S2VEC: Compiler-Driven Stream Specialization for Linearized Vectorization

Luís Crespo
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
luis.miguel.crespo@tecnico.ulisboa.pt

Ana Fernandes
Instituto de Telecomunicacoes, University of Coimbra
Coimbra, Portugal
ana.fernandes@co.it.pt

Gabriel Falcao
Instituto de Telecomunicacoes, University of Coimbra
Coimbra, Portugal
gff@co.it.pt

Pedro Tomás
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
pedro.tomas@inesc-id.pt

Nuno Roma
INESC-ID, Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
nuno.roma@inesc-id.pt

Nuno Neves
INESC-ID, Instituto Superior Tecnico, Universidade de Lisboa
Lisbon, Portugal
nuno.neves@inesc-id.pt

Abstract

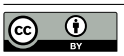
The performance benefits of vectorization are often limited by data movement, which remains a dominant bottleneck in modern processors. To mitigate this bottleneck, data stream-based mechanisms have been recently proposed, allowing general-purpose architectures to specialize memory accesses, leading to a reduction of load-to-use latency and loop code optimizations. However, compiler support for stream-based execution remains limited. Accordingly, this paper presents S2VEC, an LLVM-based compilation toolchain that automatically extracts, represents, and optimizes memory access patterns as data streams to enable stream-specialized vector execution. At its core, the Stream-Dataflow IR models computation as a dataflow over parameterized data streams, bridging the gap between conventional compiler infrastructures and stream-oriented architectures. By transforming complex and indirect memory accesses into linearized stream representations, S2VEC simplifies vectorization and expands its applicability to complex memory access patterns. When targeting a RISC-V stream-vector ISA extension, S2VEC demonstrates improved vectorization coverage and produces code comparable to hand-optimized implementations. Conducted evaluations on a gem5-based in-order stream-vector model show an overall speedup of 5.9× over an equivalently provisioned RVV-based in-order core and 1.5× over a wide out-of-order processor, demonstrating the performance benefits of compiler-driven stream specialization.

CCS Concepts

• **Software and its engineering → Compilers**; **Source code generation**; • **Computer systems organization → Single instruction, multiple data**; **Heterogeneous (hybrid) systems**.

Keywords

Compiler, LLVM, Intermediate Representations, Stream Specialization, SIMD, Vectorization



This work is licensed under a Creative Commons Attribution 4.0 International License.
ICS '26, Belfast, United Kingdom
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2522-7/26/07
https://doi.org/10.1145/3797905.3807866

ACM Reference Format:
Luís Crespo, Ana Fernandes, Gabriel Falcao, Pedro Tomás, Nuno Roma, and Nuno Neves. 2026. S2VEC: Compiler-Driven Stream Specialization for Linearized Vectorization. In *2026 International Conference on Supercomputing (ICS '26), July 06–09, 2026, Belfast, United Kingdom.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3797905.3807866

1 Introduction

Data movement and communication are still the dominant bottlenecks in modern computing systems, often limiting performance and energy efficiency despite continued growth in compute capability [13, 14, 21, 51]. While Single-Instruction Multiple-Data (SIMD) architectures improve computational throughput by exploiting data-level parallelism [25, 41, 49], they frequently exacerbate the memory bottleneck, due to the increased pressure on memory bandwidth and latency. In fact, the effectiveness of SIMD is frequently and strongly constrained by complex memory accesses.

From the compiler perspective, auto-vectorizers often struggle to expose the available parallelism when memory access patterns are complex or indirect, resulting in missed vectorization opportunities and suboptimal performance [2, 15, 24, 54]. From the architectural perspective, conventional SIMD execution models remain inefficient when handling irregular or dependent memory accesses, limiting their applicability across data-intensive workloads [10, 18, 24].

Recent works have shown that integrating data streaming mechanisms into General-Purpose Processors (GPPs) enables memory accesses to be offloaded to specialized hardware units, commonly referred to as streaming engines [6, 9, 29, 37, 40, 46, 48]. In particular, these engines eliminate indexing and address computation overhead in conventional execution by decoupling address generation and memory access from the core pipeline. This decoupling exposes a more regular, linearized view of data, improving coalescing and revealing vectorization opportunities that conventional auto-vectorizers often struggle to capture [1, 26, 27, 43]. Consequently, data streaming emerges as a powerful booster to SIMD execution, offering streamlined memory handling to enhance efficiency.

Moreover, GPP *stream specialization* [9, 40, 46] provide two additional and remarkable advantages. First, it enables precise, timely, and implicit prefetching, reducing effective memory latency without complex runtime prediction. Second, by removing explicit memory access instructions, address calculations, and induction-variable

management from the loop body, it significantly reduces dynamic instruction count and loop overhead. These mechanisms allow stream-specialized execution to outperform traditional SIMD approaches, particularly in memory-bound and irregular workloads [9]. Despite this progress, compiler support for data streaming remains limited and fragmented. Existing approaches mainly focus on detecting and encoding stream patterns, but overlook key optimization opportunities required to fully exploit stream-based execution, particularly in the context of SIMD and Vector-Length Agnostic (VLA) architectures [22, 29, 40, 42, 46, 48]. As a result, the lack of a robust compiler infrastructure remains a major barrier to the practical adoption of stream-enabled Instruction Set Architecture (ISA) extensions.

Compiling for stream-specialized execution introduces fundamental challenges. Stream-based models rely on implicit memory access and control flow, which contrasts with the explicit representation enforced by modern Static Single Assignment (SSA)-based compiler infrastructures [6, 27]. Furthermore, conventional auto-vectorizers are designed for regular and affine memory accesses, and struggle with complex, multi-dimensional, and indirect patterns [10, 16, 24, 31, 54], which streaming hardware can naturally linearize [9, 27]. Simply extending existing vectorizers is insufficient, as they lack mechanisms to eliminate explicit load/store memory operations and induction-variable-driven loop control. These limitations motivate the need for a dedicated compilation pipeline tailored to stream-specialized architectures. While higher-level compilation frameworks (such as MLIR [20]) facilitate the construction of domain-specific abstractions, general-purpose code must ultimately be lowered to LLVM IR for optimization and code generation. Consequently, supporting stream-specialized ISA extensions requires explicit integration with the LLVM middle-end, rather than relying solely on high-level representations.

In this work, we address this gap by proposing a novel LLVM-based compilation flow that automatically extracts, models, and optimizes memory access patterns as data streams, enabling stream-specialized SIMD execution. The implemented framework targets the Unlimited Vector Extension (UVE) ISA extension [9], which integrates vector execution with hardware data streaming and provides a concrete platform to demonstrate the benefits of our approach.

The primary contributions of this work are:

- Definition of a novel intermediate representation, termed Stream-Dataflow IR, that explicitly models stream-based memory access and computation, bridging the semantic gap between conventional compiler infrastructures and stream-specialized architectures.
- A set of LLVM middle-end analysis and transformation passes that automatically extract, represent, and optimize complex memory access patterns, including multidimensional, dependent, and indirect accesses.
- New stream-aware vectorization and loop transformation passes, including loop collapsing, that uniquely exploit implicit memory accesses and stream-driven control to reduce loop control overhead and improve SIMD utilization.
- A backend code generator for RISC-V that targets VLA SIMD execution with hardware data streaming.

The effectiveness of the proposed compilation flow is demonstrated by generating stream-specialized VLA-SIMD code for the UVE RISC-V extension [9, 11]. To evaluate the resulting performance, we extended the gem5 [3] simulator with a model of an in-order vector architecture with integrated data streaming support and compared it against RISC-V cores with RISC-V Vectors (RVV). When executing a comprehensive benchmark suite, our results show that stream-specialized execution achieves an average speedup of 5.9× over an equivalent in-order core. Furthermore, when compared against a superscalar out-of-order (OoO) processor, the prototype in-order stream-vector design still delivers considerable speedup, with an overall performance improvement of 1.5×, highlighting the benefits of compiler-driven stream specialization.

2 Background: Data Streaming

A data stream represents a sequence of memory accesses that follow a predictable address progression, typically derived from memory operations within loop nests. Such accesses can be compactly described using a parametric representation based on *offset* (initial address), *size* (number of accesses), and *step* (element-wise increment between consecutive accesses) [7, 11], with *step* correlating with *stride* through: stride = step×sizeof(element). Based on the access pattern structure, prior work [6, 9, 30, 37, 40, 46] distinguishes between direct and indirect data streams. Direct streams correspond to memory accesses whose parameters are statically determined, whereas indirect streams involve parameters that depend on values loaded from memory and are resolved at runtime (e.g., `A[B[i]]`).

2.1 Pattern Representation

To construct the full address sequence of a data stream, prior work models memory accesses as an n-dimensional pattern derived from the loop structure [9, 40, 50]. Each dimension corresponds to a loop induction variable and is characterized by a recurrence described by the {*offset*, *size*, *step*} tuple, analogous to the initialization, bound, and increment of a loop iterator. This formulation naturally captures nested-loop execution and enables the representation of multi-dimensional and dependent access patterns.

