Using Source-to-Source to Target RISC-V Custom Extensions: UVE Case-Study

Miguel Henriques
up201604343@up.pt
University of Porto
Porto, Portugal

João Bispo
jbispo@fe.up.pt
University of Porto / INESC TEC
Porto, Portugal

Nuno Paulino
nuno.m.paulino@inesctec.pt
INESC TEC / University of Porto
Porto, Portugal

ABSTRACT

Hardware specialization is seen as a promising venue for improving computing efficiency, with reconfigurable devices as excellent deployment platforms for application-specific architectures. One approach to hardware specialization is via the popular RISC-V, where Instruction Set Architecture (ISA) extensions for domains such as Edge Artifical Intelligence (AI) are already appearing. However, to use the custom instructions while maintaining a high (e.g., `C/C++`) abstraction level, the assembler and compiler must be modified. Alternatively, inline assembly can be manually introduced by a software developer with expert knowledge of the hardware modifications in the RISC-V core.

In this paper, we consider a RISC-V core with a vectorization and streaming engine to support the Unlimited Vector Extension (UVE), and propose an approach to automatically transform annotated C loops into UVE compatible code, via automatic insertion of inline assembly. We rely on a source-to-source transformation tool, Clava, to perform sophisticated code analysis and transformations via scripts. We use *pragmas* to identify code sections amenable for vectorization and/or streaming, and use Clava to automatically insert inline UVE instructions, avoiding extensive modifications of existing compiler projects.

We produce UVE binaries which are functionally correct, when compared to handwritten versions with inline assembly, and achieve equal and sometimes improved number of executed instructions, for a set of six benchmarks from the Polybench suite. These initial results are evidence towards that this kind of translation is feasible, and we consider that it is possible in future work to target more complex transformations or other ISA extensions, accelerating the adoption of hardware/software co-design flows for generic application cases.

CCS CONCEPTS

• **Software and its engineering** → **Source code generation**; *Development frameworks and environments*; **Compilers**.

KEYWORDS

Instruction Set Extension, Source-to-Source Compilation, Hardware/Software co-design

ACM Reference Format:
Miguel Henriques, João Bispo, and Nuno Paulino. 2024. Using Source-to-Source to Target RISC-V Custom Extensions: UVE Case-Study. In *Rapid Simulation and Performance Evaluation for Design (RAPIDO '24), January 18, 2024, Munich, Germany*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3642921.3642930

1 INTRODUCTION

There is an increased demand for power efficient computing, which has driven a resurgence in hardware/software co-design approaches. However, hardware acceleration in reconfigurable architectures typically requires expert knowledge on hardware design, and several approaches have been used to bridge this gap. For example, the recent investment in High-Level-Synthesis (HLS), which has found most of its use cases in HPC, where PCI-Express boards with Field-Programmable-Gate-Array (FPGA) act as GPU-like accelerators [15, 17, 25]. In contrast, in the edge/embedded domains, devices with a single custom core are more promising solutions. The RISC-V ecosystem has occupied this space due to its paradigm of user definable ISA extensions, a concept with a long history of study [6], experiencing renewed interest.

Implementing ISA extensions presents several challenges: firstly, new hardware must be designed to implement the functionality which we wish to abstract into new assembly instructions; secondly, compiler modifications are needed in order to use the new instructions; finally, for faster design cycles, functional verification of the co-design must be performed efficiently [9]. Extensions can be application specific, or target a domain where common operations can be reused by several software applications. One such case are applications where data streaming and vectorization may provide significant gains. Streaming is a method of performing memory accesses based on pre-configured memory patterns, with benefits such as improved data prefetching, reduced memory access latency, and simplified application of vectorization.

The Unlimited Vector Extension (UVE) [5] is a custom extension to the RISC-V instruction set that implements data streaming and vectorization through a new set of instructions that pre-configure the loop memory access patterns, thus reducing the loop overhead associated with access loading and storing. The extension shows significant performance benefits [5] compared to similar cases, such as the ARM's Scalable Vector Extension (SVE) [26], and is a more compact alternative to RISC-V's "V" Vector Extension (RVV) [22]. In the implementation of UVE [5], the authors manually insert inline assembly in C programs. This process can be error-prone when

RAPIDO '24, January 18, 2024, Munich, Germany

ACM ISBN 979-8-4007-1791-8/24/01
https://doi.org/10.1145/3642921.3642930

developing more complex programs. However, this is a common approach for use of ISA extensions, as we alluded to previously. In [10], an early implementation of a RISC-V supporting the official RVV also employs this method.

In summary, to attain more robust testing and deployment of extensions, compiler support is needed, i.e., forks of established compilers (e.g. Clang, gcc) are required. This can have a high implementation and maintenance cost, as compilers have complex code bases, with a high learning curve. Also, by implementing support for a specific compiler, the implementation becomes vendor locked, and because updating the implementation to a new compiler version might not be trivial, it may also become version locked.

We explore how to reduce the development and verification efforts of extensions early, by using source-to-source compilation to automatically insert the assembly instructions in the source code. For this purpose, we employed the existing Clava compiler [4], which applies transformations based on scripts written by the user. Given this, our contributions in this paper are as follows. We:

(1) present a workflow to support early ISA extension development and testing
(2) propose an approach that transforms standard imperative C loops into a streaming paradigm
(3) automatically transform unmodified C loops into inlined UVE assembly as a use-case
(4) add partial UVE support to the known Spike [23] simulator for functional validation
(5) validate the automatic generation of inline UVE assembly for a set of six benchmarks

The approach allows for a more nimble hardware/software co-design flow for application specific system architectures. Most of the transformations we developed are generic, and can be reused to support other custom ISA extensions and speedup the deployment onto reconfigurable platforms, without the need for forking or re-designing compiler passes or in-depth knowledge of the hardware by the software developer.

