Indirection Stream Semantic Register Architecture for Efficient Sparse-Dense Linear Algebra

Paul Scheffler*, Florian Zaruba*, Fabian Schuiki*, Torsten Hoefler†, Luca Benini*
* *Integrated Systems Laboratory, ETH Zurich*, Switzerland
† *Scalable Parallel Computing Laboratory, ETH Zurich*, Switzerland
{paulsc,zarubaf,fschuiki}@iis.ee.ethz.ch, htor@inf.ethz.ch, lbenini@iis.ee.ethz.ch

***Abstract*—Sparse-dense linear algebra is crucial in many domains, but challenging to handle efficiently on CPUs, GPUs, and accelerators alike; multiplications with sparse formats like CSR and CSF require indirect memory lookups. In this work, we enhance a memory-streaming RISC-V ISA extension to accelerate sparse-dense products through streaming indirection. We present efficient dot, matrix-vector, and matrix-matrix product kernels using our hardware, enabling single-core FPU utilizations of up to 80% and speedups of up to 7.2x over an optimized baseline without extensions. A matrix-vector implementation on a multi-core cluster is up to 5.8x faster and 2.7x more energy-efficient with our kernels than an optimized baseline. We propose further uses for our indirection hardware, such as scatter-gather operations and codebook decoding, and compare our work to state-of-the-art CPU, GPU, and accelerator approaches, measuring a 2.8x higher peak FP64 utilization in CSR matrix-vector multiplication than a GTX 1080 Ti GPU running a cuSPARSE kernel.**

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

I. Introduction

Sparse vector, matrix, and tensor operations pose a formidable challenge for today's processor architectures, which are optimized towards highly regular and vectorizable workloads. In sparse tensors, a significant fraction of elements is zero. A popular approach to reducing the runtime and memory requirements of sparse multiplications is to skip these zeros and process only nonzero elements. Consider the compressed sparse rows (CSR) format, which represents the rows of a sparse matrix A as a list of *nonzeros* A_vals and a list of their positions A_idcs. In this format, sparse matrix-vector multiplication (SpMV) can be implemented as follows:

```
for i in 0 to A_nrows:                          ] SpMV
  for j in A_ptr[i] to A_ptr[i+1]:   ] SpVV     ]
    y[i] += A_vals[j] * x[A_idcs[j]] ]          ]
```

where A_ptr delimits the rows of A. The SpMV consists of multiple sparse vector-vector multiplications (SpVVs), one for each element of the result vector y. Let us therefore focus on SpVV, which compiles to the following RISC-V assembly:

```
loop:
  lw       t0, 0(%A_idcs)          ] Address indirection
  slli     t0, t0, 3               ] x[A_idcs[j]]
  add      t0, t0, %x              ]
  fld      ft0, 0(%A_vals)         ] Memory access
  fld      ft1, 0(t0)              ] A_vals[j], x[...]
  addi     %A_idcs, %A_idcs, 4     ] Pointer increments
  addi     %A_vals, %A_vals, 8     ]
  fmadd.d  %y, ft0, ft1, %y        ] Computation
  bne      %A_vals, %A_end, loop   ] Loop branch
```

On a single-issue core, each loop iteration takes at least nine cycles to execute. Only one instruction performs necessary computation, using the floating-point unit (FPU) at most 11 % of the time. A superscalar core, even when ignoring dependencies, would need nine issue slots to achieve full FPU utilization.

The main issue is that the *indirection* x[A_idcs[j]], which accesses elements of the dense vector x, requires multiple integer instructions. The low FPU utilization a simple core achieves, and the significant instruction frontend complexity a superscalar core requires, severely limit our ability to execute sparse-dense operations in an area- and energy-efficient manner.

Efficient processing of sparse data structures is essential as sparsity is ubiquitous in many domains [1]; the SuiteSparse Matrix collection [2] curates sparse matrices from various real-world problems, covering physical simulation, mathematics, computer science, biology, and economics among other fields. Moreover, sparsification techniques in machine learning (ML) can significantly reduce the computational footprint incurred to attain a given accuracy [3]. These frontiers motivate new computer architectures that handle sparse problems efficiently.

