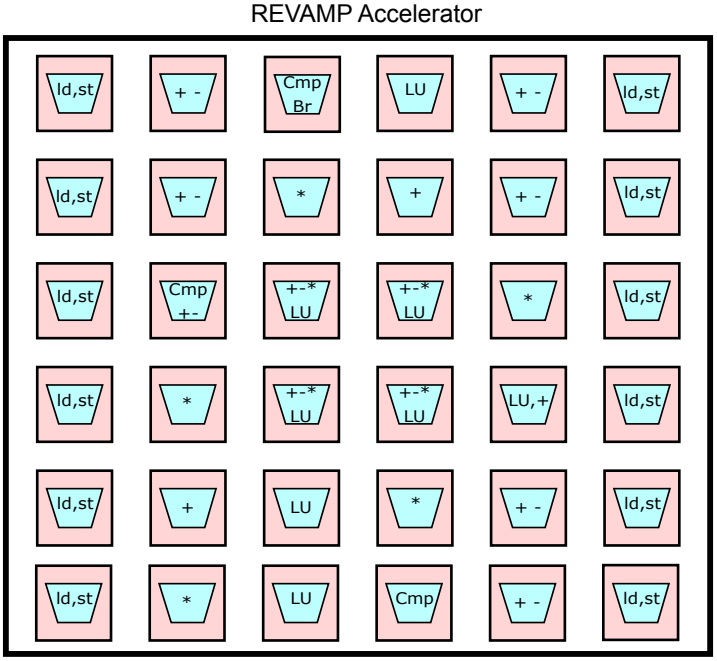
6.6.3.1 DSAGen

DSAGen [516] is a framework for generating domain-specific architectures. These architectures share many properties with CGRAs in that they expose architectural details to the compiler and present coarse-grained reconfigurable blocks. We make minor modifications[3] to the architectures shown in Figure 4(b) and 4(c) in [516] so they can be represented within OpenCGRA. The architectures we use are shown in figure 6.3 and figure 6.7a.

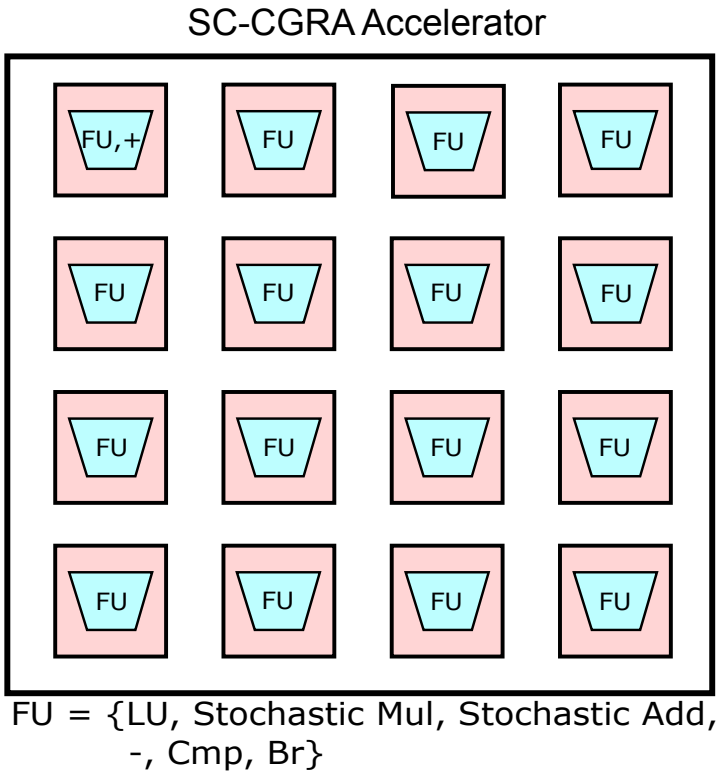
[3]OpenCGRA requires more routing to be present between compute elements, so the architectures we use are more flexible than those presented in DSAGen. OpenCGRA also does not support architectural features like distribution trees, which we have omitted. We further add the nodes required by OpenCGRA to support loop pipelining (an `add` and an integer compare) to enable it to compile to these architectures.



(a) Architecture used for Maeri CGRA.



(b) Architecture used for REVAMP CGRA.



(c) Architecture used for SC-CGRA. One traditional adder is added for loop-index variables.

Figure 6.7: Diagrams of the case study accelerators. The CCA-like accelerator is shown in Figure 6.3.

6.6.3.2 REVAMP

REVAMP [66], a framework for generating domain-specific CGRAs provides an example of a CGRA for heterogeneous compute optimization, with nodes for addition, subtraction, multiplication and some logic operations implemented within a 6x6 CGRA. A diagram of this CGRA is shown in figure 6.7b.

6.6.3.3 SC-CGRA

SC-CGRA [508] is a stochastic-computing-based CGRA. Typical exact multipliers are replaced with approximate multipliers, and similarly for adders within a 4x4 CGRA. We implement this in OpenCGRA, providing approximate adders/multiplers instead of exact ones[4]. The architecture we use is shown in figure 6.7c.

6.7 Results

We evaluate FlexC against traditional heterogeneous-CGRA compilers, improving the number of benchmarks that can be compiled to heterogeneous CGRAs by 2.2×, and demonstrate that despite making the computation more complex, the rewrite rules do not introduce slow-downs, showing geomean speedups of 3×.

6.7.1 Existing Domain-Specific Accelerators: Compilation Rate

We apply FlexC to the four accelerators presented in Section 6.6.3, comparing to three other rewriting strategies. Figure 6.8 shows that FlexC increases the number of loops that these CGRAs can support by a factor of 2.2×. Figure 6.9 gives details split by benchmark suite for each accelerator.

6.7.1.1 DSAGen

Figure 6.8 shows that using FlexC increases the number of loops that can be supported on the CCA and Maeri architectures by a factor of 2.2× and 1.6× respectively. Maeri does particularly well on LivermoreC (figure 6.9), especially once equality saturation is

[4]The authors discuss different accuracies of adder/multiplier, but do not state the number of each used, so we use a simple assignment of one multiplier and one adder per node. We also omit node-fusing, as we use OpenCGRA to target this accelerator. The operators other than the multipliers and adders are not specified completely. For this evaluation, we assume each node has logical operations, and arithmetic operations simpler than multiplication. To enable OpenCGRA to compile some things on it's own, we add one exact adder, which is required for induction variables in almost all loops.

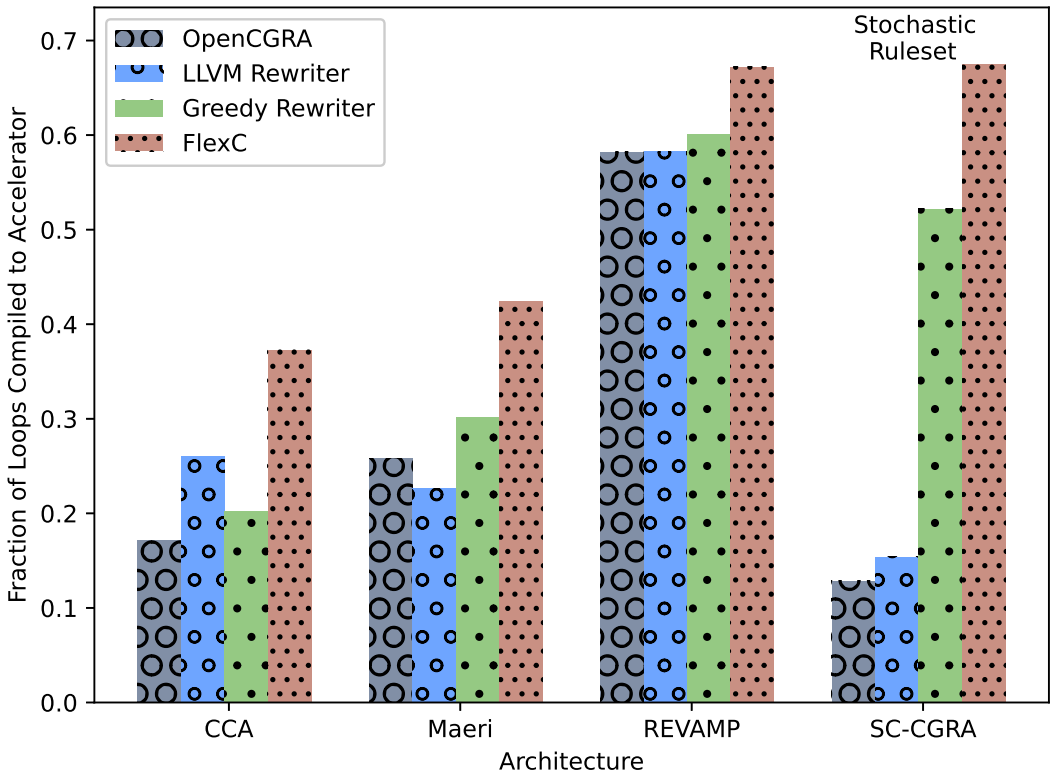


Figure 6.8: We consider four different architectures, adapted from DSAGen [516] (CCA and Maeri), REVAMP [66] and SC-CGRA [508]. All archtectures use the integer and floating-point rulesets, and SC-CGRA uses the stochastic ruleset. These architectures are specialized to different degrees: the more specialized architectures, CCA and Maeri, benefit from FlexC more than the more generic architecture from REVAMP.

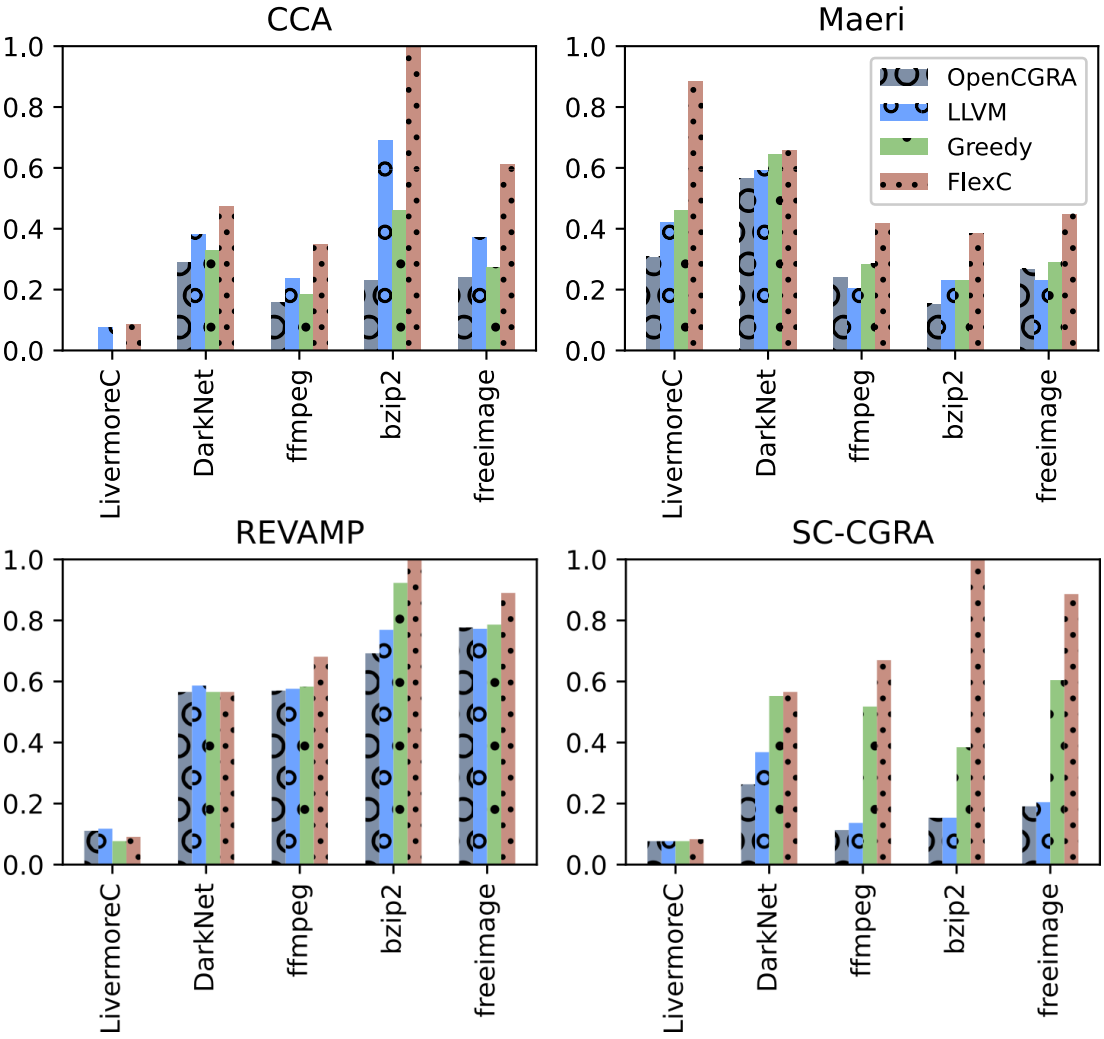


Figure 6.9: Results for each accelerator pairing by benchmark suite. Equality saturation often dramatically improves coverage for particular workload-accelerator combinations (e.g. bzip2 on CCA and SC-CGRA, and LivermoreC on Maeri), where otherwise the accelerator would appear entirely unsuitable. In these cases, the accelerator has the right class of operator for the tasks required (logical operators for bzip2 and floating-point operators for LivermoreC) but the code still requires transformation to fit the individual available operations.

used, because of the workload's heavy use of floating-point operations, though it is less suited to Bzip2 than CCA because of Maeri's lack of boolean arithmetic.

LLVM performs well on the CCA architecture as it has a more comprehensive set of rewrite rules than have been implemented in FlexC, and on the CCA architecture, the canonicalization rules it uses are appropriately targeted. Nevertheless, FlexC outperforms it due to more comprehensive exploration of the rewrite space.

This case study, on non-CGRA architectures, reveals the generality of FlexC: while we do not claim that this comprehensively demonstrates that our rewriter can compile to different architectures (as we still rely on the OpenCGRA backend in this example), it does demonstrate that FlexC may be applicable to more diverse computation models than CGRAs.

6.7.1.2 REVAMP

We implement REVAMP in the OpenCGRA framework and compile each of our benchmarks to it (fig. 6.8). FlexC increases the number of loops that can be supported on this CGRA by 15%, consistently across different workloads (fig. 6.9).

This increase is small because REVAMP's example already supports almost all the required operations for non-floating point code. We will see in other examples that FlexC becomes more important as the domain becomes more restricted.

6.7.1.3 SC-CGRA

Figure 6.8 shows FlexC increases the number of loops that can be supported by a factor of 5.2×.

This case study demonstrates FlexC is not only relevant within heterogeneous fabrics: if a homogeneous CGRA lacks operations that compilers typically assume to be available, FlexC's methods may still be necessary to generate working code. Bzip2 in particular (fig. 6.9) more than doubles the amount of targeted code once FlexC's equality saturation is used, compared to greedy-only, because otherwise it gets stuck in local minima and fails to explore the space enough to find a match.

6.7.2 Compilation Rate: Architectures Specialized for Loops

We demonstrate that the rewriting technique used by FlexC is applicable to many different specialized CGRAs within a varied design space.

Using 300 randomly selected loops in our benchmark suite, we first build a heterogeneous CGRA designed for that loop in particular. We run FlexC across the other loops in the benchmark suite and measure which loops can and cannot be compiled. Figure 6.10 shows what fraction of loops can be compiled, making a distinction between loops that are in the same suite (and so are often more likely to share the same class of operation) and loops from different domains.

FlexC improves the applicability of the accelerators, both within the domain they were designed for by a factor of 2.3×, and between domains by a factor of 2.9×, demonstrating the applicability of FlexC to many different types of heterogeneous CGRA. In some cases, a typical accelerator for a loop in one benchmark will actually do better on the other workloads (e,g, for freeimage and ffmpeg). This is because freeimage and ffmpeg are highly diverse, and so an accelerator designed for one loop is less likely to match others in the same diverse benchmark.

Figure 6.11 shows FlexC is able to support CGRAs across a wide range of specializations, from very specialized CGRAs with only a few operators available to more complex heterogeneous CGRAs with many operators. In this figure, we take each of the 300 generated architectures, and count the number of operations they support. We can see that for architectures with fewer operations, equality saturation is more important, as there are fewer paths to a valid rewrite.

6.7.2.1 Speedups

This section demonstrates that rewriting code in ways that at first-glance are inefficient can result in speedup by enabling accelerator utilization. Compiling to CGRA implementations typically improves performance *and* reduces power usage. We consider speedup in this evaluation. In line with other CGRA work, we consider speedup in the case that loops are executing large numbers of iterations, so one-time overheads like offloading costs for loosely coupled accelerators are ignored.

We compare two systems with similar specifications. For a CGRA system, we take architectural parameters for ADRES [339], a 6x6 CGRA which we clock at 200 MHz. We use the initialization interval generated by OpenCGRA to obtain performance estimates for the CGRA. To obtain a realistic CPU baseline, we execute the loops on an Arm A5 running at 500 MHz using an Analog Devices SC-589EZKit development board [17] and methodology for generating inputs from Exebench [57]. Speedups are shown in figure 6.12, showing a geomean performance improvement of 3×, demonstratng that FlexC's rewrite rules are not only effective in enabling targeting of CGRAs,

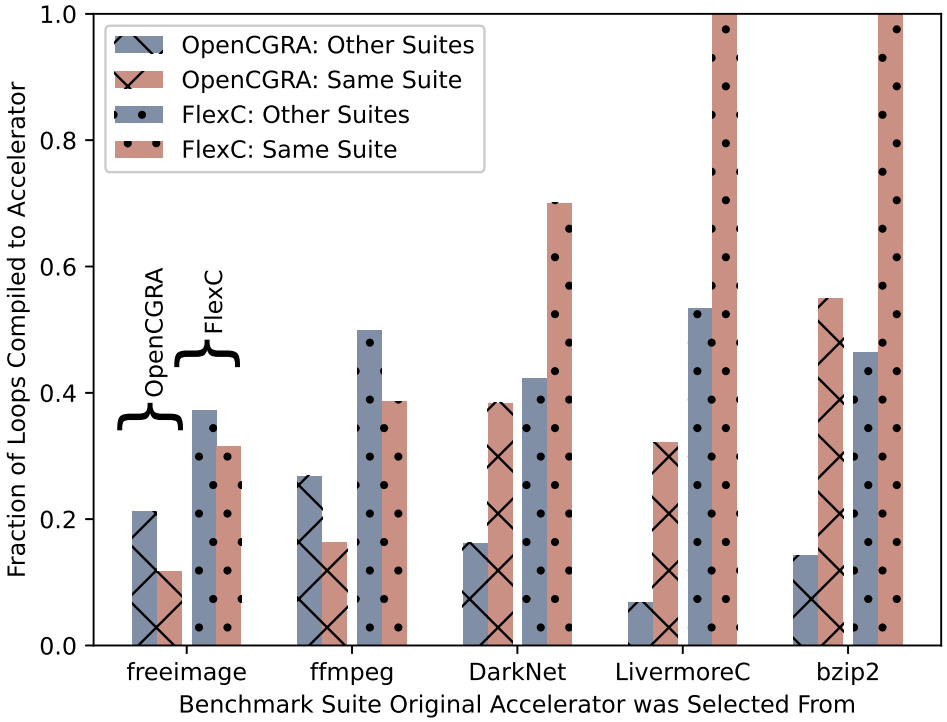


Figure 6.10: Using accelerators designed for individual loops in each benchmark suite, how much code in the same suite (red) and other suites (blue) can be compiled to these accelerators. FlexC increase the compilation rate by a factor of 2.3× in the same suite and 2.9× between suites.

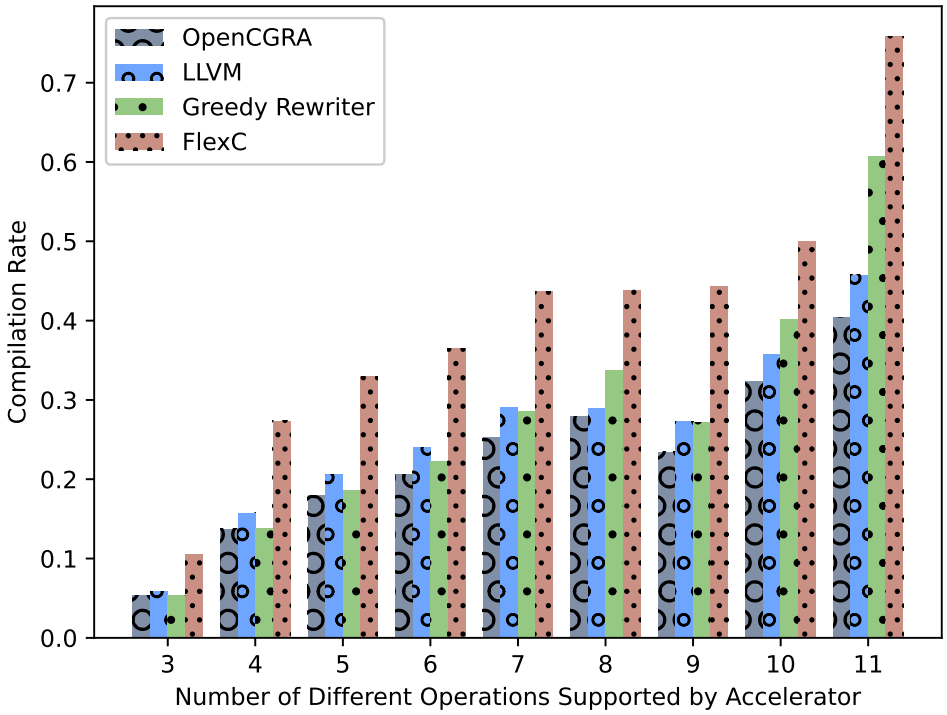


Figure 6.11: How the number of different operations in a CGRA influences compilation rate. We can see that FlexC performs consistently across many levels of generality, from very specialized accelerators with only a few operators to much more generic accelerators with many different operators, and that equality saturation is more important for more specialized architectures.