Fig. 1 depicts several examples of memory access patterns, together with their formal description as data streams. In particular, Fig. 1.A depicts a simple memory access to variable *A* with two dimensions, each associated with an induction variable (*i*, *j*). Each dimension is described by the {*offset*, *size*, *step*} tuple, inferred according to the corresponding address calculation expression and associated induction variable recurrence. Nested loops do not always follow a perfect n-dimensional structure, instead, they frequently present dependencies on other induction variables, such

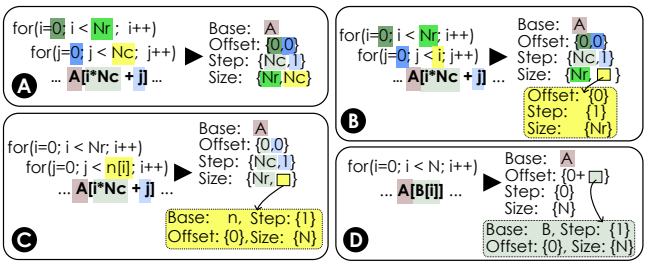


Figure 1: Example memory access patterns and the respective pattern description as data streams.

as the example depicted in Fig. 1.B, where the limit of the loop with induction variable *j* depends on the value of the induction variable *i*. In such cases, the affected parameter (loop size) needs to be described by a dependency on another tuple. Similarly to the previous example, a dependency that affects the number of repetitions of the pattern can also originate from another data stream (see Fig. 1.C). Hence, although the parameters may vary at runtime, since they are loop invariant throughout the affected loop execution (i.e., dimension), these relations can still be described [9, 29]. A last example is present in Fig. 1.D, where indirect patterns introduce dependencies on another data stream, where the memory access to *A* considers an offset that depends on a previous access to *B*.

2.2 Stream Specialization

Stream specialization involves enhancing typical processors with dedicated units to handle memory accesses in parallel with the core. These accesses are usually described with dedicated instructions that formally define data streams with a predictable memory access pattern (direct or indirect) configured at the preamble of loops (see Fig. 2). In this respect, recent solutions propose dedicated ISA extensions to configure data streams [9, 37, 38, 40, 46, 52] and mainly adopt two stream iteration approaches: explicit [46] and implicit [9, 40] (see Fig. 2.B and 2.C, respectively). Although both approaches perform the address computation and memory accesses without the direct intervention of the processor, data streams in explicit solutions require dedicated stepping instructions to advance a data stream. On the other hand, implicit approaches consider an automatic consumption and advancement of the data streams,

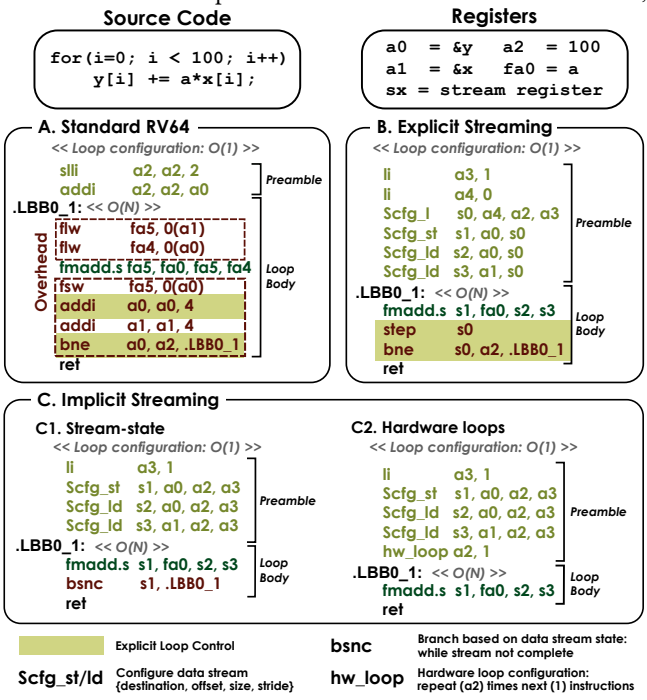


Figure 2: Pseudo-assembly of the saxpy kernel: (A) Standard RV64; (B) explicit and (C) implicit streaming. With induction variables removed, loop control can be managed by stream completion state (C1) or using hardware loops (C2).

making memory accesses completely transparent to the processor, and allowing the application code to be devoid of instructions for indexing, load/store, and induction variable-based loop control. Nonetheless, implicit data streaming models often have slight variations to ensure the correct loop iteration. In particular, some solutions leverage the iteration state of the data streams to control the loop flow [9] (see Fig. 2.C1), while others use hardware loops to configure the number of iterations [37, 38, 40, 53] (see Fig. 2.C2).

As this work offers a new compilation flow for stream-specialized SIMD processing, it will target the UVE [9, 11] ISA implicit streaming approach shown in Fig. 2.C1, since it is the most generic extension and the only one featuring SIMD execution. Yet, the proposed compilation toolchain could be extended to other targets.

3 Compilation Toolchain

Prior work on data streaming [27, 40, 42, 46, 50] has demonstrated the value of compiler support for identifying and encoding streamable memory accesses, typically through dedicated LLVM extensions. These approaches provide an important foundation by enabling the extraction of data streams from loop-based programs. However, they largely focus on stream detection and encoding, without fully addressing the broader compilation challenges required to exploit stream-specialized vector execution.

To bridge this gap, we propose a holistic compilation toolchain that integrates stream extraction, encoding, and optimization within the LLVM middle-end. Our approach leverages standard compiler analysis and transformation passes, including Scalar Evolution (SCEV), loop simplification, Loop Invariant Code Motion (LICM), Dead Code Elimination (DCE), and Instruction Combining (IC), preserving canonical loop structure while exposing the information required for stream-driven vectorization and code generation. **This integration enables stream specialization to be supported throughout the compilation pipeline, rather than being confined to a standalone stream-detection and encoding pass.**

3.1 Overview

The proposed compilation toolchain is depicted in Fig. 3. It takes a source program as input and lowers it to LLVM Intermediate Representation (IR) through a series of standard optimizations (without vectorization or loop unrolling). The IR is then analyzed and transformed into a Stream-Dataflow IR, where memory operations are represented as data streams and computations are organized into computational graphs and then into stream loops.

The Stream-Dataflow IR generation process consists of a series of custom analysis and transformation passes performed at the loop level. It starts with the **induction variables analysis** that consists of the extraction of induction variables and their pattern (i.e., *offset*, *size*, *step*). Then, during the **data stream description**, data streams are extracted by identifying memory accesses inside loops and extracting the corresponding access patterns, which are later optimized. In the **compute graph extraction** step, compute graphs are obtained via a Depth-First Search (DFS), starting at store instructions and tracing operands until finding the memory accesses corresponding to load instructions or constants. The **stream loop description** is obtained by doing a DFS through nested loops, to identify compute graphs and other nested loops.

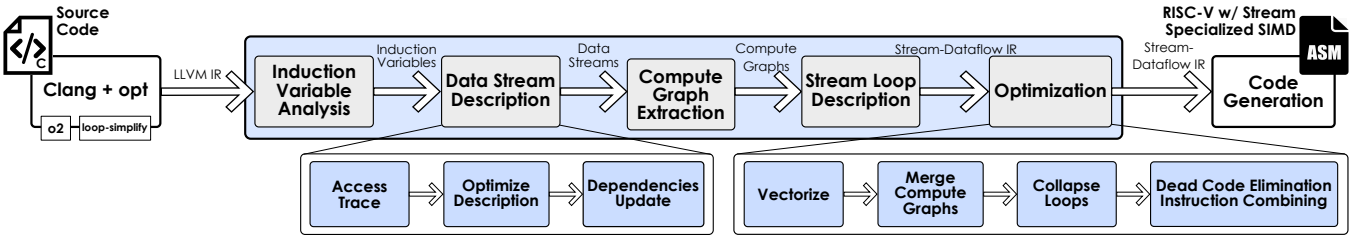


Figure 3: Overview of the proposed compilation flow.

Finally, a series of targeted **optimizations** are applied to the Stream-Dataflow IR, focusing on vectorization, compute graph merging, and loop collapsing. Additional standard optimizations, such as DCE and IC, are also integrated as available in LLVM. The last stage of the flow handles the stream-specialized RISC-V source code generation (from the Stream-Dataflow IR).

3.2 Stream-Dataflow IR

To support compilation for stream-vector architectures, we introduce a custom intermediate representation called the **Stream-Dataflow IR**. This IR aims to model a loop-based execution, where memory accesses are expressed as parameterized data streams and computation is represented as dataflow graphs evaluated over stream iterations. The Stream-Dataflow IR is the primary output of the set of analysis and transformations performed over the LLVM IR. It acts as an intermediary representation between the LLVM IR and the lower machine code, ensuring that any necessary transformations are applied to fully exploit stream-vector architectures. At a high level, the Stream-Dataflow IR is composed of **Stream Loops**, **Compute Graphs**, **Data Streams**, and **Induction Variables**, as illustrated in Fig. 4. Each component is defined in more detail below.

Stream Loops: A *Stream Loop* is the top-level abstraction of the Stream-Dataflow IR, representing the execution of compute graphs over a stream-driven iteration space. It is defined as:

$$Stream\ Loop = (I,\ S,\ B),$$

where $I = \{i_0, \ldots, i_n\}$ is the set of induction variables associated with the loop, S is a control stream together with a designated dimension governing the loop termination, and B is an ordered sequence of loop body elements (either compute graphs or nested stream loops). Unlike traditional loops, Stream Loops do not require explicit induction variable updates or branch-based control. Instead,

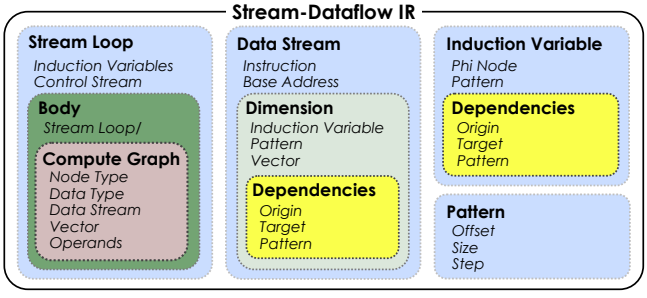


Figure 4: Stream-Dataflow IR composition.

loop progress and termination can be driven by the state of the control stream, enabling implicit loop control, as shown in Fig. 2.C.

