ETH zürich

Sparse Stream Semantic Registers: A Lightweight ISA Extension Accelerating General Sparse Linear Algebra

Journal Article

Author(s):
Scheffler, Paul; Zaruba, Florian (iD); Schuiki, Fabian; Hoefler, Torsten; Benini, Luca (iD)

Sparse Stream Semantic Registers: A Lightweight ISA Extension Accelerating General Sparse Linear Algebra

Paul Scheffler, *Student Member, IEEE,* Florian Zaruba, Fabian Schuiki,
Torsten Hoefler, *Fellow, IEEE,* Luca Benini, *Fellow, IEEE*

Abstract—Sparse linear algebra is crucial in many application domains, but challenging to handle efficiently in both software and hardware, with one- and two-sided operand sparsity handled with distinct approaches. In this work, we enhance an existing memory-streaming RISC-V ISA extension to accelerate both one- and two-sided operand sparsity on widespread sparse tensor formats like compressed sparse row (CSR) and compressed sparse fiber (CSF) by accelerating the underlying operations of streaming indirection, intersection, and union. Our extensions enable single-core speedups over an optimized RISC-V baseline of up to 7.0x, 7.7x, and 9.8x on sparse-dense multiply, sparse-sparse multiply, and sparse-sparse addition, respectively, and peak FPU utilizations of up to 80% on sparse-dense problems. On an eight-core cluster, sparse-dense and sparse-sparse matrix-vector multiply using real-world matrices are up to 4.9x and 5.9x faster and up to 2.9x and 3.0x more energy efficient. We explore further applications for our extensions, such as stencil codes and graph pattern matching. Compared to recent CPU, GPU, and accelerator approaches, our extensions enable higher flexibility on data representation, degree of sparsity, and dataflow at a minimal hardware footprint, adding only 1.8% in area to a compute cluster. A cluster with our extensions running CSR matrix-vector multiplication achieves 9.9x and 1.7x higher peak floating-point utilizations than recent highly optimized sparse data structures and libraries for CPU and GPU, respectively, even when accounting for off-chip main memory (HBM) and on-chip interconnect latency and bandwidth effects.

Index Terms—Computer Architecture, Hardware Acceleration, Linear Algebra, Sparse Computation, Sparse Tensors

✦

1 INTRODUCTION

S PARSE linear algebra (LA) poses a formidable challenge for contemporary processor architectures optimized for highly regular and vectorizable workloads. In sparse tensors, a significant portion of entries is zero. To reduce their memory footprint and eliminate redundant computation, they are usually compressed by storing only the values and positions of nonzero elements. However, this results in nested indirect data structures that cannot be randomly accessed, irregular memory access patterns, and significant control overheads. Even the most fundamental sparse tensor operations, *addition* and *multiplication*, are inherently irregular. Worse, their optimal execution strategy depends on operand sparsity: while one sparse operand can be handled through indirect memory accesses, two sparse operands require comparing nonzero positions in both tensors.

Consider the multiplication of a sparse matrix `A`, stored in the widespread compressed sparse rows (CSR) format, with a dense vector `b`, which can be implemented as shown in Listing 1a. Here, `A_vals` and `A_idcs` store the nonzero values and their column indices, and `A_ptrs` delimits the rows of `A`. For each element of the result vector `c`, we perform a *sparse-dense* dot product `sV×dV`* as shown. Compiling this `sV×dV` for the RISC-V instruction set yields a loop of nine instructions, of which only one, a fused multiply-accumulate (MAC), performs necessary compute [1]. Thus, an in-order core can achieve at most $11\,\%$ floating-point unit (FPU) utilization, while a superscalar core would need at least nine issue slots to keep a single FPU busy, even when ignoring dependencies and control bottlenecks. In both cases, the main efficiency bottleneck is the *indirection* `b[A_idcs[j]]`, which requires half of the non-compute instructions and is also needed for sparse-dense addition.

Now, consider the multiplication of `A` with a sparse vector `b` stored in the common compressed sparse fiber (CSF) format. The *sparse-sparse* dot product `sV×sV` computed for each result element can be implemented as shown in Listing 1b. We *intersect* the nonzero indices in the two vectors since only positions where both vectors are nonzero contribute to the result. Compared to `sV×dV`, the control overhead is even higher and dynamically depends on the sparsity patterns of both operands. Sparse-sparse tensor

- *P. Scheffler and L. Benini are with the Integrated Systems Laboratory (IIS), ETH Zurich. E-mail: {paulsc,lbenini}@iis.ee.ethz.ch*
- *F. Zaruba is with Axelera AI. Email: florian.zaruba@axelera.ai*
- *F. Schuiki is with SiFive. Email: fabian@schuiki.ch*
- *T. Hoefler is with the Scalable Parallel Computing Laboratory (SPCL), ETH Zurich. Email: htor@inf.ethz.ch*
- *L. Benini is also with the Department of Electrical, Electronic, and Information Engineering (DEI), University of Bologna.*

*We henceforth use the prefixes `s` and `d` to denote *sparse* and *dense* tensors, respectively. Analogously, `M` denotes a *matrix* and `V` a *vector*.

```
for i in 0 to A_nrows:
  for j in A_ptr[i] to A_ptr[i+1]:
    c[i] += A_vals[j] * b[A_idcs[j]]
```

(a) Multiplication of a sparse matrix `A` in CSR format with a dense vector `b` (sM×dV), iterating sparse-dense dot products (sV×dV).

```
while ia < len_a and ib < len_b:
  if a_idcs[ia] == b_idcs[ib]:      ⎱ If indices match,
    c_i += a_vals[ia++] * b_vals[ib++]  ⎰ multiply-accum.
  while ia < len_a and              ⎱ Skip nonzeros
      a_icds[ia] < b_idcs[ib]:       ⎱ in a until index
    ia++                            ⎰ ≥ that in b
  while ib < len_b and              ⎱ Skip nonzeros
      b_idcs[ib] < a_icds[ia]:       ⎱ in b until index
    ib++                            ⎰ ≥ that in a
```

(b) Dot product of two sparse vectors `a` and `b` in CSF format (sV×sV).

Listing 1: Example sparse-dense and sparse-sparse product kernels.

```
call    cfg_sssr_a_ft0    ⎱ SSR ft0: read values of a
call    cfg_sssr_b_ft1    ⎱ SSR ft1: read values of b
hwloop  1                 ⎱ loop over 1 next instr.
fmadd.d %c, ft0, ft1, %c  ⎱ multiply-accumulate
```

Listing 2: sV×dV and sV×sV in RISC-V assembly using SSSRs and a hardware loop. The kernels differ only in the used SSSR configuration.

addition needs yet another approach: we must form a *union* of indices, since a nonzero in either operand produces a nonzero in the result.

Sparse compute is essential in many applications [2]. The SuiteSparse matrix collection [3] hosts sparse matrices from numerous real-world problems in domains including computational physics, mathematics, economics, and biology. In machine learning (ML), sparsification can significantly reduce the operations and memory required for a given inference accuracy [4]; while some approaches target sparse-dense compute by sparsifying only weights, others also exploit activation sparsity. Graphs, commonly represented as highly sparse matrices, are operands in both sparse-dense and sparse-sparse workloads such as PageRank [5] and triangle counting [6].

Established superscalar out-of-order architectures struggle with the high control overhead of sparse compute: a Fujitsu A64FX running sM×dV with a highly optimized format [7] achieves at most 130.9 Gflop/s or 4.7% of its peak compute. While recent CPU, GPU, and accelerator works improve sparse workload performance, they are limited by their lack of generality or large microarchitectural impact. Most general-purpose instruction set architecture (ISA) extensions consider only one-sided sparsity [8], [9], and those targeting sparse-sparse workloads incur large area impacts and cannot efficiently handle the former [6]. Sparse ML solutions often rely on custom, domain-specific formats and low or structured sparsity [10], [11], while more general sparse LA accelerators [12], [13] incur significant silicon area impacts compared to their host systems and other solutions.

To address these shortcomings, we present *sparse stream semantic registers* (SSSRs), a modular ISA extension significantly accelerating and improving FPU utilization on both sparse-dense *and* sparse-sparse LA *without* major microarchitectural impact. SSSRs extend stream semantic registers (SSRs) [14], which enable near-complete FPU utilization in in-order cores for dense linear algebra and other data-oblivious workloads[†] [15] by mapping memory streams with regular address patterns directly to architectural registers. We extend SSRs to accelerate the *indirection*, *intersection*, and *union* operations necessary for general sparse LA, incur-

[†]A workload whose memory access patterns are input-invariant.

ring only a small increase in overall hardware complexity while significantly improving sparse performance. For example, SSSRs allow us to write both sV×dV and sV×sV in RISC-V assembly as in Listing 2, where the registers ft0 and ft1 have stream semantics. Since all index processing and comparison is handled in hardware, the loop body consists of only the useful fmadd.d instruction, which we iterate over using a hardware loop.

We evaluate SSSRs by integrating them into the RISC-V Snitch [16] core and compute cluster, in which we evaluate their benefits on various sparse LA workloads using generated vectors and real-world matrices. On a single core, SSSR indirection, intersection, and union enable speedups of up to $7.0\times$, $7.7\times$, and $9.8\times$ over an optimized RISC-V baseline on sparse-dense multiply, sparse-sparse multiply, and sparse-sparse addition, respectively, with peak indirection FPU utilizations of up to 80%[‡]. On an eight-core cluster, sparse-dense sM×dV using SSSRs is up to $4.9\times$ faster and $2.9\times$ more energy efficient, while sparse-sparse sM×sV up to $5.9\times$ faster and $3.0\times$ more energy efficient. SSSRs incur only 1.8% in additional area in an eight-core cluster (128 KiB data memory and 8 KiB instruction cache) over regular SSRs. The synthesizable register transfer level (RTL) description for SSSRs and their integration into Snitch are available free and open-source[§].

SSSRs extend our prior work on *indirection SSRs* accelerating sparse-dense tensor products [1]. We include these contributions in this work to provide a holistic, coherent description of our SSR sparsity extensions. All contributions regarding sparse-sparse LA, most parameterizability work, and many minor improvements are original to this work, and we updated all existing evaluation to a newer version of Snitch [16] and a new technology node. In detail, our contributions are:

1) A lightweight, modular extension to SSRs handling streaming indirection, intersection, and union in hardware to efficiently handle sparse general LA in single-issue cores, with a complete open-source hardware implementation in an open RISC-V core and multi-core cluster. (Section 2).
2) A programming model for SSSRs, efficient sparse-dense and sparse-sparse LA kernels operating on vectors and matrices, and further applications for our hardware (Section 3).
3) Significant performance and energy efficiency benefits on single cores ($\leq 7.0, 7.7, 9.8\times$) and eight-core clusters with realistic interconnect and main memory models ($\leq 4.9, 5.9\times$ and $2.9, 3.0\times$), incurring a minimal 1.8% impact on cluster area (Section 4).

[‡]Full FPU utilization would require separate index memory ports, which we forego to minimize our architectural impact (see Section 2.2).

[§]https://github.com/pulp-platform/snitch: SSR IP in hw/ip/snitch_ssr and integration in hw/ip/snitch_cluster.

(a) Indirection address generator with datapath for each mode differentiated.

(b) Indirection and egress SSRs with their new address generators and preserved data mover in gray.

