LLVM Backend Support for Data Streaming Extensions

Tiago Pires
Instituto Superior Técnico
Universidade de Lisboa
Lisboa, Portugal
tiago.c.pires@tecnico.ulisboa.pt

***Abstract*—Unlimited Vector Extension (UVE) is a novelty scalable vector extension that incorporates streams. To do so, it implements a decoupled streaming unit that handles all memory processing instructions related to streams, releasing some pressure from the main pipeline. This extension emits much less instructions during loops, increasing the potential time saved. As a new extension that implements target specific instructions, it needs support from a compiler to produce target code. This is achieved by instantiating a new subtarget from RISC-V's LLVM backend and creating and encoding the matching instructions. To give some integration with LLVM Intermediate Representation (IR), some intrinsics were also created to match with the instructions through patterns. To solve potential register streams rewriting and to overcome the SSA form of LLVM IR, pseudo instructions are used. At the end an evaluation was carried out to test this implementation with three benchmarks that highlight the features of the new backend and corresponding intrinsics.**

***Index Terms*—Scalable Vector Processing; Stream Computing; LLVM IR; Compiler Backend**

I. Introduction

One of the main focus when dealing with computers is the time it takes to perform a task. To improve its performance on applications that feature Data-Level Parallelism (DLP), some extension were developed that implement Single Instruction Multiple Data (SIMD) instruction sets. SIMD instruction allow to exploit the DLP and can processes multiple data while only issuing one single instruction. This approach is an improvement over the previous methods but still has some issues: it has a fixed value for the vector width. This constraint forces applications to only being able to perform optimally on machines that feature registers of the same size as the one defined. Two well know extensions that make use of these methods are Intel's SSE and AVX extensions [1], [2] and Arm's NEON extension [3].

The next improvement over this methods was to remove the fixed value constraint for vector width and use a Vector-Length Agnostic (VLA) methodology. Vectors were now without a fixed size, and adaptable to the configuration of the application. Extensions that implement such feature are usually designed by scalable vectorial extensions. The two major extensions that make use of scalable vectors are Arm's Scalable Vector Extension (SVE) [4] and RISC-V "V" extensions [5]. Arm's SVE extension includes 32 scalable vector registers with a vector length comprehended between 128 to 2048 bits, with 128 increments. Supported data-types are byte (8 bits), half word (16 bits), word (32 bits) and doubleword (64 bits). SVE also supports predication mechanisms. On the other end is RISC-V "V" scalable extension can have registers with minimum element size of 8 bits, and all the others that represent a power of 2 from 8. The vector length should be higher or equal to the element size, and can also be iterated in powers of 2. This extension also support predication over the vectors. However these methods can still be improved upon.

II. Related Work

The use of stream abstractions is already present in many works such as Imagine Stream Processor [17], RSVP [18], Q100 [19], Stream-dataflow acceleration [20],VEAL [21], and CoRAM++ [22]. However, these works do not target a general purpose out-of-order core. Regarding the decoupled execution of memory instructions from the main core, Outrider [23] enables the use of multiple simultaneous threads and provides memory latency tolerance. Another work that implements such strategy is Decoupled Supply-Compute (DeSC) [24], by separating the main processing core from a second one or an accelerator that handles memory instructions.

For general purpose computing, Wang et al. [25] proposed an ISA extension for decoupled streams that can enable prefetch stream accesses and remove address computations instructions from the main core to hide some of the latency introduced by memory access operations. This work also implements compiler support using LLVM, that follows the process of first recognizing stream candidates and it's selection and then generating the code for the target. Another work that handles streams with compiler support has also been developed by Neves et al. [26], where the frontend of LLVM is used to identify and modify memory accesses with a dedicated representation. Compiler implementations that range from auto-vectorizations, implementation through directives and accelerator support is presented in works [27]–[31].

III. Compilers

To support the use of these new extensions, it is pretty useful to be able to translate a source code in a particular programming language directly into that extension instructions. That is the purpose of the compiler.

A. Structure of a Compiler

For the most part compiler can be divided into the the structure presented in Figure 1 [10]. This structures usually fall into the category of the frontend or the backend of the compiler. The frontend is responsible for parsing, manipulation and verification of the used programming language while the backend is responsible for taking an IR and generate the specified target code.

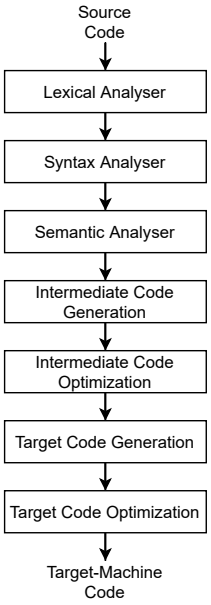

Fig. 1. Usual structure of a compiler.

The lexical analyser is the component responsible to break down the characters received as input from a source file into meaningful sequences called lexemes. Lexemes are then used to create tokens that are associated with them. Each individual token represents a logical piece of source material, such as a variable name or a keyword, and can contain attributes to transmit necessary information. They are usually implement with finite automatas.

Syntax analysers are also known as parsers, and they receive the input from the previous stage and build a tree-like object, often a parse tree or an Abstract Syntax Tree (AST), to give grammatical structure to the token streams with the intention of recovering the structure that was previously represented by the tokens. Two popular strategies to implement a parser are the top-down and bottom-up approaches. The first one begins with the a starting symbol and from there it tries to guess the productions necessary in order to get to the input program. Bottom-up does the opposite, it starts with the input program and tries to reach the starting symbol.

Semantic analyses ensure that a program is consistent with the definition of the language semantics. To do so, it goes through the tree-like structure produced in the previous step and verifies the use of the language symbols, their type and the interactions between them, usually with the help of symbol tables. Some of the properties checked during this step are the use of undeclared variables, scope checking and type checking.

After performing this steps, the structure in transformed into one or more IRs that are between high-level languages and assembly, or some kind of data structure. An advantage of using such representations is to create an independence from the frontend as well as from the backend. This allows to perform optimizations that are language and target independent. One other advantage of using such representations it the simplification of the development of new languages or architectures. To create a new programming language, it is only necessary to implement the features that transform such language into an IR and it will be automatically supported on the backends implement, apart for some specific constraints.