This workflow can be used as a first stage for development and testing, prior to committing to the mentioned efforts of compiler modification, postponing this decision when it becomes clear that the use case justifies the effort. At the time of writing, the authors of UVE have adopted this approach [16], via Low Level Virtual Machine (LLVM) [13] integration, although a comparison is out of the scope of this paper.

2 RELATED WORK

We propose fast development and prototyping of custom ISA extensions via source-to-source. As a use case, we focus on the domain of streaming and vectorization. This section summarizes approaches regarding the abstraction of streaming computation models onto a higher-level, and existing `C/C++` source-to-source tools.

- The *OXiGen* [18] tool uses `C` written programs and separate configuration files, which specify optimization opportunities. Although standard code `C` can be processed by *OXiGen*, the user must manually analyze each source file to identify optimization opportunities, and add this information to the separate configuration files.
- The *SARA* [30] compiler uses Spacial [11], a domain-specific language, as a frontend language to explore optimization opportunities before generating code. This approach does address the issue of preventing extra configuration work when working from project to project. However, we are required to write the kernel algorithm in the Spacial language, compromising portability.
- The *FAST* [8] language, based on `C99`, is an imperative language used to describe dataflow models that create high-performance designs. The code is transformed using aspect-oriented programming to produce dataflow code. This work introduces the possibility of performing source-to-source compilation targeting a custom runtime. This workflow addresses all previously pointed issues. We can use the written scripts across different projects without having to configure them and use existing programs as inputs since the conversion tool accepts C/C++ programs. However, the approach is specifically geared for generating hardware/software partitioned designs targeting FPGAs, and is locked to the use of the MaxCompiler [27] to produce the partitioned design.
- The *StreamIt* approach [7] explicitly expresses streaming computation at high-level source code. Several methods represent and operate on abstractions which represent data streams. This allows the programmer to have control of the configuration of the dataflow and explore design variants. It is implemented in Java, meaning it benefits from mature language features. However, as the authors state, there is room for improvement regarding clarity of the syntax.

Approaches for source-to-source transformations have also been studied at some length for multiple purposes, e.g., testing code variations for performance improvements, or applying optimizations for specific domains. The following are significant works on source-to-source compilers:

- The ROSE [21] compiler supports `C/C++` and FORTRAN, and supports generic AST-based transformations by directly extending the compilation framework, which is written in C++. Examples of already provided transformations are auto-parallelization of loops, as well as optimizations such as loop fissioning and fusion.
- Cetus [2] is written in Java and also supports AST-based generic transformations. However, it supports only ANSI C, and needs to be extended in order to implement custom transformations.
- The Artisan approach [29] is focused on source-to-source compilation for heterogeneous platforms. It accepts analysis and transformations implemented as Python scripts, and its main use case is hardware/software partitioning targeting CPU + FPGA systems.

Like Clava, Artisan can be extended using scripts, without the need to change the compiler, although it mainly focuses on transformations targeting HLS-based application specific accelerators. While Cetus and ROSE are more generic and mature source-to-source compilers, both approaches require using their respective code bases as libraries by the developers, thus not allowing users to define new transformation strategies as an input to the compiler. In contrast, Clava allows new transformations to be specified

as interpreted Javascript code, easily allowing for (among other transformations) insertion of inline assembly for any custom ISA extension under development.

Finally, there have been many works on ISA customization [6] that due to the RISC-V are being re-adopted. As our scope is on the flow for testing, we only briefly list some cases of RISC-V extensions: bit-level manipulation instructions are implemented in [12], common cryptography operations in [28], graphics rendering is addressed in [31], in [1] AI inference is targeted, and in [24] another recent extension for streaming computing is introduced.

From our literature review, we find our approach viable for quick use and development of custom extensions. Clava allows for the user to specify their desired transformations as input using only *pragmas* (similarly to *FAST*). This syntax is already part of the `C/C++` language, it is supported by IDEs, and is familiar to software developers. Since compilers ignore unknown *pragmas*, standard compilation is not hindered.

3 UVE ARCHITECTURE AND CODE TRANSFORMATION

As mentioned, we use the UVE developed by Domingos *et al.* [5] as a use case for the viability of fast deployment of custom instructions without manual effort. The extension is a good use-case for our approach, as it is significantly complex to exploit manually.

Besides the difficulty of designing the hardware capable of supporting ISA extensions, realizing their potential also requires compilers capable of identifying opportunities from arbitrary `C/C++` code. When such compilers are unavailable (e.g., yet nonexistent, under-development, or sub-optimal), inline assembly is an alternative, but requires expert knowledge of the extension and hardware.

We developed and tested a sequence of transformations to automatically generate inlined UVE code from arbitrary `C/C++`, using the Clava tool [4]. For context, we briefly present the UVE extension and Clava in this section.

3.1 The Unlimited Vector Extension (UVE)

Figure 1 shows a RISC-V pipeline modified with a vector and streaming engine, and vector registers, which are utilized by the UVE ISA extension. Proper use requires knowledge of the underlying hardware, and of streaming and vectorization paradigms in general.

Vector architectures use long and sequential registers to exhange data with the main memory, and the actual size of the registers is implementation dependent. The UVE extension allows for flexibility regarding register sizes, and populates these registers in conjuction with the streaming paradigm, by prefetching data using stream descriptors and predicates.

The UVE is composed of approximately 80 instructions (450 if including variations), including streaming and vector computation, predicate manipulation, and branch control. The UVE extension allows for 32 configurable vector/streams registers (`u0` to `u31`) and 16 predicates registers (`p0` to `p15`). The computation instructions only interact with UVE registers, so operations over data and stream manipulation must be mapped to them. To configure streams, a descriptor tuple of the offset address, size and stride is used. Tuples can be chained to configure complex multi-dimensional patterns.