Existing superscalar out-of-order architectures struggle significantly with the high control-to-compute ratio inherent to sparse formats, reaching only a fraction of their peak compute throughput. Consider the Intel Xeon Phi 7250 processor [4]: with a highly optimized machine-specific matrix compression scheme, it achieves 21 Gflop/s, a mere 0.7 % of its peak compute, and even less with the more common CSR format. Even worse, the extensive out-of-order capabilities necessary to process these matrices are not free: at a thermal design power (TDP) of 230 W and die sizes exceeding 500 mm^2, such a processor achieves very low area and energy efficiencies. To achieve high efficiency, it is crucial to maximize the silicon area and energy dedicated to FPUs, which actually do useful work, and that these units are highly utilized.

In this paper, we propose *indirection stream semantic registers* (ISSRs), an architectural extension to in-order, single-issue cores enabling high FPU utilization in sparse-dense multiplication *without* major impact on microarchitectural complexity. ISSRs are based on Schuiki et al.'s stream semantic registers (SSRs) [5], which offer a way to reach close to 100% FPU utilization with single-issue cores on dense data structures. This is achieved by allowing register reads/writes to implicitly encode a load/store operation together with an associated address computation. Zaruba et al.'s Snitch [6] shows that an extremely small 10 kGE core can fully utilize a large double-precision

100 kGE FPU when SSRs are combined with floating-point repetition (FREP) hardware loops, greatly boosting efficiency. Our work's key rationale is extending SSRs to support the indirection necessary for sparse-dense linear algebra, incurring only a small 4.4 kGE increase in hardware complexity. This allows us to rewrite our SpVV example as

```
  call    cfg_ssr_ft0  } Config. stream ft0 over A_vals[j]
  call    cfg_issr_ft1 } Config. stream ft1 over x[A_idcs[j]]
  mv      t0, %A_ptr[i+1]  ]  Compute loop trip count
  sub     t0, t0, %A_ptr[i] ]  A_ptr[i+1]-A_ptr[i]
  frep    t0, 1             }  Hardware loop over next 1 ins.
loop:
  fmadd.d %y, ft0, ft1, %y  }  Computation
```

where ft0 and ft1 now have stream semantics. Since address calculation, indirection, and load/stores are now fully handled in hardware, the loop body consists of only the fmadd.d instruction, which we execute using the frep hardware loop. Intuitively the expected speedup is 9×.

Since the indirection incurs additional memory bandwidth, the speedup achieved in practice depends on whether this traffic is multiplexed onto existing memory channels or whether additional channels are provided. We will show that in the former case, a single core can reach up to 80 % FPU utilization on SpMV using real-world matrices, yielding a 7.2× peak speedup over hand-optimized RISC-V assembly and 5.6× over the best implementation using dense SSRs. In an eight-core cluster with added memory contention and parallelization overheads, our extension still achieves a 5.8× speedup over our RISC-V baseline. To summarize, our contributions are as follows:

1) A lightweight extension to SSRs to handle streaming indirection, enabling highly efficient sparse-dense products in single-issue cores (Section II).
2) A programming model for the proposed ISSRs and corresponding efficient sparse-dense dot (SpVV), CSR matrix-vector (CsrMV), and CSR matrix-matrix (CsrMM) product implementations (Section III).
3) Significant performance and energy efficiency benefits, achieving up to 7.2× speedups on a single core and 5.8× and 2.7× gains on an eight-core cluster, respectively. We evaluate the impact of ISSR on area and power in a 22 nm technology, increasing the area of an eight-core cluster with SSRs by only 0.8 % (Section IV).
4) A comparison to state-of-the-art CPUs, GPUs, and sparse accelerator approaches, with ISSR achieving 2.8× higher peak floating-point utilization than GPUs (Section V).

II. Architecture

Our indirection hardware extension is mostly confined to the SSR's *address generator*; we will focus on its architecture before presenting the ISSR streamer and its integration into the Snitch cluster [6] in which we evaluate our work.

A. Address Generator

Fig. 1 shows our extended ISSR address generator. As in the SSR, it exposes a *shadowed* configuration register interface to the host core ❶, allowing the setup of a new job while another is running, and a control interface to the data mover ❷. It adds a read-only memory interface for index streaming ❸.