Compute Graphs: A *Compute Graph* represents the sequence of operations performed during each iteration of a Stream Loop. A compute graph is organized as a Directed Acyclic Graph (DAG):

$$Compute\ Graph = (V, E),$$

where each node $v \in V$ is defined as:

$$v = (op,\ \tau,\ S,\ vec),$$

with op denoting the node type (e.g., arithmetic operation, data streams, constants), τ the data type, S an optional associated data stream, and vec indicating scalar or vector execution. Edges E represent the operand relationship of the node.

Data Streams: A *Data Stream* allows to model a memory access with a predictable and parameterized address sequence as a data stream. A data stream is defined as:

$$S = (T,\ A,\ D)$$

where T is the originating memory instruction (load or store), A is the base address, and $D = \{D_0, \ldots, D_k\}$ is the set of stream dimensions, each corresponding to a loop nesting level, defined as:

$$D_j = (i_j,\ P_j,\ vec_j,\ M_j)$$

where i_j is the associated induction variable, P_j describes the pattern (*offset*, *size*, and *step*) for that dimension, vec_j indicates whether the dimension is vectorized, and M_j is a set of optional dependencies affecting this dimension. While the induction variable drives each dimension, the address sequence is given by the pattern and its dependencies (as seen in Fig. 2).

Dependencies: *Dependencies* are entities in the Stream-Dataflow IR that capture relationships between induction variables and other induction variables or data streams. A dependency is defined as:

$$Dependency = (O,\ Tg,\ P)$$

where the origin (O) is either an induction variable or a data stream, and the target (Tg) identifies the affected pattern field (offset, size, or step). The dependencies of data streams and induction variables inherit the pattern from the O that produces it.

Induction Variables and Patterns: *Induction Variables* and *Patterns* are the base structures that define the iteration behavior of loops and data streams. An *Induction Variable* models a loop iterator and its iteration behavior. Each induction variable is defined as:

$$i = (\phi,\ P,\ M)$$

where ϕ is the PHI node corresponding to the loop iterator, P denotes the iteration pattern, and M is a set of dependencies affecting the induction variable. P is defined as a tuple (*offset*, *size*, *step*), which is the fundamental abstraction used throughout the IR. The *offset* is the initial value, *size* is the number of iterations, and *step* is the increment or decrement per iteration.

Stream-Dataflow IR Grammar. The abstract syntax of the Stream-Dataflow IR is summarized as follows:

$$
\begin{aligned}
\textit{Kernel} &::= \textit{Stream Loop}^{+} \\
\textit{Stream Loop} &::= (I,\ S,\ B) \\
I &::= \{i_0, \ldots, i_n\} \\
B &::= \textit{Body Elem}^{+} \\
\textit{Body Elem} &::= \textit{Compute Graph} \mid \textit{Stream Loop} \\
\textit{Compute Graph} &::= (V,\ E) \\
v \in V &::= (op,\ \tau,\ S?,\ vec) \\
S &::= (T,\ A,\ D) \\
D &::= \{D_0, \ldots, D_k\} \\
D_j &::= (i_j,\ P_j,\ vec_j,\ M_j) \\
i &::= (\phi,\ P,\ M) \\
P &::= (\textit{offset},\ \textit{size},\ \textit{step}) \\
M &::= (O,\ Tg,\ P)
\end{aligned}
$$

A kernel consists of one or more stream loops, each composed of induction variables, a control stream, and an ordered body of compute graphs or nested stream loops. Computation within a stream loop is expressed as dataflow graphs evaluated once per stream iteration, where vertices represent operations or stream accesses and edges represent operands. Data streams model memory accesses as multi-dimensional objects, whose dimensions are driven by induction variables and described by iteration patterns. Dependencies relate induction variables and data streams, and are attached to stream dimensions to capture complex pattern relationships.

3.3 Automatic Stream-Dataflow IR Extraction

The proposed extraction method starts by examining the LLVM IR of the application kernel by using its built-in analysis passes to identify the loop hierarchy. Then, each loop hierarchy is analyzed and transformed to Stream-Dataflow IR. As mentioned above, this process consists of identifying induction variables, data streams, compute graphs, and stream loops in each loop hierarchy.

Induction Variable Analysis: Induction variables are identified in the top-level loop and corresponding nested loops by analyzing *PHI* instructions in the loop headers. The iteration pattern and dependencies are obtained by tracing the IR (see Algorithm 1). The *offset* (i.e., the starting value) is directly extracted from the *PHI* instruction. The *size* and *step* are derived from the loop exit condition (in the loop latch) by tracing its operands: the *step* corresponds to the increment (or decrement) value over the induction variable, while the *size* is computed from the loop *limit*. As shown in Fig. 1, dependencies may arise from loop limits (*init* or *limit*) of the loop, either through another induction variable or data accesses. These

Algorithm 1 Induction Variable Analysis

1: **Procedure** IndVarAnalysis(L) // L: loop
2: $\phi \leftarrow$ primary induction PHI in L's header
3: *offset* $\leftarrow$ initial value of ϕ
4: *step* $\leftarrow$ increment operand traced from ϕ update
5: *limit* $\leftarrow$ bound extracted from latch exit condition
6: **if** *offset* or *limit* depends on PHI or access **then**
7: record dependency (OFFSET or/and LENGTH)
8: $size \leftarrow \left\lceil \frac{limit - offset}{step} \right\rceil$
9: store IndVar {ϕ, *offset*, *step*, *size*, *dependencies*}

are identified during the trace and are translated to dependencies over the corresponding target. Dependencies over data accesses that are also data streams are resolved after extracting the data streams (at dependency update, where the access pattern is known).

Data Stream Description: One of the most important aspects of the compilation flow is the automatic identification of data streams and their access patterns. Data streams are detected in the LLVM IR by searching for memory access instructions inside loops. For each access, a trace over the corresponding *GetElementPtr* instruction is performed to obtain the address sequence of the data streams. The process is described in Algorithm 2. While the base address is directly exposed by the *GetElementPtr*, the address sequence is obtained by analyzing its offset through a DFS. The operands are recursively searched until finding the instructions corresponding to induction variables, constants, or other memory accesses. These are translated into several stream dimensions in the following manner:

- Induction variables are associated with the data stream and have their pattern and dependencies copied as dimensions.
- Constants have no induction variable. They are added to an adjacent dimension or multiplied by the step value.
- Memory accesses that are streams have their highest dimension (most nested loop) copied with offset and step values in the pattern set to 0. This is encoded as an offset dependency with a pointer to the respective data stream.

The identified stream dimensions are then associated with the data stream description, according to the traced computation instructions. During the DFS, transformations are applied depending on the instruction type: additions (or subtractions) append new dimensions, while multiplications (or shifts) modify the *step* of the found dimensions (as described above). With these, the access trace is completed, and the memory accesses are represented in data stream form. These data streams are then optimized by collapsing offset dependencies into *step* values, merging constant dimensions as offsets, and collapsing dimensions with the same induction variables. The *step* and *offset* of the resulting dimension are then adjusted accordingly. Alternatively, SCEV could be used [6] by applying the same process to the recurrences.

Finally, the dependencies on the loop limits on other data streams are handled (i.e., association of data stream and pattern), both in the induction variables and the data streams that are driven by these induction variables. The inclusion of these dependencies on the induction variables and data streams is postponed until this step to guarantee that all data streams are fully optimized.

Algorithm 2 Data Stream Extraction

```
1:  Procedure ExtractDataStreams(BLL, IndVars)
2:    Streams ← ∅
3:    for each basic block BB in BLL do
4:      for each instruction Inst in BB do
5:        if Inst is Load or Store then
6:          S ← new DataStream
7:          S.I ← Inst
8:          Ptr ← (Inst is Load ? Inst.Op[0] : Inst.Op[1])
9:          S.BaseAddr ← Ptr.Op[0]
10:         CreateDims(S, Ptr.Op[1], IndVars)
11:         Streams ← Streams + S
12:   return Streams
13: Procedure CreateDims(S, offset, IndVars)
14:   if offset is constant or outside loop then
15:     S.Dimensions + Dim(Offset= offset, Step= 0)
16:   else if offset == induction variable IV then
17:     S.Dimensions + Dim(IndVar= IV)
18:   else if offset is another stream S2 then
19:     S.Dimensions + Dim(Stream= S2)
20:   else                                  // recursive trace by operands
21:     CreateDims(S, offset.operands, IndVars)
22:     if Add/Sub then append dimensions
23:     else if Mul/Shl then update Step and Init
```

Compute Graph Extraction: Since data stream specializations decouple the memory accesses from the computations, the loop bodies of stream-specialized code are mainly composed of computations (see Fig. 2). Accordingly, these computation sequences are extracted as compute data-flow graphs, where nodes represent constant values, input streams, or computations. The adopted method consists of performing a DFS starting at each store data stream, moving backwards in the compute graph to obtain the respective computations and operand dependencies until finding the input data streams or constants. In Fig. 5.A, it is depicted an example compute graph with computation and data stream nodes represented in a hybrid IR form. This information, together with the respective data stream description, is kept in the Compute Graph within the Stream-Dataflow IR (see Fig. 4). Additionally, during the DFS process, indirect dependencies (i.e., dependencies on other data streams) are also recorded as operands of the respective data stream for synchronization and code generation purposes (see stream S3 in Fig. 5.A that is registered as an operand of stream S2 to be later synchronized in Fig. 5.C).

