Compiling for Vector Extensions with Stream-based Specialization

Nuno Neves *†

Joao Mario Domingos *

Nuno Roma *†

Pedro Tomás *†

Gabriel Falcao ‡§

*INESC-ID, †Instituto Superior Técnico, Universidade de Lisboa, Lisboa, Portugal
‡University of Coimbra and §Instituto de Telecomunicações, Coimbra, Portugal

***Abstract*—To overcome the current performance wall, data streaming and data-flow computing paradigms have been gradually making their way into the general-purpose domain. However, the proliferation of such paradigms is often hindered by the lack of compilation support, as their execution model is usually incompatible with the internal static single-assignment (SSA) form used in modern compilers. Accordingly, we propose a new compilation flow that leverages the LLVM infrastructure to automatically extract and encode the memory access pattern and computation data-flow graph with streaming representations. The proposed compilation flow is used to generate code for the recently presented Unlimited Vector Extension (UVE), which tackles the shortcomings of vector-length agnostic SIMD extensions by deploying a data streaming paradigm with implicit memory access and loop control. We show that our proposed tool is capable of detecting, representing, and vectorizing a much wider range of loop patterns than existing solutions while providing significant performance gains.**

■ DATA-LEVEL PARALLELISM explored by Single-Instruction Multiple-Data (SIMD) Instruction Set Architecture (ISA) extensions is currently viewed as a *de facto* solution to accelerate general-purpose High-Performance Computing (HPC) workloads [1]–[3]. However, these extensions traditionally rely on fixed-size registers (e.g., Intel AVX or ARM NEON) posing a non-trivial question regarding the vector length [1], [3], as its optimal size is workload-dependent and leads to portability issues when scaling the vector size.

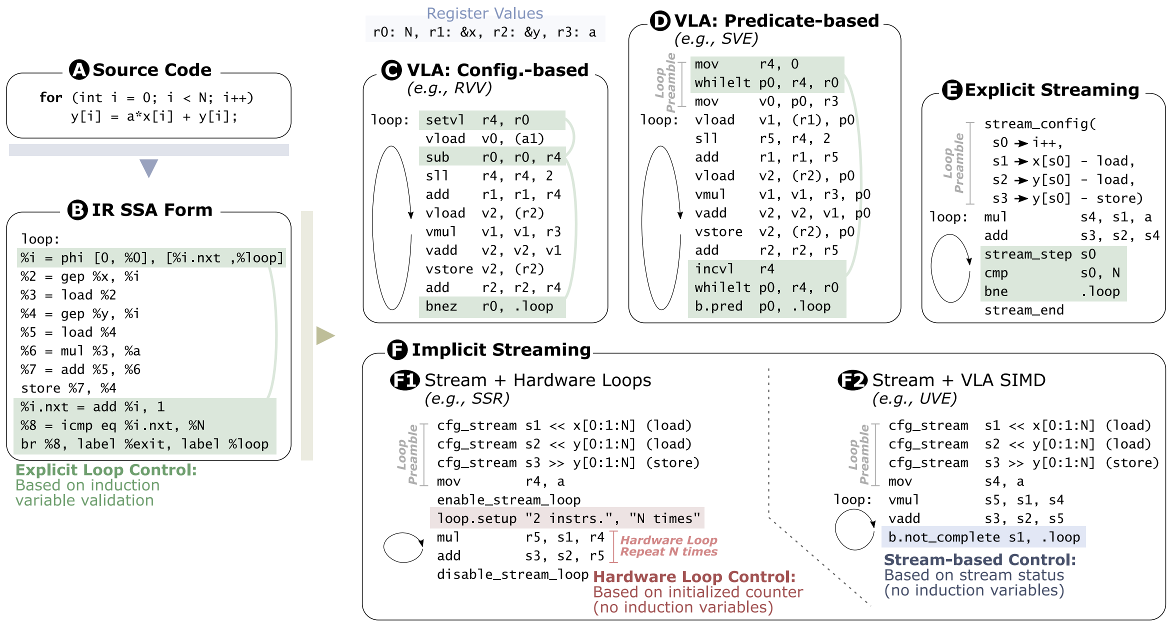
Vector-length agnostic (VLA) SIMD extensions mitigate this issue by allowing runtime configuration of the vector length (as in RISC-V Vector extension (RVV) [2] – see Fig. 1.C) or by relying on loop predication (as in ARM SVE [1] – see Fig. 1.D). As an added benefit, VLA also helps to solve the classic vectorization issue, where the compiler has to inject loop tails when the loop trip count does not evenly divide the vector length. However, the additional instruction overhead required to control the vector length and its execution may actually hinder the application throughput when compared to traditional SIMD extensions [3].

Meanwhile, data-flow and stream-based approaches have been gaining momentum, mostly driven by the emergence of domain-specific architectures. Although not applicable to all classes of applications, they allow programmers to explore several complementary features to increase throughput, such as memory access decoupling and specialization, data prefetching, and efficient parallel computation. These benefits recently drove the adoption of data streaming beyond domain-specific computing into general-purpose processors, as an attempt to push past the limitations of the von Neumann architecture [4], [5]. In particular, most dynamic accesses in known workloads are characterized by affine or indirect data patterns susceptible to data streaming [4]. This allows data transfers to be offloaded to specialized modules to increase the application throughput and improve the functional units utilization and energy efficiency [5].

However, data streaming is not circumscribed to affine and indirect access patterns [6]. As long as the address sequence of the memory access is mathematically deterministic, it is possible to describe it using an hierarchical combination of affine equations. Based on this knowledge, we recently took a step forward from existing general-purpose streaming solutions by proposing the Unlimited Vector Extension (UVE) [7] (see Appendix A). In essence, UVE is a vector extension with an execution model that distinguishes itself by combining both VLA and data streaming paradigms. The latter leverages a formal mathematical model to encode complex, multi-dimensional, strided, and indirect memory accesses in a descriptor-based representation, thus covering a wide range of HPC access patterns. UVE enables automatic streaming of data to the processor vector registers, while linearizing scatter-gather operations, simplifying vectorization. It also effectively transforms each computational loop into a data-flow execution scheme with implicit, indexing-free memory accesses and control-flow, thus simplifying the loop code and reducing the number of executed instructions. As a result, it provides significant throughput gains regarding other VLA extensions, such as ARM SVE [7].