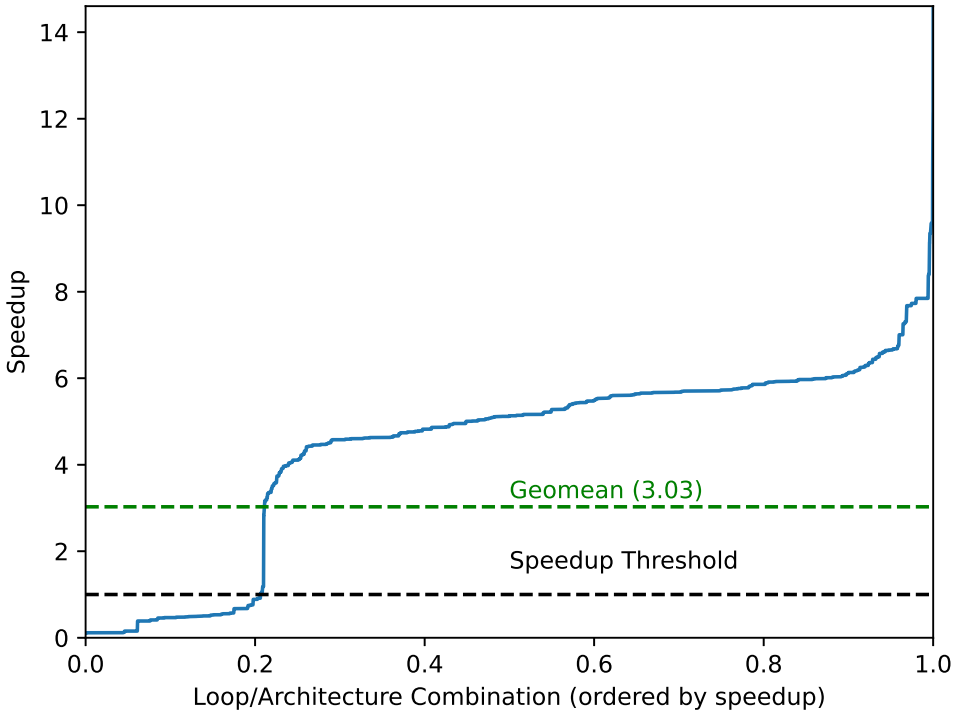


Figure 6.12: Speedup achieved by rewriting applications to run on a low-power CGRA vs running on a comparable low-power CPU. The geomean speedup is 3× over running on a CPU.

but also in achieving speedup on them.

6.7.3 Existing Domain-Specific Accelerators: End-to-End Evaluation

We demonstrate that FlexC also performs well on well-known and computationally important kernels. To do this, we take the OpenCGRA benchmark suite [468], along with the LivermoreC benchmark suite previously explored. We use the same setup as in Section 6.7.2.1.

The results are shown in Figure 6.13. Compared to running on an Arm Cortex-A5, FlexC achieves a speedup on 2.0× across all applications. This compares to the LLVM rewriter, which is only able to extract 1.5× performance increase across all applications.

[5]We have omitted ADPCM Encoder/Decoder as OpenCGRA is unable to compile it to any accelerators due to size, and Conv, FFT, MVT and Relu due to presence of divide operations that cannot be eliminated as none of our case-study accelerators support divide operations.

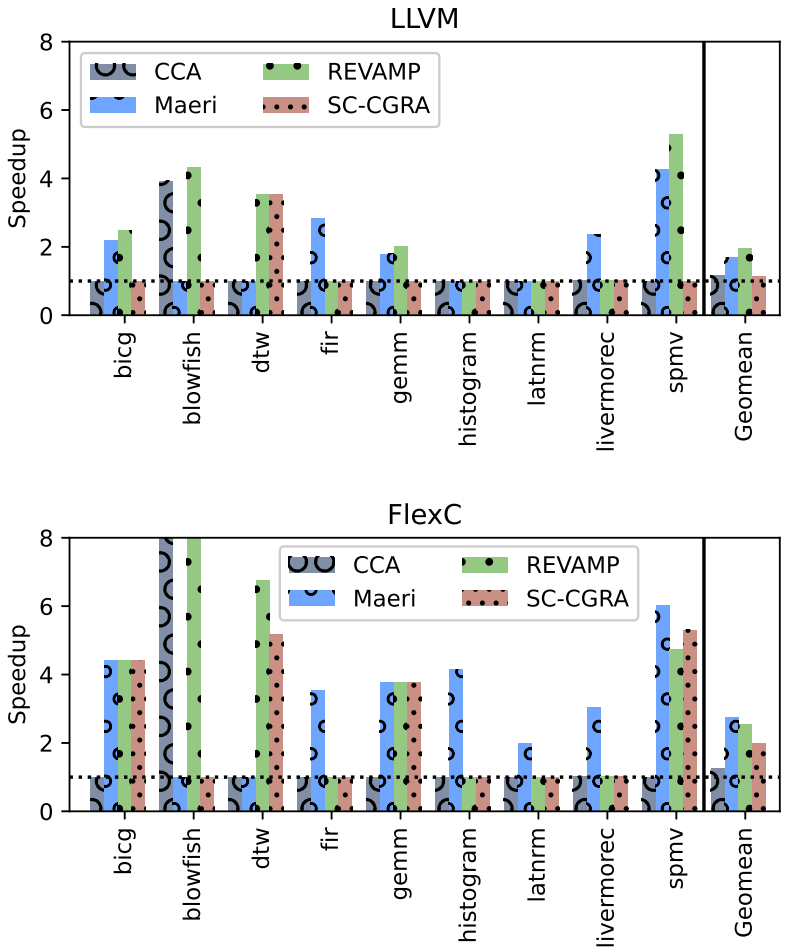


Figure 6.13: Speedups using the OpenCGRA benchmark suite and the Livermore C benchmark suite, comparing various CGRA architectures to an Arm Cortex-A5. Benchmarks that were unsupported by any architecture/compiler pairs have been omitted[5]. The top figure shows the speedup achieved using the LLVM rewriter to target each CGRA, and the bottom figure shows the speedup achieved using FlexC to target each CGRA.

6.7.4 Using Different Rulesets

FlexC provides a generic rewriting framework that can be applied to many different rulesets. These rulesets may be flagged by the programmer as valid for particular loops, or valid for a particular program.

We inspect four different rulesets here (covered in more detail in Section 6.5.1), an integer ruleset, derived of rules that may always be applied, a floating-point ruleset, derived of rules that may be applied under assumptions such as `-ffast-math`, a logical operations-as-binary operations ruleset that can be used to provide greater flexibility of rewrites involving logical operators and a stochastic computing ruleset that enables typical stochastic computing transformations. These secondary rulesets can be activated by the programmer using a flag. Figure 6.14 shows how these different rulesets provide different compilation performance. Rulesets are run in combination with the int ruleset as it contains many *enabling* rewrites for the specialized rewrites in the other rulesets. We can see that determining which rulesets are useful is architecture-specific. For example, Maeri benefits a lot from the logic-as-boolean ruleset, as it does not have logic operators, while CCA benefits from the stochastic rules as it does not have multipliers.

6.7.5 Most Frequently Applied Rewrite Rules

Part of the power of FlexC is that the rewrite rules that need to be applied vary by architecture. By using equality saturation, FlexC is able to use one standard set of rules for all architectures and apply the relevant rules in each case. Table 6.6 shows the most frequently applied rules for the CCA and Maeri architectures (when compilied using the integer and floating point rulesets): two architectures with nearly disjoint sets of operators.

6.7.6 Compile Time

A challenge with Equality Saturation is in keeping the search-space manageably sized, as e-graphs can grow rapidly, causing excessive compile times and resource usage [272, 279]. We avoid these issues in FlexC by limiting the number of explorative iterations, still finding good solutions in many cases.

Figure 6.15 shows the time taken by FlexC to rewrite and schedule the DFG. We use a cutoff time of 300 s to avoid exploring the rewrite space fruitlessly — we can see that the rate of successful compilations drops off rapidly after 60 s, followed first

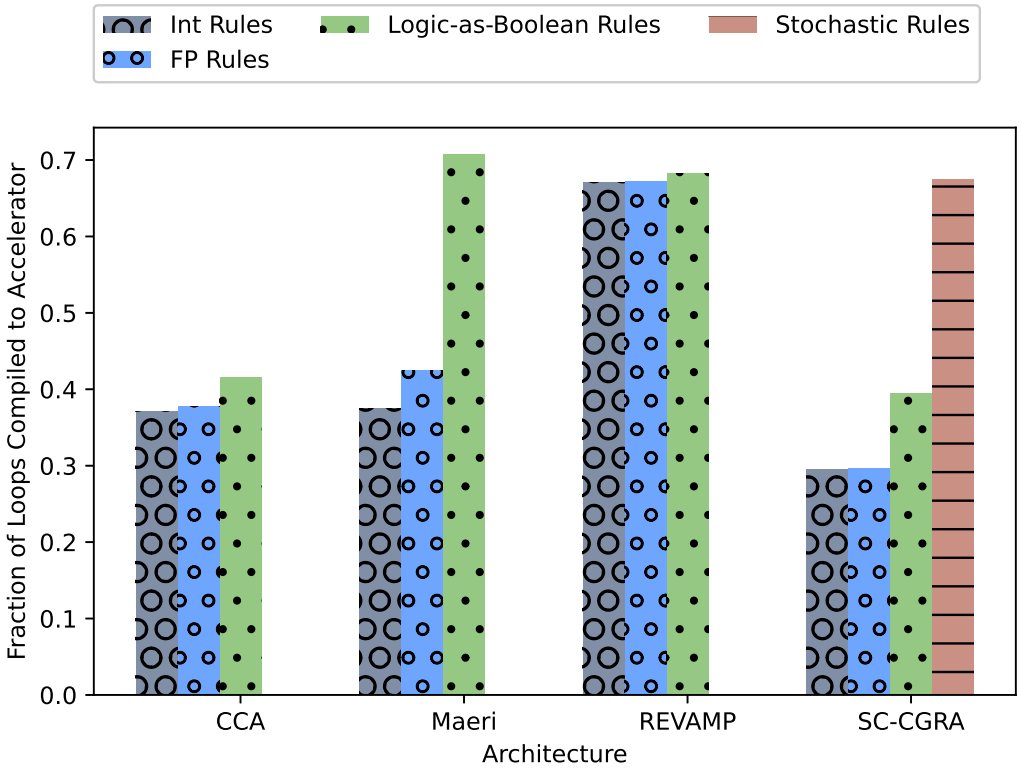


Figure 6.14: Comparing how different sets of rewrite rules improve the code coverage of an accelerator. All rulesets are run with the int ruleset. The stochastic computing rules are only applied to SC-CGRA as they require specialized hardware support not available in other accelerators.

	CCA
1	`x * 2 => x << 1`
2	`x * 4 => x << 2`
3	`x * 1 => x`
4	`-x (Floating Point) => x + 2^32 (Int)`

	Maeri
1	`x * 1 => x`
2	`x << 1 => x * 2`
3	`x << y => mul(x, load(csel(y > 32, 33, y)))`
4	`x - y => x + -y`

	REVAMP
1	`x * 1 => x`
2	`-x (Floating Point) => x + 2^32 (Int)`
3	`x / 2 => x >> 1`
4	`x / 8 => x >> 3`

	SC-CGRA
1	`x * y => x & y`
2	`x * y => ISC(x, y)`
3	`x * 1 => x`
4	`-x (Floating Point) => x + 2^32 (Int)`

Table 6.6: The most commonly applied rules for each architecture. We omit LLVM-specific rewrites for SC-CGRA. As the CCA and Maeri provide nearly disjoint operators, they are excellent examples of the need for rewrite rules to apply both forward and backward.

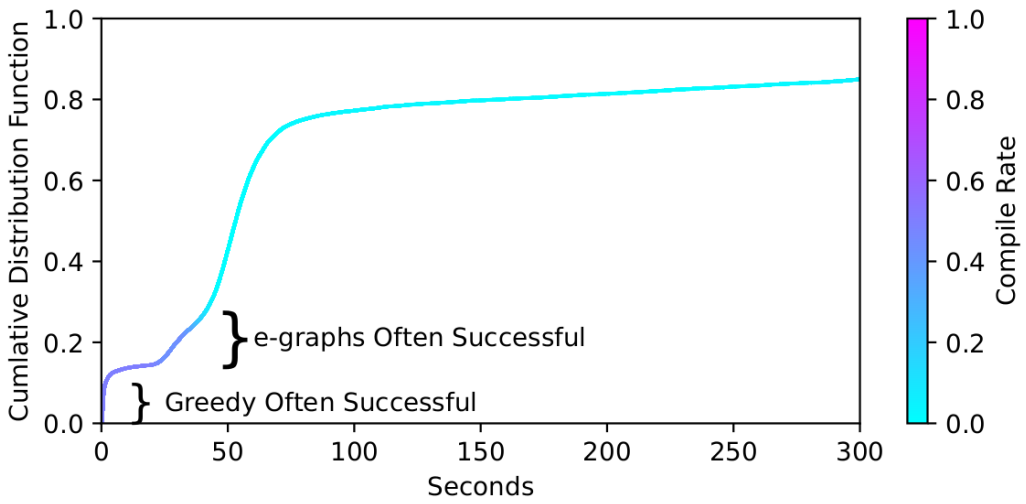


Figure 6.15: Time taken to schedule code on a CGRA using FlexC and OpenCGRA. We cut-off rewriting at 300 s to avoid excessive exploration. After 60 s, the compilation rate is very low, so FlexC is not missing many compilations at longer timeouts.

by a large number of early terminations without successful scheduling (most likely due to reaching saturation, iteration or node limit), then by stagnation in progress for infeasibly large search spaces. These compile times are fast enough that more exhaustive CGRA schedulers will be able to incorporate this strategy within the existing order of magnitude of compilation time. For example Beidas and Anderson report ILP compile times with a geomean of 60 s [72].

We can also see the effect of using a greedy rewriter as a preliminary step here. In 10% of cases, FlexC is able to rely on the greedy rewriter and find a compiling loop rapidly. We can further see that when FlexC uses equality saturation, it is more successful early on in the exploration.

6.8 Conclusion

We introduce FlexC, a compiler for domain-specific CGRAs that addresses the domain-restriction problem: where CGRAs that have been designed for a particular domain are hard to apply to software outside that domain. FlexC uses equality-saturation to rewrite software from different domains so it can run on hardware not designed for it. FlexC increases the number loops that can be supported by a factor of 2.2× over existing CGRA compilers and enables acceleration of loops leading to geomean speedup of 3×.

FlexC demonstrates the potential that rewriting software to match novel hardware

has: the techniques developed here are applicable to other kinds of accelerators with programmable networks. We present the first study that characterizes how different decisions surrounding heterogeneity effect the fraction of code supported by an accelerator, showing that the more specialized an accelerator is, the more important FlexC is. FlexC opens up new development possibilities by promising that even if software requirements change in a heartbeat, accelerators with a large sunk-cost can still be applied.

FlexC demonstrates a strategy to search for accelerator configurations in reconfigurable accelerators. This lays the groundwork for application to the accelerator equation (Chapter 2), leaving the finding of suitable f and g to further broaden the applicability of domain-specific CGRAs to future work.

Chapter 7

Conclusions

This thesis has identified a key challenge in programming the heterogeneous accelerators that enable the next generation of computing performance. We introduce a generic framework to overcome limitations of fixed-function hardware accelerators and reconfigurable hardware accelerators in the form of the acceleration equation in Chapter 2, and we demonstrate three real-world solutions to these equations.

We demonstrate large increases in the volume of software that can be supported by these domain-specific accelerators, showing reductions of processing power by a factor of ten for 84% of expressions using RXPSC. Finally, using FACC, we compile 18 high-level code implementations to an API-programmable FFT accelerator that would not have been directly supported. Using FlexC, we enable 2.2x more software to be supported on CGRAs than with traditional compilers. For each of these examples, we show speedups.

Although this thesis's case studies focus on three examples, the equations we introduce are more widely applicable and practical solutions exist for many hardware accelerators.

7.1 Contributions

This section gives a brief overview of the contributions of this thesis.

7.1.1 Large-Scale Benchmarks

This thesis diverges from traditional compilers research in its benchmarking strategy. Designing compilers for hardware accelerators that work across the variety of code that

programmers write means that traditional monolithic benchmark suites do not work. These traditional benchmark suites typically scrutinize the performance of a benchmark across a narrow range of programs, rather than the compiler across a broad range of programs.

The approach taken in this thesis is to benchmark broadly, across as wide a range of code as is possible. In Chapter 5, we introduce a benchmark suite of 24 different implementations of the FFT, to demonstrate that FACC works across a variety of different implementations. Similarly, in Chapter 6, we introduce a benchmark suite of more than 2,000 loops taken from large projects. This approach is extremely relevant when benchmarking compilers in an effort to make it more likely that compilers will actually work in the face of unseen code, rather than compilers that are simply fine-tuned to make individual benchmarks run as fast as possible.

7.1.2 The Acceleration Equations

The key contribution of this thesis is the acceleration equation:

$$U = g \circ A \circ h$$

Solutions to this equation enable the use of hardware accelerators for a much wider variety of code than the accelerators are designed to run.

This thesis explores solutions to this equation on two different types of accelerators, regular expression accelerator, Fourier transform accelerators. It further explores the background required to find solutions to the accelerator equation for domain-specific CGRAs.

7.1.3 Regular Expression Accelerators

The work in Chapter 4 introduces RXPSC, a compiler that solves the acceleration equation for regular expression accelerators. This compiler compiles new regular expressions to old accelerators, enabling fast updates to FPGA-based accelerators. RXPSC is built using a novel intermediate representation that models the structure of a regular expression, and novel analysis algorithms that enable deduction of g and h using stateless translators.

7.1.4 Fourier Transform Accelerators

The work in Chapter 5 introduces FACC, a compiler that solves the acceleration equation for Fourier transform accelerators. This compiler uses a novel program-synthesis approach to generate the adaptor functions g and h.

This chapter analyzes the concept of *mismatch* in traditional programming languages, categorizing it in four ways: code mismatch, data mismatch, domain mismatch and behaviour mismatch. These mismatch classifications will drastically simplify the design of similar program-synthesis tools for different domains.

7.1.5 Domain-Specific CGRAs

Finally, the work in Chapter 6 introduces FlexC, which solves the *reconfigurable* acceleration equation. FlexC uses equality saturation to explore the space of program rewrites. This is a novel approach for CGRAs, which have traditionally relied on the same rewriting approaches that are used for CPUs — even though these rewriting approaches are totally unsuitable for these domain-specific bits of hardware.

7.2 Critical Evaluation

This section discusses the threats to validity of the research in this thesis.

7.2.1 Compilation Time

The techniques developed in this thesis are far more compute intensive than traditional compiler passes. For example, the passes developed for RXPSC can be exponential in the worst case, the extractor for FlexC relies on the NP-Complete ILP task, and the program synthesis in FACC relies on running the underlying program.

These compile times limit the applicability of these techniques within traditional compilers, but do not defeat the purpose of these tools, as they can be run in the background and provide their compilation results when available without impacting the behaviour of the program.

7.2.2 Changing Accelerator Landscapes

A key challenge of developing tools for hardware accelerators is that the hardware accelerators often change. For example, in many modern FPGAs, partial reconfiguration

can replace the fast reconfigurability that RXPSC provides. However, there are still smaller FPGAs without this feature, and, more generally, the compilation strategies developed for RXPSC are more broadly applicable (and have been applied to different types of regular-expression accelerator [529]).

Although accelerators may come and go, the techniques developed in this thesis have broader conceptual applicability. For example, they apply in the case of FlexC to any operation-restricted accelerators and in the case of FACC to any API-programmable accelerators.

7.2.3 Overheads

All three tools discussed in this thesis introduce overheads: RXPSC in the form of extra hardware requirements for the stateless translators, FACC in the form of the adaptor functions, and FlexC in the form of less optimal code.

In all these cases, the overheads are low compared to the potential speedup. For RXPSC, we estimate 10% overhead. For FACC, we have shown that there are significant speedups available even with modest accelerator speeds and with huge overheads in Section 2.1.1. For FlexC, we have included an evaluation of program speed after rewriting to enable the use of an accelerator, although a more thorough theoretical analysis of the potential overheads from rewriting is left to future work.