Stream Loops Description: The process of obtaining the Stream-Dataflow IR is concluded by defining a *Stream Loop* for each top-level loop being analyzed and traversing its nested loops in a DFS manner. This builds a structured representation of the compute graphs at each loop level, synchronizing data streams to match the nesting depth. An example *Stream Loop* is depicted in Fig. 5.B, where induction variables, compute graphs, and nested loops are assigned to a parent *Stream Loop*. Additional steps ensure loop control by assigning a data stream to control the loop whose dimension matches the loop iteration. This way, a representation of the control based on the stream state is achieved (see Fig. 2.C1). Finally, data streams that are iterated within a particular loop but are invariant to

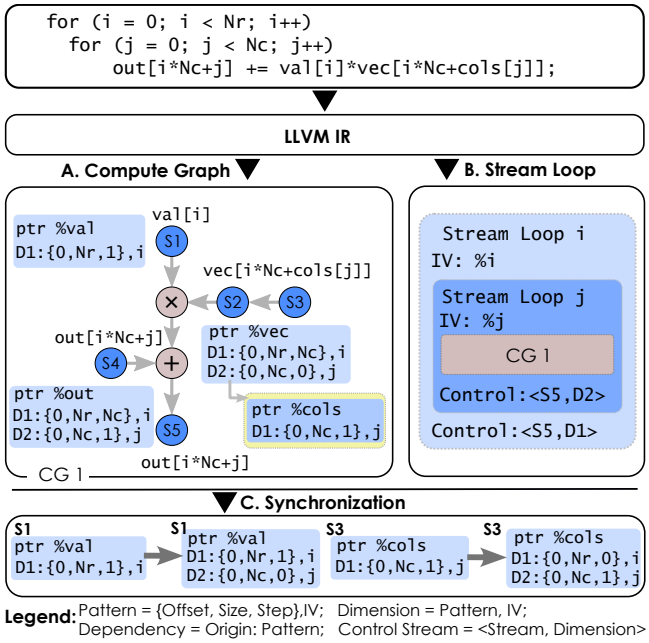


Figure 5: Hybrid IR form of a kernel. Compute Graphs are obtained (A) and inserted in the corresponding Stream Loop (B). Then, the data streams within the Compute Graphs are synchronized according to the parent Stream Loops (C).

that loop are modified to include one (or multiple) extra dimensions to ensure a correctly synchronized stream iteration (see Fig. 5.C). The parameters of that particular dimension follow the associated induction variable of the loop, except for the step, which is set to 0.

3.4 Optimization Passes

The proposed compilation toolchain features dedicated optimization passes that can be applied over to the Stream-Dataflow IR, including vectorization, merging of compute graphs, and collapsing of stream loops. The tool also leverages the native LLVM functionalities to apply the DCE and IC native passes to further optimize the code. They are especially important to avoid redundant operations when computing the *size* and *steps* of the data stream patterns.

Stream-based Vectorization: The first optimization pass is responsible for vectorizing the compute graphs within the extracted Stream-Dataflow IR. To do so, each compute graph is individually analyzed, with vectorization applied to the innermost loops. Vectorization is functionally achieved by allowing stream data to effectively fill in vector registers, resulting in a simplified vectorization process. Specifically, each compute graph is checked for conditional control paths (indicated by an IR `select` instruction), and the data streams are checked for dependencies. While control is converted to masking and predication instructions, loop-carried dependencies make the code non-vectorizable. Nonetheless, non-vectorizable code can still be stream-specialized, albeit only in scalar form. Furthermore, reductions in vectorizable compute graphs are identified and transformed with horizontal operations to move data between vector and scalar streams. This process involves breaking the compute graph and inserting vector-to-scalar reduction operations.

Merging of Compute Graphs: This functionality is essential to ensure vectorized code correctness with vector-to-scalar reductions, to handle computations at the loop exit (inserted after reductions), and to avoid loading a value that is guaranteed to be available. Merging of compute graphs uses the following criteria: *C1)* when a compute graph is split to vectorize a reduction, any compute graph outside the loop using the reduced value needs to be updated (merged) with the correct scalar value; *C2)* when there are two consecutive store data streams to the same position, one is removed and the compute graphs are merged; and *C3)* when there is a load-after-store on the same memory address within the same stream loop, the compute graphs are merged to avoid the redundant load.

Loop Collapse: The third pass attempts to collapse perfectly nested loops into a single loop. This applies when there are no compute graph nodes or other nested stream loops in the outer loop. The resulting stream loop preserves all implicit induction variable details (including those from removed loops) to ensure correct stream-state control. Due to the decoupled execution nature of stream specialization (i.e., no induction variables and memory indexing at machine code level), this technique reduces loop control overhead (usually found in conventional systems [34]). Moreover, stream specialization also allows removing the overhead of modulo and division operations otherwise required to compute array indexing functions of typical loop collapsing optimizations [34].

3.5 Code Generation

The extracted information is used to generate RISC-V code by considering the target ISA, namely UVE [9, 11][1], as it currently is the only RISC-V extension supporting stream-specialized SIMD execution. Data streams are first encoded with the corresponding stream configuration instructions and placed in the proper loop preambles. Each stream begins with *start* instruction specifying the access type (load/store), the *size* of the data elements (8- to 64-bits), the *base address*, and the *vectorized dimension*, among other optional configurations. Then, for each dimension, an *append* instruction follows, specifying its (*offset*, *size*, *step*), until the final *end* instruction. Dependencies are represented as modifiers coupled to a specific dimension, and can be *static* (`SM`, for dependencies on other induction variables), dynamic (`DM`, for loop invariant dependencies on other data streams), and scatter-gather (`SG`, for offset dependencies on other data streams, i.e., indirection). Each modifier also specifies the affected dimension, the modified pattern field, and a displacement value. Dynamic and scatter-gather modifiers link another data stream instead of a displacement to the target dimension.

The replication of inputs to vector registers is also inserted on the stream loop preamble. Stream loops and the corresponding bodies containing nested loops and compute graphs are then generated. This is done by generating the stream computing instructions corresponding to each operation and wrapping the loop with stream-driven branch instructions specifying the control stream and respective dimension. Finally, the generated code for each top-level stream loop is linked back to the remaining application code, eliminating the original loop code, which is compiled with the typical compilation flow (through the LLVM backend).

[1]Documentation: https://github.com/hpc-ulisboa/UVE2/blob/main/documentation.pdf

4 Toolchain Validation

The proposed compilation tool was validated and evaluated in terms of coverage and resulting code efficiency according to the publicly available specification UVE [11] ISA[1].

4.1 Validation Environment

Compilation Setup: The proposed compilation flow was implemented in LLVM 16 as an IR-level pass, and the LLVM back-end was extended to support UVE [11]. To evaluate the impact of stream specialization on auto-vectorization, we also used the ARM and LLVM 21 compilers to generate vectorized code for ARM SVE [41] and RISC-V RVV [49] with `-O3` optimizations.

Benchmark Applications: The experimental setup considers 31 vectorizable benchmarks from the PolyBench [23] and MachSuite [35] suites (see Table 1). Alongside each benchmark, it is specified the respective number of kernels, maximum loop nesting, number of data streams, and most complex memory access pattern. The patterns are described with the number of dimensions (nD) plus associated dependencies. Table 1 also presents the encoding capabilities of XSSR [37, 38, 40] (a stream specialization solution for RISC-V without vectorization support), the auto-vectorization results for the RVV [49] and ARM SVE [41] SIMD extensions, the original UVE compiler [27] capabilities, and the proposed compilation tool.

Functional Validation: The generated code was validated on a modified version of the *Spike* Instruction Set Simulator (ISS) [11]. This simulation infrastructure allows for a functional validation of the UVE code, outputting the dynamic instruction count metric, which will be used for validation of the compiler-generated code.

4.2 Compilation and Vectorization Coverage

The effectiveness of the proposed approach was assessed across the selected benchmarks by measuring the coverage at both the compiler and vectorization levels.

Compiler level: Observing the supported benchmarks presented in Table 1, it can be concluded that the proposed compilation flow and the considered SIMD stream specialization abstraction significantly broaden the coverage of supported workloads when compared to previous stream-based approaches. In particular, the XSSR compiler [37, 38, 40] can only extract simple square affine patterns, and this limitation was not considered in later works to automatically extract and encode dependencies [37, 38]. Furthermore, while the UVE compiler [27] supports the extraction of more complex memory access patterns than XSSR [37, 38, 40] and features SIMD operations, it still encounters challenges with benchmarks involving: multiple dependencies (e.g., *Cholesky*, *SpMV-ind*), dependencies spanning multiple nested loops (e.g., *SYMM*, *SYRK*), or particular operations (e.g., sqrt in *Gram-Schmidt* is unsupported). Moreover, as it only defines data streams and compute graphs, it has limited synchronization capabilities and nested loop support (e.g., *DoitGen*).

Vectorization level: As shown in Table 1, the proposed compilation tool promotes the vectorization of a broader spectrum of benchmarks when compared to ARM SVE [41] and RVV [49] compilers, which struggle with complex memory access patterns, particularly when handling loops with intricate dependency chains (e.g., *LU*)

Table 1: Supported benchmarks and their characterization.

