Unlimited Vector Extension with Data Streaming Support

Joao Mario Domingos
INESC-ID
Instituto Superior Técnico
Universidade de Lisboa
Lisboa, Portugal
joao.mario@tecnico.ulisboa.pt

Nuno Neves
INESC-ID
Instituto de Telecomunicações
Lisboa, Portugal
nuno.neves@inesc-id.pt

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

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

***Abstract*—Unlimited vector extension (UVE) is a novel instruction set architecture extension that takes streaming and SIMD processing together into the modern computing scenario. It aims to overcome the shortcomings of state-of-the-art scalable vector extensions by adding data streaming as a way to simultaneously reduce the overheads associated with loop control and memory access indexing, as well as with memory access latency. This is achieved through a new set of instructions that pre-configure the loop memory access patterns. These attain accurate and timely data prefetching on predictable access patterns, such as in multidimensional arrays or in indirect memory access patterns. Each of the configured data streams is associated to a general-purpose vector register, which is then used to interface with the streams. In particular, iterating over a given stream is simply achieved by reading/writing to the corresponding input/output stream, as the data is instantly consumed/produced. To evaluate the proposed UVE, a proof-of-concept gem5 implementation was integrated in an out-of-order processor model, based on the ARM Cortex-A76, thus taking into consideration the typical speculative and out-of-order execution paradigms found in high-performance computing processors. The evaluation was carried out with a set of representative kernels, by assessing the number of executed instructions, its impact on the memory bus and its overall performance. Compared to other state-of-the-art solutions, such as the upcoming ARM Scalable Vector Extension (SVE), the obtained results show that the proposed extension attains average performance speedups over 2.4× for the same processor configuration, including vector length.**

***Index Terms*—Scalable SIMD/Vector Processing, Data Streaming, Instruction Set Extension, Processor Microarchitecture, High-Performance Computing.**

I. Introduction

Single Instruction Multiple Data (SIMD) instruction set extensions potentiate the exploitation of Data-Level Parallelism (DLP) to provide significant performance speedups [1]–[6]. However, most conventional SIMD extensions have been focused on operating with fixed-size registers (e.g., Intel MMX, SSE, AVX, etc, or ARM NEON), as it simplifies the implementation and limits the hardware (HW) requirements. This poses a non-trivial question regarding the vector length [2], [7], since its optimal size often depends on the target workload. Moreover, any modification to the register length usually requires the adoption of newer instruction-set extensions, which inevitably makes previously compiled code obsolete [7].

To overcome such issues, new solutions have recently emerged. In particular, ARM SVE [7] and RISC-V Vector extension (RVV) [8] are agnostic to the physical vector register size from the software (SW) developer/compiler point of view, since it is only known at runtime. Hence, different processor implementations can adopt distinct vector sizes, while requiring no code modifications. For example, High-Performance Computing (HPC) processors can make use of large vectors to attain a high throughput, while low-power processors can adopt smaller vectors to fulfill power and resource constraints. However, this usually forces the presence of predicate [9] and/or vector control instructions to disable vector elements outside loop bounds, eventually increasing the number of loop instructions [2]. Fig. 1 illustrates this problem in the `saxpy` kernel implementations on SVE [7] and RVV [8]. As it can be observed, both SVE and RVV require a large instruction overhead (shaded instructions in Figs. 1.B and 1.C), resulting from memory indexing, loop control and even memory access, as neither of these directly contribute to maximize the data processing throughput. These overhead instructions often represent the majority of the loop code, wasting processor resources and significantly impacting performance.

On the other hand, the performance of data-parallel applications is also often constrained by the memory hierarchy. Hence, most high-performance processors rely on SW and/or HW prefetchers to improve performance [10]–[24]. However, SW prefetching solutions [16], [17] typically impose additional instructions in the loop code, increasing overheads. HW prefetchers have been improving their accuracy and coverage with each new generation. However, their timeliness is dependent on the used mechanisms [21]–[24] to identify memory access patterns and predict future accesses, while prediction inaccuracies increase energy consumption and pollute caches.

In accordance, the new Unlimited Vector Extension (UVE) that is now proposed distinguishes from common SIMD extensions by the following features:

F1 **Decoupled memory accesses**. By relying on a stream-based paradigm, input data is directly streamed to the register file, effectively decoupling memory accesses from computation, and allowing data load/store to occur in parallel with data manipulation. This implicit stream prefetching facilitates the

978-1-6654-3333-4/21/$31.00 ©2021 IEEE
DOI 10.1109/ISCA52012.2021.00025

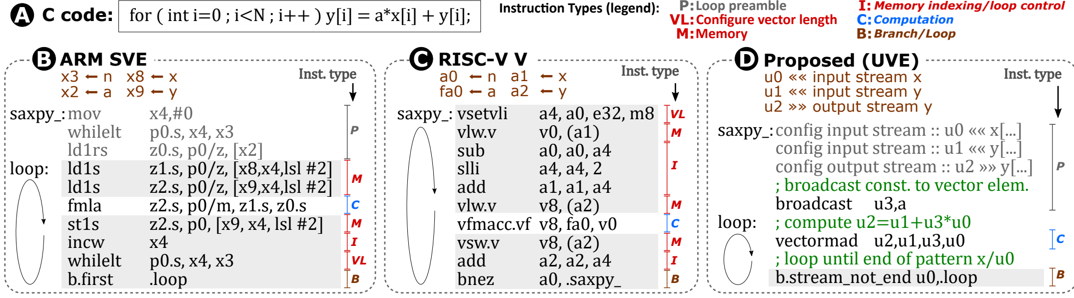


Fig. 1. Saxpy kernel implementation on ARM SVE, RISC-V V, and the proposed UVE. The RISC-V V and ARM SVE code is taken from official documentation. The instructions highlighted in grey represent loop overhead.

acquisition of data, significantly reducing the memory access latency and decreasing the time that memory-dependent instructions remain in the pipeline. This also reduces potential hazards at the rename and commit stages of out-of-order processors, improving performance.

F2 Indexing-free loops. By describing memory access patterns (loads/stores) at the loop preamble, it minimizes the number of instructions dedicated to memory indexing, significantly reducing the instruction pressure on the processor pipeline (see the UVE `saxpy` implementation in Fig. 1.D). Furthermore, by relying on exact representations of memory access patterns (descriptors), it avoids erroneous transfers of data due to miss-predicted prefetching.

F3 Simplified vectorization. Since the loop access patterns are exactly described by descriptor representations, supporting complex, multi-dimensional, strided, and indirect patterns, the *Streaming Engine* is able to transparently perform all scatter-gather operations, transforming non-coalesced accesses into linear patterns. Hence, from the execution pipeline viewpoint, data is always sequentially aligned in memory, simplifying vectorization. Fig. 2 illustrates this feature with the computation of the maximum function across the rows of three input patterns: (A) full matrix; (B) lower triangular matrix; and (C) indirect (matrix-like) access. Since the operations are the same (the differences concern only the read pattern), the UVE code is exactly the same for all patterns (*for each row, compute the maximum, regardless of the row size and pattern*).

F4 Implicit load/store. Since all streams are described at the loop preamble, not only can one remove indexing instructions, but also all explicit loads and stores, by simply associating each active stream with a different vector register. Hence, reading/writing from a register associated to an input/output (load/store) stream, implicitly triggers a stream iteration, requiring no additional stepping instruction – see Fig. 1.D. Naturally, reading multiple times from the same element of the stream is still possible, by either copying the value to a different register, or by inserting a specific instruction in the code to temporarily suspend the stream iteration process.

F5 Register-size agnostic. Just as in SVE and RVV, the UVE code is agnostic to the register size. However, to prevent operating over out-of-bound elements (e.g., when the number of elements to process is not a multiple of the vector length), SVE and RVV explicitly require a set of control instructions to define which vector elements are active. In contrast, UVE does not generally require such instructions (even though they are also included in the instruction-set specification). This is achieved because the *Streaming Engine* automatically disables all vector elements that fall out of bounds, which is equivalent to an automated padding of all streams to a multiple of the vector length. As a consequence, the loops become simpler and require a minimal set of control instructions, which in most cases corresponds to a single branch instruction.

To understand the impact of the proposed UVE extension on modern out-of-order general-purpose processors, a proof-of-concept base implementation is also presented. When compared to other processor architectures supporting SIMD extensions, it includes only minor modifications in the rename and commit stages, besides introducing a *Streaming Engine* to manage all stream operations. Notwithstanding, such minor modifications allow the exploitation of a set of architectural opportunities:

A1 Reduced load-to-use latency. At the architectural level, the proposed solution leads to a mitigation of the load-to-use

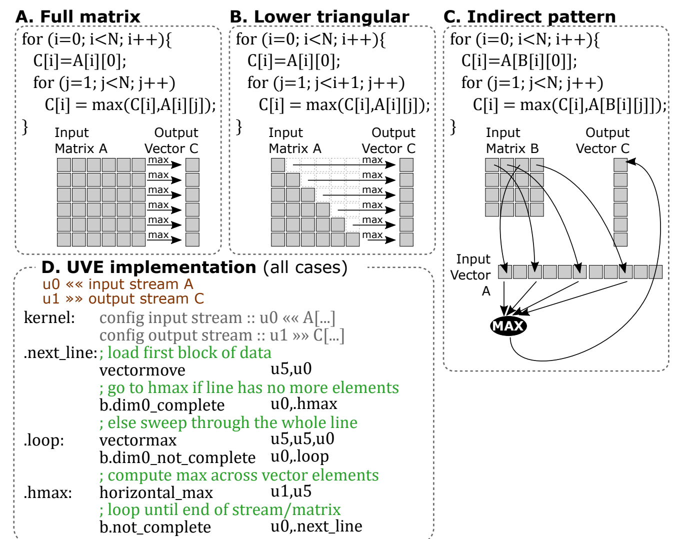


Fig. 2. UVE pseudo-code implementation for the computation of the maximum across the rows on three possible matrices: (A) full matrix, (B) lower triangular matrix, and (C) full matrix with pointers to an array.

latency, by directly streaming data into vector registers. This contrasts with conventional prefetching structures, where input data is first loaded to either a cache, or cache-level buffers.