Following the intermediate code generation comes various stages of optimizations and target code generation. this steps can often be interchanged between one another, performing multiple optimizations between the code generation itself.

Some common task that are done during target code generation include instruction selection, instruction scheduling and register allocation.

B. LLVM Infrastructure

The LLVM Infrastructure provides multiple tools and libraries that can be used in the development of a new compiler or part of it. One feature that sets LLVM apart from other compilers is its structure [17]. Instead of adopting a monolithic approach, it is divided into multiple modules and libraries, that implement a specific feature.

For the frontend, the LLVM Infrastructure offers Clang as default. Clang was developed as a replacement for the GCC frontend for easier integration with integrated development environments and better performance. It makes use of hash tables during lexical analysis to save identifiers and builds an AST during syntax analysis to aid with type checking, that will later be transformed into the LLVM IR.

LLVM IR is one of the main components of LLVM. It is an independent language that presents resemblances to the assembly language and is used to separate the frontend from the backend. It is a strongly typed Static Single Assignment (SSA) based representation that uses an infinite set of temporary registers as is designed to be used as a representation in memory for the compiler to manipulate in the form of C++ classes, as a binary bitcode representation that is saved on disk and can be used with just-in-time compilers or as a human readable form. The human readable files come in the ".ll" extension and feature components such as modules, functions, variables, target data layout and the target triple. A module is the top structure that holds all other LLVM IR objects, such as functions and variables. A function is a collection of basic blocks that performs some task, similar to the C language equivalent. Basic blocks are groups of instructions that execute sequentially between their entry point and the terminator instruction. The target data layout and target triple are strings that transmit information about the target data layout and architecture, respectively.

After the generation of the IR, LLVM starts the process of code generation for the specific target. Optimization passes are performed between some of this stages. To generate target code, LLVM starts by creating a Directed Acyclic Graph. Instructions are mapped into DAG nodes and are linked together accordingly to the flow of the program and data dependencies. The next step is to legalize every operation and operator inside of the DAG. The compiler checks if the operations and operators are supported or the target backend, and if they are not they are considered illegal. At this stage the illegal operator and operands need to be converted or transformed into legal ones, either by promoting or expand illegal types, or by performing custom legalization steps. With all the operator and operations legal, the compiler starts the process of instruction selection. Here the generic IR instructions are replaced by specific target supported instructions. To do so, it performs pattern matching by using defined patterns on the target backend. The next step is scheduling the instructions accordingly to the target machine constraints and after that is complete, the DAG is destroyed and the instructions are replaced by another representation, Machine Instruction, that are grouped into a list. Because the target has a limited amount of resources, the next step is to allocate physical registers to the virtual register used up until now. When there are no physical register available, the compiler moves the value of one the the registers into main memory, in a processed called spilling. The default register allocator used by LLVM backend is the Greedy register allocator. It calculates the live ranges for variables (where, during the execution of the program, that variable is used) and makes use of global live range splitting to decide which register to attribute to each variable. The final step is to takes the resulting instruction and emit machine code in binary or assembly format.

IV. Unlimited Vector Extension

UVE takes the concept of scalable vectors and unites it with the implementation of data streams. A stream is a contiguous flow of data, and UVE makes use of them by decoupling the memory accesses to scalable register into a streaming unit, separate from the main pipeline. By doing so, it relieves the pressure of data prefetching, loads and stores into the streaming unit, which leaves more room to execute other instruction. The implementation of this extension also allows to represent streams by using descriptors, to describe the patterns done during memory access. Streams are handled implicitly inside the streaming unit, so the branch instructions are no longer necessary to iterate over scalable vectorial registers as it is done by a specific target instruction. Loop instruction that update the current iteration are also no longer necessary, for the same reason.

A. Definition of streams

To define streams, the new extension makes use of descriptors to describe patterns on memory accesses. Throughout his works [6]–[8], Neves et al. proposes that any address sequence can be described by the affine function in equation 1

$$y(X) = y_{base} + \sum_{k=0}^{dim_y} x_k \times stride_k, \quad \text{with } x_k \in [\alpha_k, \beta_k],\ X = \{x_0 \cdots x_{dim_y}\} \tag{1}$$

where each stream access ($y(x)$) can be described by the sum of a base address (y_{base}) with dim_x pairs of indexing variables (x_k) and stride multiplication factors ($stride_k$). This representation is capable of describing any pattern but can introduce a huge amount of descriptors, so another proposal by Neves et al. is introduced in Figure 2.

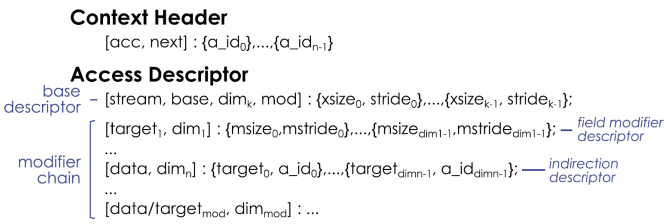

Fig. 2. Descriptor specification proposed by Neves et al. [6].

Using the proposal in Figure 2, the specification is composed of a context header, a base descriptor and a modifier chain. The base descriptor is composed by the stream identification ($stream$), memory base address ($base$), the number of dimensions the descriptor has (dim_y) and the description of each dimension by the pairs dimension size ($xsize$) and stride ($stride$). This base description allows to represent any N-dimension memory description but it would use too many descriptors to do so.

That is why the modifier chain is used. The modifier chain allows to describe non-linear memory patterns, as a way to decrease the number of descriptors necessary to represent such memory accesses. Each modifier is associated with a dimension, and is applied after each iteration of said dimension. They can be divided into two categories: field modifiers and indirect modifiers.

Field modifiers are described by the target of such modification ($target_{mod}$), the dimension of the modifier (dim_{mod}) and by a pair that specifies the modifier size and stride, ($msize$) and ($mstride$), respectively. This allows, as an example, to target the field $xsize_0$ to increase the size of that dimension by one unit every time the modifier iterates.