	Benchmark	#Kernels (Nesting)†	#Streams (Pattern)‡	XSSR Stream Support [38]	Vectorizes			
					RVV [49]	SVE [41]	UVE Current Compiler [27]	*Proposed*
Mem.	**Memcpy**(C)	1 (1)	1 (1D)	✓	✓	✓	✓	✓
	Stream(C)	4 (2)	10 (2D)	✓	✓	✓	✓	✓
BLAS	**AXPY**(C)	1 (1)	3 (1D)	✓	✓	✓	✓	✓
	GEMM(P)	2 (3)	6 (3D)	✓	✓	✓	✓	✓
	GEMM-ncubed(M)	2 (3)	6 (3D)	✓	✓	✓	✓	✓
	GEMM-blocked(M)	2 (3)	6 (3D)		✓	✓	✓	✓
	GEMVER(P)	4 (2)	17 (2D)	✓	✓	✓	✓	✓
	GESUMMV(P)	1 (2)	10 (2D)	✓	✗	✗	✓	✓
	SYMM(P)	1 (3)	13 (3D+SM)	✗	✓	✓	✗	✓
	SYRK(P)	1 (3)	6 (3D+SM)	✗	✓	✓	✗	✓
	SYR2K(P)	1 (3)	8 (3D+SM)	✗	✓	✓	✗	✓
	TRMM(P)	1 (3)	4 (3D+SM)	✗	✓	✗	✗	✓
Kernels	**ATAX**(P)	1 (2)	7 (2D)	✓	✓	✓	✓	✓
	2MM(P)	2 (3)	8 (3D)	✓	✓	✓	✓	✓
	3MM(P)	3 (3)	9 (3D)	✓	✓	✓	✓	✓
	BIGC(P)	1 (2)	8 (2D)	✓	✗	✓	✓	✓
	DoitGen(P)	1 (4)	5 (4D)	✓	✓	✓	✗	✓
	MVT(P)	2 (2)	8 (2D)	✓	✓	✓	✓	✓
Solvers	**Cholesky**(P)	1 (3)	13 (3D+2SM)	✗	✓	✗	✗	✓
	Gram-Schmidt(P)	1 (3)	13 (3D+SM)	✗	✓	✓	✗	✓
	LU(P)	1 (3)	9 (3D+2SM)	✗	✗	✗	✗	✓
	TriSolv(P)	1 (2)	5 (2D+SM)	✗	✓	✗	✓	✓
Stencil	**FDTD-2D**(P)	1 (3)	16 (3D)	✓	✓	✓	✓	✓
	Jacobi-1D(P)	2 (1)	8 (1D)	✓	✓	✓	✓	✓
	Jacobi-2D(P)	2 (2)	12 (2D)	✓	✓	✓	✓	✓
	Stencil-2D(M)	2 (2)	12 (2D)	✓	✓	✓	✓	✓
Misc.	**Convolution**(C)	1 (1)	10 (2D)	✓	✓	✓	✓	✓
	Covariance(P)	3 (3)	9 (4D+SM)	✓	✓	✓	✓	✓
	KNN(M)	1 (2)	5 (2D+SG)	✓	✗	✓	✓	✓
	SpMV(M)	1 (2)	5 (2D+SG)	✓	✗	✗	✓	✓
	SpMV-ind(C)	1 (2)	8 (2D+SG+DM)	✓	✗	✗	✗	✓

† Maximum nested loops in the benchmark across all kernels
‡ Most complex memory access pattern (load or store) across all streams
SG – Scatter-gather (stream dependency); S/DM – Static/Dynamic Modifier (induction dependency)
(P) / (M) / (C) Benchmark from PolyBench [23], MachSuite [35] or custom, respectively

and indirections (e.g., *KNN*, *SpMV*). All the considered benchmarks were vectorized with stream specialization by the proposed tool.

4.3 Code Efficiency

The efficiency of the code generated by the proposed compilation flow was measured by counting the number of executed instructions in Spike. In a first step, this metric was compared against hand-optimized implementations available online to establish a baseline comparison. In a second step, this metric was compared with RVV and scalar RISC-V code to assess architecture-agnostic advantages. Particular attention was also given to the impact of stream specialization, studying how code efficiency scales with different numbers of streams and varying the workload sizes.

Compiler: Fig. 6 shows the percentage reduction in executed instructions achieved by handwritten UVE code, LLVM-generated RVV code [49], and auto-vectorized code produced by the proposed compilation flow, all relative to scalar execution. Compared with handwritten implementations, the compiler-generated code attains similar instruction-count reductions in most cases, despite minor advantages from manual optimizations. Differences arise mainly in kernels such as *3MM* and *Covariance*, where handwritten code exploits Fused Multiply-Add (FMA) operations, not yet fully supported by the compiler, which must distinguish between register accumulations and fused operations (incompatible with store streams). In most benchmarks, instruction counts are identical or differ slightly due to precomputed stream parameters (*offset*, *size*, and *step*) in handwritten versions. Overall, these results show that the proposed compilation flow generates near hand-optimized code.

Vectorization: The comparison of the number of executed instructions of the auto-vectorized stream specialized code with RVV [49] (see Fig. 6) highlights key differences. Our approach consistently shows a notable advantage, while RVV [49] performs worse than scalar code in some cases (e.g., *2MM*, *TRMM*). Overall, while both extensions surpass scalar code, joining SIMD with stream specialization consistently allows executing fewer instructions than RVV [49] across all benchmark scenarios, while handling more complex cases.

Scaling: As it is characteristic of vector ISAs, increasing the vector size amortizes loop iterations and reduces the total number of executed instructions. Fig. 7 shows this variation on a synthetic benchmark that scales with the problem size and number of streams, by comparing the proposed toolchain with RVV. In particular, Fig. 7.A shows that both RVV [49] and the code generated by the proposed compiler yield greater instruction reduction with increasing vector sizes and number of concurrent data streams. Nevertheless, the introduced data stream specialization consistently executes fewer instructions than RVV [49]. Fig. 7.B shows that as the number of input streams increases for the same number of arithmetic operations (i.e., moving from compute-bound to memory-bound), the code generated by the proposed compiler maintains a high instruction reduction (slightly increasing), while RVV shows a declining trend.

5 Performance Evaluation

After quantifying instruction-count reductions using the Spike functional simulator, we evaluate the performance impact of the proposed compilation toolchain through detailed cycle-accurate simulation. As prior UVE compiler implementations [27] rely on outdated ISAs and non-reproducible toolchains, direct comparison is not possible. Nevertheless, besides supporting a wider set of applications, performance gains are expected from the additional optimizations (e.g., compute-graph merging, description optimization, loop collapsing). Due to the lack of support for the latest UVE specifications in existing tools, we implemented a UVE-capable vector coprocessor. Then, we compared the performance of compiler-generated code, executing under a stream-vector model, against a closely matched in-order processor with RVV, to isolate the benefits of data streaming. Finally, we extended this evaluation to a superscalar OoO core, also supporting RVV, to assess competitiveness with modern high-performance microarchitectures.

5.1 Evaluation Environment

The devised UVE-based vector coprocessor, implemented on top of NDPmulator [44] and illustrated in Fig. 8, adopts a simple in-order, single-issue pipeline organized into five stages: *fetch*, *decode*, *issue*, *execute*, and *write-back*. To ensure a realistic scenario, this vector

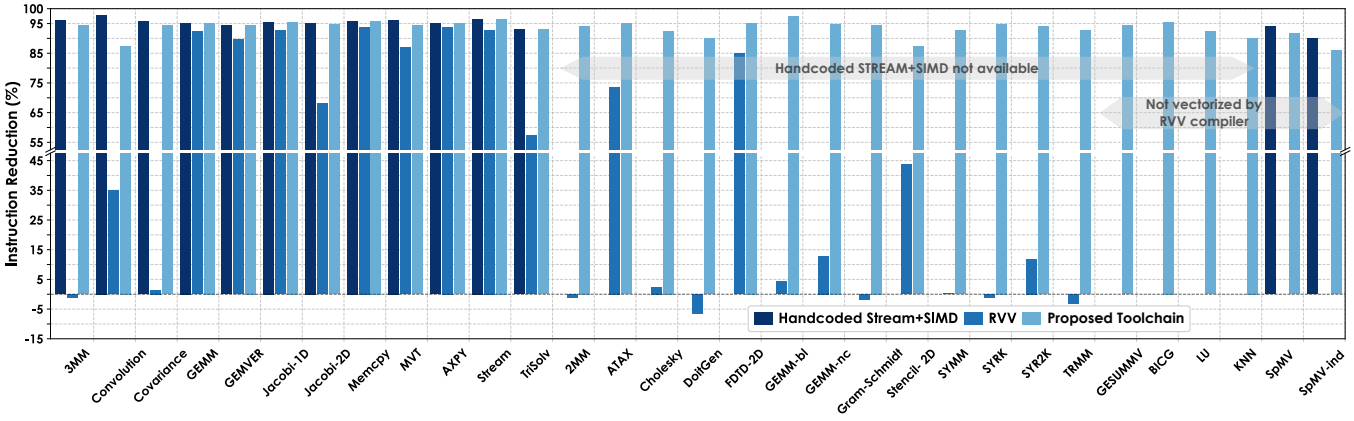


Figure 6: Reduction of retired instructions by handcoded stream-specialization, RVV [49], and auto-vectorized code produced by the proposed compilation flow, relative to scalar code (higher is better).

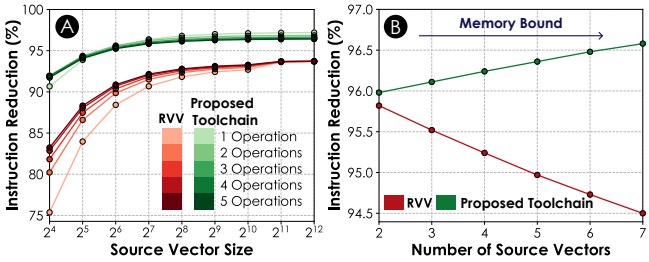


Figure 7: Instruction reduction provided by RVV [49] and by the proposed toolchain according to (A) dataset size and number of loop operations, and (B) number of source vectors. Both setups considered a 512-bits vector length, 7 loop operations, and a 10^4 source vector size.

