FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Improving Compilation Flows for RISC-V Machine Learning Custom Instructions

Guilherme Soares Sequeira

Master in Informatics and Computing Engineering

Supervisor: Prof. João Carlos Viegas Martins Bispo

Second Supervisor: Prof. Nuno Miguel Cardanha Paulino

September 30, 2025

Improving Compilation Flows for RISC-V Machine Learning Custom Instructions

Guilherme Soares Sequeira

Master in Informatics and Computing Engineering

September 30, 2025

Resumo

A procura por *edge devices* de baixo custo e de baixo consumo energético capazes de executar *workloads* de IA tem aumentado nos últimos anos. O interesse em combinar RISC-V, uma Instruction Set Architecture (ISA) de padrão aberto e livre de *royalties* concebido de raiz com costumização em mente, com hardware especializado, capaz de executar as tarefas para as quais foi concebido com uma eficiência excepcional, surge naturalmente, dando origem a múltiplos Intellectual Properties (IPs) baseados em RISC-V. No entanto, poucos parecem interessados em desenvolver os compiladores juntamente com o seu hardware, seja por exigirem um investimento muito elevado, por ter uma curva de aprendizagem acentuada, ou outros fatores.

Esta dissertação propõe uma alternativa: a introdução de uma etapa de compilação *source-to-source* antes da compilação final, permitindo a inserção automática de *custom instructions* diretamente no código-fonte utilizando *in-line assembly*, usando uma Application Programming Interface (API) e um ecossistema muito mais acessíveis.

Discutimos os detalhes da aceleração automática de produtos escalares vetor-vetor com a utilização de uma *custom instruction* Multiply-And-Accumulate (MAC), bem como a análise estática necessária ao longo do processo. No final das contas, conseguimos encontrar oportunidades de aceleração em *benchmarks* de terceiros. Correndo um programa num Field-Programmable Gate Array (FPGA) programado com um IP *closed-source*, conseguimos atingir um *speedup* de até 7,1 vezes em comparação com o programa original não otimizado, e acompanhando o desempenho de código otimizado manualmente.

Palavras-chave
Compilação Source-to-Source, RISC-V, Análise Estática, Edge AI

Abstract

The demand for low-cost, low-power edge devices capable of performing Artificial Intelligence (AI) workloads has been increasing in the last few years. Interest in pairing RISC-V, an open-standard, royalty-free ISA built from the ground-up with customizability in mind, with specialized hardware, capable of performing the tasks they are designed for with exceptional efficiency, naturally begins to emerge, spawning multiple RISC-V based IPs. However, few seem interested in developing the compilers alongside their hardware, either due to requiring too big of an investment, steep learning curve, or other factors.

This thesis proposes an alternative: the introduction of a source-to-source compilation step right before compilation, allowing the automatic insertion of custom instructions directly into the source code using in-line assembly using a much more accessible API and ecosystem.

We discuss the details of automatically accelerating vector-vector dot products with the use of a MAC custom instruction as well as the necessary static analysis along the way. At the end of the day, we are able to find acceleration opportunities in third-party benchmarks. When running our program in an FPGA programmed with a closed-source IP we achieve a speedup of up to 7.1 times compared to the original, unoptimized program and matching the performance of manually optimized code.

Keywords
Source-to-Source Compilation, RISC-V, Static Analysis, Edge AI

Acknowledgments

I would first like to express my utmost gratitude to my supervisors, who have shown me unthinkable amounts of patience, kindness, support and friendship. There is nothing else I could've possibly asked for.

Next I would like to thank all my friends who have stuck with me through thick and thin, who have made bearable the toughest of periods, and who have given me so many moments to cherish. I think I'll keep on looking forward to tomorrow.

Of course, I couldn't fail to mention my family, they who have always prioritized me above all else. For their kindness, love and encouragement I will be eternally grateful.

Finally, and most importantly, I would like to thank my cats, they who keep my feet warm during winter and my heart warm year-round.

Guilherme Sequeira

"Off we go
into the Wild Pale yonder"

Elysium

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Context

Although it has been around since the 50s AI has seen explosive growth over the past decade, spurred by breakthroughs in Machine Learning (ML) and Deep Learning (DL) [54]. As the models' capabilities in various fields increased, so did their use cases [49].

To achieve any meaningful applicability, the computational power required to run the majority of models is very high, effectively restricting its use to costly, powerful computers, typically available only to scientific and governmental institutions or large corporations [49]. As models became more capable and computers smaller and more efficient, new use cases in the Internet of Things (IoT) field emerged, such as smart home appliances, smart watches, surveillance systems, etc. Though these systems usually rely on a centralized server for intensive workloads, there are many benefits to delegating some of the computation to edge devices [50]. Smart watches benefit highly from lower latency, smart home appliances from lower risk of personal information leakage, and cameras can perform movement detection, allowing them to only send new images when needed therefore reducing network bandwidth consumption.

RISC-V [2] is a relatively recent ISA that distinguishes itself from its competitors on account of its open standard status and focus on extensibility. Due to being royalty-free and open-source, it is an appealing option to small and medium-sized companies looking to compete in the embedded systems market.

Additionally, as the standard was built from the ground up taking into account processor customization, it enables the addition of custom instructions, or even the coupling of the processor with accelerators, with relative ease. Thus, this specialized hardware can perform the tasks it was engineered for with exceptional speed and efficiency, enabling significant improvements in performance, power consumption, and overall system efficiency.

The combination of these two key characteristics makes RISC-V a compelling option for custom hardware designers tackling a problem that benefits from custom instructions to accelerate AI tasks with strict budget and power efficiency constraints.

Custom instructions have the potential to significantly accelerate execution by performing operations that would require multiple other instructions to achieve the same result. These custom instructions are commonly used in one of three ways, though each comes with disadvantages.

Firstly, the programmer can use the instructions directly with in-line assembly, however this is not desirable as it locks the software to specific RISC-V implementations which support the respective extension, is time-consuming and error-prone for the programmer to think at instruction-level and bypasses compiler optimizations.

Secondly, hardware built-ins, that is mappings provided by the compiler for a given instruction, can be used, allowing the compiler to capture the semantics of the instructions they implement [23], though they still suffer from the other drawbacks of in-line assembly.

Finally, the compiler can automatically detect opportunities to use the custom instructions (instruction auto-generation), solving the previous disadvantages at the cost of complexity. If the compiler's internals do not already provide the framework for detecting these opportunities a compiler extension is typically used, adding yet another set of challenges.

In the next sections we will list some of the challenges currently faced by custom ISA extension developers, focusing mostly on the issues alluded to on the previous paragraph.

1.2 Motivation

As referenced above, custom ISA extension developers have implemented extensions for GCC [20] [38] and LLVM [32] [39], in part to be able to auto-generate instructions when the compiler does not provide the necessary framework to do so. Developing these new compiler features has a steep learning curve, requires significant engineering effort, restricts the utilization of the instructions to one compiler and its distribution is impractical.

A Source-to-Source Compiler (S2S Compiler), that is, a compiler whose output is also source code (e.g. TypeScript to JavaScript or C to C), can find portions of the source code that benefit from the custom instructions and optimize the program by replacing them with inline assembly, emulating optimization passes present in compilers. Although traditional compiler support is often desirable for the end product, utilizing this approach to prototype the support for custom instructions and accelerators has many benefits. Firstly, a S2S Compiler can provide an API or Domain Specific Language (DSL) which has a smoother learning curve and requires less effort to use, relative to the environment provided by traditional compilers such as GCC or LLVM. Secondly, it causes no compiler lock-in and there is no need to catch up to new compiler updates, meaning that because passes are implemented in a separate tool they are compatible with any compiler, so long as they both support the same language version.

In [13], an open-source compiler focusing on optimizing AI models for a wide range of IPs with custom instructions was presented. However, this is a very specialized tool that only accepts PyTorch, TensorFlow or Open Neural Network Exchange (ONNX) as input. In contrast, a S2S Compiler usually supports large portions, if not fully, their target languages.

1.3 Problem

A wide range of devices is currently used to run AI workloads, including Central Processing Units (CPUs), Graphics Processing Units (GPUs), FPGAs, Coarse-Grained Re-configurable Arrays (CGRAs), Tensor Processing Units (TPUs), and other Application-Specific Integrated Circuits (ASICs) [13], and the number of different devices used will increase even further with the up-and-coming RISC-V IPs featuring custom ISA extensions support. Optimizing DL workloads for these hardware targets requires careful consideration of the underlying memory architectures [13] and is becoming increasingly difficult for operator-level level libraries to keep up with their diversity. Many ignore a significant part of the market, opting instead to focus on server-class GPU devices and delegating the optimization to vendor-specific libraries which are difficult to port to other IPs [13].

On the other hand, developing GCC and LLVM extensions to support custom hardware instructions currently requires significant engineering effort for IP authors [13].

We believe that the process of preparing an environment for adapting and testing an application to use custom hardware components has the potential to be streamlined and that IP authors developing a RISC-V ISA extension could greatly benefit from it.

1.4 Research Questions

This dissertation aims to answer the following research questions:

1. What are the main state-of-the-art RISC-V ISA extensions targeting AI acceleration, how accessible are their development environments and how much effort does it take to simulate their custom instructions?

2. Is it possible to complement the current compilation flow of RISC-V with a S2S Compiler in a way that reduces effort, increases ease-of-use or smoothens the learning curve?

3. Can source code analysis discover opportunities for custom AI instructions, enabling optimization passes without the need of using large compiler frameworks such as GCC or LLVM?

1.5 Hypothesis

Utilizing S2S Compilers (e.g. Clava [9]) to apply transformations to sections of source code can accelerate the prototyping of new instructions provided by custom RISC-V ISA extensions, while not being as encumbering as developing a GCC or LLVM extension.

Chapter 2

Literature Review

For the purposes of this dissertation, we carried out a systematic review of the literature roughly following the Preferred Reporting Items for Systematic Reviews and Meta-Analyses (PRISMA) guidelines, through which we intend on making this process rigorous, transparent, and replicable.

2.1 Research Domain

As the demand for low-cost and low-power devices capable of running AI tasks efficiently increased, various initiatives sought to develop their own RISC-V based ASICs. Though there are various ways of implementing the acceleration, either via co-processors, Processing in Memory (PiM) modules, separate processing units such as TPUs, the focus is on those programmable via an ISA. This is more common in smaller functional units integrated in the processor's pipeline, as opposed to the more powerful GPUs and TPUs, and co-processors.

2.2 Identification Phase

We searched four prominent digital libraries, namely SCOPUS, ACM Digital Library, IEEE Xplore and Web of Science, for articles, conference papers and preprints, by applying small variations to the two query strings we concocted, adapting them to each search engine.

2.2.1 Query String 1

```
("RISC-V" OR "riscv")
AND
("Instruction Set Architecture extension" OR "ISA extension" OR "custom
    instruction")
AND
("AI" OR "artificial intelligence" OR "ML" OR "machine learning" OR "NN"
    OR "neural network" OR "inference")
```

The first query matches results that mention three topics:

Initial Search Results			
Source / Query String	QS1	QS2	Total
SCOPUS	48	11	59
ACM DL	6	8	14
IEEE Xplore	15	1	16
Web of Science	18	11	29
Total	87	31	118

Table 2.1: Number of matches of each query string per digital library.

- RISC-V related keywords
- Keywords related to the extension of the instruction set architecture
- Keywords related to the broad category of AI

As none of these categories are too specific, we can limit the search in most of the digital libraries to records that specify the designated terms in their abstracts, resulting in a moderately sized collection of on-topic records.

2.2.2 Query String 2

```
("RISC-V" OR "riscv")
AND
("Instruction Set Architecture extension" OR "ISA extension" OR "custom
    instruction")
AND
("edge intelligence" OR "AIoT" OR "Artificial Intelligence of Things" OR
    tinyML)
```

The second query string features the same first two categories, however the third restricts the results to the rather specific topic of edge intelligence. RISC-V based ASICs with extended ISAs are of particular interest to this area, and although AI isn't explicitly mentioned we believe that the records that would be otherwise missed, should we omit these terms, will nonetheless be pertinent.

2.2.3 Search Results

The results of the search of both query strings across the digital libraries is summarized in Table 2.1.

2.3 Screening Phase

The goal of this phase is to methodically remove records that are not relevant to the topic at hand. Upon defining inclusion and exclusion criteria, we test each record according to them and exclude those that do not hold up. To this effect, we utilized a mix of the reference manager's capabilities

Phase	Inclusion / Exclusion Criteria
1	Search engine results after executing a search query.
2	Not duplicated.
3	Published since 2020 (inclusive).
4	Article, conference paper or preprint.
5	Published in the english language.
6	Not a retracted publication.
7	Full-text access.

Table 2.2: List of chosen inclusion and exclusion criteria.

and manual labour. The defined criteria can be found in Table 2.2 and each will be expanded upon below. We will justify its selection, how it was applied and present the updated list of matches after its application. The first is omitted, as that is simply the initial table of results mentioned above.

2.3.1 Not duplicated

Records with matching Digital Object Identifiers (DOIs), or those with the same title, authors and publication date can be automatically excluded by the reference manager. Duplicated instances of records thus stop being counted in multiple sections of the results table. Priority is given to the first query string, and higher priority to SCOPUS, then the ACM Digital Library, IEEE Xplore and finally Web of Science. For example, if the same record was found by both the first and the second query strings in SCOPUS, it will hereupon be counted only as a result of the first query string, and if the same record is found both in SCOPUS and the ACM Digital Library, it will be counted only as a result of SCOPUS.

The results of removing the duplicate records is summarized in Table 2.3. As we can see, all records matched by IEEE Xplore were duplicates, thus this entry will omitted henceforth.

2.3.2 Published since 2020 (inclusive)

Not all papers can be reviewed due to time constraints. As newer papers are likely to be closer to the state-of-the-art, these will be evaluated first. Older but still influential papers are likely to be referenced in these newer papers, so they might still be considered at a later stage.

IEC2			
Sources / Query String	QS1	QS2	Total
SCOPUS	48	5	53
ACM DL	0	8	8
IEEE Xplore	0	0	0
Web of Science	0	4	4
Total	48	17	65

Table 2.3: Number of records accounting with inclusion and exclusion criteria 2.

IEC2 + IEC3			
Sources / Query String	QS1	QS2	Total
SCOPUS	42	5	47
ACM DL	0	8	8
Web of Science	0	4	4
Total	42	17	59

Table 2.4: Number of records accounting with inclusion and exclusion criteria 2 and 3.

The reference manager's capabilities allowed us to easily filter out the records that did not match this criteria. The results are summarized in Table 2.4.