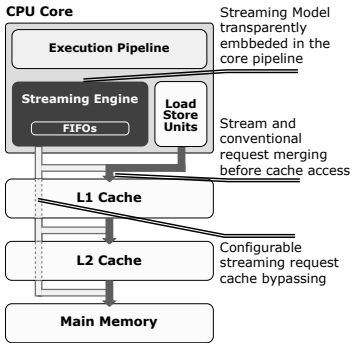

Figure 1: RISC-V with UVE Streaming Vector Engine [5].

In summary, the complexity of vectorization is not particular to this extension, but is an inherent characteristic of any ISA which intends to configure and execute vector based computation, i.e., a Single-Instruction Multiple-Data (SIMD) paradigm. Therefore, adapting existing code to streaming and/or vector implementations is not trivial or universally applicable.

3.2 The Clava Source-to-Source Compiler

Clava [4] is a `C/C++` source-to-source compiler built on top of the LARA engine [19], which enables the specification of complex code analyses and transformations via JavaScript scripts. Clava relies on an unmodified version of Clang's parser to generate a `C/C++` Abstract Syntax Tree (AST) [14] that is very similar to Clang's, but extended to allow for transformations to be applied directly to the AST, which is kept in memory. Valid `C/C++` is only emitted after all transformations are applied, unlike *FAST*, which emits and re-parses the code after each transformation.

We developed Clava scripts to select, analyse, and transform imperative style loops into the streaming paradigm compatible with UVE, while maintaining its operational behaviour. We believe the transformations can be generalized to target development of other extensions, and we are currently working in this direction.

4 PROPOSED APPROACH

Figure 2 summarizes the flow for automatic analysis, transformation, and production of UVE compatible code, starting from unmodified source, requiring only the insertion of *pragmas* to identify which loops should be targeted. In this section we exemplify the `C/C++` code that is consumed by the transformation flow, and respective desired output of a functionally equivalent kernel implemented manually with UVE instructions. We then detail the sequence of transformation steps that automatically achieve the same result.

4.1 Input `C/C++` and Desired Inline UVE Output

Clava receives user input via the *#pragma clava data* directive [4]. This *data* tag uses JSON syntax, which we used to hard-code some aspects such as the UVE stream registers to use (e.g., *u0*, *u1*, etc) as JSON input, allowing for incremental development and debugging. As the implementation matured, external input was no longer required, and only a single boolean value indicating *uve:*

true is required to apply the transformation sequence into UVE code to the tagged loop. Currently, the *#pragma* is only applicable to *for* loops. Also, since pre-processors ignore unknown *pragmas*, annotating the code in this manner does not break compatibility with existing compilers (i.e., the annotated source code could be compiled with a standard RISC-V compiler).

Listing 1 exemplifies this *#pragma* applied to the single loop within the *saxpy* kernel from the Polybench benchmark suite [20], which we will use to illustrate transformations (not all transformations are applicable to this kernel, so we use others when necessary).

```
void saxpy(uint32_t* dest, uint32_t* src,
      uint32_t size, uint32_t A) {
  #pragma clava data uve: true
  for (uint32_t i = 0; i < size; i++)
    dest[i] += src[i] * A;
}
```

Listing 1: saxpy function with the Clava pragma applied

Regarding the output to produce, expressing vectorization and streaming in UVE has the advantage of following a simple pattern. Specifically, prior to the loop body, the source and destination stream registers are configured. This includes specifying the total size and stride, as previously explained. Variants of each compute operation also allow for definition of the vector element widths, or to populate registers with constants. This is followed by the computation section, typically composed of a loop label, a sequence of vector operations, and a final conditional backwards branch. All explicit memory access operations within the loop body are abstracted away, and innermost loops which operate on individual data elements are replaced by the vector instructions.

```
  ss.st.d u0, dest, size, stride  // Config. store stream
  ss.ld.d u1, dest, size, stride  // Config. load stream
  ss.ld.d u2, src,  size, stride  // Config. load stream
  so.v.dp.d u3, A, p0          // Replicate "A" on all lanes of u3
  loop:                        // Loop label
  so.a.mul u4, u2, u3, p0      // Compute u2 * u3, predicated on p0
  so.a.add u0, u1, u4, p0      // Compute u1 + u4, predicated on p0
  so.b.nc u0, loop             // Branch condition
```

Listing 2: UVE which implements the *saxpy* kernel

Listing 2 contains a manually written example pseudo-code we aim to generate when processing the code of Listing 1. We now illustrate how the same result is automatically achieved by applying our sequence of tranformation steps.

4.2 Developed Transformations

The entry point for the transformation of input C/C++ code annotated with the *uve pragma* is shown in Listing 3. It is a JavaScript, which loads the existing libraries for code analysis, along with our *UVEContext* package which holds information about the ongoing sequence of transformations for each run.

```
laraImport("Globals"); laraImport("LoadKernel");
laraImport("UVEContext"); laraImport("LoopFinder");
kernelName = laraArgs.kernelName ?? "memcpy";
compFlags = laraArgs.compFlags ?? "";

// Load all wanted transformations
var transformations = getTransformations();
loadBenchmark(kernelName, compFlags);

// Find all loop candidates -> apply all transformations
var $loops = getUVECandidates();
for (var i = 0; i < $loops.length; i++) {
  var context = new UVEContext(i);
  for (var func of transformations)
    func($loops[i], context);
}

// Store produced code into separate file
Clava.writeCode("output/" + kernelName);
```

Listing 3: Top-level JS file for Clava, to apply the sequence of transformations

The target kernel is passed externally to the script, along with any Clava flags. The *getTransformations* call simply returns a list of all transformations to apply. The *getUVECandidates* call parses the entire code, and returns a list of all innermost loops, *$loops*. A single *pragma* at the outer most level suffices for this depth search of innermost candidates. Finally, a nested loop (lines 14 and 15) applies the transformation sequence to each candidate loop.