Indirect modifiers are described by specifying the dimension of the modifier (dim_{mod}) and the pairs target ($target$) and stream descriptor (a_id). This non-linear modifier represents data dependencies between descriptors, and can be used to change a modifier field base on another modifier. As an example, an indirect descriptor can be setup targeting the field $xsize_0$ with an outside stream descriptor a_1. This way, every time the stream associated with descriptor a_1 produces a new value, the stream with this modifier will create an indirect memory access using, the value produces by descriptor a_1 as an offset.

B. UVE Streams

The implementation of streams in UVE is described in the works of Doomingos et al. [9] and follows closely from the description used in Figure 2. Streams are defined as a continuous flow of data and make use of descriptors to describe memory patterns for its configuration and manipulation.

Linear patterns are described in UVE with three parameters: the size of the descriptor, its offset, equivalent to the base from base descriptor, and stride. With such representation, UVE is able to define streams. To create a stream with multiple dimension, it is necessary to create more descriptor, using the same methods, and associate them between each other.

For more complex memory accesses, the extension also makes use of modifiers. Field modifiers, or direct modifiers, are configured by specifying the target of such modification (choice between offset, size and stride), the behaviour it wants to implement (increment or decrement), the amount it will perform such modification and the size of the modifier.

Indirect modifiers are also used to describe indirect memory accesses, originated from another variable. The setup of such modifier is pretty similar to the one used by direct modifiers, except in this case, as it is am indirect modifier, it is representing a data dependency between two variables, so instead of selecting the amount it will modify the targeted dimension this value will come from an outside stream. This value can then be used to add, subtract, increment, decrement or set one of the fields of the targeted dimension.

A summary of this three descriptor used in UVE can be seen in Figure 3.

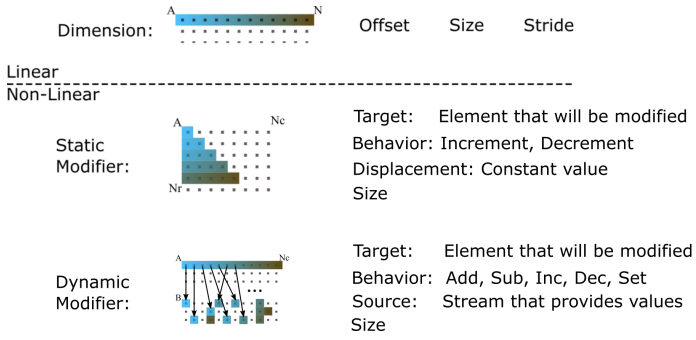

Fig. 3. Descriptors summary containing linear descriptors and dynamic modifiers by Domingos et al. [9].

C. UVE Instruction Set Architecture

UVE is implemented as an extension to the RISC-V, leveraging the fact that it is an open-source architecture.

Its architecture is composed of 32 scalable registers and 16 scalable predicate register. It supports elements widths equal to byte, half-word, word and doubleword with the minimum vectorial register size equal to maximum element size implemented.

Predication mechanisms are also present, allowing to select which lanes of the vector are going to execute by configuring the predicate scalable vector. Although it is composed of 16 scalable predicate registers, only 8 are used to perform memory and arithmetic operations. The predicate register ”p0” is always configured to 1, enabling all the lanes to execute.

UVE also makes use of Control Status Registers (CSR), that can be used to discover the vector maximum length and to configure its working length.

The new extension supports 82 instructions with a total of around 450 variants. To save space on the encoding of each vector, the data-type of each element is not specified there and is instead delegated to the instructions, who must represent such types. To show how the new instructions work, a presentation of some of them is done in detail.

The vertical add instruction instruction is an example of an arithmetic instruction. It takes as input two scalable vectors and performs the addition element by element, saving the result in the output scalable vector. As stated before, the instruction must specify the data-type of the elements inside the vectors. This instruction supports predication of its elements, allowing to execute the addition only for the specified lanes. Another version of this instruction is the horizontal add, where instead of adding two scalable vectors, it performs the summation of one single scalable vector and return the result into a scalar register of the same element size.

Logical instructions can described by the Shift Logical Left (SLL) and Shift Logical Left Scalar (SLLS) instructions. The first takes as input also two scalable vectors and uses the second one to perform individual shifts to the elements of the first vector by their value of the elements in the second, with the result saved to a scalable register. The scalar version SLLS only takes one scalable register as input and replaces the second one by a scalar register. The value on the scalar register is then used to perform a shift on the scalable register elements, that will be the same for all.

UVE also features data transfer instructions, to give support to some programs that might need them, although the main intention of the extension is to perform them implicitly through the use of streams. Load and store instructions take as input the base memory from where the elements are going to be read from and the number of elements that are going to be processed. For the load instructions, it is also required to specify the data-type of the elements inside the scalable vector. Both the load and store instructions come with two different versions, and the reason for that is to maintain the coherent state of the program if the load or store instruction is not able to perform the request (ex. not being able to read all the requested elements). For that the instructions come in the address variant and the size variant, where after each operation the base address or the number of elements to be processed are update accordingly.

To move data to and from scalable register some instructions are offered. There is an instruction that allows to move data from a scalable register into a scalar register, only moving the first element; there is an instruction for the exact opposite, to move data from a scalar register into a scalable vector an element width specification is required and the scalar register moves the element into the scalable vector with possible loss

of data. To replicate this behaviour but for all the elements, UVE also support the duplicate instruction. It is also possible to move data from one scalable vector into another, with the option to transpose all the elements. Still on the same family of instructions, the extension gives support for with conversion instruction, with element widening and element narrowing. To perform such instructions, the elements are copied from the input register into the output register, but because the elements width are unmatched, the register that holds the elements with the smaller width will have to shift them every time elements are copied to or from that register. If elements are being transferred from a vector with smaller element width than the output vector, then the elements are shifted to the left on the input vector every time a transfer of data occurs. If the operations is the other way around, then the elements in the output vector are shifted to the right before a transfer of data occurs.

All the instructions stated above make use of streams, and to configure streams UVE takes the approach stated on section IV-B. There are instructions to configure a simple stream, with size, offset and stride values as input. To configure multidimensional streams, it is only necessary to reuse the same register associated with the stream and append another dimension to it by following the same procedure, or end the description of the stream, with the end instruction instead. To implement direct modifiers, it is necessary to specify if it be appended to a dimension or if it will end the description of the stream, the target of such modifier, the behaviour it will operate with, the size of the modifier and the displacement value. To configure an indirect modifier, the requirements are almost the same, with the replacement of the displacement value by a source stream where the data will come from. This fields for all these instruction were already all detailed in the previous sections.