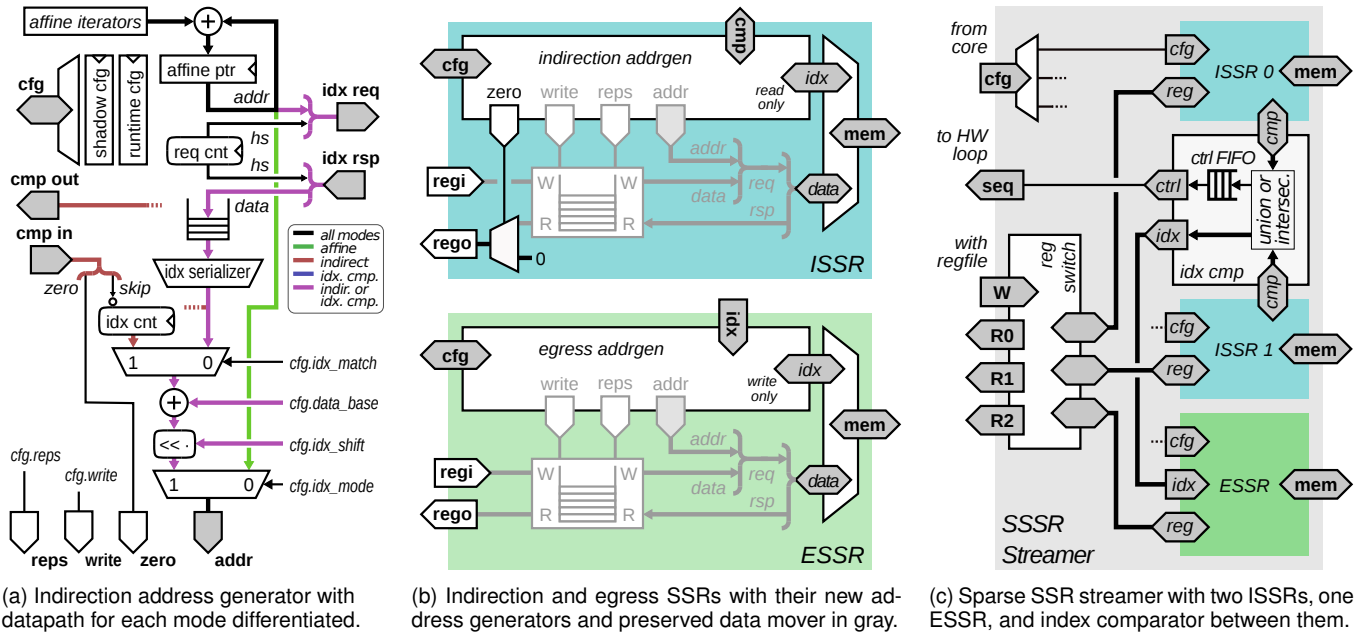(c) Sparse SSR streamer with two ISSRs, one ESSR, and index comparator between them.

Figure 1: Sparse SSR extension architecture, showcasing address generation, data movers, SSSR streamer, and inter-SSR index comparison.

4) A comparison to state-of-the-art CPU, GPU, and accelerator approaches, achieving higher generality, a lower area impact, and $1.7\times$ higher parallel sM×dV FPU utilization than GPUs (Section 5).

The paper is organized as follows: Section 2 describes our SSSR architecture and its integration into the Snitch core and cluster. Section 3 presents our SSSR programming model and sparse LA kernels. Section 4 evaluates the performance, timing, area, and energy efficiency benefits of SSSRs. Finally, Section 5 discusses related CPU, GPU, and accelerator work and compares it to our approach.

2 ARCHITECTURE

Most of our hardware extensions are confined to the address generator inside each SSR. We will first propose two new generator designs, the *indirection* and *egress* address generators, as well as two extended, backward-compatible SSR hardware variants using them. We will then present the sparse SSR streamer, which combines and connects the index datapaths of three extended SSRs to enable index intersection and union. Finally, we will discuss the streamer's integration into the Snitch core and cluster [16] in which we evaluate our extensions. We note that SSSRs are not strictly dependent on a specific core or instruction set and can be used with any core, ISA, or hardware loop.

2.1 Address Generators

We designed two extended SSR address generators to handle indexed streams. The *indirection* generator can *read* an index stream from memory to either generate indirect addresses or match read indices against those of another indirection SSR (ISSR) for intersection or union. The *egress* generator can *write* an externally provided index stream to memory while emitting corresponding data addresses.

Section 2.3 explains how these features can be combined to enable end-to-end sparse-sparse index processing.

2.1.1 Indirection Address Generator

Figure 1a shows the indirection address generator, highlighting the datapaths used by different generation modes. Like the original generator design, it exposes shadowed configuration registers to the host core, enabling the setup of a new stream while another is still running, and a control interface to the surrounding SSR logic. The index stream is fetched through an additional read-only memory port.

All generation modes reuse the existing affine address generator with up to four nested levels, but for different purposes. During *affine iteration*, it directly provides the address generator's output as shown. During *indirection* or *index matching*, it generates addresses for the words in memory containing the indices to be read; unlike fetching individual indices, this fully utilizes the memory bus. These words are fetched into a decoupling first in, first out (FIFO) queue. To prevent downstream blocking or a queue overflow, an outstanding request counter limits inflight requests.

SSSRs can handle indices of any unsigned 2^n B integer type that fits onto the used memory bus (i.e., 8, 16, 32, and 64 bit on Snitch). An *index serializer* extracts indices of the configured size from the buffered index words. To increase flexibility and programmability, we support arbitrary index array base alignment in memory. In indirection mode, the serialized indices are then shifted by a small programmable offset, enabling power-of-two striding; this enables indirection into the upper axes of power-of-two-strided tensors without incurring a costly hardware multiplier. Finally, the shifted indices are added to the configured base address, resulting in the desired indirect address stream.

During index matching, the indices are instead passed to an external matching unit, detailed in Section 2.3, which compares the read indices with those in another SSR's gen-
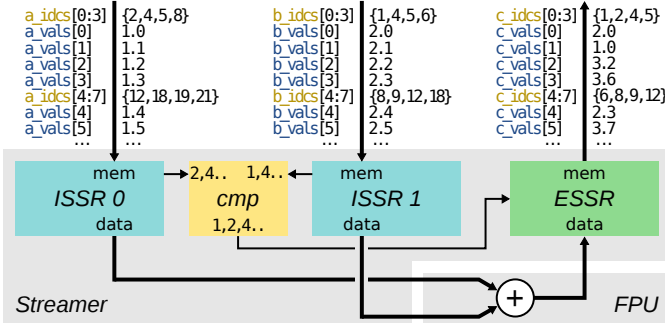
Figure 2: Example SSSR memory request sequences and index union dataflow for end-to-end `sV+sV` with 64-bit FP data and 16-bit indices.

Figure 3: Snitch Cluster used in evaluation with SSSR streamer

erator streaming the other operand. Unlike in indirection, the emitted addresses simply count up from the configured data base address with a unit stride, since there is exactly one nonzero value per index. However, depending on which index stream is ahead, the matching unit may instruct either generator to *skip* addresses or *insert* null data elements into the SSR data stream, forming the *intersection* or *union* of the index streams.

2.1.2 Egress Address Generator

The *egress* address generator is much simpler. It addresses the need to write back the indices of a joint stream alongside its data. For this purpose, it reads joint indices generated by an index matching unit through an additional port, passes them to an index *coalescer*, and writes them back through a *write-only* memory port, effectively reversing the indirection generator's index fetch datapath. It writes exactly one index per emitted address, but allows index writing to lead by a parameterized number of elements, decoupling the index and data streams and preventing unnecessary blocking.

Both extended generators are fully parameterized. The address and data widths, affine loop count, and index queue depth can all be adapted. Reduced internal index and address widths may be used for area-efficient designs operating on smaller memories.

2.2 Sparse SSRs

Figure 1b shows our extended SSRs built around the proposed address generators, the ISSR and egress SSR (ESSR). Like the original SSR, both provide a data queue decoupling the register and memory ports which is used in both read and write directions. Both expose their address generator's index matching ports, and the ISSR adds some multiplexing logic to enable the indirection generator to inject zero data elements into read streams for stream union.

In both the ISSR and ESSR, we chose to combine the index and data memory ports using a round-robin arbiter instead of requiring two memory ports. During affine streaming, this does not affect data throughput as the index port is unused. For both indirect and egress streaming, this tradeoff is motivated as follows: only one index word must be transmitted each n data words, where n is the number of indices per memory word. Thus, the peak achievable data mover utilization is $n/n+1$; on our evaluation platform with 64-bit data, this corresponds to 67%, 80%, and 88%
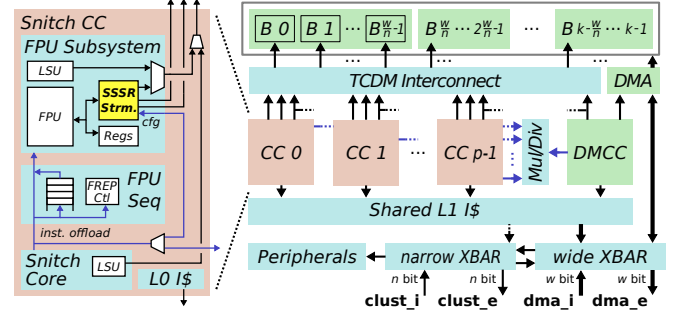
for 32-, 16-, and 8-bit indices, respectively. Alternatively, we could arbitrate memory accesses in the streamer or expose two core memory ports per SSR, trading higher peak utilization on sparse streams for a significantly larger cluster interconnect. In this work, we consider area-efficient SSSR variants with one port, enabling the drop-in replacement of and direct comparisons to the existing single-port SSRs.

2.3 Sparse SSR Streamer

Figure 1c shows the SSSR *streamer* in its default configuration. Like the original SSR streamer, it combines all available SSRs and provides shared configuration and register file interfaces to the host processor. A *register switch* maps each SSR's data channel to a predefined register when enabled.

The SSSR streamer is fully parameterizable and may be configured to map any number of SSRs, ISSRs, or ESSRs to any register indices. However, its optional inter-SSR index stream join capabilities, if present, are subject to some configuration constraints. A single streamer provides at most *one* index comparator; thus, only *two* ISSRs in a streamer may support index comparison between each other, and one optional third ESSR may consume the joined index stream for writeback. Thus, the default configuration comprises two index-comparing ISSRs and one ESSR.

The index comparator determines which, if either, of the ISSR index streams is ahead, and advances both as necessary to emit the union or intersection of indices. Figure 2 demonstrates on an `sV+sV` example how the index comparator generates a union of ISSR indices and forwards them to an ESSR for writeback, while the host core's FPU performs the required additions. The desired comparison mode is configured in and forwarded from the ISSRs. Since the number of index-data pairs in a joint stream is unknown ahead of time, the comparator also exposes a generic *stream control* queue, which provides a single bit indicating either that another element pair is available or that the joint data stream is complete. This queue can be read by a host core's branch unit or hardware loop unit in a decoupled fashion to determine the end of a joint stream.

2.4 Core and Cluster Integration

Figure 3 shows the integration of the default SSSR streamer into the Snitch core complex (CC) and cluster [16] used in evaluation. Like the previously integrated SSR streamer, it is multiplexed with the double-precision FPU's register file.

	Parameter	Possible values	Used
p	Worker core count	$\mathbb{N}^{>0}$	8
n	Narrow (core, bank) width	32, 64	64
w	Wide (L1 I\$, DMA) width	$\{n \leq 2^j \leq 1024 \mid j \in \mathbb{N}\}$	512
k	Memory bank count	multiples of w/n	32
D	TCDM size in KiB	$\{2^j \mid j \in \mathbb{N}\}$	128
I	L1 I\$ size in KiB	$\{2^j \mid j \in \mathbb{N}\}$	8

Table 1: Snitch cluster parameters and values used in evaluation.