A2 **Data streaming across virtual page boundaries.** As the proposed solution relies on an accurate representation of access patterns, it can safely cross page boundaries and continue streaming, even if the access pattern covers multiple pages.

A3 **Data re-use across miss-predictions.** To benefit performance, the proposed streaming implementation provides full support for speculative execution. Since the stream access pattern is exactly described, the processor keeps all speculatively consumed data on load buffers until the consuming instruction commits. Hence, in the event of a miss-speculation, data can be immediately reused (after correction of the execution path) without duplicated memory loads.

Finally, when compared with ARM SVE, and considering similar processor configurations and equal vector lengths, this cumulative set of contributions results in an average 2.4× performance speed-up on a representative set of benchmarks.

II. Data Streaming

In this work, a stream is defined as a predicable n-dimensional sequence of data that is transferred between the memory and the processor. Hence, the adopted model for stream representation must fulfill the following properties:

P1 **Generality:** It must be able to represent any loop-based affine or indirect memory address sequence, with complex multi-dimensional patterns supported by multi-level constructs.

P2 **HW feasibility:** The representation must adopt an encoding scheme that can be easily and efficiently interpreted by dedicated hardware and in parallel with data manipulation.

P3 **Decoupled memory indexing:** Memory indexing should be performed outside the computation pipeline, without requiring explicit instructions in the computational loop.

P4 **Autonomy:** It should allow the stream infrastructure to be able to autonomously move data between the processor and the memory, without the processor intervention.

A. Memory access modeling

The proposed model aims at describing the exact sequence of addresses that comprise each access to array-based variables. To make it compiler-friendly, it follows the typical structure of nested for loops, converting nested loop-based indexing and loop- or data-dependent (indirect) index dynamic ranges into an n-dimensional affine function:

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

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}).

From this base model, further complexity can be achieved by combining multiple functions. This is achieved by assigning the base address and/or the offset value of a function to the result of another affine function. Additionally, the data that is obtained by the sequence of addresses generated by an affine function can also be used to perform the same associations, resulting in the description of indirect memory access patterns.

B. Stream descriptor representation

The stream model defined in (1) is represented by an hierarchical descriptor-based representation that encodes the variables of each dimension of the affine function in a hardware-friendly scheme. It also provides mechanisms to combine multiple functions to allow the representation of complex and/or indirect access patterns.

1) Base stream descriptors: Each uni-dimensional access pattern is represented by a three-parameter tuple {O, E, S}, corresponding to the offset (O_k), size (E_k), and stride (S_k) variables of a single dimension (k), as represented in (1).

Fig. 3.B1 depicts the simplest pattern supported by this description, based on a linear memory access pattern starting at memory position A (offset) and ending at memory position $A+N$ (size N), with an element spacing (stride) of 1.

To encode the full extent of an n-dimensional access pattern, multiple descriptors are hierarchically combined in a linearly cascaded scheme (as in a nested loop), where the descriptor corresponding to dimension $k \in [0, n[$, is used to calculate a displacement that is added to the offset of the descriptor corresponding to the dimension $k-1$ (see Fig. 3.A1).

Two examples of 2-D stream descriptions are illustrated in Figs. 3.B2 and B3. Both cases represent a matrix access, where the first dimension encodes an horizontal row scan pattern and the second dimension iterates vertically through rows. In the particular case of the pattern in Fig. 3.B2, the first dimension generates a linear pattern (with stride 1) and the second dimension iterates over each consecutive row (with stride equal to the row size – Nc). Conversely, for the pattern in Fig. 3.B3, the stride parameters of both dimensions are used to skip memory elements and generate a scattered pattern.

2) Static descriptor modifiers: In some access patterns, the loop conditions of an inner for-loop may be generated by the iteration of an outer loop. As an example, for each outer loop (i) iteration of the lower triangular pattern depicted in Fig. 3.B4, the parameter size of the inner loop (j) should be incremented. As a result, the size (E) parameter of the descriptor corresponding to the inner loop is updated in every iteration of the descriptor corresponding to the outer loop.

To represent this behaviour, an optional static descriptor modifier is introduced, represented by the tuple {T, B, D, E}, where each parameter encodes the following information:

- *Target* (T): the tuple parameter to modify.

- *Behavior* (B): the modification operator (`add` - addition or `sub` - subtraction).
- *Displacement* (D): the constant value that is applied to the target descriptor parameter, according to B.
- *Size* (E): the total number of iterations that the modification is applied.

Hence, each modifier implicitly adds/subtracts the displacement value to the target parameter, each time the corresponding descriptor is iterated.

The association of a static modifier to a descriptor corresponding to dimension k is realized by pairing it with the descriptor that corresponds to dimension $k+1$, as depicted in Fig. 3.A2. As a result, all descriptors (dimensional or modifiers) that affect dimension k are bound to dimension $k+1$.

3) Indirect descriptor modifiers: The proposed descriptor specification can be further extended by allowing the contents of one stream to modify the values of another stream, making it possible to create indirect (see Fig. 3.B5) and indexed scatter-gather patterns. This interaction is realized with an optional indirect descriptor modifier, represented by the tuple {T, B, P}. In this descriptor, the displacement parameter is replaced by a pointer to the origin data stream (P). For each iteration, a value (dynamic displacement) is loaded from the origin stream and used to modify the target parameter (T). The indirection relation between the stream and the origin stream makes the size of the first dependent on the size of the second. Hence, the indirection modifier does not require any size parameter. The behavior (B) parameter supports the following operators:

- `set-add`: adds the dynamic displacement to the target.
- `set-sub`: subtracts the displacement from the target.
- `set-value`: sets the value given by the origin stream to the target value.

Contrasting with the static modifier behaviour operators (see Section II-B2), the indirect modifier does not perform any implicit addition/subtraction of the displacement to the target parameter. Instead, in each iteration, the target parameter is either set to the addition/subtraction of the displacement with the original value of the parameter, or solely set to the displacement value.

Finally, the syntax of an indirect descriptor modifier is equivalent to that of static modifiers, as depicted in Fig. 3.A3.

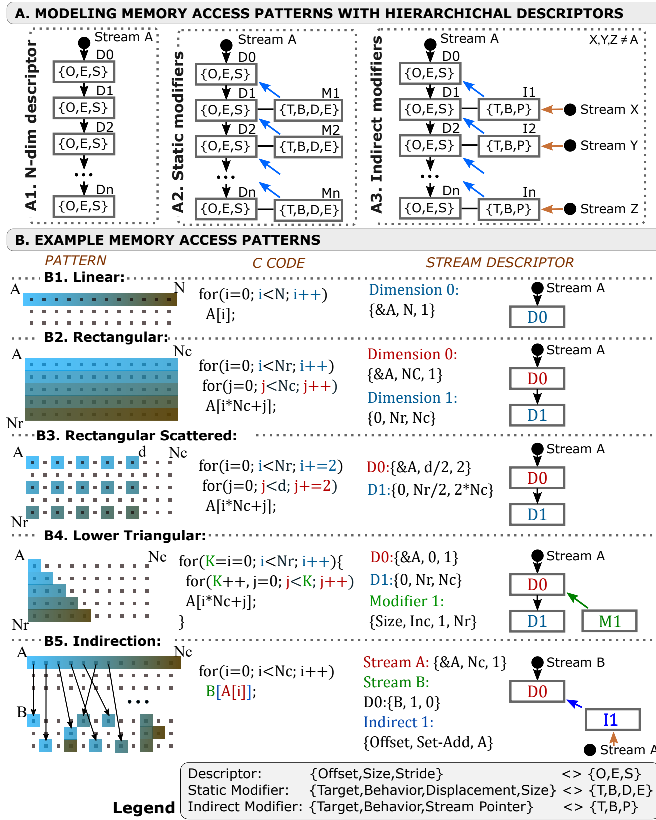


Fig. 3. Memory access pattern representation with hierarchical descriptors and examples cases.

III. Proposed UVE Extension

To fully exploit the data streaming specification presented in Section II, it is first necessary to provide a programming interface at the Instruction Set Architecture (ISA) level. This section introduces the Unlimited Vector Extension (UVE), which was designed by considering the following principles:

- **RISC style**: it must comprehend hardware-friendly instructions, allowing for an efficient implementation and avoiding complex decoding paths (e.g., each instruction should correspond to a processor μOp);
- **Scalable**: its programming model should fully support scalable and non-constrained vector lengths;
- **Coherent**: it should inherit and keep the structural principles of the base ISA; in particular, RISC-V was selected because of its open source nature, and due to its simple, clean and extensible instruction set.

A. Extension Design

The definition of the proposed UVE comprises both the processor base architecture and the streaming infrastructure[1].

1) Architectural State: Besides the natural adoption of a vector register file, the proposed UVE provides a streaming interface to the vector instructions and a set of predicate registers to facilitate the execution control on each lane. Furthermore, some instructions will make use of scalar registers (e.g. Loads and Stores), mostly inheriting the RISC-V scalar register bank [25].