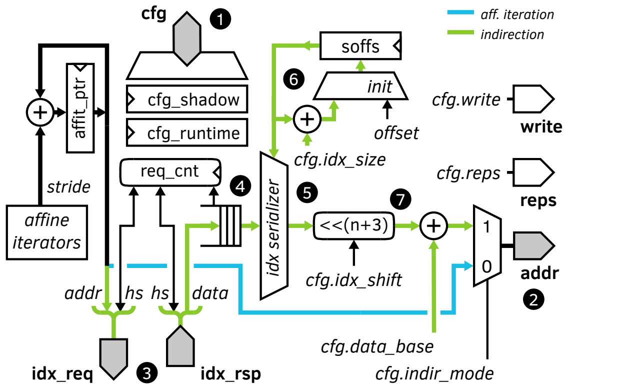


Figure 1: ISSR address generator architecture

The four nested SSR affine address iterators are left unchanged: at each emitted datum, the stride of the outermost iterating loop is added onto a shared memory pointer. In the existing *affine iteration* mode, this pointer directly provides addresses to the *streamed data* to the SSR's data mover. In the new *indirection* mode, it provides addresses to the *index array* indicating the data to be streamed: we must first fetch the desired indices from memory and add their corresponding offsets to a base address before emission to the data mover.

In indirection mode, we fix the affine iterator configuration to one dimension with a stride of eight bytes, loading a contiguous stream of 64-bit words from the shared, cluster-local tightly-coupled data memory (TCDM) into a decoupling FIFO ❹. An *outstanding request counter* regulates address emission to prevent downstream blocking or a FIFO overflow.

Our hardware can read arrays of either 32-bit or 16-bit indices. To this end, an *index serializer* ❺, backed by a two-bit *short offset* counter ❻, extracts 16- or 32-bit indices from the buffered 64-bit index words. To simplify the programming model, arbitrary index array alignment is supported.

The serialized indices are statically shifted to 64-bit word offsets to serve the double-precision FPU before being added to the preconfigured base address of the data array we index into ❼. Optionally, the indices can be shifted further by a programmable offset, enabling indirection into higher-level axes of power-of-two-strided tensors or tiles. This offers an attractive tradeoff between arbitrary striding, requiring a hardware multiplier, and no higher-level iteration support.

As with the affine iteration path, the indices and addresses used in the indirection logic have design-time-parameterizable widths between 16 and 32 bit, both defaulting to 18 bit to cover our default 256 KiB cluster TCDM.

B. ISSR Streamer

Fig. 2 shows the architecture of ISSR and its containing *streamer*. The streamer maintains the existing IO, exposing a shared configuration interface to the core Ⓐ, a register file interface to the FPU Ⓑ, and one port per SSR to the memory system Ⓒ. The switch maps each SSR to a specific architectural register while enabled Ⓓ.

Like the SSR, the ISSR provides a FIFO decoupling the register and memory streams, which is reused in both reading

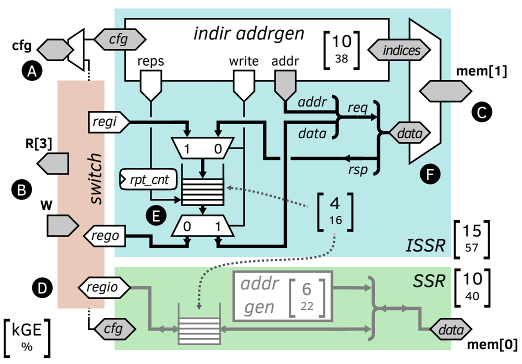


Figure 2: ISSR and ISSR streamer architecture

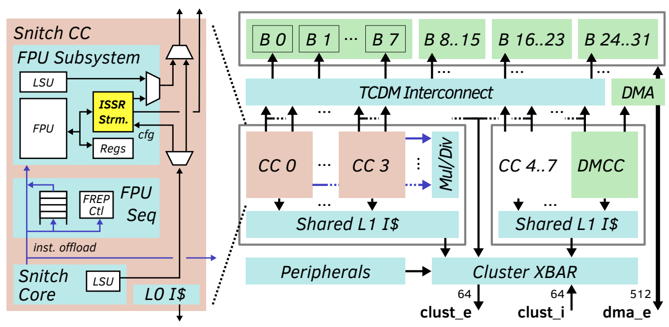


Figure 3: Snitch core complex (CC) and cluster architecture. Our ISSR streamer is highlighted in yellow and with bold font.

and writing E. In our design, the ISSR's index and data memory ports are combined by a round-robin multiplexer F. For each 16- or 32-bit index word, two or four 64-bit words are fetched, yielding a peak data mover utilization of $2/3$ or $4/5$, respectively. This arbitration could also be done on the streamer or core levels, or omitted completely by providing three ports per core, trading higher utilization and performance for approximately $1.5\times$ larger interconnect logic. In this work, we prioritize an area-optimized extension with one independent port per SSR, simplifying direct comparisons to the existing two-port, two-SSR streamer.