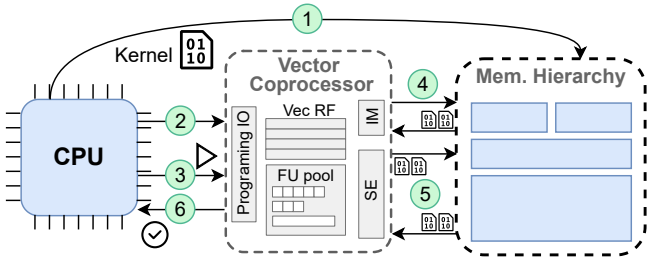


Figure 8: Operation of the coprocessor: (1) the CPU stores the kernel in memory; (2) the CPU writes to the programming IO the register values, kernel address, and kernel size; (3) the CPU triggers the coprocessor execution; (4) the coprocessor fetches the kernel from memory hierarchy and saves it in a local instruction memory; (5) the coprocessor exchanges operands and results with the memory hierarchy through the streaming engine; (6) the CPU polls the coprocessor.

coprocessor follows a long-vector execution model [25], with a physical vector register of 4×512 bits, that is split into four 512-bit slices. The engine sustains a peak execution throughput of 512 bits per cycle, processing one 512-bit slice per cycle. Consequently, a full 4×512-bit vector requires four cycles to complete. A key component is the streaming engine, which manages the data streams independently of the core pipeline, decoupling memory accesses from computation. It maintains the stream state, generates memory requests, and buffers data through dedicated FIFOs.

We compare the compiler-generated code running on the coprocessor prototype against a 4-wide in-order CPU with RVV support, which closely reflects the execution model of our design and therefore provides the most direct and fair comparison. Table 2 (left) characterizes the reference core. Aside from the issue width, both architectures are provisioned with identical vector datapath width (512-bit), memory hierarchy parameters, arithmetic functional units, and a 64-byte cache line size. Hence, the reference CPU sustains the same peak vector throughput of 512 bits per cycle, ensuring identical peak compute bandwidth; therefore, the observed performance differences stem from the streaming execution model rather than the raw vector throughput. Moreover, the in-order reference CPU can issue 4 instructions per cycle, enabling multiple instructions to be executed concurrently when targeting different functional units (contrasting with the single-issue evaluation prototype).

We further compare against a significantly more powerful OoO microarchitecture, modeled as an 8-wide superscalar RISC-V core (also supporting RVV) configured to approximate an AMD Zen 5 processor (Table 2, right). Unlike the coprocessor, this design features wider issue width, dynamic scheduling, speculative execution, and improved latency tolerance, providing a strong performance-oriented baseline. It also delivers twice the peak vector execution throughput (1024 bits/cycle versus 512 bits/cycle in our design) and incorporates a substantially more advanced load/store pipeline.

The performance evaluation described in the following subsections considers a representative subset of the benchmarks considered in Table 1, using different dataset configurations.

5.2 Comparison with Equivalent In-order Core

Fig. 9.A reports the performance relative to the 4-wide in-order baseline processor. As expected, the speedup generally increases

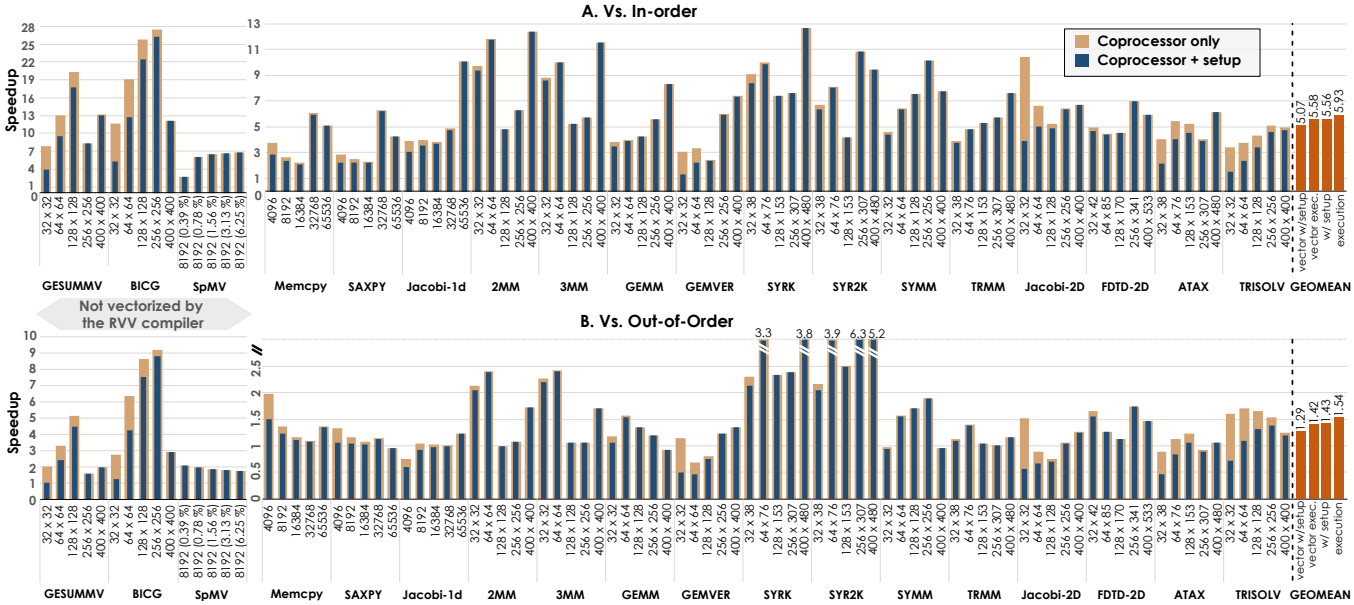


Figure 9: Speedup obtained by compiler-generated code running on the coprocessor against (A) a comparable in-order reference and (B) an OoO CPU model based on a large superscalar core (AMD Zen 5). Both references have support for RVV [49]. Different datasets are considered, and kernels not vectorized by the RVV compiler are identified. For the *SpMV*, it is considered a 8192x8192 matrix with different density values. The rightmost orange bars report the geometric mean values for four scenarios (from left to right): (i) vectorized benchmarks including coprocessor setup, (ii) vectorized benchmarks only considering execution, (iii) all benchmarks including coprocessor setup, and (iv) all benchmarks only considering execution. The setup corresponds to the time taken to configure the coprocessor, according to the steps 1, 2, and 3 of Fig. 8.

Table 2: Reference CPUs used in the comparison with the implemented evaluation prototype. The in-order core is parameterized to represent a fair comparison with the evaluation prototype, while the large OoO superscalar model is based on an AMD Zen 5, with information taken from [36].

REF. #1: In-order	**REF. #2: out-of-order (Zen5)**
In-order CPU @ 2GHz: 4-wide fetch/issue/commit 10 IQ, 4 LSQ, 10 SB 32 Int RF, 32 FP RF, 32×512-bit VRF	**Out-of-order CPU @ 2GHz:** 8-wide fetch & commit 12-wide issue/disp./writeback 630 IQ, 630 LQ, 100 SQ, 630 ROB entries 128 Int RF, 192 FP RF, 384×512-bit VRF
Functional Units: 1×Scalar LSUs, 1×Int ALU 1×Vector LSUs, 1×Vector ALUs/FPUs	**Functional Units:** 2×Scalar LSUs, 1×Int ALU 8×Vector LSUs, 2×Vector ALUs/FPUs
L1-D: 32 KB, 4-way, 4-cycle latency, 8 MSHRs, 20 targets/MSHR	
L2: 1 MB, 16-way, 10-cycle latency, 32 MSHRs, 12 targets/MSHR	
DRAM: Dual-Channel DDR3-1600 8×8	

with the dataset size, as the (fixed) setup overhead at the coprocessor is amortized (steps 1, 2, and 3 of Fig. 8). Naturally, for small problem sizes, the execution time is more influenced by the setup phase, limiting gains. As the working set grows and the kernel phase becomes dominant, the stream-vector code achieves substantial improvements, outperforming the baseline across all configurations.

When considering only the vectorized configurations, the coprocessor achieves 5.07× average speedup when including the setup phase, and 5.58× when considering only the kernel execution. Across all the configurations (including non-vectorized kernels by the RVV compiler), the coprocessor achieves 5.56× average speedup when including the setup phase, and 5.93× when considering only the kernel execution. Notably, the latter (5.93× improvement) corresponds to the expected performance if UVE were directly integrated into the core. The largest gains from vectorized kernels are observed in *2MM*, *3MM*, *SYRK*, and *SYR2K* with speedups of up to 12.4×, 11.5×, 11.5×, and 10.8×, respectively. Naturally, kernels that the reference RVV compiler fails to vectorize (i.e., *GESUMMV*, *BICG*, and *SpMV*) exhibit even greater speedups, reaching up to 27.4×.

Hence, these results demonstrate that the data streaming code generated by the proposed compiler effectively promotes the overlap of computations with memory accesses, delivering consistent improvements relative to a comparable in-order reference.

5.3 Comparison with an 8-Wide OoO Core

Compared with the OoO reference, modeled using publicly available AMD Zen 5 specifications, the generated code remains highly competitive (Fig. 9.B). Across all benchmarks, the coprocessor achieves average speedups of 1.43× with the setup phase and 1.54× for kernel execution only. Restricting the analysis to kernels vectorized by both the proposed toolchain and the RVV compiler, the speedup is 1.29× (with setup) and 1.42× (kernel-only). Consistent with the in-order comparison, the largest gains among commonly vectorized

kernels occur in *2MM*, *3MM*, *SYRK*, and *SYR2K*, reaching up to 6.3×. Even greater improvements are observed in kernels that the reference RVV compiler fails to vectorize, with speedups of up to 9.2× in *BICG*. In a few rare cases, slowdowns occur due to inefficiencies in UVE when handling scalar regions outside vectorized loops, as well as the higher compute throughput of the OoO core.

These results indicate that the stream-specialized code generated by the proposed compiler is very efficient, highlighting the effectiveness of the approach. Despite relying on a significantly simpler single-issue and in-order microarchitecture, the coprocessor performance matches or surpasses the OoO reference in most cases. This advantage stems from decoupling memory access from the core through data streaming, reducing memory-related stalls without relying on the complexity of speculative execution, dynamic scheduling, or register renaming typical of OoO designs.