Conversely, the stream-based execution model of UVE (and other competing solutions [5], [8]) brings several new challenges regarding compiler support, particularly due to its implicit data-flow execution model. Non-conventional paradigms tackle similar issues by making use of high-level languages with domain-specific abstractions [9], [10]. However, it is not straightforward to transform regular application code from ubiquitous languages (e.g., C/C++) into a stream-based execution model embedded in a general-purpose ISA.

To tackle this issue, we present a new compilation flow that automatically extracts the memory access patterns and the execution Data-Flow Graph (DFG) of a target application, to generate the corresponding code for a stream-based execution model. We start by identifying that the main problem (and part of the solution itself) lies with the abstraction level provided by the typical static single assignment (SSA) form of the intermediate representation (IR) used in modern compilers (e.g., LLVM IR). In fact, IRs are often fundamentally incompatible with the implicit memory accesses and loop control introduced by the stream-based execution model of UVE (and similar extensions), since loop induction variables are no longer explicitly required. Nevertheless, it is possible to take advantage of the loop canonicalization that is performed in the IR to expose the memory access pattern. With this information, we shown how to detect and encode data streams into a sequence of hierarchical descriptors outside the compiler typical flow and the IR SSA form, as well as how to apply stream-related transformations and vectorization over the computational DFG. Finally, we also show how to generate machine code for streaming extensions, particularly UVE. With the proposed compilation tool, we provide a significant contribution and complement to our



```
r0: N, r1: &x, r2: &y, r3: a
```

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

```
loop:
%i = phi [0, %0], [%i.nxt ,%loop]
%2 = gep %x, %i
%3 = load %2
%4 = gep %y, %i
%5 = load %4
%6 = mul %3, %a
%7 = add %5, %6
store %7, %4
%i.nxt = add %i, 1
%8 = icmp eq %i.nxt, %N
br %8, label %exit, label %loop
```

```
loop:  setvl  r4, r0
       vload  v0, (a1)
       sub    r0, r0, r4
       sll    r4, r4, 2
       add    r1, r1, r4
       vload  v2, (r2)
       vmul   v1, v1, r3
       vadd   v2, v2, v1
       vstore v2, (r2)
       add    r2, r2, r4
       bnez   r0, .loop
```

```
       mov     r4, 0
       whilelt p0, r4, r0
       mov     v0, p0, r3
loop:  vload   v1, (r1), p0
       sll     r5, r4, 2
       add     r1, r1, r5
       vload   v2, (r2), p0
       vmul    v1, v1, r3, p0
       vadd    v2, v2, v1, p0
       vstore  v2, (r2), p0
       add     r2, r2, r5
       incvl   r4
       whilelt p0, r4, r0
       b.pred  p0, .loop
```

```
       stream_config(
        s0 ➔ i++,
        s1 ➔ x[s0] - load,
        s2 ➔ y[s0] - load,
        s3 ➔ y[s0] - store)
loop:  mul         s4, s1, a
       add         s3, s2, s4
       stream_step s0
       cmp         s0, N
       bne         .loop
       stream_end
```

```
cfg_stream s1 << x[0:1:N] (load)
cfg_stream s2 << y[0:1:N] (load)
cfg_stream s3 >> y[0:1:N] (store)
mov        r4, a
enable_stream_loop
loop.setup "2 instrs.", "N times"
mul        r5, s1, r4
add        s3, s2, r5
disable_stream_loop
```

```
       cfg_stream  s1 << x[0:1:N] (load)
       cfg_stream  s2 << y[0:1:N] (load)
       cfg_stream  s3 >> y[0:1:N] (store)
       mov         s4, a
loop:  vmul        s5, s1, s4
       vadd        s3, s2, s5
       b.not_complete s1, .loop
```

Figure 1. Representation of the saxpy kernel in: (A) C code and (B) LLVM IR format; in VLA SIMD extensions pseudo-assembly based on (C) vector length configuration and (D) loop predication; and pseudo-assembly stream-based extensions with (E) explicit and (F) implicit stream iteration (F1 – using hardware loops, F2 – with stream vectors). Note the presence of the induction variable exposed in (B) in examples (C), (D), and (E) and its implicit elimination in (F).

previous work [7], which was solely focused on the design and implementation of the scalable streaming vector ISA extension in out-of-order pipelines.

Stream Specializations and UVE

Early attempts to bring stream-based paradigms into modern processors experienced some success by specializing their memory accessto facilitate the prefetching of repeated access patterns [4], allowing the execution pipeline to read data directly from dedicated stream buffers (see Fig. 1.E). However, it was later realized that streaming specializations could be extended all the way into the processor pipeline, by configuring all memory access patterns in the loop preamble and automatically streaming data directly to the processor's registers [5], [7]. This way, memory accesses become totally implicit to the processor, allowing the application code to be devoid of instructions for indexing, load/store, and induction variable-based loop control (see Fig. 1.F). In turn, it results in a two-fold acceleration by speeding up data acquisition and decreasing loop instructions. This approach is the core of solutions such as the SSR [5] (see Fig. 1.F1) and the UVE [7] ISA extensions.

However, while SSR [5] was conceived as a dedicated streaming extension, UVE [7] was designed as a VLA extension with transparent data streaming (see Appendix A for more details). In fact, the main goal of UVE is to address the limitations of VLA SIMD, while offering a new level of acceleration. Hence, besides its implicit memory access and loop control, its stream-based paradigm combines some of the scalable vectorization characteristics of VLA, such as vector-length predication and auto-scaling (to automatically disable vector elements that fall out of loop bounds), with enhanced vectorization capabilities (e.g., handling reductions with uneven element counts without loop tails). This was achieved through a formal mathematical model to represent complex multi-dimensional, strided, and indirect memory access patterns through exact descriptor representations encoded in the loop preamble. These representations also allow linearizing memory access patterns, simplifying vectorization. The model follows the typical structure of nested *for* loops and is represented by an n-dimensional affine function:

$$
\begin{aligned}
&y(X) = y_{base} + \sum_{k=0}^{dim_y} x_k \times S_k \\
&\textbf{with} \quad X = \{x_0, \ldots, x_{dim_y}\} \\
&\textbf{and} \quad x_k \in [O_k,\ E_k + O_k]
\end{aligned} \qquad (1)
$$

Hence, each stream access $y(X)$ is described as the sum of the base address of an n-dimensional variable (y_{base}) with dim_y pairs of indexing variables (x_k) and stride multiplication factors (S_k), where each k value corresponds to a dimension of the pattern (usually, bound to a different loop in the code). Each indexing variable x_k is represented by an integer range, varying between O_k and $E_k{+}O_k$, where E_k is the number of data elements in dimension k and O_k represents an indexing offset. The indexing variable x_0, corresponding to the first dimension of the variable, has an offset (O_0) equal to 0 and is associated with the variable's base address (y_{base}).