Vector Registers. The proposed UVE makes use of 32 vector registers (named from "u0" to "u31"), satisfying a compromise between operand encoding and register file pressure. While the vector-length is not limited to any maximum size, a minimum value is defined, and corresponds to the width of the supported elementary data types: byte (8-bits), half-word (16-bits), word (32-bits), and double-word (64-bits). Naturally, the maximum vector length must be a multiple of the minimum length, while

[1]Upon paper acceptance, the complete specification of UVE will be made available at the authors' GitHub, along with the evaluation source code.

the element width is independently configured for each vector register.

Streaming interface. Each data stream is implicitly associated with a specific vector register (”u0” to ”u31”). Hence, by reading (or writing) to such register, the instruction is transparently consuming from (or producing to) the corresponding stream.

Predicate Registers. The predicate register file is composed of 16 predicate registers (”p0” to ”p15”). However, only 8 (p0-p7) are used in regular memory and arithmetic instructions, where predicate register p0 is always hardwired to 1 (all valid lanes execute), eliminating the need to pre-configure the register in non-conditional loops. The remaining predicate (p8-p15) are used in the configuration of the first 8, or to allow for context saving. This balance was based on related work analysis of compiled and hand-optimized codes, having the benefit of mitigating the predicate register pressure [7], [26].

2) Streaming Support: The streaming model implicit to the proposed UVE was devised upon the following principles:

Scalability. The design principle adopted by UVE is focused on a runtime variable vector length. In particular, the consumption/production of streams automatically enforces its iteration, ensuring an automatic progression of loop iterations. Moreover, since reading/writing outside the stream elements is automatically disabled by the streaming infrastructure (as in a false predicate), loop control can be performed with only a basic set of stream-conditional branches (e.g. stream end, stream dimension end).

Destructive behavior. The consumption from a data-stream mapped to a register automatically iterates over the stream. Hence, if such data element is to be used again by the program, it should be saved in a register or memory position, as the already streamed data is not restorable. This design option eliminates the need for additional step-instructions in each loop, promoting code reduction (see also Fig. 4).

Compiler optimizations. Despite the straightforward benchmark kernels that were adopted in the presented experimental evaluation, the proposed extension supports the usual set of compiler optimization techniques, such as loop unrolling and software pipeling. Future work will consider the implementation of a compiler toolchain that will exploit convenient compiler techniques to *i)* identify linear combinations of loop induction variables (1) used to calculate the address sequence of streamable memory accesses; *ii)* eliminate memory indexing and loop control; and *iii)* apply vectorization techniques.

Complexity limitation. Although UVE supports complex memory access patterns, reasonable limits are defined to constrain the hardware resources. In particular, the presented analysis shows that most patterns have a dimensionality no greater than 4. Moreover, high-dimensional patterns (e.g., 5D+) can also be designed by forcing the outer loop(s) to reconfigure the access pattern at each new iteration, with no significant performance differences. Nonetheless, the current implementation supports up to 8 dimensions and 7 modifiers.

3) Streaming memory model: To potentiate the offered performance, the proposed extension is designed to allow run-ahead execution of input streams, resulting in an automatic pre-loading of input data, similarly to what would otherwise be achieved through loop unrolling or software register prefetching. Naturally, this imposes that the source memory locations of an input stream cannot be modified in runtime by either conventional memory stores or by an output stream, as the runtime aggressiveness of prefetching may generate read-after-write hazards. However, this does not represent a hard constraint in most practical situations, as streamed loops do not require the explicit use of conventional load/stores, and because the codes that would lead to such hazards are generally not vectorizable nor amenable to scalable vector extensions (e.g., as in infinite impulse response filters). Write-after-read and write-after-write dependencies are normally dealt and well handled by the proposed streaming model, as the loop code encodes the dependencies between memory operations, ensuring the support for in-place computations.

The adopted streaming model also assumes that the processor is responsible for the synchronization between input streams and pending store instructions (which precede the configuration of the input stream, but are delayed due to execution latencies). Similarly, when exiting a streamed loop, the processor is also responsible for ensuring the synchronization between an output stream and a following load instructions.

Finally, and by keeping the same premises that were considered in the RISC-V specification, a weak memory model is herein adopted, where synchronization between different hardware threads is achieved only through the use of explicit *fence* instructions.

B. Instruction Set

UVE currently features 26 integer, 15 floating-point and 19 memory (including streaming) major instructions, resulting in 450 instructions when including all variations.

Stream configuration. The descriptor/modifier configuration parameters described in Section II (e.g., width, behaviour, etc.) are encoded using stream configuration instructions (identified by the prefix `ss` in Fig. 4). Simple 1-D patterns are configured using a single `ss.ld`/`ss.st` instruction, while more complex data patterns require multiple instructions (one per dimension/modifier) using specific stream configuration instructions, namely for start (`ss.{ld|st}.sta`), append (`ss.app[.mod|.ind]`, with `mod`≡static modifier, `ind`≡indirect modifier) and end (`ss.end[.mod|.ind]`. Additionally, the starting configuration of each stream must also specify the data size (suffix `{b|h|w|d}` for byte, half-word, word, and double-word) – see example in Fig. 4.

Stream control. A subset of instructions are responsible for the control of streams (e.g. suspend (`ss.suspend`), resume (`ss.resume`), stop (`ss.stop`)). Their purpose is to allow the momentary freeing/restoring of vector registers, and to enable context switching, allowing the concurrent execution of multiple processes without interfering with the streams configuration. It is also possible to manually “force” a load/store from/to a suspended stream, providing precise stream control.

Predication. Individual lanes of an SIMD execution are controlled with a subset of predication and masking instructions

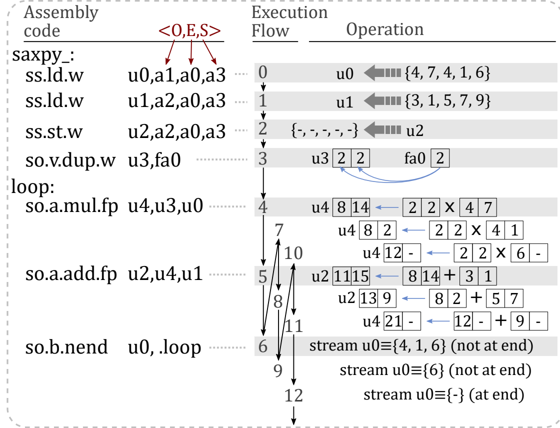


Fig. 4. UVE saxpy code emulation. The fused multiply-add operation cannot be used in this example because it only accepts 3 operands and a stream cannot simultaneously operate in both read and write modes.

that allow disabling the selected lanes. For such purpose, UVE provides instructions to configure the predicate based on vector register comparisons (e.g., less or equal to) and based on the valid elements of a vector register. Predicate manipulation instructions are also available, offering predicate moving, predicate width configuration, and element-wise negation.

Loop control. Control is assured with three distinct conditional branch formats: *i) predicate-based* - a condition (e.g., AND) is tested on a specified predicate register; *ii) end-of-stream* - a condition on the end of a stream, and *iii) end-of-dimension* - a condition on a stream dimension end. The *end-of-dimension* branch format allows a simple and fine control over the iterations of streams with different size. A particular use-case is when the accesses to the streamed variables are bound to different loops in the original scalar code (as in Fig. 2).

Vector manipulation. UVE also provides instructions to support vector manipulation and processing - e.g., vertical and horizontal arithmetic, logic and shift operations. All these instructions are optionally predicated by a predicate register. Fig. 4 illustrates the use of three vector manipulation and data processing instructions: *i) duplication* (`so.v.dup`), where a scalar is replicated to all elements of vector `u3`; and *ii) multiplication* (`so.a.mul.fp`) and *addition* (`so.a.add.fp`), where the processing of stream registers and vector registers is carried out in the same instruction, with automatic load/store, and no explicit register distinctions. Finally, it is worth noting that UVE is not completely dependent on streaming to transfer data to/from memory, as conventional (non-streaming) vector load/store instructions are kept in the ISA (`ss.load/store`). In particular, all these legacy instructions support post-fix address increment, not requiring explicit memory indexing for linear memory accesses.

Scalar processing. Although the proposed extension targets vector processing, the streaming infrastructure can also be exploited for scalar processing, through a specific set of vector-to-scalar and scalar-to-vector instructions. These impose an element-wise shift of the vector elements, forcing the consumption/production of one element in the stream.

Advanced control. UVE includes specialized instructions to explicitly read (`ss.getvl`) and configure the vector length (`ss.setvl`), allowing for narrower vector-length emulation and vector-length aligned processing. In particular, UVE supports any dimension of a stream description to be configured as vector-aligned, by automatic padding the vector registers when the dimension is not a multiple of the vector-length. To cope with different stream memory access footprints and temporal/spatial locality profiles of the streamed data, UVE supports cache-level access selection during stream configuration. As an example, the `so.cfg.mem`*x* instruction directs the corresponding stream to operate over the L*x* cache.

Concurrent streams. The UVE programming model assumes that concurrent streams operate independently, as the patterns do not directly encode dependencies across streams. However, this does not compromise general-purpose high-performance computing codes (particularly in-place computations), as the dependencies are described in the loop code. Such an example is the `saxpy` kernel illustrated in Fig. 1. Since the result stream `u2` depends on a read from stream `u1`, an ordering to the streaming process is imposed, avoiding potential hazards.

IV. Microarchitecture Support

Most streaming operations supported by UVE are performed within a *Streaming Engine*, embedded in the main architecture pipeline (see Section IV-B). Besides this new block, the CPU processing pipeline only needs to be extended with some minor streaming structures. Fig. 5 highlights such modifications, when applied on a traditional out-of-order processing pipeline. This same pipeline will be used in the experimental evaluation presented in Section VI.