6 Related Work

The proposed compilation flow tackles the following research topics: *i)* data streaming; *ii)* compilation for stream specialization; and *iii)* vectorization. The following paragraphs provide a review of related works in these topics, while framing the proposed toolchain.

Data streaming: To circumvent limitations imposed by traditional caches, data streaming schemes [6] were first developed to transfer large contiguous data blocks directly between global memory and hardware accelerators [5, 8, 17]. Over time, more sophisticated schemes were introduced to describe multi-dimensional patterns [28, 33, 39], handle indirection [30], and data dependencies [7]. Recently, new solutions started deploying such data streaming mechanisms in GPPs, introducing the first concepts of stream specialization. The first attempt targeted an x86 architecture [46]. After that, and by following the same trend, the SSR RISC-V extension [40] was proposed. Several RISC-V ISA extensions followed, such as UVE [9], which combines data streaming and vectorization to solve the shortcomings of VLA extensions. ISSR [37] and SSSR [38] extended SSR [40] to support indirection and sparse computations. Meanwhile, alternative approaches emerged within this paradigm, such as stream-based prefetching [29, 42, 48], and near-data processing [45, 47]. Importantly, S2VEC could be adapted to support these solutions through back-end extensions, even without vectorization.

Compilation for stream specialization: Most of the mentioned solutions provide some compiler support, typically in the middle-end [27, 42, 46, 50] or back-end [40]. However, details on the compiler implementation are often limited [40, 42, 46]. To the best of the authors' knowledge, there is a single compilation proof-of-concept work for stream specialization [27], and two for accelerators with data streaming [22, 50]. The previous UVE compiler [27] lacks support for several essential features (as seen in Section 4), including multiple and cross-loop dependencies, data stream description and compute graph optimization passes, data stream synchronization, and full nested-loop support. For accelerators, Weng et al. [50] describe only a stream extraction method, which is limited in both pattern and code support. It does not consider stream loops, multiple induction dependencies, loop collapsing, and compute graph merging. In contrast, Lopoukhine et al. [22] rely exclusively on high-level MLIR abstractions [20], whereas our approach operates within the conventional LLVM compilation pipeline, supporting standard C and, more generally, any language that lowers to LLVM IR and follows the same optimization flow.

For most stream specialization approaches, the complexity of the compilation process is tightly coupled with the complexity of the data streaming model, the supported patterns, and optimizations. In fact, most existing solutions have limited support for complex patterns (with dependencies on other induction variables) [37, 38, 40, 42], or rely on less disruptive models to maintain a simple compilation process [46]. Moreover, they are limited to pattern detection, not considering code optimizations, such as vectorization.

Vectorization: SIMD extensions have long been a primary mechanism for exploiting data-level parallelism in HPC workloads [12]. Recent architectures, including ARM SVE [41], RISC-V Vectors [49], and UVE [9], introduce VLA instruction sets that adapt vector width at runtime. On the compiler side, auto-vectorization techniques [10, 16, 24, 31, 54] – such as outer-loop vectorization [32] and Superword Level Parallelism (SLP) [4, 19] – have been widely studied and integrated into modern compilers. However, existing approaches often fail to expose vectorization opportunities, particularly for complex memory access patterns [2, 15, 24, 54]. The proposed compilation framework facilitates and extends auto-vectorization by leveraging the linearization and coalescing effect of stream specialization and stream-aware loop transformations.

7 Conclusions

This paper presented a novel compilation tool that leverages stream-based dataflow mechanisms. Developed within LLVM, it fills a critical gap in compiler support for stream-based specialization. Key innovations include a dedicated IR for stream-based approaches, new compiler passes to identify and encode memory access patterns into streams, and a complete code generation pipeline designed for stream-vector architectures. The proposed toolchain was evaluated on the latest UVE version, achieving code quality comparable to hand-optimized implementations while covering and vectorizing a broader range of loops than existing vector-length agnostic auto-vectorizers. Furthermore, preliminary results on a single-issue, in-order vector coprocessor with data streaming support demonstrate speedups of 5.9× over an equivalently configured in-order core and 1.5× over a wide OoO processor, both featuring RVV.

Acknowledgments

Work supported by national funds through Fundação para a Ciência e a Tecnologia, I.P. (FCT) under projects UID/50021/2025 (DOI: 10.54499/UID/50021/2025), UID/PRR/50021/2025 (DOI: 10.54499/UID/PRR/50021/2025), UID/50008/2025 (DOI: 10.54499/UID/50008/2025), 2022.06780.PTDC (DOI: 10.54499/2022.06780.PTDC), 2022.11626.BD (DOI: 10.54499/2022.11626.BD), and 2025.01517.BD.

Funded by the European Union under grant agreement 101189551 (DOI: 10.3030/101189551). Views and opinions expressed are however those of the author(s) only and do not necessarily reflect those of the European Union or the European Health and Digital Executive Agency. Neither the European Union nor the granting authority can be held responsible for them.

References

[1] Neil Adit and Adrian Sampson. 2022. Performance left on the table: An evaluation of compiler autovectorization for RISC-V. *IEEE Micro* 42, 5 (2022), 41–48.

[2] Hossein Amiri and Asadollah Shahbahrami. 2020. SIMD programming using Intel vector extensions. *J. Parallel and Distrib. Comput.* 135 (2020), 83–100.

[3] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. 2011. The gem5 simulator. *ACM SIGARCH computer architecture news* 39, 2 (2011), 1–7.

[4] Yishen Chen, Charith Mendis, and Saman Amarasinghe. 2022. All you need is superword-level parallelism: systematic control-flow vectorization with SLP. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation* (San Diego, CA, USA) *(PLDI 2022)*. Association for Computing Machinery, New York, NY, USA, 301–315. doi:10.1145/3519939.3523701

[5] Silviu Ciricescu, Ray Essick, Brian Lucas, Phil May, Kent Moat, Jim Norris, Michael Schuette, and Ali Saidi. 2003. The Reconfigurable Streaming Vector Processor (RSVPTM). In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, USA, 141.

[6] Luís Crespo, Nuno Neves, Pedro Tomás, and Nuno Roma. 2025. A Survey on Stream-Based Architectures: From Accelerators to CPUs. *Proc. IEEE* 113, 8 (2025), 713–751. doi:10.1109/JPROC.2025.3642972

[7] Vidushi Dadu, Jian Weng, Sihao Liu, and Tony Nowatzki. 2019. Towards General Purpose Acceleration by Exploiting Common Data-Dependence Forms. In *Proceedings of the 52nd Annual IEEE/ACM International Symposium on Microarchitecture* (Columbus, OH, USA) *(MICRO-52)*. Association for Computing Machinery, New York, NY, USA, 924–939. doi:10.1145/3352460.3358276

[8] William J. Dally, Francois Labonte, Abhishek Das, Patrick Hanrahan, Jung-Ho Ahn, Jayanth Gummaraju, Mattan Erez, Nuwan Jayasena, Ian Buck, Timothy J. Knight, and Ujval J. Kapasi. 2003. Merrimac: Supercomputing with Streams. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing* (Phoenix, AZ, USA) *(SC '03)*. Association for Computing Machinery, New York, NY, USA, 35. doi:10.1145/1048935.1050187

[9] Joao Mario Domingos, Nuno Neves, Nuno Roma, and Pedro Tomás. 2021. Unlimited vector extension with data streaming support. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 209–222. doi:10.1109/ISCA52012.2021.00025

[10] Alexandre E Eichenberger, Peng Wu, and Kevin O'brien. 2004. Vectorization for SIMD architectures with alignment constraints. *Acm sigplan notices* 39, 6 (2004), 82–93.

[11] Ana Fernandes, Luís Crespo, Nuno Neves, Pedro Tomás, Nuno Roma, and Gabriel Falcao. 2025. Functional Validation of the RISC-V Unlimited Vector Extension. *IEEE Embedded Systems Letters* 17, 1 (2025), 2–5. doi:10.1109/LES.2024.3416820

[12] Simon Hammond, Courtenay Vaughan, and Clay Hughes. 2018. Evaluating the Intel Skylake Xeon processor for HPC workloads. In *2018 International Conference on High Performance Computing & Simulation (HPCS)*. IEEE, 342–349.

[13] John L Hennessy and David A Patterson. 2019. A new golden age for computer architecture. *Commun. ACM* 62, 2 (2019), 48–60.

[14] Mark Horowitz. 2014. 1.1 computing's energy problem (and what we can do about it). In *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. IEEE, 10–14.

[15] Vijay Kandiah, Daniel Lustig, Oreste Villa, David Nellans, and Nikos Hardavellas. 2023. Parsimony: Enabling SIMD/Vector Programming in Standard Compiler Flows. In *Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization* (Montréal, QC, Canada) *(CGO '23)*. Association for Computing Machinery, New York, NY, USA, 186–198. doi:10.1145/3579990.3580019

[16] Ralf Karrenberg and Sebastian Hack. 2011. Whole-function vectorization. In *Proceedings of the 9th Annual IEEE/ACM International Symposium on Code Generation and Optimization (CGO '11)*. IEEE Computer Society, USA, 141–150.

[17] Brucek Khailany, William J Dally, Ujval J Kapasi, Peter Mattson, Jinyung Namkoong, John D Owens, Brian Towles, Andrew Chang, and Scott Rixner. 2001. Imagine: Media processing with streams. *IEEE micro* 21, 2 (2001), 35–46.

[18] Seonggun Kim and Hwansoo Han. 2012. Efficient SIMD code generation for irregular kernels. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (New Orleans, Louisiana, USA) *(PPoPP '12)*. Association for Computing Machinery, New York, NY, USA, 55–64. doi:10.1145/2145816.2145824

[19] Samuel Larsen and Saman Amarasinghe. 2000. Exploiting superword level parallelism with multimedia instruction sets. *Acm Sigplan Notices* 35, 5 (2000), 145–156.