These transformations remove some of the syntactic variances seen in the C language. We can iteratively replace a few expressions with equivalent ones that produce code closer to our intended target. Each transformation should unify expressions with similar meaning into a form that will make sequential steps easier to process, as we can expect less expression variance. The transformations we employ progressively transform the code into a format that resembles a classical compiler intermediate representation (such as SSA).

Table 1 summarizes all implemented transformations, their function, and why they are required to achieve the final UVE compliant code. The next paragraphs describe each of the transformations we developed, in the sequence they are applied, and we provide the respective input and output for some of them. Each one is implemented as a separate JS script, which is available as open-source[1].

4.2.1 Compound Assignment Unfolding. C and C++ offer some compound assignment operators to reduce verbosity (e.g., +=), which applies the operator to the left hand side value of the expression and the result of the right hand side expression (itself potentially a composition of several expressions). However, the compound assignment obscures the read operation that is performed. Decomposing these assignments exposes this operation and simplifies the next steps.

```
for (int i = 0; SIZE > i; i++)
  dest[i] += src[i] * A;
```

```
for (int i = 0; SIZE > i; i++)
  dest[i] = dest[i] + src[i] * A;
```

Listing 4: *saxpy* before and after assignment simplification

4.2.2 Comparison Operator Switching. The UVE extension set does not have instructions to express a "greater than" (>) nor the "less than or equal" (<=) comparison operators. This decision usual in RISC architectures, including the RISC-V core specifications, as these operators can easily be synthesized by switching the arguments and using other operators.

```
for (int i = 0; i < SIZE; i++)
    dest[i] = dest[i] + src[i] * A;
```

Listing 5: *saxpy* after comparison operator switching

[1]Miguel Henriques, 2022, "Clava Based Transforms for UVE Code Insertion", Github Repository, https://github.com/MiguelPedrosa/Dissertacao

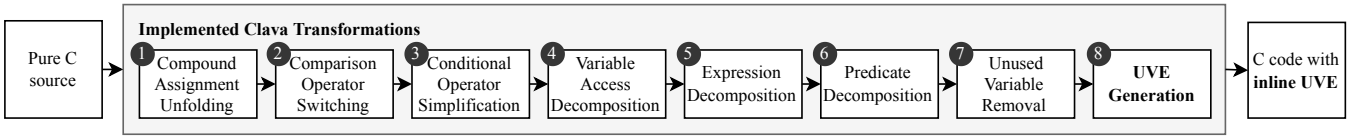

Figure 2: Flow describing the applied transformations. The Clava front-end [4] parses the C/C++ code, and the LARA engine [19] applies the transformations. Clava re-emits valid C/C++ code only after all transformations have been applied.

Table 1: Transformations to convert C/C++ imperative code into a streaming and vector paradigm using UVE inline assembly

Transformation Description	Reason for Implementation
❶ Replaces compound operators with equivalents	Exposes the implicit access and creates a direct mapping to UVE streams/constants
❷ Replaces comparisons by switching operand order	UVE instructions only support *lesser than* comparisons
❸ Replaces the ternary operator with *if-else*	Simplifies a complex expression and removes conditional syntatic variance
❹ Replaces variable accesses with temporary variables	eases mapping of variables to vector UVE registers prior to compute
❺ Simplifies computations with more that 2 operands	UVE supports up to 2 vector input operands; produced instructions become easily translatable
❻ Prefixes expression with *&&* execution predicate	UVE uses predicates instead of branches; *if* statements are removed to map to this behaviour
❼ Removes declared but unused variables	Simplifies output of our code
❽ Replaces each statement with a UVE instruction	inserting inline UVE and removing original kernel

4.2.3 Conditional Operator Simplification. In a ternary operation, `E1?E2:E3`, the target variable is assigned `E2` if `E1` is true, and `E3` otherwise. We transform this operator into an *if-else*, with `E1` as the *if* condition. Depending on the result, either `E2` or `E3` are stored in a newly declared temporary variable of the matching type. After the *if* statement, we write the value of the temporary variable into the original target variable of the ternary operator.

```
path[i][j] =
  path[i][j] < path[i][k] + path[k][j]
  ? path[i][j] : path[i][k] + path[k][j];
```

```
float u0;
if(path[i][j]<path[i][k]+path[k][j])
  u0 = path[i][j];
else
  u0 = path[i][k] + path[k][j];
path[i][j] = u0;
```

Listing 6: *Floyd-Marshall* before and after cond. simplification

4.2.4 Variable Access Decomposition. Every variable access is assigned a new temporary variable in this transformation, similar to single-static assignment. This results in a separation between memory accesses and computations. This includes array accesses and values that remain constant throughout their usage in the loop's body. For memory operations, we declare the temporaries as pointers (e.g., *u0* in Listing 7), which aids in mapping the operations to stream instructions during the next lowering steps. This also creates a quick method to assert if the equivalent stream corresponds to a read or, in this case, a write operation.

```
for (int i = 0; i < SIZE; i++) {
    float *u0 = &dest[i]; // Variable access decomposition
    float u1 = dest[i];   // aids in mapping operands,
    float u2 = src[i];    // results, and intermediate
    float u3 = A;         // values into UVE registers
    *u0 = u1 + u2 * u3;   // during register assignment
}
```