Such modifications can be summarized as:

– **Decode, register file & execution units:** convenient support for the decoding of the proposed UVE instruction-set extension, vector registers, and corresponding logic,

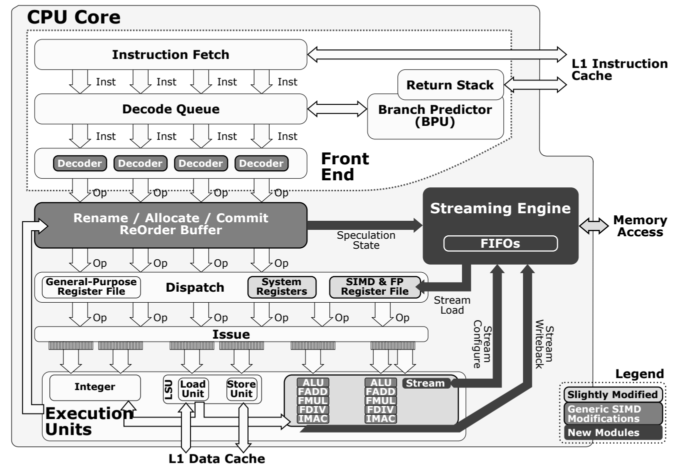


Fig. 5. Microarchitecture diagram, highlighting the introduced modifications. To alleviate the representation complexity, some load and store ports and coherency mechanisms are not represented.

arithmetic and branch functional units - a requirement similar to RVV [8] and SVE [7].
  - **Rename:** Besides the support for vector register renaming (common in most vector extensions), it should also support the renaming of streams. This allows for the speculative configuration of new streams while others (with the same logical naming) are still executing.
  - **Commit:** Support for the commit and squash of streams, by signaling the *Streaming Engine* all miss-speculation and commit events related to the processing of streams (namely, configuration, iteration, suspension/resuming and termination).

A. Stream Operation

To understand the architectural changes required to support the proposed streaming mechanisms, the following paragraphs describe the operation of streams, while the architecture of the *Streaming Engine* is described in Section IV-B.

Stream Configuration. Complex data patterns requires multiple instructions that, depending on the resolution order of data dependencies, may result in an out-of-order execution of the configuration instructions. Although the commit buffer could be used for reordering, it would prevent the speculative configuration of streams, impacting performance.

Hence, an alternative solution is herein adopted. Whenever a new stream configuration instruction reaches rename, it is registered on a stream configuration reordering structure (*Stream Configuration Reorder Buffer (SCROB)*, in Fig. 7), which is part of the *Streaming Engine*. This engine, similar to a re-order buffer, processes the configuration instructions in order, and as soon as the corresponding operands are available. Upon completion of the configuration, the streaming engine immediately starts processing the stream, either by pre-loading data (for input streams), or by calculating store addresses (on output streams), and waiting for the commit of store data.

To support speculative execution, the configuration of a stream generates two stream states (*speculative* and *commit*), which are dynamically iterated when an instruction that manipulates a stream reaches rename and commit stages, respectively.

Stream Renaming. The renaming mechanism is essential to allow the speculative configuration of streams. In particular, when configuring a stream, the corresponding identification register may still be associated with another running stream, due to miss-speculation and/or to pipeline latency. To avoid blocking the pipeline, a *Stream Alias Table (SAT)* (identical to a Register Alias Table (RAT)) was introduced, which performs the mapping between logical stream register and the physical *Streaming Engine* identifier. The *SAT* is also used to keep track of the registers currently associated with an active (i.e., not suspended) stream, allowing to verify whether an instruction operand corresponds to a read/write from a stream, or to a normal register operation.

Stream Iteration. A stream iteration process is logically performed by reading/writing from/to an input/output stream. Architecturally, this is achieved at rename, by signaling the *Streaming Engine* to iterate the speculative stream state. For an output stream, this also implies reserving space in the *Streaming Engine Store FIFO* (First-In First-Out) buffers (see Fig. 7 and Section IV-B) and then waiting for both data and commit signals to arrive, to complete the operation. On input streams, the devised solution attempts to minimize the load-to-use latency, by allocating the head of input streams on physical registers. As a consequence, when a stream consuming instruction reaches rename, the operand is immediately read and a new data element is pre-loaded to a different physical register. This is achieved by performing register renaming and requesting the *Streaming Engine* to load the corresponding data to the target physical register.

Stream Termination. The termination of a Stream is achieved at commit, either through an explicit termination instruction, or by committing an instruction that signals the completion of the streaming pattern. When such an event occurs, all the structures in the *Streaming Engine* are released, allowing the resources to be allocated to a new stream configuration.

Miss-Speculation. A possible miss-speculation may result in an erroneous stream configuration or iteration. On a miss-speculated configuration all associated structures are released, allowing the *Streaming Engine* to accept a new configuration. In contrast, on a miss-speculated iteration, when the corresponding instruction is squashed, two actions are performed: (i) the pipeline reverts the physical register state to the previously committed value; and (ii) the *Streaming Engine* is signalled to revert the speculated pointers on the load/store circular buffers to the current commit point. Hence, there is no direct impact on the buffered data (on loads) and addresses (on stores), as the streaming data access pattern is deterministic (it was simply consumed in the wrong way). Accordingly, all elements that were miss-speculatively read from an input stream remain valid and can immediately be re-used without requiring a new load from the cache hierarchy, saving time and energy. Naturally, committed stream reads and writes signal the streaming engine to advance the corresponding (commit) iterators, releasing the load and store buffers. Finally, stream termination (i.e., the release of all stream structures) is only performed at commit, hence not being affected by miss-speculation.

Cache Access. 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. 6). Although this may impose an additional pressure on the same channels and an eventual delay on concurrent read/writes from the scalar pipeline, the real impact of this option is usually very small, since streaming loops do not usually require scalar memory operations. As a consequence, stream and conventional load/store memory operations typically end up being performed in a mutually exclusive fashion (as it occurs in all benchmarks presented in Section V), mitigating the impact of such merge.

On the other hand, and as it was previously referred and illustrated in Fig. 6, UVE also supports direct data loading from any Lx cache level or from the main memory. This is particularly important, as caching data with low temporal

locality wastes cache capacity, imposes higher energy consumption, and hurts the performance. At the hardware level, and to avoid significant changes in the memory hierarchy infrastructure, this can be simply achieved by issuing the corresponding read requests as non-cacheable loads on all lower levels Ly, with $y < x$. For example, when streaming from the L2 cache (the considered default case), non-cacheable requests are first sent by the CPU to the L1. As they likely result in an L1 cache miss, they are forward to the L2, where they are treated as a normal (cacheable) L2 load. On the other hand, a stream configured for direct memory access will issue the corresponding input requests as non-cacheable at all levels (avoiding cache pollution), likely resulting in a direct memory load (as the requests will probably result in a miss at all cache levels).

In what concerns the output streams, the considered implementation forces stores to be issued to the L1 cache. However, a solution similar to the one that was adopted for the stream loads could also be considered, by simply forcing a write-no-allocate write-through policy on lower-level caches, attaining direct stream stores to any cache level or even to the main memory.

Memory Coherence. Coherence is ensured through two distinct mechanisms:

- On the core, stream and non-stream operations are kept coherent by matching the stream load/store state with the core load/store queues and by solving possible stream load/store dependencies through typical request delay, replay, or squash mechanisms . Hence, data written by the conventional pipeline can be immediately read by a newly configured input stream, and data written by an output stream can be loaded using a conventional load instruction. This ensures a reliable transition between sequential code and stream loops.
- On the caches, coherence is ensured through a conventional MOESI cache snooping protocol. Naturally, directory-based approaches could equally be used.

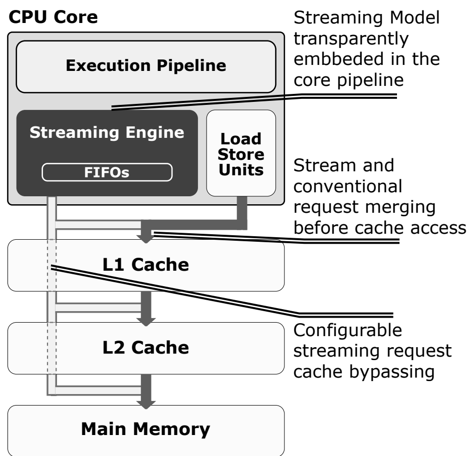


Fig. 6. System overview, depicting the Streaming Engine embedded in an out-of-order core and its connections to the memory hierarchy.

Finally, it should be noted that the proposed mechanism does not require any coherence mechanism at the level of the stream FIFOs, as it is assumed that preloaded data is already consumed by the core, similar to what would occur with loop unrolling and register pre-fetching (this will be further detailed in Section IV-B – *Load/Store FIFOs*). Naturally, a different solution could also be adopted by extending the coherence protocol to the (internal) stream buffers. However, this alternative solution was left to future work, since data-parallel kernels that are amenable to scalable vector extensions usually do not impose read-after-write dependencies on the same memory location. An exception to this case are some iterative algorithms (e.g., k-Means). However, such cases require stream reconfiguration at each algorithm iteration, imposing synchronization.

Exception Handling. The main source of UVE-specific exceptions is page-faults. They are handled at the commit stage by considering the first active faulting element whose access was attempted (just as in ARM SVE [7]). Nevertheless, page-faults have no consequence on the execution if they are a result of a miss-speculation, if the stream terminates earlier, or if the faulting element is otherwise inactive (i.e., predicated false).