7.2.4 Correctness Challenges

With the advanced analyses that this thesis has developed come correctness challenges. In the case of RXPSC, we are able to address these with a post-compute correctness check. However, in the case of FACC, proving correctness of the transformations is infeasible with existing theorem provers, as the FFTs it accelerates rely on large floating-point arrays. Beyond this tough technical challenge, there are further questions on what it means for two algorithms to be equivalent. Finally, although FlexC provides correct-by-construction compilation, simple extensions with incorrect rules (such as the stochastic computing ruleset we explore) raise quesitons about how to measure correctness.

In all these cases, we rely on the programmer to verify that the accelerated code behaves appropriately. Developement of truly automated techniques is left to future work, but in many cases, the speedups enabled by accelerators are significant enough that the programmer overhead is acceptable.

7.3 Future Work

The work in this thesis opens numerous avenues for future work.

7.3.1 More General Solutions to the Acceleration Equation

This thesis has explored three different solutions to the accelerator equation for three specific accelerators. To further develop this work, we are exploring a wider range of accelerators. The techniques developed for FACC have already been applied to matrix multiplications [331] and the techniques developed for RXPSC have already been applied to symbol-only-reconfigurable regex accelerators [529].

Anticipating that new domains will tax the analyses that we have developed, we intend to develop solutions to the synthesis problems that the acceleration equation poses across a much wider range of hardware accelerators.

7.3.2 Overcoming Correctness Challenges

The correctness challenges that these new compilation strategies bring also raise new opportunities. The first opportunity is for the development of better theorem-proving techniques that can handle that diversity of code that tools like FACC are able to generate.

More generally, many accelerators make assumptions about code that are often not valid in the context the programmer has written the code. We propose the development of compilers that are able to communicate these assumptions in a tractable manner to the programmer so the programmer can guide the compiler to overcome these correctness issues.

7.3.3 Co-design

The development of better compilers for hardware accelerators opens the question of whether better hardware accelerators can be designed in the first place. For example, could the overheads that running g and h in the acceleration equation be eliminated by introducing more accelerator flexibility? We are actively exploring codesign opportunities with the FlexC compiler, using it to enhance existing design-space opportunties in the CGRA space.

7.4 Summary

This dissertation has introduced the acceleration equation, and three tools that address this equation in different contexts, addressing compilation challenges in regular expression accelerators, Fourier transform accelerators and CGRAs.

This dissertation has demonstrated the potential that advances in compilation techniques have to make hardware accelerators usable by the many, not the few.

Chapter 8

Appendix

8.1 Derivation of the complexity of the (plus) case (Chapter 4)

This is a dual partitioning problem. In the dual partitioning problem, we consider two strings of length m and k and consider how many different ways they can be partitioned into different substrings with the following properties:

1. The substrings may be any length greater than zero
2. The substrings must represent the entire string
3. Both strings must have the same number of substrings.

We begin by considering the single partitioning problem, which considers a single string and excludes property 3. We show that the number of partitions is exponential in the length of the string. We first attempt to find the number of ways a string of length m can partitioned n times such that each partition is non-empty, where n is a most m.

We can envision this as a problem where there are $m-1$ "gaps" between the characters of the string. Each gap may either contain a partition between substrings, or no partitions. This leaves us with 2^{m-1} possible partitions.

8.1.1 Application to the Dual Partitioning Problem

As this is a dual partitioning problem, a second string of length k has 2^{k-1} partitions.

However, the two partitions are only compatible if they have the same number of substrings. Notice that we may also write this result as the number of partitions possible for each number of substrings:

$$2^{k-1} = \binom{k-1}{0} + \binom{k-1}{1} + \cdots + \binom{k-1}{k-1}$$

Likewise, consider that we may write:

$$2^{m-1} = \binom{m-1}{0} + \binom{m-1}{1} + \cdots + \binom{m-1}{m-1}$$

Without loss of generality, assume that $m \leq k$. And that the dual partition, where we require the same number of partitions in each string is:

$$\begin{aligned} &DualPartitions(k,m) = \\ &\binom{k-1}{0}\binom{m-1}{0} + \cdots \binom{k-1}{m-1}\binom{m-1}{m-1} + \\ &\binom{k-1}{m} + \cdots \binom{k-1}{k-1} \\ &\geq 2^{k-1} \geq 2^{m-1} \end{aligned}$$

Where the greater than property holds as a consequence of each choose function being strictly greater than zero. From this, we can conclude that $DualPartitions(k,m) \geq 2^{\min(k,m)}$ and that the dual partitioning discussed is indeed exponential in the lengths of the inputs.

8.1.2 Resolution of other Discrepancies

It should be noted that the assumption that "each partition is non-empty" is not true — partitions may be empty, for example disabling terms such as loops. The above bound still holds, as the proof may simply be amended to allow for k partitions between every string character resulting in a strictly greater number of partitions.

8.2 Correctness Guarantees in FACC (Chapter 5)

Proving equivalence of two arbitrary pieces of code is infeasible. However, in this section we show that FACC can provide high quality matches, with arbitrarily high probability of being correct. This can allow for programmer-verification on replacements that are nearly guaranteed to work, or automatic replacement where a tolerance for such optimizations exists. Further, achieving these high accuracies is feasible using a relatively small number of examples.

8.2.1 Algorithm

At a high level, for user code function U and accelerator A, FACC finds functions g, h such that $\forall x.U(x) = g(A(h(x)))$.

To achieve this, FACC comes up with a number of candidate bindings, where each binding binds variables to each other and makes assumptions about the length of arrays to allow for appropriate copying to avoid issues with differences between in-place vs out-of-place Fourier transforms. Testing is random, and requires the following properties:

Random-Inputs Input values are chosen randomly.

Different-Inputs There will always be at least one input example where two parameters have different values.

Whole-Input Inputs must be chosen so that the entire input must be read to determine the correct output.

8.2.2 Equivalence Under Ideal Circumstances

We show that the bindings, the solution to the data mismatch problem, are correct under the assumption that the user code and the accelerator are otherwise bit-for-bit equivalent and that they are injective functions. The intuition here is that if the functions are injective, then a single input/output example can be used to show that a particular binding is correct; any different binding would have a guaranteed different input and therefore guaranteed different output. Handling domain mismatches does not affect this proof, but handling behavioral mismatches does, and is discussed in Section 8.2.4.

8.2.3 Potential Sources of Incorrectness

The binding problem aims to achieve two concepts:

Array Dimensions Each array has the correct dimension specified.

Correct Mappings All data is mapped to the correct accelerator parameter.

From these properties, it is easy to infer a number of further important properties about the generated binding, e.g. that the data alignment is correct and that the input-ranges have been checked appropriately.

8.2.3.1 Proof of Array Dimensions

We first consider the array dimension problem. Suppose that an array that actually has a length n is spuriously assigned a length of m and proceed by cases:

$m = n$ By the different-input condition, this can only occur if the binding is correct, provided at least one input is the size of the array. Otherwise, the incorrect binding would need to have the same value as the correct one, which is disallowed in the test set.

$m < n$ In this case, the array passed to the accelerator will be shorter than the corresponding array in user code.

In this case, when FACC generates inputs and passes them to the user code, the user code will access out-of-bounds (because the variable specifying the length of the array is larger than the array). By the whole-input property, we know either that this access will happen, or that the answer will be incorrect. We use AddressSanitizer [427] to ensure this undefined behavior is detected.

$m < n$ In this case, the programmer's code will produce a valid output, but only for a part of the problem. By the injectivity assumption, we know that the two sets of results will be distinct between the program with the correctly-set array length and this program — therefore we will be able to tell by the IO examples that this is the incorrect array length assignment.

8.2.3.2 Proof of Correct Mappings

If parameters are incorrectly mapped, by the different-input property they will produce output differences which can be detected by the injectivity property. To see this, suppose we map some user parameters x to some accelerator parameters y, but that the *correct* assignment of parameters is to some y'. That is; $U(x) = A(y')$. Then, by injectivity of the accelerator, and different-input property, we know that $A(y) \neq A(y')$ and so we cannot find a false-positive.

8.2.4 Probabilistic Correctness in Practice

The assumptions of bit-for-bit equivalence and injectivity of real implementations are not likely to hold in practice. Both of these concepts can be modelled with a concept of near-injectivity, which allows compilation with arbitrarily-high probability.

8.2.4.1 Near-Injectivity

True injectivity in code is rare even if the code is attempting to model some injective mathematical function due to issues such as error codes and floating-point limitations. In reality, we expect near-injectivity, which accounts for a portion of the uncertainty (and need for programmer sign-off). We say that a function f is $\varepsilon-$injective if for any two random distinct input vectors x, y $\mathbb{P}(f(x) \neq f(y)) \geq \varepsilon$. Handling floating-point inequalities (discussed below) reduces ε further. Excluding error conditions, the $\varepsilon-$injectivity of FFT implementations is very close to 1, meaning that code is very easy to distinguish with near certainty using a moderate number of examples. That is, if we use n examples, our probability of finding at least one set of inputs that do not spuriously share an output is $(1-\varepsilon)^n$.

Intuition here can be taken by considering functions with that are very far from being injective. An example is the `iseven` function, which has an ε-injectivity of approximately 0.5. Even in this case, showing the bindings to be correct could achieve very high accuracy with only a few examples.

8.2.4.2 Post-Synthesis

Post-synthesis, such as inferring normalization code, is important in extending the coverage of FACC to more varied user code. However, it has a marginal effect on the ε-injectivity of the code.

Individual inputs may incorrectly report the same results under post synthesis. For example, an accelerator that normalizes the result and user code that does not could be provided with input so that incorrect bindings produce identical outputs, but this situation is extremely rare. The chances of false positives increase due to testing more functions, but this is easily accounted for by increasing the number of examples, since each additional example exponentially increases the probability of correctness.

8.2.4.3 Incorrect Algorithm Identification

The potential for the identification pass to identify code that does not perform the same algorithm as the accelerator is harder to quantify. While any significantly different algorithms are unlikely to generate probable bindings, and vanishingly unlikely to match on even a single IO example (much less many IO examples), examples where partial matching is possible are easy to construct, if unlikely in real code for Fourier transforms. In these cases, FACC cannot guarantee correctness, and the programmer

must give assent that the algorithm they have implemented is the same as the one the accelerator accelerates. They are assisted in this verification by being given the FACC bindings in their source language (figure 5.7), which directly show the assumptions and constraints the technique has generated.

8.2.4.4 Compiling Unidentical Implementations

In reality, accelerators will not match bit-for-bit the results of real code. Attempting to preserve bit-for-bit equivalence of floating-point implementations is fruitless, as floating-point semantics are not intended to capture the wider semantics of functions but rather single operations. Fortunately, as long as both code and accelerator implement Fourier transforms, the result is likely to be acceptable to the programmer.

We can capture this using a probability analysis. Above, we have assumed bit-for-bit equivalence to enable equivalence checking. That is, for some randomly selected x, $f(x) = g(x) \implies P(f = g > 1 - \varepsilon)$. In floating-point computations, we often have an error tolerance. We can model this with a tolerance function T which captures an idea of closeness if $g(x) \in T(f(x))$.

Supposing that the average cardinality of $T(f(x))$ is K[1], and that an incorrect binding produces a random result, the probability that a single example induces equality is:

$$g(x) \in T(f(x)) \implies P\left(f = g > (1-\varepsilon)\left(\frac{K}{\#\text{range}(g)}\right)\right) \tag{8.1}$$

8.2.4.4.1 Floating-Point Equality Despite the portability of the IEEE 754 floating-point standard [20], it is designed around small-step operations. Ultimately, requiring such a precise definition of equality for big operations such as FFTs is akin to specifying the implementation step-by-step. Although definitions of floating-point equality at a function-level exist [285], these focus on introducing concrete constraints surrounding edge cases rather than focusing on the crux of the issue. Allowing replacements regardless of accuracy is likely to be unproductive, as mathematical functions exist on a spectrum of accuracy/performance [86, 448], with FFTs no exception [317, 342]. As a result, replacements must have defined error properties that the programmer is aware of — a common analysis on FFT algorithms [107, 355, 420] to enable the programmer to make informed decisions about whether the replacement accelerators provide high enough accuracy.

[1] That is, $K = (\sum_{x \in X} \#T(f(x)))/\#X$

8.2.5 So What Does This Mean for the User?

The correctness guarantees here are understandably nuanced; complete proofs of equality would be paradoxical. However, the correctness guarantees mean that the programmer must only check one thing: “does the API compiled to perform the same function as the code they wrote on the inputs for which it is valid?”

Supposing the answer to this is yes, and near-injectivity holds for the function in question, the user can be assured with arbitrarily-high probability that the mapping is correct.

Bibliography

[1]

[2] Available at bluespec.com.

[3] B4860 QorIQ Qonverge Multi-Accelerator Platform Engine Baseband 4 (MAPLE-B3) reference manual. Available at https://www.nxp.com/files-static/training_pdf/vFTF09_AN149.pdf.

[4] bzip2. Available from git://sourceware.org/git/bzip2.git.

[5] Chisel/FIRRTL hardware compiler framework. Available at chisel-lang.org.

[6] Darknet. Available from git://github.com/pjreddie/darknet.

[7] Ffmpeg. Available from git://git.ffmpeg.org/ffmpeg.git.

[8] Freeimage. Available from http://downloads/sourceforge.net/freeimage/FreeImage3180.zip.

[9] Livermore loops. Available from https://netlib.org/benchmark/livermorec.

[10] Opencl. Available at opencl.org.

[11] Processor benchmark limitations.

[12] Defense science board task force on high performance microchip supply. 2005.

[13] ADSP-214xx SHARC processor hardware reference. Technical Report 82-000469-01, Analog Devices, 2010. Available at https://www.analog.com/media/en/dsp-documentation/processor-manuals/ADSP-214xx_HRM_rev0.3.pdf.

[14] KeyStone architecture fast Fourier transform coprocessor (FFTC). Technical Report SPRUGS2C, Texas Instruments, 2011. Available at https://www.ti.com/lit/ug/sprugs2c/sprugs2c.pdf.

[15] SHARC processor: ADSP-21467/ADSP-21469. 2013.

[16] Keystone II architecture fast Fourier transform coprocessor (FFTC). Technical Report SPRUHE0A, Texas Instruments, 2015. Available at https://www.ti.com/lit/ug/spruhe0a/spruhe0a.pdf.

[17] Analog devices SHARC+ dual-core DSP with Arm Cortex-A5: ADSP-SC582/SC583/SC584/SC589/ADSP21583/21584/21587. 2018. Available at https://www.analog.com/media/en/technical-documentation/data-sheets/ADSP-SC582_583_584_587_589_ADSP-21583_584_587.pdf.

[18] AN12282: Digital signal processing for NXP LPC5500 using PowerQuad. (AN12282), January 2019. Available at https://www.nxp.com/docs/en/application-note/AN12282.pdf.

[19] CrossCore embedded studio 2.9.0: C/C++ library manual for SHARC processors. (82-100118-01), 2019. Avalable at https://www.analog.com/media/en/dsp-documentation/software-manuals/cces-sharclibrary-manual.pdf.

[20] IEEE standard for floating-point arithmetic. Technical Report 754-2019, Microprocessor Standards Committee, 2019.

[21] Intel oneAPI math kernel library — data parallel C++ developer reference. Technical report, Intel, 2020. Available at https://docs.oneapi.com/versions/latest/onemkl/index.html.

[22] Accelerated computing with a reconfigurable dataflow architecture. 2021. Available at https://sambanova.ai/wp-content/uploads/2021/06/SambaNova_RDA_Whitepaper_English.pdf.

[23] ADSP-SC58x FFTA benchmarks. Technical report, Analog Devices, 2021. Available at https://ez.analog.com/dsp/sharc-processors/w/documents/5017/adsp-sc58x-ffta-benchmarks.

[24] FACC souce code (blinded), 2021. Available at `https://github.com/FourierACceleratorCompiler/FACC`.

[25] FFT classification environment, 2021. Available at `https://github.com/FourierACceleratorCompiler/FFTClassification`.

[26] FFTA evaluation environment (blinded), 2021. Available at `https://github.com/FourierACceleratorCompiler/FFTAEnvironment`.

[27] An introduction to CPU performance benchmarks and how this applies to the home market. November 2021.

[28] NXP PowerQuad evaluatoin environment (blinded), 2021. Available at `https://github.com/FourierACceleratorCompiler/NXPEnvironment`.

[29] International roadmap for devices and systems 2022 update: More moore. Technical report, IEEE, 2022.

[30] Xtensa instruction set architecture (ISA) summary. 2022. Available at `https://www.cadence.com/content/dam/cadence-www/global/en_US/documents/tools/ip/tensilica-ip/isa-summary.pdf`.

[31] Mohamed S. Abdelfattah, Lukasz Dudziak, Thomas Chau, Royson Lee, Hyeji Kim, and Nicholas D. Lane. Best of both worlds: Automl codesign of a cnn and its hardware accelerator. *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 7 2020.

[32] Riadh Ben Abdelhamid, Yoshiki Yamguchi, and Taisuke Boku. Condensing an overload of aprallel computing ingredients into a single architecture recipe. *ASAP 2020*, 2020.

[33] Mithun Acharya and Tao Xie. *Mining API Error-Handling Specifications from Source Code*, pages 370–384. Springer Berlin Heidelberg, 2009.

[34] Stefan Ackermann, Vojin Jovanovic, Tiark Rompf, and Martin Odersky. Jet: An embedded DSL for high performance big data processing. *Big Data 2012*, 2012.

[35] Boma Adhi, Carlos Cortes, Tomohiro Ueno, Yiyu Tan, Takuya Kojima, Artur Podobas, and Kentaro Sano. Exploring inter-tile connectivity for HPC-oriented CGRA with lower resource usage. *FPT*, 2022.

[36] Maaz Bin Safeer Ahmad, Jonathan Ragan-Kelley, Alvin Cheung, and Shoaib Kamil. Automatically translating image processing libraries to halide. *ACM Transactions on Graphics*, 38:1–13, 11 2019.

[37] Hameeza Ahmed, Paulo C Santos, Joao P C Lima, Rafael F Moura, Marco A Z Alves, Antonio C S Beck, and Luigi Carro. A compiler for automatic selection of suitable processing-in-memory instructions. *DATE*, 2019.

[38] Junwhan Ahn, Sungjoo Yoo, Onur Mutlu, and Kiyoung Choi. PIM-enabled instructions: A low-overhead, locality-aware processing-in-memory architecture. *ISCA*, 2015.

[39] Omid Akbari, Mehdi Kamal, Ali Kusha-Afzali, Massoud Pedram, and Muhammad Shafique. PX-CGRA: Polymorphic approximate coarse-grained reconfigurable architecture. *DATE*, 2018.

[40] Berkin Akin, Franz Franchetti, and James C. Hoe. Understanding the design space of DRAM-optimized hardware FFT accelerators. *ASAP*, 6 2014.

[41] Armin Alaghi and John P Hayes. Survey of stochastic computing. *Transactions on Embedded Computing Systems*, 12(2), 2013.

[42] Enseih Aliagha and Diana Gohringer. Energy efficient design of coarse-grained reconfigurable architectures: Insights, trends and challenges. *FPT*, 2022.

[43] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, 8 2015.

[44] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.

[45] Miltiadis Allamanis and Charles Sutton. Mining idioms from source code. *International Symposium on Foundations for Software Engineering*, 2014.

[46] Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. *ICLR*, 08 2018.

[47] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3:1–29, 1 2019.

[48] Muhammad Shoaib Bin Altaf and David A Wood. LogCA: A high-level performance model for hardware accelrators. *ICSA*, 2017.

[49] Amazon. Amazon ec2 f1 instances. 2020. Available at `https://aws.amazon.com/ec2/instance-types/f1/`.

[50] Shaahin Angizi and Deliang Fan. ReDRAM: A reconfigurable processing-in-DRAM platform for accelerating bulk bit-wise operations. *ICCAD*, 2019.

[51] Shaahin Angizi, Zhezhi He, Amro Awad, and Deliang Fan. MRIMA: An MRAM-based in-memory accelerator. *CADICS*, 2020.

[52] Kevin Angstadt, Jean-Baptiste Jeannin, and Westley Weimer. Accelerating legacy string kernels via bounded automata learning. *ASPLOS*, 3 2020.

[53] Kevin Angstadt, Arun Subramaniyan, Elaheh Sadredini, Reza Rahimi, Kevin Skadron, Westley Weimer, and Reetuparna Das. ASPEN: A scalable in-SRAM architecture for pushdown automata. *2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 10 2018.

[54] Kevin Angstadt, Jack Wadden, Westley Weimer, and Kevin Skadron. Portable programming with RAPID. *IEEE Transactions on Parallel and Distributed Systems*, 30:939–952, 4 2019.

[55] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. EGRA: A coarse grained reconfigurable architectural template. *VLSI*, 2011.

[56] Hassan Anwar, Syed M A H Jafri, Sergei Dytckov, Masoud Daneshtalab, Masoumeh Ebrahimi, and Ahmed Hemani. Exploring spiking neural network on coarse-grain reconfigurable architectures. *Mes*, 2014.