UVE leverages this model with a set of descriptors that encode each affine function's parameters (depicted later in Fig.3). This scheme allows achieving higher representation complexity through the combination of multiple descriptors and specific modifiers. To model inter-loop induction-variable dependencies (e.g., when the loop conditions are generated by the iteration of an outer loop – *imperfect loops*), a *static modifier* allows assigning the result of an affine function to the limits of the indexing variables of another function. To model indirect memory accesses, an *indirect modifier* is used to assign the data obtained by the sequence of addresses generated by one an affine function to the offset, stride, or indexing variable limits of another function.

However, the viability and proliferation of stream specializations, such as those proposed in UVE, can only be assured through an effective support on standard compilers. This requires detecting the memory access patterns in a loop that are amenable to streaming, as well as their representation using the target streaming model, and extending the compiler's capabilities to support implicit memory accesses and loop control.

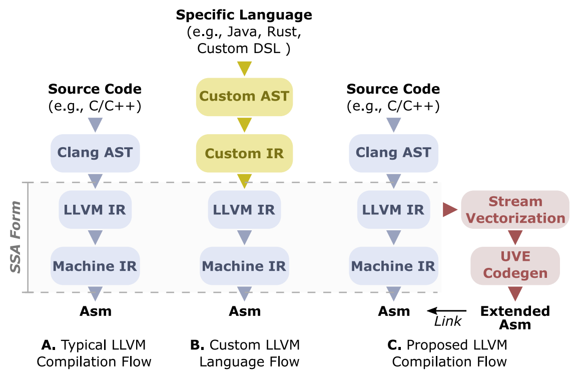

Specific Language
(e.g., Java, Rust, Custom DSL)
Source Code (e.g., C/C++)
Custom AST
Source Code (e.g., C/C++)
Clang AST
Custom IR
Clang AST
SSA Form
LLVM IR
LLVM IR
LLVM IR
Stream Vectorization
Machine IR
Machine IR
Machine IR
UVE Codegen
Asm
Asm
Asm
Link
Extended Asm
A. Typical LLVM Compilation Flow
B. Custom LLVM Language Flow
C. Proposed LLVM Compilation Flow

Figure 2. Typical compilation flows for (A) the base LLVM infrastructure, (B) LLVM-based custom compilers, and (C) the proposed compilation flow.

Compiler Support for Data Streaming

The typical structure of a modern compiler is composed of a sequence of phases that progressively lower the level of abstraction from the programming language to the machine code (see Fig. 2.A). The LLVM compiler performs these operations by first parsing the source code into a structural syntax representation (Abstract Syntax Tree (AST)) and then translating it to its intermediate representation (IR). The IR is then optimized and, finally, lowered to a machine-specific IR and compiled to machine code.

Hence, the LLVM IR can be regarded as a low-level representation of the application code that follows the SSA form to canonicalize the loop structure and the iteration of induction variables (see Fig. 1.B). As such, it represents a powerful baseline for the implementation of several analyses and transformations for optimizing the application code. However, it is known to struggle with the modeling and optimization of high-level abstractions used by domain-specific constructs or non-conventional computing paradigms (such as data streaming) [11]. As a result, these paradigms often either rely on Domain Specific Languages (DSLs) or develop custom IRs to interface with the compiler and implement domain-specific optimizations [9], [10] (as illustrated in Fig. 2.B).

Despite their success in some cases [8], [11], DSLs and custom IRs work at a higher level of abstraction and eventually map to LLVM IR. However, the implicit memory access and loop control characteristics of stream-based extensions are present on a lower level of abstraction than the IR and, as such, are themselves fundamentally incompatible with its operation. In fact, explicitly removing load/store instructions and induction variables from the loop would not only invalidate its iteration (since a trip count could not be kept) but would also make the computation code invariant (since the source input/output values would be eliminated), effectively turning the loop into dead code, resulting in its later elimination during IR optimization.

Nonetheless, it is still possible to take advantage of the IR loop canonical form to detect memory access patterns and encode data streams, as demonstrated by the SSR supporting toolchain [5]. Although SSR only supports constant-strided patterns, the underlying mathematical model of UVE [7] hints that it is possible to take a step further. In fact, since our model is based on the same affine relations that are used to canonicalize loops, it can not only be used to encode much more complex multi-dimensional, induction-dependent, and indirect memory access patterns, but also used as a reference to detect the combinations of induction variables that match those patterns.

Hence, our proposed compilation flow relies on a dedicated pass that uses the model from Equation 1 to extract all the loop information from the IR. It operates outside the typical compilation pipeline (as depicted in Fig. 2.C) to perform stream encoding, stream-based code transformations, and apply stream vectorization without being limited by the SSA form. The code is then compiled to UVE and linked to the remaining compiled code.

Compilation Flow for Stream-based Vector Extensions

Our proposed compilation flow (see Fig. 2.C) is fully implemented as an LLVM IR analysis and transformation pass. It works by analysing an application kernel represented in LLVM IR to encode each loop's data streams and obtain the corresponding computational DFG. The obtained information is then used to perform stream vectorization and generate UVE machine code. To fully take advantage of the LLVM IR, we let the compiler run all the passes that fully optimize the code, but without running any form of vectorization or loop unrolling (by using flags `-O3 -fno-unroll-loops -fno-vectorize`). This way, we ensure that all loops and induction variables are in the canonical form and that loop-invariant code motion is applied.