Listing 7: *saxpy* after access decomposition

4.2.5 Expression Decomposition. This is the most complex transformation, since many different types of compound expressions can be composed in C/C++. However, the computation instructions in UVE support up to two input vectors and one output vector. This transformation step decomposes all types of compound expressions into statements with only one output and two inputs, declaring additional variables as required. This includes compound expressions within *if* statements, replacing them with a single predicate. This simplifies the implementation of the next transformation.

```
for (int i = 0; i < SIZE; i++) {
    float *u0 = &dest[i];
    float u1 = dest[i];
    float u2 = src[i];
    float u3 = A;
    float u4;
    u4 = u2 * u3;  // Expression decomposition aids in
    *u0 = u1 + u4; // matching operations to UVE instructions
}
```

Listing 8: *saxpy* after expression decomposition

4.2.6 Predicate Decomposition. To abstract away *if-else* statements and simplify streaming, UVE relies on the use of predicates to perform conditional assignments to target registers. Predicate registers contain boolean values in separate lanes which act as write enables to the individual elements in a target vector register. The number of lanes depends on the configured vector register size (i.e., predicates, as with data, can be evaluated at byte width, word width, *etc*). Values are assigned to predicates using any operation producing booleans, i.e., comparisons. Predicate register *p0* is read-only, and is always *true*. To implement this behaviour, we use C's *short-circuit* operator, &&. When applied to the left-hand-side of an assignment,

the assignment will not occur if the *short-circuit* evaluates to false, e.g., $a \;\&\&\; 0 = 2$; will not assign 2 to a as the logical operation on the left-hand-side is false. Use of the && operator is thus a very close representation of UVE's predication behaviour, making this transformation particularly useful for later lowering to assembly.

```
for (uint32_t i = 0; i < size; i++) {
  float* u0 = &dest[i];
  float  u1 =  dest[i];
  float  u2 =  src[i];
  float  u3 =  A;
  float u4;
  int p0 = 1;
  p0 && (u4 = u2 * u3);
  p0 && (u0 = u1 + u4);
}
```

Listing 9: *saxpy* after predicate decomposition

4.2.7 Unused Variable Removal. This transformation removes extra variables and declarations if they are not used in the body's scope. It filters any unused variables that might be found at this stage, before the final transformation.

4.2.8 UVE Generation. At this stage, almost all instructions in Listing 9 can be directly translated into the inline assembly shown in Listing 10, replacing the original kernel.

```
asm volatile(
 "ss.st.d u0, %[RENCWJ], %[LNWHEG], %[VLXSSS] \n\t"
 "ss.ld.d u1, %[RENCWJ], %[LNWHEG], %[VLXSSS] \n\t"
 "ss.ld.d u2, %[NOBGQU], %[LNWHEG], %[VLXSSS] \n\t"
 "so.v.dp.d u3, %[CEFWVY], p0 \n\t"
 ".uve_loop_0_0_%=:  \n\t"
 "so.a.mul.fp u4, u2, u3, p0 \n\t"
 "so.a.add.fp u0, u1, u4, p0 \n\t"
 "so.b.nc u0, .uve_loop_0_0_%= \n\t"
 :: [RENCWJ] "r" (dest), [LNWHEG] "r" (128),
    [VLXSSS] "r" (1), [NOBGQU] "r" (src), [CEFWVY] "r" (A));
```

Listing 10: Generated code for *saxpy* after all transformations. Equivalent to the manual implementation in Listing 2.

The first four UVE instructions configure the start addresses, stride, and size of the input and output data vectors to access. Next, it defines a branch target (*.uve_loop*), implementing the original loop over the incoming streaming data via the following three vector instructions, namely a floating-point vector multiply *so.a.mul.fp*, a floating-point vector addition *so.a.mv*, and a conditional branch instruction *so.b.nc* back to the defined tag. We can observe how the produced output is equivalent to the manually written code presented as the desired output in Listing 2.

5 RESULTS AND DISCUSSION

We now describe our experimental setup and results, and how the available tools influenced the results we could obtain.

5.1 Experimental Setup

We used a subset of the Polybench [20] benchmark suite, which contains C/C++ implementations of general purpose mathematical algorithms. Some cases include complex memory access patterns, which we do not yet support. Therefore, we selected the following kernels: *saxpy*, *memcpy*, *gemm*, *3mm*, *gemver* and *bicg*.

In [5], the UVE is validated by forking the *gem5* simulator [3], adding the additional emulated hardware components, and support for a subset of the proposed extension. In our validation we did not rely on *gem5*, as it lacked support for the UVE instructions our approach introduced via source-to-source, and adding this support proved too complex. Instead, we added partial UVE support to a fork of the Spike simulator[23], which proved easier to modify[2]. Modifying Spike entailed adding the opcode masks of the UVE instructions (which make use of the RISC-V's free opcodes for custom instructions) and one C++ function per added instruction, to implement the respective functional behaviour (e.g., vector operations, stream accesses, *etc.)*. We added support for all the UVE instructions produced by our process, some of which were not used in the handwritten UVE of our reference approach [5].

The instructions added to the simulator include one-dimensional and multi-dimensional *load* and *store* stream configuration instructions, various static descriptor modifiers, and arithmetic instructions, such as *add*, and the *move* instruction. During this process, we modified the simulator to output the expected incoming/outgoing data-stream values, to ensure correctness, which was verified with simple cases for different vector sizes (outputting of the values was later removed to avoid slower simulation times). Finally, we added one of UVE's branch instructions.