[57] Jordi Armengol-Estapé, Jackson Woodruff, Alexander Bruackmann, José Wesley de Souza Magalhães, and Michael F.P. O'Boyle. Exebench: An ML-scale dataset of executable C functions. *International Symposium on Machine Programming*, 2022.

[58] ArtSim. GraphSim. Available at `https://aws.amazon.com/marketplace/seller-profile?id=70ae1916-82bd-44d8-b7b5-b13790e2d0cb&ref=dtl_B07MDHPL53`.

[59] Kubilay Atasu, Raphael Polig, Jonathan Rohrer, and Christoph Hagleitner. Exploring the design space of programmable regular expression matching accelerators. *Journal of Systems Architecture*, 59:1184–1196, 11 2013.

[60] Michael Attig and Gordon Brebner. 400 Gb/s programmable packet parsing on a single FPGA. *ANCS*, 2011.

[61] Muhammad Waqar Azhar, Tung Thanh Hoang, and Per Larsson-Edefors. Cyclic redundancy checking (CRC) accelerator for the FlexCore processor. *13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*, 2010.

[62] Rick Bahr, Clark Barrett, Nikhil Bhagdikar, Alex Carsello, Ross Daly, Caleb Donovick, David Durst, Kayvon Fatahalian, Kathleen Feng, Pat Hanrahan, Teguh Hofstee, Mark Horowitz, Dillon Huff, Fredrik Kjolstad, Taeyoung Kong, Qiaoyi Liu, Makai Mann, Jackson Melchert, Ankita Nayak, Aina Niemetz, Gedeon Nyengele, Priyanka Raina, Stephen Richardson, Raj Setaluri, Jeff Setter, Kavya Sreedhar, Maxwell Strange, James Thomas, Christopher Torng, Leonard Truong, Nestan Tsiskaridze, and Keyi Zhang. Creating an agile hardware design flow. *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 7 2020.

[63] Timothy J Baker and John P Hayes. The hypergeometric distribution as a more accurate model for stochastic computing. *DATE*, 2020.

[64] Mahesh Balasubramanian, Shail Dave, Aviral Shrivastava, and Reiley Jeyapaul. LASER: A hardware/software approach to accelerator complicatd loops on CGRAs. *DATE*, 2018.

[65] Mahesh Balasubramanian and Aviral Shrivastava. Pathseeker: A fast mapping algorithm for cgras. *DATE*, 2022.

[66] Thilini Kaushalya Bandara, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. REVAMP: A systematic framework for heterogeneous CGRA realization. *ASPLOS*, 2022.

[67] Mohamad Barbar, Yulei Sui, and Shiping Chen. Object versioning for flow-sensitive pointer analysis. *CGO*, 2 2021.

[68] Henrik Barthels, Christos Psarras, and Paolo Bientinesi. Automatic generation of efficient linear algebra programs. *Proceedings of the Platform for Advanced Scientific Computing Conference*, 6 2020.

[69] Alberto Bartoli, Andrea De Lorenzo, Eric Medvet, and Fabiano Tarlao. Inference of regular experssions for text extraction from examples. *IEEE Transactions on Knowledge and Data Engineering*, 28(5), 2016.

[70] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. *ANCS*, 2007.

[71] Andreas Becher, Stefan Wildermann, and Jürgen Teich. Optimistic regular expression matching on FPGAs for near-data processing. *Proceedings of the 14th International Workshop on Data Management on New Hardware — DAMON '18*, 6 2018.

[72] Rami Beidas and Jason H Anderson. CGRA mapping using zero-supporessed binary decision diagrams. *ASP-DAC*, 2022.

[73] G. Bergland. Fast Fourier transform hardware implementations — A survey. *IEEE Transactions on Audio and Electroacoustics*, 17:109–119, 6 1969.

[74] Gilbert Louis Bernstein, Chinmayee Shah, Crystal Lemire, Zachary Devito, Matthew Fisher, Philip Levis, and Pat Hanrahan. Ebb. *ACM Transactions on Graphics*, 35:1–12, 5 2016.

[75] Luca Bertaccini, Luca Benini, and Francesco Conti. To buffer, or not to buffer? A case study on FFT accelerators for ultra-low-power multicore clusters. *ASAP*, 2021.

[76] Somashekaracharya G Bhaskaracharya, Julien Demouth, and Vinod Grover. Automatic kernel generation for Volta tensor cores. 2020. Available at `https://arxiv.org/pdf/2006.12645.pdf`.

[77] BigZetta Systems. Accelerating business intelligence. Available at `http://bigzetta.com/BigZetta_whitepaper.pdf`.

[78] Francesco Biletta. Automotive radar processing optimization by exploiting the hardware accelerator of radar sensor chip. Technical report, Politecnico di Torino, 2021.

[79] David Binkley. Source code analysis: A road map. *Future of Software Engineering (FOSE '07)*, 5 2007.

[80] Daniel Bittman, Robert Soule, Ethan L Miller, Vishal Shrivastav, Pankaj Mehra, Matthew Boisvert, Avi Silberschatz, and Peter Alvaro. Don't let RPCs constrain your API. *HotNets*, 2021.

[81] Gabriel Hjort Blindell. *Universal Instruction Selection*. PhD thesis, KTH Royal Institute of Technology, 2018.

[82] Chunkun Bo. *Automata Processing: from Application Acceleration to Hardware Design*. PhD thesis, University of Virginia, 2019.

[83] Chunkun Bo, Vinh Dang, Ted Xie, Jack Wadden, Mircea Stan, and Kevin Skadron. Automata processing in reconfigurable architectures. *ACM Transactions on Reconfigurable Technology and Systems*, 12:1–25, 6 2019.

[84] James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. Optimizing synthesis with metasketches. page 775–788, 2016.

[85] Andrew Boutros, Sadegh Yazdanshenas, and Vaughn Betz. You cannot improve what you do not measure: FPGA vs ASIC efficiency gaps for convolutional neural network inference. *Transactions on Reconfigurable Technology and Systems*, 11(3), 12 2018.

[86] Ian Briggs and Pavel Panchekha. Faster math functions, soundly. *CoRR 2021*, 2021. Availabe at `https://arxiv.org/abs/2107.05761`.

[87] Iulian Brumar, Georgios Zacharopoulos, Yuan Yao, Saketh Rama, Gu-Yeon Wei, and David Brooks. Early DSE and automatic generation of coarse grained merged accelerators. *CoRR*, 2021. Available at `https://arxiv.org/pdf/2111.09222.pdf`.

[88] Erik Brunvand, Donald Kline, and Alex K Jones. Dark silicon considered harmful: A case for truly green computing. *International Green and Sustainable Computing Conference*, 2018.

[89] Lutz Buch and Artur Andrzejak. Learning-based recursive aggregation of abstract syntax trees for code clone detection. *SANER*, 2 2019.

[90] Wooseok Byun, Minkyu Je, and Ji-Hoon Kim. An energy-efficient domain-specific reconfigurable array processor with heterogeneous pes for wearable brain-computer interface socs. *Transactions on Circuits and Systems*, 2022.

[91] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. *OSDI*, 2008.

[92] Brad Calder, Peter Feller, and Alan Eustae. Value profiling. *Micro*, pages 259–269, 1997.

[93] Michael Canesche, Westerly Carvalho, Lucas Reis, Metheus Oliveira, Salles Magalhaes, Peter Jamieson, Jaugusto M Nacif, and Richardo Ferreira. You only traverse twice: A YOTT placement, routing, and timing approach for CGRAs. *ACM Transactions on Embedded Computing Systems*, 20:1–25, 2021.

[94] Andrew Canis, Jongsok Choi, Blair Fort, Ruolong Lian, Qijing Huang, Nazanin Calagar, Marcel Gort, Jia Jun Qin, Mark Aldham, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. From software to accelerators with legup high-level synthesis. *2013 International Conference on Compilers, Architecture and Synthesis for Embedded Systems (CASES)*, 9 2013.

[95] C. Cascaval, S. Chatterjee, H. Franke, K. J. Gildea, and P. Pattnaik. A taxonomy of accelerator architectures and their programming models. *IBM Journal of Research and Development*, 54:5:1–5:10, 9 2010.

[96] Matthew Casias, Kevin Angstadt, Tommy Tracy II, Kevin Skadron, and Westley Weimer. Debugging support for pattern-matching languages and accelerators. *ASPLOS 2019: Architectural Support for Programming Languages and Operating Systems*, 4 2019.

[97] CAST. CAMFE: Camera front-end processor. Available at `https://www.cast-inc.com/interfaces/image-sensors/camfe/`.

[98] CAST. JPEG-D-S: Baseline JPEG decoder. Available at `https://www.cast-inc.com/compression/jpeg-jpeg-ls-image-compression/jpeg-d-s/`.

[99] CAST. JPEG-E-S: Baseline JPEG encoder. Available at `https://www.cast-inc.com/pdfs/cast_jpeg-e-s-brief-xilinx.pdf`.

[100] CAST. UDPIP-100G: 100G UDP/IP hardware protocol stack. Available at https://www.cast-inc.com/pdfs/cast_udpip-100g-brief-asic.pdf.

[101] CAST. WDR: Low-power, low-latency HDR/WDR image processor. Available at https://www.cast-inc.com/pdfs/cast_wdr-brief.pdf.

[102] CAST. ZipAccel-C: GZIP/ZLIB/Deflate data compression. Available at https://www.cast-inc.com/compression/gzip-lossless-data-compression/zipaccel-c/.

[103] Oscar Castaneda, Maria Bobbett, Alexandra Gallyas-Sanhueza, and Christoph Studer. PPAC: A versatile in-memory accelerator for matrix-vector-product-like operations. *ASAP*, 2019.

[104] Adrian M. Caulfield, Eric S. Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, Daniel Lo, Todd Massengill, Kalin Ovtcharov, Michael Papamichael, Lisa Woods, Sitaram Lanka, Derek Chiou, and Doug Burger. A cloud-scale acceleration architecture. pages 7:1–7:13, 2016.

[105] Milan Ceska, Vojtech Havlena, Lukás Holík, Jan Korenek, Ondrej Lengál, Denis Matousek, Jirí Matousek, Jakub Semric, and Tomás Vojnar. Deep packet inspection in FPGAs via approximate nondeterministic automata. *CoRR*, 2019. Available at http://arxiv.org/abs/1904.10786.

[106] Andre Xian Ming Chang, Parth Khopkar, Bashar Romanous, Abhishek Chaurasia, Patrick Estep, Skyler Windh, Doug Vanesko, Sheik Dawood Beer Mohideen, and Eugenio Culurciello. Reinforcement learning approach for mapping applications to dataflow-based coarse-grained reconfigurable array. *CoRR*, 2022. Available at https://arxiv.org/abs/2205.13675.

[107] Wei-Hsin Chang and Q Nguyen, Truong. On the fixed-point accuracy analysis of FFT algorithms. *IEEE Transactions on Signal Processing*, 56(10), 2008.

[108] G Charitopoulos, I Papaefstathiou, and D N Pnevmatikatos. Creating customized CGRAs for scientific applications. *Electronics*, 10, 2021.

[109] Samit Chaudhuri and Asmus Hetzel. SAT-based compilation to a non-von neumann processor. *ICCAD*, 2017.

[110] Lorenzo Chelini, Andi Drebes, Oleksandr Zineko, Albert Cohen, Nicolas Vasilache, Tobias Frosser, and Henk Corporaal. Progressive raising in multi-level IR. *CGO*, 2021.

[111] Jiyang Chen, Yuanwu Lei, Yuanxi Peng, Tingting He, and Ziye Deng. Configurable floating-point FFT accelerator on FPGA based multiple-rotation CORDIC. *Chinese Journal of Electronics*, 25(6), 2016.

[112] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. TVM: An automated end-to-end optimizing compiler for deep learning. pages 578–594, October 2018.

[113] Xiaowen Chen, Yuanwu Lei, Zhonghai Lu, and Shuming Chen. A variable-size FFT hardware accelerator based on matrix transposition. *VLSI*, 26:1953–1966, 10 2018.

[114] Xinyu Chen, Ronak Bajaj, Yao Chen, Jiong He, Bingsheng He, Weng-Fai Wong, and Deming Chen. On-the-fly parallel data shuffling for graph processing on OpenCL-based FPGAs. 2019.

[115] Xiao Cheng, Lingxiao Jiang, Hao Zhong, Haibo Yu, and Jianjun Zhao. On the feasibility of detecting cross-platform code clones via identifier similarity. *Proceedings of the 5th International Workshop on Software Mining — SoftwareMining 2016*, 9 2016.

[116] S. Alexander Chin, Noriaki Sakamoto, Allan Rui, Jim Zhao, Jin Hee Kim, Yuko Hara-Azumi, and Jason Anderson. CGRA-ME: a unified framework for CGRA modelling and exploration. *2017 IEEE 28th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 7 2017.

[117] Benjamin Y Cho, Jaegeun Jung, and Mattan Erez. Accelerating bandwidth-bound deep learning inference with main-memory accelerators. *SC*, 2021.

[118] Muslim Chochlov, Michael English, Jim Buckley, Daniel Ilie, and Maria Scanlon. [engineering Paper] Identifying feature clones in a suite of systems. *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 9 2018.

[119] Lauranne Choquin and Fred Piry. Arm custom instructions: Enabling innovation and greater flexibility on Arm. 2020. Available at https://armkeil.blob.core.windows.net/developer/Files/pdf/white-paper/arm-custom-instructions-wp.pdf#:~:text=What%20are%20Arm%20Custom%20Instructions%3F%20Arm%20Custom%20Instructions,Cortex-M33%20processor.%20In%202021%2C%20Arm%20Custom%20Instructions%20will.

[120] Zamshed I Chowdhury, Masoud Zabihi, S Karen Khatamifard, Zhengyang Zhao, Salonik Resch, Meisam Razaviyayn, Jian-Ping Wang, Sachin S Sapatnekar, and Ulya R Karpuzcu. A DNA read alignment accelerator based on computational RAM. *Journal on Exploratory Solid-State Computational Devices and Circuits*, 2020.

[121] Eleanor Chu and Alan George. *Inside the FFT Black Box: Serial and Parallel Fast Fourier Transform Algorithms*. CRC Press, 1999.

[122] Husrev Cilasun, Salonik Resch, Zamshed Iqbal Chowdhury, Erin Olson, Masoud Zabihi, Zhengyang Zhao, Thomas Peterson, Jian-Ping Wang, Sachin S. Sapatnekar, and Ulya Karpuzcu. CRAFFT: High resolution FFT accelerator in spintronic computational RAM. *DAC*, 7 2020.

[123] Nathan Clark, Wilkin Tang, and Scott Mahlke. Automatically generating custom instruction set extensions. *Workshop on Application-Specific Processors*, 2002.

[124] Valentin Clement, Sylvaine Ferrachat, Oliver Fuhrer, Xavier Lapillonne, Carlos E. Osuna, Robert Pincus, Jon Rood, and William Sawyer. The claw dsl. *Proceedings of the Platform for Advanced Scientific Computing Conference*, 7 2018.

[125] Lucian Codrescu, Willie Anderson, Suresh Venkumanhanti, Mao Zheng, Erich Plondke, Chris Koob, Ajay Ingle, Charles Tabony, and Rick Maule. Hexagon DSP: An architecture optimized for mobile multimedia and communications. *IEEE Micro 2014*, 2014.

[126] Bruce Collie, Philip Ginsbach, and Michael F.P. O'Boyle. Type-directed program synthesis and constraint generation for library portability. *PACT*, 2019.

[127] Bruce Collie, Philip Ginsbach, Jackson Woodruff, Ajitha Rajan, and Michael F.P. O'Boyle. M3: Semantic API migration. *ASE*, 2020.

[128] Bruce Collie and Michael F P O'Boyle. Program lifting using gray-box behaviour. *PACT*, 2021.

[129] Bruce Collie, Jackson Woodruff, and Michael FP O'Boyle. Modeling black-box components with probabilitic synthesis. *GPCE*, 2020.

[130] Alessandro Comodi, Davide Conficconi, Alberto Scolari, and Marco D. Santambrogio. Tirex: Tiled regular expression matching architecture. *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 5 2018.

[131] Jason Cong, Mohammad Ali Ghodrat, Michael Gill, Beayna Grigorian, Hui Huang, and Glenn Reinman. Composable accelerator-rich microprocessor enhanced for adaptivity and longevity. *Symposium on Low Power Electronics and Design*, 2013.

[132] James W Cooley and John W Tukey. An algorithm for the machine calculation of complex Fourier series. *Math. Comput.*, (19):297–301, 1965.

[133] Keith D. Cooper, L. Taylor Simpson, and Christopher A. Vick. Operator strength reduction. *ACM Trans. Program. Lang. Syst.*, 23(5):603–625, sep 2001.

[134] H Corporaal and E de Bruin. What is a CGRA? *ACM/SIGDA e-newsletter*, 2023.

[135] Meghan Cowan, Thierry Moreau, Tianqi Chen, James Bornholt, and Lius Ceze. Automatic generaiton of high-performance quantized machine learning kernels. *CGO*, 2020.

[136] Samuel Coward, George A Constantinides, and Theo Drane. Automatic datapath optimizaiton using e-graphs. *IEEE 29th Symposium on Computer Arithmetic*, 2022.

[137] CTAccel. CTAccel image processng (CIP) accelerator for Alveo U200. Available at `http://www.ct-accel.com/home-page/cip-for-alveo-u200/`.

[138] Chris Cummins, Zacharias V. Fisches, Tal Ben-Nun, Torsten Hoefler, and Hugh Leather. PROGRAML: A graph-based program representation for data flow analysis and compiler optimizations. *ICML*, 2021.

[139] Anderson Faustino da Silva, Bruno Conde Kind, Jose Wesley de Souza Magalhaes, Jeronimo Nunes Rocha, Breno Campos Ferreira Guimaraes, and Fernando

Magno Quintao Pereira. AnghaBench: a suite with one million compilable C benchmarks for code-size reduction. *CGO*, 2021.

[140] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. Towards general purpose acceleration by exploiting common data-dependence forms. *MICRO*, 10 2019.

[141] Lei Dai, Ying Wang, Cheng Liu, Fuping Li, Huawei Li, and Xiaowei Li. Reexamining CGRA memory sub-system for higher memory utilization and performance. *ICCD*, 2022.

[142] William J Dally, James Balfour, David Black-Shaffer, James Chen, Curtis Harting, Vishal Parikh, Jongsoo Park, and David Sheffield. Efficient embedded computing. *IEEE Computer*, 2008.

[143] William J. Dally, Yatish Turakhia, and Song Han. Domain-specific hardware accelerators. *Communications of the ACM*, 63:48–57, 6 2020.

[144] Anthony Danalis, Gabriel Marin, Collin McCurdy, Jeremy S. Meredith, Philip C. Roth, Kyle Spafford, Vinod Tipparaju, and Jeffrey S. Vetter. The scalable heterogeneous computing (SHOC) benchmark suite. *Proceedings of the 3rd Workshop on General-Purpose Computation on Graphics Processing Units - GPGPU '10*, 3 2010.

[145] Shail Dave, Mahesh Balasubramanian, and Aviral Shrivastava. RAMP: Resource-aware mapping for CGRAs. *DAC*, 2018.

[146] Joao P L De Carvalho, Braedy Kuzma, Ivan Korostelev, Jose Nelson Amaral, Christopher Barton, Jose Moreira, and Guido Araujo. KernelFaRer: Replacing native-code idioms with high-performance library calls. *TACO*, 2021.

[147] Joao Paulo C de Lima, Paulo Cesar Cantos, Marco A Z Alves, Antonio C S Beck, and Luigi Carro. Design space exploration for pim architectures in 3d-stacked memories. *CF*, 2018.

[148] João Paulo Cardoso de Lima, Marcelo Brandalero, Michael Hübner, and Luigi Carro. Stap: An architecture and design tool for automata processing on memristor tcams. *ACM Journal on Emerging Technologies in Computing Systems*, 18:1–22, 4 2022.

[149] Daniel DeFreez, Aditya V. Thakur, and Cindy Rubio-González. Path-based function embedding and its application to error-handling specification mining. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering - ESEC/FSE 2018*, 11 2018.

[150] Florian Deissenboeck, Lars Heinemann, Benjamin Hummel, and Stefan Wagner. Challenges of the dynamic detection of functionally similar code fragments. *2012 16th European Conference on Software Maintenance and Reengineering*, 3 2012.

[151] deneb f:Genetics. DENEB genetics white paper, 2019. Available at `https://drive.google.com/file/d/1pqXUPsfCid6ZMycTpGsG0DgHDqWTwVPg/view`.

[152] Benoit W Denkinger, Miguel Quiros-Peon, Mario Konijnenburg, David Atienza, and Francky Catthoor. VWR2A: A very-wide-register reconfigurable-array architecture for low-power embedded devices. *DAC*, 2022.

[153] Alexandar Devic, Siddhartha Balakrishna Rai, Anand Sivasubramaniam, Ameen Akel, Sean Eilert, and Justin Eno. To PIM or not for emerging general purpose processing in DDR memory systems. *ISCA*, 2022.