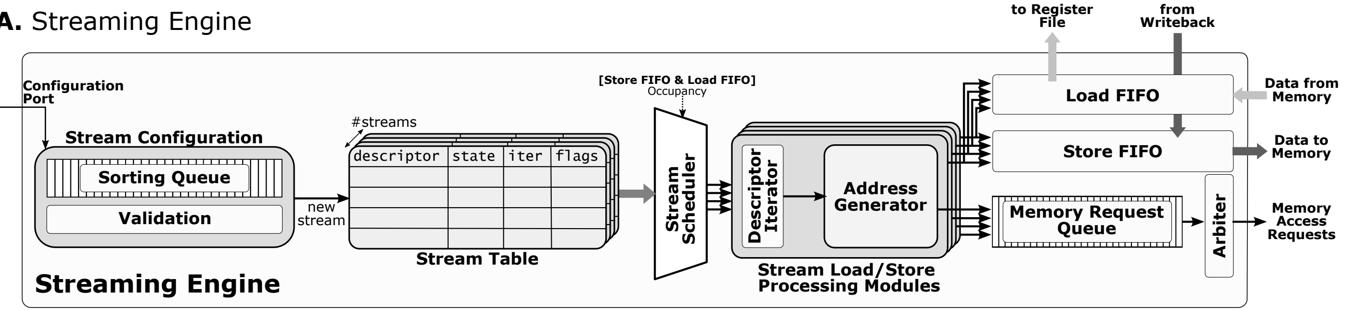
Context Switching. The context of running threads can be saved by suspending all active streams and then storing the commit stream iteration state. As a stream iteration describes a scalar access, resuming the execution (due to context switch or upon recovering from an exception) is simply performed by loading the saved iteration state and resuming from the last commit point (naturally all pre-fetched data in internal buffers is lost and must be re-loaded). The size of the saved stream state depends on the considered data pattern and varies between 32-Bytes (for 1-D patterns) and 400-Bytes (for a maximum of 8-D patterns and 7 modifiers).

B. Streaming Engine

The Streaming Engine is responsible for managing all input and output streams. As depicted in Fig. 7.A, its architecture consists of: (i) a *Stream Configuration* module, which is responsible for re-ordering the stream configuration requests; (ii) a *Stream Table*, which holds the information regarding the configuration of each stream on its multiple dimensions, as well as the information regarding current speculative and committed iteration states; (iii) a *Stream Scheduler*, responsible for the selection of a set of n (load/store) descriptors to be iterated by the *Stream Load/Store Processing Modules* (see Fig. 7.B); and a set of queues (*Memory Request*, and *Load and Store FIFOs*), which are used for buffering purposes. These modules operate as follows:

Stream Configuration. Whenever a new stream configuration reaches the rename stage of the processor pipeline, it is registered (in order) on the *SCROB*, where it awaits until all data dependencies are satisfied. Instructions are retrieved in order (one per clock cycle), validated, and used to write the data pattern configuration into the *Stream Table*.

Stream Processing. The stream processing is managed by the *Stream Scheduler* (see Fig. 7.A). At each cycle, it selects a set



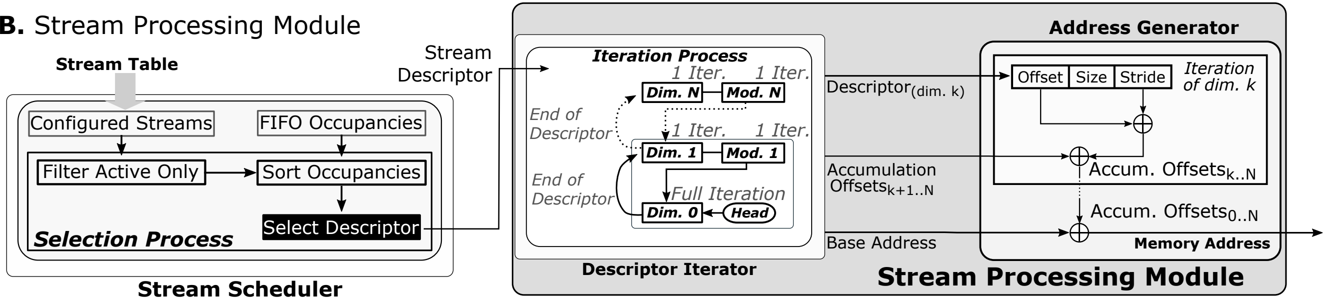


Fig. 7. Streaming Engine and Stream Processing Module logical block diagrams.

of n load/store streams from the *Stream Table*, which are then iterated by the *Address Generators* on the *Stream Processing Modules* (depicted in Fig. 7.B). Upon each iteration, the new stream state is registered on the *Stream Table* (unless the same stream is selected for processing), allowing to time-multiplex the iteration process of the several streams. Stream prioritization is achieved in the selection process, where streams with lower FIFO occupancy take precedence over the others, ensuring that the most consumed FIFO has the highest priority. To lower the hardware complexity (and size), each stream processing module only processes a dimension j (plus respective modifier) at a time, as detailed in Section II-B. When the j domain (dimension + modifier) reaches its end, it is restarted after a single iteration of descriptor $j+1$ (updating the accumulated offset from descriptors $(j+1)...N$). The *Address Generator* architecture is based on 2×16-bit incrementers, 2×64-bit adders, and some multiplexing logic, allowing to process one descriptor iteration and one modifier per cycle. This allows generating up to one cache line request per clock cycle, with one additional clock cycle being required to switch between descriptor dimensions. When succeeding descriptor iterations (from the same stream) hit the same cache line, a single memory request is issued, with the cache line data from the first request being reused on the following ones.

Iterating load streams generates new load requests, which are registered on the *Load FIFO* and *Memory Request Queues*. Then, the *Arbiter* picks such requests, performs the virtual-to-physical page translation (through a TLB access) and issues them to the memory, with the subsequent responses being placed on the corresponding *Load FIFO* queue slots. In the event of a page fault, the corresponding vector element is immediately flagged with an exception, which is subsequently handled at the commit stage. This avoids requests to invalid memory positions (even if speculatively) and allows the prefetching of stream data even across page boundaries.

Iterating store streams simply generates store addresses, which are directly written to the *Store FIFO*, until data is written back by the processor, and subsequently committed.

Load/Store FIFOs. As described in Section IV-A – *Memory Coherence*, since the *Streaming Engine* is integrated in the internal structure of the core, the *Load/Store FIFOs* work as an additional dimension of the vector registers. Consequently, the stream data buffered in these structures is regarded (from the memory hierarchy point-of-view) as already consumed data (by the core) and data coherence with the memory hierarchy is no longer enforced nor necessary. Accordingly, each stream is associated with an independent fixed-length FIFO queue, whose size (depth) was set to 8 in order to constraint the required hardware resources. Naturally, a better management could be attained by designing a single queue and sharing it across all streams. However, this would impose an additional hardware complexity and is therefore left for future work.

Stream Scheduler Policy. To select the set of n streams to be iterated, the *Stream Scheduler* relies on a simple policy, which prioritizes streams whose FIFO queues are less occupied.

C. Architectural impact overview

Besides the addition of the *Streaming Engine*, the proposed UVE imposes a minimal impact to the core's microarchitecture. Most of the necessary modifications are achieved by establishing direct connections to the *Streaming Engine* from the vector register file, execution, rename, and commit stages (registers can be added in-between to minimize impact on operating frequency). The decode stage is also extended to

TABLE I
CPU MODEL CONFIGURATION PARAMETERS BASED ON PUBLIC INFORMATION ABOUT THE ARM CORTEX A76 AND [7]. THE PARAMETERS REGARDING THE STREAMING ENGINE ARE SPECIFIC FOR UVE, WITH THE REMAINING BEING COMMON TO UVE AND THE BASELINE (ARM).

CPU (@1.5GHz)	4-wide instruction fetch, 4-wide μOp commit 8-wide μOp issue/dispatch/writeback 80 IQ, 32 LQ, 48 SQ, 128 ROB entries 128 Int RF, 192 FP RF, 48×512-bit Vector RF
Functional Units	2×Int ALUs with a 24-entry scheduler 2×Int vector/FP FUs with a 24-entry scheduler 2×Load + 1×Store ports with a 24-entry scheduler
Streaming Engine	2×Stream Load/store Processing Modules 8-entry Load/Store FIFOs per stream (default) 1×Load + 1×Store ports with a 24-entry scheduler
L1-I / L1-D	64KB 4-Way LRU Stride Prefetcher with depth 16
L2	256 KB 8-Way LRU AMPM Prefetcher [20], QueueSize 32 Snoop-based cache coherence protocol
DRAM	Dual-Channel DDR3-1600 8x8 11-11-11

include support for the new UVE instructions (much like any other SIMD extension). This results in an architectural impact in the core not much different from other SIMD extensions.

Conversely, aside the stream processing logic (described in Section IV-B), the *Streaming Engine* is mostly composed of storage and buffering structures, which are particularly dimensioned to minimize area impact while attaining significant performance gains (further detailed in Section VI). As it was referred before, stream and conventional load/store ports are typically used in mutually exclusive fashion and can be merged together - this was verified in the set of considered benchmarks in Section V.

V. EXPERIMENTAL METHODOLOGY

Simulation environment. The proposed microarchitecture was simulated on a modified version of Gem5 [27] using the parameters described in Table I. Although featuring RISC-V as the base instruction set [28], the adopted configuration is based on public information regarding the ARM Cortex-A76 and on [7]. Furthermore, since the main goal is to evaluate the per-core performance, it features no L3 cache memory, which is typically used for cross-core data sharing on multi-core processors. Additionally, to provide support for the proposed set of streaming extensions, it features a *Streaming Engine* with only two *Stream Load/Store Processing Modules* (see Fig. 7) and an 8-entry FIFO queue for each stream. Coherence between cache levels is ensured through snooping using a *MOESI* protocol.