2.3.3 Article, conference paper or preprint

Other publication types, such as books or proceedings, were discarded as they take too much time to review or are not very relevant. Once again, we relied on the reference manager for this task. The results are summarized in Table 2.5.

2.3.4 Published in the english language

Only records published in the english language were considered, as these should be accessible to us and the majority of the scientific community. We relied on the reference manager's capabilities when possible, and manually inspected the remaining ones, however this did not exclude any records, therefore a table is omitted.

2.3.5 Not a retracted publication

These records were excluded for obvious reasons. We once again relied on the reference manager for this task, however no records were excluded again. A table is again omitted for the same reasons.

2.3.6 Full-text access

The records resulting from this search will be reviewed in full on the State of the Art section of this dissertation, for which reading the text in full is required. We sought the records that were

IEC2+IEC3+IEC4			
Sources / Query String	QS1	QS2	Total
SCOPUS	42	5	47
ACM DL	0	1	1
Web of Science	0	4	4
Total	42	10	52

Table 2.5: Number of records accounting with inclusion and exclusion criteria 2 to 4.

IEC2+IEC3+IEC4+IEC5+IEC6+IEC7			
Sources / Query String	QS1	QS2	Total
SCOPUS	39	5	44
ACM DL	0	1	1
Web of Science	0	4	4
Total	39	10	49

Table 2.6: Number of records accounting with inclusion and exclusion criteria 2 to 7.

not automatically caught by the reference manager, however some still eluded us in the end. The results are summarized in Table 2.6.

2.4 Eligibility and Selection

Upon trimming the number of records in the previous section, we manually reviewed each of the remaining records' titles, abstracts and conclusions as needed, further reducing the number of those that do not fit in the desired topic.

At this stage, we manually select the records that present a custom ISA extension for RISC-V (i.e. not a standard extension such as "P" or "V") with the purpose of accelerating AI-related workloads, those that provide insight into process or those that present a survey on the existing projects in this area.

The result is a list of 42 selected papers presented in Table 2.7.

Table 2.7: List of selected records.

Reference	Record Name
[35]	NARS: Neuromorphic Acceleration through Register-Streaming Extensions on RISC-V Cores
[53]	Vortex: Extending the RISC-V ISA for GPGPU and 3D-Graphics
[31]	Accelerate Binarized Neural Networks with Processing-in-Memory Enabled by RISC-V Custom Instructions
[60]	RV-SCNN: A RISC-V Processor With Customized Instruction Set for SNN and CNN Inference Acceleration on Edge Platforms
[5]	A Scalable RISC-V Hardware Platform for Intelligent Sensor Processing
[11]	RISCV-FNT: A Fast FNT-based RISC-V Processor for CNN Acceleration
[61]	AI-ISP Accelerator with RISC-V ISA Extension for Image Signal Processing
[46]	An 2.31uJ/Inference Ultra-Low Power Always-on Event-Driven AI-IoT SoC With Switchable nvSRAM Compute-in-Memory Macro
[17]	RISC-V Instruction Set Architecture Extensions: A Survey
[44]	xTern: Energy-Efficient Ternary Neural Network Inference on RISC-V-Based Edge Systems

Reference	Record Name
[7]	MiniFloat-NN and ExSdotp: An ISA Extension and a Modular Open Hardware Unit for Low-Precision Training on RISC-V Cores
[15]	A Lightweight Posit Processing Unit for RISC-V Processors in Deep Neural Network Applications
[52]	RISC-HD: Lightweight RISC-V Processor for Efficient Hyperdimensional Computing Inference
[58]	Optimizing CNN Computation Using RISC-V Custom Instruction Sets for Edge Platforms
[12]	An Eight-Core RISC-V Processor With Compute Near Last Level Cache in Intel 4 CMOS
[62]	FlexBits: A Configurable Lightweight RISC-V Micro-architecture for Flexible Bit-Width Execution
[43]	Variable Bit-Precision Vector Extension for RISC-V Based Processors
[42]	Yun: An Open-Source, 64-Bit RISC-V-Based Vector Processor With Multi-Precision Integer and Floating-Point Support in 65-nm CMOS
[63]	Case Study: Optimization Methods With TVM Hybrid-OP on RISC-V Packed SIMD
[6]	MiniFloats on RISC-V Cores: ISA Extensions with Mixed-Precision Short Dot Products
[28]	Value Similarity Extensions for Approximate Computing in General-Purpose Processors
[21]	LiteAIR5: A System-Level Framework for the Design and Modeling of AI-extended RISC-V Cores
[1]	Implementation and analysis of custom instructions on RISC-V for Edge-AI applications
[37]	Instruction Set Extension of a RiscV Based SoC for Driver Drowsiness Detection
[59]	RV-GEMM: Neural Network Inference Acceleration with Near-Memory GEMM Instructions on RISC-V
[65]	RISC-V based Fully-Parallel SRAM Computing-in-Memory Accelerator with High Hardware Utilization and Data Reuse Rate
[18]	ISA Modeling with Trace Notation for Context Free Property Generation
[29]	DBPS: Dynamic Block Size and Precision Scaling for Efficient DNN Training Supported by RISC-V ISA Extensions
[26]	Using Source-to-Source to Target RISC-V Custom Extensions: UVE Case-Study
[25]	A Multi-mode Convolution Coprocessor Based on RISC-V Instruction Set Architecture
[14]	Low DRAM Memory Access and Flexible Dataflow Convolutional Neural Network Accelerator based on RISC-V Custom Instruction
[55]	EXTREM-EDGE—EXtensions To RISC-V for Energy-efficient ML inference at the EDGE of IoT
[22]	A 1.15 TOPS/W, 16-Cores Parallel Ultra-Low Power Cluster with 2b-to-32b Fully Flexible Bit-Precision and Vector Lockstep Execution Mode

Reference	Record Name
[10]	VITAMIN-V: Virtual Environment and Tool-Boxing for Trustworthy Development of RISC-V Based Cloud Services
[27]	RISC-VR-Extension: Advancing Efficiency with Rented-Pipeline for Edge DNN Processing
[57]	Customized instruction on RISC-V for Winograd-based convolution acceleration
[4]	Designing RISC-V Instruction Set Extensions for Artificial Neural Networks: An LLVM Compiler-Driven Perspective
[19]	The Scale4Edge RISC-V ecosystem
[41]	Enhancement in IoT through Custom Instruction Set Architectures and TinyML: Review
[56]	AI-PiM-Extending the RISC-V processor with Processing-in-Memory functional units for AI inference at the edge of IoT
[47]	Sparse Stream Semantic Registers: A Lightweight ISA Extension Accelerating General Sparse Linear Algebra
[30]	APPEND: Rethinking ASIP Synthesis in the Era of AI

Chapter 3

State of the Art

In this section we introduce key concepts necessary to understand the state-of-the-art of RISC-V ISA extensions. We also discuss some literature, part of which was identified in the previous section, and others added afterwards if deemed relevant, presenting the authors' work, how it is relevant to our work, their results and limitations.

3.1 Background

3.1.1 RISC-V

RISC-V is an open standard, royalty-free ISA based on Reduced Instruction Set Computer (RISC) principles capable of powering devices ranging from low-power specialized ASICs to highly performant linux capable machines. It has seen considerable growth in adoption over the last decade thanks to its open licensing model and its extensibility.

RISC-V's instruction encoding is very regular, meaning it is relatively easy to reserve a section of the encoding space for a group of related instructions, such as branching instructions, 32 bit integer operations, and others. Most of the encoding space is allocated to common instructions in what are called standard extensions. However, the *I* extension, containing only basic operations natively on 32 bit, is the only compulsory one. Before a group of instructions is added to the RISC-V specification as a standard extension it must first go through a thorough process of evaluation by the RISC-V International Association.

Although support for all standard extensions isn't mandatory, the encoding space is reserved nonetheless, meaning no two instructions from any extension can be encoded in the same way. However, a portion of the encoding space is reserved for non-standard extensions, meaning that it is not used by any standard extension. Thus, IP developers can use this space to implement their custom specialized instructions enjoying a full guarantee of compatibility with every present and future RISC-V standard extension. As there are no official requirements of custom extensions, other than limiting the encoding of instructions to the allocated "custom instruction" section, two custom extensions may use the same encoding for their respective instructions, making them mutually incompatible.

3.1.2 Compilation, Toolchains and SDKs

Modern software development usually involves software engineers writing code in a high-level programming language that is then translated into machine code that can be executed by the target computer in a process commonly denominated as compilation. In truth, the source code is processed in multiple steps by different tools in (usually) decreasing levels of abstraction. A traditional compilation flow, would start with a preprocessor, if the language requires it, taking as input the high-level code and expanding compile-time directives and functions such as C's macros. The expanded code is then fed into a compiler, which is itself composed of multiple parts.

The compiler first parses the expanded code, transforming it into an intermediary representation, usually an Abstract Syntax Tree (AST), which is significantly easier to analyze. Next, the compiler inspects the program to ensure its validity and, if it is valid, applies a set of operations that try to optimize the code to varying extents. Finally, the compiler translates the program to an intermediate representation with even lower levels of abstraction, such as LLVM-IR, which operates close to the register transfer level. To execute this step it must take into account the limitations of the target machine. For one, it must consider how many registers the target machine has and manage their uses in a process called register allocation. For the purposes of this work, however, code selection is the most relevant task performed at this stage. Code selection concerns itself with selecting the most appropriate set of hardware instructions that execute the logic defined in the program. As an example, let us consider a program that sums two numbers and then multiplies the result by a third, an operation known as MAC. As the pattern is very common, many architectures choose to support it directly in hardware, performing it with one MAC instruction instead of an addition followed by a multiplication, which not only reduces code size but is often faster as well. The result is assembly code, a platform-dependent language close to executable machine code, but still with some abstractions and human-readable.

Next, the assembler consumes the assembly code and outputs non-executable machine code, usually denoted as an object file. The machine code is at this stage memory position independent, meaning that memory addresses are still encoded as relative to the entry point, and the use of unresolved symbols which can be used, for example, to refer to procedures not yet located in the program.

Finally, a linker joins all object files into a single, executable program in machine code, resolving all previously unresolved symbols and finalizing the memory addresses.

The set of these tools, along with a few unmentioned others, such as a debugger, that enable the creation of software constitutes a toolchain.

A Standard Developer Kit (SDK) is usually composed of a toolchain along with other utilities such as libraries, documentation, program examples, APIs, and other tools that ease software development.

3.1.3 S2S Compilation

As the name suggests, has a high-level language as both input and output, which might be the same or a different one. S2S Compilers have seen mainstream use in web development, powering the translation of TypeScript and Svelte to EcmaScript, however their usefulness is not limited to the web.

A S2S Compiler can be used to implement partial code selection, identifying uses of custom instructions and inserting in-line assembly. Delegating the code selection to a tool separate to the compiler means that compiler lock-in is eliminated, meaning that any compiler that accepts the same language as the output of the S2S Compiler is supported. Additionally, updating the compiler to a newer version is trivial. Furthermore, S2S Compiler usually have much smoother learning curves than traditional compilers. These factors result in massively reduced engineering effort required to support custom instructions.

3.1.4 Verilog, HDL and Simulators

Verilog, a popular Hardware Description Languages (HDLs), enables hardware designers to model digital circuits at the Register Transfer Level (RTL). It can be used to automatically generate a list of basic logic primitives such as logic gates in a process called synthesis. The result, denoted as netlist, can then be used to generate FPGA bitstreams or blue prints for ASICs.

Additionally, simulators generated from the description can verify the correctness of the design without needing to invest physical resources. For example, by describing a microprocessor in Verilog, one can then create a Instruction Set Simulators (ISSs), that is a simulator that accurately emulates the values in the register file and their changes, can verify the correctness of an implemented custom ISA extension. If the simulator also accurately simulates the behaviour of the system on a per cycle basis, taking into account pipeline stalls and the like, it is deemed a cycle-accurate simulator.

Verilog and other similar HDLs are a key part in modern hardware design and are extensively used in the state-of-the-art projects aiming to extend the RISC-V ISA.

3.2 Listing

3.2.1 Supporting 64-bit Operands in 32-bit RISC-V Cores

Kumar et al. [1] extended the RISC-V 32 bit ISA with two custom instructions to load and store 64 bit operands. This is implemented by grouping the 32 32 bit general purpose registers into 16 pairs of 64 bit registers, which are then used as operands for the custom instructions. There is also an on-going effort by the RISC-V consortium to standardize such an approach[1]. The ISA extension is implemented on Codasip [16], a framework for designing RISC-V IPs, which also generates a supporting SDK. Although this simplifies the inclusion of compiler hints for

[1] https://github.com/riscv/riscv-zilsd

the custom instructions, the approach is not yet able to optimize generic C code, and explicit consecutive memory accesses must be written at the source code level.

The extension was tested on a single C program compiled and profiled with the proprietary Codasip compiler. The code declares a two dimensional data array, and copies its elements to a distinct target array. The implemented 64 bit memory instructions resulted in a a 50 % reduction in the number of clock cycles, and on 30 % reduction of power consumption during memory and load instruction with only a 4 % area increase. Future work will address support for generic user C code and the `bfloat16` datatype.

3.2.2 Platform for Intelligent Sensor Processing

Bernardo et al. [5] discuss the implementation of a RISC-V core targeting audio event detection (Intelligent Sensor Processing) acceleration for edge applications. The design features a core built upon Minres TGC and an UltraTrail AI hardware accelerator, while the underlying System-on-Chip (SoC) is an adapted version of Pulpissimo [48]. The core features three unspecified custom instructions as well as a zero overhead loop feature encoded using CoreDSL, a language for ISA specification. The UltraTrail accelerator provides an TVM UMA (Universal Modular Accelerator Interface) API, which allows an easy integration of new hardware accelerators into TVM. All instructions were manually implemented as RTL and then integrated into the core using SCAIE-V, an open-source hardware interface custom extensions, while the ISS was based on the CoreDSL. Toolchain related aspects are mentioned, but not detailed regarding the extent of compiler support for the custom instructions. When running an audio event detection application the core was able to achieve a 2.15× speedup and a 27 % power reduction compared to a base implementation without the ISA extension. No code is provided.

3.2.3 NARS

Manoni et al. [35] present NARS, an open-source Spiking Convolutional Neural Network (S-CNN) mapping methodology implemented on a RISC-V core based on Snitch with Indirection Streaming Semantic Registers (ISSR) and FREP, which manages an Floating Point Unit (FPU) sequence buffer that implements hardware loops on floating-point instructions, targeting Spiking Neural Network (SNN) acceleration. Additionally, the Snitch core also supports non standard floating point representations from FP8 to double precision.