[154] Roberto DiCecco, Griffin Lacey, Jasmina Vasiljevic, Paul Chow, Taylor Graham, and Shawki Areibi. Caffeinated FPGAs: FPGA framework for convolutional neural networks. *FPT*, 2016.

[155] Danny Dig and Ralph Johnson. How do APIs evolve? a story of refactoring. *SMR*, 18:83–107, 2006.

[156] Steven H.H̃. Ding, Benjamin C.M̃. Fung, and Philippe Charland. Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. *2019 IEEE Symposium on Security and Privacy (SP)*, 5 2019.

[157] Paul Dlugosch, Dave Brown, Paul Glendenning, Michael Leventhal, and Harold Noyes. An efficient and scalable semiconductor architecture for parallel automata processing. *IEEE PDS*, 25:3088–3098, 12 2014.

[158] Stefan Döbrich and Christian Hochberger. Exploring online synthesis for CGRAs with specialized operator sets. *International Journal of Reconfigurable Computing*, 2011.

[159] Jens Domke, Emil Vatai, Aleksandr Drozd, Peng Chen, Yosuke Oyama, Lingqi Zhang, Shweta Salaria, Daichi Mukunoki, Artur Podobas, Mohamed Wahib, and Satoshi Matsuoka. Matrix engines for high performance computing: A paragon of performance or grasping at straws? *IEEE PDPS*, 2021.

[160] Bo Duan, Wendi Wang, Xingjian Li, Chunming Zhang, Peiheng Zhang, and Ninghui Sun. Floating-point mixed-radix FFT core generation for FPGA and comparison with GPU and CPU. *FPT*, 12 2011.

[161] P Duhamel and M Vetterli. Fast Fourier transforms: A tutorial review and a state of the art. *Signal Processing*, pages 259–299, 1990.

[162] Sultan Durrani, Muhammad Saad Chughtai, Mert Hidayetoglu, Rashid Tahir, Abdul Dakkak, Lawrence Rauchwerger, Fareed Zaffar, and Wen-mei Hwu. Accelerating fourier and number theoretic transforms using tensor cores and warp shuffles. *PACT*, 2021.

[163] Arpan Dutta, Saransh Gupta, Khaleghi Behnam, Rishikanth Chandrasekaran, Weihong Xo, and Tajana Rosing. HDnn-PIM: Efficient in memory design of hyperdimensional computing with feature extraction. *Great Lakes Symposium on VLSI*, 2022.

[164] Zahra Ebrahimi and Akash Kumar. BioCare: An energy-efficient CGRA for bio-signfal processing at the edge. *ISCASS*, 2021.

[165] Bernhard Egger, Hochan Lee, Duseok Kang, Mansureh S. Moghaddam, Youngchul Cho, Yeonbok Lee, Sukjin Kim, Soonhoi Ha, and Kiyoung Choi. A space- and energy-efficient code compression/decompression technique for coarse-grained reconfigurable architectures. *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2 2017.

[166] Venmugil Elango, Norm Rubin, Mahesh Ravishankar, Hariharan Sandanagobalane, and Vinod Grover. Diesel: Dsl for linear algebra and neural net computations on gpus. *Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages - MAPL 2018*, 6 2018.

[167] Joel S Emer, Vivienne Sze, and Yannan Nellie Wu. An architecture-level energy and area estimator for processing-in-memory accelerator designs. *ISPASS*, 2020.

[168] Emoshape. Emotion processing unit III, 2018. Available at `https://emoshape.com/wp-content/uploads/2019/04/EPU-III-Brochure.pdf`.

[169] Kento Emoto, Kiminori Matsuzaki, Zhenjiang Hu, Akimasa Morihata, and Hideya Iwasaki. Think like a vertex, behave like a function! a functional dsl for vertex-centric big graph processing. *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, 9 2016.

[170] H. Esmaeilzadeh, E. Blem, R. S. Amant, K. Sankaralingam, and D. Burger. Dark silicon and the end of multicore scaling. *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 365–376, June 2011.

[171] Hadi Esmaeilzadeh, Emily Blem, Renée St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Power challenges may end the multicore era. *Communications of the ACM*, 56:93, 2 2013.

[172] Vinicius Espindola, Luciano Zago, Herve Yviquel, and Guido Araujo. Source matching and rewriting for MLIR using string-based automata. *TACO*, 2022.

[173] Thomas Faingnaert, Tim Besard, and Bjorn De Sutter. Flexible performant GEMM kernels on GPUs. *IEEE Transactions on Parallel and Distributed Systems*, 2021.

[174] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. Functional code clone detection with syntax ans semantics fusion learning. *ISSTA*, 2020.

[175] Yuanwei Fang, Tung T. Hoang, Michela Becchi, and Andrew A. Chien. Fast support for unstructured data processing. *Proceedings of the 48th International Symposium on Microarchitecture — MICRO-48*, 12 2015.

[176] None Fang-Hsiang Su, Jonathan Bell, Gail Kaiser, and Simha Sethumadhavan. Identifying functionally similar code in complex codebases. *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*, 5 2016.

[177] Umer Farooq, Leon Welicki, and Dieter Zirkler. API usability peer reviews: A method for evaluating the usability of application programming interfaces. *CHI*, 2010.

[178] Matthew Feldman. *Software-Defined Hardware Without Sacrificing Performance*. PhD thesis, Stanford University, 2021.

[179] Basilio B Fraguela, Jose Renau, Paul Feautrier, David Padua, and Josep Torrellas. Programming the FlexRAM parallel intelligent memory system. *PPoPP*, 2003.

[180] Björn Franke. C compilers and code optimization for DSPs. *Handbook of Signal Processing Systems*, pages 575–601, 7 2010.

[181] Florian Fricke, Andre Werner, Keyvan Shahin, and Michael Heubner. CGRA tool flow for fast run-time reconfiguration. *ARC*, 2018.

[182] Matteo Frigo. A fast Fourier transform compiler. *ACM SIGPLAN Notices*, 34:169–180, 5 1999.

[183] Zhoulai Fu and Zhendong Su. XSat: A fast floating-point satisfiability solver. *Computer Aided Verification*, pages 187–209, 2016.

[184] Adi Fuchs and David Wentzlaff. The accelerator wall: Limits of chip specialization. *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2 2019.

[185] Brian Gaide, Dinesh Gaitonde, Chirag Ravishankar, and Trevor Bauer. Xilinx adaptive compute acceleration platform. *Proceedings of the 2019 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2 2019.

[186] Yi Gao, Zan Wang, Shuang Liu, Lin Yang, Wei Sang, and Yuanfang Cai. TECCD: A tree embedding approach for code clone detection. *ICSME*, 2019.

[187] Mario Garrido. A survey on pipelined FFT hardware architectures. *Journal of Signal Processing Systems*, 2021.

[188] Tong Geng, Chunshu Wu, Cheng Tan, Bo Fang, Ang Li, and Martin Herbordt. CQNN: a CGRA-based QNN framework. *HPEC*, 2020.

[189] Philip Ginsbach. *From Constraint Programming to Heterogeneous Parallelism*. PhD thesis, University of Edinburgh, 2019.

[190] Philip Ginsbach, Bruce Collie, and Michael F. P. O'Boyle. Automatically harnessing sparse acceleration. *CC*, 2 2020.

[191] Philip Ginsbach, Toomas Remmelg, Michel Steuwer, Bruno Bodin, Christophe Dubach, and Michael F. P. O'Boyle. Automatic matching of legacy code to heterogeneous APIs. *ASPLOS*, 3 2018.

[192] Graham Gobieski, Ahmet Oguz Atli, Kenneth Mai, Brandon Lucia, and Nathan Beckmann. Snafu: An ultra-low-power, energy-minimal CGRA-generation framework and architecture. *ISCA*, 2021.

[193] Graham Gobieski, Souradip Ghosh, Tony Nowatzki, Todd C Mowry, Nathan Beckmann, and Brandon Lucia. Riptide: A programmable, energy-minimal dataflow compiler and architecture. *Micro*, 2022.

[194] Vaibhav Gogte, Aasheesh Kolli, Michael J. Cafarella, Loris D'Antoni, and Thomas F. Wenisch. HARE: Hardware accelerator for regular expressions. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 10 2016.

[195] Manil Dev Gomony, Floran de Putter, Anteneh Gebregiorgis, Gianna Paulin, Linyan Mei, Vikram Jain, Said Hamdioui, Victor Sanchez, Tobias Grosser, Marc Geilen, Marian Verhelst, Friedemann Zenke, Frank Gurkaynak, Carry de Bruin, Sander Stuijk, Simon Davidson, Sayandip De, Mounir Ghogho, Alexandra Jimborean, Sherif Eissa, Luca Benini, Dimitrios Soudris, Rajendra Bishnoi, Sam Ainsworth, Federico Corradi, Ousassim Karrakchou, Tim Guneysu, and Henk Corporaal. CONVOLVE: Smart and seamless design of smart edge processors. *CoRR*, 2023. Available at shttps://arxiv.org/pdf/2212.00873.pdf.

[196] Mathias Gottschlag, Peter Brantsch, and Frank Bellosa. Automatic core specialization for avx-512 applications. *Proceedings of the 13th ACM International Systems and Storage Conference on ZZZ*, 5 2020.

[197] N Goulding-Hotta, J Sampson, G Venkatesh, S Garcia, J Auricchio, P Huang, M Arora, S Nath, V Bhatt, J Babb, S Swanson, and M Taylor. The GreenDroid mobile application processor: An architecture for silicon's dark future. *IEEE Micro*, 31:86–95, 3 2011.

[198] Venkatraman Govindaraju, Chen-Han Ho, Tony Nowatzki, Jatin Chhugani, Nadathur Satish, Karthikeyan Sankaralingam, and Changkyu Kim. DySER: Unifying functionality and parallelism specialization for energy-efficient computing. *Micro*, 2012.

[199] Petr Grachev, Roman Bezborodov, Ivan Smetannikov, and Andrey Filchenkov. Exploring the relationship between the structural and the actual similarities of automata. *Proceedings of the 3rd International Conference on Machine Learning and Soft Computing - ICMLSC 2019*, 1 2019.

[200] Gregor Gramlich and Georg Schnitger. Minimizing NFA's and regular expressions. *STACS 2005*, 2005.

[201] Grovf. GCache. Available at `https://grovf.com/products/gcache`.

[202] Grovf. GRegeX. Available at `https://grovf.com/products/gregex`.

[203] Grovf. MonetX. Available at `https://grovf.com/products/monetx`.

[204] Grovf. MongoDB acceleration using Grovf's MonteX platform, 2019. Available at `https://grovf.com/useCases/mongodb-acceleration-using-grovfs-monetx-platform`.

[205] Grovf. Grovf extends its security IP protfolio by probabilistic search engine, 2020. Available at `'https://grovf.com/news/cyber-security-probabilistic-search-engine-ip-lawful-interception'`.

[206] Qi Guo, Nikoaos Alachiotis, Berkin Akin, Fazle Sadi, Guanglin Xu, Tze Meng Low, Larry Pileggi, James C Hoe, and Franz Franchetti. 3D-stacked memory-side acceleration: Accelerator and system design. *Workshop on Near-Data Processing*, 2014.

[207] Yijiang Guo and Guojie Luo. Pillars: An integrated CGRA design framework. *WOSET*, 2020.

[208] Udit Gupta, Mariam Elgamal, Gage Hills, Gu-Yeon Wei, Hsien-Hsin S Lee, David Brooks, and Carole-Jean Wu. ACT: Designing sustainable computer systems with an architectural carbon modeling tool. *ISCA*, 2022.

[209] Matthew R Guthaus, Jeffrey S Ringenberg, and Dan Ernst. MiBench: A free, comercially representative embedded benchmark suite. *Workshop on Workload Characterization*, pages 3–14, 2001.

[210] Tae Jun Ham, Lisa Wu, Narayanan Sundaram, Nadathur Satish, and Margaret Martonosi. Graphicionado: A high-performance and energy-efficient accelerator

for graph analytics. *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 10 2016.

[211] Mahdi Hamzeh, Aviral Shrivastava, and Sarma Vrudhula. EPIMap: Using epimorphism to map applications on CGRAs. *DAC*, 2012.

[212] Xueyu Han, Jiajia Chen, Boyu Qin, and Susanto Rahardja. A novel area-power efficient design for approximated small-point FFT architecture. *CADICS*, 39:4816–4827, 12 2020.

[213] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. *CGO*, 4 2011.

[214] W.H. Harrison. Compiler analysis of the value ranges for variables. *IEEE Transactions on Software Engineering*, SE-3:243–250, 5 1977.

[215] J.R. Hauser and J. Wawrzynek. Garp: a MIPS processor with a reconfigurable coprocessor. *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, 1997.

[216] Xincheng He, Lei Xu, Xiangyu Zhang, Yang Feng, and Baowen Xu. PyART: Python API recommendation in real-time. *ICSE*, 2021.

[217] John L Hennessy. The 50 year history of the microprocessor as five technology eras. *IEEE Micro*, 2021.

[218] Michi Henning. API design matters. *Communications of the ACM*, 52(5):46–56, 2009.

[219] Paul M Heysters, Gerard K Rauwerda, and Lodewijk T Smit. A flexible, low power, high performance DSP IP core for programmable systems-on-chip. *Design and Reuse*, 2005.

[220] Tran Trung Hieu, Tran Ngoc Thinh, and Shigenori Tomiyama. ENREM: An efficient NFA-based regular expression matching engine on reconfigurable hardware for NIDS. *Journal of Systems Architecture*, 59:202–212, 4 2013.

[221] Yoshiki Higo and Shinji Kusumoto. How should we measure functional sameness from program source code? An exploratory study on Java methods. *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering — FSE 2014*, 11 2014.

[222] Rahul K. Hiware and Dinesh Padole. Configuration memory based dynamic coarse grained reconfigurable multicore architecture for 8 point fft. *ICETET*, 11 2015.

[223] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. CC2Vec: Distributed representations of code changes. *ICSE 2020*, 2020.

[224] Sara Hooker. The hardware lottery. *Communications of the ACM*, 2021.

[225] Kenneth Hoste, Aashish Phansalkar, Lieven Eeckhout, Andy Georges, Lizy K. John, and Koen De Bosschere. Performance prediction based on inherent program similarity. *Proceedings of the 15th international conference on Parallel architectures and compilation techniques - PACT '06*, 9 2006.

[226] Tseng Hung-Wei Hsu, Kuan-Chieh. Accelerating applications using edge tensor processing units. *SC*, 2021.

[227] Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Thierry Tambe, Gus Henry Smith, Akash Gaonkar, Vishal Canumalla, Gu-Yeon Wei, Aarti Gupta, Zachary Tatlock, and Sharad Malik. Specialized accelerators and compiler flows: Replacing accelerator APIs with a formal software/hardware interface. *PLDI*, 2022.

[228] Bo-Yuan Huang, Steven Lyubomirsky, Yi Li, Mike He, Thierry Tambe, Gus Henry Smith, Akash Gaonkar, Vishal Canumalla, Gu-Yeon Wei, Aarti Gupta, Zachary Tatlock, and Sharad Malik. Specialized accelerators and compiler flows: Replacing accelerator apis with a formal software/hardware interface, 2022.

[229] Qiao Huang, Xin Xia, Zhenchang Xing, David Lo, and Xinyu Wang. API method recommendation without worrying about the task-API knowledge gap. *ASE*, 9 2018.

[230] Yi Huang, Zhiyu Chen, Dai Li, and Kaiyuan Yang. CAMA: energy and memory efficient automata processing in content-addressable memories. *CoRR*, 2021.

[231] Lily Hugerich, Apporv Shukla, and Georgios Smaragdakis. No-hop: In-network distributed hash tables. *ANCS*, 2021.

[232] Yuka Ikarashi, Gilbert Bernstein, Alex Reinking, Hasan Genc, and Jonathan Ragan-Kelley. Exocompilation for productive programming of hardware accelerators. *PLDI*, 2022.

[233] Lucian Ilie, Gonzalo Navarro, and Sheng Yu. *On NFA Reductions*, pages 112–124. Springer Berlin Heidelberg, 2004.

[234] Mohsen Imani, Saransh Gupta, and Tajana Rosing. GenPIM: Generalized processing in-memory to accelerate data intensive applications. *DATE*, 2018.

[235] Texas Instruments. Keystone architecture network coprocessor (NETCP). 2010. Available at `www.ti.com/lit/ug/sprugz6/sprugz6.pdf`.

[236] Intel. Delivering the visual cloud. faster. https://www.intel.co.uk/content/www/uk/en/products/docs/servers/accelerators/vca2-product-brief.html, 2018.

[237] Intel. Advanced settings for intel ethernet 10 gigabit server adapters, 2019.

[238] Tomoya Ishihara, Keisuke Hotta, Yoshiki Higo, Hiroshi Igaki, and Shinji Kusumoto. Inter-project functional clone detection toward building libraries — an empirical study on 13,000 projects. *2012 19th Working Conference on Reverse Engineering*, 10 2012.

[239] Won Seob Jeong, Changmin Lee, Keunsoo Kim, Myung Kik Yoon, Won Jeon, Myouhgsoo Jung, and Won Woo Ro. REACT: Scalable and high-performance regular expression pattern matching accelerator for in-storage processing. *IEEE Transactions on Parallel and Dsitributed Systems*, 31(5), 2020.

[240] Theo Jepsen, Daniel Alvarez, Nate Foster, Changhoon Kim, Jeongkeun Lee, Masoud Moshref, and Robert Soulé. Fast string searching on PISA. *Proceedings of the 2019 ACM Symposium on SDN Research*, 4 2019.

[241] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. DECKARD: Scalable and accurate tree-based detection of code clones. *ICSE*, 5 2007.

[242] Lingxiao Jiang and Zhendong Su. Automatic mining of functionally equivalent code fragments via random testing. *Proceedings of the eighteenth international symposium on Software testing and analysis - ISSTA '09*, 7 2009.

[243] Renhe Jiang, Zhengzhao Chen, Zejun Zhang, Yu Pei, Minxue Pan, and Tian Zhang. [Research Paper] Semantics-based code search using input/output examples. *SCAM*, 9 2018.

[244] Hai Jin, Cong Liu, Haikun Liu, Ruikun Luo, Jiahong Xu, Fubing Mao, and Xiaofei Liao. ReHy: A ReRAM-based digital/analog hybrid PIM architecture for accelerating CNN training. *Transactions on Parallel and Distributed Systems*, (11), November 2022.

[245] Ajay Joshi, Aashish Phansalkar, L. Eeckhout, and L.K. John. Measuring benchmark similarity using inherent program characteristics. *IEEE Transactions on Computers*, 55:769–782, 6 2006.

[246] Norman P. Jouppi, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Cliff Young, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Nishant Patil, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, David Patterson, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Gaurav Agrawal, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Raminder Bajwa, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Sarah Bates, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, Doe Hyun Yoon, Suresh Bhatia, and Nan Boden. In-datacenter performance analysis of a tensor processing unit. *Proceedings of the 44th Annual International Symposium on Computer Architecture - ISCA '17*, 6 2017.

[247] Oren Kalinsky, Benny Kimelfeld, and Yoav Etsion. The TrieJax architecture: Accelerating graph operations through relational joins. *ASPLOS*, 2020.

[248] H. Kalva. The H.264 video coding standard. *IEEE Multimedia*, 13:86–90, 10 2006.

[249] Shoaib Kamil, Alvin Cheung, Shachar Itzhaky, and Armando Solar-Lezama. Verified lifting of stencil computations. *PLDI*, 2016.

[250] Anil Kanduri, Amir Rahmani, Ahmed Hemani, Axel Jantsch, and Hannu Tenhuenen. *A Perspective on Dark Silicon*. Springer, 2017.

[251] Hong Jin Kang, Tegawende F. Bissyande, and David Lo. Assessing the generalizability of Code2Vec token embeddings. *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, November 2019.

[252] Sheng-Chun Kao, Geonhwa Jeong, and Tushar Krishna. Confuciox: Autonomous hardware resource assignment for DNN accelerators using reinforcement learning. *Micro*, 2020.

[253] Puneet Kapur, Brad Cossette, and Robert J Walker. Refactoring references for library migration. *OOPSLA*, 2010.

[254] Rasha Karakchi, Charles Daniels, and Jason Bakos. An overlay architecture for pattern matching. *ASAP*, 7 2019.

[255] Manupa Karunaratne, Aditi Kulkarni Mohite, Tulika Mitra, and Li-Shiuan Peh. Hycube. *Proceedings of the 54th Annual Design Automation Conference 2017 on - DAC '17*, 6 2017.

[256] Manupa Karunaratne, Cheng Tan, Aditi Kulkarni, Tulika Mitra, and Li-Shiuan Pen. DNestMap: Mapping deeply-nested loops on ultra-low power CGRAs. *DAC*, 2018.

[257] Manupa Karunaratne, Dhananjaya Wijerathne, Tulika Mitra, and Li-Shiuan Peh. 4D-CGRA: Introducing branch dimension to spatio-temporal applicaiton mapping on CGRAs. *ICCAD*, 2019.

[258] David Kawrykow and Martin P. Robillard. Improving API usage through automatic detection of redundant code. *2009 IEEE/ACM International Conference on Automated Software Engineering*, 11 2009.