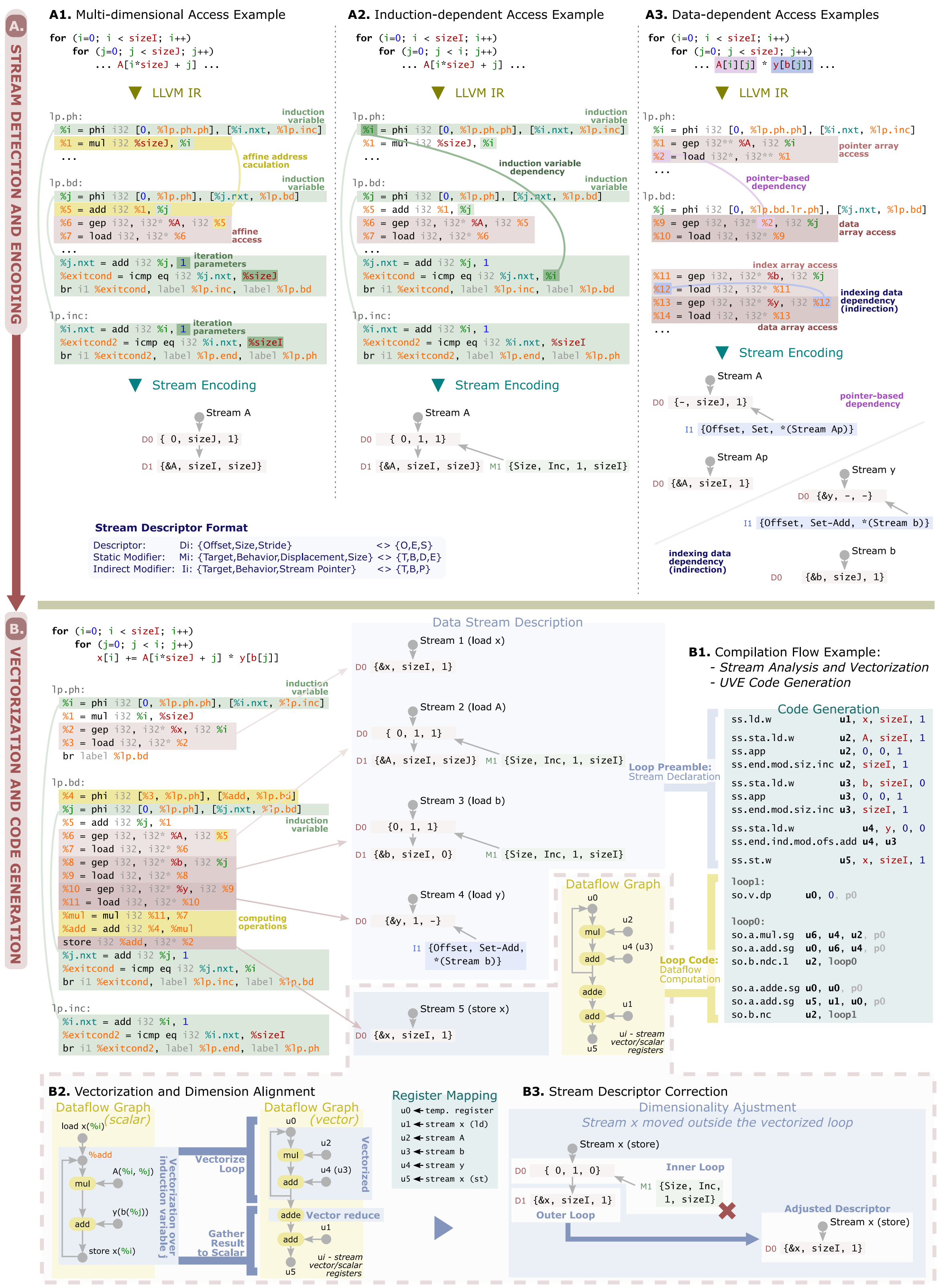
Memory Access Pattern Detection

The proposed flow starts by analysing the optimized IR of an application kernel to detect its loop hierarchy and control-flow, by making use of the built-in LLVM IR functions.

Data streams are detected and encoded to descriptors with an initial search for load/store instructions in the loop structure, while marking them as stream candidates. Then, for each candidate, we perform a Depth-First Search (DFS) trace to find the corresponding *GetElementPtr* (GEP) instruction and its corresponding offset and induction variable. With this reference pair, two dependency analyses are performed to: *i)* identify the relation between the induction variable and the loop control, obtaining its iteration parameters according to the affine model of Equation 1; *ii)* relate the offset with other induction variables (higher data pattern dimensions) or other load/store instructions (access indirection).

Each stream candidate is analysed to validate if stream vectorization is possible. In particular, if aliasing can occur between two (or more) memory accesses, the tool discards them as stream candidates to avoid intersections between streams. Similarly, the presence of loop-carried dependencies (e.g., read-after-write memory accesses) will also cause stream candidates to be discarded, since they are not susceptible to vectorization. Although they could be translated to scalar streams, new stream coherence mechanisms currently not supported by stream-based extensions would be required [5], [7].

When considering the trace performed over the induction variables, several situations may occur that immediately dictate the type of memory access pattern present in the loop. The most common scenario is the detection of a purely affine iteration, where the variable's initial value and limit are statically defined outside the loop, and the step is found by tracing the parent *phi* and *add* instructions. This results in a straightforward *n*-dimensional descriptor encoding (as illustrated in Fig. 3.A1). However, in the presence of induction-



A1. Multi-dimensional Access Example

```
for (i=0; i < sizeI; i++)
    for (j=0; j < sizeJ; j++)
        ... A[i*sizeJ + j] ...
```

```
lp.ph:
  %i = phi i32 [0, %lp.ph.ph], [%i.nxt, %lp.inc]
  %1 = mul i32 %sizeJ, %i
  ...

lp.bd:
  %j = phi i32 [0, %lp.ph], [%j.nxt, %lp.bd]
  %5 = add i32 %1, %j
  %6 = gep i32, i32* %A, i32 %5
  %7 = load i32, i32* %6
  ...
  %j.nxt = add i32 %j, 1
  %exitcond = icmp eq i32 %j.nxt, %sizeJ
  br i1 %exitcond, label %lp.inc, label %lp.bd

lp.inc:
  %i.nxt = add i32 %i, 1
  %exitcond2 = icmp eq i32 %i.nxt, %sizeI
  br i1 %exitcond2, label %lp.end, label %lp.ph
```

A2. Induction-dependent Access Example

```
for (i=0; i < sizeI; i++)
    for (j=0; j < i; j++)
        ... A[i*sizeJ + j] ...
```

```
lp.ph:
  %i = phi i32 [0, %lp.ph.ph], [%i.nxt, %lp.inc]
  %1 = mul i32 %sizeJ, %i
  ...

lp.bd:
  %j = phi i32 [0, %lp.ph], [%j.nxt, %lp.bd]
  %5 = add i32 %1, %j
  %6 = gep i32, i32* %A, i32 %5
  %7 = load i32, i32* %6
  ...
  %j.nxt = add i32 %j, 1
  %exitcond = icmp eq i32 %j.nxt, %i
  br i1 %exitcond, label %lp.inc, label %lp.bd

lp.inc:
  %i.nxt = add i32 %i, 1
  %exitcond2 = icmp eq i32 %i.nxt, %sizeI
  br i1 %exitcond2, label %lp.end, label %lp.ph
```