Although the code is available[2], the authors do not mention any software or toolchain support. As for the testing, the focus was on the mapping methodology rather than the core or ISA extensions. The project was simulated on QuestaSim and, under the aforementioned conditions, demonstrated speedups from 33× to 10.23× on a dense baseline and from 1.12× to 2.66× on an optimized dense implementation.

[2] https://github.com/smanoni/NARS

3.2.4 RISCV-FNT

RISCV-FNT [11] is a RISC-V based on RI5CY [24] that aims to accelerate Convolutional Neural Networks (CNNs) via the Fermat Number Theorem (FNT), which replaces complex numbers with real numbers in Fast Fourier Transforms (FFTs). The core, which includes an FNT acceleration unit, is written in Verilog and was implemented on an FPGA. The acceleration unit is controlled using custom instructions, features Single Instruction Multiple Data (SIMD) and compute-in-memory capabilities. It can process an 8×8 matrix at once or larger matrices in a sliding window fashion. There is no mention of software or toolchain support, and instead, inline assembly is used to control the unit. A speedup of $8.5\times$ is achieved versus baseline RISC-V processors without FNT – presumably the RI5CY core – though this speedup is specific to the *conv2d* layer. For 1D Convolution (conv1d) the speedup is $6.2\times$, and the overall speedup for the neural network is $3.6\times$, together with a 40 % power consumption reduction.

3.2.5 RV-GEMM

RV-GEMM [59] is a RI5CY [24] based core featuring a coprocessor to accelerate general matrix multiplication, a common operation in AI kernels. The coprocessor implements SIMD operations and is controlled by the core via three instructions custom instructions. It features 16 parallel multiplication units with configurable precision integer operands (32 bit, 16 bit or 8 bit), as well as 8 input data registers, 8 control registers and 16 output data registers for a total of 32 32 bit registers. The custom instructions set the dimensions of the input matrices and operand precision, trigger the calculation of up to 4×4 blocks of the output matrix, and copy these blocks to a target address. The modified core achieved speedups of $15.8\times$, $28.7\times$, and $42.5\times$ for GEneral Matrix Multiplication (GEMM) computations under 32 bit, 16 bit and 8 bit operand precision respectively when compared to the unmodified RI5CY core, while only increasing the overall power consumption by 2.6 %. However, although control and address generation units are mentioned, no detailed explanation is given on how the custom instructions need to be orchestrated to access matrix data which is non-contiguous in memory. Example code is not shown nor is the supporting toolchain mentioned or open-sourced.

3.2.6 RNNASIP

Andri et al. [3] aim to accelerate Recurrent Neural Network (RNN) inference for Radio Resource Management (RRM) applications. In this field, devices typically stay on the field for long periods of time, yet the algorithms evolve at a fast pace. As such, devices must be efficient but adaptable, and the authors consider FPGAs to be too costly for dense and widespread deployment. Instead, three instructions were added to the RI5CY [24], a predecessor core to the CV32E40P developed as part of the PULP platform, with existing support for various ISA extensions. An algorithm was implemented using only the standard IMFC instructions, and re-implemented with resort to the PULP's extensions. By comparing performance results, performance critical operations were

identified that informed the design of the three custom instructions. Namely, instructions which each compute a sigmoid, a hyperbolic tangent, and a combination of a sum-dot-product operation.

When combining the existing PULP extensions with these three custom instructions and software optimizations, the core is on average 15× faster versus code using only the base RISC-V IMFC ISA, and the overall energy efficiency shows a 10× improvement. The baseline implementation is compiled using the standard GCC 7.1.1 for RISC-V RV32IMFC ISA, but any required modifications to support the added instructions are not explicitly explained. However, the code repository hosting the work contains `C` test files for the custom instructions where these are introduced using inline assembly. Thus, to the best of our understanding, the compiler does not automatically optimize programs to use these instructions.

3.2.7 CV32E40P

CV32E40P [40] is a RISC-V core supporting the RV32IMC instruction set, with configurable support for either the F or Zfinx standard ISAs, as well as the CORE-V custom ISA extensions group, containing various ISA extensions developed by PULP which have been shown to accelerate AI workloads. The extensions include instructions for hardware loops, load and store operations with pointer post-increment, various bit manipulation operations, integer MAC operations and SIMD operations, which are common operations in AI kernels. A modified GCC allows for compiler-level support for a subset of the introduced instructions. However, a set of built-ins is also available for the majority of the extensions, eliminating at least the need for manual in-line assembly.

The authors show that for typical Digital Signal Processing (DSP) workloads the proposed core is on average 3.5× faster and 3.2× more energy efficient. Overall, the proposed CV32E40P, as well as PULP – the platform it originated from – are some of the most advanced open-source RISC-V non-standard ISA extensions presently available, and their toolchain support is also mostly unmatched. Despite this, some current compiler-level shortcomings include limited automatic opportunity detection and insertion of some instructions. For example, compiler generated post-increment load and store operations are limited to single-nested loops, and instructions that implement dot-products can only be used explicitly in source-code through built-ins.

3.2.8 Minifloat-NN

Bertaccini et al. [8] extend and open-source FPU called FPnew [33], in order to support an existing extension called *smallFloat* [51]. They then implement, to the best of our understanding, three new custom instructions extending *smallFloat* itself. These are an Expanding Sum-of-dot-product (ExSdotp) operation, as well as expanding and non-expanding three-operand additions named Vector Inner Sum (Vsum) and Expanding Vector Inner Sum (ExVsum), where an "expanding" operation produces an output with wider bit-width than its inputs. Specifically, ExSdotp computes the dot product of `rs1` and `rs2`, and performs a three component sum with the results and the value of `rd`. Vsum computes a three component sum `rs1`, `rs2`, and `rd`. ExVsum computes a three component sum between the packed values in `rs1` with the value in `rd`. The `rd` register functions

as an accumulator for all instructions, and for ExSdotp and ExVsum it contains data packed in higher precision than `rs1` and `rs2`. Variants of the instructions distinguished by different suffixes indicate the precisions of the inputs and outputs, as either 8 bit, 16 bit, or 32 bit.

The FPU accepts 64 bit inputs and 64 bit outputs, and through parametrization during instantiation it can be used to compute SIMD operations over different bit widths. Specifically, the unit can perform up to two 16 bit to 32 bit operations per cycle, or four 8 bit to 16 bit operations per cycle, by packing the inputs and outputs into the full width 64 bit ports. The FPU supports all conventional floating-point operations for every enabled floating-point format, except the square root and division operations.

The proposed FPU achieves the highest energy efficiency when executing low-precision kernels when compared to two state-of-the-art architectures [36, 64] that also support low-precision operations as well as the unmodified FPnew unit, being 1.7× more efficient than one, and 14.4× then the second. It is also 1.3× more efficient than the original FPNew when performing operations of the lowest precision and doubles its performance when using the expanding operations. We have also not identified any other work supporting 8 bit floating-point operations.

However, no details are given on code generation or compiler support to exploit the instructions. Furthermore, performance for higher-precision kernels is not evaluated, and although the FPU is compared to two existing works, the comparison is skewed as data of different precisions is employed for each case. Finally, one of the reference works for comparison outperforms the proposed FPU in terms of throughput by 1.42×.

Chapter 4

Constant Propagation

Constant propagation is a common normalization step in static analysis, facilitating subsequent analyses that require knowledge of the value of variables at certain points in the program. We implemented a simple normalization pass based on Clava and Clava Flow, introduced below, that replaces read `varrefs` with the variable's value at that point in the program whenever it can be determined. A `varref` is the name given to the nodes in Clang's AST and adopted by Clava that correspond to the use of a variable in a C/C++ program, e.g. the `x` in `x+1` or the `y` in `y++`. A `varref` corresponds to a read operation if the value of the variable is used to perform an operation such as in the former example, a write if the value of the variable is changed as a result of an operation, or read *and* write such as the latter example.

4.1 Flow and Clava Flow

Flow[1] is a graphing library for Lara[2] focused on control-flow, data-flow, and static-call analyses built upon Cytoscape[3].

Clava Flow[4], an extension of Flow, enables the construction of control-flow and static-call graphs of C and C++ programs, made possible by parsing said programs with Clava, hence the name.

As we used Clava Flow, we eventually felt the need to add some missing features to suit our needs. The rest of this section details our contributions to the project.

4.1.1 For-Loops with incomplete headers

Previously, when analyzing a program, the library assumed that for-loops always had an initialization, that is, a statement in the loop's header in which variables may be declared and/or initialized or assigned a value. All generated control-flow graphs therefore would follow the same structure:

[1] https://github.com/specs-feup/flow
[2] https://github.com/specs-feup/lara-framework
[3] https://cytoscape.org/
[4] https://github.com/specs-feup/clava-flow

the last statement before the loop would have an outgoing edge to the loop's initialization statement, which would have an outgoing edge to the to the loop's test condition statement, responsible for ascertaining whether the loop's body should be run or not. An example of this can be seen in Figure 4.1. The C and C++ standards, however, allow for-loops without an initialization, in which case an empty statement is used. This resulted in a crash when generating the graph. We modified the library to account for this case. Now, in the case that the initialization step is an empty statement, the last statement before the loop connects to the loop's test condition statement, as can be seen in Figure 4.2.

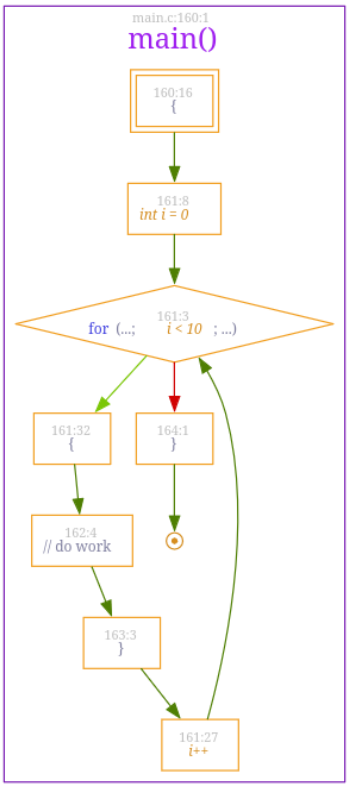

Figure 4.1: A visual representation of the control-flow graph generated of a program with a for-loop with a non-empty initialization and step

Similarly, the library also assumed that the for-loops always had a step, that is, a statement in the loop's header where a control variable is modified. The structure of the for-loops would then be as follows: the last statement of the loop's body would have an outgoing edge to the loop's step statement, which would in turn have an outgoing edge to the loop's test condition statement, observable in Figure 4.1. Once again, the step can be an empty statement according to the C and C++ standard. We modified the library to support such cases, in which case the last step of the loop's body now has an outgoing edge to the loop's test condition statement. Figure 4.3 illustrates this.

Naturally, both of these cases can be true at the same time. Our modified version of the library also behaves as expected in this scenario, as can be seen in Figure 4.4.

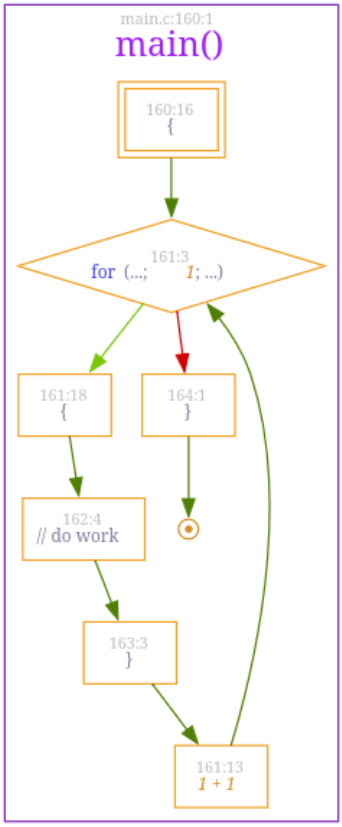

Figure 4.2: A visual representation of the control-flow graph generated of a program with a for-loop with an empty initialization and non-empty step

4.1.2 In-line assembly statements

Different statements affect the generated graph in different ways: a variable initialization simply connects to the next statement in the program, while an `if` statement has two outgoing edges, and a function call connects the caller function to the callee. Additionally, the nodes offer information pertaining to the statements they represent: the user can access the condition of an `if` node, for example, without needing to use Clava's representation of the `if` statement itself.

However, the library did not handle in-line assembly statements, which resulted in crashes whenever they were present in the program. We extended the library, creating an appropriate node for such statements that behaves similarly to other statements that do not alter control-flow. It should be noted that in-line assembly instructions may in fact alter the control-flow of the program. However, Clava and Clava Flow are by design target-agnostic. As such, parsing the in-line assembly and inferring its semantics are naturally outside their scope. It is up to the user to ensure that there are no instructions inside the in-line assembly code that jump to other parts of the program.

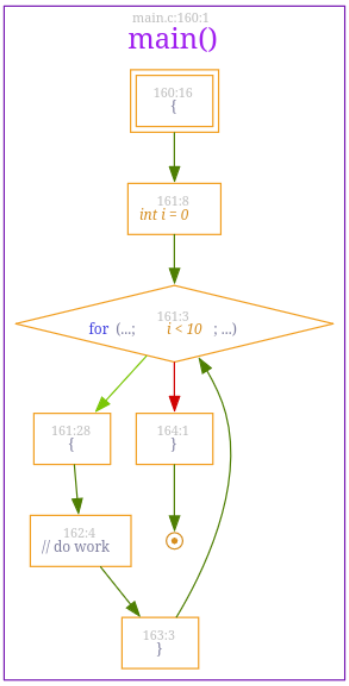

Figure 4.3: A visual representation of the control-flow graph generated of a program with a for-loop with a non-empty initialization and empty step

4.2 Limitations

Our current implementation has some limitations that shape the way we handle certain scenarios, hence the need to mention them first.

4.2.1 Pointers

Our implementation currently ignores pointers completely. This means that, for example, a read on a dereference of a pointer to a variable with a constant value is not substituted. It also means that, if a value is written to a dereferenced pointer, it will not count as a write on the variable it is currently pointing to, possibly producing incorrect results.

4.2.2 Function calls, global variables and C++ references

At the time of writing, Clava Flow does not connect nodes containing function calls to the callee's starting node. As mentioned previously, there is an outgoing edge from the caller to the callee, but it is external to the control-flow of the caller. Usually this does not constitute an issue since variables in the caller's scope cannot be modified in the callee, however, there are some notable exceptions.