[259] Iman Keivanloo, Feng Zhang, and Ying Zou. Threshold-free code clone detection for a large-scale heterogeneous Java repository. *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 3 2015.

[260] Ronan Keryell, Andrew Gozillon, Gauthier Harnisch, Hyun Kwon, Ravikumar Chakaravarthy, and Ralph Wittig. SYCL for Xilinx Versal ACAP AIE CGRA. *IWOCL SYCLCon*, 2021.

[261] Marcus Kessel and Colin Atkinson. On the efficacy of dynamic behavior comparison for judging functional equivalence. *2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 9 2019.

[262] Christoph W. Kessler. Compiling for VLIW DSPs. *Handbook of Signal Processing Systems*, pages 979–1020, 10 2019.

[263] Artavazd Khachatryan. Personal communications, Grovf. 2020.

[264] Moein Khazraee, Ikuo Magaki, Luis Vega Gutierrez, and Michael Taylor. ASIC clouds: Specializing the datacenter. *IEEE Micro*, pages 1–1, 2017.

[265] Changmoo Kim, Mookyoung Chung, Yeongon Cho, Mario Konijnenburg, Soojung Ryu, and Jeongwook Kim. Ulp-srp: Ultra low power samsung reconfigurable processor for biomedical applications. *2012 International Conference on Field-Programmable Technology*, 12 2012.

[266] Heejung Kim, Yungbum Jung, Sunghun Kim, and Kwankeun Yi. MeCC. *ICSE11*, 5 2011.

[267] Jin Hyun Kim, Shin-haeng Kang, Sukhan Lee, Hyeonsu Kim, Woongjae Song, Yuhwan Ro, Seungwon Lee, David Wang, Hyunsung Shin, Phuah, Bengseng, Jihyun Choi, Jinin So, YeonGon Cho, JoonHo Song, Jandseok Choic, Jeongheyon Cho, Kyomin Sohn, Youngsoo Sohn, Park Kwangil, and Nam Sung Kim. Aquabolt-XL: Samsung HBM2-PIM with in-memory processing for ML accelerators and beyond. *Hot Chips 33 Symposium*, 2021.

[268] Wonsub Kim, Yoonseo Choi, and Jaehyun Kim. CGRA compilation boost up for acceleration of graphics. 2014.

[269] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ICLR*, 2015.

[270] Jakapong Klainongsuang, Sulistyo Nugroho, Yusuf, Hideaki Hata, Bundit Manaskasemsak, Arnon Rungsawang, Pattara Leelaprute, and Kenichi Matsumoto. Identifying algorithm names in code comments. *CoRR*, 2019. Available at `https://arxiv.org/abs/1907.04557`.

[271] Dmitrii Kochkov, Jamie A Smith, Ayya Alieva, Qing Wang, Michael P Brenner, and Stephan Hoyer. Machine learning-accelerated computational fluid dynamics. *Proceedings of the National Academy of Sciences*, 118, 2021.

[272] Thomas Koehler, Phil Trinder, and Michel Steuwer. Sketch guided equality-saturation. *CoRR*, 2022. Available at `https://arxiv.org/pdf/2111.13040.pdf`.

[273] David Koeplinger, Christos Kozyrakis, Kunle Olukotun, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, and Ardavan Pedram. Spatial: a language and compiler for application accelerators. *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation - PLDI 2018*, 6 2018.

[274] Lukas Kohutka, Lukas Nagy, and Viera Stopjakova. A novel hardware-accelerated priority queue for real-time systems. *Euromicro Conference on Digital System Design*, 2018.

[275] Takuya Kojima, Ayaka Ohwada, and Hideharu Amano. Mapping-aware kernel partitioning method for CGRAs assested by deep learning. *Parallel and Distributed Systems*, 33(5), May 2022.

[276] Shijn Kong, Randy Smith, and Cristian Estan. Efficient signature matching with multiple alphabet compression tables. *Secure Comms 2008*, 2008.

[277] Mingyang Kou, Jiangyuan Gu, Shaojun Wei, Hailong Yao, and Shouyi Yin. TAEM: Fast transfer-aware effective loop mapping for heterogeneous resources on CGRA. *DAC*, 2020.

[278] Mingyang Kou, Jun Zeng, Boxiao Han, Fei Xu, Jiangyuan Gu, and Hailong Yao. GEML: GNN-based efficient mapping method for large loop applications on CGRA. *DAC*, 2022.

[279] Smail Kourta, Adel Abderahmane Namani, Fatima Benbouzid-Si Tayeb, Kim Hazelwood, Chris Cummins, Hugh Leather, and Riyadh Baghdadi. Caviar: an e-graph based trs for automatic code optimization. *Proceedings of the 31st ACM SIGPLAN International Conference on Compiler Construction*, pages 54–64, 2022.

[280] Vlad Krasnov. On the dangers of Intel's frequency scaling. 2017. Available at https://blog.cloudflare.com/on-the-dangers-of-intels-frequency-scaling/.

[281] Jens Krinke and Chaiyong Ragkhitwetsagul. BigCloneBench considered harmful for machine learning. 2022.

[282] Anil Krishna, Timothy Heil, Nicholas Lindberg, Farnaz Toussi, and Steven VanderWiel. Hardware acceleration in the IBM PowerEN processor. *Proceedings of the 21st international conference on Parallel architectures and compilation techniques — PACT '12*, 9 2012.

[283] Venkata Krishnan, Olivier Serres, and Michael Blocksome. Configurable network protocol accelerator (copa). *IEEE Micro*, 41:8–14, 1 2021.

[284] Martin Kristien, Bruno Bodin, Michel Steuwer, and Christophe Dubach. High-level synthesis of functional patterns with lift. *ARRAY*, 6 2019.

[285] V. V. Kuliamin. Standardization and testing of implementations of mathematical functions in floating point numbers. *Programming and Computer Software*, 33(3):154–173, 2007.

[286] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to accelerator multiple regular expressions matching for deep packet inspection. *SIGCOMM 2006*, 2006.

[287] Snehasish Kumar, Nick Sumner, Vijayalakshmi Srinivasan, Steve Margerm, and Arrvindh Shriraman. Needle: Leveraging program analysis to analyze and extract accelerators from whole programs. *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2 2017.

[288] Snehasish Kumar, William N. Sumner, and Arrvindh Shriraman. SPEC-AX and PARSEC-AX: extracting accelerator benchmarks from microprocessor benchmarks. *2016 IEEE International Symposium on Workload Characterization (IISWC)*, 9 2016.

[289] Snehasish Kumar, Naveen Vedula, Arrvindh Shriraman, and Vijayalakshmi Srinivasan. Dasx: Hardware accelerator for software data structures. *ICS*, 2015.

[290] Iman Kundu, Edward Cottle, Florent Michel, Joseph Wilson, and Nick New. The dawn of energy efficient computing: Optically accelerating the fast Fourier transform core. *OSA*, 2021.

[291] Nate Kushman and Regina Barzilay. Using semantic unification to generate regular expressions from natural language. *North American Chapter of the Association for Computational Linguistics*, 2013.

[292] C. Lattner and V. Adve. Llvm: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pages 75–86, 2004.

[293] Chris Lattner, Mehdi Amini, Uday Bondhugla, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zineko. MLIR: A compiler infrastructure for the end of moore's law. *CoRR 2020*, 2020. Available at `https://arxiv.org/abs/2002.11054`.

[294] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. *PLDI*, 6 2007.

[295] Gary Lauterbach. The path to successful wafer-scale integration: The cerebras story. *IEEE Micro*, 2021.

[296] Maysam Lavasani, Hari Angepat, and Derek Chiou. An FPGA-based in-line accelertor for memcached. *Computer Architecture Letters*, (2), 2014.

[297] Jinho Lee and Trevor E. Carlson. Ultra-fast cgra scheduling to enable run time, programmable cgras. *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 12 2021.

[298] Jungi Lee and Jongeun Lee. NP-CGRA: Extending CGRAs for efficient processing of light-weight deep neural networks. *DATE*, 2021.

[299] Kevin Lee, Vijay Rao, and William Christie Arnold. Accelerating Facebook's infrastructure with application-specific hardware, 2019. Available at `https://engineering.fb.com/data-center-engineering/accelerating-infrastructure/`.

[300] Yunsup Lee, David Sheffield, Andrew Waterman, Michael Anderson, Kurt Keutzer, and Krste Asanovic. Measuring the gap between programmable and

fixed-function accelerators: A case study on speech recognition. *2013 IEEE Hot Chips 25 Symposium (HCS)*, 8 2013.

[301] Binrui Li, Shenggan Cheng, and James Lin. tcFFT: Accelerating half-precision FFT through Tensor Cores. *CoRR*, 2021. Available at `https://arxiv.org/pdf/2104.11471.pdf`.

[302] Dai Li, Akhil Pakala, and Kaiyuan Yang. MeNTT: A compact and efficient processing-in-memory number theoretic transform (ntt) accelerator. *VLSI*, 2022.

[303] Jiajie Li, Yuze Chi, and Jason Cong. Heterohalide. *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2 2020.

[304] Yetting Li, Shuaimin Li, Zhiwu Xu, Jialun Cao, Zixuan Chen, Yun Hu, Haiming Chen, and Shing-Chi Cheung. TransRegex: Multi-modal regular expression synthesis by generate-and-repair. *ICSE*, 2021.

[305] Yijia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. Gated graph sequence neural networks. *ICLR*, 2016.

[306] Zhaoying Li, Dhananjaya Wijerathne, Xianzhang Chen, Anuj Pathania, and Tulika Mitra. Chordmap: Automated mapping of streaming applications onto CGRA. *Computer-aided Design of Integrated Circuits and Systems*, 41(2), 2022.

[307] Zhihong Lin, Jagadeesh Sankaran, and Tom Flanagan. Empowering automotive vision with TI's Vision AccelerationPac. 2013. Available at `https://www.ti.com/lit/wp/spry251/spry251.pdf`.

[308] Alex X. Liu and Eric Torng. Overlay automata and algorithms for fast and scalable regular expression matching. *IEEE/ACM Transactions on Networking*, 24:2400–2415, 8 2016.

[309] Cheng Liu, Ho-Cheung Ng, and Hayden Kowk-Hay So. Quickdough: A rapid FPGA loop accelerator design framework using soft CGRA overlay. *FPT*, 2015.

[310] Dajiang Liu, Shouyi Yin, Leibo Liu, and Shaojun Wei. Polyhedral model based mapping optimization of loop nests for CGRAs. *DAC*, 2013.

[311] Dajiang Liu, Shouyi Yin, Guojie Luo, Jiaxing Shang, Leibo Liu, Shaojun Wei, Yong Feng, and Zhou Shangbo. Data-flow graph mapping optimization for

CGRA with deep reinforcement learning. *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38(12), December 2019.

[312] Daofu Liu, Tianshi Chen, Shaoli Liu, Jinhong Zhou, Shengyuan Zhou, Olivier Teman, Xiaobing Feng, Xuehai Zhou, and Yunji Chen. Pudiannao: A polyvalent machine learning accelerator. *ASPLOS*, 2015.

[313] Leibo Liu, Guiqiang Pen, and Shaojun Wei. Dynamic reconfigurable chips for massive MIMO detection. *Massive MIMO Detection Algorithm and VLSI Architecture*, pages 229–306, 2019.

[314] Leibo Liu, Jianfeng Zhu, Zhaoshi Li, Yanan Lu, Yangdong Deng, Jie Han, Shouyi Yin, and Shaojun Wei. A survey of coarse-grained reconfigurable architecture and design. *ACM Computing Surveys*, 52:1–39, 10 2019.

[315] Nian Liu. Characterizing deprecated deep learning python APIs: An empirical study on tensorflow. 2021.

[316] Qiaoyi Liu, Dillon Huff, Jeff Setter, Maxwell Strange, Kathleen Feng, Kavya Sreedhar, Ziheng Wang, Keyi Zhang, Mark Horowitz, Priyanka Raina, and Fredrik Kjolstad. Compiling halide programs to push-memory accelerators. *CoRR*, 2021.

[317] Weiqiang Liu, Qicong Liao, Fei Qiao, Weijie Xia, Chenghua Wang, and Fabrizio Lombardi. Approximate designs for fast Fourier transform (FFT) with application to speech recognition. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 66:4727–4739, 12 2019.

[318] D. Brandon Lloyd, Chas Boyd, and Naga Govindaraju. Fast computation of general Fourier Transforms on GPUs. *ICME*, 6 2008.

[319] Joao D Lopes and Jose T de Sousa. Fast fourier transform on the versat CGRA. *Silicon Errors Logic-System Effects*, pages 174–187, 2016.

[320] Thorben Louw and Simon McIntosh-Smith. Using the Graphcore IPU for traditional HPC applications. *3rd Workshop on Accelerated Machine Learning (AccML)*, 2021.

[321] Alisa J. Maas, Henrique Nazaré, and Ben Liblit. Array length inference for c library bindings. *ASE*, 8 2016.

[322] Raju Machupalli, Masum Hossain, and Mrinal Mandal. Review of ASIC accelerators for deep neural network. *Microprocessors and Microsystems*, 89, 2022.

[323] Andrew MacLeod. Ranger: An on-demand range generate for GCC. 2018. Available at `https://gcc.gnu.org/wiki/AndrewMacLeod/Ranger`.

[324] Fernanda Madeiral and Thomas Durieux. A large-scale study on human-cloned changes for automated program repair. *MSR*, 2021.

[325] Kavitha T Madhu, Saptarsi Das, Nalesh S, S K Nandy, and Ranjani Narayan. Compiling HPC kernels for the REDEFINE CGRA. *HPCC*, 2015.

[326] Kingshuk Majumder and Uday Bondhugula. HIR: An MLIR-based intermediate representation for hardware accelerator description. *CoRR 2021*, 2021. Available at `https://arxiv.org/abs/2103.00194`.

[327] Teddy Mantoro and Fifit Alfiah. Comparison methods of DCT, DWT and FFT techniques approach on lossy image compression. *International Conference on Computing Engineering, and Design*, 2017.

[328] Haiyu Mao, Mingcong Song, Tao Li, Yuting Dai, and Jiwo Shu. LerGAN: A zero-free, low data movement and PIM-based GAN architecture. *Micro*, 2018.

[329] Kevin J M Martin. Twenty years of automated methods for mapping applications on CGRA. *IDDD International Parallel and Dsitributed Processing Symposium Workshops (IPDPSW)*, 2022.

[330] Pablo Antonio Martinez, Gregorio Bernabe, and Jose Manuel Garcia. POAS: A high-performance scheduling framework for exploiting accelerator level parallelism. *PACT*, 2022.

[331] Pablo Antonio Martinez, Jackson Woodruff, Jordi Armengol-Estapé, Gregorio Bernabé, José Manuel García, and Michael O'Boyle. Matching linear algebra and tensor code to specialized hardware accelerators. *CC*, 2023.

[332] Pedro Martins, Rohan Achar, and Cristina V. Lopes. 50K-C: a dataset of compilable, and compiled, Java projects. *MSR*, 2018.

[333] George Mathew, Chris Parnin, and Kathryn T Stolee. SLACC. *ICSE*, 6 2020.

[334] Florian Mayer, Julian Brandner, Matthias Hellmann, Jesko Schwarzer, and Michael Philippsen. The ORKA-HPC compiler — practical OpenMP for FPGAs. *ICPC*, 2021.

[335] Joel Mandebi Mbongue, Alex Shuping, Pankaj Bhowmik, and Christophe Bobda. Architecture support for FPGA multi-tenancy in the cloud. *ASAP 2020*, 2020.

[336] Mark McKeown. FFT implementation on the TMS320VC5505, TMS320C5505, and TMS320C5515 DSPs. Technical Report SPRABB6B, Texas Instruments, 2013. Available at `https://www.ti.com/lit/an/sprabb6b/sprabb6b.pdf`.

[337] B. Mei, B. De Sutter, T. Vander Aa, M. Wouters, A. Kanstein, and S. Dupont. Implementation of a coarse-grained reconfigurable media processor for avc decoder. *Journal of Signal Processing Systems*, 51:225–243, 6 2008.

[338] Bingfeng Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. DRESC: a retargetable compiler for coarse-grained reconfigurable architectures. *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings.*, 2002.

[339] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. *ADRES: An Architecture with Tightly Coupled VLIW Processor and Coarse-Grained Reconfigurable Matrix*, pages 61–70. Springer Berlin Heidelberg, 2003.

[340] Jackson Melchert, Kathleen Feng, Caleb Donovick, Ross Daly, Clark Barrett, Mark Horowitz, Pat Hanrahan, and Priyanka Raina. Automated design space exploration of CGRA processing element architectures using frequent subgraph analysis. *CoRR*, 2021. Available at `https://arxiv.org/abs/2104.14155`.

[341] Charith Mendis, Jeffrey Bosboom, Kevin Wu, Shoaib Kamil, Jonathan Ragan-Kelley, Sylvain Paris, Qin Zhao, and Saman Amarasinghe. Helium: lifting high-performance stencil kernels from stripped x86 binaries to halide DSL code. *PLDI*, 6 2015.

[342] R. Meyer. Error analysis and comparison of FFT implementation structures. *International Conference on Acoustics, Speech, and Signal Processing*, 1989.

[343] Microsoft. HoloLens 2 hardware, 2019. Available at `https://docs.microsoft.com/en-us/hololens/hololens2-hardware`.

[344] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. *ANCS 2007*, 2007.

[345] Yukio Miyasaka, Masahiro Fujita, Alan Mishchenko, and John Wawrzynek. SAT-based mapping of data-flow graphs onto coarse-grained reconfigurable arrays. *IFIP Advances in Information and Communication Technology*, 2021.

[346] Adrian Mizzi, Joshua Ellul, and Gordon Pace. D'artagnan. *Proceedings of the Real World Domain Specific Languages Workshop 2018 on - RWDSL2018*, 2 2018.

[347] Ashwin Vishnu Mohanan, Cyrille Bonamy, and Pierre Augier. FluidFFT: Common API (C++ and Python) for Fast Fourier Transform HPC libraries. *Journal of Open Research Software*, 7, 4 2019.

[348] Soumak Mookherjee, Linda DeBrunner, and Victor DeBrunner. A low power radix-2 FFT accelerator for FPGA. *2015 49th Asilomar Conference on Signals, Systems and Computers*, 11 2015.

[349] William S Moses, Lorenzo Chelini, Ruizhe Zhao, and Oleksandr Zienko. Polygeist: Affine C in MLIR. *IMPACT*, 2021.

[350] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. *AAAI 2016*, 2016.

[351] Roger Moussalli, Mariam Salloum, Robert Halstead, Walid Najjar, and Vassilis J. Tsotras. A study on parallelizing XML path filtering using accelerators. *ACM Transactions on Embedded Computing Systems*, 13:1–28, 12 2014.

[352] Movidius. Enhanced visual intelligence at the network edge, 2018. Available at `www.movidius.com/MyriadX`.

[353] Bryon Moyer. Scramble for the white space. 2020. Available at `https://semiengineering.com/scramble-for-the-white-space/`.

[354] Song Mu, Yi Zeng, and Bo Wang. Routability-enhanced scheduling for application mapping on CGRAs. *Access*, 2021.

[355] D. Munson and B. Liu. Floating point roundoff error in the prime factor FFT. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 29:877–882, 8 1981.

[356] Charles D Murphy. Low-complexity FFT structures for OFDM transceivers. *IEEE Transactions on Communications*, 50(12), 2002.

[357] Alastair Murray and Björn Franke. Compiling for automatically generated instruction set extensions. *Proceedings of the Tenth International Symposium on Code Generation and Optimization - CHO '12*, 3 2012.

[358] Alastair Colin Murray. *Customising Compilers for Customisable Processors*. PhD thesis, University of Edinburgh, 2011.

[359] Chandrakana Nandi, Max Willsey, Adam Anderson, James R Wilcox, Eva Darulova, Dan Grossman, and Zachary Tatlock. Synthesizing structured cad models with equality saturation and inverse transformations. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 31–44, 2020.

[360] Akshay Narayan, Aurojit Panda, Mohammad Alizadeh, Hari Balakrishnan, Arvind Kreshnamurthy, and Scott Shenker. Bertha: Tunneling through the network API. *HotNets*, 2020.

[361] Rashid Naseem, Onaiza Maqbool, and Siraj Muhammad. An improved similarity measure for binary features in software clustering. *2010 Second International Conference on Computational Intelligence, Modelling and Simulation*, 9 2010.

[362] Rashid Naseem, Onaiza Maqbool, and Siraj Muhammad. Improved similarity measures for software clustering. *2011 15th European Conference on Software Maintenance and Reengineering*, 3 2011.

[363] Ho-Cheung Ng, Shuanglong Liu, Izaak Coleman, Ringo S.W. Chu, Man-Chung Yue, and Wayne Luk. Acceleration of short read alignment with runtime reconfiguration. *FPGA 2021*, 2021.

[364] Hung K Nguyen and Xuan-Tu Tran. Design and implementation of a coarse-grained dynamically reconfigurable multimedia accelerator. *ACM Transactions on Parallel Computing*, 9(3), 2022.

[365] Quan M. Nguyen and Daniel Sanchez. Fifer: Practical acceleration of irregular applications on reconfigurable architectures. *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 10 2021.