The CCs preserve their existing memory topology, combining the core, FPU, and first ISSR into one memory port and providing exclusive ports to the second ISSR and ESSR. We also extend the existing floating-point repetition (FREP) hardware loop with a new mode using our stream control interface to issue one iteration per joint stream element.

Table 1 lists the essential design parameters and their possible values for the fully configurable cluster. All cores share an L1 instruction cache and an integer multiply-divide unit. A direct memory access (DMA) engine connected to a wide memory bus enables high-bandwidth data transfers between the tightly-coupled data memory (TCDM) and main memory. It is controlled by a lightweight data movement CC (DMCC) which can also be used for cluster management and coordination.

3 PROGRAMMING AND APPLICATIONS

The SSSRs are configured by the host core using a register interface accessed through custom instructions. This enables fast job setups: in our kernels, it takes at most 10 cycles to set up and launch new jobs for all three SSSRs from a cold state. In our configuration, ISSRs 0 and 1 map to temporary FPU registers `ft0` and `ft1` and the ESSR to `ft2`. The redirection of register accesses to SSRs is toggled through a common control register [14]. We focus on accelerating sparse LA at the instruction level by creating a library of hand-optimized operators, as is commonly done in high-performance LA packages. Future work will focus on extending compilers to automatically infer indirections, intersections, and unions in high-level code and map them to SSSRs.

3.1 Accelerable Formats and Operations

While we explore further workloads in Section 3.3, our main motivation is accelerating general sparse LA on a set of established tensor formats. Unlike approaches focused on low sparsities [10], [11], [17], SSSRs target flexibility and scalability to achieve notable speedups across a wider sparsity range (see Section 4.1). SSSRs support iterating along the major axis of any tensor format where said axis is given by two arrays: a *value* array storing nonzero values and an *index* array storing their positions. We call this pair of arrays a sparse *fiber*. Fiber-based tensor formats include the common CSR [18] and CSC [19] matrix formats, the generalized CSF [20] tensor format, and many format variations that slice data, pack it into blocks, or further compress upper tensor dimensions. The index shifter discussed in Section 2.1 also enables iterating on higher-level axes if all axes beneath it have power-of-two dimensions, which can be ensured through data chunking in practice. Indices can be of any unsigned 2^n B integer type that fits onto the memory bus.

The data width of SSSRs is fixed to that of the host processor, but single-instruction, multiple-data (SIMD) computations on blocked formats like BCSR [21] are trivially supported through zero padding. For example, the 64-bit FPUs in Snitch can leverage their 2- and 4-way SIMD capabilities on 32- and 16-bit formats when computing on blocks of two or four elements, respectively. We can operate on nonzero SIMD blocks the same way as on single double-precision elements; this substitution is orthogonal to the use of SSSRs and simply consolidates narrow nonzero elements into 64-bit nonzero SIMD blocks. Therefore, SSSRs should enable similar peak speedups and somewhat reduced speedups for a given nonzero pattern, as proportionally more nonzero elements must pack together to overcome SSSR setup costs.

SSSRs are not bound to specific sparsity regimes. They enable significant efficiency gains on sparse-sparse, sparse-dense, and even dense problems through SSR backward-compatibility. The supported formats have no structural sparsity requirements and scale well to extreme sparsities. Furthermore, SSSRs do not impose a specific higher-order dataflow: for example, sparse-sparse matrix multiplication (`sM×sM`) may be done in an inner-product, outer-product, or row-wise fashion to best suit the application and data.

To optimize area and leakage, the SSSR streamer parameterization can be tailored to specific application requirements. For efficient sparse-dense multiplication, one ISSR and one SSR are sufficient, with an additional ISSR needed for addition. For efficient sparse-sparse multiplication, two ISSRs suffice, with an additional ESSR incurred for addition.

3.2 Sparse Linear Algebra Kernels

To demonstrate the versatility of our extensions and evaluate their benefits, we create a set of sparse-dense and sparse-sparse linear algebra kernels for the RISC-V Snitch system extended with SSSRs. The kernels operate on CSR matrices and CSF vectors. Each is implemented for 8-, 16-, and 32-bit index types and, except for intersection-based kernels, in three variants:

- `BASE`: Stock RISC-V optimized baseline
- `SSR`: RISC-V extended with FREP loop and SSRs
- `SSSR`: RISC-V extended with FREP loop and SSSRs

To obtain the `BASE` kernels, we compile high-level C code with `clang -Ofast` and hand-optimize the resulting assembly for Snitch CCs through instruction reordering and loop unrolling to minimize memory or dependency stalls. The `SSR` kernels further improve performance by mapping regular, affine value streams such as sparse value arrays to SSRs, but without using our sparsity extensions. We note that this excludes kernels using intersection, since regular SSRs cannot further accelerate conditional stream loads. Finally, we implemented all kernels as `SSSR` variants fully leveraging our extensions, which we will discuss in detail.

3.2.1 Sparse-dense Kernels

sV×dV: Listing 3 shows our `SSSR` sV×dV kernel. Versions for different index sizes differ only in the indirection configuration for `ft1`. Our goal is to continuously issue the MAC instruction `fmadd.d` with minimal overhead to maximize FPU utilization. We achieve this in three phases:

```
csrsi    ssr_redir, 1
call     sssr_affine_setup_ft0
call     sssr_indir_setup_ft1
fcvt.d.w ft3, zero
...
frep     %len_a, 1, %n, 0b1001
fmadd.d  ft3, ft0, ft1, ft3
...
fadd.d   fa0, fa1, ft3
csrci    ssr_redir
```
- Redirect ft0, ft1, ft2 to SSRs
- SSR ft0: sparse a_vals
- SSR ft1: dense b at a_vals
- Zero-init ft3 ... ft[3+n]
- Stagger accumulator n-fold
- Reduce accumulators to return register fa0
- Disable SSR redirect

Listing 3: SSSR variant of sV×dV kernel: ft0 streams the values of sparse vector *a* and ft1 streams indirected values from dense vector *b*.

```
csrsi    ssr_redir, 1
call     sssr_union_setup_ft0
call     sssr_union_setup_ft1
call     sssr_egress_setup_ft2
frep.s   1, %n, 0b000
fadd.d   ft2, ft0, ft1
call     core_fpu_fence
ssrcfg.r a0, strctl_len
csrci    ssr_redir
```
- Redirect ft0, ft1, ft2 to SSRs
- SSR ft0: matched a_vals or 0
- SSR ft1: matched b_vals or 0
- SSR ft2: resulting c_vals
- emit until union stream ends
- Wait until FPU idle (job done)
- Read result length into a0
- Disable SSR redirect

Listing 4: SSSR variant of sV+sV kernel: ft0 and ft1 stream in sparse operands *a* and *b* and ft2 streams out sparse result *c*.

1) *Setup*: We first enable register redirection to SSRs. We then set up ISSR ft0 to linearly stream the sparse vector a's values in affine mode and ISSR ft1 to stream the dense vector b indirected at a's indices. Finally, we zero-initialize a contiguous block of accumulator registers starting at ft3.

2) *Compute*: We use FREP to iterate the useful fmadd.d instruction, streaming both multiplicands from the previously configured ISSRs. To prevent stalls caused by register dependencies on the accumulated sum, we use FREP's existing register staggering feature to increment the accumulator register index on each iteration, maintaining several partial sums in our contiguous block of registers; this is described in detail by Zaruba et al. [16]. Due to the limited peak throughput our ISSRs with one memory port impose, the larger the index type, the fewer accumulators are needed to hide these stalls.

3) *Teardown*: We reduce our accumulators into the return register fa0 and disable SSR redirection.

Since the compute and teardown phases contain only FPU instructions, the core is free to continue execution while the FPU is kept busy by FREP; this also enables the seamless setup of new shadowed SSR jobs. Synchronization between the core and FPU can be enforced whenever necessary.

sM×dV: Instead of simply iterating our sV×dV kernel for each matrix row, we further optimize sM×dV to reduce setup and reduction overheads and maximize FPU utilization. We stream the entire matrix fiber in single SSR and ISSR jobs, significantly reducing our setup overhead. Furthermore, we unroll the first few fmadd in each row, adding branches to shorter reductions, and issue an FREP loop and a full reduction only when necessary to accelerate short rows.

We use 32-bit row pointers in all variants to maximize row scaling. While we default on multiplying a CSR matrix from the left, our kernels provide runtime parameters to use different power-of-two vector and arbitrary result strides; this enables multiplying a CSR or CSC matrix with any power-of-two-strided dense tensor axis from either side.

sM×dM: We multiply a CSR matrix with a power-of-two-column row-major matrix. We simply iterate our sMxdV kernel here, as the additional overhead of iterating over dense data in a third-order loop is negligible (see Section 4.1.1). We again provide parameters for custom dense matrix and result strides, enabling multiplication of CSR or CSC matrices with row- and column-major matrices from either side. The only restriction is that our hardware index shifter requires the dense axis we indirect into to be power-

of-two-strided; in practice, this can be accommodated by tiling matrices using the cluster DMA's strided transfers.

sV+dV: We assume that the result is accumulated onto the dense vector; if necessary, a copy can be created using SSRs beforehand. We configure ft2 to stream the sparse vector values, ft0 to gather the dense addends, and ft2 to scatter sum elements back into the dense vector, and use FREP to execute fadd.d ft1, ft0, ft2 once for each sparse value. Either vector could also be scaled by a constant with the same performance by iterating an fmadd instead. This element-wise addition can be extended to fiber-based tensors of *any* shape through higher-level iteration.

sV⊙dV: Since one operand is dense, the result indices are the same as the sparse operand indices; they can optionally be copied with SSRs if needed. We multiply each sparse value with a gathered dense co-operand as for sV+dV, but instead linearly write out the results to yield the corresponding result value array. Again, this element-wise multiplication can be iterated to accommodate tensors of any shape.

3.2.2 Sparse-sparse Kernels

sV×sV: Since both indirection and intersection are fully handled inside SSSRs, the SSSR sV×sV kernel is identical to its sV×dV counterpart except for the SSSR and FREP configurations used. We configure ft0 and ft1 to stream the fibers of each operand and activate intersection such that only value pairs with matching indices are streamed to the FPU. Since the number of emitted pairs is unknown ahead of time in intersection, we use our stream-controlled FREP variant (frep.s) to issue one fmadd.d for each.

sM×sV: We iterate our sV×sV code for each result element. Since our intersection approach requires increasing indices in fibers, we launch new SSSR jobs for each row unlike for sM×sV. However, we can hide some of this configuration overhead through the decoupling of core and FPU subsystem and the shadowed SSSR configuration interface.

sM×sM: Three sparse-sparse matrix multiply dataflows, all accelerable with SSSRs, are commonly differentiated:

- *inner*: compute output element-wise by computing inner products (sV×sV) for each row-column pair.
- *row-wise*: compute output row-wise by accumulating sparse rows of left matrix (sV+sV) scaled with nonzeros in each row of right matrix.
- *outer*: compute outer product for each row-column pair (dense vector scaling), then accumulate all partial sparse result matrices (sM+sM).