First, we have global variables. As the name suggests, they are in-scope during the entirety of the program, and to properly account for writes to these variables, whenever there is a function call, one would need to stray from the current position on the Control-Flow Graph (CFG), traverse the

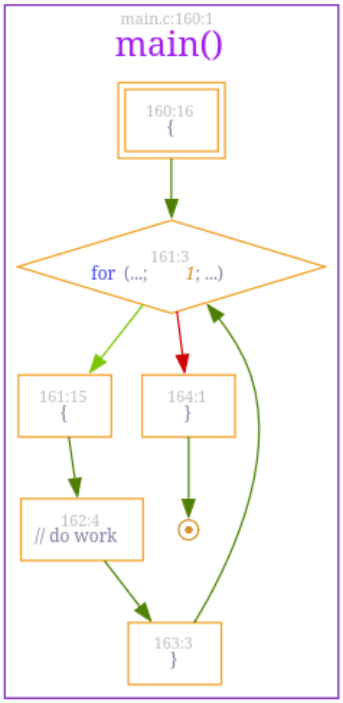

Figure 4.4: A visual representation of the control-flow graph generated of a program with a for-loop with an empty initialization and step

entirety of the callee's graph, and every function call within, repeating until we reach the beginning of the main function, if there is one. For these reasons, we perform constant propagation separately and only on purely constant global variables. First, we check all available code and mark all global variables without any writes and that are initialized with a literal. Then we replace every reference to each marked variable with the literal with which it is initialized.

Passing a pointer as an argument to a function, or of a variable by reference in C++, would also require analyzing the callee's CFG. However, our implementation does not currently support it and may therefore produce incorrect results.

4.2.3 Undefined behaviour

Whenever there are two or more writes to the same variable in the same statement, except when the statement is an assignment to the very same variable, it is considered undefined behavior as per the C and C++ standard. Therefore, whenever our program detects this, it throws an error, alerting the user to the impossibility of performing the necessary analysis on the given source code.

4.2.4 Poor optimization

Due to time constraints, most of the development time was focused on adding features and testing, leaving little time for optimizing the code. As a result, it may take upwards of 120 seconds when propagating constants on programs larger than a few hundred lines of code on a modern computer.

4.3 Algorithm

The first step of the algorithm is to search for every `varref` that can potentially be replaced by a `literal`. To be eligible, the value of the variable must not be altered by the operation, discarding operations such as assignments or post-increments. Furthermore, it cannot be the subject of an `addressof` operator or of a C++ member access. Finally, global variables are also skipped for the reasons outlined in Limitations.

After building the CFG we loop through each eligible `varref` and traverse the graph backwards from the `varref` until we find a write to the variable, the start of the current function or a node that has already been explored. Due to the program's control possibly branching out, e.g. after an `if` statement, a `switch` or a loop, multiple writes have the possibility of being the last. As of the declaration of the `b` variable in Figure 4.5, for instance, both the declaration on line 2 and the post-increment on line 4 may constitute the last write to the variable `a`, depending on the value of `x`. If no writes to the variable are found, then its value cannot be known at compile-time,

```
1 void foo(int x) {
2   int a = 0;
3   if (x) {
4     a++;
5   }
6   int b = a;
7 }
```

Figure 4.5: Example of a program with multiple possible last writes for the variable `a`

therefore the `varref` is not replaced. If one or more writes are found, the replacement can only happen if all writes are assignments of the same `literal`.

Finally, this process repeats until no further replacements are possible. An iteration-by-iteration demonstration of the algorithm is presented in 4.6 and its pseudocode in Figure 4.7.

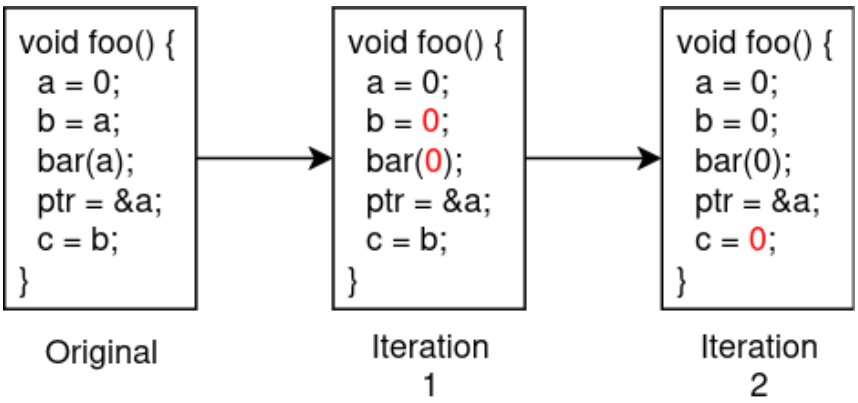

Figure 4.6: Iteration-by-iteration demonstration of the constant propagation algorithm

```
fn propagateConstants(): int {
  totalChanges: int = 0;
  do {
    currentChanges: int = 0;
    candidateVarrefs: Varref[] = getCandidateVarrefs();
    cfg: CFG = buildControlFlowGraph();

    for (candidateVarref: Varref of candidateVarrefs) {
      lastWrites: Expression[] = getLastWrites(cfg, candidateVarref);

      if (lastWrites.isEmpty()) continue;
      if (not areAllLiterals(lastWrites)) continue;
      if (not areAllEqual(lastWrites)) continue;

      candidateVaref.replaceWith(lastWrites.first());
      currentChanges++;
      totalChanges++;
    }
  } while (changes > 0);

  return totalChanges;
}
```

Figure 4.7: Constant propagation algorithm pseudocode

4.3.1 Determining the last writes to a variable at a given point in the program

The procedure that searches for the last writes to a variable accepts a Clava Flow CFG and a Clava `Varref` object as input. The first step is to situate the CFG node where the search should start. Since simple expressions such as `varrefs` do not have a directly corresponding node in a CFG, instead we must find its first ancestor in Clava's AST that does have a directly corresponding node in the CFG. In the example program presented in Figure 4.8, for instance, the `a varref` in line 6 is a descendant of the `foo` function, the function's scope, the `if` statement in line 4, the `if` statement's scope, and finally the expression statement corresponding to `b`'s assignment in line 6, all of which have corresponding nodes in the CFG. However, since the search is done by following the program's control-flow in reverse order, starting the search in any but the last mentioned node would miss the assignment to `a` in line 5. In contrast to the previous example, the first ancestor of the `b varref` in line 4 with a directly corresponding node in the CFG is the `if` statement, and thus should be the start of the search.

```
1 void foo() {
2   int a = 0;
3   int b = 1;
4   if (b) {
5     a = 2;
6     b = a + 1;
7   }
8 }
```

Figure 4.8: Example C program

Upon identifying the node in which the search should start, as previously mentioned, we traverse the CFG in reverse order, inspecting each node for write operations to the variable under analysis. Scope and function nodes are skipped entirely, and loop and `if` nodes only have their headers checked. If a node does have one write operation to the variable under analysis then the value assigned to the variable is added to the last writes list. In the case of an assignment, its right-hand side is added, while unary operations such as pre- or post-increments and decrements or binary operations that read and write to the variable, such as add and assign (+=) or subtract and assign (−=) are added themselves. If a statement has multiple write operations to the variable that aren't on the right-hand side of an assignment to the variable then it is considered undefined behavior and an error is thrown.

In the case that a node does not contain any write operations to the variable, we iterate over all of the node's incoming nodes, i.e. the nodes that have an out-going edge to the current node, and add all of each branch's last writes.

Additionally, nodes that have already been explored once are ignored to prevent infinite loops. A pseudocode version of the algorithm is presented in Figure 4.9.

```
fn lastWrites(cfgNode, variable): Expression[] {
  if (cfgNode.wasAlreadyChecked()) return [];

  writeVarrefs = cfgNode.getWriteVarrefs(variable);
  if (writeVarrefs.length == 1 || cfgNode.isAssignmentOf(variable) {
    return [writeVarrefs[0]];
  } else if (writeVarrefs > 1) {
    throw UndefinedBehaviourError;
  }

  previousLastWrites = []
  for incomingNode of cfgNode.incomingNodes {
    previousLastWrites.append(lastWrites(incomingNode, variable);
  }
  return previousLastWrites;
}

fn getLastWrites(cfg, varref): Expression[] {
  initialNode = cfg.find(varref);

  return lastWrites(initialNode, varref.variable);
}
```

Figure 4.9: Pseudocode of the algorithm for searching for the last writes of a variable

4.4 Constant Folding

Constant folding is a similar pass that simplifies arithmetic expressions by computing them at compile-time, replacing the operation by its result whenever it can be determined. It serves a complementary role in our program, allowing even more robust static analysis.

We combine both the constant propagation developed by us and the constant folding provided Clava Code Transforms[5], a code transformation library for Clava, interweaving each until no more changes are possible. A step-by-step demonstration of the algorithm is presented in Figure 4.10 and a pseudocode version of the algorithm can be seen in Figure 4.11.

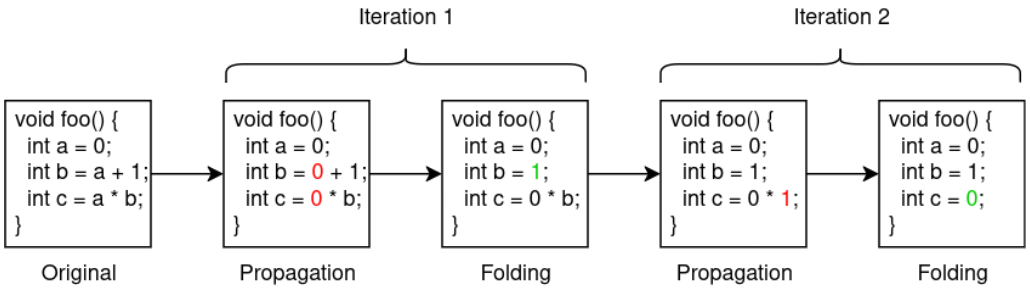

Figure 4.10: Step-by-step demonstration of the constant propagation and folding algorithm

```
import foldConstants from "clava-code-transforms";

fn propagateAndFoldConstants(): int {
  totalChanges: int = 0;

  do {
    currentChanges: int = 0;

    currentChanges += propagateConstants();
    currentChanges += foldConstants();

    totalChanges += currentChanges;
  } while (currentChanges > 0);

  return totalChanges;
}
```

Figure 4.11: Pseudocode of the constant propagation and folding algorithm

[5] https://github.com/specs-feup/clava-code-transforms

Chapter 5

Acceleration of vector-vector dot product loops with a MAC instruction

In this chapter we present an optimization pass built on Clava and Clava Flow that automatically accelerates vector-vector dot products in C/C++ programs using custom instructions supported by Multiplator[45], an unit of which has been integrated into the X-HEEP[34] platform, a state-of-the-art RISC-V based SoC.

Multiplator can perform one 32 bit multiplication in scalar mode and, depending on the configuration, perform two 16 bit, four 8 bit, eight 4 bit, and up to sixteen 2 bit MAC operations in parallel, which can be accessed in C via a set of custom instructions in inline assembly code blocks, which have been given C bindings, a C function for convenience, as can be seen in Figure 5.1. The MAC custom instruction accepts two 32 bit packed operands, and accumulates the result on an internal register, which can be read and cleared via a similar custom instruction, also available with its own C binding. We say that a value is packed whenever we interpret multiple lower bitwidth values that are placed consecutively in memory as a single higher bitwidth value. For instance, treating four consecutive 8 bit numbers as a single 32 bit would be considered packing the numbers by a factor of 4. However, due to pipelining constraints, the MAC custom instruction must always be used in pairs, hence why its C binding accepts four packed operands rather than two.

Our program detects vector-vector dot products in C code that can be accelerated using the MAC instructions and automatically replaces the relevant code with an equivalent version that uses the custom instructions. The modified code outputted by our program can subsequently be compiled with an unmodified C RISC-V compiler such as GCC or Clang.

The simplest program with 8 bit operands that can be accelerated is presented in Figure 5.2. The modified code outputted by our program is presented in Figure 5.3.

As we can see, the loop that performs the multiplications has been replaced by a call to a wrapper function responsible for performing the same computation more efficiently by leveraging the custom instructions. The wrapper function's code has been added to the program, as well as

```
// .insn r opcode6, func3, func7, rd, rs1, rs2
static inline void __mac_8b (signed int a,
 signed int b,signed int c, signed int d) {
 asm volatile (
  ".insn r 0b0001011, 0x02, 0x0, x0, %[RS1], %[RS2]\n"
  // 8b config - MAC - custom0 (Instruction 1)
  ".insn r 0b0001011, 0x02, 0x0, x0, %[RS3], %[RS4]"
  // 8b config - MAC - custom0 (Instruction 2)
  :
  : [RS1] "r"(a), [RS2] "r"(b), [RS3] "r"(c), [RS4] "r"(d)
 );
}
```

Figure 5.1: Macro encoding the pair custom instructions which configure the accelerator for a 4-bit configuration. Configurations for 8 bit, 16 bit, or 32 bit are analogous, by specifying different arguments for *RS*1 and *RS*2.

C bindings for the custom instructions, though the in-line assembly has been omitted for brevity's sake.

A pointer to the accumulator has also been created, and the accumulator is also assigned its own value after the call to the wrapper function. Although seemingly useless, they are necessary for the compiler to know that the value of the accumulator changes. Without them, the compiler would not correctly update the accumulator's value, which becomes relevant in more complex scenarios.

The second for-loop in the wrapper function handles the cases where the input arrays' elements cannot be perfectly iterated over in packed values. All inserted functions use the inline specifier so that function call overhead is avoided as much as possible. The pointer manipulation in the wrapper function is also optimized away by the compiler, also resulting in no overhead.

These functions are always added to every source file, whether they are needed or not. Since they are inline and static, and therefore they are only visible inside their own translation unit, that is, the file they are in, there are no linking problems due to function body redeclarations.

If the user were to configure the pass for other operand bitwidths, e.g. 16 bit, similar functions would have been added instead with slightly different logic, though no code would have been replaced since there are no dot products between arrays whose elements' size is 16 bit, i.e. `short` arrays, in the program.

5.1 Algorithm

The pass starts off by adding forward declarations of the main wrapper function and the bindings for the custom instructions. The bodies of the functions are omitted to avoid spending time unnecessarily analyzing their code while still providing Clava the information needed to create calls to these functions later on.

Next we propagate and fold constants in the program before saving the current state of the AST. We do this so that later, if unnecessary normalizations are made, we can restore the program

```
signed int foo(signed char A[8], signed char B[8]) {
  signed int accum = 0;
  for (int i = 0; i < 8; i++) {
    accum += A[i] * B[i];
  }

  return accum;
}
```

Figure 5.2: Simplest supported C program that our program can optimize

to the state in which it was saved, keeping the outputted program as recognizable to the end-user, in case debugging is necessary.

After building the Clava Flow CFG we instantiate a `LoopReplacer` object. Using Clava's API we iterate over every loop in the program and analyze them using the `LoopReplacer`. `LoopReplacer` will keep track of all loops that can be accelerated based on the chosen operand bitwidth and the information required to transform them.

