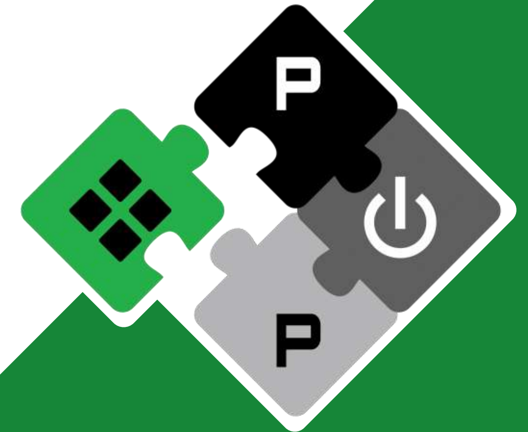
ETH zürich ALMA MATER STUDIORUM UNIVERSITÀ DI BOLOGNA

Accelerating Sparse and Irregular Workloads with Stream Extensions

Integrated Systems Laboratory (ETH Zürich)

Paul Scheffler paulsc@iis.ee.ethz.ch
Chi Zhang chizhang@iis.ee.ethz.ch



PULP Platform
Open Source Hardware, the way it should be!

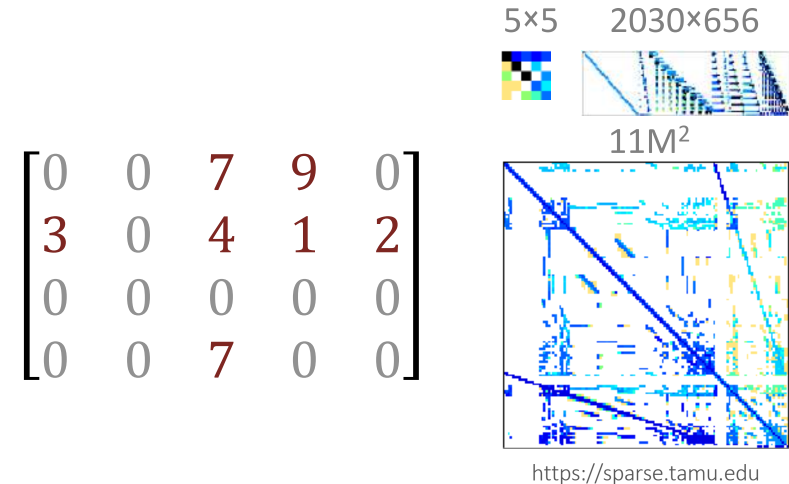
@pulp_platform

pulp-platform.org

youtube.com/pulp_platform

Sparsity and Irregular Workloads

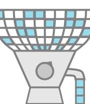
- *Sparse*: describes **tensors with many zeros**
  - Usually arise from graph abstractions
  - *Everywhere:* ML, physics, economics, math, …
- *Irregular*: describes ***messy* workloads**
  - Noncontiguous memory accesses, complex control flow
  - Often data-dependent and low reuse
  - Sparse LA, stencil codes, graph matching, …
- **Challenging** for SoA hardware
  - *CPUs & GPUs*: inefficient (~1-10% FP util.)
  - *Accelerators*: limited capabilities or high cost
  - *Stream extensions*: simple, cheap, flexible… 🤔



$$\begin{bmatrix} 0 & 0 & 7 & 9 & 0 \\ 3 & 0 & 4 & 1 & 2 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 7 & 0 & 0 \end{bmatrix}$$

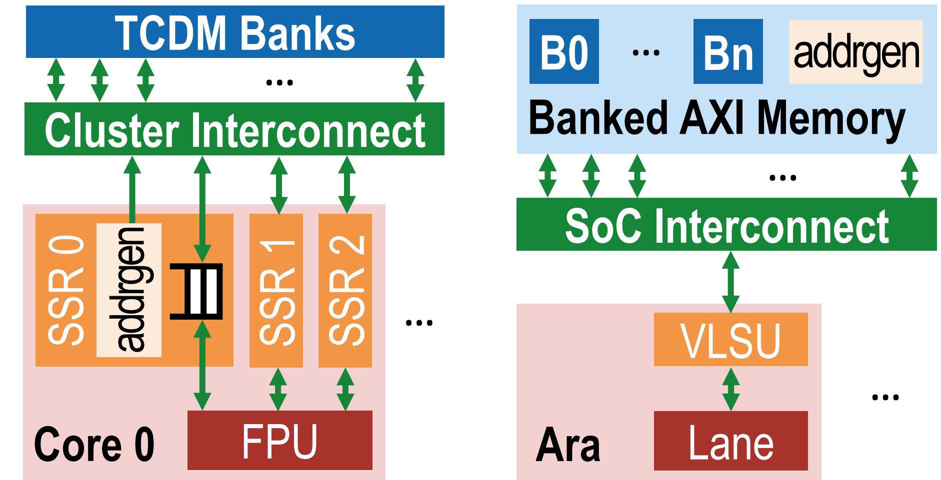
TOP500 rank	HPL Tflop/s	HPCG Tflop/s	HPCG util.
52	9991	748	5.6%
69	7893	529	5.0%
156	3250	203	4.7%