Each method has a different iteration order, memory footprint, communication overhead, and data reuse, mak-

ing each attractive for different applications and sparsity regimes. SSSRs can accelerate all three methods by accelerating the subkernels pointed out above. Here, we limit ourselves to implementing a CSR×CSC *inner* kernel iterating our sM×sV kernel.

sV+sV: As shown in Listing 4, we read two sparse vectors from ISSRs ft0 and ft1 and write their sparse sum to ESSR ft2. The index comparator performs index *union* so that ft0 and ft1 emit either an element value or *zero* if an index only occurs in the co-operand. This allows us to simply iterate fadd.d as shown, with our stream-controlled frep.s ensuring the correct iteration count. Since ft2 is configured as an egress, one compared index will be written to memory for each computed sum value, resulting in the sum index vector. We read the result length from an ESSR configuration register after the job is done, which we ensure by synchronizing the core and FPU pipelines. When iterating this kernel to add higher-dimensional sparse tensors (e.g., sM+sM), we can defer this read to overlap computation with configuration as for sM×sV.

sV⊙sV: This kernel is almost identical to sV+sV; we instead configure the index comparator for *intersection* and iterate fmul.d. The same insights on higher-dimensional tensors and control-compute overlap also apply here.

3.3 Further SSSR Applications

SSSRs target general sparse LA, accelerating the many applications built on it including finite element analysis (FEM) simulations, sparse and graph neural networks, and graph workloads like PageRank [5]. However, the indirection, intersection, and union they accelerate are *general-purpose* operations that underlie many more irregular workloads, some of which we want to discuss here:

Codebook decoding: ISSRs can stream compressed data stored as index arrays pointing into smaller arrays of repeated values. Such *codebooks* could be used to efficiently store quantized deep learning parameters [22], image channels [23], and nonzero values of sparse tensors. Each ISSR in a streamer can independently stream codebooked vectors. Two ISSRs can handle sparse-dense LA with codebooked nonzeros (one gathering dense operands, the other nonzeros) with similar performance as our sparse-dense kernels, significantly reducing the size of compressible tensors.

Stencil codes: Iterative stencil codes process data on n-dimensional grids by accessing arrays in fixed, but irregular patterns relative to each grid point. Examples include partial differential equations and image processing [24]. ISSRs can accelerate stencils by storing them as index arrays and streaming them for each grid point, using its offset as a base address. The same approach can also accelerate sparse convolutions without im2col-like preprocessing.

Graph pattern matching: Intersection between the sparse adjacency vectors of graph nodes can be leveraged to identify and count subgraph embeddings and assess the similarity of graphs [6]. Thus, SSSRs may be used to accelerate graph pattern matching workloads in fields like computer vision and drug discovery.

Scatter-gather data transforms: ISSRs enable a streaming variant of *scatter-gather*. Thus, they can accelerate any existing scatter-gather-based transforms rearranging data, including sparse matrix transpose [25], parallel radix sort [26], or densifying sparse tensors by scattering their nonzeros.

4 EVALUATION

We will first evaluate the performance benefits of SSSRs on our LA kernels on a single Snitch CC and in parallel cluster sM×dV and sM×sV implementations backed by a realistic dynamic random access memory (DRAM) simulator. Next, we will discuss the area and timing impacts of different hardware configurations of our streamer. Finally, we will investigate the energy efficiency benefits of our extensions.

Like the baseline Snitch platform we integrate them into, our hardware extensions are designed and modeled as parametric, fully synthesizable SystemVerilog RTL hardware descriptions. This allows us to simulate both the baseline and our extended systems with full cycle accuracy and visibility, enabling the precise tracing and utilization tracking of all components. While we could use any SystemVerilog-capable simulator, we use *Questa Advanced Simulator* here. Our implementations used to evaluate area, timing, and power use the same RTL descriptions we simulate as inputs. Hardware resources external to the simulated CC or cluster are modeled behaviorally. Unless noted otherwise, the evaluated cluster and CC use default SSSR streamers and are parameterized as specified in Table 1.

For all experiments, dense test tensors are obtained by sampling normally distributed values and sparse vectors are generated for a given nonzero count and dimension with normally distributed values and uniformly distributed indices. The sparse matrices used are all real-world-problem matrices from the SuiteSparse Matrix Collection [3]; they cover various problem domains and aspect ratios and have 2k to 3.2k columns and 2.8k to 543k nonzeros.

4.1 Single Core Performance

To evaluate the performance benefits of SSSRs without the influence of other system components, we first evaluate our LA kernels in RTL simulation of a single CC by connecting it to an exclusive instruction cache and exclusive three-port data memory. These behave similarly to the shared instruction cache and TCDM channels in a cluster, respectively, except for additional cache misses and bank conflicts whose effects we include and discuss in Section 4.2.

4.1.1 Sparse-dense Kernels

sV×dV: Figure 4a shows the FPU utilization for our sV×dV kernels for varying sparse vector nonzero counts n_{nz}. We note that the kernel runtimes do not depend on the dense vector's length as long as it fits into the TCDM. Non-SSSR variants perform identically for all index sizes as a RISC-V load of any size incurs one instruction. SSSR kernel FPU utilizations are shown with and without accumulator reductions (dashed). Since an 8-bit-indexed CSF vector can hold at most 256 nonzeros, we also show results for 8-bit-index indirection with repeated indices (sssr8r).

All three SSSR index variants significantly outperform the BASE and SSR kernels, with 16- and 32-bit variants approaching their arbitration-imposed utilization limits of 67% and 80% and achieving $4.7\times$ and $5.6\times$ higher FPU

(a) CC `sV×dV` FPU utilizations vs. sparse vector nonzeros (dashed: without reductions).

(b) CC `sV+dV` FPU utilizations vs. sparse vector nonzeros (no reductions needed).

(c) CC `sM×dV` speedups of `SSR` and `SSSR` over `BASE` kernels for selected matrices.

(d) CC `sV×sV` speedups of `SSSR` over `BASE` kernel. Both vectors have dense size 60k.

(e) CC `sV+sV` speedups of `SSSR` over `BASE` kernel. Both vectors have dense size 60k.

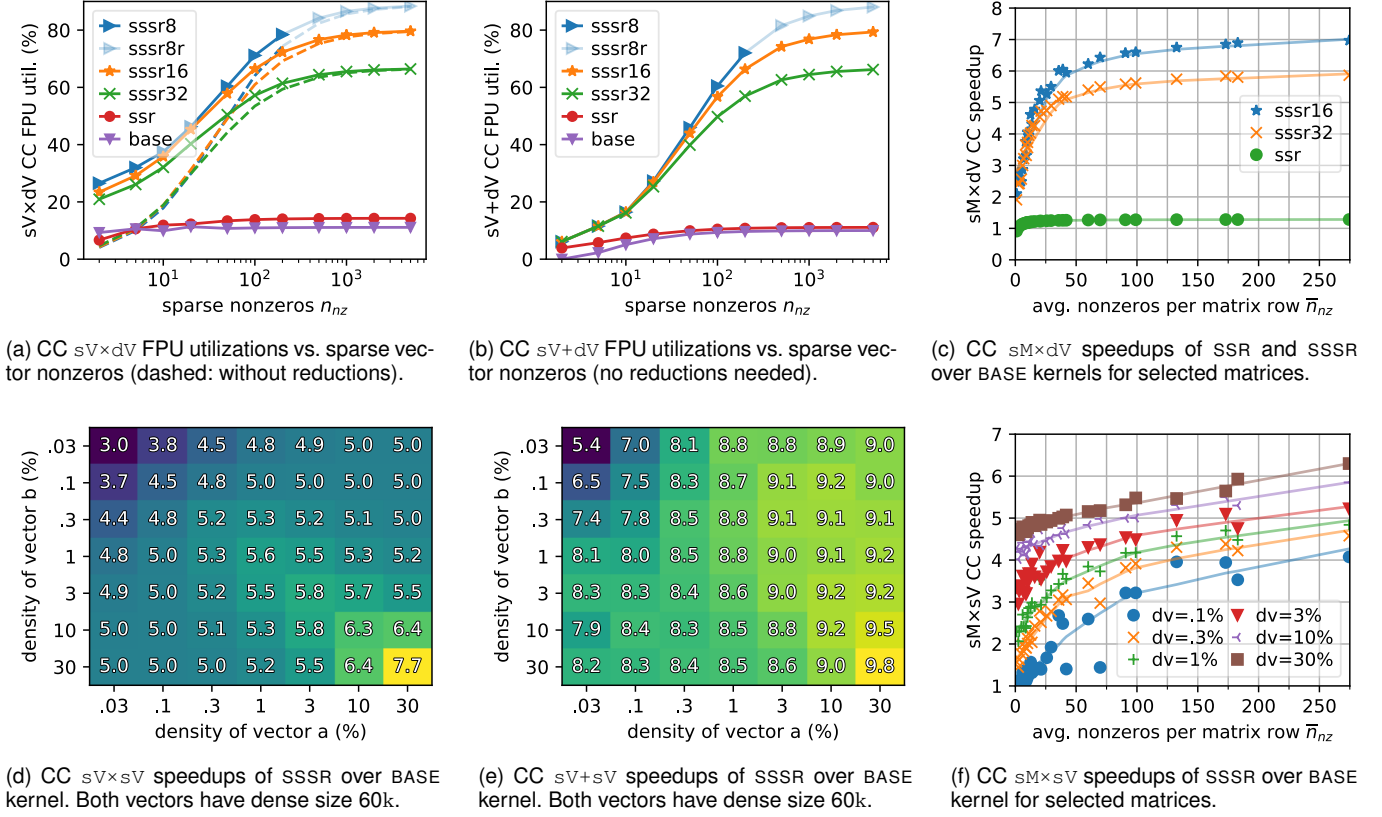(f) CC `sM×sV` speedups of `SSSR` over `BASE` kernel for selected matrices.

Figure 4: Single-CC performance results for selected kernels leveraging indirection, intersection, and union. 4a and 4b include results assuming 8-bit indices occur multiple times (`sssr8r`). The overlaid trend lines in 4c and 4f are fitted using locally estimated scatterplot smoothing (LOESS).

utilization than `SSR`. The 8-bit-indexed kernel reaches up to $5.8\times$ higher utilization, and 8-bit indirection with repeated indices approaches its arbitration limit of 88%. We note that `BASE` and `SSR` also approach their issue-bound utilization limits of one MAC every nine and seven cycles, respectively.

Because they process nonzeros much faster, the `SSSR` kernels need a significantly higher n_{nz} to fully overcome their setup and reduction overheads. For few nonzeros, their utilization without reductions is even lower than for `BASE` and `SSR`, motivating our row unrolling in `SSSR` `sM×dV` kernels. Since `SSSR` kernels with smaller indices need more accumulators to sustain their peak utilization, they outperform their larger-index counterparts only for higher n_{nz}.

sV+dV: Figure 4b shows the FPU utilization for our `sV+dV` kernels. Here, the result is *accumulated* onto the dense vector. We observe similar trends as for `sV×dV` with two key differences. Since a result value is written back for each nonzero, `BASE` and `SSR` peak utilizations further decrease to one MAC every ten and nine cycles. For the same reason, no more reduction is necessary in `SSSR` kernels as results are immediately written back by streaming indirection using the second ISSR. The `sssr` kernels using 32-bit indices, 16-bit indices, and 8-bit indices with reuse still approach their arbitration-imposed limits of 67, 80, and 88%.

sM×dV: Figure 4c shows the speedups of `SSSR` `sM×dV` kernels over `BASE` against the average nonzeros per matrix row $\overline{n}_{nz}$, reflecting the MACs performed per row. We assume here that the TCDM is large enough to store the full matrix. We do not evaluate 8-bit indices as they are not large

enough to index the chosen matrices' columns.