The presented streamer provides one ISSR and one regular SSR, but it could combine any number of SSRs and ISSRs given sufficient memory ports. Furthermore, the described indirection semantic does not change exception handling, which is described in detail by Schuiki et al. [5].

C. Core and Cluster Integration

Fig. 3 illustrates the streamer's integration into the existing Snitch core complex (CC) and cluster. Like the SSR streamer, it is part of the double-precision FPU subsystem in each CC and multiplexed with the floating-point register file.

We kept the existing CC memory topology, providing an exclusive port to the ISSR while combining the core, FPU, and SSR requests into another: this maximizes sparse-dense performance by giving the core opportunities for memory requests while the ISSR is fetching indices without blocking the SSR and impacting performance. Furthermore, this allows existing code to run without performance degradation.

The cluster contains eight worker CCs organized into two *hives*, sharing an L1 instruction cache and an integer multiply-divide unit. Our TCDM has 32 banks totaling 256 KiB. A 512-bit direct memory access (DMA) engine efficiently moves data blocks between the TCDM and main memory [7]. It is controlled by a lightweight data movement CC (DMCC) without an FPU, which can also be used for cluster coordination.

III. Programming and Applications

The ISSR's address stream is configured by the core through its memory-mapped register interface. Our backward-compatible register map encodes the job type in the register used to launch jobs, enabling few-to-single-cycle setups.

A. Accelerable Sparse Formats

Our main motivation for ISSRs is accelerating sparse-dense linear algebra. As already outlined, indirection enables the pairwise streaming of *nonzero-product operands* for any sparse tensor format whose major axis is given by two arrays: a *value array* storing nonzeros, and an *index array* storing their positions on the axis. We call such an array pair a *sparse fiber*.

In addition to inherently representing sparse vectors, sparse fibers form the base of the CSR [8] and compressed sparse columns (CSC) [9] matrix formats, concatenating the sparse rows or columns of a matrix as fibers and adding an array of pointers delimiting them. They also underlie the compressed sparse fiber (CSF) tensor format [10], generalizing this idea.

ISSRs therefore accelerate sparse-dense linear algebra with vectors, matrices, and general tensors in fiber-based formats; many format variations such as blocking and slicing can be supported through high-level iterators on the Snitch core.

B. Core Product Kernels

To evaluate our streamer, we created a set of single-core sparse-dense product kernels operating on sparse fiber vectors and CSR matrices. We implemented dot product (SpVV), CSR matrix-vector multiplication (CsrMV), and CSR matrix-matrix multiplication (CsrMM) kernels, each for 16- and 32-bit indices and in three variants:

- BASE: Stock RISC-V optimized baseline
- SSR: RISC-V with FREP using SSR
- ISSR: RISC-V with FREP using SSR and ISSR

Our BASE kernels are derived from the minimal indirection loop discussed in Section I; we compile C kernels with `gcc -O3` and hand-optimize the resulting assembly by instruction reordering and unrolling to minimize stalls. The SSR kernels stream the sparse vector values and are unrolled further where necessary to avoid stalls. Finally, we want to focus on our ISSR kernels which we will discuss in detail:

SpVV: Listing 1 outlines our ISSR SpVV kernel, with 16- and 32-bit index versions varying in the ISSR configuration registers used. We aim to issue a continuous stream of *fused multiply-add* instructions (`fmadd.d`) with minimal overhead to maximize FPU utilization. We achieve this in three steps:

i) *Setup*: we configure the SSR to stream the sparse vector's values and the ISSR to indirect into the dense vector's values at the sparse vector's indices. We enable redirection to

```
call       ssr_setup_ft0          # ft0: sparse a_vals
call       issr_setup_ft1         # ft1: dense b at a_idcs
csrsi      ssr_redir, 1           # redirect ft0, ft1 to SSRs
fcvt.d.w   ft2, zero              # zero-init ft2
...                               # zero-init ft3 .. ft[2+n]
frep       %len, n, 0b1001        # stagger accumulator n-fold
fmadd.d    ft2, ft0, ft1, ft2
...                               # reduce accumulators
fadd.d     ft8, ft6, ft7          # final reduction
fsd        ft8, 0(%res)           # store result
csrci      ssr_redir              # disable redirect
```

Listing 1: ISSR-accelerated SpVV kernel: SSR `ft0` streams sparse vector a and ISSR `ft1` values from dense vector b.

our streamer and initialize a contiguous set of accumulator registers, here starting at `ft2`, with zero.

ii) *Compute*: We loop over an `fmadd.d` instruction using FREP, streaming both operands from SSRs. To prevent stalls due to read-after-write register dependencies, we use FREP's *register staggering* feature to auto-increment the accumulator register number on each iteration, maintaining several partial sums. Due to its lower peak utilization, the 32-bit kernel requires fewer accumulators. The FREP register staggering is described in more detail in [6].

iii) *Teardown*: we additively reduce our accumulators and store the result before disabling SSR register redirection.