[366] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N. Nguyen. Exploring API embedding for API usages and applications. *ICSE*, 5 2017.

[367] Ansong Ni, Daniel Ramos, Aidan Yang, Ines Lynce, Vasco Manquinho, Ruben Martins, and Claire Le Goues. SOAR: A synthesis approach for data science API refactoring. *ICSE*, 2021.

[368] Chris Nicol. A coarse grain reconfigurable array (CGRA) for statically scheduled data flow computing. Technical report, Wave Computing, 2017.

[369] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Moller. Semantic patches for adaption of JavaScript programs to evolving libraries. *ICSE*, 2021.

[370] Marziyeh Nourian, Xiang Wang, Xiaodong Yu, Wu-chun Feng, and Michela Becchi. Demystifying automata processing: GPUs, FPGAs or Micron's AP? *ICS 2017*, 2017.

[371] Tony Nowatzki, Vinay Gangadhar, Karthikeyan Sankaralingam, and Greg Wright. Domain specialization is generally unnecessary for accelerators. *IEEE Micro*, 37:40–50, 2017.

[372] Tony Nowatzki, Michael Tarm-Sartin, Lorenzo De Carli, Karthikeyan Sankaralingam, Cristian Estan, and Behnam Robatmili. A general constraint-centric scheduling framework for spatial architectures. *PLDO*, 2013.

[373] Seiya Numata, Norihiro Yoshida, Eunjong Choi, and Katsuro Inoue. On the effectiveness of vector-based approach for supporting simultaneous editing of software clones. *Product-Focused Software Process Improvement*, pages 560–567, 11 2016.

[374] Eriko Nurvitadhi, David Sheffield, Jaewoong Sim, Asit Mishra, Ganesh Venkatesh, and Debbie Marr. Accelerating binarized neural networks: Comparison of FPGA, CPU, GPU and ASIC. *FPT*, 2016.

[375] Nvidia. NVENC — nvidia hardware video encoder: Application note. Available at `https://docs.nvidia.com/video-technologies/video-codec-sdk/12.1/nvenc-application-note/index.html`, 2014.

[376] Ayaka Ohwada, Takuya Kojima, and Hideharu Amano. An efficient compiation of coarse-grained reconfigurable architectures utilizing pre-optimized sub-graph mappings. *PDP*, 2022.

[377] Subhankar Pal, Jonathan Beaumont, Dong-Hyeon Park, Aporva Amarnath, Siying Feng, Chaitali Chakrabarti, Hun-Seok Kim, David Blaauw, Trevor Mudge, and Ronald Dreslinski. OuterSPACE: An outer product based sparse matrix multiplication accelerator. *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2 2018.

[378] Subhankar Pal, Siying Feng, Dong-hyeon Park, GSung Kim, Aprova Amarnath, Chisheng Yang, Xin He, Jonathan Beaumont, Kyle May, Yan Xiong, Kuba Kaszyk, John Magnus Morton, Jiawen Sun, Michael F P O'Boyle, Murray I Cole, Chaitali Charkrabarti, David Theodore Blaauw, Hunseok Kim, Trevel Nigel Mudge, and Ronald G Drewlinski. Transmuter: Briding the efficiency gap using memory and dataflow reconfiguration. *PACT*, 2020.

[379] Pavel Panchekha, Alex Sanchez-Stern, James R Wilcox, and Zachary Tatlock. Automatically improving accuracy for floating point expressions. *ACM SIGPLAN Notices*, 50(6):1–11, 2015.

[380] Salvatore Pantarelli, Claudio Greco, Enrico Nobile, Simone Teofili, and Guiseppe Bianchi. Exploiting dynamic reconfiguration for FPGA based network intrusion detection systems. *FPL*, 2010.

[381] Dongjoon Park, Yuanlong Xiao, Nevo Magnezi, and Andre DeHon. Case for fast fpga compilation using partial reconfiguration. *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, 8 2018.

[382] Taejune Park, Jaehyun Nam, and Seung Ho Na. Reinhardt: Real-time reconfigurable hardware architecture for regular expression matching in DPI. *ACSAC*, 2021.

[383] Yongjun Park, Hyunchul Park, and Scott Mahlke. CGRA express: Accelerationg execution using dynamic operation fusion. *CASES*, 2009.

[384] Daniele Parravicini, Davide Conficconi, Emanuele Del Sozzo, Christian Pilato, and Marco D. Santambrogio. Cicero: A domain-specific architecture for efficient

regular expression matching. *ACM Transactions on Embedded Computing Systems*, 20:1–24, 10 2021.

[385] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. PyTorch: An imperative style, high-performance deep learning library. *NeurIPs*, 2019.

[386] Darayus Adil Patel, Viet Phoung Bui, Kevin Tshun Chuan Chai, Amit Lal, and Mohamed M Sabry Aly. SonicFFT: A system architecture of ultrasonic-based FFT acceleration. *ASP-DAC*, 2022.

[387] Biagio Peccerillo, Micro Mannino, Andrea Mondelli, and Sandro Bartolini. A survey on hardware accelerators: Taxonomy, trends, challenges and perspectives. *Journal of Systems Architecture*, 2022.

[388] Ardavan Pedram, Andreas Gerstlauer, and Robert A van de Geijin. A high-performance, low-power linear algebra core. *ASAP*, 2011.

[389] Ardavan Pedram, John McCalpin, and Andreas Gerstlauer. Transforming a linear algebra core to an FFT accelerator. *ASAP*, 6 2013.

[390] Yvan Petillot. Fast fourier transfrom. 2004. Lecture. Available at `http://www.eece.hw.ac.uk/~ceeyrp/WWW/Teaching/B39SE1/DSP4.pdf`.

[391] Francesco Peverelli, Alberto Zeni, and Enrico Cabri. CircFA: Circular RNA FPGA aligner. *Xilinx Open Hardware 2018*, 2018.

[392] Hung Dang Phan, Anh Tuan Nguyen, Trong Duc Nguyen, and Tien N. Nguyen. Statistical migration of api usages. *ICSE-C*, 5 2017.

[393] Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. Chlorophyll: Synthesis-aided compiler for low-power spatial architectures. *PLDI*, 2014.

[394] Teemu Pitkanen, Tero Partanen, and Jarmo Takala. Low-power twiddle factor unit for FFT computation. *SAMOS 2007*, pages 65–74, 2007.

[395] Artur Podobas, Kentaro Sano, and Satoshi Matsuoka. A template-based framework for exploring coarse-grained reconfigurable architectures. *ASAP*, 2020.

[396] S Pophale and D Oryspayev. Outcomes of OpenMP hackathon: OpenMP application experiences with the offloading mode. *IWOMP*, pages 68–80, September 2021.

[397] Raghu Prabhakar, Yaqi Zhang, David Koeplinger, Matt Feldman, Tian Zhao, Stefan Hadjis, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. Plasticine. *Proceedings of the 44th Annual International Symposium on Computer Architecture — ISCA '17*, 6 2017.

[398] Rohit Prasad, Satyajit Das, Kevin J M Martin, and Philippe Coussy. Floating point CGRA based ultra-low power DSP accelerator. *Journal of Signal Processing Systems*, 93:1159–1171, 2021.

[399] Seth H. Pugsley, Arjun Deb, Rajeev Balasubramonian, and Feifei Li. Fixed-function hardware sorting accelerators for near data MapReduce execution. *2015 33rd IEEE International Conference on Computer Design (ICCD)*, 10 2015.

[400] Markus Puschel, Jose M Moura, Jeremy Johnson, David Padua, Manuela M Veloso, Bryan W Singer, Jianxin Xiong, Franz Franchetti, Aca Gacic, Yevgen Voronenko, Kang Chen, Robert W Johnson, and Nicholas Rizzolo. SPRIAL: Code generation for DSP transforms. *Proceedings of the IEEE*, 93, 2004.

[401] Jonathan Ragan-Kelley, Andrew Adams, Dillon Sharlet, Connelly Barnes, Sylvain Paris, Marc Levoy, Saman Amarasinghe, and Frédo Durand. Halide. *Communications of the ACM*, 61:106–115, 12 2017.

[402] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide. *Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation - PLDI '13*, 6 2013.

[403] Reza Rahimi, Elaheh Sadredini, Mircea Stan, and Kevin Skadron. Grapefruit: An open-source, full-stack, and customizable automata processing on FPGAs. *FCCM*, 2020.

[404] Annapurna P Rajarajeswari, S nd Patil, Aditya Madhyastha, Akshat Jaitly, Himangshu Shekar Jha, Sahil Rajesh Bhave, Mayukh Das, and N S Preadeep.

Design and develop hardware aware DNN for faster inference. *Lecture Notes in Networks and Systems*, 544, 2023.

[405] Zoltan Endre Rakossy, Dominik Stengele, Axel Acosta-Aponte, Saumitra Chafekar, Paolo Bientinesi, and Anupam Chattopadhyay. Scalable and efficient linear algebra kernel mapping for low energy consumption on the layers CGRA. *International Symposium on Applied Reconfigurable Computing*, 2015.

[406] Rahul Razdan and Michael D. Smith. A high-performance microarchitecture with hardware-programmable functional units. *Proceedings of the 27th annual international symposium on Microarchitecture - MICRO 27*, 11 1994.

[407] Brandon Reagen, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. Quantifying acceleration: Power/performance trade-offs of application kernels in hardware. *International Symposium on Low Power Electronics and Design (ISLPED)*, 9 2013.

[408] Albert Reuther, Peter Michaleas, Michael Jones, Vijay Gadepally, Siddharth Samsi, and Jeremy Kepner. Survey of machine learning accelerators. *HPEC*, 2020.

[409] Tony Robinson, Jim Harkin, and Priyank Shukla. Hardware acceleration of genomics data analysis: challenges and opportunities. *Bioinformatics*, 37:1785–1795, 2021.

[410] Tejas Ruschke, Johannes Jung, Lukas, dennis Wolf, and Christian Hochberger. Scheduler for inhomogeneous and irregular CGRAs with support for complex control flow. *PDS*, 2016.

[411] Rafael H. Saavedra and Alan J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. *ACM Transactions on Computer Systems (TOCS)*, 14:344–384, 11 1996.

[412] Amir Hossein Nodehi Sabet, Junqiao Qiu, Zhijia Zhao, and Sriram Krishnamoorthy. Reliability analysis for unreliable FSM computations. *ACM Transactions on Architecture and Code Optimization*, 17:1–23, 5 2020.

[413] Elaheh Sadredini, Reza Rahimi, Marzieh Lenjani, Mircea Stan, and Kevin Skadron. FlexAmata: A universal and efficient adaption of applications to spatial automata processing accelerators. *ASPLOS 2020*, 2020.

[414] Elaheh Sadredini, Reza Rahimi, Vaibhav Verma, Mircea Stan, and Kevin Skadron. eAP. *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture*, 10 2019.

[415] Vaibhav Saini, Farima Farmahinifarahani, Yadong Lu, Pierre Baldi, and Cristina V. Lopes. Oreo: detection of clones in the twilight zone. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering — ESEC/FSE 2018*, 11 2018.

[416] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. SourcererCC. *ICSE*, 5 2016.

[417] Malavika Samak, Jose Pablo Cambronero, and Martin C Rinard. Searching for replacement classes. *CoRR*, 2021. Available at `https://arxiv.org/abs/2110.05638`.

[418] Malavika Samak, Deokhwan Kim, and Martin C Rinard. Synthesizing replacement classes. *POPL*, 2020.

[419] Paulo C Santos, Bruno E Forlin, and Luigi Carro. Providing plug n' play for processing-in-memory accelerators. *ASPDAC*, 2021.

[420] O. Sarbishei and K. Radecka. Analysis of mean-square-error (MSE) for fixed-point FFT units. *2011 IEEE International Symposium of Circuits and Systems (ISCAS)*, 5 2011.

[421] Kaz Sato, Cliff Young, and David Patterson. An in depth look at google's first tensor processing unit (tpu). *Google Cloud Blog*, 2017. Available at `https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu`.

[422] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic optimization of floating-point programs with tunable precision. *PLDI*, 2014.

[423] Christof Schlaak, Tzung-Han Juang, and Christophe Dubach. Optimizing data reshaping operations in functional IRs for high-level synthesis. *TACO*, 2022.

[424] Fabian Schuiki, Andreas Kurth, Tobias Grosser, and Luca Benini. Llhd: a multi-level intermediate representation for hardware description languages. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 6 2020.

[425] Mathijs Schuts, Jozef Hooman, and Paul Tielemans. Industrial experience with the migration of legacy models using a DSL. *RWDSL*, 2018.

[426] Frank Schwierz and Juin J Liou. Status and future prospects of CMOS scaling and Moore's law — a personal perspective. *Latin America Electron Devices Conferece 2020*, 2020.

[427] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. *Usenix ATC*, 2012.

[428] Francois Serre and Markus Puschel. A DSL-based FFT hardware generator in scala. *FPL 2018*, 2018.

[429] Ofer Shacham and Masumi Reynders. Pixel visual core: image processing and machine learning on Pixel 2, 2017. Available at `https://www.blog.google/products/pixel/pixel-visual-core-image-processing-and-machine-learning-pixel-2/`.

[430] Amin Shafiee Sarvestani, Erik Hansson, and Christoph Kessler. Extensible recognition of algorithmic patterns in DSP programs for automatic parallelization. *International Journal of Parallel Programming*, 41:806–824, 12 2013.

[431] Muzammil Shahbaz, Phil McMinn, and Mark Stevenson. Automatic generation of valid and invalid test data for string validation routines using web searches and regular expressions. *Science of Computer Programming*, 97:405–425, 1 2015.

[432] R. Sidhu and V.K. Prasanna. Fast regular expression matching using FPGAs. *FCCM*, 2001.

[433] Adam Siemieniuk, Lorenzo Chelini, Asif Ali Khan, Jeronimo Castrillon, Andi Drebes, Henk Corporaal, Tobias Grosser, and Martin Kong. OCC: An autoamted end-to-end machine learning optimization compiler for computing in-memory. *TCAD*, 2021.

[434] Mark Silberstein. Accelerators in data centers: the systems perspective. *ACM SIGARCH*, 2017.

[435] Silex Insight. Blockchain hardware accelerator: Product sheet BA452. Available at https://www.silexinsight.com/content/uploads/BA452-Blockchain-Hardware-Accelerator_Web.pdf.

[436] Silex Insight. Custom OEM Board for 4K HDMK AV over IP. Available at https://www.silexinsight.com/content/uploads/Viper_OEM_Product_sheet.pdf.

[437] Silex Insight. How to achieve low latency audio/video streaming over IP network. Available at https://www.silexinsight.com/content/uploads/Whitepaper_Low_Latency_AV_over_IP_SilexInsight.pdf.

[438] Silex Insight. JPEC Encoder IP Core: BA116 product sheet. Available at https://www.silexinsight.com/content/uploads/BA116_JPEG_Encoder.pdf.

[439] Silex Insight. JPEG 2000. Available at https://www.silexinsight.com/content/uploads/J2K-Brochure.pdf.

[440] Silex Insight. MPEG2 decoder IP. Available at https://www.silexinsight.com/content/uploads/BA119MPEG2.pdf.

[441] Silex Insight. Public key crypto engine: Product sheet BA414EP. Available at https://www.silexinsight.com/content/uploads/BA414EP_Public_Key_IP_Core-1.pdf.

[442] Silex Insight. True 4K multiview — AV over IP. Available at https://www.silexinsight.com/content/uploads/Feature-Sheet-True_4K_Multiview_V1.1_Web.pdf.

[443] Silex Insight. VC-2 HQ — the superior lightweight video codec. Available at https://www.silexinsight.com/content/uploads/Whitepaper_VC2-HQ_SilexInsight_V1.1_web.pdf.

[444] Lucas Silva, Michael Canesche, Ricardo Ferreira, and José Augusto Nacif. Hpcgra - an orthogonal designed cgra generator for high performance spatial accelerators. *Anais do XXI Simpósio em Sistemas Computacionais de Alto Desempenho (WSCAD 2020)*, 10 2020.

[445] Aishwarya Sivaraman, Rui Abreu, Andrew Scott, Tobi Akomolede, and Satish Chandra. Mining idioms in the wild. *ICSE-SEIP*, 2022.

[446] Calvin Smith and Aws Albarghouthi. MapReduce program synthesis. *PLDI*, 2016.

[447] Gus Henry Smith, Andrew Liu, Steven Lyubomirsky, Scott Davidson, Joseph McMahan, Michael Taylor, Luis Ceze, and Zachary Tatlock. Pure tensor program rewriting via access patterns (representation pearl). *Proceedings of the 5th ACM SIGPLAN International Symposium on Machine Programming*, pages 21–31, 2021.

[448] Peter Soderquist and Miriam Leeser. Area and performance tradeoffs in floating-point divide and square-root implementations. *ACM Computing Surveys*, 28:518–564, 9 1996.

[449] Armando Solar-Lezama. *Program Synthesis by Sketching*. PhD thesis, 2008.

[450] Anumeena Sorna, Xiaohe Cheng, Eduardo D'Azevedo, Kwai Won, and Stanimire Tomov. Optimizing the Fast Fourier Transform using mixed precision on Tensor Core hardware. *HiPCW*, 12 2018.

[451] Ioannis Stamelos, Elias Koromilas, Chris Kachris, Fred Tsang, Raj Subramani, Graham McKenzie, and Natalia Poliakova. Accelerating quantitative finance with FPGA-based acceleration cards. inaccel. Available at `https://inaccel.com/wp-content/uploads/Solution-Brief-Finlib-Flumaion.pdf`.

[452] Samuel Steffl and Sherief Reda. LACore: A supercomputing-like linear algebra accelerator for soc-based designs. *2017 IEEE International Conference on Computer Design (ICCD)*, 11 2017.

[453] Michael Benjamin Stepp. Equality saturation: engineering challenges and applications, 2011.

[454] Gordon Stewart, Mahanth Gowda, Geoffrey Mainland, Bozidar Radunovic, Dimitrios Vytiniotis, and Cristina Luengo Agullo. Ziria. *ASPLOS*, 3 2015.

[455] M. Stojilovic, D. Novo, L. Saranovac, P. Brisk, and P. Ienne. Selective flexibility: Creating domain-specific reconfigurable arrays. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 32:681–694, 5 2013.

[456] Fang-Hsiang Su, Jonathan Bell, Kenneth Harvey, Simha Sethumadhavan, Gail Kaiser, and Tony Jebara. Code relatives: detecting similarly behaving software. *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering — FSE 2016*, 11 2016.

[457] Arun Subramaniyan, Jingcheng Wang, Ezhil R. M. Balasubramanian, David Blaauw, Dennis Sylvester, and Reetuparna Das. Cache automaton. *Proceedings of the 50th Annual IEEE/ACM International Symposium on Microarchitecture — MICRO-50 2017*, 10 2017.

[458] Wei Sun, Savvas Sioutas, Sander Stuijk, Andrew Nelson, and Henk Corporaal. Efficient tensor cores support in TVM for low-latency deep learning. *DATE 2021*, 2021.

[459] D Sundararajan. *The Discrete Fourier Transform*. World Scientific Publishing Co. Pte. Ltd., 2001.

[460] Dam Sunwoo, William Wang, Mrinmoy Ghosh, Chander Sudanthi, Geoffrey Blake, Christopher D. Emmons, and Nigel C. Paver. A structured approach to the simulation, analysis and characterization of smartphone applications. *2013 IEEE International Symposium on Workload Characterization (IISWC)*, 9 2013.

[461] Marcelo Suzuki, Adriano Carvalho de Paula, Eduardo Guerra, Cristina V. Lopes, and Otavio Augusto Lazzarini Lemos. An exploratory study of functional redundancy in code repositories. *2017 IEEE 17th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 9 2017.

[462] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. Towards a big data curated benchmark of inter-project code clones. *2014 IEEE International Conference on Software Maintenance and Evolution*, 9 2014.

[463] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. WaveScalar. *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 291–, 2003.

[464] Andreas Sæbjørnsen, Jeremiah Willcock, Thomas Panas, Daniel Quinlan, and Zhendong Su. Detecting code clones in binary executables. *Proceedings of the*

eighteenth international symposium on Software testing and analysis — ISSTA '09, 7 2009.

[465] Tac Fintech. Low latency trading solution: World class trading infrastructure for world class trading teams. *Xilinx Alveo Solution Brief*. Available at `https://www.xilinx.com/publications/solution-briefs/partner/tac-fintech-solution-brief.pdf`.

[466] Kai Sheng Tai, Richard Socher, and Christopher D. Manning. Improved semantic representations from tree-structured long short-term memory networks. *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, 7 2015.

[467] Cheng Tan, Nicolas Bohm Agostini, Jeff Zhang, Marco Minutoli, Vito Giovanni Castellana, Chenhao Xie, Tong Geng, Ang Li, Kevin Barker, and Antonino Tumeo. Opencgra: Democratizing coarse-grained reconfigurable arrays. *2021 IEEE 32nd International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, 7 2021.

[468] Cheng Tan, Chenhao Xie, Ang Li, Kevin J. Barker, and Antonino Tumeo. Aurora: Automated refinement of coarse-grained reconfigurable accelerators. *2021 Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2 2021.