The efforts necessary for these modifications (which are out-of-scope of this paper) illustrate how this component of the eco-system should also be the focus of future work, e.g., by passing new instructions to existing simulators as external definitions. Additionally, since our focus is on verifying the functional behaviour of manually inserted UVE code versus automatically inserted code, we did not address aspects such as accuracy of low-level execution details (e.g. estimation of the application execution time).

5.2 Experimental Evaluation

The objective of our experiments was to generate code that was functionally equivalent to the original C kernel section while exploiting streaming and vectorization in a similar way to the handwritten UVE version. Therefore, we performed two main experiments. Firstly, we compared the instruction counts when executing the kernels on a baseline RISC-V vs UVE (Section 5.3), and secondly, we compared automatically generated UVE to the handwritten version [5], for the cases where this was possible (Section 5.4).

Since multiple transformations are applied, we must validate that each transformation produces valid C/C++ an does not change the functionality of the code. Figure 3 illustrates the flow employed for this process. The unmodified benchmark code is compiled normally, producing a baseline executable. Then, we apply each transformation to the original C source code, and compile the result into a second executable containing UVE instructions. Both programs are then executed in our modified Spike simulator, to verify functional correctness. After functional validation, we employed the same flow to produce the total executed instruction count of the baseline versus the UVE compatible code. Specifically, we used a proxy-kernel that emits this instruction count. This requirement limits the results we can showcase, but it estimates the amount of work expected for each execution.

[2]Miguel Henriques, 2022, "Spike ISA Simulator Fork with UVE Support", Github Repository, https://github.com/MiguelPedrosa/riscv-isa-sim/tree/uve

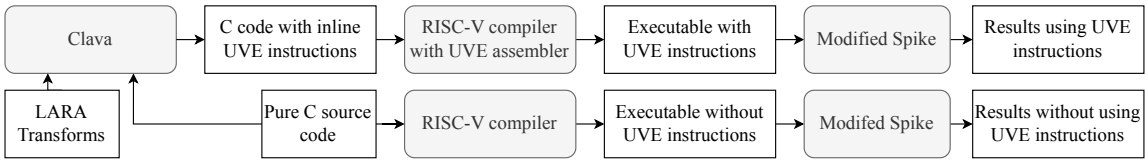

Figure 3: Flow for functional validation of manually inserted UVE code versus Clava driven transformations to annotated code

Table 2: Instruction count for baseline RISC-V vs. Clava-UVE

(a) with *-O0*

Benchmark	Original	Clava-UVE	Reduction*
SAXPY	6001113	5997977	1.00052×
MEMCPY	5557966	5555834	1.00038×
GEMM	349272099	260101057	1.34283×
3MM	326896377	241769873	1.35210×
GEMVER	8773559	8013822	1.09480×
ATAX	5153083	4635125	1.11175×
BICG	4574025	4056061	1.12770×
Geometric Mean	18659273	16375187	1.13948×

(b) with *-O2*

Benchmark	Original	Clava-UVE	Reduction*
SAXPY	5998825	5997977	1.00014×
MEMCPY	5556444	5555834	1.00011×
GEMM	278211475	260101057	1.06963×
3MM	256804682	241769873	1.06219×
GEMVER	7354326	8013822	0.91771×
ATAX	4290001	4635125	0.92554×
BICG	3546594	4056061	0.87440×
Geometric Mean	15983235	16375187	0.97604×

** Executed Instruction Count Reduction*

5.3 Automatic UVE vs. Base RISC-V

Spike is deterministic in its executions. From our experience running these kernels multiple times, the results were the same if the code did not change. As such, we did not perform multiple runs to remove outliers. In our UVE reference paper, the vector width was set to 64 bytes in order to directly compare performance with the ARM SVE, where the vector registers are of this size. Since we wanted to determine the functional correctness of our generated code versus this reference, we chose the same vector register width. Kernels *gemm* and *3mm* were executed with 2D arrays of 128x128 elements, while kernels *saxpy*, *memcpy*, *gemver*, *atax* and *bicg* used 1D arrays of 128 elements. The tested values were *doubles*, meaning they were 8-byte floating-point values.

Table 2a and Table 2b present the results using the *-O0* and *-O2* compilation flags. To isolate only the instruction count of the kernel section, we executed the same binary without the kernel code of the benchmark, to obtain an estimation of the instructions that should be subtracted. In the last column, a number greater that 1 represents a reduction in the number of total executed instructions of the kernel with UVE code.

These results show that we were able to generate code that always presents a lower instruction count compared with the non-UVE code. These are most noticeable in kernels *gemm* and *3mm* that showcase an instruction count reduction of over 1.3×. Even against an optimising compiler running at the *-O2* level of optimisation, we were able to produce similar instruction counts. Although kernels *gemmver*, *atax* and *bicg* present increased instruction counts for the UVE version, inspection of the generated code explains this result. Since we only transform the innermost loops, the assembly section is repeated for each outer loop iteration. The compiler embeds these sections with multiple load operations to read each requested variable. This is only necessary to ensure the correct interface between the assembly section and the surrounding code. If we could compile outer loops, these loads would be executed less and the total instruction count would be further reduced.

Table 3: Instruction counts of handwritten and generated UVE code, and handwritten UVE Speedup vs ARM SVE ([5])

Benchmark	Clava Generated	Handwritten	Speedup [5]
SAXPY	5997977	5997979	≈ 4.5×
MEMCPY	5555834	5555836	≈ 1.2×

A more precise execution time evaluation would require running or simulating a hardware implementation, or using a cycle-accurate simulator. As a reminder, note that the extension itself has already been validated [5] on a cycle-accurate simulator that showed an average 2.4× performance increase compared to ARM's SVE. We leverage these results for the analysis in the next section.