As the teardown contains only FPU subsystem instructions, the core is free to move on with execution after issuing all FPU instructions until data from the FPU is needed, enabling *pseudo-dual-issue* operation; synchronization with the FPU can be enforced with a dummy register move if needed.

CsrMV: While we could reuse our SpVV kernel for each matrix row, we further optimize the ISSR CsrMV kernel to maximize FPU utilization:

- We stream the entire matrix fiber in single SSR and ISSR jobs, significantly reducing setup overhead.
- We unroll the first few `fmadd` in each row with branches to shorter reductions for rows with few elements, issuing an FREP loop and a full reduction only when necessary.

All kernels use 32-bit row pointers, enabling broad scaling in rows. Our kernels primarily target CSR matrix multiplication from the left, but support power-of-two vector and arbitrary result strides, enabling multiplication of any power-of-two-strided dense axis with a CSR or CSC matrix from either side.

CsrMM: We multiply a CSR matrix with a power-of-two-column, dense row-major matrix and produce a dense row-major output. We reuse our CsrMV kernels, iterating on the dense matrix and result along their columns. Since we now iterate over dense columns in a third-order loop, the overhead over our CsrMV kernels is small to negligible.

As with our CsrMV kernels, we can specify custom dense matrix and result strides, enabling products between row- and column-major matrices with CSR or CSC matrices from either side. Our main restriction is a power-of-two stride on the indirected dense axis due to our shifter tradeoff in the address generator; in practice, this can be accommodated by tiling matrices into the TCDM using the cluster DMA's 2D transfers.

C. Further Indirection Applications

While ISSRs accelerate sparse-dense products and related operations like accumulating sparse onto dense tensors, indirection is a *general-purpose* operation accelerating many more problems, some of which we want to highlight here:

Codebook decoding: ISSRs can stream codebook-compressed data, representing arrays with repeated values as a series of indices pointing to a compact value array. Codebooks can be used in the quantization of image components [11], deep learning weights and activations [12], or to compress nonzeros in sparse matrices with few unique values among other uses.

A single ISSR can stream a codebooked vector; a streamer with two ISSRs could accelerate sparse-dense products with codebook-compressed sparse values with near-identical code and performance as our existing kernels; this could strongly reduce the memory footprint of compressible tensors.

Improved convolutions: Accelerators often map dense convolutions to matrix multiplications through data transforms [12]. SSRs can accelerate convolutions with *rectangular* stencils without such transforms; ISSRs could extend this capability to *arbitrarily-shaped* sparse stencils by streaming an offset index array providing the stencil's shape and incrementing the data base address on the core.

Scatter-gather streaming: ISSRs are, in effect, streaming *scatter-gather units* as found in vector processors. As such, they could be used to accelerate scatter-gather algorithms including parallel radix sort [13], sparse matrix transpose [14], and the densification of sparse tensors by nonzero scattering.

IV. Results

We will first present the performance of our kernels on a single core and in a cluster CsrMV implementation. We will then discuss the area and timing impacts as well as the energy efficiency benefits for our ISSR streamer.

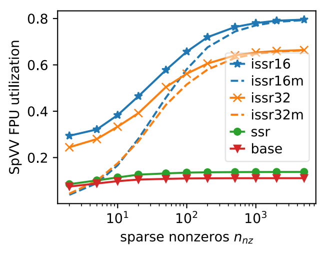
For all experiments, we obtained dense test tensors by sampling normally-distributed values. Sparse vectors were generated with normally-distributed values and uniformly-distributed indices given a fixed nonzero count and dimension. The sparse matrices used are all real-world-problem matrices from the SuiteSparse Matrix Collection [2]: they have 2k to 3.2k columns, 1.3k to 680.3k nonzeros, varying aspect ratios, and cover various problem domains.