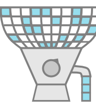
https://hpcg-benchmark.org/custom/index.html%3Flid=155&slid=313.html



Streams To The Rescue

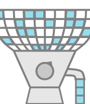
- *Streams*: ***sequences* of memory accesses**
  - Generate addresses in HW
  - Decouple accesses from execution
  - Feed data directly to/from compute units
- Notably accelerate *regular* workloads
  - Clusters (Snitch, PULP, …): *stream semantic registers*
  - SoC interconnects (iDMA, Ara, …): *burst transfers*
  - **Near ideal (>80%) BW & compute utilizations**
- Can streams accelerate *irregular* workloads?
  - Strided, indirect, *arbitrary* address sequences
  - Joins (intersection & union) of streams
  - Cluster- and SoC-level acceleration





Outlook: Irregular Streams throughout PULP

- Clusters: **Sparse Stream Semantic Registers**
  - Indirect streams: *sparse-dense LA*
  - Indexed stream joins: *sparse-sparse LA*
  - Arbitrary stream patterns: *stencil codes*
- SoC level: **AXI-Pack**
  - Efficient strided and indirect bursts
  - Banked SRAM and Ara extension
  - DRAM endpoints and coalescing
- Next steps: **scaling out**
  - SSSR multi-cluster workloads
  - AXI-Pack extensions and NoC transport

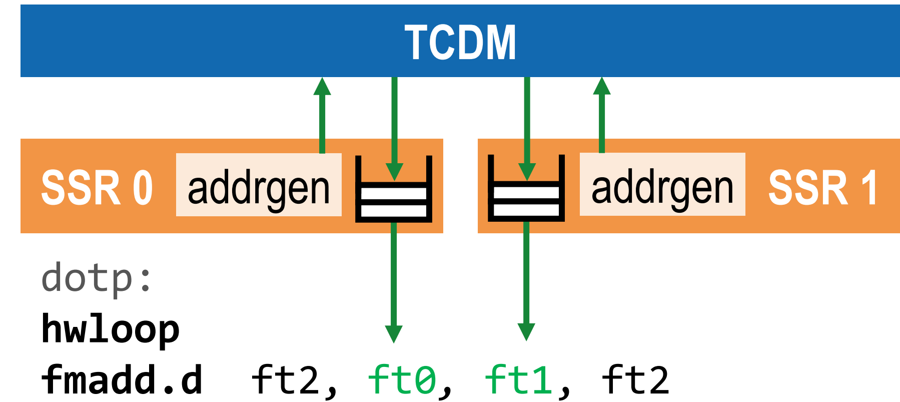


Stream Semantic Registers

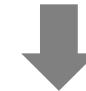
- Map FP reg accesses to streams
  - *Affine* address generators
- **Cannot accelerate irregular workloads**
  - Sparse-dense LA: *indirect* streams
  - Sparse-sparse LA: *indexed stream joins*
  - Non-LA (e.g. stencils): *arbitrary sequences*

➢ **Extend SSRs** for irregular streaming

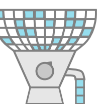
- Sparse LA: use *fiber-based tensor formats*
  - Lowest tensor axis: **value + index array**
  - ✓ *Scalable*: only nonzeros stored
  - ✓ *Widespread*: CSR, CSC, CSF, DCSR, ...