5.4 Automatic UVE vs. Handwritten UVE

Table 3 shows the comparison between the handwritten UVE code and our automatically generated version. Unfortunately, with the exceptions of *saxpy* and *memcpy*, we could not compare with handwritten UVE kernels, as their implementations contain instructions that are not currently supported in our modified Spike simulator. Like-wise, the modified *gem5* simulator does not support all UVE instructions produced by our translation. For the kernels we could execute, we show that we produce code with similar instruction count as the handwritten version. Also, the handwritten version contains redundant load operations, whereas we can detect identifiers with the same value and reuse them, avoiding this overhead.

We were not able to execute most of our generated code in the *gem5* simulator, which is cycle-accurate, however the handwritten versions are supported, and these were compared against an ARM SVE implementation [5]. Manual inspection of the generated code of *saxpy* and *memcpy* indicates that we would achieve similar speedups for these two cases. Note that even if both examples obtained similar instruction counts to plain RISC-V (Table 2b), they likely have performance improvements when compared with *vectorized* code.

6 CONCLUSION

This work presents an automated approach to generate code for the UVE extension that is sufficiently flexible to adapt to changes in the UVE specification. We explored the viability of using source code transformations to apply techniques typically used at lower levels of abstraction. We belive that future further extraction of streaming paradigms from existing imperative style loops is possible.

Using the UVE extension and its implementation [5] as a case-study, we presented a flow for testing ISA extensions based on applying user defined transformations through the Clava tool. We consider that this accelerates adoption of hardware/software co-design, is less error-prone than handwritten inline assembly, frees the software developer from hardware concerns, executes the applications without loss of performance, and avoids the need for extensive modifications in established compilers.

We demonstrated, for set of 6 kernels, that automatically inserted UVE assembly achieves a reduction in the number of executed instructions versus an unmodified RISC-V core. Additionally, we demonstrated that, for the two supported cases, our method of automatic inline assembly insertion matches the instruction count of manual insertion performed in [5], where speedups were attainable versus an ARM core with the SVE extension. These are promising results regarding the remaining cases that could not be compared.

Regarding future work, the entire UVE extension may yet be added to Spike (or similar) to explore additional benchmarks, and we are currently working on generalizing the developed transformations for other extensions. Support for transformation of nested loops into a streaming paradigm is also important. Finally, we note that efforts are also required regarding quick integration of the extensions into existing ISA simulators, for faster design cycles.

ACKNOWLEDGMENTS

This work was supported by national funds through Fundação para a Ciência e a Tecnologia (FCT), under project 2022.06780.PTDC (DOI:10.54499/2022.06780.PTDC).

REFERENCES

[1] Imad Al Assir, Mohamad El Iskandarani, Hadi Rayan Al Sandid, and Mazen A. R. Saghir. 2021. Arrow: A RISC-V Vector Accelerator for Machine Learning Inference. https://doi.org/10.48550/ARXIV.2107.07169

[2] Hansang Bae, Dheya Mustafa, Jae-Woo Lee, Aurangzeb, Hao Lin, Chirag Dave, Rudolf Eigenmann, and Samuel P. Midkiff. 2013. The Cetus Source-to-Source Compiler Infrastructure: Overview and Evaluation. *Intl. Journal of Parallel Programming* 41, 6 (01 Dec 2013), 753–767. https://doi.org/10.1007/s10766-012-0211-z

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Comput. Archit. News* 39, 2 (aug 2011), 1–7. https://doi.org/10.1145/2024716.2024718

[4] João Bispo and João M.P. Cardoso. 2020. Clava: C/C++ source-to-source compilation using LARA. *SoftwareX* 12 (2020). https://doi.org/10.1016/j.softx.2020.100565

[5] Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. 2021. Unlimited Vector Extension with Data Streaming Support. In *ACM/IEEE 48th Annual Intl. Symp. on Computer Architecture (ISCA)*. 209–222. https://doi.org/10.1109/ISCA52012.2021.00025

[6] Carlo Galuzzi and Koen Bertels. 2011. The Instruction-Set Extension Problem: A Survey. *ACM Trans. Reconfigurable Technol. Syst.* 4, 2, Article 18 (may 2011), 28 pages. https://doi.org/10.1145/1968502.1968509

[7] Michael I. Gordon, William Thies, Michal Karczmarek, Jasper Lin, Ali S. Meli, Andrew A. Lamb, Chris Leger, Jeremy Wong, Henry Hoffmann, David Maze, and Saman Amarasinghe. 2002. A Stream Compiler for Communication-Exposed Architectures. In *Proc. of the 10th Intl. Conference on Architectural Support for Programming Languages and Operating Systems*. 291–303. https://doi.org/10.1145/605397.605428

[8] Paul Grigoras, Xinyu Niu, Jose G. F. Coutinho, Wayne Luk, Jacob Bower, and Oliver Pell. 2013. Aspect driven compilation for dataflow designs. In *IEEE 24th Intl. Conference on Application-Specific Systems, Architectures and Processors*. 18–25. https://doi.org/10.1109/ASAP.2013.6567545

[9] Marie-Christine Jakobs, Felix Pauck, Marco Platzner, Heike Wehrheim, and Tobias Wiersema. 2021. Software/Hardware Co-Verification for Custom Instruction Set Processors. *IEEE Access* 9 (2021). https://doi.org/10.1109/ACCESS.2021.3131213

[10] Matthew Johns and Tom J. Kazmierski. 2020. A Minimal RISC-V Vector Processor for Embedded Systems. In *Forum for Specification and Design Languages (FDL)*. 1–4. https://doi.org/10.1109/FDL50818.2020.9232940