A3. Data-dependent Access Examples

```
for (i=0; i < sizeI; i++)
    for (j=0; j < sizeJ; j++)
        ... A[i][j] * y[b[j]] ...
```

```
lp.ph:
  %i = phi i32 [0, %lp.ph.ph], [%i.nxt, %lp.inc]
  %1 = gep i32** %A, i32 %i
  %2 = load i32*, i32** %1
  ...

lp.bd:
  %j = phi i32 [0, %lp.bd.lr.ph], [%j.nxt, %lp.bd]
  %9 = gep i32, i32* %2, i32 %j
  %10 = load i32, i32* %9

  %11 = gep i32, i32* %b, i32 %j
  %12 = load i32, i32* %11
  %13 = gep i32, i32* %y, i32 %12
  %14 = load i32, i32* %13
  ...
```

Stream Descriptor Format

```
Descriptor:          Di: {Offset,Size,Stride}                     <> {O,E,S}
Static Modifier:     Mi: {Target,Behavior,Displacement,Size}      <> {T,B,D,E}
Indirect Modifier:   Ii: {Target,Behavior,Stream Pointer}         <> {T,B,P}
```

```
for (i=0; i < sizeI; i++)
    for (j=0; j < i; j++)
        x[i] += A[i*sizeJ + j] * y[b[j]]
```

```
lp.ph:
  %i = phi i32 [0, %lp.ph.ph], [%i.nxt, %lp.inc]
  %1 = mul i32 %i, %sizeJ
  %2 = gep i32, i32* %x, i32 %i
  %3 = load i32, i32* %2
  br label %lp.bd

lp.bd:
  %4 = phi i32 [%3, %lp.ph], [%add, %lp.bd]
  %j = phi i32 [0, %lp.ph], [%j.nxt, %lp.bd]
  %5 = add i32 %j, %1
  %6 = gep i32, i32* %A, i32 %5
  %7 = load i32, i32* %6
  %8 = gep i32, i32* %b, i32 %j
  %9 = load i32, i32* %8
  %10 = gep i32, i32* %y, i32 %9
  %11 = load i32, i32* %10
  %mul = mul i32 %11, %7
  %add = add i32 %4, %mul
  store i32 %add, i32* %2
  %j.nxt = add i32 %j, 1
  %exitcond = icmp eq i32 %j.nxt, %i
  br i1 %exitcond, label %lp.inc, label %lp.bd

lp.inc:
  %i.nxt = add i32 %i, 1
  %exitcond2 = icmp eq i32 %i.nxt, %sizeI
  br i1 %exitcond2, label %lp.end, label %lp.ph
```

B1. Compilation Flow Example:
- *Stream Analysis and Vectorization*
- *UVE Code Generation*

Code Generation

```
ss.ld.w                 u1, x, sizeI, 1

ss.sta.ld.w             u2, A, sizeI, 1
ss.app                  u2, 0, 0, 1
ss.end.mod.siz.inc u2, sizeI, 1

ss.sta.ld.w             u3, b, sizeI, 0
ss.app                  u3, 0, 0, 1
ss.end.mod.siz.inc u3, sizeI, 1

ss.sta.ld.w                 u4, y, 0, 0
ss.end.ind.mod.ofs.add u4, u3

ss.st.w                 u5, x, sizeI, 1

loop1:
so.v.dp      u0, 0, p0

loop0:
so.a.mul.sg  u6, u4, u2, p0
so.a.add.sg  u0, u6, u4, p0
so.b.ndc.1   u2, loop0

so.a.adde.sg u0, u0, p0
so.a.add.sg  u5, u1, u0, p0
so.b.nc      u2, loop1
```

Register Mapping

```
u0 ← temp. register
u1 ← stream x (ld)
u2 ← stream A
u3 ← stream b
u4 ← stream y
u5 ← stream x (st)
```

Figure 3. Illustration of each step of the LLVM IR pass that implements our proposed compilation flow, including depictions of (A) the memory access pattern detection (multi-dimensional, induction-, and data-dependent) and stream encoding schemes; and (B) the vectorization and code generation procedures.

variable dependencies, the trace will detect a dependency between the *phi* node of an outer loop and the *phi* node of the induction variable. This indicates that either the initial or limit values of the induction variable are generated by an induction variable of an outer loop (see Fig. 3.A2). In this case, the descriptor encoding will be performed with static modifiers to represent the evolution of the induction variable interval over the loop iteration.

Finally, a data-dependency between an induction variable and another load instruction translates into an indirection between another stream and the induction variable's limits (see Fig. 3.A3). In this case, it is necessary to encode the output data as input of another stream with the aid of an indirect descriptor modifier. Similarly, dependencies can also occur between the offset and another load instruction. Such a scenario occurs when the original application code defines multi-dimensional accesses with pointer table structures (e.g., $A[i][j]$) instead of affine relations between indexing variables (e.g., $A[i \times sizeJ + j]$). In this case, the stream is also encoded with indirect modifiers linking the output of the outer loop stream to the offset of the inner loop stream (see Fig. 3.A3).

Stream-based Vectorization

As previously discussed, our mathematical model allows the stream descriptors generated by the initial analysis step to effectively linearize multi-dimensional and indirect memory accesses. As such, vectorization is functionally achieved by allowing stream data to fill stream registers. However, it is necessary to ensure the synchronization between dimension iterations of each stream, according to the nesting level of the original data access and that of the corresponding computation.

Hence, an analysis is performed that detects the data flow of the computation between streams, matching it to their loop nesting level. This is done by building a DFG between the original scalar load/store instructions corresponding to each stream (see Fig. 3.B2). To ensure its correct computation flow, the DFG is checked for conditional control paths (indicated by an IR *select* instruction). If found, the paths are fused through the generation of the corresponding masking and predication instructions, latter applied to the final vector instructions. Then, vectorization is applied by synchronizing the iteration of descriptor dimensions that correspond to the induction variable of the lower nesting level of the loop. If required, the DFG is transformed by including scatter-gather operations to move data between vectorized and scalar streams (which are only affected by higher nesting level induction variables). Fig. 3.B2 illustrates the process of moving a non-vectorized stream access outside the loop and the introduction of a vector-to-scalar reduction operation to gather the loop result vector. Finally, when transformations are applied to the DFG, any streams that are moved in/out of the loop have their descriptors adjusted accordingly (see Fig. 3.B3).

After this procedure, the transformed DFG and stream descriptors are passed to the code generation phase of the proposed compilation flow.