A. Performance: Single Core

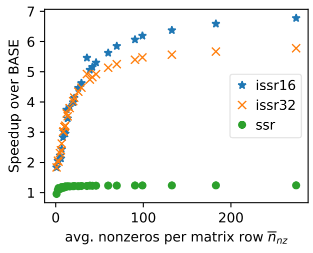
To measure the architectural benefits of our ISSR streamer without the influence of other system components, we evaluate our product kernels in RTL simulation of a single CC by coupling it to ideal single-cycle instruction and two-port data memories. The latter behave similarly to the instruction cache and TCDM in a cluster, except for misses and bank conflicts whose effects are included in Section IV-B.

SpVV: Fig. 4a shows the FPU utilization for our dot product kernels, excluding load-store operations idling the datapath, against the sparse vector's nonzero count n_{nz}. We note that the runtime is independent of the dense vector's size as long as it fits into the TCDM. Non-ISSR kernels perform identically for 16- and 32-bit indices as we cannot optimize halfword index loads. ISSR kernel FPU utilizations are shown including and excluding accumulator reductions (m suffix).

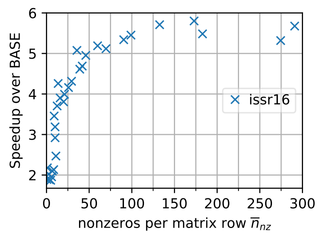
Both 16- and 32-bit ISSR kernels significantly outperform BASE and SSR kernels, increasing utilization by up to 4.7× and



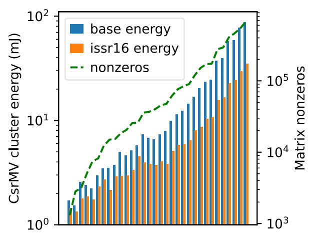
(a) CC SpVV FPU utilizations



(b) CC CsrMV speedups



(c) ISSR cluster CsrMV speedup



(d) Cluster CsrMV energies

Figure 4: Sparse-dense product kernel results

5.6× over the latter and approaching their arbitration-imposed upper utilization limits of 0.8 and 0.67. Note that BASE and SSR kernels also approach their utilization limits of one multiply-accumulate every nine and seven cycles, respectively.

Because ISSR kernels process each nonzero significantly faster than others, they require a proportionally higher n_{nz} to overcome their SSR configuration and accumulator reduction overhead. For $n_{nz} < 5$, their reduction-free utilization is even lower than in non-ISSR kernels, motivating our CsrMV row unrolling in such cases. As the 16-bit ISSR kernel needs more accumulators to sustain peak utilization, it outperforms the 32-bit variant only at higher n_{nz}.

CsrMV: Fig. 4b shows the speedup of CsrMV kernels over the BASE kernel against the average nonzeros per matrix row $\overline{n}_{nz}$, reflecting the inner loop iterations. Our experiments assume that the TCDM is large enough to store the full matrix.

As for SpVV, our ISSR kernel speedups over BASE approach their theoretical 7.2× and 6.0× limits, requiring a higher $\overline{n}_{nz}$ than slower kernels to compensate for their overhead. Again, the 16-bit ISSR kernel outperforms its 32-bit counterpart only past $\overline{n}_{nz} \approx 20$ due to its longer reduction.

CsrMM: speedups and utilizations are near identical to the CsrMV kernels we use, even in edge cases: for a tiny sparse matrix like *Ragusa18* with only 64 nonzeros, FPU utilization changes by only 0.12 % for a 2-column dense matrix.

B. Performance: Cluster

To evaluate performance on a system level, we implement multicore CsrMV on a Snitch cluster, reusing our single-core kernels, *distributing rows* among cores, and employing a *double-buffered* data movement scheme for the matrices using the cluster DMA. Our cluster is served by a 512-bit duplex main memory modeled as ideal. All data initially resides in main memory and results are written back to it.

Fig. 4c shows the speedup of the 16-bit ISSR kernel over the BASE kernel in cluster CsrMV. Speedups are significant even for $\overline{n}_{nz} = 1$ at 1.9× and reach up to 5.8×, sustaining over 5× for $\overline{n}_{nz} > 50$. They follow the same trend as single-CC CsrMV with *reduced speedup* and *stronger variations* from trends, which we attribute to multiple aspects:

Most notably, TCDM *bank conflicts*, accented by the random bank access patterns of indirection, lower peak FPU utilization from 0.8 to 0.71. Additionally, the *initial vector transfer* cannot be fully overlapped with computation, modulating speedups with the vector length. Finally, our *row distribution* and *double-buffering* schemes cannot fully prevent computation imbalance and overhead, and we encounter some instruction cache stalls. Despite these inherent parallelization challenges, ISSRs enable extreme efficiency gains with minimal area overhead: *eight* cores with ISSRs achieve the same peak floating-point throughput as *46* cores running BASE.