After restoring the AST, undoing any normalization passes done after the save, `LoopReplacer` applies the transformations to every valid loop it recorded. Finally, we add the missing bodies of the wrapper function and the custom instructions bindings and write the final version of the program to a new file. A high level pseudocode version of the algorithm can be seen in Figure 5.4.

5.1.1 Loop Analysis

Ensuring that a loop can be accelerated is a complex task. We impose some restrictions that, while not strictly necessary, significantly simplify analysis.

For a loop to be considered a valid optimization target it must abide all of the following conditions (restrictions not strictly necessary are marked with an asterisk):

- Be a for-loop*
- Have regular and predictable control flow, which entails:
  - Containing no `breaks`, `returns`, `gotos`* or `continues`* (trivial `continues` and `gotos` are not allowed)
  - The initial value (value to which the loop's control variable is initialized to) must be known to be 0 at compile-time*
  - The step value (amount by which the loop's control variable is modified each iteration) must be an integer, constant and its value must be known to be 1 at compile-time.
  - The control variable must not be modified inside the loop's body*
  - The end value (value to which the control variable is compared with to determine if the loop's body should be run every iteration) must be an integer and constant
- The loop should not have any side effects except for the increase of the accumulator, which entails:

```c
#include <stddef.h>
inline static void __mac_8b(int a, int b, int c, int d);
inline static int __read_clear();
inline static void __mac_wrapper_8b(signed char *A, signed char *B, int volatile *accum, size_t l
int foo(signed char A[8], signed char B[8]) {
   int accum = 0;
   int __accum_var_0HzmJlmu = 0;
   int* __accum_ptr_0HzmJlmu = (int*) &__accum_var_0HzmJlmu;
   __mac_wrapper_8b(A, B, __accum_ptr_0HzmJlmu, 8);
   accum += *__accum_ptr_0HzmJlmu;

   return accum;
}

inline static void __mac_8b(int a, int b, int c, int d) { /* ... */ }
inline static int __read_clear() { /* ... */ }
inline static void __mac_wrapper_8b(signed char *A, signed char *B, volatile int *accum,
                           size_t length) {
  int mac_len = length / 8;
  int *A_cast = (int *)A;
  int *B_cast = (int *)B;

  for (int i = 0; i < mac_len; i++) {
    __mac_8b(A_cast[i * 2], B_cast[i * 2], A_cast[i * 2 + 1],
             B_cast[i * 2 + 1]);
  }

  *accum += __read_clear();

  for (int i = (length / 8) * 8; i < length; i++) {
    *accum += A[i] * B[i];
  }
}
```

Figure 5.3: The simplest supported program after being modified

  - Having no function calls within its body
  - Having one and only one write to a variable (or array) declared outside its body, henceforth referred to as the accumulator

- The accumulator must be either a `varref` or the element of an array. The element must remain the same, however, e.g. if the accumulator is `arr[i]`, then `i` must be constant in the loop
- The accumulator's type must be an integer and its bitwidth must be less than or equal to 32 bit as per the C language specification for the RV32I architecture (i.e. must be a `char`, `short`, `int` or a `long`)
- The accumulator increase must be valid, as described below

Clava exposes the information required to check most conditions listed above. The loop kind, control variable, step value, initial value and end value can be directly accessed via the loop's Clava node. Its API also provides ways to search for every node in the entirety of the program, or

```
fn pass(useSwSim: boolean = false, operandBitwidth: int = 8, silent: boolean = true): void {
  addFunctionHeaders(operandBitwidth);

  propageAndFoldConstants(silent);

  saveAst();
  // normalization passes that shouldn't be in the final outputted code

  cfg = buildCfg();
  loopReplacer = new LoopReplacer(useSwSimulation, operandBitwidth, silent);

  for (loop of Clava.getLoops()) {
    loopReplacer.analyze(loop);
  }

  restoreAst();

  loopReplacer.applyTransformations();

  addFunctionBodies(useSwSimulation, operandBitwidth);
}
```

Figure 5.4: High-level pseudocode of the pass

limit the scope to the descendants of a given node, making the rest of the conditions also relatively simple to check, except for the last.

As listed above, the only variable or array declared outside of the loop body's scope that is written to is considered the accumulator, and the node used for that write is considered the starting point to check for the validity of the increase.

If its parent is a simple assignment, then its left-hand side expression must be the accumulator, and the right-hand side must be the sum of the accumulator and a valid vector multiplication, or a `varref` whose value is equal to the aforementioned sum. In the case that the right-hand side is indeed a `varref`, we use the `getLastWrites` function described in the previous chapter. However, there must be one and only one last write, meaning no branching paths between the use of the `varref` and its last write are allowed. The value of the last write is then used to verify the condition specified above. For the remainder of this section, when we mention that an expression must fit a particular description it is implicit that it can also be a `varref` whose last write matches the description.

In the case that the accumulator's parent is an add and assign operation then its right-hand side must be a valid vector multiplication. Any other type of operation results in the accumulator increase being considered invalid.

A valid vector multiplication is a multiplication between two valid array accesses.

An array access is considered value only if all of the following conditions are met:

- Its elements' bitwidth must be equal to the configured operand bitwidth as per the C language specification for the RV32I architecture, e.g. for operands of 8 bit it must be a `char` array

- The loop's control variable must not appear in any subscript except the last
- The loop's control variable must appear once and only once in the last subscript. The only valid ancestors of the loop's control variable `varref` must be only sums, parenthesis or the array access itself
- If a variable that was declared inside the scope of the loop, except for the control variable, is used in any of the subscripts then its value must be known at compile time so that it can be properly propagated and folded

Upon verifying that a loop meets all the required criteria, the `LoopReplacer` stores the unique AST identifier of the loop, the accumulator's write `varref` and the two array accesses.

5.1.2 Loop Transformation

The `LoopReplacer` iterates over all recorded valid loops and transforms all of them, one at a time.

When transforming a loop it first searches for the loop, the accumulator, and both array access nodes using their stored unique AST identifiers.

The goal is to replace the loop with a call to the wrapper function, which accepts the base pointer of the first and second vectors, a pointer to a temporary integer variable to be used as an output parameter and the length of the vectors.

The idea behind the pointer parameter is to be able to propagate the value which is supposed to be added to the accumulator outside the bounds of the function. In truth, this could be achieved by simply creating a variable inside the wrapper function, and then having it return that value at the end. However, I've had no time to change it, and so it stayed like this. The supposed use of this pointer value can be seen in 5.5. Since the accumulator can have less than 32 bit, we cannot

```
void foo(char A[], char B[], size_t len) {
  short accum = 0;
  for (;;) {
    int accum_proxy = 0;
    int* out_param = &accum_proxy;
    __wrapper_mac_8b(A, B, out_param, len);
    accum = *out_param;
  }
}
```

Figure 5.5: High-level pseudocode of the pass

directly pass its pointer to the wrapper function. Additionally, the write to the accumulator must be done with a dereference to the `out_param` pointer, and never the `accum_proxy`. This is because if the pointer is never used again, the compiler assumes that its value is unused, and therefore optimizes away the writes inside the wrapper function, even though it is specified to be a volatile pointer.

To find the base pointer of one of the vectors we start with the array access and remove the control variable's `varref` from its last subscript; the base pointer is then the address of the remainder of the array access. If nothing is left on the subscript, it is removed entirely and it is not necessary to take the address of the resulting expression. For instance, if one of the original array access was `A[i][j]`, where `j` is the control variable, the base pointer is `A[i]`, but if the original array access was `A[i][j + 2]` then the base array pointer would be `&(A[i][2])`.

Finally, we leverage Clava's API to replace the loop with the call to the wrapper function and the required operations with the pointer to the accumulator.

5.2 Integration into the compilation flow

The way the pass is written allows the user to use it in a multitude of ways. All of the logic of the pass is encapsulated into a function. When that function is called, whatever is on the AST at that moment is analyzed and subsequently transformed as described above. The user can, therefore, write a TypeScript program that accepts, for instance, the path to a file and an output path, parse the file with Clava, call the pass function, and then use Clava again to write the resulting program to the output path. The pass function accepts three parameters. The first dictates whether in-line assembly should be used in the C bindings or use C code that simulates their behavior. This way, the user may test their program without needing to compile for and run on a RISC-V processor. The second configures the number of bits on the operands, though due to time constraints only 8 works as intended. The third and final argument dictates whether debug information should be printed as the analyses are being performed. Great care has been put into these debug messages, so that the user can easily tell why any given loop is not valid. If the loop is valid, though, only a confirmation message is printed and no further information is given. An example of these debug messages can be seen in Figure 5.6.

The main problem with using the pass in the manner I have just described, however, is that it is not practical for any project larger than a handful of files. Any decently large C/C++ project is most likely using a build system like Make or CMake to manage the complexities of compiling the project, keeping track of necessary flags and includes and such. The most practical way to integrate a pass, then, is to include it directly in the build system.

Fortunately Clava has a CMake package[1] that does precisely that. After it is installed, one can incorporate a Clava pass into the compilation flow by adding the following lines into the CMakeLists.txt file:

```
find_package(Clava REQUIRED)
clava_weave(<TARGET_NAME> <CLAVA_SCRIPT>)
```

5.3 Limitations

The pass currently has some limitations that are worth keeping in mind:

[1] https://github.com/specs-feup/clava/tree/master/CMake

- Cannot parse in-line assembly correctly due to limitation in Clava
- Does not accept code with for-loops without conditions due to limitation in Clava Flow
- The wrapper function and C bindings are inserted with fixed names. If the program happened to have a variable or function with the same name, the program will crash
- While and for-each loops are not supported
- The "tail" loop at the end of the wrapper function is always inserted, even if unnecessary
- Does not support for-loops whose `initValue` isn't 0, even though it could theoretically be supported
- 8-bit was the prioritized during development, so despite the program having been built in a way that the number of bits of the operand can be customized, 16, 4 and 2 bit operand wrapper functions and tests haven't been added, and they are therefore unsupported
- Cumbersome to integrate into complex build systems other than CMake
- `goto` instructions that do not skip any part of the loop, e.g. a `goto` that jumps to the next instruction, essentially not doing anything, are considered disruptive to the loop's control-flow even though they are not
- continues at the end of the loop body should also not count as disruptive to the loop's control flow, and other similar cases
- Regression testing is minimal due to minimal test suite
- Poorly optimized code

```
Propagated & folded constants a total of 0 times
Analysing Loop [13:4]
    Loop is invalid since its init value is not 0 but «1»
      (currently unsupported)

Analysing Loop [16:4]
    Loop is invalid since its step value is not 1 but «-1»
      (memory accesses must be continuous)

Analysing Loop [20:4]
    endValue is a variable that is not constant inside the
      loop: variable += A[i] * B[i]

    Loop's end value is not constant

    Loop is invalid since its control flow is not regular

Analysing Loop [23:4]
    «A[i] * B[i]» [24:15] is not a sum, therefore it is not
      a valid accumulator increase

    «accum = A[i] * B[i]» [24:7] is not a valid accumulator
      assignment of type 'simple assignment'

    Loop is not valid since it does not contain a valid
      accumulator assignment

Analysing Loop [26:4]
    «B[i]» [27:23] of type «short» is not of a valid type
      for the elements of one of the vectors, since its bit
      size is «16» in rv32, and the hardware instructions expect
      operands with a bit width of «8»

    «A[i] * B[i]» [27:16] is not a valid vector multiplication
      since its operands are not valid array accesses

    «accum += A[i] * B[i]» [27:7] is not a valid accumulator
      assignment of type 'add and assign'

    Loop is not valid since it does not contain a valid accumulator assignment

Analysing Loop [33:4]
    Loop is valid

---------------

Transforming Loop with id 0x61ebdcc2e2c0_1
    Transformed Loop [33:4]

Applied transformations to 1 loops
```

Figure 5.6: Example output of running the pass with debug enabled

Chapter 6

Evaluation and Testing

6.1 Constant Propagation

6.1.1 Experimental Setup

The hardware and software specifications of the system in which the constant propagation normalization pass was entirely developed and tested can be found in Table 6.1 and Table 6.2, respectively.

6.1.2 Methodology

We developed a comprehensive suite of unit tests written in Jest[1], a mature testing framework for JavaScript and TypeScript. Every unit test follows the same structure:

1. Parse a C program with Clava, generating an AST
2. Run the constant propagation, altering the AST
3. Verify that the AST has been altered correctly, as described below

The C programs used in the tests cover a wide range of scenarios, from a simple program with linear control-flow containing only a few variable assignments to programs with complex, branching control-flow, involving multiple nested loops. A simple and a complex program used for testing can be seen in Figure 6.1 and Figure 6.2, respectively.

[1] https://jestjs.io/

Component	Specification
CPU	AMD Ryzen 7 4700U
Integrated GPU	AMD ATI Radeon RX Vega 6
Memory	16GB

Table 6.1: Constant Propagation and Loop Analysis Experimental Setup Hardware Specification

Software	Version
Operating System	Ubuntu 24.04.3 LTS
NodeJS	22.16.0
GCC	13.3.0

Table 6.2: Constant Propagation and Loop Analysis Experimental Setup Software Specification

The tests succeed if all `varrefs` whose values could be determined at compile time, except for the cases outlined in the limitations section, have in fact been replaced by the correct literal, and all `varrefs` whose value cannot be determined have not.

6.1.3 Results

All tests on the suite succeed, and Jest's coverage report indicates 100% statement coverage, 93.1% branch coverage, 100% function coverage and 100% line coverage on the code developed strictly for constant propagation, that is, not counting code that is wholly unrelated to constant propagation (e.g. the acceleration pass' code), or general helper functions developed and tested separately, but that are still under the same project.

6.2 Vector-Vector Dot Product Acceleration

6.2.1 Loop Analysis

As described in the previous section, loop analysis consists of analyzing C programs and identifying which loops can be safely accelerated using the MAC custom instruction. Although the analysis logic is not independent of the transformation logic, since the latter dictates what constitutes a valid acceleration target, the correctness of the analysis can be tested independently. In other words, we can test whether the acceleration targets are being identified correctly without having to run or even compile the transformed code.

6.2.1.1 Experimental Setup

The loop analysis part of the pass was developed and tested in the same experimental setup as the constant propagation pass. The system's hardware and software specifications can be found in Table 6.1 and Table 6.2, respectively.