Code Generation

The final stage of the proposed flow is responsible for translating the accelerated loops to UVE instructions. First, all loops where stream vectorization can be successfully applied are extracted into new functions (with built-in LLVM IR tools). Then, they are individually compiled for UVE [7].

At this stage, the code generation step operates by first encoding all stream descriptors with the corresponding UVE instructions, placing them in the correct loop preambles, according to the loop hierarchy. The DFG generated in the previous step is then used to build the corresponding UVE computation loop. This is done by generating the stream computing instructions corresponding to each operation and wrapping the loop with branch instructions tied to the correct stream dimensions, and according to the performed locality synchronization.This process is illustrated in Fig. 3.B.

Finally, the generated UVE code for each stream-vectorized loop is linked back to the remaining application code, which is compiled with the typical compilation flow (see Fig. 2.C).

EVALUATION

The proposed compilation flow was fully implemented in the LLVM 10.0.1 compiler infrastructure as an IR pass.

We evaluated the proposed compilation tool by comparing it with the data stream detection and encoding capabilities of the SSR compiler [5], and with the vectorization capabilities of the ARM SVE Compiler (configured with flags `-O3`, `-march=armv8-a+sve` and `-fsimdmath`). Comparisons against ARM SVE are done by using out-of-order processor setups (see Fig. 5.A) featuring 512-bit vectors, modeled using modified versions of the Gem5 simulator [7], [12]. A representative set of benchmarks from several application domains was used, as characterized in Fig. 4.A. When compiled with the proposed tool, all benchmarks resulted in assembly codes equivalent to the manual implementations from the originalUVE evaluation [7].

Data Streaming and Vectorization Comparison

The base mathematical model of the proposed compilation flow allows a significant coverage regarding the detection and description of memory access patterns. This is emphasized when comparing our tool with the SSR compilation flow [5] (see left column of Fig. 4.B), as SSR is only capable of streaming up to 4D access patterns with constant strides. Conversely, our mathematical model allows to describe induction and data dependencies in the form of inter-loop induction variable relations and indirect memory accesses. Such capabilities allow the proposed tool to accelerate a much broader range of loop hierarchies and enable vectorization, as it is often hardly done by existing compilers.

To highlight such advantages, the proposed tool was also compared with the ARM Compiler, which fails to vectorize five benchmarks (see Fig. 4.B), namely `Seidel-2D`, `MAMR` (both variants), `Covariance` and `Floyd-Warshall`. Conversely, the proposed tool is capable of handling the higher loop complexities of these benchmarks and achieve vectorization. This is a direct result of the memory access linearization that occurs with the introduction of the UVE data streaming paradigm.

Performance Comparison

The performance results (speed-up) presented in Fig. 4.C show that the compiled UVE code provides an average performance advantage (as high as 2.4×) over the ARM SVE (considering only the vectorized benchmarks). These gains result from two main contributions: *i)* significant code reductions (see Fig. 4.D), with an average 60.9% less committed instructions than ARM SVE; and *ii)* the streaming infrastructure, which significantly reduces the load-to-use latency and increases the effective memory hierarchy utilization. These advantages also contribute to a consequent reduction of the pipeline stalls, particularly at the rename stage. In fact, by decreasing the number of instructions in the code, UVE alleviates the pressure at the reorder-buffer and issue queue. On the other hand, by reducing the load-to-use latency, UVE allows incoming instructions to leave the pipeline earlier, decreasing the pressure on the physical register file.

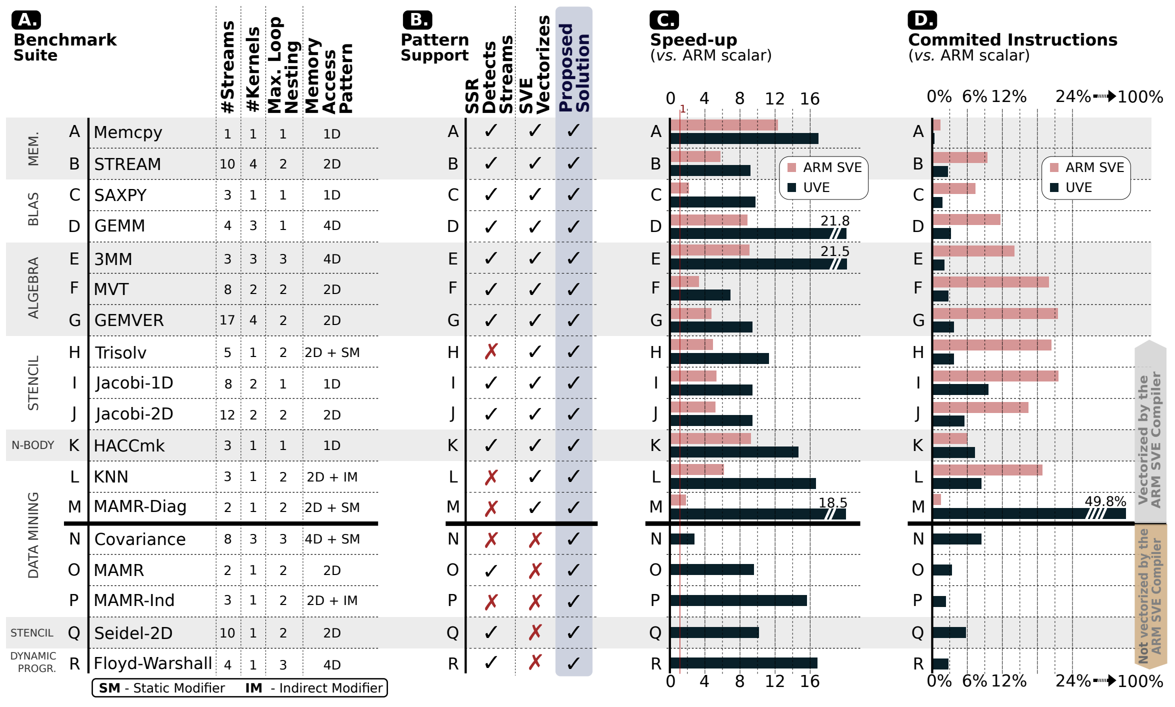
Such advantages are particularly highlighted when considering that the proposed compilation tool allows to vectorize a much wider range of applications than the ARM Compiler. In particular, they allow the UVE ecosystem to produce as much as 17× performance improvements over ARM SVE for the subset of benchmarks which the ARM compiler was not able to vectorize (see Figs. 4.B and C).