As for `sV×dV`, our 16-bit and 32-bit `SSSR` kernel speedups over `BASE` approach their theoretical limits and reach up to $5.9\times$ and $7.0\times$, requiring a higher $\overline{n}_{nz}$ than `BASE` and `SSR` to compensate their row iteration and reduction overheads. FPU utilizations reach up to 66% and 79%, respectively. Again, the 16-bit `SSSR` kernel outperforms the 32-bit variant only once $\overline{n}_{nz} \gtrsim 20$ due to its longer reduction sequence.

sM×dM: Even in edge cases, speedups and FPU utilizations are nearly identical to the `sM×dV` kernels we iterate: for the tiny sparse matrix *Ragusa18* with only 64 nonzeros, FPU utilization changes by only 0.12% when multiplying it with a two-column dense matrix instead of a dense vector.

4.1.2 Sparse-sparse Kernels

sV×sV: Figure 4d shows the speedup of our 16-bit `SSSR` `sV×sV` kernel over its `BASE` counterpart for vector densities of $0.03\text{-}30\%$ and vector length 60k. We observe speedups of $3.0\text{-}7.7\times$, increasing near-symmetrically with the density of both operands. Speedups are generally higher for similar densities; as densities diverge, speedups converge to $5.0\times$.

To understand these trends, we first consider two steady-state edge cases. While encountering *no* intersections and scanning one vector's nonzeros, `base` needs *five* and SSSRs *one* cycle per nonzero, explaining the $5.0\times$ speedup limit for divergent densities. While processing *only* intersections, `BASE` needs 18 and SSSRs 1.25 cycles per nonzero pair, yielding a peak achievable speedup of $14.4\times$ and FPU utilization of 80% if nonzero positions were to coincide exactly. Thus,
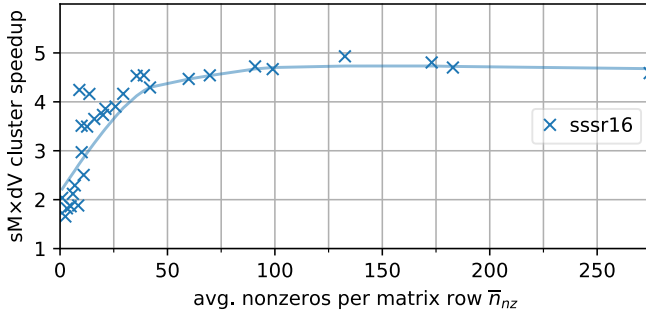
(a) Cluster `sM×dV` speedups of `SSSR` over `BASE` for 16-bit indices and selected matrices.
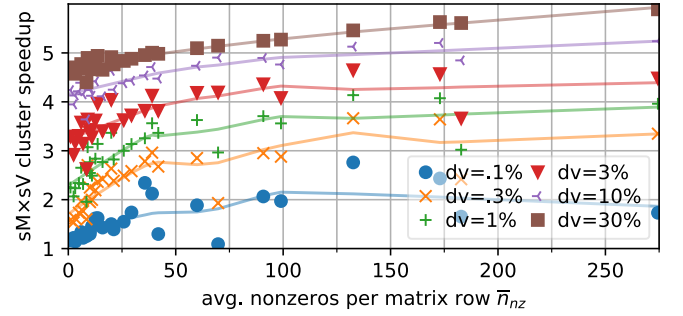
(b) Cluster `sM×sV` speedups of `SSSR` over `BASE` for 16-bit indices, varying vector densities, and selected matrices.

Figure 5: Eight-core cluster performance results for selected LA kernel scaleouts. Trend lines are fitted using LOESS.

as index matching likelihood increases with both densities, so do speedups. Finally, as both vectors become very sparse and contain few elements, intersection quickly terminates and speedups are offset by SSSR setup costs. They are also modulated by the random vectors' nonzero positions and sequences, introducing slight asymmetries.

sV+sV: Figure 4e shows the speedup of our 16-bit SSSR `sV+sV` kernel over BASE for the same vectors as for `sV×sV`. Speedups range from 5.4 to 9.8× and increase with the density of both operands with slight operand asymmetry.

To understand these trends, we again consider edge cases. While reading only summand a or b, SSSRs process each element up to 9.6× and 8.8× faster assuming continuous index comparison. In practice, occasional ESSR index write backpressure decreases speedups to the observed 9.0× and 8.2×. Speedups are asymmetric in a and b as ternary branching code in BASE incurs either one or two branch cycles. For continuous index matches, peak speedups are again 14.4×. Thus, speedups increase with increasing density in both operands and for similar densities.

sM×sV: Figure 4f shows the speedup of our 16-bit SSSR `sM×sV` kernel over BASE against the average nonzeros per matrix row $\overline{n}_{nz}$ for different vector densities. We again assume that the TCDM stores the full matrix.

We observe similar trends as for the `sV×dV` kernel we iterate: speedups increase with the density of both arguments and approach those of `sV×dV` for corresponding densities, reaching up to 6.3×. However, the smaller dense vector dimensions result in an earlier onset of offsetting and modulating effects as densities decrease: for $0.1\,\%$ vector density or 2-3 nonzeros, speedups are significantly reduced and strongly vary with nonzero patterns. However, speedups remain above 1, meaning we still amortize SSSR setup even for few nonzeros in both arguments.

4.2 Cluster Performance

To evaluate multi-core and system-level performance, we implement parallel scaleouts of `sM×dV` and `sM×dV` on a Snitch cluster. We reuse our architecture-optimized single-core kernels, distribute dynamically sized chunks of rows among cores, and implement a double-buffered data movement scheme for the matrix using the cluster DMA.

To ensure realistic memory throughput, latency, and scheduling, we connect our cluster to one of eight chan-

nels of a Micron MT54A16G808A00AC-36 HBM2E DRAM ($3.6\,\mathrm{Gb/s/pin}$ or $57.6\,\mathrm{GB/s}$ peak bandwidth, $88\,\mathrm{ns}$ average round-trip latency[¶]), which we simulate using *DRAM-Sys* [27]. All input data and instructions initially reside in this DRAM and all results are written back to it. A $16\,\mathrm{KiB}$ 4-way L2 instruction cache, bypassed by DMA data reads and writes, caches instruction fetches. In addition to DRAM, PHY, and controller delays modeled by *DRAMSys*, we model the delay of an on-chip interconnect with 16 additional cycles of forward and backward latency. We further explore the impact of interconnect latency and DRAM channel bandwidth sharing in Section 4.2.1.

sM×dV: Figure 5a shows the speedups of the 16-bit SSSR kernel over the BASE kernel on `sM×dV` parallelized on a cluster. Improvements are notable even for $\overline{n}_{nz} = 1$ at 1.7× and reach up to 4.9×, sustaining over 4× for $\overline{n}_{nz} > 30$. We observe the same trends as for single-core `sM×dV`, but with reduced speedups and stronger variations, which we attribute to multiple causes. Most importantly, *bank conflicts*, aggravated by the pseudorandom access patterns of indirection, lower the achievable TCDM bandwidth and thus ISSR throughput. Additionally, the *initial vector transfer* cannot be overlapped with useful computation, modulating speedups with the vector's length. Our matrix *double-buffering* and *row distribution* schemes also incur some computational imbalance and overhead. The high latency, limited throughput, and scheduling of our *DRAM model*, though mostly hidden through the double-buffered DMA transfer of long, contiguous matrix chunks, also slightly decrease speedups by a median of $1.8\,\%$ and up to $6.9\,\%$. Finally, we observe occasional stalls due to instruction cache misses. Despite these inherent parallelization and scaleout overheads, SSSRs enable up to $46.8\,\%$ overall FPU utilization and extreme efficiency gains at minimal area cost: *eight* cores with SSSRs achieve the same computational throughput as *39* cores running BASE.

sM×sV: Figure 5b shows 16-bit SSSR `sM×sV` speedups for different vector densities, which peak at 5.9×. As with `sM×dV`, we see similar trends as in the used single-core kernel with diminished speedups due to initial vector copy, instruction cache and TCDM stalls, work-sharing imbalances, and DRAM inefficiencies. For high vector densities,

---

[¶]Averaged from DMA matrix chunk reads in SSSR `sM×dV` using peak-speedup matrix `mycielskian12`. Excludes on-chip interconnect.
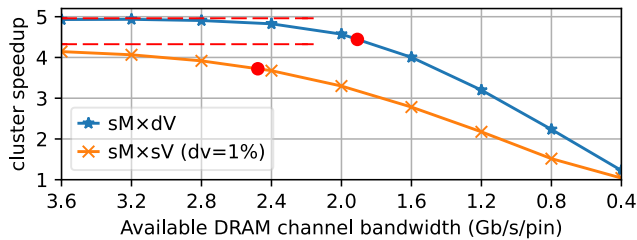
(a) Cluster sM×∗V speedups vs. limited DRAM channel bandwidth. The red dots indicate where speedups have degraded by 10%.
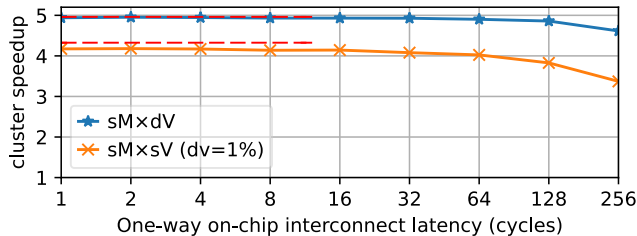
(b) Cluster sM×∗V speedups vs. on-chip interconnect latency.

Figure 6: Sensitivity of SSSR Snitch cluster speedups to limited DRAM channel bandwidth and varying on-chip interconnect latency. The red dashed lines show speedups for unlimited bandwidth and zero latency.

performance is mainly affected by TCDM stalls and setup overheads: the peak speedup is only 6.9% lower than in the single-core case. Because each core processes only a subset of matrix rows and due to variations in DRAM response time, few-nonzero modulation is more pronounced overall. As the vector density d_v decreases and computational imbalance increases, DRAM inefficiencies increase: for $d_v = 30\%$, speedups are only 0.40% lower on average than with an ideal memory system, while for $d_v = 0.1\%$, they are 16% lower. The minimum observed speedup of $1.1\times$ is *higher* than in the single-core case, which we attribute to the notably smaller code size and thus fewer instruction cache stalls the SSSR kernel incurs.

4.2.1 Bandwidth and Latency Sensitivity

To evaluate the sensitivity of SSSR speedups to bandwidth sharing and interconnect scaling, we rerun cluster sM×dV and sM×sV with limited DRAM throughput and varying interconnect latencies. We use the peak-sM×dV-speedup matrix mycielskian12 ($\overline{n}_{nz} = 133$, 4.3% density) to ensure high DRAM pressure and use 1% vector density for sM×sV. In both cases, we also indicate ideal speedups assuming a memory system with unlimited bandwidth and zero latency.