6.2.1.2 Methodology

Previously, we introduced the simplest C program our pass should be able to transform, shown in Figure 5.2. At this point, we tested the program by running the analysis and verifying that a valid loop was found. As we started to support more cases, we updated the C program so that it included loops that test each scenario, and increased the number of valid loops that had to be found accordingly. An excerpt of the current version of the test program can be seen in Figure 6.3.

```
int foo() {
  int a = 5;
  int b = a; // 'a' expected to be replaced with '5'
  int c = a; // 'a' expected to be replaced with '5'
  int d = b; // 'b' expected to be replaced with '5'
  int e = d; // 'd' expected to be replaced with '5'

  return e;  // 'e' expected to be replaced with '5'
}
```

Figure 6.1: Program with simple control-flow used in constant propagation unit tests

In addition to the hand-written test cases we also tested the loop analysis on PolyBench[2] using Clava Lite Benchmarks[3], which we modified to suit our purposes.

6.2.1.3 Clava Lite Benchmarks

Clava Lite Benchmarks is a benchmark suite loader for Clava projects, allowing the user to test their passes across a wide array of scenarios in a convenient manner. Rosetta[4], SPEC2017[5] and CortexSuite[6] are some of the suites currently supported. The suites are composed of one or more applications, and each application has its own C program.

When the user is loading a benchmark suite, the library parses its source files as needed. To facilitate this process, each suite has its files organized in a similar way, and the details are specified in a JSON-like description of the suite. In general, every suite has its own dedicated directory under a directory named "benchmarks". Each application also has its own directory, placed directly under the suite's directory, inside which lay all the C files that the program is composed of. An example of this file structure can be seen in Figure 6.4.

We added support for PolyBench version 4.2 for the C language. Contrary to previously existing benchmarks, PolyBench aggregates logic common to all applications in a single, separate source file. As a result, every application includes its header, and the implementation file should be compiled alongside the rest of the applications' files. As mentioned previously, the library expected that every file of a given application was included in its directory, which meant that this common file would be duplicated once for every application in the suite. Instead, we added support for including extra source files that do not need to be in the same directory as the rest of the application. To achieve this, the library maintainer can now mention the extra source file(s)' path, or their parent directory, in the application's JSON-like description and the files are now loaded as if they were in the same directory as the other source files.

To load a benchmark supported by the library, the end-user first imports a TypeScript object corresponding to the benchmark they wish to use and then passes it as an argument to a load function, both provided by the library.

[2] https://www.cs.colostate.edu/~pouchet/software/polybench/
[3] https://github.com/specs-feup/clava-lite-benchmarks
[4] https://github.com/cornell-zhang/rosetta
[5] https://www.spec.org/cpu2017/
[6] https://michaeltaylor.org/cortexsuite/

```
int main() {
  int a = 0;
  int changed = 1;
  for (int i = 2; i < 3; i++) {      // 'i' in 'i < 3' expected to not be replaced
    int b = a;                       // 'a' expected to be replaced with '0'
    int c = b;                       // 'b' expected to be replaced with '0'

    int d = changed;                 // 'changed' expected to not be replaced
    int e = d;                       // 'd' expected to not be replaced

    for (int j = 5; j < 8; j++) {    // 'j' in 'j < 8' expected to not be replaced
      changed = 13;
    }
  }
}
```

Figure 6.2: Program with complex control-flow used in constant propagation unit tests

As a library maintainer, one defines these objects in the JSON-like description mentioned previously, an example of which can be seen in Figure 6.5. The "flags" field allows maintainers to specify which flags should be used for every application in the suite, while the optional "extraFlags" field in each application's declaration controls additional flags that should only be used when loading that particular application.

Some benchmark applications allow for the configuration of the size of the dataset used. The "trmm" application of PolyBench, for instance, performs computations over two matrices, the sizes of which can be configured. This is done by defining a certain preprocessor macro, e.g. MEDIUM_DATASET, which can be achieved by passing a flag to the compiler ("-DMEDIUM_DATASET").

Various challenges arise when trying to support this kind of configurable applications in the framework already provided by the library. Firstly, it should be noted that the load functions do not allow the end-user to specify additional flags to pass to the compiler when loading the applications. The problem could be solved by allowing this, however it introduces a new problem: the user must be aware of the configurability of the dataset sizes through flags and which flags are allowed by each application, at which point the end-user would naturally turn to the library's documentation to seek out such information. The maintainers would thus need to be tasked with maintaining these details accurate and up-to-date in the documentation, an important but nonetheless easy-to-miss task when maintaining a library. Additionally, the maintainer would have to create multiple copies of the same benchmark for each possible configuration flag, adding up to a considerable level of code repetition. The whole process would quickly become error prone and we argue it would stray from the library's convenience.

Instead, we encode the necessary information directly into the objects exposed to the end-user. Leveraging TypeScript's type system, the information about which dataset sizes are valid for a given suite can be encoded directly into its type, not only immediately presenting the information to the end-user, but also catching errors at compile-time rather than run-time. In addition, it provides an abstraction layer for the configuration of the dataset size. The end-user is not required to know how it works internally, and if in the future the library is to support a suite whose dataset

size is not configured with flags then it can be done internally without it concerning the end-user. We created a base, abstract `BenchmarkSuite` class that accepts one generic type argument, representing the dataset sizes its applications accept. A library maintainer then extends this base class for every suite supported by the library, restricting it to a specific type for its dataset sizes. In case one of the applications doesn't support one of the dataset sizes, it can be specified in the application's declaration. Finally, the maintainer instantiates this class and makes it available for the end-user, resulting in a not so different experience for both the maintainer and the end-user, which can be seen in Figure 6.6.

At run-time, the load function accepts a suite and a list of dataset sizes. Then, it iterates first over the dataset sizes and subsequently over the applications of the suite, loading each of them. If an invalid dataset size and application combination is found, it is not loaded. An example workflow is presented in Figure 6.7

Similarly to input sizes, PolyBench supports configuration of the data type used in its applications via C macros. However, this support was limited to integers, floats and doubles. Since the MAC custom instruction only supports operands whose bitwidths are lower than 32 bit, we needed to extend PolyBench's data type configurability so that the data type is supported by our pass. We edited the header files of the applications so that they could now be configured by a DATA_TYPE_IS_CHAR macro. The changes are as follows: the program now only defines DATA_TYPE_IS_DOUBLE, the default data type, if DATA_TYPE_IS_CHAR is also not defined. Additionally, if the DATA_TYPE_IS_CHAR macro is defined, we define the macros that the application requires with appropriate values. The modified portion of the header files can be seen in Figure 6.8. Finally, we add the -DDATA_TYPE_IS_CHAR flag to the PolyBench suite declaration in Clava Lite Benchmarks so that the selected data type is always `char`.

6.2.1.4 Results

We ran our loop analysis, including the necessary constant propagation and folding beforehand, on the whole PolyBench suite 10 times, the results of which we have summarized in Table 6.3. Upon manually inspecting every application, we confirmed that the analysis correctly identified the only loop, the one in the `atax` application, that fit all the criteria for optimization. There were no false positives.

On average, the pass took between 10.66 and 19.45 seconds depending on the application with a significantly higher standard deviation occurring in the first two.

6.2.2 Loop Transformation

The correctness of a transformation necessarily requires that the modified program is compilable and runnable on the target platform and that the behavior of the program is not modified in any noticeable way.

App name	Average Duration (ms)	Duration Stdev (ms)	Suitable Loops Found	Correct
2mm	1706.4	298.4	0	Yes
3mm	1897.0	261.4	0	Yes
adi	1622.6	73.0	0	Yes
atax	1449.6	85.2	1	Yes
bicg	1248.1	83.5	0	Yes
cholesky	1541.4	77.3	0	Yes
correlation	1500.5	98.9	0	Yes
covariance	1338.7	87.7	0	Yes
deriche	1945.5	66.1	0	Yes
doitgen	1494.2	78.8	0	Yes
durbin	1116.0	82.3	0	Yes
gemm	1400.5	84.7	0	Yes
gemver	1516.6	92.5	0	Yes
gesummv	1147.8	84.2	0	Yes
gramschmidt	1515.0	60.2	0	Yes
heat-3d	1592.8	102.0	0	Yes
jacobi-1d	1052.3	119.8	0	Yes
jacobi-2d	1297.6	136.0	0	Yes
lu	1595.6	99.4	0	Yes
ludcmp	1850.6	63.2	0	Yes
mvt	1267.5	77.9	0	Yes
seidel-2d	1106.4	81.2	0	Yes
symm	1354.8	101.4	0	Yes
syr2k	1317.2	107.6	0	Yes
syrk	1256.0	124.9	0	Yes
trisolv	1066.0	99.8	0	Yes
trmm	1193.7	80.1	0	Yes
Entire Suite	**74152.4**	**6266.4**	**1**	**Yes**

Table 6.3: Results of running loop analysis on the PolyBench 4.2 benchmark, N=10

Software	Version
Operating System	Ubuntu 22.04.5
NodeJS	22.16.0
GCC	11.4.0
RISC-V GCC[7]	11.1.0 (tag 2022.01.17)
Python	3.10.12
Verilator	4.210
X-HEEP	Commit 2c6d7d8

Table 6.4: Loop Transformation Experimental Setup Software Specification

6.2.2.1 Experimental Setup

To evaluate the correctness of the transformation, we ran programs in an `x86` computer, simulating the behavior of the custom instructions with software, and on an FPGA.

The setup in which we ran the software simulated version is the same as in the previous sections.

The setup to run the program in the FPGA, however, differed significantly. The software specifications of the machine used to run our automatic acceleration pass, cross-compile the modified C code to RISC-V and program the FPGA is presented in Table 6.4. The board used to run the custom instructions was a PYNQ-Z2 programmed with a closed-source IP.

6.2.2.2 Methodology

To test the loop transformation logic during development we wrote a C program with 4 functions. Each calculates the dot product between two vectors, the sizes and contents of which vary per function, and writes the result to standard output. The code is shown in Figure 6.9. We then write a simple TypeScript program that applies the pass, making sure to enable the software simulation mode, which transforms the loops and adds the code that simulates the hardware instructions. Some excerpts are presented in Figure 6.10. The final step is to compile and run both the original and the modified version of the program and compare their output.

To test the loop transformation logic using the hardware instructions and measure the performance gain we adapted a program written by the developers of the IP. Originally, the program compared the number of clock cycles that it took to perform the same matrix-vector multiplication with and without the use of custom instructions, and with and without loop unrolling, all written by hand. The algorithm without the use of custom instructions and without loop unrolling is presented in Figure 6.11, and the algorithm incorporating the use of custom instructions but without loop unrolling is presented in Figure 6.12.

The program also compared the result of all algorithms to ensure correctness.

Unfortunately, due to the program making use of in-line assembly, which presently isn't handled well by Clava, the complex build system of X-HEEP and lack of time made the program incompatible with our automatic acceleration pass. Instead, we isolated the important part of the program, the matrix-vector multiplication with no custom instructions and no loop unrolling, ran

	Original	Modified
foo	50	50
bar	19	19
baz	29	29
foobar	46	46

Table 6.5: Comparison between the output of each of the functions of the original and modified versions of the software loop transformation testing program

the automatic acceleration pass and then inserted the transformed algorithm back into the original program, along with the automatically added wrapper function and C bindings depicted in Figure 6.13. The output vector was also changed so as to not overwrite the original loop's results. Due also to a lack of time, the hand-written algorithm that made use of custom instructions and loop unrolling had to be removed.

Finally, using X-HEEP's build system, we cross-compiled the program targeting the X-HEEP SoC with the closed-source IP programmed into the PYNQ-Z2 and ran the program.

6.2.2.3 Results

As we can see in Table 6.5, the software-emulated loop transformation does not alter the result of the dot product, even when the size of the input vectors is not a multiple of the size of the packed data that the custom instructions accept.

The results of the program that runs on the FPGA are presented in Table 6.6. As expected, loop unrolling is significantly faster than the baseline, using around 2.1 times less clock cycles. Both versions of the algorithm that make use of the custom instructions are around 7.1 times faster than the baseline, though unexpectedly the algorithm with the automatically inserted custom instructions uses five fewer instructions than hand-written algorithm.

	Clock Cycles	Correct
No Custom Instr No Loop Unroll	502217	N/A (Baseline)
No Custom Instr w/ Loop Unroll	238858	Yes
Hand Written Custom Instr	69898	Yes
Automatically Added Custom Instr	69893	Yes

Table 6.6: Clock Cycle and Correctness comparison between the four Matrix-Vector multiplications run in the FPGA

```
void foo(int8_t vector[8], int8_t matrix_col[8]) {
  int32_t result = 0;

  // result += vector * vector
  for (size_t i = 0; i < 8; i++) {
    result += vector[i] * matrix_col[i];
  }

  // result = result + vector * vector
  for (size_t i = 0; i < 8; i++) {
    result = result + vector[i] * vector[i];
  }

  // result = result + multp
  for (size_t i = 0; i < 8; i++) {
    int multp = vector[i] * vector[i];
    result = result + multp;
  }

  // single assignment, 1 op per assignment
  for (size_t i = 0; i < 8; i++) {
    int vec1 = vector[i];
    int vec2 = matrix_col[i];
    int multp = vec1 * vec2;
    int temp = multp + result;
    result = temp;
  }

  /* ... */
}

void bar(int8_t matrix[3][8], int8_t vector[8]) {
  for (int i = 0; i < 3; i++) {
    int accum = 0;

    for (int j = 0; j < 8; j++) {
      accum += matrix[i][j] * vector[j];
    }
  }

  /* ... */

}

void baz(int8_t matrix[3][8], int8_t vector[8], int8_t out_vec[8]) {
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 8; j++) {
      out_vec[i] += matrix[i][j] * vector[j];
    }
  }

  /* ... */

}

void foobar(int8_t matrix[24], int8_t vector[8], int8_t out_vec[8]) {
  for (int i = 0; i < 3; i++) {
    for (int j = 0; j < 8; j++) {
      out_vec[i] += matrix[i*8+j] * vector[j];
    }
  }

  /* ... */
}
```

Figure 6.3: Excerpt of the program used for testing the loop analysis

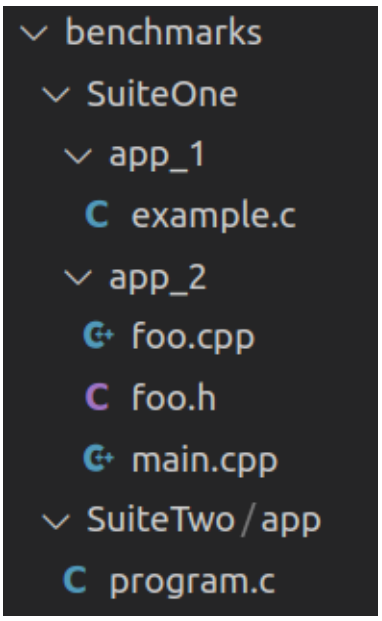

Figure 6.4: A possible file structure for two benchmark suites of Clava Lite Benchmarks

```
export const MyBenchmarkSuite = {
  name: "Suite One", // for debug purposes
  path: "SuiteOne/", // the suite's directory's name
  apps: {
    "app 1": {
      standard: "c11",
      topFunction: "main",
      canonicalName: "app_1", // the app's directory's name
      extraFlags: ["-DMEDIUM_DATASET"],
    },
    "app 2": {
      standard: "c11",
      topFunction: "foo",
      canonicalName: "app_2"
    },
  flags: ["-lm"],
};
```