CONCLUSIONS

The recent resurgence of data streaming and data-flow paradigms on general-purpose contexts opened space for a new era of computing acceleration. However, their sustained viability can only be assured with the development of proper compilation support. Nevertheless, modern compilers still struggle to handle the unconventional data-flow and data-streaming execution models, generally incompatible with the SSA form often used by compiler IRs. To circumvent this limitation, we propose a new alternative compilation flow that leverages the LLVM IR to analyze a loop memory access patterns and data-flow graphs. With the gathered information, we obtain a high-level data streaming representation that can be directly compiled for stream-based vector extensions and linked to conventional machine code.

APPENDIX A - UNLIMITED VECTOR EXTENSION

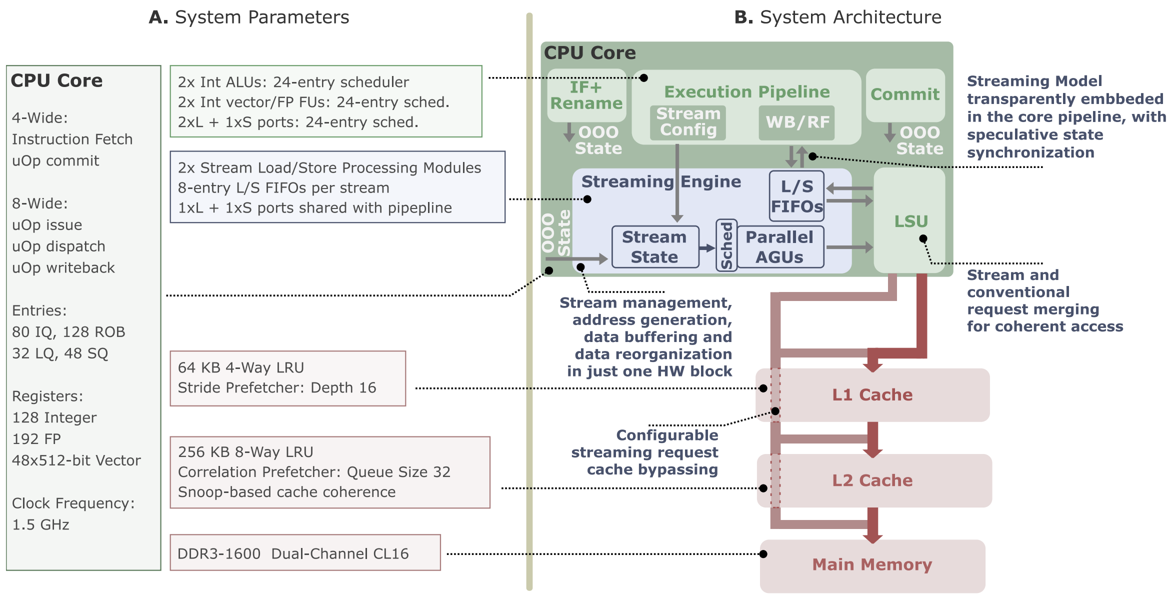
The main aim of the recently proposed Unlimited Vector Extension (UVE) is the combination of VLA processing with data streaming in mod-



		Benchmark	#Streams	#Kernels	Max. Loop Nesting	Memory Access Pattern	SSR Detects Streams	SVE Vectorizes	Proposed Solution
MEM.	A	Memcpy	1	1	1	1D	✓	✓	✓
	B	STREAM	10	4	2	2D	✓	✓	✓
BLAS	C	SAXPY	3	1	1	1D	✓	✓	✓
	D	GEMM	4	3	1	4D	✓	✓	✓
ALGEBRA	E	3MM	3	3	3	4D	✓	✓	✓
	F	MVT	8	2	2	2D	✓	✓	✓
	G	GEMVER	17	4	2	2D	✓	✓	✓
STENCIL	H	Trisolv	5	1	2	2D + SM	✗	✓	✓
	I	Jacobi-1D	8	2	1	1D	✓	✓	✓
	J	Jacobi-2D	12	2	2	2D	✓	✓	✓
N-BODY	K	HACCmk	3	1	1	1D	✓	✓	✓
DATA MINING	L	KNN	3	1	2	2D + IM	✗	✓	✓
	M	MAMR-Diag	2	1	2	2D + SM	✗	✓	✓
	N	Covariance	8	3	3	4D + SM	✗	✗	✓
	O	MAMR	2	1	2	2D	✓	✗	✓
	P	MAMR-Ind	3	1	2	2D + IM	✗	✗	✓
STENCIL	Q	Seidel-2D	10	1	2	2D	✓	✗	✓
DYNAMIC PROGR.	R	Floyd-Warshall	4	1	3	4D	✓	✗	✓

SM - Static Modifier **IM** - Indirect Modifier

Figure 4. Characterization of the adopted benchmark set regarding (A) loop structure and memory access pattern, (B) SSR stream detection and SVE vectorization coverage, (C) UVE performance, and (D) code reductions (measured in number of committed instructions during execution).


A. System Parameters
B. System Architecture
CPU Core
IF+ Rename
OOO State
Execution Pipeline
Stream Config
WB/RF
Commit
OOO State
Streaming Engine
L/S FIFOs
LSU
OOO State
Stream State
Sched
Parallel AGUs
L1 Cache
L2 Cache
Main Memory
Streaming Model transparently embbeded in the core pipeline, with speculative state synchronization
Stream and conventional request merging for coherent access
Stream management, address generation, data buffering and data reorganization in just one HW block
Configurable streaming request cache bypassing

CPU Core

4-Wide:
Instruction Fetch
uOp commit

8-Wide:
uOp issue
uOp dispatch
uOp writeback

Entries:
80 IQ, 128 ROB
32 LQ, 48 SQ

Registers:
128 Integer
192 FP
48x512-bit Vector

Clock Frequency:
1.5 GHz

2x Int ALUs: 24-entry scheduler
2x Int vector/FP FUs: 24-entry sched.
2xL + 1xS ports: 24-entry sched.

2x Stream Load/Store Processing Modules
8-entry L/S FIFOs per stream
1xL + 1xS ports shared with pipepline

64 KB 4-Way LRU
Stride Prefetcher: Depth 16

256 KB 8-Way LRU
Correlation Prefetcher: Queue Size 32
Snoop-based cache coherence

DDR3-1600 Dual-Channel CL16

Figure 5. System overview, depicting (A) the system model configuration parameters (based on public information about the ARM Cortex A76, and both [1], [7]), and (B) the system architecture featuring the UVE Streaming Engine embedded in an out-of-order core and its connections to the memory hierarchy. The Streaming Engine parameters are specific for UVE, with the remaining being common to UVE, baseline (ARM) and SVE.

ern general-purpose processors (RISC-V based), providing significant performance improvements over the state-of-the-art counterparts (e.g., ARM SVE). UVE essentially allows to describe data streams with specific instructions, as illustrated in Fig. 3.B1 and explained in the next subsections. A detailed specification can be found in [7].