Lastly, there are instructions to support branches based on the state of a stream, by configuring the instruction to jump if a stream dimension has or has not yet completed.

V. Implementing UVE on LLVM

To give support for the new vectorial extension, LLVM Infrastructure was chosen as the target. The reasons that led to it are the richness and completeness of its assembly-like intermediate representation and its modular backend style, making it easier to implement an extension to the RISC-V architecture without having to deal with any other components. LLVM is also an open source software that has been gaining traction and is already being used to give compiler support for Arm's SVE and Intel RISC-V "V" extensions [12]–[14].

A. LLVM Supporting Extension Requirements

UVE presents some particularities that are, in some cases, different to other extensions. It makes use of scalable registers that are implemented as an extension to RISC-V scalar registers, it uses custom instructions to handle loop control, with loop iterations being done implicitly by the streaming unit, and it also uses streams associated with the scalable registers, with its definition for multidimensions and modifiers being done on the same output register. With that, the following requirements are established:

1) The supporting implementation must be able to represent the scalable vectors and corresponding predication vectors with their own type and associated registers.
2) Remove the standard loop instructions, including the loop iteration control and the branch instruction itself, by replacing them with specific UVE loop control instructions.
3) Allow writing multiple times into the same register to exploit UVE streams mechanic, for configuration of streams and writing to a register associated with a stream, performing an implicit store on the streaming unit.
4) Implement the rest of the extension instructions, following the template described in section IV-C
5) At the end, the supporting implementation must be able to fully translate an LLVM IR file written using all the elements described on the points stated above.

B. Registers Definition

The first step in implementing a new extension to an already defined architecture is to define the new extension as a subtarget of that architecture.

Because UVE uses scalable registers, this scalable vector type needs to be defined on the backend in order to define the extension registers. This task was already done by the SVE extension [15], and the defined Machine Value Type is expressed as

nxv<size><type>

where <size> represents the minimum number of elements inside the vector and <type> represents the the basic types of the elements inside the vector.

The register definition itself is done by instantiating the *RISCVReg* class and then all the registers are grouped into a *RegisterClass*, using the types defined by SVE.

C. Defining new Instructions

To define a new instruction on the backend, it can be used the *RVInst* class, that encodes the parameters with the instruction definitions of RISC-V. The new instructions are subdivided into various categories, such as arithmetic instructions and logic instructions, so they will follow a different encoding from one another. RISC-V specifies the instructions encoding formats in its manual [16], and for UVE the instructions can follow two different formats, the R-type for the generic instructions that perform operations between registers and the B-type, used to encode conditional branch instructions. Inside this encoding types, UVE still subdivides the instructions by their functions, so the creation of complementary classes that specialize on each instruction category eases the definition of new instructions. After the definition of new instruction specialized classes, the instructions are defined by instantiating these classes and referencing as input their encoding, the

number of input and output parameters and the registers they use, the assembly code that should be generated and some configuration flags. Listing 1 show the definition of an intermediary class to later define arithmetic instruction that have one scalable vector as output and two scalable vectors and one scalable predicate register as input. On line 3 the flag that defines the format of the instruction is used, signaling the instruction as having R-type format.

Listing 1. Definition of *UVE_ARITH_V_VVP_f* derived class

```
1  class UVE_ARITH_V_VVP_f<bits<4> funct4,
       bits<1> funct1, UVE_DataType usftype,
2                      RISCVOpcode opcode,
       dag outs, dag ins, string opcodestr,
       string argstr>
3       : RVInst<outs, ins, opcodestr, argstr,
       [], InstFormatR> {
4    bits<3> ps1;
5    bits<5> us2;
6    bits<5> us1;
7    bits<5> ud;
8
9    let Inst{31-28} = funct4;
10   let Inst{27-25} = ps1;
11   let Inst{24-20} = us2;
12   let Inst{19-15} = us1;
13   let Inst{14}    = funct1;
14   let Inst{13-12} = usftype.Value;
15   let Inst{11-7}  = ud;
16   let Opcode = opcode.Value;
17 }
```

The same procedure is used to implement the rest of the instructions with one detail for the branch instruction. As stated before, they follow a different instruction encoding format. Because of that, a different flag is used instead of the *InstFormatR* flag used in listing 1. The new flag indicates that the instruction is of the conditional branch type and this flag is used further on the backline to emit fixups. In LLVM, fixups are used to represent information in instructions which is currently unknown (such as a memory location). During instruction encoding the unknown information is encoded as if the value was equal to 0 and a fixup is emitted which contains information on how to rewrite the value when information is known. If the fixup can not be resolved before the emission of the Execution and Linkable Format (ELF), it is converted into a relocation.

D. Defining new Intrinsics

With the instruction already defined, they now need to be integrated into LLVM IR. There are two possible options to do so:

- **LLVM Intrinsics** : LLVM intrinsics are similar to function calls in C language. They are defined by declaring its inputs, outputs and optionally any flags and are transparent to optimization passes.
- **LLVM Instructions** : LLVM instructions, in the LLVM IR sense, are more similar to assembly instructions and are represented in IR with the result value before the instruction keyword. They are not transparent to optimization passes.

Given the alternatives, the main advantage from using instructions is being able to use the optimization passes for any manipulations or optimizations that may seem necessary. However, when compared to intrinsics, these are much more intricate. Because UVE uses streams, optimizations become more complex and may not even be supported for this kind of representation. Also, LLVM suggests that, if the intended functionality can be represented as a function call, then it should probably start as an intrinsic. Therefore, the method used to implement a representation of UVE instructions in LLVM IR are the intrinsics.

To define new intrinsics, it is necessary to instantiate from the class *Intrinsic*. This class expects as arguments a list of return types, a list of input types and a list or flags. The flags are used to transmit additional information about the intrinsic to the rest of the backend. The process to implement new intrinsics follows the same structure as implementing new instructions: derive a more specialized class from the *Intrinsic* class and then define the intrinsics by instantiating those classes.