Baseline architecture. For performance evaluation, the proposed solution is compared with an ARM core with full SVE support and featuring 512-bit vectors (as in the proposed solution). Apart from the *Streaming Engine*, which is non-existent in the ARM cores, all other processor parameters have the same configuration. In particular, to validate the proposed solution on equal grounds, the baseline architecture features two different prefetchers: a stride prefetcher associated to the L1-D and the *AMPM* prefetcher associated to the unified L2 [20].

Evaluation benchmarks. A representative set of benchmarks from several suites [29]–[33] (considering both workload and memory access patterns) was adopted for the evaluation of the proposed UVE instruction set and its corresponding implementation, as presented in Fig. 8 (left table). The list includes representative benchmarks relevant from several application domains, such as memory, linear algebra/BLAS, stencil, data mining, dynamic programming, and n-body (physics) systems.

Compilation toolchain. As it was referred, the development of a full compilation toolchain for UVE is a work-in-progress and is left for future work. As a consequence, the benchmarks' computational loops were hand-coded, although adopting straightforward implementations and avoiding the use of loop unrolling or software pipelining techniques (see example in Fig. 4). Hence, the GNU Compiler was extended to support all UVE instructions (when written in assembly mode), with the remaining code being compiled with the usual `-O2` flag.

For ARM SVE, the ARM compilation toolchain was used with flags `-O3`, `-march=armv8-a+sve` and `-fsimdmath`. However, the compilation toolchain failed to vectorize four benchmarks (identified in Fig. 8 with $\star$), namely `Seidel-2D`, `MAMR` (all variants), `Covariance` and `Floyd-Warshall`. All other benchmarks were inspected to guarantee that vectorization was correctly performed. It was observed that the compiler cost-benefit model decided not to use loop unrolling on top of vectorization, which is consistent with the UVE implementations.

VI. RESULTS

A. Performance evaluation

Fig. 8 presents the UVE performance evaluation in comparison with two ARM cores: the first featuring only NEON extension, the second expanded to also support the upcoming SVE. Both SVE and UVE extensions are configured to operate over vectors of 512 bits, with all UVE streams being configured (by default) to operate over the L2 cache. The presented evaluation considered the following aspects: (A) code reduction, (B) performance speed-up, (C) average number of blocks (stalls) per cycle at the rename stage, and (D) DRAM memory bus utilization, which is herein limited by the benchmark memory access pattern, and by the ratio between memory and core operating frequencies.

By analyzing the results in Fig. 8.B, it can be observed that the proposed extension provides a significant (average) performance advantage of 2.4× over ARM SVE (considering only vectorized benchmarks). The significant speed-ups are attained without relying on specific code optimizations, such as loop unrolling, as these would provide even greater performance improvements (see also Fig. 8.E).

The observed performance speed-ups come as a direct consequence of two main contributions: *i)* a significant code reduction (see Fig. 8.A), with an average 60.9% (93.2%) less committed instructions than ARM SVE (NEON); and *ii)* the

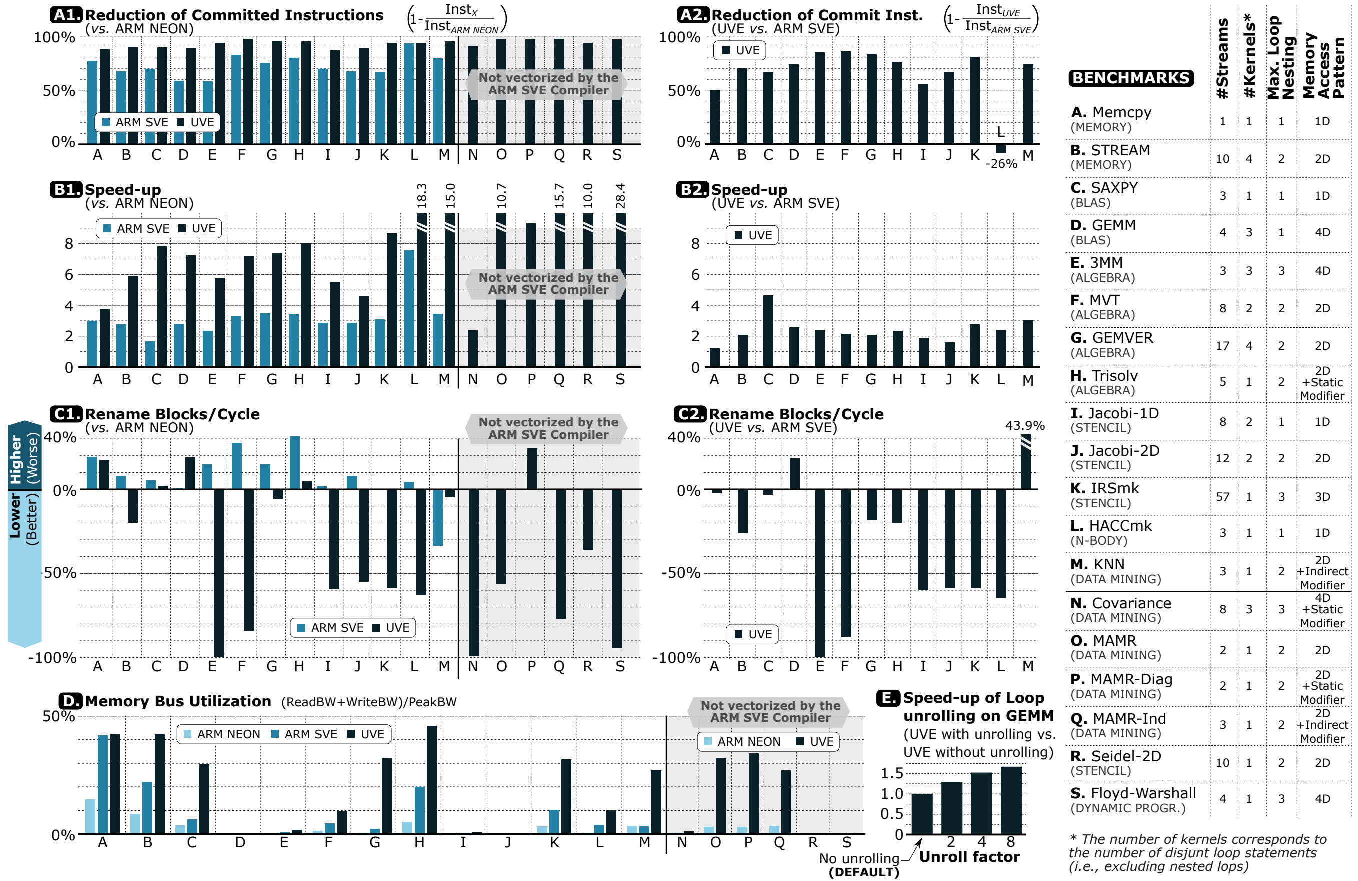


Fig. 8. Evaluation of the proposed vector extension (UVE). By default streaming is performed from/to the L2 cache. None of the UVE benchmark implementations feature loop unrolling on top of vectorization, with the exception on the results in E.

streaming infrastructure, which is able to significantly reduce the load-to-use latency and increase the effective memory hierarchy utilization.

In particular, it was observed a considerable improvement of the memory bus utilization, resulting on an average increase as high as 41× (see Fig. 8.D). This is especially observed for the `STREAM`, `SAXPY`, `GEMVER`, `IRSmk` and `MAMR` benchmarks. `GEMM`, `3MM`, `Jacobi-1/2D`, `Seidel-2D`, `Covariance`, and `Floyd-Warshall` did not affect this utilization because these benchmarks are L2-bound, leading to insignificant changes in this rate.

These two advantages also contribute to a reduction of the introduced stalls in the pipeline, particularly at the rename stage. In fact, by decreasing the number of instructions in the code, it is possible to significantly alleviate the pressure at the reorder-buffer and issue queue. On the other hand, the reduction of the load-to-use latency allows the incoming instructions to leave the pipeline earlier, decreasing the pressure on the physical register file. As a consequence, it is observed a significant decrease (33.4%) of rename blocks per cycle, when compared with the SVE-enabled core (considering only the benchmarks vectorized by the ARM compiler), as it is shown in Fig. 8.C.

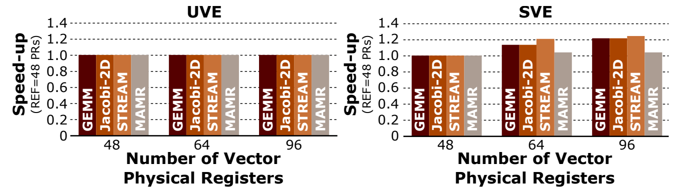


Fig. 9. Performance sensitivity to the number of physical vector registers.

B. Sensitivity to parameter variation

Number of Vector Registers. To complement the analysis about the register file pressure, a sensitivity evaluation regarding the number of physical vector registers (PVRs) was performed using a subset of the previously referred benchmarks. As it is observed in Fig. 9, increasing the number of PVRs provides no significant performance improvement for the UVE case, as the decreased load-to-use latency also results in a decreased pressure on the register file. This contrasts with the ARM SVE case, where an increase in the number of PVRs provides an increase in performance, showing a higher sensitivity to the number of PVRs.

Load/Store FIFO depth. The evaluation also considered the impact of the *Streaming Engine* FIFO buffers depth on the

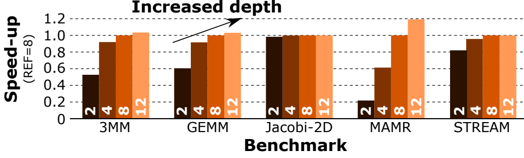


Fig. 10. Performance sensitivity to the depth of the FIFO buffers. Results normalized to the default value of 8 vector entries per buffer.