[469] Wen Jun Tan, Wai Teng Tang, Rick Siow Mong Goh, Stephen John Turner, and Weng-Fai Wong. A code generation framework for targeting optimized library calls for multiple platforms. *PDS*, 26(7):1789–1799, July 2015.

[470] Prateek Tandon, Faissal M. Sleiman, Michael J. Cafarella, and Thomas F. Wenisch. HAWK: Hardware support for unstructured log processing. *2016 IEEE 32nd International Conference on Data Engineering (ICDE)*, 5 2016.

[471] Ping Tak Peter Tang. Dfti—a new interface for fast fourier transform libraries. *ACM Transactions on Mathematical Software*, 31:475–507, 12 2005.

[472] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 264–276, 2009.

[473] M.B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, None Jae-Wook Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22:25–35, 3 2002.

[474] Michael B. Taylor. Is dark silicon useful? *DAC*, 6 2012.

[475] Tekle Tefai, Huruy, Hani Saleh, Temesghen Tekeste, Mahmoud Alqutayri, and Baker Mahammad. ASIC implementation of a pre-trained neural network for ECG feature extraction. *ISCAS*, 2020.

[476] Jens Teubner, Louis Woods, and Chongling Nie. Skeleton automata for FPGAs: Reconfiguring without reconstructing. *SIGMOD*, 2012.

[477] Tung Thanh-Hoang, Amirali Shambayati, Calvin Deutschbein, Henry Hoffmann, and Andrew A. Chien. Performance and energy limits of a processor-integrated FFT accelerator. *HPEC*, 9 2014.

[478] The Snort Project. SNORT users manual: 2.9.16, 2020. Available at `http://manual-snort-org.s3-website-us-east-1.amazonaws.com/snort_manual.html`.

[479] Michael E Thomadakis. The architecture of the nehalem processor and nehalem-EP SMP platforms. 2011.

[480] Titan IC. RXP for ASIC: accelerated search and analytics, 2019. Acquired by Mellanox — Not clear if this product will continue to be available. https://www.mellanox.com/titan-ic.

[481] Jesmin Jahan Tithi, Fabrizio Petrini, Hongbo Rong, Andrei Valentin, and Car Ebeling. Mapping stencils on coarse-grained reconfigurable spatial architecture. *CoRR*, 2021.

[482] Tsung-Han Tsai and Hsing-Chuang Liu. Design and implementation of filterbank for MPEG-2/4 AAC system. *Integration*, 2021.

[483] Richard Uhrie. *Automatic Computational Domain Detection*. PhD thesis, Arizona State University, 2021.

[484] Ecenur Ustun, Ismail San, Jiaqi Yin, Cunxi Yu, and Zhiru Zhang. Impress: Large integer multiplication expression rewriting for fpga hls. *2022 IEEE 30th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 1–10, 2022.

[485] Marcel D van de Burgwal, Pascal T Wolkotte, and Gerard J M Smit. Non-power-of-two FFTs: Exploring the flexibility of the montium TP. *International Journal of Reconfigurable Computing*, 2009.

[486] Brian Van Essen, Robin Panda, Aaron Wood, Carl Ebeling, and Scott Hauck. Energy-efficient specialization of functional units in a coarse-grained reconfigurable array. *FPGA*, 2011.

[487] Alexa VanHattum, Rachit Nigam, Vincent T. Lee, James Bornholt, and Adrian Sampson. A synthesis-aided compiler for DSP architectures (WiP paper). *LCTES*, 6 2020.

[488] Alexa VanHattum, Rachit Nigam, Vincent T Lee, James Bornholt, and Adrian Sampson. Vectorization for digital signal processors via equality saturation. *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 874–886, 2021.

[489] Pranathi Vasireddy, Krishna Kavi, and Gayatri Mehta. Spase-T: hardware accelerator thread for unstructured sparse data processing. *ICCAD*, 2022.

[490] Margus Veanes, Caleb Stanford, Olli Saarikivi, and Nikolaj Bjorner. Symbolic extended regular expression matching and analysis. *PLDI 2020 (Talk)*, 2020. Available at `https://www.youtube.com/watch?v=rMS4rTbY0So`.

[491] Stylianos I Venieris, Ioannis Panopoulos, Ilias Leontiadis, and Iakovos Venieris. How to read real-time AI on consumer devices? solutions for programmable and custom architectures. *ASAP*, 2021.

[492] Rangharajan Venkatesan, Yakun Sophia Shao, Miaorong Wang, Jason Clemons, Steve Dai, Matthew Fojtik, Ben Keller, Alicia Klinefelter, Nathaniel Pinckney, Priyanka Raina, Yanqing Zhang, Brian Zimmer, William J Dally, Joel Emer, Stephen W Keckler, and Brucek Khailany. MAGNet: A module accelerator generator for neural networks. *ICCAD*, 2019.

[493] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson, and Michael Bedford Taylor. Conservation cores. *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems - ASPLOS '10*, 3 2010.

[494] Alvaro Videla. Meaning and context in computer programs. *Communicationsof the ACM*, 65(5):56–58, 2022.

[495] Maria Vieira, Michael Caneshe, Lucas Braganca, Josue Campos, and Mateus Silva. RESHAPE: A run-time dataflow hardware-based mapping for CGRA overlays. *ISCAS*, 2021.

[496] Jason Villarreal, Adrian Park, Walid Najjar, and Robert Halstead. Designing modular hardware accelerators in C with ROCCC 2.0. *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, 5 2010.

[497] Kizheppatt Vipin and Suhaib A. Fahmy. Fpga dynamic and partial reconfiguration. *ACM Computing Surveys*, 51:1–39, 9 2018.

[498] Jack Wadden, Kevin Angstadt, and Kevin Skadron. Characterizing and mitigating output reporting bottlenecks in spatial automata processing architectures. *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, 2 2018.

[499] Jack Wadden, Vinh Dang, Nathan Brunelle, Tommy Tracy II, Deyuan Guo, Elaheh Sadredini, Ke Wang, Chunkun Bo, Gabriel Robins, Mircea Stan, and Kevin Skadron. ANMLZoo: A benchmark suite for exploring bottlenecks in automata processing engines and architectures. *IISWC*, 2016.

[500] Stefan Wagner, Asim Abdulkhaleq, Ivan Bogicevic, Jan-Peter Ostberg, and Jasmin Ramadani. How are functionally similar code clones syntactically different? an empirical study and a benchmark. *PeerJ Computer Science*, 2, 3 2016.

[501] Shin'ichi Wakabayashi, Shinobu Nagayama, Yosuke Kawanka, and Sadatoshi Mikami. Hardware accelerators for regular expression matching and approximate string matching. *Asia-Pacific Signal and Information Processing Association*, 2009.

[502] Matthew J P Walker and Jason H Anderson. Generic connectivity-based CGRA mapping via integer linear programming. *FCCM*, 2019.

[503] Chao Wang, Lei Gong, Qi Yu, Xi Li, Yuan Xie, and Xuehai Zhou. DLAU: A scalable deep learning accelerator unit on FPGA. *Transactions on Computer-Aided Design of integrated Circuits and Systems*, 36(3), 2017.

[504] Ke Wang and Zhendong Su. Learning blended, precise semantic program embeddings. *CoRR*, 2018. Available at `https://arxiv.org/abs/1907.02136`.

[505] Minjie Wang, Da Zheng, Zihao Ye, Quan Gan, Mufei Li, Xiang Song, Jinjing Zhou, Chao Ma, Lingfan Yu, Yu Gai, Tianjun Xiao, Tong He, George Karypis, Jinyang Li, and Zheng Zhang. Deep graph library: A graph-centric, highly-performant package for graph neural networks. *CoRR*, 2019. Available at `https://arxiv.org/pdf/1909.01315`.

[506] Shuai Wang and Dinghao Wu. In-memory fuzzing for binary code similarity analysis. *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 10 2017.

[507] Xiang Wang, Yang Hong, Harry Chang, KyoungSoo Park, Geoff Langdale, Jiayu Hu, and Heqing Zhu. Hyperscan: A fast multi-pattern regex matcher for modern CPUs. *Proceedings of the 16th USENIX Symposium on Networked Systems Design and Implementation*, 2019.

[508] Xin Wang, Yasheng Wang, Yao Wan, Fei Mi, Yitong Li, Pingui Zhou, Jin Liu, Hao Wu, Xin Jiang, and Qun Liu. Compilable neural code generation with compiler feedback. *CoRR*, 2022.

[509] Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. Spores: sum-product optimization via relational equality saturation for large scale linear algebra. *arXiv preprint arXiv:2002.07951*, 2020.

[510] Yuanrong Wang, Qiangqiang Li, and Guangming Tan. Application taxonomy via algorithmic commonality for domain-specific architecture desgin. *2015 IEEE 22nd International Conference on High Performance Computing (HiPC)*, 12 2015.

[511] Rick Weber, Akila Gothandaraman, Robert J Hinde, and Gregory D Peterson. Comparing hardware accelerators in scientific applications: A case study. *PDS*, 2011.

[512] Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence*, 8 2017.

[513] Westley Weimer and George C. Necula. *Mining Temporal Specifications for Error Detection*, pages 461–476. Springer Berlin Heidelberg, 2005.

[514] Yuanbo Wen, Qi Guo, Qiang Fu, Xiaqing Li, Jianxing Xu, Yanlin Tang, Yonwei Zhao, Xing Hu, Zidong Du, Ling Li, Chao Wang, Xuehai Zhou, and Yunji Chen. BabelTower: learning to auto-parallelized program translation. *ICML*, 2022.

[515] Jian Weng, Animesh Jian, Jie Wang, Leyuan Wang, Yide Wang, and Tony Nowatzki. UNIT: Unifying tensorized instruction compilation. *CGO*, 2021.

[516] Jian Weng, Sihao Liu, Vidushi Dadu, Zhengrong Wang, Preyas Shah, and Tony Nowatzki. DSAGEN: synthesizing programmable spatial accelerators. *ISCA*, 2020.

[517] Jian Weng, Sihao Liu, Dylan Kupsh, and Tony Nowatzki. Unifying spatial accelerator compilation with idiomatic and modular transformations. *Micro*, 2022.

[518] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. Deep learning code fragments for code clone detection. *ASE*, 2016.

[519] Dhananjaya Wijerathne, Zhaoying Li, Thilini Kaushalya Bandara, and Tulika Mitra. PANORAMA: divide-and-conquer approach for mapping complex loop kernels on CGRA. *DAC*, 2022.

[520] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunarathne, Anuj Pathania, and Tulika Mitra. CASCADE: High throughput data streaming via decoupled access-execute CGRA. *Transactions on Embedded Computing Systems*, 18(5s), 2019.

[521] Dhananjaya Wijerathne, Zhaoying Li, Manupa Karunaratne, Li-Sihuan Peh, and Tulika Mitra. Morpher: An open-source integrated compilation and simulation framework for CGRA. *Workshop on Open-Source EDA Technology*, 2022.

[522] Mark Wijvliet, Luc Waeijen, and Henk Corporaal. Coarse grained reconfigurable architectures in the past 25 years: Overview and classification. *SAMOS*, 2016.

[523] Max Willsey, Vincent T. Lee, Alvin Cheung, Rastislav Bodik, and Luis Ceze. Iterative search for reconfigurable accelerator blocks with a compiler in the loop. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 38:407–418, 3 2019.

[524] Max Willsey, Chandrakana Nandi, Yisu Remy Wang, Oliver Flatt, Zachary Tatlock, and Pavel Panchekha. egg: Fast and extensible equality saturation. *POPL*, 2021.

[525] Ramon Wirsch and Christian Hochberger. Towards transparent dynamic binary translation from RISC-V to a CGRA. *ARCS*, 2021.

[526] M.J. Wirthlin and B.L. Hutchings. A dynamic instruction set computer. *Proceedings IEEE Symposium on FPGAs for Custom Computing Machines*, 1995.

[527] Weunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. CENTRIS: A precise and scalable approach for identifying modified open-source software reuse. *ICSE*, 2021.

[528] Aaron Wood. *Offset Pipelining for Coarse Grain Reconfigurable Arrays*. PhD thesis, 2017.

[529] Jackson Woodruff, Sam Ainsworth, and Michael F.P. O'Boyle. Secco: Codesign for resource sharing in regular expression accelerators. *ASP-DAC*, 2024.

[530] Jackson Woodruff, Jordi Armengol-Estapé, Sam Ainsworth, and Michael F P O'Boyle. Bind the gap: Compiling real software to hardware FFT accelerators. *PLDI*, 2022.

[531] Jackson Woodruff and Michael F P O'Boyle. New regular expressions on old accelerators. *DAC 2021*, 2021.

[532] Cheng-Shing Wu and An-Yeu Wu. Modified vector rotational CORDIC (MVR-CORDIC) algorithm and its application to FFT. *Circuits and Systems*, 2000.

[533] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. Q100. *Proceedings of the 19th international conference on Architectural support for programming languages and operating systems — ASPLOS '14*, 3 2014.

[534] Haojun Xia, Lei Gong, Chao Wang, Xianglan Chen, and Xuehai Zhou. LAP: A lightweight automata processor for pattern matching tasks. *DATE*, 2021.

[535] Ted Xie, Vinh Dang, Jack Wadden, Kevin Skadron, and Mircea Stan. REAPR: Reconfigurable engine for automata processing. *FPL*, 9 2017.

[536] Xinfeng Xie, Zheng Liang, Peng Gu, Abanti Basak, Lei Deng, Ling Liang, Xing Hu, and Yuan Xie. SpaceA: Sparse matrix vector multiplication on processing-in-memory accelerator. *HPCA*, 2021.

[537] Yuan Xie. A brief guide of xPU for AI accelerators, 2018. Available at `https://www.sigarch.org/a-brief-guide-of-xpu-for-ai-accelerators/`.

[538] Xilinx. Xilinx ML suite accelerates AI/ML on Alveo data ceneter accelerator cards, 2018. Available at `https://www.xilinx.com/products/acceleration-solutions/xilinx-machine-learning-suite.html`.

[539] Xilinx. General matrix operation, 2019. Available at `https://github.com/Xilinx/gemx`.

[540] Xilinx. Versal: The first adaptive computer acceleration platform (ACAP), 2019. Available at `https://www.xilinx.com/support/documentation/white_papers/wp505-versal-acap.pdf`.

[541] Xilinx. Vitis BLAS library documentation, 2020. Available at `https://xilinx.github.io/Vitis_Libraries/blas/user_guide/L1/L1_compute_api.html`.

[542] Xilinx. AI engine: Meeting the compute demands of next-generation applications. Accessed 2022. Available at `https://www.xilinx.com/products/technology/ai-engine.html`.

[543] Xilinx. MLIR-based AIEngine toolchain. Accessed 2022. Available at `https://xilinx.github.io/mlir-aie/`.

[544] Chengcheng Xu, Shuhui Chen, Jinshu Su, S. M. Yiu, and Lucas C. K. Hui. A survey on regular expression matching for deep packet inspection: Applications, algorithms, and hardware platforms. *IEEE Communications Surveys & Tutorials*, 18:2991–3029, 2016.

[545] Congying Xu, Xiaobing Sun, Bin Li, Xintong Lu, and Hongjing Guo. MULAPI: Improving API method recommendation with API usage location. *Journal of Systems and Software*, 142:195–205, 8 2018.

[546] Weixiang Yan, Haitian Liu, Yunkun Wang, Yunzhe Li, Qian Chen, Wen Wang, Tingyu Lin, Weishan Zhao, Li Zho, Shuiguang Deng, and Hari Sundaram. CodeScope: An execution-based multilingual multitask multidimesional benchmark for evaluating llms on code understanding and generation. 2023. Available at `https://arxiv.org/pdf/2311.08588.pdf`.

[547] Chen Yang, Leibo Liu, Kai Luo, Shouyi Yin, and Shaojun Wei. CIACP: A correlation- and iteration- aware cache partitioning mechanism to improve performance of multiple coarse-grained reconfigurable arrays. *Transactions on Parallel and Distributed Systems*, 28(1):29–43, 2017.

[548] Tao Yang, Dongyue Li, Yibo Han, Yilong Zhao, Fangxin Liu, Xiaoyao Liang, Zhezhi He, and Li Jiang. PIMGCN: A ReRAM-based PIM design for graph convolutional network accelerators. *DAC*, 2021.

[549] Yang Yang, Sanmukh R. Kuppannagari, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. Fasthash: Fpga-based high throughput parallel hash table. *Lecture Notes in Computer Science*, pages 3–22, 6 2020.

[550] Yi-Hua E Yang, Weirong Jiang, and Viktor K Prananna. Compact architecture for high-throughput regular expression matching on FPGA. *ANCS 2008*, 2008.

[551] Yi-Hua Edward Yang and Viktor K Prasanna. High-performance and compact architecture for regular expression matching on FPGA. *Transactions on Computers*, 2012.

[552] Yichen Yang, Phitchaya Phothilimthana, Yisu Wang, Max Willsey, Sudip Roy, and Jacques Pienaar. Equality saturation for tensor graph superoptimization. *Proceedings of Machine Learning and Systems*, 3:255–268, 2021.

[553] Hasan Erdem Yantir, Wenzhe Guo, Ahmed M. Eltawil, Fadi J. Kurdahi, and Khaled Nabil Salama. An ultra-area-efficient 1024-point in-memory FFT processor. *Micromachines*, 10:509, 7 2019.

[554] Joshua J. Yi, David J. Lilja, and Douglas M. Hawkins. A statistically rigorous approach for improving simulation methodology. *ACM Trans. Comput. Syst.*, 14(4):281, November 2003.

[555] Jonghee W Yoon, Sanghyun Park, Minwook Ahn, Reiley Jeyapaul, and Y Paek. SPKM: a novel graph drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures. *ASP-DAC*, 2008.

[556] Hangchen Yu, Arthur M. Peters, Amogh Akshintala, and Christopher J. Rossbach. Automatic virtualization of accelerators. *ASPLOS 2020*, 2020.

[557] Jintao Yu, Muath Abu Lebdeh, Hoang Anh Du Nguyen, Mottaqiallah Taouil, and Said Hamdioui. Apmap: An open-source compiler for automata processors. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, pages 1–1, 2021.

[558] Baofen Yuan, Jianfeng Zhu, Xingchen Man, Zijiao Ma, and Shouyi Yin. Dynamic-II pipeline: Compiling loops with irregular branches on static-scheduling CGRA. *TCAD*, 2021.

[559] Georgios Zacharopoulos, Lorenzo Ferretti, Giovanni Ansaloni, Giuseppe Di Guglielmo, Luca Carloni, and Laura Pozzi. Compiler-assisted selection of hardware acceleration candidates from application source code. *2019 IEEE 37th International Conference on Computer Design (ICCD)*, 11 2019.

[560] Shulin Zeng, Hanbo Sun, Yu Xing, Xuefei Ning, Yi Shan, Xiaoming Chen, Yu Wang, and Huazhong Yang. Black box search space profiling for accelerator-aware neural architecture search. *ASP-DAC*, 2020.

[561] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. *ICSE*, 5 2019.

[562] Yaqi Zhang, Nathan Zhang, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. SARA: Scaling a reconfigurable dataflow accelerator. *ISCA*, 2021.

[563] Yilong Zhao, Zhezhi He, Naifeng Jing, Xiaoyao Liang, and Li Jiang. Re2PIM: A reconfigurable ReRAM-based PIM design for variable-sized vector-matrix multiplication. *GLSVLSI*, 2021.

[564] Zhipeng Zhao, Hugo Sadok, James Hoe, Vyas Sekar, and Justine Sherry. Achieving 100gbps intrusion prevention on a single server. *OSDI 2020*, 2020.

[565] Zhongyuan Zhao, Weiguang Sheng, Weifeng He, ZhiGang Mao, and Zhaoshi Li. A static-placement, dynamic-issue framework for cgra loop accelerator. *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2017*, 3 2017.

[566] Li Zhou, Lie Hengzhu, and Jianfeng Zhang. Loop acceleration by cluster-based CGRA. *IEICE*, pages 1–8, 2013.

[567] Peipei Zhou, Jiayi Sheng, Cody Hao Yu, Peng Wei, Jie Wang, Di Wu, and Jason Cong. MOCHA: Multinode cost optimization in heterogeneous clouds with accelerators. *FPGA 2021*, 2021.

[568] Yan Zhuang, Zhihao Zhang, and Dajiang Liu. Towards high-quality CGRA mapping with graph neural networks and reinforcement learning. *ICCAD*, 2022.

[569] Minhaz F Zibran, Fariana Z Eishita, and Chanchal Roy. Useful, but usable? factors affecting the usability of APIs. *Working Conference on Reverse Engineering*, 2011.

[570] Farzaneh Zokaee, Fan Chen, Guangyu Sun, and Lei Jiang. Sky-sorter: A processing-in-memory architecture for large-scale sorting. *Transactions on Computers*, 2022.

[571] Maksim Zubkov, Egor Spirin, Egor Bogomolov, and Timofey Bryksin. Evaluation of constrastive learning with various code representations for code clone detection. *CoRR*, 2022. Available at `https://arxiv.org/pdf/2206.08726.pdf`.