Same as with the instruction, the branch intrinsics will have a slight difference. As stated before, LLVM IR defines blocks by having an entry point and a returning instruction at the end. Intrinsics are not instruction and cant be used as a returning instruction. As a result, the intrinsics for UVE branch instructions will be designed to be used in conjunction with regular conditional branches; they will take as input a scalable vector but they will return an integer, that will be used as input to a conditional branch.

LLVM now supports the new instructions and the use of the new intrinsics but it still can't translate from one to another. To do that, it is necessary to define patterns. They allow to make an association between backend instructions and the matching intrinsic. This way, on the instruction selection step, the compiler will match the intrinsics to the instructions accordingly to the defined patterns.

E. Overcoming LLVM IR SSA Form

This solutions has some problems. One of them is the fact that LLVM IR is strongly typed in the SSA form. This implies that when a variable is defined on the IR, it cant be assigned again with any value. This is incompatible with the UVE streaming paradigm, that uses the same register multiple times to configure and implicitly write to streams.

The first implementation to solve such problem was letting the instruction take one additional element as input, the variable associated with the streaming register that is supposed to be written to. Then, constraints on the instruction were defined to force that new input register to be the same as the output. This two registers then become tied together and trigger one pass on the backend that resolves this constraint. However the pass that resolves this constraint did not lead to the expected solution. Instead, it emits a copy instruction before the targeted instruction and replaces the linked register with a temporary one, that is used inside the copy instruction inserted above.

The second and final implementation to solve this issue makes use of pseudo instruction. Pseudo instruction are machine instructions that don't correspond to any specific

assembly instruction, and are used as a placeholder to later be replaced by a pass. One important detail to point out is that this pass runs after register allocation. The idea behind this solution is to create pseudo instructions that are similar in outputs and inputs to the corresponding UVE instruction but they take one additional argument, the stream variable. This pseudo instruction will later be replaced by the pass with the intended UVE instruction, using the extra input register that was allocated as an output register for that instruction. To support this feature in LLVM IR, new intrinsics need to be defined that also take one additional input, and additional patterns to make a connection between the two. In particular, the intrinsics that append new dimensions to a stream and modifiers should always use this implementation, as the previous definition is only useful if the output register can be any available.

F. Register Coalescing

During register allocation, the standard register allocator for LLVM (Greedy) will try to reuse registers if they are no longer used anymore in any future part of the program. This clashes with the streaming paradigm, in particular, with the use of indirect modifiers. A stream can be configured to later be used as an indirect modifier for another stream, and no where else on the program. The register allocator will notice this and reuse that register for another variable, in a process called register coalescing. That register will be written over with another variable but the register in reality is not available. It is being used implicitly by the streaming unit to resolve the indirect modifier that was defined previously.

As the problem is associated with the register allocation process, one possible solution would be to rewrite a new one, or tweak some features on the standard implementation to account for streaming registers. However, both of these methods are complex and there is one simpler way to solve this. Because the problem is associated with the register allocation step, pseudo instruction can be used again, as they are only resolved after that stage. The idea is to create a new pseudo instruction and matching intrinsic that are simply used as placeholders to freeze a variable, and consequently the register it uses. This new intrinsic takes only has one input argument, the variable associated with the stream, and should be placed on the IR where the stream is no longer in use. This stops the register allocator from reusing that register until the place in the program where the freeze intrinsic is used. The corresponding pseudo instruction is then handled in a pass, and is simply removed.

VI. Results

To evaluate the correct operation of the new implementation, a group of 3 benchmarks was used, each one of them with features that make use of different parts of the implementation.

The first benchmark used was the SAXPY kernel as an introduction, to test the definition of one dimension streams and arithmetic instructions. The kernel can be represented using the new intrinsics in LLVM IR by listing 2.

Listing 2. SAXPY kernel described in LLVM IR.

```
1  ...
2  define dso_local void @kernel_saxpy(i64* %x
       , i64* %y, i64 %A, i64 %sizeN) #0 {
3  entry:
4    %streamx = call <vscale x 1 x i64> @llvm.
       riscv.uve.stream.dim.sta.ld.d(i64 %sizeN
       , i64* %x, i64 1)
5    %streamyL = call <vscale x 1 x i64> @llvm
       .riscv.uve.stream.dim.sta.ld.d(i64 %
       sizeN, i64* %y, i64 1)
6    %streamyS = call <vscale x 1 x i64> @llvm
       .riscv.uve.stream.dim.sta.st.d(i64 %
       sizeN, i64* %y, i64 1)
7    %streamA = call <vscale x 1 x i64> @llvm.
       riscv.uve.move.duplicate.d(i64 %A, <
       vscale x 2 x i1> undef)
8    br label %loop
9  loop:
10   %tempStream = call <vscale x 1 x i64>
       @llvm.riscv.uve.mul.s.nxv1i64(<vscale x
       1 x i64> %streamx, <vscale x 1 x i64> %
       streamA, <vscale x 2 x i1> undef)
11   %dummy1 = call <vscale x 1 x i64> @llvm.
       riscv.uve.add.s.save.nxv1i64(<vscale x 1
        x i64> %streamyL, <vscale x 1 x i64> %
       tempStream, <vscale x 1 x i64> %streamyS
       , <vscale x 2 x i1> undef)
12   %loopRes = call i64 @llvm.riscv.uve.
       branch.comp.1.nxv1i64(<vscale x 1 x i64>
        %streamx)
13   %branch1 = trunc i64 %loopRes to i1
14   br i1 %branch1, label %loop, label %
       return_label
15 return_label:
16   ret void
17 }
18 ...
```

Lines 4 through 7 define four stream, three to load data, defined using the ”ld” appended string, and one to store data, defined by the ”st” appended string on line 6. The arithmetic operation occur inside the loop and then a UVE branch intrinsic is called. The compiler processes this code and emits UVE specific assembly, detailed in listing 3.