Figure 6.5: Example definition of a benchmark suite in Clava Lite Benchmarks

```
export enum MySuiteSize {
  SMALL = "-DSMALL_DATASET",
  MEDIUM = "-DMEDIUM_DATASET",
  LARGE = "-DLARGE_DATASET"
}

class MySuite extends BenchmarkSuite<MySuiteSize> {
  public constructor() {
    super(
      "Suite One",
      MySuiteSize,
      "SuiteOne/",
      {
        "app 1": {
          standard: "c11",
          topFunction: "main",
          canonicalName: "app_1",
          invalidInputSizes: [] // accepts all sizes i.e. SMALL, MEDIUM and LARGE
        },
        "app 2": {
          standard: "c11",
          topFunction: "foo",
          canonicalName: "app_2",
          invalidInputSizes: [MySuiteSize.SMALL] // accepts only MEDIUM and LARGE
        },
      },
      ["-lm"]
    )
  }
};

export const MyBenchmarkSuite = new MySuite();
```

Figure 6.6: Example definition of a benchmark suite in the updated Clava Lite Benchmarks

```
for (const res of loadSuite(MyBenchmarkSuite, MySuiteSize.MEDIUM, MySuiteSize.LARGE)) {
  const name = res.appSummary.canonicalName;

  if (res.success) {
    console.log(`Loaded app ${name}}`);

    // user's code
  }
  else {
    console.log(`Failed to load app: ${name}`);
  }
}
```

Figure 6.7: Example workflow using the updated Clava Lite Benchmarks

```
/* Default data type */
# if !defined(DATA_TYPE_IS_INT) && !defined(DATA_TYPE_IS_FLOAT) && \
!defined(DATA_TYPE_IS_DOUBLE) && !defined(DATA_TYPE_IS_CHAR)
#  define DATA_TYPE_IS_DOUBLE
# endif

#ifdef DATA_TYPE_IS_CHAR
#  define DATA_TYPE signed char
#  define DATA_PRINTF_MODIFIER "%d "
#  define SCALAR_VAL(x) (signed char) x
#endif
```

Figure 6.8: Modified section of PolyBench applications' header files

```c
#include <stdio.h>

signed int foo() {
  signed char A[8] = {0, 1, 2, 3, 2, 4, 6, 7};
  signed char B[8] = {2, 4, 5, 1, 1, 3, 2, 1};

  signed int accum = 0;
  for (int i = 0; i < 8; i++) {
    accum += A[i] * B[i];
  }

  return accum;
}

signed int bar() {
  signed char A[5] = {0, 1, 2, 3, 2};
  signed char B[5] = {2, 4, 5, 1, 1};

  signed int accum = 0;
  for (int i = 0; i < 5; i++) {
    accum += A[i] * B[i];
  }

  return accum;
}

signed int baz() {
  signed char A[10] = {0, 1, 2, 3, 2, 1, 1, 0, 1, 2};
  signed char B[10] = {2, 4, 5, 1, 1, 0, 1, 3, 2, 3};

  signed int accum = 1;
  for (int i = 0; i < 10; i++) {
    accum += A[i] * B[i];
  }

  return accum;
}

signed int foobar() {
  signed char A[16] = {0, 1, 2, 3, 2, 1, 1, 0, 1, 2, 1, 2, 1, 2, 1, 3};
  signed char B[16] = {2, 4, 5, 1, 1, 0, 1, 3, 2, 3, 2, 1, 0, 1, 2, 3};

  signed int accum = 1;
  for (int i = 0; i < 16; i++) {
    accum += A[i] * B[i];
  }

  return accum;
}

int main(void) {
  printf("foo: %d\n", foo());
  printf("bar: %d\n", bar());
  printf("baz: %d\n", baz());
  printf("foobar: %d\n", foobar());
}
```

Figure 6.9: Loop Transformation Test Program

```
int foo() {
   signed char A[8] = {0, 1, 2, 3, 2, 4, 6, 7};
   signed char B[8] = {2, 4, 5, 1, 1, 3, 2, 1};
   int accum = 0;
   int __accum_var_nJFT7ctf = 0;
   int* __accum_ptr_nJFT7ctf = (int*) &__accum_var_nJFT7ctf;
   __mac_wrapper_8b(A, B, __accum_ptr_nJFT7ctf, 8);
   accum += *__accum_ptr_nJFT7ctf;

   return accum;
}

/* ... */

static signed int __sim_accum = 0;

inline static void __mac_8b(int a, int b, int c, int d) {
  signed char *a_cast = (signed char *)(&a);
  signed char *b_cast = (signed char *)(&b);
  signed char *c_cast = (signed char *)(&c);
  signed char *d_cast = (signed char *)(&d);

  __sim_accum += a_cast[0] * b_cast[0];
  __sim_accum += a_cast[1] * b_cast[1];
  __sim_accum += a_cast[2] * b_cast[2];
  __sim_accum += a_cast[3] * b_cast[3];

  __sim_accum += c_cast[0] * d_cast[0];
  __sim_accum += c_cast[1] * d_cast[1];
  __sim_accum += c_cast[2] * d_cast[2];
  __sim_accum += c_cast[3] * d_cast[3];
}

inline static int __read_clear() {
  int temp = __sim_accum;
  __sim_accum = 0;

  return temp;
}

inline static void __mac_wrapper_8b(signed char *A, signed char *B, volatile int *accum,
                                    size_t length) {
  int mac_len = length / 8;
  int *A_cast = (int *)A;
  int *B_cast = (int *)B;

  for (int i = 0; i < mac_len; i++) {
    __mac_8b(A_cast[i * 2], B_cast[i * 2], A_cast[i * 2 + 1],
             B_cast[i * 2 + 1]);
  }

  *accum += __read_clear();

  for (int i = (length / 8) * 8; i < length; i++) {
    *accum += A[i] * B[i];
  }
}
```

Figure 6.10: Modified Loop Transformation Test Program

```
static inline void MxV_sw(void) {
  for (int i=0; i<OV_LEN; i++) {
    ov_elem_t accum = bias[i];
    for (int j=0; j<IV_LEN; j++) {
      accum += in_mat[i][j] * in_vec[j];
    }
    out_vec_sw[i] = accum;
  }
}
```

Figure 6.11: Matrix-Vector multiplication algorithm without custom instructions or loop unrolling

```
static inline void MxV_custom_8b(void) {
  for (int i = 0; i < OV_LEN; i++) {
    // Initialize result vector with biases
    out_vec_custom_instr[i] = bias[i];
    for (int j = 0; j < PACKED_LEN; j=j+2) {
      // Perform MAC operation with pre-packed values
      mac_8b_custom(packed_in_mat[i][j], packed_in_vec[j], packed_in_mat[i][j+1], packed_in_vec[j
    }
    // Read and clear accumulator
    out_vec_custom_instr[i] += read_clear_custom();
  }
}
```

Figure 6.12: Hand-made Matrix-Vector multiplication algorithm with custom instructions but no loop unrolling

```
#include <stddef.h>
inline static void __mac_8b(int a, int b, int c, int d);
inline static int __read_clear();
inline static void __mac_wrapper_8b(signed char *A, signed char *B, int volatile *accum, size_t l
inline static void MxV_Automatic_HW() {
  for(int i = 0; i < 64; i++) {
    int accum = bias[i];
    int __accum_var_Bj4x8QAu = 0;
    int* __accum_ptr_Bj4x8QAu = (int*) &__accum_var_Bj4x8QAu;
    __mac_wrapper_8b(in_mat[i], in_vec, __accum_ptr_Bj4x8QAu, 784);
    accum += *__accum_ptr_Bj4x8QAu;
    out_vec_auto_hw[i] = accum;
  }
}

inline static void __mac_8b(int a, int b, int c, int d) {
/* ... */
}

inline static int __read_clear() {
/* ... */
}

inline static void __mac_wrapper_8b(signed char *A, signed char *B, volatile int *accum,
                          size_t length) {
/* ... */
}
```

Figure 6.13: Matrix-Vector multiplication algorithm with automatically inserted custom instructions

Chapter 7

Conclusion and Future Work

This thesis explored the viability of using source-to-source compilation to automatically map certain operations to specialized hardware via the insertion of custom RISC-V instructions. Though there are many state-of-the-art hardware accelerators making use of RISC-V's extensible, open-standard ISA, few provide compiler support for their instructions, possibly due to the steep learning curve of modifying a traditional compiler, or its costly nature.

We have not only shown that it is possible to implement robust analyses capable of detecting optimization opportunities in real-world benchmarks and significantly decrease execution time in specific cases, but that it is possible to do it while building upon the others' work, and at the same time enabling others to build upon your own.

With the experience we have acquired thus far, we can now answer the research questions we previously proposed.

RQ1 *What are the main state-of-the-art RISC-V ISA extensions targeting AI acceleration, how accessible are their development environments and how much effort does it take to simulate their custom instructions?*

A1 PULP stands out as the largest and well adopted family of ISA extensions. Their level of compiler support is, as far as we know, unheard of in the RISC-V ecosystem, and there are multiple IPs that support their ISAs. With X-HEEP's out-of-the-box support for many of their cores, and their relatively recent Docker image available on Docker Hub[1], the developer environment has become tremendously more accessible.

RQ2 *Is it possible to complement the current compilation flow of RISC-V with a S2S Compiler in a way that reduces effort, increases ease-of-use or smoothens the learning curve?*

A2 Definitely. Not only is Clava plenty powerful by itself, there is already a budding ecosystem growing around it. Although those tools are not yet mature, it also means that you can easily extend their functionalities and shape them however it suits your needs.

[1] https://hub.docker.com/r/luigi2898/x-heep

RQ3 *Can source code analysis discover opportunities for custom AI instructions, enabling optimization passes without the need of using large compiler frameworks such as GCC or LLVM?*

A3 Absolutely. AI workloads are in no small part matrix operations, and so AI instructions tend to be relatively similar. The work we have presented proves that it is possible to detect and accelerate this kind of workloads.

7.1 Future Work

There are many possible paths in which this work can be expanded upon. Both constant propagation and the dot product passes are ripe with limitations that I have already outlined. There are many parts of the dot product pass that haven't been properly tested yet, practically none of the code has been optimized yet.

Clava Flow is currently missing some features, such as support for loops without a condition (e.g. a for-loop with a completely empty header), and this limits which analyses can be done with it, indirectly impacting this work.

Finally, there are many other custom instructions that haven't been tackled yet, and each is a candidate for a pass of their own.

References

[1] M. Ajay Kumar, V. Kumar, D. John, and S. Shanker. Implementation and analysis of custom instructions on RISC-v for edge-AI applications. pages 126–129.

[2] Andrew Waterman and Yunsup Lee and David Patterson and Krste Asanović. *The RISC-V Instruction Set Manual, Volume I: User-Level ISA*. RISC-V Foundation, 2019. Editors: Andrew Waterman and Krste Asanović.

[3] Renzo Andri, Tomas Henriksson, and Luca Benini. Extending the risc-v isa for efficient rnn-based 5g radio resource management. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6. IEEE, 2020.

[4] K.K. Balasubramanian, M.D. Salvo, W. Rocchia, S. Decherchi, and M. Crepaldi. Designing RISC-v instruction set extensions for artificial neural networks: An LLVM compiler-driven perspective. 12:55925–55944.

[5] Paul Palomero Bernardo, Patrick Schmid, Oliver Bringmann, Mohammed Iftekhar, Babak Sadiye, Wolfgang Mueller, Andreas Koch, Eyck Jentzsch, Axel Sauer, Ingo Feldner, and Wolfgang Ecker. A scalable RISC-v hardware platform for intelligent sensor processing. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–5. ISSN: 1558-1101.

[6] L. Bertaccini, G. Paulin, M. Cavalcante, T. Fischer, S. Mach, and L. Benini. MiniFloats on RISC-v cores: ISA extensions with mixed-precision short dot products. pages 1–16.

[7] Luca Bertaccini, Gianna Paulin, Tim Fischer, Stefan Mach, and Luca Benini. MiniFloat-NN and ExSdotp: An ISA extension and a modular open hardware unit for low-precision training on RISC-v cores. In *2022 IEEE 29TH SYMPOSIUM ON COMPUTER ARITHMETIC (ARITH 2022)*, pages 1–8. IEEE Computer Soc. ISSN: 1063-6889 Num Pages: 8 Series Title: Proceedings Symposium on Computer Arithmetic Web of Science ID: WOS:000906842700001.

[8] Luca Bertaccini, Gianna Paulin, Tim Fischer, Stefan Mach, and Luca Benini. Minifloat-nn and exsdotp: An isa extension and a modular open hardware unit for low-precision training on risc-v cores. In *2022 IEEE 29th Symposium on Computer Arithmetic (ARITH)*, pages 1–8. IEEE, 2022.

[9] João Bispo and João M.P. Cardoso. Clava: C/c++ source-to-source compilation using lara. *SoftwareX*, 12:100565, 2020.

[10] R. Canal, C. Chenet, A. Arelakis, J.-M. Arnau, J.L. Berral, A. Call, S. Di Carlo, J.J. Costa, D. Gizopoulos, V. Karakostas, F. Lubrano, K. Nikas, Y. Nikolakopoulos, B. Otero, G. Papadimitriou, I. Papaefstathiou, D. Pnevmatikatos, D. Raho, A. Rigo, E. Rodriguez,

A. Savino, A. Scionti, N. Tampouratzis, and A. Torregrosa. VITAMIN-v: Virtual environment and tool-boxing for trustworthy development of RISC-v based cloud services. pages 302–308.

[11] Bingzhen Chen, Xingbo Wang, Yucong Huang, and Zhiyuan Xu. RISCV-FNT: A fast FNT-based RISC-v processor for CNN acceleration. In *2024 IEEE 6TH INTERNATIONAL CONFERENCE ON AI CIRCUITS AND SYSTEMS, AICAS 2024*, pages 292–296. IEEE. ISSN: 2834-9830 Num Pages: 5 Series Title: IEEE International Conference on Artificial Intelligence Circuits and Systems Web of Science ID: WOS:001280469200051.