Instruction Set Architecture

UVE is a scalable vector extension with support for all conventional data types, from byte to double-word, and the usual set of operations provided by RISC-V. Besides the natural adoption of a vector register file, it includes 32 predicate registers (allowing per-lane execution control and enabling control-dependent memory accesses), as well as a streaming interface that provides effective and timely prefetching of data, while facilitating vectorization by linearizing non-coalesced memory accesses. Each data stream is implicitly associated with a specific vector register, allowing any instruction to transparently consume data from (or produce to) the corresponding stream. As such, the progression/iteration of streams is automatic and happens after each interaction with the vector. With the adopted register predication, the boundary conditions of vector processing are automatically solved by disabling the outbound elements. This, in turn, allows loop control to be performed with only a basic set of stream-conditional branches.

The stream model is defined by using a hierarchical descriptor-based representation. It encodes each dimension of the affine formulation defined in Equation 1 in a set of dedicated instructions (see Fig 3.B1-*Code Generation*), while also providing mechanisms to combine multiple functions and to allow for complex and indirect access patterns.

Microarchitecture Baseline

To support UVE, the processor pipeline aggregates a dedicated *Streaming Engine* (Fig. 5) besides minor adaptations to its architecture. In particular, the decode stage, the register file, and some execution units are extended to support the UVE instruction-set extension, embracing new vector registers and the corresponding logic, arithmetic and branch functional units. Stream renaming (analogous to vector register renaming) is introduced to support the speculative configuration of new streams while others (with the same logical naming) are still executing. In the commit stage, it is added support for the commit and squash of streams, by signaling the *Streaming Engine* with all miss-speculation and commit events related to the streams under processing (configuration, iteration, and termination).

The *Streaming Engine* itself is responsible for managing the state of the streams and issuing memory requests. It consists of: *i)* a *Stream State Management* block, where the state is synchronized with the pipeline speculation state; *ii)* multiple *Address Generation Units (AGUs)*, which process the configured streams into the respective addresses (in parallel), issuing them to the core load/store unit (LSU); *iii)* a set of Load and Store FIFOs that buffer data between the core and memory, providing lower latency in the accesses.

The speculation state is synchronized between the relevant pipeline stages and the streaming engine, transparently embedding the latter, and ensuring that data is always processed in the program order with all data dependencies satisfied. The stream iteration process (after each read/write from/to a stream register) is handled through the iteration of the streaming engine FIFOs, where both speculative and effective (committed) states are present. Finally, to minimize the impact on caches and avoid the inclusion of additional L1 access ports, input/output stream requests are merged with conventional memory loads and stores, before accessing the L1 (see Fig. 5).

Acknowledgements

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects UIDB/50021/2020, UIDB/EEA/50008/2020, and PTDC/EEI-HAC/30485/2017, and by funds from the European Union Horizon 2020 Research and Innovation programme under grant agreement No. 101036168.

REFERENCES

1. N. Stephens *et al.*, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, pp. 26–39, 3 2017.
2. A. Waterman and K. Asanovic, "RISC-V "V" Vector Extension," tech. rep., RISC-V Foundation, 2019.
3. A. Pohl *et al.*, "A performance analysis of vector length agnostic code," in *International Conference on High Performance Computing & Simulation (HPCS)*, pp. 159–164, 2019.
4. Z. Wang and T. Nowatzki, "Stream-based Memory Access Specialization for General Purpose Processors,"

in *ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*, pp. 736–749, 2019.

5. F. Schuiki *et al.*, “Stream Semantic Registers: A Lightweight RISC-V ISA Extension Achieving Full Compute Utilization in Single-Issue Cores,” *IEEE Transactions on Computers*, vol. 70, no. 2, pp. 212–227, 2021.
6. N. Neves *et al.*, “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.
7. J. M. Domingos *et al.*, “Unlimited vector extension with data streaming support,” in *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, pp. 209–222, 2021.
8. T. Nowatzki *et al.*, “Stream-dataflow acceleration,” in *ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 416–429, 2017.
9. A. E. Şuşu, “A vector-length agnostic compiler for the connex-s accelerator with scratchpad memory,” *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 19, no. 6, pp. 1–30, 2020.
10. A. Brahmakshatriya *et al.*, “Taming the zoo: The unified graphit compiler framework for novel architectures,” in *ACM/IEEE 48th Annual International Symposium on Computer Architecture (ISCA)*, 2021.
11. C. Lattner *et al.*, “Mlir: Scaling compiler infrastructure for domain specific computation,” in *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 2–14, IEEE, 2021.
12. N. Binkert *et al.*, “The gem5 simulator,” *ACM SIGARCH Computer Architecture News*, vol. 39, p. 1, 8 2011.

Nuno Neves is a Researcher with INESC-ID and Instituto Superior Técnico, Universidade de Lisboa, Portugal. His research interests include data stream computing, domain-specific accelerators and compilers. He is a Member of the IEEE. Contact him at nuno.neves@inesc-id.pt.

Joao Mario Domingos is a Researcher with INESC-ID. His research interests include computer architectures and high-performance computing. He is a Student Member of IEEE. Contact him at joao.mario@tecnico.ulisboa.pt.

Nuno Roma is an Associate Professor with Instituto Superior Técnico, University of Lisbon, Portugal, and a senior researcher with INESC-ID. His research interests include computer architectures and high-performance computing. He is a Senior Member of both IEEE and ACM. Contact him at Nuno.Roma@inesc-id.pt.

Pedro Tomás is an Associate Professor with Instituto Superior Técnico, University of Lisbon, and a senior researcher with INESC-ID. His research interests include energy-efficient and high-performance computer architectures and systems. He is a Senior Member of IEEE. Contact him at pedro.tomas@inesc-id.pt.

Gabriel Falcao is a Tenured Assistant Professor with the University of Coimbra, and a Researcher with Instituto de Telecomunicações. His research interests include parallel computer architectures, energy-efficient processing, GPU- and FPGA-based accelerators, and compute-intensive signal processing applications. Gabriel is a Senior Member of the IEEE. Contact him at gff@co.it.pt.