Figure 6a shows how speedups decrease as the available DRAM channel bandwidth is restricted from its full throughput of 3.6 to $0.4\,\mathrm{Gb/s/pin}$, simulating channel sharing with other bus agents. We maintain the one-way interconnect latency of 16 cycles. For SSSR-accelerated sM×dV with full DRAM bandwidth, the cluster incurs an average throughput of $R_T = 1.6\,\mathrm{Gb/s/pin}$; we thus expect our accelerated sM×dV to be compute-bound above R_T and memory-bound below it. Indeed, speedups start decreasing linearly below R_T until they reach $1\times$ just below $0.4\,\mathrm{Gb/s/pin}$, past which both the accelerated and baseline variants are memory-bound and perform the same. However, we also
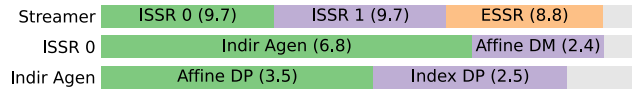
(a) Area breakdown of the default streamer and major subcomponents (areas in kGE). The residual area is shown rightmost in light gray.
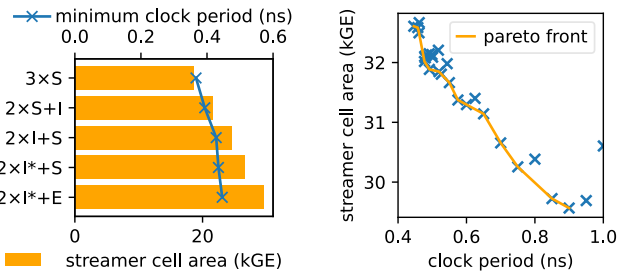
(b) Area and min. clock period for different streamer configurations.
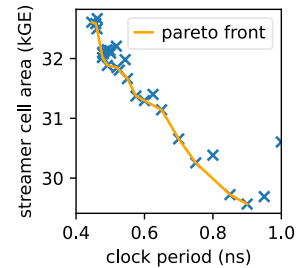
(c) Streamer area vs. clock period for different target clocks.

Figure 7: GF12LP+ synthesis area and timing results for SSSR streamer. S: SSR, I: ISSR, E: ESSR, I*: ISSR with shared comparator.

observe some speedup degradation *above* R_T; this is because DRAM bandwidths higher than R_T can hide minor throughput variations due to computational imbalances and DRAM scheduling. For sM×sV, we observe similar trends: in this case, R_T is higher due to faster processing of the matrix fiber, and speedups in the compute-bound region degrade slightly faster due to increased computational imbalance.

Figure 6b shows how speedups decrease as on-chip interconnect latency increases, assuming full DRAM channel throughput. For sM×dV, speedups remain near-constant until one-way latency exceeds 64 cycles, comparable to that of the downstream DRAM subsystem. This is because the cluster's DMA engine moves matrix data in double-buffered chunks tens of KiB in length, which saturate the bus for hundreds of cycles; performance is negatively impacted only once the total access latency becomes comparable to the durations of these transfers. For sM×sV, the same trends hold as matrix data is still moved in large, double-buffered chunks. However, greater variations in computation rate and imbalance result in slightly more pronounced speedup losses as latency increases.

Overall, the efficiency of SSSRs in removing instruction bottlenecks moves sparse workloads closer to the memory bound, and they still provide notable speedups well into the memory-bound regime and only fully lose their benefits when DRAM throughput is reduced to an order of magnitude less than regular channel bandwidth. This is despite sM×dV and sM×sV being very memory-intensive; sparse workloads allowing for matrix data reuse, such as matrix-matrix multiplications and iterative solvers, are significantly less memory-bound. Furthermore, our double-buffered data movement approach proves highly latency-resilient, compensating for hundreds of cycles in interconnect latency in addition to DRAM delays with minimal speedup losses.

4.3 Area and Timing

We synthesize the SSSR streamer in various configurations using Synopsys *Design Compiler* for GlobalFoundries' 12LP+

FinFET technology. We target the typical (TT) process corner at $25\,°C$ with $0.8\,V$ supply voltage. Unless stated otherwise, a $1\,GHz$ clock and IO delays of $35\,\%$ and $60\,\%$ on the core and TCDM interfaces are constrained. In our default streamer configuration, each of the two ISSRs and single ESSR has four data FIFO stages, 17-bit addresses to cover our $128\,KiB$ TCDM, and four affine loop levels.

Figure 7c shows the default streamer's hierarchical area distribution. Each ISSR contributes $9.7\,kGE$ and the ESSR $8.8\,kGE$ to the $30\,kGE$ streamer. ISSR area is dominated by its address generator, only $37\,\%$ of which is fully dedicated to index streaming. Figure 7b compares the area and minimum achievable clock period for different streamer configurations. If only indirection capabilities are needed, only $3.0\,kGE$ ($16\,\%$) in additional area is incurred per ISSR. Intersection between two SSRs incurs another $2.1\,kGE$. The full SSSR streamer with added union capability incurs $11\,kGE$ ($60\,\%$) area overhead over the baseline and only moderately increases the minimum clock period from $367\,ps$ to $446\,ps$, still easily meeting Snitch's $1\,GHz$ clock target. Figure 7a shows the achieved minimum period and area for different period goals, demonstrating that SSSR streamer area scales gracefully as timing pressure increases. At the cluster level, adding SSSR streamers to all eight worker cors incurs only a minimal $1.8\,\%$ cell area overhead compared to providing only regular SSRs.

4.4 Energy and Power

We estimate the power and total energy consumption of a Snitch cluster running our workloads from Section 4.2 with 16-bit-index BASE and SSSR kernels. We use the same interconnect and main memory models as in Section 4.2. We target the TT process corner of GlobalFoundries' 12LP+ FinFET technology at $1\,GHz$. We implement the cluster using Synopsys *Fusion Compiler* and estimate power for the low- and high-efficiency matrices cryg2500 and cavity12 using Synopsys *PrimeTime*, then scale dynamic power with component utilizations measured in RTL simulation.

Figure 8a shows the total energies for sM×dV using both kernel variants for each matrix. While the median cluster power is predictably lower for the BASE kernel ($195\,mW$ vs. $285\,mW$), SSSRs reduce the minimum energy per fmadd from $282\,pJ$ to $103\,pJ$ and achieve energy efficiency improvements of up to $2.9\times$. Figure 8b shows the total energies for sM×sV for a vector density of $1\,\%$. Again, SSSRs incur higher median power, but lower the minimum energy per matrix nonzero from $107\,pJ$ to $43\,pJ$ and achieve energy efficiency improvements of up to $3.0\times$.

5 RELATED WORK

In this section, we compare our approach to numerous state-of-the-art CPU, GPU, and accelerator solutions, covering both hardware and software proposals. We summarize our findings in two tables. Table 2 collects the evaluation platform, matrix format, and highest fraction of peak compute achieved on all discussed FP64 sM×dV software implementations. Table 3 compares the features, flexibility, and architectural cost of all discussed hardware designs.

Scalar processor architectures: A processor with SSSRs can be considered an extended and specialized decoupled

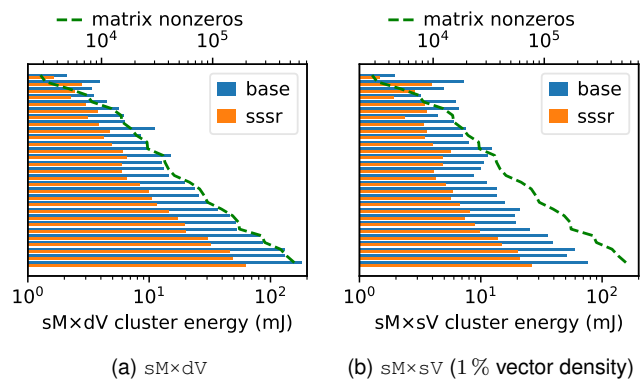(a) sM×dV

(b) sM×sV ($1\,\%$ vector density)

Figure 8: Snitch Cluster workload energy estimates in GF12LP+. The benchmarks and data are the same as for the results in Section 4.2.

access/execute (DAE) architecture [28]. DAE architectures combine two decoupled processors, one handling *memory accesses* and another *execution*, which communicate through register-mapped FIFOs. SSSRs provide a lightweight access processor design with multiple performance enhancements. They leverage multi-element *streams* as abstractions, which improves decoupling, minimizes offloads from the main instruction stream, and attains near-ideal compute and memory throughputs. Their job types define a compact, orthogonal instruction set that minimizes access code complexity and enables lightweight implementation. Unlike the original DAE concept, SSSRs support multiple concurrent streams on different registers. Our low-cost sparsity extensions maintain high throughput on indirect and joint streams without the hardware and bookkeeping overheads of a more conventional access processor architecture, enabling efficient general sparse compute without major execute processor modifications.

Vector processor architectures: Indirection in ISSRs is similar to *scatter-gather* in vector architectures, which finds recent adoption in superscalar out-of-order processors. Arm's Scalable Vector Extensions [29] and Intel's Knights Landing (KNL) [30] add support for SIMD scatter-gather. However, the short lengths and alignment requirements of SIMD vectors limit scatter-gather benefits on sparse workloads compared to ISSRs. The Unlimited Vector Extension (UVE) [31] relaxes vector length constraints and introduces vector-register-bound indirect streams similar to ISSRs. However, it requires vectorizable workloads and cannot operate efficiently at scalar granularity. Gong et al. [32] extend a vector multiply-add unit to skip ineffectual and coalesce useful computations. While this is orthogonal to SSSRs, it is only effective on low, regular sparsity in uncompressed data.

Software optimizations for scalar and vector processors: Numerous software solutions aim to accelerate sparse LA on scalar and vector processors. The CVR matrix format [33] aims to improve sM×dV SIMD lane and cache efficiency on x86 CPUs. Still, it utilizes at most $0.69\,\%$ of peak FP64 compute on Xeon Phi 7250. Zhang et al. [34] optimize sM×dV for Intel's AVX-512 extensions, proposing a format based on sliced ELLPACK (SELL) that enables peak FP64 compute utilizations of up to $1.5\,\%$ on Xeon Phi 7230. Regu2D [35] leverages an adaptive 2D partitioning scheme and arranges rows as similar-sized groups with zero-padding as neces-

	Work	Platform	Format	Peak FP Ut.
CPU	CVR [33]	Xeon Phi 7250	CVR	0.69 %
	Zhang et al. [34]	Xeon Phi 7230	SELL-like	1.5 %
	Regu2D [35]	Xeon Gold 6132	Regu2D	3.1 %
	Alappat et al. [7]	A64FX	SELL-C-σ	4.7 %
GPU	Tsai et al. [37]	V100	CSR	1.6 %
	Merrill et al. [38]	K40	CSR	2.0 %
	TileSpMV [39]	A100	tile-adapt.	2.9 %
	Tsai et al. [37]	Radeon VII	CSR	3.2 %
	cuSPARSE [40]	GTX 1080 Ti	CSR	17 %
	TileSpMV [39]	Titan RTX	tile-adapt.	27 %
	SSSRs (ours)	Snitch + SSSRs	CSR	47 %

Table 2: Overview of discussed FP64 `sM×dV` software implementations.

	Work	Open source	1-sided accel.	2-sided accel.	Usage flexib. *	Sparsity flexib. †	kGE per core‡
CPU	SVE S/G [29]		✓		M	H	-
	KNL S/G [30]		✓		M	H	-
	UVE [31]	✓**	✓		H	H	72¶
	Gong et al. [32]		✓	✓	M	L	31¶
	Prodigy [8]		✓		M	H	10¶
	SpZip [41]		✓		M	H	116
	Z. Wang et al. [9]		✓		H	H	-
	SparseCore [6]			✓	H	H	619
GPU	A100 [17]		✓		M	L	-
	Zhu et al. [42]		✓		L	M	12§
	Y. Wang et al. [10]			✓	L	M	157§
Accelerators	MatRaptor [43]			✓	L	H	-
	OuterSPACE [44]			✓	L	H	-
	Sadi et al. [45]		✓		L	H	-
	EIE [46]		✓	✓	L	H	-
	ExTensor [12]			✓	M	H	-
	Dadu et al. [13]		✓	✓	M	H	-
	SISA [47]			✓	M	H	-
	SSSRs (ours)	✓	✓	✓	H	H	30

L/M/H = Low/Medium/High. *Qualitative, based on operand granularity, use constraints, overheads. †Qualitative, based on efficiently handled range, structure. ‡Based on authors' data, different cores used. **gem5 model only, no RTL. ¶Arch. state only at 6T=1.5GE per bit. §Only node given; we assume GlobalFoundries' 22nm and 12nm, resp.