[20] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: scaling compiler infrastructure for domain specific computation. In *Proceedings of the 2021 IEEE/ACM International Symposium on Code Generation and Optimization* (Virtual Event, Republic of Korea) *(CGO '21)*. IEEE Press, 2–14. doi:10.1109/CGO51591.2021.9370308

[21] Charles E Leiserson, Neil C Thompson, Joel S Emer, Bradley C Kuszmaul, Butler W Lampson, Daniel Sanchez, and Tao B Schardl. 2020. There's plenty of room at the Top: What will drive computer performance after Moore's law? *Science* 368, 6495 (2020), eaam9744.

[22] Alexandre Lopoukhine, Federico Ficarelli, Christos Vasiladiotis, Anton Lydike, Josse Van Delm, Alban Dutilleul, Luca Benini, Marian Verhelst, and Tobias Grosser. 2025. A Multi-level Compiler Backend for Accelerated Micro-kernels Targeting RISC-V ISA Extensions. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 163–178. doi:10.1145/3696443.3708952

[23] Louis-Noel Pouchet. 2012. PolyBench: The polyhedral benchmark suite. https://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[24] Saeed Maleki, Yaoqing Gao, Maria J. Garzarán, Tommy Wong, and David A. Padua. 2011. An Evaluation of Vectorizing Compilers. In *Proceedings of the 2011 International Conference on Parallel Architectures and Compilation Techniques (PACT '11)*. IEEE Computer Society, USA, 372–382. doi:10.1109/PACT.2011.68

[25] Francesco Minervini, Oscar Palomar, Osman Unsal, Enrico Reggiani, Josue Quiroga, Joan Marimon, Carlos Rojas, Roger Figueras, Abraham Ruiz, Alberto Gonzalez, Jonnatan Mendoza, Ivan Vargas, César Hernandez, Joan Cabre, Lina Khoirunisya, Mustapha Bouhali, Julian Pavon, Francesc Moll, Mauro Olivieri, Mario Kovac, Mate Kovac, Leon Dragic, Mateo Valero, and Adrian Cristal. 2023. Vitruvius+: An Area-Efficient RISC-V Decoupled Vector Coprocessor for High Performance Computing Applications. *ACM Trans. Archit. Code Optim.* 20, 2, Article 28 (March 2023), 25 pages.

[26] Sourena Naser Moghaddasi, Haris Smajlović, Ariya Shajii, and Ibrahim Numanagić. 2025. Vectron: A Dynamic Programming Auto-vectorization Framework. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 644–659. doi:10.1145/3696443.3708963

[27] Nuno Neves, Joao Mario Domingos, Nuno Roma, Pedro Tomás, and Gabriel Falcao. 2022. Compiling for Vector Extensions With Stream-Based Specialization. *IEEE Micro* 42, 5 (Sept. 2022), 49–58. doi:10.1109/MM.2022.3173405

[28] Nuno Neves, Pedro Tomás, and Nuno Roma. 2017. Adaptive in-cache streaming for efficient data management. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 25, 7 (2017), 2130–2143.

[29] Nuno Neves, Pedro Tomás, and Nuno Roma. 2020. Compiler-assisted data streaming for regular code structures. *IEEE Trans. Comput.* 70, 3 (2020), 483–494.

[30] Tony Nowatzki, Vinay Gangadhar, Newsha Ardalani, and Karthikeyan Sankaralingam. 2017. Stream-Dataflow Acceleration. In *Proceedings of the 44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA '17)*. Association for Computing Machinery, New York, NY, USA, 416–429. doi:10.1145/3079856.3080255

[31] Dorit Nuzman and Richard Henderson. 2006. Multi-platform Auto-vectorization. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO '06)*. IEEE Computer Society, USA, 281–294. doi:10.1109/CGO.2006.25

[32] Dorit Nuzman and Ayal Zaks. 2008. Outer-loop vectorization: revisited for short SIMD architectures. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques* (Toronto, Ontario, Canada) *(PACT '08)*. Association for Computing Machinery, New York, NY, USA, 2–11. doi:10.1145/1454115.1454119

[33] Sérgio Paiágua, Frederico Pratas, Pedro Tomás, Nuno Roma, and Ricardo Chaves. 2013. Hotstream: Efficient data streaming of complex patterns to multiple accelerating kernels. In *2013 25th International Symposium on Computer Architecture and High Performance Computing*. IEEE, 17–24.

[34] Constantine D Polychronopoulos. 1987. *Loop coalescing: A compiler transformation for parallel machines.* Technical Report. Illinois Univ., Urbana (USA).

[35] Brandon Reagen, Robert Adolf, Yakun Sophia Shao, Gu-Yeon Wei, and David Brooks. 2014. Machsuite: Benchmarks for accelerator design and customized architectures. In *2014 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 110–119.

[36] Tiago B Rodrigues, Alexandre Rodrigues, Manuel Goulão, Pedro Tomás, and Leonel Sousa. 2025. Accelerating NTT with RISC-V Vector Extension for Fully Homomorphic Encryption. *IACR Transactions on Cryptographic Hardware and Embedded Systems* 2025, 4 (2025), 711–736.

[37] Paul Scheffler, Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. 2021. Indirection stream semantic register architecture for efficient sparse-dense linear algebra. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 1787–1792.

[38] Paul Scheffler, Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. 2023. Sparse Stream Semantic Registers: A Lightweight ISA Extension Accelerating General Sparse Linear Algebra. *IEEE Trans. Parallel Distrib. Syst.* 34, 12 (Dec. 2023), 3147–3161. doi:10.1109/TPDS.2023.3322029

[39] Fabian Schuiki, Michael Schaffner, Frank K Gürkaynak, and Luca Benini. 2018. A scalable near-memory architecture for training deep neural networks on large in-memory datasets. *IEEE Trans. Comput.* 68, 4 (2018), 484–497.

[40] Fabian Schuiki, Florian Zaruba, Torsten Hoefler, and Luca Benini. 2020. Stream semantic registers: A lightweight RISC-V ISA extension achieving full compute

utilization in single-issue cores. *IEEE Trans. Comput.* 70, 2 (2020), 212–227.

[41] Nigel Stephens, Stuart Biles, Matthias Boettcher, Jacob Eapen, Mbou Eyole, Giacomo Gabrielli, Matt Horsnell, Grigorios Magklis, Alejandro Martinez, Nathanael Premillieu, et al. 2017. The ARM scalable vector extension. *IEEE micro* 37, 2 (2017), 26–39.

[42] Nishil Talati, Kyle May, Armand Behroozi, Yichen Yang, Kuba Kaszyk, Christos Vasiladiotis, Tarunesh Verma, Lu Li, Brandon Nguyen, Jiawen Sun, et al. 2021. Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 654–667.

[43] Jubi Taneja, Avery Laird, Cong Yan, Madan Musuvathi, and Shuvendu K. Lahiri. 2025. LLM-Vectorizer: LLM-Based Verified Loop Vectorizer. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization* (Las Vegas, NV, USA) *(CGO '25)*. Association for Computing Machinery, New York, NY, USA, 137–149. doi:10.1145/3696443.3708929

[44] João Vieira, Nuno Roma, Gabriel Falcao, and Pedro Tomás. 2024. Ndpmulator: Enabling full-system simulation for near-data accelerators from caches to dram. *IEEE Access* 12 (2024), 10349–10365.

[45] Zhengrong Wang, Christopher Liu, Aman Arora, Lizy John, and Tony Nowatzki. 2023. Infinity Stream: Portable and Programmer-Friendly In-/Near-Memory Fusion. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3* (Vancouver, BC, Canada) *(ASPLOS 2023)*. Association for Computing Machinery, New York, NY, USA, 359–375. doi:10.1145/3582016.3582032

[46] Zhengrong Wang and Tony Nowatzki. 2019. Stream-based memory access specialization for general purpose processors. In *2019 ACM/IEEE 46th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 736–749.

[47] Zhengrong Wang, Jian Weng, Sihao Liu, and Tony Nowatzki. 2022. Near-Stream Computing: General and Transparent Near-Cache Acceleration. In *2022 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 331–345.

[48] Zhengrong Wang, Jian Weng, Jason Lowe-Power, Jayesh Gaur, and Tony Nowatzki. 2021. Stream floating: Enabling proactive and decentralized cache optimizations. In *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*. IEEE, 640–653.

[49] Andrew Waterman and Krste Asanovic. 2019. RISC-V "V" Vector Extension. https://github.com/riscv/riscv-v-spec.

[50] Jian Weng, Sihao Liu, Dylan Kupsh, and Tony Nowatzki. 2022. Unifying spatial accelerator compilation with idiomatic and modular transformations. *IEEE Micro* 42, 5 (2022), 59–69.

[51] Wm A Wulf and Sally A McKee. 1995. Hitting the memory wall: Implications of the obvious. *ACM SIGARCH computer architecture news* 23, 1 (1995), 20–24.

[52] Yifan Yang, Joel S. Emer, and Daniel Sanchez. 2021. SpZip: architectural support for effective data compression in irregular applications. In *Proceedings of the 48th Annual International Symposium on Computer Architecture* (Virtual Event, Spain) *(ISCA '21)*. IEEE Press, 1069–1082. doi:10.1109/ISCA52012.2021.00087

[53] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. 2020. Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads. *IEEE Trans. Comput.* 70, 11 (2020), 1845–1860.

[54] Cyrus Zhou, Zack Hassman, Dhirpal Shah, Vaughn Richard, and Yanjing Li. 2024. YFlows: Systematic Dataflow Exploration and Code Generation for Efficient Neural Network Inference using SIMD Architectures on CPUs. In *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction* (Edinburgh, United Kingdom) *(CC 2024)*. Association for Computing Machinery, New York, NY, USA, 212–226. doi:10.1145/3640537.3641566