[11] David Koeplinger, Matthew Feldman, Raghu Prabhakar, Yaqi Zhang, Stefan Hadjis, Ruben Fiszel, Tian Zhao, Luigi Nardi, Ardavan Pedram, Christos Kozyrakis, and Kunle Olukotun. 2018. Spatial: A Language and Compiler for Application Accelerators. In *Proc. of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. New York, NY, USA, 296–311. https://doi.org/10.1145/3192366.3192379

[12] Bastian Koppelmann, Peer Adelt, Wolfgang Mueller, and Christoph Scheytt. 2019. RISC-V Extensions for Bit Manipulation Instructions. In *29th Intl. Symp. on Power and Timing Modeling, Optimization and Simulation (PATMOS)*. 41–48. https://doi.org/10.1109/PATMOS.2019.8862170

[13] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *Intl. Symp. on Code Generation and Optimization*. 75–86. https://doi.org/10.1109/CGO.2004.1281665

[14] LLVM Project. 2022. *Clang: a C language family frontend for LLVM.* https://clang.llvm.org/

[15] Sparsh Mittal. 2020. A survey of FPGA-based accelerators for convolutional neural networks. *Neural Computing and Applications* 32, 4 (01 Feb 2020), 1109–1139.

[16] Nuno Neves, Joao Mario Domingos, Nuno Roma, Pedro Tomas, and Gabriel Falcao. 2022. Compiling for Vector Extensions with Stream-based Specialization. *IEEE Micro* (2022), 49–58. https://doi.org/10.1109/MM.2022.3173405

[17] Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. 2020. Optimizing OpenCL Code for Performance on FPGA: k-Means Case Study With Integer Data Sets. *IEEE Access* 8 (2020). https://doi.org/10.1109/ACCESS.2020.3017552

[18] Francesco Peverelli, Marco Rabozzi, Emanuele Del Sozzo, and Marco D. Santambrogio. 2018. OXiGen: A tool for automatic acceleration of c functions into dataflow FPGA-based kernels. In *Proc. of the IEEE 32nd Intl. Parallel and Distributed Processing Symp. Workshops, IPDPSW 2018*. 91–98. https://doi.org/10.1109/IPDPSW.2018.00023

[19] Pedro Pinto, Tiago Carvalho, João Bispo, and João M. P. Cardoso. 2017. LARA as a Language-Independent Aspect-Oriented Programming Approach. In *Proc. of the Symp. on Applied Computing*. New York, NY, USA, 1623–1630. https://doi.org/10.1145/3019612.3019749

[20] Pouchet Louis-Noël. 15. PolyBench/C - the Polyhedral Benchmark suite. http://web.cse.ohio-state.edu/$\sim$pouchet.2/software/polybench/

[21] Dan Quinlan and Chunhua Liao. 2011. The ROSE source-to-source compiler infrastructure. In *Cetus users and compiler infrastructure workshop, in conjunction with PACT*, Vol. 2011. Citeseer, 1.

[22] RISC-V Software. 2022. RISC-V Vector Extension 1.0. https://github.com/riscv/riscv-v-spec/releases/tag/v1.0.

[23] RISC-V Software. 2022. Spike RISC-V ISA Simulator. https://github.com/riscv-software-src/riscv-isa-sim.

[24] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. 2021. Stream Semantic Registers: A Lightweight RISC-V ISA Extension Achieving Full Compute Utilization in Single-Issue Cores. *IEEE Trans. Comput.* 70, 2 (2021), 212–227. https://doi.org/10.1109/TC.2020.2987314

[25] Hafsah Shahzad, Ahmed Sanaullah, and Martin Herbordt. 2021. Survey and Future Trends for FPGA Cloud Architectures. In *IEEE High Performance Extreme Computing Conference*. 1–10. https://doi.org/10.1109/HPEC49654.2021.9622807

[26] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, Alastair Reid, Alejandro Rico, and Paul Walker. 2017. The ARM Scalable Vector Extension. *IEEE Micro* 37, 2 (2017), 26–39. https://doi.org/10.1109/MM.2017.35

[27] S. Summers, A. Rose, and P. Sanders. 2017. Using MaxCompiler for the high level synthesis of trigger algorithms. *Journal of Instrumentation* 12, 02 (feb 2017), C02015. https://doi.org/10.1088/1748-0221/12/02/C02015

[28] Etienne Tehrani, Tarik Graba, Abdelmalek Si Merabet, and Jean-Luc Danger. 2020. RISC-V Extension for Lightweight Cryptography. In *23rd Euromicro Conference on Digital System Design*. 222–228. https://doi.org/10.1109/DSD51259.2020.00045

[29] Jessica Vandebon, Jose G. F. Coutinho, Wayne Luk, Eriko Nurvitadhi, and Tim Todman. 2020. Artisan: a Meta-Programming Approach For Codifying Optimisation Strategies. In *IEEE 28th Annual Intl. Symp. on Field-Programmable Custom Computing Machines (FCCM)*. 177–185. https://doi.org/10.1109/FCCM48280.2020.00032

[30] Yaqi Zhang, Nathan Zhang, Tian Zhao, Matt Vilim, Muhammad Shahbaz, and Kunle Olukotun. 2021. SARA: Scaling a Reconfigurable Dataflow Accelerator. In *ACM/IEEE 48th Annual Intl. Symp. on Computer Architecture (ISCA)*. 1041–1054.

https://doi.org/10.1109/ISCA52012.2021.00085
[31] Yuzhi Zhou, Xi Jin, and Tian Xiang. 2020. RISC-V Graphics Rendering Instruction Set Extensions for Embedded AI Chips Implementation. In *Proc. of the 2020 2nd Intl. Conference on Big Data Engineering and Technology*. 85–88. https://doi.org/10.1145/3378904.3378926

Received 20 February 2007; revised 12 March 2009; accepted 5 June 2009