Table 3: Overview of discussed hardware designs.

sary to fill SIMD lanes, improving their platform's peak FP64 utilization to 3.1 %. Alappat et al. [7] model the performance of `sM×dV` on Fujitsu's A64FX using the SELL-C-σ matrix format and report up to 130.9 Gflop/s or 4.7 % of peak FP64 compute. Nevertheless, this is 9.9× less than what we achieve on a Snitch cluster with SSSRs. Z. Xie et al. [36] present a CSR-interfaced `sM×sM` library for multi- and manycores selecting the best internal format and algorithm with a deep learning model based on nonzero patterns.

Streaming ISA extensions: General-purpose processor extensions increasingly leverage streams as abstractions to accelerate compute. Prodigy [8] uses compile-time-inferred metadata to prefetch nested indirect streams. However, it does not eliminate load-store or bookkeeping instructions. SpZip [41] reads and writes compressed irregular data using decoupled streaming units. This is similar to SSSRs reading and writing fiber-based formats, but SpZip does not support intersection or union, and a fetcher-compressor pair incurs 116 kGE or 3.9× more area than an SSSR streamer. Z. Wang et al. [9] propose a decoupled register-mapped extension accelerating affine and indirect memory streams similar to ISSRs. However, streams need to stepped explicitly, severely limiting possible speedups on in-order cores, and intersection and union are not supported. SparseCore [6] accelerates intersection, union, and subtraction of indexed streams and provides dedicated functional units. However, indirection is not accelerated, making the handling of one-sided sparsity inherently inefficient. Additionally, SparseCore incurs $0.183 \, \text{mm}^2$ per stream unit in Open-Cell 15 nm, which is comparable to two entire Snitch CCs with SSSRs.

GPU software: Moving from general-purpose processors to more specialized architectures, GPUs have been the subject of intensive investigations aimed at improving their efficiency on sparse data computations. One-sided sparsity on GPUs is often accelerated in *software* through specialized algorithms and tensor formats. Merrill et al. [38] improve the performance consistency of CSR `sM×dV` on GPUs using a merge-based decomposition, reaching up to 2.0 % of peak FP64 compute on a K40 GPU. Nvidia provides optimized sparse LA kernels for their GPUs through *cuSPARSE* [40]. We evaluate their CSR `sM×dV` kernels from CUDA Toolkit 10.0 on a Jetson AGX Xavier (FP32 only) and a GTX 1080 Ti GPU (FP32 and FP64), profiling 100 invocations with *nvprof* on the same matrices as for SSSRs. For FP32, both GPUs exhibit high peak streaming multiprocessor (SM) occupancies of 96 % and 87 %, but low peak floating-point utilizations of

and 2.1 % and 0.75 % on active SMs, suggesting low thread parallelism among warps. FP64 on the GTX 1080 Ti shows similar SM occupancies, but a notably higher peak floating-point utilization of 17 %, likely due to the 32× fewer FP64 cores per SM. Tsai et al. [37] create optimized `sM×dV` library kernels for both Nvidia and AMD GPUs and various formats. Their CSR kernels regularly outperform vendor code and reach up to 1.6 % and 3.2 % of peak FP64 utilization on Nvidia V100 and AMD Radeon VII GPUs, respectively. TileSpMV [39] uses warp-optimized kernels on sparse tiles in a given or adaptively selected format. It achieves peak FP64 utilizations of 2.9 % on A100 and 27 % on Titan RTX, which again has 32× fewer FP64 than FP32 cores per SM. Still, this is 1.7× lower than in a Snitch cluster with SSSRs. Shi et al. [48] accelerate `sM×dM` by encoding matrices as arrays of coordinate-encoded tiles, improving access contiguity and data reuse. However, the average FP32 speedups of 1.7× over cuSPARSE on SuiteSparse matrices are insufficient to reach FP utilizations comparable to those of Snitch with SSSRs. Karimi et al. [49] design a set of memory-aware matrix formats accelerating `sM×dV` by improving cache usage and thread-level parallelism. While improving performance, their best format achieves median FP32 speedups of only 1.8× (K40) and 3.5× (V100) over cuSPARSE CSR on SuiteSparse matrices, still not enough to match our utilizations.

Some sparse-sparse software approaches for GPUs also exist: Zachariadis et al. [50] accelerate `sM×sV` using a tensor-core-based outer-product approach with dense 16×16 tiles, accepting low flexibility and losses of unnecessary computations within blocks. Li et al. [51] optimize general sparse matrix-vector multiply by dynamically choosing one of eight `sM×dV`/`sM×sV` approaches based on tensor properties using a ML model. This solution inherently cannot exceed

the peak performance of the existing approaches chosen from, but can be adapted to improve overall performance on other systems including ones with SSSRs.

GPU hardware: Extensions to GPU *hardware* accelerating sparsity have also been proposed. Nvidia's A100 architecture [17] introduces support for one-sided *structured* sparsity by efficiently handling up to two zeros in every block of four values. SSSRs efficiently handle a much wider range of sparsities ($\gg 50\,\%$) without any structural requirements and enable random accesses into a full TCDM. Zhu et al. [42] present an algorithm and co-optimized tensor core modifications for efficient sparse-dense neural network inference. Y. Wang et al. [10] extend V100 tensor cores to efficiently handle sparse-sparse matrix multiply and convolution with an outer product approach. However, the bitmap encoding used hinders efficient performance and memory scaling beyond low sparsities, and the dense output allows for the overall approach to be replicated with scattering ISSRs.

Hardware accelerators: Sparse accelerators usually target a single application domain like deep learning (DL) [22] or specific LA operations and dataflows. Most use custom data formats and precisions tailored to their use case and are incapable of general-purpose computation. Notably, a general-purpose processor with SSSRs can implement many of the employed specialized computation schemes. MatRaptor [43] and OuterSPACE [44] accelerate sM×sM with row-wise and outer product dataflows, respectively; SSSRs can efficiently handle both dataflows. Sadi et al. [45] accelerate large-scale sM×dV by column-slicing matrices and then accumulating the sparse results, improving vector locality and reuse; iterating over slices could be done on a host core while SSSRs handle multiplication and result accumulation. Han et al.'s EIE [46] minimize DRAM access energy in DL inference by keeping full, weight-shared models in on-chip SRAM; SSSRs also enable high utilization on sparse data with codebooked values, but are not limited to one application, schedule, or dataflow. ExTensor [12] hierarchically intersects CSF tensors to avoid redundant computation and transfers; a processor with SSSRs can replicate this by handling the upper axes in software while the streamer intersects the performance-critical primary axes. Dadu et al. [13] present a compute fabric accelerating both stream joins and streaming indirection. However, the loosely-coupled spatial compute approach incurs notable costs in area and programmability. SISA [47] introduces a set-centric ISA to accelerate graph mining algorithms. In addition to set intersection and union, it also accelerates set difference, membership and size queries, and element insertion or removal. It supports both indexed and bitmap sets and a binary-search-based "galloping" mode for index joins between sets of divergent sizes. While SISA is highly specialized for graph mining and does not target one-sided sparsity or "low-complexity" algorithms, many of its additional capabilities could be added to SSSRs. However, SISA's architectural cost is discussed only in loose analogy to other designs, making such tradeoffs hard to assess.

6 CONCLUSION

In this work, we extend SSRs to accelerate *indirection*, *intersection*, and *union*, creating a modular, backward-compatible ISA extension enabling efficient general sparse linear algebra on sparse-fiber-based formats including the widespread CSR and CSF. We evaluate our sparse SSRs in the RISC-V Snitch system by presenting efficient single-core sparse-dense and sparse-sparse LA kernels and parallelizing matrix-vector products on an eight-core cluster.

In a single core using 16-bit indices, the streaming indirection of SSSRs enables peak sV×dV FPU utilizations of $80\,\%$ and sM×dV speedups of up to $7.0\times$ over our optimized baseline. The intersection and union of SSSRs enable speedups of 3.0-$7.7\times$ and 5.4-$9.8\times$ on sparse-sparse dot product and vector addition across a wide sparsity range, enabling sM×sV speedups of up to $6.3\times$. Scaled out to a Snitch cluster with interconnect and DRAM models exhibiting realistic latency, throughput, and scheduling, sM×dV using SSSRs is up to $4.9\times$ faster and $2.9\times$ more energy efficient, while sM×sV is up to $5.9\times$ faster overall and $3.0\times$ more energy efficient for a vector density of $1\,\%$. These improvements incur only $11\,\mathrm{kGE}$ in additional area per core or $1.8\,\%$ across a cluster, which may further be reduced thanks to the streamer's modular design.

Unlike recent CPU, GPU, and accelerator proposals, SSSRs accelerate one- *and* two-sided sparse compute while remaining flexible in dataflow and operand sparsity and incurring a minimal area impact. A Snitch cluster with SSSRs running sM×dV achieves $9.9\times$ and $1.7\times$ higher peak floating-point utilizations than recent CPU and GPU software.

REFERENCES

[1] P. Scheffler, F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Indirection stream semantic register architecture for efficient sparse-dense linear algebra," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 1787–1792.

[2] Z. Zhang, Y. Xu, J. Yang, X. Li, and D. D. Zhang, "A survey of sparse representation: Algorithms and applications," *IEEE Access*, vol. 3, pp. 490–530, 2015.

[3] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, pp. 1:1–1:25, 2011.

[4] T. Hoefler, D. Alistarh, T. Ben-Nun, N. Dryden, and A. Peste, "Sparsity in deep learning: Pruning and growth for efficient inference and training in neural networks," *J. Mach. Learn. Res.*, vol. 22, pp. 241:1–241:124, 2021.

[5] L. Page, S. Brin, R. Motwani, and T. Winograd, "The pagerank citation ranking : Bringing order to the web," in *WWW 1999*, 1999.

[6] G. Rao, J. Chen, J. Yik, and X. Qian, "Sparsecore: stream isa and processor specialization for sparse computation," in *Proc. 27th ACM Int. Conf. Architectural Support for Program. Lang. and Operating Sys.*, 2022, p. 186–199.

[7] C. L. Alappat, J. Laukemann, T. Gruber, G. Hager, G. Wellein, N. Meyer, and T. Wettig, "Performance modeling of streaming kernels and sparse matrix-vector multiplication on a64fx," *2020 IEEE/ACM Perform. Modeling, Benchmarking and Simul. High Perform. Comput. Syst. (PMBS)*, pp. 1–7, 2020.

[8] N. Talati, K. May, A. Behroozi, Y. Yang, K. Kaszyk, C. Vasiladiotis, T. Verma, L. Li, B. Nguyen, J. Sun, J. M. Morton, A. Ahmadi, T. M. Austin, M. F. P. O'Boyle, S. A. Mahlke, T. N. Mudge, and R. G. Dreslinski, "Prodigy: Improving the memory latency of data-indirect irregular workloads using hardware-software co-design," in *2021 IEEE Int. Symp. High-Performance Comput. Architecture*, 2021, pp. 654–667.