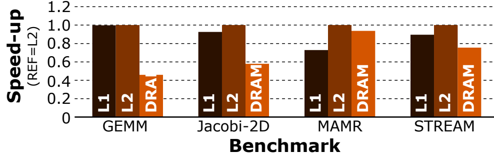


Fig. 11. UVE Performance sensitivity when streaming from/to different cache/memory levels.

overall performance, by considering four different values: 2, 4, 8 (default), and 12. Fig. 10 presents the obtained results in a subset of benchmarks. A minimum value of 4 is required to attain an adequate performance, with the setup using 8 slots (used by default in the previous experiments) showing to slightly improve the performance. The performance saturates for higher values, except for applications more sensitive to data access latency (as in MAMR), where a higher sensitivity to FIFO depth is observed.

Streaming cache level. The evaluation also measured the impact of streaming from different cache/memory levels. As it can be observed in Fig. 11, streaming from/to L2 generally provides the best performance benefits. However, there are specific benefits from fetching from different levels. An example use-case of L1 streaming regards small data structures that are reused across the program.

Stream Processing modules. Finally, the overall impact of the number of *Stream Processing Modules* in the *Streaming Engine* was also evaluated, by varying it between 2 and 8. Although the conducted simulation of such block considered that only one stream dimension (or modifier) can be processed per clock cycle, there is no significant difference in the overall performance, with the results varying by less than 0.1%.

C. Hardware overheads

According to the configuration in Table I, the *Streaming Engine* (depicted in Fig. 7) is composed of two Stream Processing modules, each featuring 2×16-bit incrementers, 2×64-bit adders and multiplexing logic, to process individual streams. The remaining modules are mostly composed of storage elements. The implemented *Stream Table* and *SCROB* accommodate up to 32 concurrent streams, each with a maximum of 8 descriptor dimensions and 7 modifiers, resulting in a total of 14 KB of storage. The Memory Request Queue maintains up to 16 outstanding requests, each packed within a 10-byte entry. Finally, the *Load/Store FIFO* buffers are composed of a 256×66-byte structure (for 32 streams, each with 8 entries), resulting in approximately 17 KB of storage. Hence, although a physical implementation of the *Streaming Engine* would be necessary to obtain accurate chip area and power values, it is still possible to conclude that the resulting structure would result in a footprint close to 1/2-th of an L1 cache. However, the impact can be mitigated by (i) reducing the number of streams from 32 to 8 and maximum number of dimensions to 4, still supporting the gains in Fig. 8) while reducing the memory footprint to only 6KB (≈ 10% of an L1 cache); and (ii) sharing the memory resources across streams (future work).

VII. Related Work

The idea behind the proposed UVE takes a significant step forward over a number of related processor architecture research areas, including *i)* scalable vectorization; *ii)* data streaming; *iii)* memory access decoupling; and *iv)* run-ahead execution/prefetching.

Scalable vectorization. For over two decades, SIMD extensions have been established as the go-to option to exploit DLP in HPC workloads [1], [2]. However, the rapid evolution of vector architecture solutions [3]–[6] has brought forth some code portability issues across different platforms. To overcome such issues, a new approach – vector-length agnostic vectorization [2], [7], [8] – has been recently proposed. In particular, SVE [7] and RVV [8] offer instruction sets whose vector length is not fixed and can be determined at execution time, allowing the same code to run on architectures with different vector lengths, without re-coding or re-compiling. SVE [7] relies on loop predication [9] to enable vector length scalability and predicated instructions to eliminate loop tails and to enable partially executed loop iterations. On the other hand, RVV [8], allows the configuration of a specific vector length and handles scalar loop tails by explicitly reducing the vector length. In contrast, the proposed UVE allows the *Streaming Engine* to automatically disable all vector elements that fall out of bounds, making loop control simpler by only requiring a minimal set of control instructions, which in most cases corresponds to a single branch instruction.

Data streaming and representation. Data streaming is traditionally associated to dedicated accelerator architectures [34]–[42] and it is often based on accurate representations of data access patterns to substantially accelerate data acquisition. Several approaches that have been proposed rely on dedicated ISAs [37], [43] and descriptor-based mechanisms [44], [45] to represent complex patterns. The underlying stream specification introduced by the proposed UVE unifies these two approaches by adopting a descriptor-based representation and by providing a stream configuration ISA interface. Moreover, it further extends these features by including stream control and predication capabilities, providing support for control- and data-dependent streams.

On the general-purpose domain, there has been a continued effort to bring stream specialization constructs [46], [47]. As an example, Wang et al. [46] introduced stream-based memory access specializations in out-of-order processors, to provide an execution-driven prefetching of repeated access patterns.

Schuiki et al. [47] also proposed a register file extension for single-issue in-order processors that introduces stream semantics on a subset of the processor's registers. However, these solutions present some limitations when compared to UVE. First, they are only capable of streaming the most common low-dimensionality patterns when they are directly exposed by the loop induction variables [47], albeit with indirection [46]. Second, although the stream-specialized applications from [46] can be susceptible to vectorization, none of the mentioned approaches explores stream vectorization at the ISA level.

Memory access decoupling. One of the main benefits of the streaming concept adopted by UVE is the decoupling of memory accesses from computation. This is achieved by configuring streams at the loop preamble and by allowing data acquisition to occur in parallel with data manipulation. A large body of work [48]–[53] has also explored this feature, by using decoupled access-execute architectures. As an example, Outrider [54] uses multiple in-order threads to decouple memory accesses from execution and to reduce the data access latency. It shows a high tolerance to memory access latency while using a low complexity in-order microarchitecture. Another example is DeSC [55], which explores this approach at the hardware-level by combining an out-of-order core for memory access with an accelerator for computation, thus fully decoupling the processing from the program control.

Run-ahead execution/prefetching. HW/SW prefetching methods have been specifically tailored to deal with memory access latency that results either from reduced data-locality [10]–[12], [24], complex [13]–[15] and indirect [16]–[18] memory access patterns, or large datasets that do not fit in cache [19], [56]. Currently, modern prefetchers achieve accuracy levels as high as 99% in data access prediction, and are mostly designed to deal with the timeliness and effectiveness of the procedure itself [20]–[24], as a way to mitigate over-prefetching and cache pollution. Contrasting to prefetching approaches, the stream-based paradigm of the proposed UVE allows the *Streaming Engine* to assure that only the exact data that is required is, in fact, obtained from memory, eliminating cache pollution issues. Nevertheless, UVE should not be viewed as an alternative to prefetching, but instead, a complementary structure that acts at the level of the processor to speed-up data acquisition.

VIII. Conclusion

In this paper a new Unlimited Vector Extension (UVE) is proposed to overcome performance degradations often observed in state-of-the-art vector-length agnostic SIMD extensions when executing costly scatter-gather and loop control operations. This new extension relies on a data streaming paradigm to explicitly decouple memory accesses from computation, removing loop-based control and memory indexing instructions from the code, while linearizing memory accesses to attain a simpler vectorization. The streaming is carried out by a dedicated streaming engine, co-located with the processor pipeline, and is supported by a descriptor-based memory access pattern specification, capable of encoding and streaming complex, multi-dimensional and indirect access patterns. A vectorized stream register file was also introduced in the processor pipeline to directly stream input/output data into/from the vector registers, mitigating the load-to-use latency. According to the evaluation that was conducted with the Gem5 simulator of a proof-of-concept implementation based on an out-of-order processor architecture, the combination of all these features introduced by UVE provides a performance increase of 2.4× when compared to ARM SVE.

Acknowledgements

This work was partially supported by national funds through Fundação para a Ciência e a Tecnologia (FCT) under projects UIDB/50021/2020 and PTDC/EEI-HAC/30485/2017, and by funds from the European Union Horizon 2020 Research and Innovation programme under grant agreement No. 826647.

References

[1] S. Hammond, C. Vaughan, and C. Hughes, "Evaluating the Intel Skylake Xeon processor for HPC workloads," in *International Conference on High Performance Computing & Simulation (HPCS)*, pp. 342–349, 2018.

[2] A. Pohl, M. Greese, B. Cosenza, and B. Juurlink, "A performance analysis of vector length agnostic code," in *International Conference on High Performance Computing & Simulation (HPCS)*, pp. 159–164, 2019.

[3] G. Xin, J. Han, T. Yin, Y. Zhou, J. Yang, X. Cheng, and X. Zeng, "VPQC: A Domain-Specific Vector Processor for Post-Quantum Cryptography Based on RISC-V Architecture," *IEEE Transactions on Circuits and Systems I: Regular Papers*, vol. 67, no. 8, pp. 2672–2684, 2020.

[4] B. Akin, Z. A. Chishti, and A. R. Alameldeen, "ZCOMP: Reducing DNN Cross-Layer Memory Footprint Using Vector Extensions," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, pp. 126–138, ACM, 2019.

[5] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, S. Baghsorkhi, and J. Torrellas, "SAVE: Sparsity-Aware Vector Engine for Accelerating DNN Training and Inference on CPUs," in *53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-53)*, pp. 796–810, IEEE, 2020.

[6] M. Cavalcante, F. Schuiki, F. Zaruba, M. Schaffner, and L. Benini, "Ara: A 1-GHz+ Scalable and Energy-Efficient RISC-V Vector Processor With Multiprecision Floating-Point Support in 22-nm FD-SOI," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 28, no. 2, pp. 530–543, 2019.

[7] N. Stephens, S. Biles, M. Boettcher, J. Eapen, M. Eyole, G. Gabrielli, M. Horsnell, G. Magklis, A. Martinez, N. Premillieu, A. Reid, A. Rico, and P. Walker, "The ARM Scalable Vector Extension," *IEEE Micro*, vol. 37, pp. 26–39, 3 2017.

[8] A. Waterman and K. Asanovic, "RISC-V "V" Vector Extension," 2019.