Listing 3. SAXPY kernel assembly format after compilation.

```
1  ...
2  kernel_saxpy:                           #
       @kernel_saxpy
3  # %bb.0:                                # %entry
4    addi  sp, sp, -16
5    sd  ra, 8(sp)
6    sd  s0, 0(sp)
7    addi  s0, sp, 16
8    addi  a4, zero, 1
9    ss.sta.ld.d u0, a3, a0, a4
10   ss.sta.ld.d u1, a3, a1, a4
11   ss.sta.st.d u2, a3, a1, a4
12   so.v.dp.d u3, a2, p0
13   j .LBB0_1
14 .LBB0_1:                              # %loop
15   so.a.mul.sg u4, u0, u3, p0
16   so.a.add.sg u2, u1, u4, p0
17   so.b.dc.1 u0, .LBB0_1
18   j .LBB0_2
19 .LBB0_2:                        # %return_label
20   ld  s0, 0(sp)
21   ld  ra, 8(sp)
22   addi  sp, sp, 16
23   ret
24 .Lfunc_end0:
25   .size kernel_saxpy, .Lfunc_end0-
       kernel_saxpy
26 ...
```

The result presented in listing 3 demonstrates that the supporting extension is able to correctly translate the intrinsics into UVE instructions and the method to overcome the SSA form was successful, as displayed on line 16 where the

instruction is writing to a register already defined before. The conditional branch in 2 is resolved in conjunction with the UVE branch intrinsic and are replaced by a single UVE branch instruction on line 17 in listing 3.

The second benchmark used was the trisolv kernel. This kernel presents multidimensional memory accesses and presents a memory access pattern that mirrors a triangular matrix. This can be implemented in UVE with the use of direct modifiers. The translation of this kernel into LLVM IR is presented in listing 4. Only the stream configuration part is displayed, as the arithmetic operations all follow the same template. The use of direct modifiers and multidimensional definitions can be observed on lines 10, 11 and 12. On line 11, an intrinsic is issued to append an additional dimension to the stream defined on line 10 and on line 12 a direct modifier is defined to increment the first dimension size 1 unit every time the second dimension is iterated and to end the configuration of the stream.

The resulting code generated by the compiler is presented in listing 5. The displayed code is not the full output file and only shows the generated code equivalent to the intrinsics in listing 4

Listing 4. Trisolv kernel described in LLVM IR.

```
1 ...
2 define dso_local void @kernel_trisolv(i64
    **%L, i64 *%b, i64 *%x, i64 %sizeN) #0 {
3 entry:
4   %streamxiStore = call <vscale x 1 x i64>
      @llvm.riscv.uve.stream.dim.sss.st.d(i64
      %sizeN, i64* %x, i64 1)
5   %streambiLoad = call <vscale x 1 x i64>
      @llvm.riscv.uve.stream.dim.sss.ld.d(i64
      %sizeN, i64* %b, i64 1)
6   %streamxjLoad = call <vscale x 1 x i64>
      @llvm.riscv.uve.stream.dim.sta.ld.d.1p(
      i64 0, i64* %x, i64 1)
7   %dummy2 = call <vscale x 1 x i64> @llvm.
      riscv.uve.stream.mod.end.siz.inc.nxv1i64
      (i64 %sizeN, i64 1, <vscale x 1 x i64> %
      streamxjLoad)
8   %streamliiLoad = call <vscale x 1 x i64>
      @llvm.riscv.uve.stream.dim.sta.ld.d.2p(
      i64 %sizeN, i64** %L, i64 1)
9   %dummy3 = call <vscale x 1 x i64> @llvm.
      riscv.uve.stream.dim.end.nxv1i64(i64 %
      sizeN, i64* null, i64 %sizeN, <vscale x
      1 x i64> %streamliiLoad)
10  %streamlijLoad = call <vscale x 1 x i64>
      @llvm.riscv.uve.stream.dim.sta.ld.d.2p(
      i64 0, i64** %L, i64 1)
11  %dummy4 = call <vscale x 1 x i64> @llvm.
      riscv.uve.stream.dim.app.nxv1i64(i64 %
      sizeN, i64* null, i64 %sizeN, <vscale x
      1 x i64> %streamlijLoad)
12  %dummy5 = call <vscale x 1 x i64> @llvm.
      riscv.uve.stream.mod.end.siz.inc.nxv1i64
      (i64 %sizeN, i64 1, <vscale x 1 x i64> %
      streamlijLoad)
13  %streamxiLoad = call <vscale x 1 x i64>
      @llvm.riscv.uve.stream.dim.sss.ld.d(i64
      %sizeN, i64* %x, i64 1)
14  br label %loop1
15 loop1:
16 ...
```

Listing 5. Trisolv kernel assembly format after compilation.

```
1 ...
2 kernel_trisolv:                       #
    @kernel_trisolv
3 # %bb.0:                              # %entry
4   addi  sp, sp, -16
5   sd  ra, 8(sp)
6   sd  s0, 0(sp)
7   addi  s0, sp, 16
8   addi  a4, zero, 1
9   ss.st.d u0, a3, a2, a4
10  ss.ld.d u1, a3, a1, a4
11  ss.sta.ld.d u2, zero, a2, a4
12  ss.end.mod.siz.inc  u2, a3, a4
13  ss.sta.ld.d u3, a3, a0, a4
14  ss.end  u3, a3, zero, a3
15  ss.sta.ld.d u4, zero, a0, a4
16  ss.app  u4, a3, zero, a3
17  ss.end.mod.siz.inc  u4, a3, a4
18  ss.ld.d u5, a3, a2, a4
19  j .LBB0_1
20 .LBB0_1:                             # %loop1
21 ...
```

In listing 5 on lines 15, 16 and 17 is possible to observe the definition of the multidimensional stream and direct modifier. All three of these instructions use the same register as output to configure the stream.