[9] Z. Wang and T. Nowatzki, "Stream-based memory access specialization for general purpose processors," in *2019 ACM/IEEE 46th Annu. Int. Symp. Comput. Architecture*, 2019, pp. 736–749.

[10] Y. Wang, C. Zhang, Z. Y. Xie, C. Guo, Y. Liu, and J. Leng, "Dual-side sparse tensor core," in *2021 ACM/IEEE 48th Annu. Int. Symp. Comput. Architecture*, 2021, pp. 1083–1095.

[11] A. Gondimalla, N. Chesnut, M. Thottethodi, and T. N. Vijaykumar, "Sparten: A sparse tensor accelerator for convolutional neural networks," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, p. 151–165.

[12] K. Hegde, H. A. Moghaddam, M. Pellauer, N. C. Crago, A. Jaleel, E. Solomonik, J. S. Emer, and C. W. Fletcher, "Extensor: An accelerator for sparse tensor algebra," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019, p. 319–333.

[13] V. Dadu, J. Weng, S. Liu, and T. Nowatzki, "Towards general purpose acceleration by exploiting common data-dependence forms," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019.

[14] 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 Trans. Comput.*, vol. 70, pp. 212–227, 2021.

[15] V. Ramachandran and E. Shi, "Data oblivious algorithms for multicores," in *Proc. 33rd ACM Symp. Parallelism in Algorithms and Architectures*, 2020, p. 373–384.

[16] F. Zaruba, F. Schuiki, T. Hoefler, and L. Benini, "Snitch: A tiny pseudo dual-issue processor for area and energy efficient execution of floating-point intensive workloads," *IEEE Trans. Comput.*, vol. 70, pp. 1845–1860, 2021.

[17] Nvidia Corporation, "NVIDIA A100 Tensor Core GPU Architecture." [Online]. Available: https://www.nvidia.com/content/dam/en-zz/Solutions/Data-Center/nvidia-ampere-architecture-whitepaper.pdf

[18] S. C. Eisenstat, M. C. Gursky, M. H. Schultz, and A. H. Sherman, "Yale Sparse Matrix Package," Department of Computer Science, Yale University, Tech. Rep., 1977.

[19] I. S. Duff, R. G. Grimes, and J. G. Lewis, "Sparse Matrix Test Problems," *ACM Trans. Math. Softw.*, vol. 15, no. 1, p. 1–14, Mar. 1989.

[20] S. Smith and G. Karypis, "Tensor-matrix products with a compressed sparse tensor," in *Proc. 5th Workshop Irregular Applications: Architectures and Algorithms*, 2015, paper 5, pp. 1–7.

[21] A. Pinar and M. T. Heath, "Improving performance of sparse matrix-vector multiplication," *ACM/IEEE SC 1999 Conference (SC'99)*, pp. 30–30, 1999.

[22] S. Dave, R. Baghdadi, T. Nowatzki, S. Avancha, A. Shrivastava, and B. Li, "Hardware acceleration of sparse and irregular tensor computations of ml models: A survey and insights," *Proc. IEEE*, vol. 109, pp. 1706–1752, 2021.

[23] M. W. Marcellin, M. A. Lepley, A. Bilgin, T. J. Flohr, T. T. Chinen, and J. H. Kasner, "An overview of quantization in jpeg 2000," *Signal Process.: Image Communication*, vol. 17, no. 1, pp. 73 – 84, 2002.

[24] G. Roth, J. M. Mellor-Crummey, K. Kennedy, and R. G. Brickner, "Compiling stencils in high performance fortran," in *ACM/IEEE SC 1997 Conf. (SC'97)*, 1997, pp. 12–12.

[25] P. Stathis, D. Cheresiz, S. Vassiliadis, and B. H. H. Juurlink, "Sparse matrix transpose unit," in *18th Int. Parallel and Distributed Processing Symp., 2004. Proc.*, 2004, pp. 90–99.

[26] M. Zagha and G. E. Blelloch, "Radix sort for vector multiprocessors," in *Proc. 1991 ACM/IEEE Conf. Supercomputing*, 1991, pp. 712–721.

[27] M. Jung, C. Weis, and N. Wehn, "Dramsys: A flexible dram subsystem design space exploration framework," *IPSJ Trans. Syst. LSI Des. Methodol.*, vol. 8, pp. 63–74, 2015.

[28] J. E. Smith, "Decoupled access/execute computer architectures," *ACM Trans. Comput. Syst.*, vol. 2, pp. 289–308, 1984.

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

[30] A. Sodani, R. Gramunt, J. Corbal, H. Kim, K. Vinod, S. Chinthamani, S. Hutsell, R. Agarwal, and Y. Liu, "Knights Landing: Second-Generation Intel Xeon Phi Product," *IEEE Micro*, vol. 36, no. 2, pp. 34–46, 2016.

[31] J. M. Domingos, N. Neves, N. Roma, and P. Tomás, "Unlimited vector extension with data streaming support," in *2021 ACM/IEEE 48th Annu. Int. Symp. Comput. Architecture*, 2021, pp. 209–222.

[32] Z. Gong, H. Ji, C. W. Fletcher, C. J. Hughes, S. S. Baghsorkhi, and J. Torrellas, "Save: Sparsity-aware vector engine for accelerating dnn training and inference on cpus," in *2020 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2020, pp. 796–810.

[33] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "Cvr: efficient vectorization of spmv on x86 processors," in *Proc. 2018 Int. Symp. Code Generation and Optimization*, 2018, p. 149–162.

[34] H. Zhang, R. T. Mills, K. Rupp, and B. F. Smith, "Vectorized parallel sparse matrix-vector multiplication in petsc using avx-512," *Proc. 47th Int. Conf. Parallel Process.*, 2018.

[35] X. Fei and Y. Zhang, "Regu2d: Accelerating vectorization of spmv on intel processors through 2d-partitioning and regular arrangement," *Proc. 50th Int. Conf. Parallel Process.*, 2021.

[36] Z. Xie, G. Tan, W. Liu, and N. Sun, "A pattern-based spgemm library for multi-core and many-core architectures," *IEEE Trans. Parallel Distrib. Syst.*, vol. 33, pp. 159–175, 2022.

[37] Y. M. Tsai, T. Cojean, and H. Anzt, "Sparse linear algebra on amd and nvidia gpus – the race is on," *High Perform. Comput.*, vol. 12151, pp. 309 – 327, 2020.

[38] D. Merrill and M. Garland, "Merge-based parallel sparse matrix-vector multiplication," in *SC16: Int. Conf. High Performance Computing, Networking, Storage and Analysis*, 2016, pp. 678–689.

[39] Y. Niu, Z. Lu, M. Dong, Z. Jin, W. Liu, and G. Tan, "Tilespmv: A tiled algorithm for sparse matrix-vector multiplication on gpus," *2021 IEEE International Parallel Distrib. Process. Symp. (IPDPS)*, pp. 68–78, 2021.

[40] Nvidia Corporation, "Cuda Toolkit 10.0." [Online]. Available: https://developer.nvidia.com/cuda-toolkit

[41] Y. Yang, J. Emer, and D. Sánchez, "Spzip: Architectural support for effective data compression in irregular applications," in *2021 ACM/IEEE 48th Annu. Int. Symp. Comput. Architecture*, 2021, pp. 1069–1082.

[42] M. Zhu, T. Zhang, Z. Gu, and Y. Xie, "Sparse Tensor Core: Algorithm and Hardware Co-Design for Vector-Wise Sparse Neural Networks on Modern GPUs," in *MICRO '52*. ACM, 2019, p. 359–371.

[43] N. Srivastava, H. Jin, J. Liu, D. H. Albonesi, and Z. Zhang, "Matraptor: A sparse-sparse matrix multiplication accelerator based on row-wise product," in *2020 53rd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2020, pp. 766–780.

[44] S. Pal, J. Beaumont, D. hyeon Park, A. Amarnath, S. Feng, C. Chakrabarti, H.-S. Kim, D. Blaauw, T. N. Mudge, and R. G. Dreslinski, "Outerspace: An outer product based sparse matrix multiplication accelerator," in *2018 IEEE Int. Symp. High Performance Comput. Architecture*, 2018, pp. 724–736.

[45] F. Sadi, J. Sweeney, T. M. Low, J. C. Hoe, L. T. Pileggi, and F. Franchetti, "Efficient spmv operation for large and highly sparse matrices using scalable multi-way merge parallelization," in *Proc. 52nd Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2019.

[46] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *2016 ACM/IEEE 43rd Annu. Int. Symp. Comput. Architecture*, 2016, pp. 243–254.

[47] M. Besta, R. Kanakagiri, G. Kwasniewski, R. Ausavarungnirun, J. Beránek, K. Kanellopoulos, K. Janda, Z. Vonarburg-Shmaria, L. Gianinazzi, I. Stefan, J. Gómez-Luna, M. Copik, L. Kapp-Schwoerer, S. D. Girolamo, M. Konieczny, O. Mutlu, and T. Hoefler, "Sisa: Set-centric instruction set architecture for graph mining on processing-in-memory systems," *MICRO-54: 54th Annu. IEEE/ACM Int. Symp. Microarchitecture*, 2021.

[48] S. Shi, Q. Wang, and X. Chu, "Efficient sparse-dense matrix-matrix multiplication on gpus using the customized sparse storage format," in *2020 IEEE 26th Int. Conf. Parallel and Distributed Syst. (ICPADS)*, 2020, pp. 19–26.

[49] E. Karimi, N. B. Agostini, S. Dong, and D. R. Kaeli, "Vcsr: An efficient gpu memory-aware sparse format," *IEEE Trans. Parallel Distrib. Syst.*, vol. PP, pp. 1–1, 2022.

[50] O. Zachariadis, N. Satpute, J. Gómez-Luna, and J. Olivares, "Accelerating sparse matrix-matrix multiplication with gpu tensor cores," *Comput. Electr. Eng.*, vol. 88, p. 106848, 2020.

[51] M. Li, Y. Ao, and C. Yang, "Adaptive spmv/spmspv on gpus for input vectors of varied sparsity," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, pp. 1842–1853, 2020.

Paul Scheffler received his BSc and MSc degree in electrical engineering and information technology from ETH Zurich in 2018 and 2020, respectively. He is currently pursuing a PhD in the Digital Circuits and Systems group of Prof. Benini. His research interests include hardware acceleration of sparse and irregular workloads, on-chip interconnects, manycore architectures, and high-performance computing.

Florian Zaruba received his BSc degree from TU Wien in 2014, his MSc and PhD from the Swiss Federal Institute of Technology Zurich in 2017 and 2021. He is a system architect at Axelera AI. His research interests include design of very large scale integration circuits and high performance computer architectures.

Fabian Schuiki received the BSc, MSc, and PhD degree in electrical engineering from the Swiss Federal Institute of Technology Zürich (ETHZ), in 2014, 2016, and 2021, respectively. He is a hardware compiler engineer at SiFive Inc. His research interests include transprecision computing as well as near- and in-memory processing.

Torsten Hoefler is a Professor of Computer Science at ETH Zürich, Switzerland. He received his PhD from Indiana University. Dr. Hoefler's interests are around performance-centric software and hardware development. He is a Fellow of the ACM and a member of the Academia Europaea.

Luca Benini holds the chair of digital Circuits and systems at ETHZ and is Full Professor at the Universita di Bologna. He received a PhD from Stanford University. Dr. Benini's research interests are in energy-efficient parallel computing systems, smart sensing micro-systems and machine learning hardware. He is a Fellow of the ACM and a member of the Academia Europaea.