[9] A. Barredo, J. M. Cebrian, M. Moretó, M. Casas, and M. Valero, "Improving Predication Efficiency through Compaction/Restoration of SIMD Instructions," in *26th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 717–728, IEEE, 2020.

[10] Y. Guo, P. Narayanan, M. A. Bennaser, S. Chheda, and C. A. Moritz, "Energy-Efficient Hardware Data Prefetching," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 19, no. 2, pp. 250–263, 2011.

[11] M. Bakhshalipour, P. Lotfi-Kamran, and H. Sarbazi-Azad, "An Efficient Temporal Data Prefetcher for L1 Caches," *IEEE Computer Architecture Letters*, vol. 16, no. 2, pp. 99–102, 2017.

[12] T. F. Wenisch, M. Ferdman, A. Ailamaki, B. Falsafi, and A. Moshovos, "Practical off-chip meta-data for temporal memory streaming," in *15th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 79–90, IEEE, 2009.

[13] M. Shevgoor, S. Koladiya, R. Balasubramonian, C. Wilkerson, S. H. Pugsley, and Z. Chishti, "Efficiently prefetching complex address patterns," in *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*, pp. 141–152, ACM, 2015.

[14] K. J. Nesbit and J. E. Smith, "Data Cache Prefetching Using a Global History Buffer," in *10th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 96–96, 2004.

[15] L. Peled, S. Mannor, U. Weiser, and Y. Etsion, "Semantic Locality and Context-based Prefetching Using Reinforcement Learning," in *42nd Annual International Symposium on Computer Architecture (ISCA)*, pp. 285–297, ACM, 2015.

[16] I. Hadade, T. M. Jones, F. Wang, and L. d. Mare, "Software Prefetching for Unstructured Mesh Applications," *ACM Transactions on Parallel Computing*, vol. 7, no. 1, 2020.

[17] S. Ainsworth and T. M. Jones, "Software prefetching for indirect memory accesses," in *2017 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pp. 305–317, IEEE, 2017.

[18] X. Yu, C. J. Hughes, N. Satish, and S. Devadas, "IMP: Indirect memory prefetcher," in *48th International Symposium on Microarchitecture (MICRO-48)*, pp. 178–190, ACM, 2015.

[19] S. Somogyi, T. F. Wenisch, A. Ailamaki, B. Falsafi, and A. Moshovos, "Spatial memory streaming," in *33rd International Symposium on Computer Architecture (ISCA)*, pp. 252–263, 2006.

[20] Y. Ishii, M. Inaba, and K. Hiraki, "Access Map Pattern Matching for Data Cache Prefetch," in *International Conference on Supercomputing (ICS'09)*, pp. 495–496, ACM, 2009.

[21] P. Michaud, "Best-offset hardware prefetching," in *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 469–480, IEEE, 2016.

[22] S. H. Pugsley, Z. Chishti, C. Wilkerson, P.-f. Chuang, R. L. Scott, A. Jaleel, S.-L. Lu, K. Chow, and R. Balasubramonian, "Sandbox prefetching: Safe run-time evaluation of aggressive prefetchers," in *20th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 626–637, IEEE, 2014.

[23] S. Kondguli and M. Huang, "Division of Labor: A More Effective Approach to Prefetching," in *ACM/IEEE 42th Annual International Symposium on Computer Architecture (ISCA)*, pp. 83–95, 2018.

[24] M. Bakhshalipour, M. Shakerinava, P. Lotfi-Kamran, and H. Sarbazi-Azad, "Bingo Spatial Data Prefetcher," in *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 399–411, IEEE, 2019.

[25] A. Waterman and K. Asanovic, "The RISC-V Instruction Set Manual, Volume I: Base User-Level ISA Document Version 20190608-Base-Ratified," 2019.

[26] S. S. Baghsorkhi, N. Vasudevan, and Y. Wu, "FlexVec: Auto-Vectorization for Irregular Loops," *37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2016)*, pp. 697–710, 2016.

[27] N. Binkert, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, D. A. Wood, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, and T. Krishna, "The gem5 simulator," *ACM SIGARCH Computer Architecture News*, vol. 39, p. 1, 8 2011.

[28] A. Roelke and M. R. Stan, "RISC5: Implementing the RISC-V ISA in gem5," *First Workshop on Computer Architecture Research with RISC-V (CARRV)*, vol. 7, no. 17, 2017.

[29] J. D. McCalpin, "Memory bandwidth and machine balance in current high performance computers," *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter*, pp. 19–25, Dec. 1995.

[30] "ASC Sequoia Benchmark Codes." https://github.com/llvm/ llvm-test-suite/tree/main/MultiSource/Benchmarks/ASC_Sequoia/.

[31] "CORAL Benchmark Codes." https://asc.llnl.gov/coral-benchmarks.

[32] S. Grauer-Gray, L. Xu, R. Searles, S. Ayalasomayajula, and J. Cavazos, "Auto-tuning a high-level language targeted to GPU codes," in *Innovative Parallel Computing (InPar), 2012*, pp. 1–10, IEEE, 2012.

[33] B. Reagen, R. Adolf, Y. S. Shao, G.-Y. Wei, and D. Brooks, "Machsuite: Benchmarks for accelerator design and customized architectures," in *2014 IEEE International Symposium on Workload Characterization (IISWC)*, pp. 110–119, IEEE, 2014.

[34] B. Khailany, W. J. Dally, U. J. Kapasi, P. Mattson, J. Namkoong, J. D. Owens, B. Towles, A. Chang, and S. Rixner, "Imagine: Media Processing with Streams," *IEEE Micro*, vol. 21, no. 2, pp. 35–46, 2001.

[35] S. Ciricescu, R. Essick, B. Lucas, P. May, K. Moat, J. Norris, M. Schuette, and A. Saidi, "The reconfigurable streaming vector processor (RSVP/spl trade/)," in *36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-36)*, pp. 141–150, IEEE, 2003.

[36] L. Wu, A. Lottarini, T. K. Paine, M. A. Kim, and K. A. Ross, "Q100: The Architecture and Design of a Database Processing Unit," in *19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, p. 255–268, ACM, 2014.

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

[38] Y. Chen, J. Emer, and V. Sze, "Eyeriss: A Spatial Architecture for Energy-Efficient Dataflow for Convolutional Neural Networks," in *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 367–379, 2016.

[39] G. Weisz and J. C. Hoe, "CoRAM++: Supporting data-structure-specific memory interfaces for FPGA computing," in *25th International Conference on Field Programmable Logic and Applications (FPL)*, 2015.

[40] S. Liu, Z. Du, J. Tao, D. Han, T. Luo, Y. Xie, Y. Chen, and T. Chen, "Cambricon: An Instruction Set Architecture for Neural Networks," in *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 393–405, IEEE, 2016.

[41] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52)*, pp. 924–939, 2019.

[42] Y. Wang, X. Zhou, L. Wang, J. Yan, W. Luk, C. Peng, and J. Tong, "SPREAD: A Streaming-Based Partially Reconfigurable Architecture and Programming Model," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 21, pp. 2179–2192, Dec 2013.

[43] S. Paiágua, F. Pratas, P. Tomás, N. Roma, and R. Chaves, "HotStream: Efficient Data Streaming of Complex Patterns to Multiple Accelerating Kernels," in *25th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'2013)*, pp. 17–24, 2013.

[44] T. Hussain, M. Shafiq, M. Pericàs, N. Navarro, and E. Ayguadé, "PPMC: A programmable pattern based memory controller," in *Reconfigurable Computing: Architectures, Tools and Applications*, vol. 7199 LNCS, pp. 89–101, Springer, Berlin, Heidelberg, 2012.

[45] N. Neves, P. Tomas, and N. Roma, "Compiler-Assisted Data Streaming for Regular Code Structures," *IEEE Transactions on Computers*, vol. 70, no. 3, pp. 483–494, 2020.

[46] 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.

[47] F. Schuiki, F. Zaruba, T. Hoefler, and L. Benini, "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.

[48] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization," in *37th International Symposium on Microarchitecture (MICRO-37)*, pp. 30–40, IEEE, 2004.

[49] V. Govindaraju, C.-H. Ho, T. Nowatzki, J. Chhugani, N. Satish, K. Sankaralingam, and C. Kim, "DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing," *IEEE Micro*, vol. 32, no. 5, pp. 38–51, 2012.

[50] S. Kumar, N. Sumner, V. Srinivasan, S. Margerm, and A. Shriraman, "Needle: Leveraging program analysis to analyze and extract accelerators from whole programs," in *2017 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 565–576, IEEE, 2017.

[51] A. Sharifian, S. Kumar, A. Guha, and A. Shriraman, "Chainsaw: Von-neumann accelerators to leverage fused instruction chains," in *49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-49)*, pp. 1–14, IEEE, 2016.

[52] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation Cores: Reducing the Energy of Mature Computations," in *15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, p. 205–218, ACM, 2010.

[53] M. Pellauer, Y. S. Shao, J. Clemons, N. Crago, K. Hegde, R. Venkatesan, S. W. Keckler, C. W. Fletcher, and J. Emer, "Buffets: An efficient and composable storage idiom for explicit decoupled data orchestration," in *24th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pp. 137–151, 2019.

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

[55] T. J. Ham, J. L. Aragón, and M. Martonosi, "DeSC: Decoupled Supply-Compute Communication Management for Heterogeneous Architectures," in *48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-48)*, 2015.

[56] E. Ebrahimi, O. Mutlu, and Y. N. Patt, "Techniques for bandwidth-efficient prefetching of linked data structures in hybrid prefetching systems," in *15th IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pp. 7–17, IEEE, 2009.