The last kernel used for evaluation was the SPMV kernel. This features dependencies between variables that will be represented as streams. This implies that it will be necessary the use of indirect modifiers to represent such memory access patterns. This kernel presents an opportunity to demonstrate 2 interesting cases and to explain it the corresponding LLVM IR representation is presented in listing 6

Listing 6. SPMV kernel described in LLVM IR.

```
1  ...
2  entry:
3    ...
4    %streamnnz_ALoad = call <vscale x 1 x i64
       > @llvm.riscv.uve.stream.dim.sss.ld.d(
       i64 %sizeN, i64* %nnz_A, i64 1)
5    ...
6    %streamidx_ALoad = call <vscale x 1 x i64
       > @llvm.riscv.uve.stream.dim.sta.ld.d.2p
       (i64 0, i64** %idx_A, i64 1)
7    %dummy3 = call <vscale x 1 x i64> @llvm.
       riscv.uve.stream.dim.app.nxv1i64(i64 %
       sizeN, i64* null, i64 %sizeN, <vscale x
       1 x i64> %streamidx_ALoad)
8    %dummy4 = call <vscale x 1 x i64> @llvm.
       riscv.uve.stream.ind.end.siz.set.1.
       nxv1i64(<vscale x 1 x i64> %
       streamnnz_ALoad, <vscale x 1 x i64> %
       streamidx_ALoad)
9    %streamxLoad = call <vscale x 1 x i64>
       @llvm.riscv.uve.stream.dim.sta.ld.d.1p(
       i64 0, i64* %x, i64 1)
10   %dummy6 = call <vscale x 1 x i64> @llvm.
       riscv.uve.stream.ind.end.off.add.1.
       nxv1i64(<vscale x 1 x i64> %
       streamidx_ALoad, <vscale x 1 x i64> %
       streamxLoad)
11   %streamyLoad = call <vscale x 1 x i64>
       @llvm.riscv.uve.stream.dim.sss.ld.d(i64
       %sizeN, i64* %y, i64 1)
12   br label %loop1
13 loop1:
14 return_label:
15   call void @llvm.riscv.uve.freeze(<vscale
       x 1 x i64> %streamyStore)
16   call void @llvm.riscv.uve.freeze(<vscale
       x 1 x i64> %streamnnz_ALoad)
17   call void @llvm.riscv.uve.freeze(<vscale
       x 1 x i64> %streamvals_ALoad)
18   call void @llvm.riscv.uve.freeze(<vscale
       x 1 x i64> %streamidx_ALoad)
19   call void @llvm.riscv.uve.freeze(<vscale
       x 1 x i64> %streamxLoad)
20   call void @llvm.riscv.uve.freeze(<vscale
       x 1 x i64> %streamyLoad)
21   ret void
22 }
23 ...
```

In listing 6, it is possible to see the definition of an indirect modifier on line 8. This instruction takes as one of the arguments the variable *%streamidx_ALoad* (stream1), that will determine the size of the this stream (stream2). One of the interesting cases of this kernel is presented on lines 9 and 10. A stream (stream3) is configured with one dimension and an indirect modifier is appended to its configuration. However, stream2 used in the indirect modifier is defined also using an indirect modifier. So what happens is that stream2 has its size set by stream1 and stream3 will have its size indirectly set by stream1 too. The second interesting point is the use of the freeze intrinsics on lines 15 to 20, at the end of the program. As stated before, this is necessary to avoid register coalescing and losing the streams used on indirect modifiers. Two different results from the compiler will be presented next. On listing 7 the result from compiling listing 6. On listing 8 the result from compiling listing 6 but without the freezing intrinsics.

On lines 9 and 11 from listing 7 are two definitions of the indirect streams. The IR code is compiled correctly into assembly and all streaming registers remain unchanged from the beginning till the end of the program.

Listing 7. SPMV kernel assembly with freezing intrinsics.

```
1  ...
2  kernel_smpv:                 # @kernel_smpv
3  # %bb.0:                         # %entry
4    ...
5    ss.ld.d u1, a5, a2, a6
6    ...
7    ss.sta.ld.d u3, zero, a1, a6
8    ss.app  u3, a5, zero, a5
9    ss.end.ind.siz.set.1  u3, u1
10   ss.sta.ld.d u4, zero, a3, a6
11   ss.end.ind.off.add.1  u4, u3
12   ss.ld.d u5, a5, a4, a6
13   j .LBB0_1
14 .LBB0_1:                         # %loop1
15 ...
16 .LBB0_4:                         # %return_label
17   ld  s0, 0(sp)
18   ld  ra, 8(sp)
19   addi  sp, sp, 16
20   ret
21 .Lfunc_end0:
22 ...
```

Listing 8. SPMV kernel assembly without freezing intrinsics.

```
1  ...
2  kernel_smpv:                 # @kernel_smpv
3  # %bb.0:                         # %entry
4    ...
5    ss.ld.d u2, a5, a2, a6
6    ...
7    ss.sta.ld.d u3, zero, a1, a6
8    ss.app  u3, a5, zero, a5
9    ss.end.ind.siz.set.1  u3, u2
10   ss.sta.ld.d u2, zero, a3, a6
11   ss.end.ind.off.add.1  u2, u3
12   ss.ld.d u3, a5, a4, a6
13   j .LBB0_1
14 .LBB0_1:                         # %loop1
15 ...
```

Listing 8 shows the compiled code from the same IR but without the freezing instructions. On lines 10 and 12 the scalable registers ”u2” and ”u3” are written over, respectively. This would cause an issue because, although this registers never appear again throughout the program, they are still being used implicitly by the streaming unit to gather data for the streams defined with indirect accesses. That is why the freezing intrinsics are necessary in listing 6.

VII. Conclusion

The new experimental scalable extension UVE to RISC V architecture presents an exciting alternative that combines the advantages of scalable vectorial architectures with the streaming paradigm. This results in the emission of less instructions, that lead to less clock cycles to process all the instructions. This extension also features memory decoupling from the main processing unit, leaving all the memory operation related to streams for a separate streaming unit. By describing the memory access patterns done by the variables inside loops, the streaming unit takes care of prefetching the necessary data to be ready for processing when the instructions are issued.

To support the UVE extension, it is implemented an extension for the LLVM Infrastructure. As the LLVM backend presents a modular structure, the implementation can be contained to the architecture it targets, RISC V. The backend is populated with new instruction encoded with their respective formats and intrinsics functions are used to represent such

instructions in LLVM IR. Although the SSA format is incompatible with format used by UVE instructions, pseudo instructions are used as placeholders, to be replaced by the correct instructions and formats after register allocation. To avoid unwanted register coalescing by the compiler during register allocation, a freezing intrinsic is used to lock a register up until that point in the code, later being removed entirely.