C. Area and Timing

We synthesize the SSR and ISSR streamers using Synopsys *Design Compiler* for GlobalFoundries' 22 nm FD-SOI technology, targeting the SSG corner at −40 °C with low-V_t cells, 0.72 V supply voltage, and no back-biasing. A 1 GHz clock and 100 ps IO delays were constrained. Our hardware configuration has one SSR and one ISSR with five data FIFO stages, 18-bit indices and addresses, and four affine loops.

Compared to the SSR, the ISSR's longest path increased from 301 ps to 425 ps, still easily meeting Snitch's 1 GHz clock target. The ISSR streamer's hierarchical area distribution is denoted in Fig. 2. Our ISSR is 4.4 kGE or 43 % larger than the equivalently parameterized SSR, incurring only a negligible 0.8 % area overhead in our eight-core Snitch cluster compared to providing only SSRs.

D. Energy and Power

We estimate the power consumption of a Snitch cluster running CsrMV with BASE and 16-bit ISSR kernels on our matrix set, targeting the TT corner in GlobalFoundries' 22 nm FD-SOI at 1 GHz. We use Synopsys *Design Compiler* to topographically synthesize our cluster and Synopsys *PrimeTime* to estimate power for the low- and high-efficiency matrices `G11` and `G7`, then scale dynamic power with hardware component utilizations measured in RTL simulation for all matrices.

Fig. 4d shows the total energy for CsrMV using both kernels for each matrix. While the peak average cluster power is predictably lower for the BASE kernel (89 mW vs. 194 mW), ISSRs achieve energy efficiency improvements of up to 2.7× (142 pJ to 53 pJ per `fmadd`).

V. Related Work

General-purpose processors: ISSRs are most related to *scatter-gather* hardware in vector processors, which is slowly adopted by superscalar out-of-order architectures: Intel's Knights Landing (KNL) [15] and Arm's Scalable Vector extensions [16] introduce single-instruction, multiple-data (SIMD) scatter-gather. However, SIMD vectors are only *a few elements* wide, severely limiting their applicability to sparsity compared to ISSRs. Xie et al. [4] optimize SpMV on KNL with their own sparse format, improving SIMD lane and cache usage efficiency over the state of the art. Still, they use at most 0.7 % of their system's peak double-precision compute, 70× less than what we achieve in a Snitch cluster with ISSRs.

GPUs: sparse-dense operations on GPUs are often approached in *software* through efficient algorithms and sparse formats: Shi et al. [17] accelerate sparse matrix-matrix products with their own matrix format and algorithm. Merrill et al. [18] propose CsrMV using a merge-based decomposition to improve performance consistency. Nvidia's *cuSPARSE* library [19] provides optimized sparse problem kernels. We evaluate their CsrMV kernels from CUDA Toolkit 10.0 on a GTX 1080 Ti GPU (Pascal GP104 architecture, FP32 and FP64) and a Jetson AGX Xavier (Volta architecture, FP32 only). We reuse the matrices from our evaluation to profile 100 consecutive kernel runs using *nvprof* [19]. For FP32, both platforms exhibit high peak streaming multiprocessor (SM) occupancy (87 % and 96 %), but low peak floating-point utilizations (0.75 % and 2.1 %) on active SMs, suggesting low thread parallelism among warps. FP64 shows similar SM occupancies with a notably higher peak floating-point utilization (17 %), likely due to the 32× fewer FP64 cores per SM. Still, peak utilization is 2.8× lower than in a Snitch cluster with ISSRs.

More recently, GPU *hardware* targeting sparsity has been proposed: Zhu et al. [20] present an algorithm and tensor core modifications for efficient sparse neural network inference. Nvidia's A100 architecture [21] supports *structured* sparsity, efficiently handling up to two zeros in every four values. This imposed structure covers only a small subset of our ISSRs' capabilities, which efficiently handle a much wider range of sparsities ($\gg 50\,\%$) and random accesses into a full TCDM.

Hardware accelerators: Sparse accelerators often target specific domains like deep learning (DL) [12]. Direct comparisons to our work are hard since most accelerators use highly application-specific data precisions and formats and are incapable of general-purpose computation. Nevertheless, a programmable Snitch cluster with ISSRs can implement many highly specialized computation schemes used in hardwired accelerators. Han et al. [22] keep entire pruned, weight-shared DL models in SRAM, minimizing DRAM access energy; ISSRs work with any TCDM stationarity scheme and enable code-booked weight reuse. Hedge et. al [23] hierarchically intersect sparse tensor indices to avoid zero products and their memory traffic; a Snitch core could intersect upper tensor axes while its ISSR-fed FPU processes nonzeros.