[12] Gregory K. K. Chen, Phil C. C. Knag, Carlos Tokunaga, and Ram K. K. Krishnamurthy. An eight-core RISC-v processor with compute near last level cache in intel 4 CMOS. 58(4):1117–1128. Num Pages: 12 Place: Piscataway Publisher: Ieee-Inst Electrical Electronics Engineers Inc Web of Science ID: WOS:000903718700001.

[13] 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. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 578–594, Carlsbad, CA, October 2018. USENIX Association.

[14] Y.-F. Chen, Y.-J. Chang, C.-T. Chiu, M.-L. Huang, G.-M. Liang, C.-L. Lee, J.-K. Lee, P.-Y. Hsieh, and W.-C. Lai. Low DRAM memory access and flexible dataflow convolutional neural network accelerator based on RISC-v custom instruction.

[15] Marco Cococcioni, Federico Rossi, Emanuele Ruffaldi, and Sergio Saponara. A lightweight posit processing unit for RISC-v processors in deep neural network applications. 10(4):1898–1908. Num Pages: 11 Place: Piscataway Publisher: Ieee-Inst Electrical Electronics Engineers Inc Web of Science ID: WOS:000905313100018.

[16] CodasipTM. Codasip. https://codasip.com/, 2023. website.

[17] Enfang Cui, Tianzheng Li, and Qian Wei. RISC-v instruction set architecture extensions: A survey. 11:24696–24711. Num Pages: 16 Place: Piscataway Publisher: Ieee-Inst Electrical Electronics Engineers Inc Web of Science ID: WOS:000953835300001.

[18] K. Devarajegowda, E. Kaja, S. Prebeck, and W. Ecker. ISA modeling with trace notation for context free property generation. volume 2021-December, pages 619–624.

[19] Wolfgang Ecker, Peer Adelt, Wolfgang Mueller, Reinhold Heckmann, Milos Krstic, Vladimir Herdt, Rolf Drechsler, Gerhard Angst, Ralf Wimmer, Andreas Mauderer, Rafael Stahl, Karsten Emrich, Daniel Mueller-Gritschneder, Bernd Becker, Philipp Scholl, Eyck Jentzsch, Jan Schlamelcher, Kim Grüttner, Paul Palomero Bernardo, Oliver Bringmann, Mihaela Damian, Julian Oppermann, Andreas Koch, Jörg Bormann, Johannes Partzsch, Christian Mayr, and Wolfgang Kunz. The scale4edge RISC-v ecosystem. In *Proceedings of the 2022 Conference & Exhibition on Design, Automation & Test in Europe*, DATE '22, pages 808–813. European Design and Automation Association.

[20] Free Software Foundation. *GNU Compiler Collection (GCC)*, 2024.

[21] Y. Gao, S. Mosanu, M.N. Sakib, V. Verma, X. Guo, and M. Stan. LiteAIR5: A system-level framework for the design and modeling of AI-extended RISC-v cores. volume 2023-September.

[22] A. Garofalo, G. Ottavi, A. Di Mauro, F. Conti, G. Tagliavini, L. Benini, and D. Rossi. A 1.15 TOPS/w, 16-cores parallel ultra-low power cluster with 2b-to-32b fully flexible bit-precision and vector lockstep execution mode. pages 267–270.

[23] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[24] Michael Gautschi, Pasquale Davide Schiavone, Andreas Traber, Igor Loi, Antonio Pullini, Davide Rossi, Eric Flamand, Frank K. Gürkaynak, and Luca Benini. Near-threshold risc-v core with dsp extensions for scalable iot endpoint devices. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10):2700–2713, 2017.

[25] W. Gong, F. Zhou, and F. Ge. A multi-mode convolution coprocessor based on RISC-v instruction set architecture.

[26] M. Henriques, J. Bispo, and N. Paulino. Using source-to-source to target RISC-v custom extensions: UVE case-study. pages 42–50.

[27] W.H. Kim, H.J. Kim, and T.H. Han. RISC-VR-extension: Advancing efficiency with rented-pipeline for edge DNN processing. pages 800–805.

[28] Y. Kim, S. Venkataramani, S. Sen, and A. Raghunathan. Value similarity extensions for approximate computing in general-purpose processors. volume 2021-February, pages 481–486.

[29] S. Lee, J. Choi, S. Noh, J. Koo, and J. Kung. DBPS: Dynamic block size and precision scaling for efficient DNN training supported by RISC-v ISA extensions. volume 2023-July.

[30] Cangyuan Li, Ying Wang, Huawei Li, and Yinhe Han. APPEND: Rethinking ASIP synthesis in the era of AI. In *2023 60TH ACM/IEEE DESIGN AUTOMATION CONFERENCE, DAC*. IEEE. Num Pages: 6 Web of Science ID: WOS:001073487300183.

[31] Che-Chia Lin, Chao-Lin Lee, Jenq-Kuen Lee, Howard Wang, and Ming-Yu Hung. Accelerate binarized neural networks with processing-in-memory enabled by RISC-v custom instructions. In *50th International Conference on Parallel Processing Workshop*, ICPP Workshops '21, pages 1–8. Association for Computing Machinery.

[32] LLVM Developer Group. *LLVM Compiler Infrastructure*, 2024.

[33] Stefan Mach, Fabian Schuiki, Florian Zaruba, and Luca Benini. Fpnew: An open-source multiformat floating-point unit architecture for energy-proportional transprecision computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 29(4):774–787, 2021.

[34] Simone Machetti, Pasquale Davide Schiavone, Thomas Christoph Müller, Miguel Peón-Quirós, and David Atienza. X-heep: An open-source, configurable and extendible risc-v microcontroller for the exploration of ultra-low-power edge accelerators, 2024.

[35] Simone Manoni, Paul Scheffler, Alfio Di Mauro, Luca Zanatta, Andrea Acquaviva, Luca Benini, and Andrea Bartolini. NARS: Neuromorphic acceleration through register-streaming extensions on RISC-v cores. In *Proceedings of the 21st ACM International Conference on*

Computing Frontiers: Workshops and Special Sessions, CF '24 Companion, pages 79–82. Association for Computing Machinery.

[36] Wei Mao, Kai Li, Quan Cheng, Liuyao Dai, Boyu Li, Xinang Xie, He Li, Longyang Lin, and Hao Yu. A configurable floating-point multiple-precision processing element for hpc and ai converged computing. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 30(2):213–226, 2022.

[37] S.K. Mousavikia, E. Gholizadehazari, M. Mousazadeh, and S.B.O. Yalcin. Instruction set extension of a RiscV based SoC for driver drowsiness detection. 10:58151–58162.

[38] OpenHW Group. CORE-V GCC Project. https://github.com/openhwgroup/corev-gcc, 2024. github repository.

[39] OpenHW Group. CORE-V LLVM Project. https://github.com/openhwgroup/corev-llvm-project, 2024. github repository.

[40] OpenHW Group. CV32E40P 32-bit RISC-V Processor Core. https://github.com/openhwgroup/cv32e40p, 2024. website Accessed: 2024-12-17.

[41] Sandhya P and Priya Chandran. Enhancement in IoT through custom instruction set architectures and TinyML: Review. In *2023 International Conference on Computer, Electronics & Electrical Engineering & their Applications (IC2E3)*, pages 1–6.

[42] Matteo Perotti, Matheus Cavalcante, Alessandro Ottaviano, Jiantao Liu, and Luca Benini. Yun: An open-source, 64-bit RISC-v-based vector processor with multi-precision integer and floating-point support in 65-nm CMOS. 70(10):3732–3736. Num Pages: 5 Place: Piscataway Publisher: Ieee-Inst Electrical Electronics Engineers Inc Web of Science ID: WOS:001079708000003.

[43] R. K. Risikesh, Sharad Sinha, and Nanditha Rao. Variable bit-precision vector extension for RISC-v based processors. In *2021 IEEE 14TH INTERNATIONAL SYMPOSIUM ON EMBEDDED MULTICORE/MANY-CORE SYSTEMS-ON-CHIP (MCSOC 2021)*, pages 114–121. IEEE Computer Soc. Num Pages: 8 Web of Science ID: WOS:000788295800016.

[44] Georg Rutishauser, Joan Mihali, Moritz Scherer, and Luca Benini. xTern: Energy-efficient ternary neural network inference on RISC-v-based edge systems. In *2024 IEEE 35TH INTERNATIONAL CONFERENCE ON APPLICATION-SPECIFIC SYSTEMS, ARCHITECTURES AND PROCESSORS, ASAP 2024*, pages 206–213. IEEE Computer Soc. ISSN: 2160-0511, 2160-052X Num Pages: 8 Series Title: IEEE International Conference on Application-Specific Systems Architectures and Processors Web of Science ID: WOS:001304429500038.

[45] Gonzalo Salinas, Guilherme Sequeira, Alfonso Rodríguez, João Bispo, and Nuno Paulino. Simd acceleration of matrix-vector operations on risc-v for variable precision neural networks. In *2025 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1140–1147. IEEE, 2025.

[46] Haoyang Sang, Wenao Xie, Gwangtae Park, and Hoi-Jun Yoo. An 2.31uj/inference ultra-low power always-on event-driven AI-IoT SoC with switchable nvSRAM compute-in-memory macro. 71(5):2534–2538. Num Pages: 5 Place: Piscataway Publisher: Ieee-Inst Electrical Electronics Engineers Inc Web of Science ID: WOS:001230987700057.

[47] Paul Scheffler, Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. Sparse stream semantic registers: A lightweight ISA extension accelerating general sparse linear algebra. 34(12):3147–3161. Num Pages: 15 Place: Los Alamitos Publisher: IEEE Computer Soc Web of Science ID: WOS:001096168400001.

[48] Pasquale Davide Schiavone, Davide Rossi, Antonio Pullini, Alfio Di Mauro, Francesco Conti, and Luca Benini. Quentin: an Ultra-Low-Power PULPissimo SoC in 22nm FDX. In *IEEE SOI-3D-Subthreshold Microelectronics Technology Unified Conference (S3S)*, pages 1–3, 2018.

[49] Tuomo Sipola, Janne Alatalo, Tero Kokkonen, and Mika Rantonen. Artificial intelligence in the iot era: A review of edge ai hardware and software. In *2022 31st Conference of Open Innovations Association (FRUCT)*, pages 320–331, 2022.

[50] Weixing Su, Linfeng Li, Fang Liu, Maowei He, and Xiaodan Liang. Ai on the edge: a comprehensive review. *Artificial Intelligence Review*, 55(8):6125–6183, 2022.

[51] Giuseppe Tagliavini, Stefan Mach, Davide Rossi, Andrea Marongiu, and Luca Benini. Design and evaluation of smallfloat simd extensions to the risc-v isa. In *2019 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 654–657, 2019.

[52] Farhad Taheri, Siavash Bayat-Sarmadi, and Shahriar Hadayeghparast. RISC-HD: Lightweight RISC-v processor for efficient hyperdimensional computing inference. 9(23):24030–24037. Num Pages: 8 Place: Piscataway Publisher: Ieee-Inst Electrical Electronics Engineers Inc Web of Science ID: WOS:000904931000051.

[53] Blaise Tine, Krishna Praveen Yalamarthy, Fares Elsabbagh, and Kim Hyesoon. Vortex: Extending the RISC-v ISA for GPGPU and 3d-graphics. In *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '21, pages 754–766. Association for Computing Machinery.

[54] Amirhosein Toosi, Andrea G Bottino, Babak Saboury, Eliot Siegel, and Arman Rahmim. A brief history of ai: how to prevent another winter (a critical review). *PET clinics*, 16(4):449–469, 2021.

[55] V. Verma, T. Tracy II, and M.R. Stan. EXTREM-EDGE—EXtensions to RISC-v for energy-efficient ML inference at the EDGE of IoT. 35.

[56] Vaibhav Verma and Mircea R. Stan. AI-PiM-extending the RISC-v processor with processing-in-memory functional units for AI inference at the edge of IoT. 3:898273. Num Pages: 15 Place: Lausanne Publisher: Frontiers Media Sa Web of Science ID: WOS:001116408100001.

[57] S. Wang, J. Zhu, Q. Wang, C. He, and T.T. Ye. Customized instruction on RISC-v for winograd-based convolution acceleration. volume 2021-text, pages 65–68.

[58] Shihang Wang, Xingbo Wang, Zhiyuan Xu, Bingzhen Chen, Chenxi Feng, Qi Wang, and Terry Tao Ye. Optimizing CNN computation using RISC-v custom instruction sets for edge platforms. 73(5):1371–1384. Num Pages: 14 Place: Los Alamitos Publisher: IEEE Computer Soc Web of Science ID: WOS:001200042600017.

[59] X. Wang, C. Feng, B. Chen, Q. Wang, Y. Huang, and T.T. Ye. RV-GEMM: Neural network inference acceleration with near-memory GEMM instructions on RISC-v. pages 302–305.

[60] Xingbo Wang, Chenxi Feng, Xinyu Kang, Qi Wang, Yucong Huang, and Terry Tao Ye. RV-SCNN: A RISC-v processor with customized instruction set for SNN and CNN inference acceleration on edge platforms. pages 1–1. Conference Name: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[61] Zong-Mao Wu, Yu-Chi Lin, and Chih-Wei Liu. AI-ISP accelerator with RISC-v ISA extension for image signal processing. In *2024 INTERNATIONAL VLSI SYMPOSIUM ON TECHNOLOGY, SYSTEMS AND APPLICATIONS, VLSI TSA*. IEEE. Num Pages: 4 Web of Science ID: WOS:001253001400073.

[62] Zhiyuan Xu, Xinyu Kang, Xingbo Wang, Bingzhen Chen, and Terry Tao Ye. FlexBits: A configurable lightweight RISC-v micro-architecture for flexible bit-width execution. In *2024 IEEE 6TH INTERNATIONAL CONFERENCE ON AI CIRCUITS AND SYSTEMS, AICAS 2024*, pages 287–291. IEEE. ISSN: 2834-9830 Num Pages: 5 Series Title: IEEE International Conference on Artificial Intelligence Circuits and Systems Web of Science ID: WOS:001280469200115.

[63] M.-S. Yu, C.-Y. Yuan, T.-L. Chen, and J.-K. Lee. Case study: Optimization methods with TVM hybrid-OP on RISC-v packed SIMD. 12:64193–64211.

[64] Hao Zhang, Dongdong Chen, and Seok-Bum Ko. Efficient multiple-precision floating-point fused multiply-add with mixed-precision support. *IEEE Transactions on Computers*, 68(7):1035–1048, 2019.

[65] H. Zhou, H. Hong, D. Liu, H. Liu, Y. Xia, K. Li, J. Liu, S. Luo, W. Mao, and H. Yu. RISC-v based fully-parallel SRAM computing-in-memory accelerator with high hardware utilization and data reuse rate.