The supporting implementation is able to represent the new instruction in LLVM IR format through intrinsics and compiles the three tested kernels that are representative of a big part of the implementation features. They are all able to be compiled without any errors and produce assembly code that is according to the UVE standard.

REFERENCES

[1] R.M. Ramanathan and Ron Curry and Srinivas Chennupaty and Robert L. Cross and Shihjong Kuo and Mark J. Buxton, "Extending the World's Most Popular Processor Architecture," White Paper, 2016.

[2] C. Lomont, "Introduction to Intel Advanced Vector Extensions," White Paper, 2011.

[3] ARM, "Introducing NEON™ Development Article," 2009.

[4] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The ARM Scalable Vector Extension," IEEE Micro, vol. 37, no. 2, pp. 26–39, 2017.

[5] AA. Amid, K. Asanovic, A. Baum, A. Bradbury, T. Brewer, C. Celio, A. Chapyzhenka, S. Chiricescu, K. Dockser, and B. Dreyer,, "RISC-V "V" Vector Extension," Tech. Rep., 2019.

[6] N. Neves, "Energy-Efficient Computing: Adaptive Structures and Data Management," Ph.D. dissertation, Instituto Superior Técnico, Universidade de Lisboa, 1 2019.

[7] N. Neves, P. Tomás, and N. Roma, "Adaptive In-Cache Streaming for Efficient Data Management," IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 25, no. 7, pp. 2130–2143, 2017.

[8] N. Neves, N. Roma, and P. Tomás, "Efficient data-stream management for shared-memory manycore systems," in 2015 25th International Conference on Field Programmable Logic and Applications (FPL), 2015, pp. 1–8.

[9] J. Domingos, "Unlimited Vector Extension with data streaming support," Master's thesis, Instituto Superior Técnico, Universidade de Lisboa, 2020.

[10] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, Compilers: Principles, Techniques, & Tools, 2nd ed. Addison Wesley, 8 2006.

[11] J. Freire, D. Koop, E. Santos, C. Scheidegger, C. Silva, and V. Huy, The Architecture of Open Source Applications, 01 2011, pp. 155–171.

[12] G. Hunter and A. Emerson, "Scalable Vectorization in LLVM," ARM, Tech. Rep., 11 2016.

[13] B. S. Center and SiFive, "Code generation for RISC-V V-extension," 10 2020. [Online]. Available: https://lists.llvm.org/pipermail/llvm-dev/2020-October/145850.html

[14] SiFive and B. S. Center, "RISC-V Vector Extension Intrinsic Document," 2021.

[15] G.Hunter, "Supporting ARM's SVE in LLVM,"

[16] A. Waterman and K. Asanović, The RISC-V Instruction Set Manual.CS Division, EECS Department, University of California, Berkeley, 5 2017, vol. I, no. 2.2.

[17] B. Khailany, W. Dally, U. Kapasi, P. Mattson, J. Namkoong, J. Owens, B. Towles, A. Chang, andS. Rixner, "Imagine: media processing with streams,"IEEE Micro, vol. 21, no. 2, pp. 35–46, 2001.

[18] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The re-configurable streaming vector processor (rsvp/spl trade/)," inProceedings. 36th Annual IEEE/ACMInternational Symposium on Microarchitecture, 2003. MICRO-36., 2003, pp. 141–150.

[19] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The architecture anddesign of a database processing unit," inProceedings of the 19th International Conference onArchitectural Support for Programming Languages and Operating Systems, ser. ASPLOS '14.New York, NY, USA: Association for Computing Machinery, 2014, p. 255–26.

[20] T. Nowatzki, V. Gangadhar, N. Ardalani, and K. Sankaralingam, "Stream-dataflow acceleration," in2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA), 2017, pp.416–429.

[21] N. Clark, A. Hormati, and S. Mahlke, "Veal: Virtualized execution accelerator for loops," in2008International Symposium on Computer Architecture, 2008, pp. 389–400.

[22] G. Weisz and J. C. Hoe, "Coram++: Supporting data-structure-specific memory interfaces for fpgacomputing," in2015 25th International Conference on Field Programmable Logic and Applications(FPL), 2015, pp. 1–8.

[23] N. C. Crago and S. J. Patel, "Outrider: Efficient memory latency tolerance with decoupled strands,"in2011 38th Annual International Symposium on Computer Architecture (ISCA), 2011, pp. 117–128.

[24] N. C. Crago and S. J. Patel, "Outrider: Efficient memory latency tolerance with decoupled strands,"in2011 38th Annual International Symposium on Computer Architecture (ISCA), 2011, pp. 117–128.

[25] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose pro-cessors," in2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture(ISCA), 2019, pp. 736–749.

[26] N. Neves, P. Tomás, and N. Roma, "Compiler-assisted data streaming for regular code structures,"IEEE Transactions on Computers, vol. 70, no. 3, pp. 483–494, 2020.

[27] R. Kruppe, J. Oppermann, L. Sommer, and A. Koch, "Extending llvm for lightweight spmd vectoriza-tion: Using simd and vector instructions easily from any language," in2019 IEEE/ACM InternationalSymposium on Code Generation and Optimization (CGO), 2019, pp. 278–279.

[28] V. Porpodas, R. C. O. Rocha, and L. F. W. Góes, "Vw-slp: Auto-vectorization with adaptive vectorwidth," ser. PACT '18. New York, NY, USA: Association for Computing Machinery, 2018.

[29] A. Mishra, A. M. Malik, and B. Chapman, "Extending the llvm/clang framework for openmp metadi-rective support," in2020 IEEE/ACM 6th Workshop on the LLVM Compiler Infrastructure in HPC(LLVM-HPC) and Workshop on Hierarchical Parallelism for Exascale Computing (HiPar), 2020, pp.33–44.

[30] S. Memeti and S. Pllana, "Hstream: A directive-based language extension for heterogeneousstream computing," in2018 IEEE International Conference on Computational Science and Engi-neering (CSE), 2018, pp. 138–145.

[31] A. E. Şuşu, "A vector-length agnostic compiler for the connex-s accelerator with scratchpadmemory,"ACM Trans. Embed. Comput. Syst., vol. 19, no. 6, Oct. 2020.