VI. Conclusion

In this work, we extend SSRs for *streaming indirection*, creating a backward-compatible ISSR which enables efficient sparse-dense tensor products with sparse-fiber-based formats including CSR and CSF. We demonstrate it in the RISC-V Snitch system by presenting efficient dot product, CsrMV, and CsrMM kernels, and propose further indirection applications.

We evaluate our ISSR streamer by comparing ISSR product kernels to optimized-baseline and SSR-only kernels on a single CC; we improve peak dot product FPU utilization from 0.11 (BASE) or 0.14 (SSR) to 0.67 and 0.80 for 16- and 32-bit indices, and enable CsrMV speedups of up to 7.2× over an optimized baseline. In a multicore cluster CsrMV implementation, speedups over our baseline reach up to 5.8× despite work sharing, vector transfer, and bank conflict overheads. The default ISSR is only 4.4 kGE or 43 % larger than an SSR with a longest path of 425 ps in GF22FDX technology, but improves cluster CsrMV energy efficiency by up to 2.7× over our baseline. A Snitch cluster with ISSRs provides wider sparse-dense support and higher floating-point utilizations (70× and 2.8×) than recent CPUs and GPUs, yet is flexible enough to adopt key innovations of sparse accelerators in software.

References

[1] Z. Zhang, Y. Xu, J. Yang, X. Li, and D. Zhang, "A Survey of Sparse Representation: Algorithms and Applications," *IEEE Access*, vol. 3, pp. 490–530, 2015.

[2] T. A. Davis and Y. Hu, "The University of Florida Sparse Matrix Collection," *ACM Trans. Math. Softw.*, vol. 38, no. 1, Dec. 2011.

[3] T. Gale, E. Elsen, and S. Hooker, "The state of sparsity in deep neural networks," arXiv:1902.09574 [cs.LG], 2020.

[4] B. Xie, J. Zhan, X. Liu, W. Gao, Z. Jia, X. He, and L. Zhang, "CVR: efficient vectorization of SpMV on x86 processors," in *CGO '18*, 2018, pp. 149–162.

[5] 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.*, pp. 1–1, 2020.

[6] 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.*, pp. 1–1, 2020.

[7] A. Kurth, W. Rönninger, T. Benz, M. Cavalcante, F. Schuiki, F. Zaruba, and L. Benini, "An Open-Source Platform for High-Performance Non-Coherent On-Chip Communication," arXiv:2009.05334 [cs.AR], 2020.

[8] 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.

[9] 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.

[10] S. Smith and G. Karypis, "Tensor-Matrix Products with a Compressed Sparse Tensor," in *IA^3 '15*. ACM, 2015.

[11] 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 Processing: Image Communication*, vol. 17, no. 1, pp. 73 – 84, 2002.

[12] 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," arXiv:2007.00864 [cs.AR], 2020.

[13] M. Zagha and G. E. Blelloch, "Radix sort for vector multiprocessors," in *SC '91*. ACM, 1991, p. 712–721.

[14] P. Stathis, D. Cheresiz, S. Vassiliadis, and B. Juurlink, "Sparse matrix transpose unit," in *1IPDPS '04*, 2004, pp. 90–.

[15] 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.

[16] 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.

[17] S. Shi, Q. Wang, and X. Chu, "Efficient Sparse-Dense Matrix-Matrix Multiplication on GPUs Using the Customized Sparse Storage Format," arXiv:2005.14469 [cs.DC], 2020.

[18] D. Merrill and M. Garland, "Merge-Based Parallel Sparse Matrix-Vector Multiplication," in *SC '16*, 2016.

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

[20] 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.

[21] 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

[22] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "EIE: Efficient Inference Engine on Compressed Deep Neural Network," *SIGARCH Comput. Archit. News*, vol. 44, no. 3, p. 243–254, Jun. 2016.

[23] K. Hegde, H. Asghari-Moghaddam, M. Pellauer, N. Crago, A. Jaleel, E. Solomonik, J. Emer, and C. W. Fletcher, "ExTensor: An Accelerator for Sparse Tensor Algebra," in *MICRO '52*. ACM, 2019, p. 319–333.