$\vec{a} = [0 \quad 0 \quad 5 \quad 6 \quad 0 \quad 13 \quad 0 \quad 0 \quad 7]$

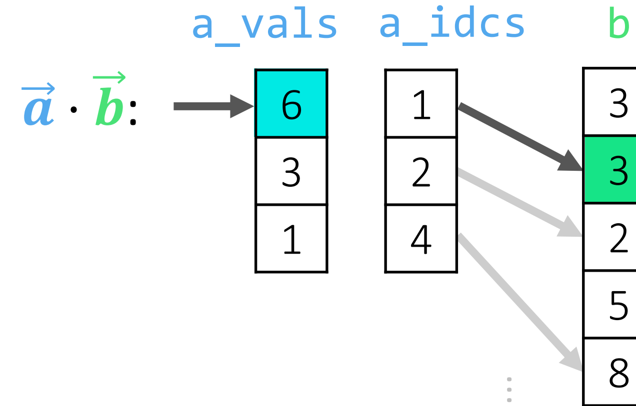


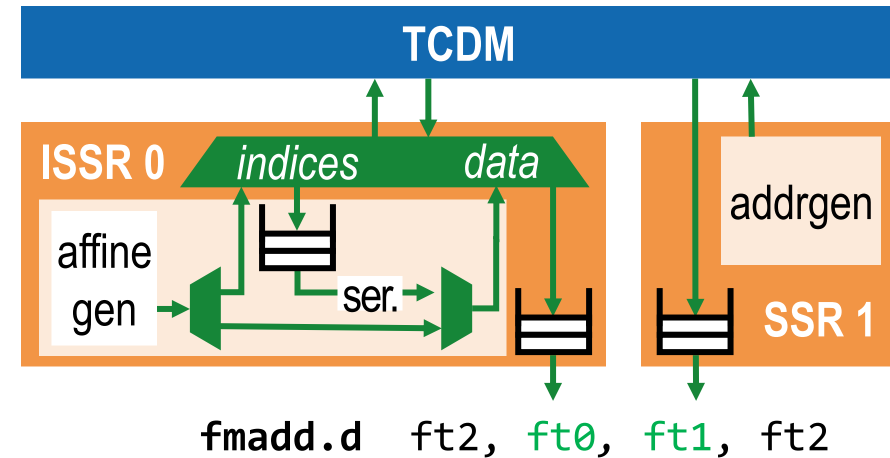
```
a_vals[4] = {5, 6, 13, 7}
a_idcs[4] = {2, 3, 5, 8}
```

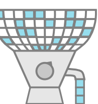



Sparse-Dense LA: Indirection SSR

- Only **nonzeros** impose work
  - Read *sparse*, fetch *needed* dense elements
  - **Streaming indirection** into b using a_idcs
  - Also underlies $\vec{a} + \vec{b}$, other irregular ops!
- ➢ Extend SSR to perform indirection
  - Emit indirect b values directly
  - Stream a_vals linearly with normal SSR
  - Continuous MAC issues like dense case!
- Fetch packed indices using shared port
  - Efficient handling of 8, 16, 32b indices

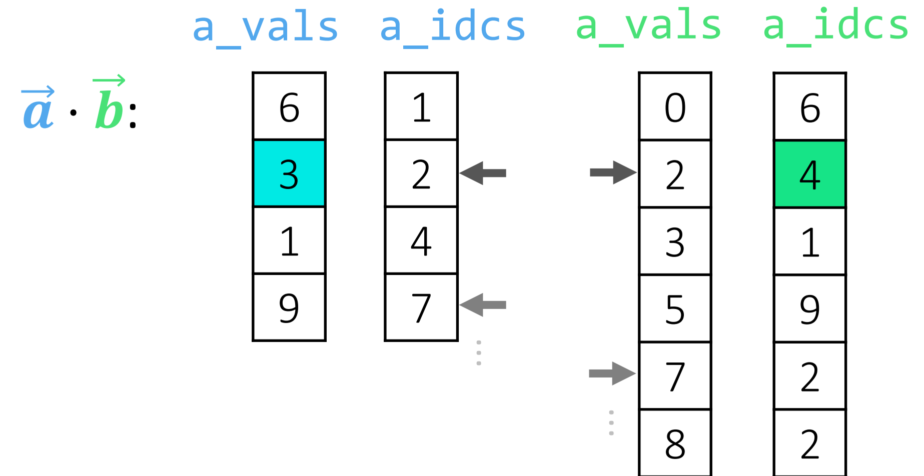


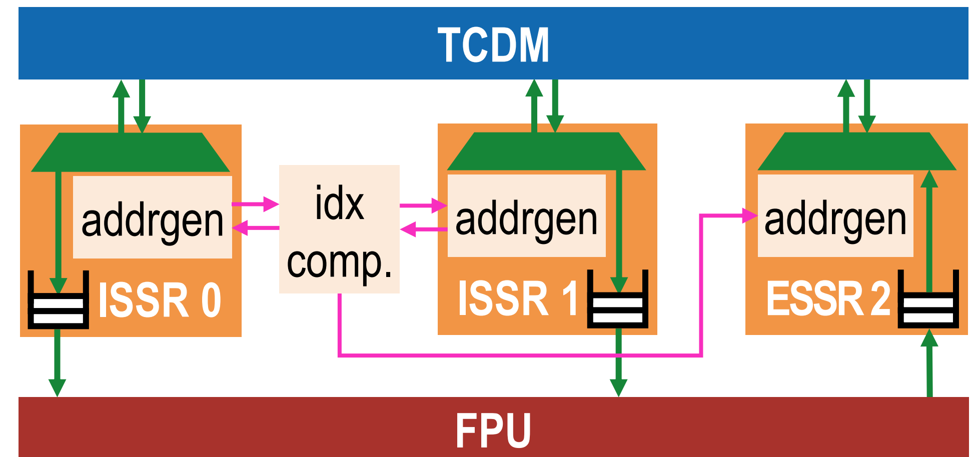


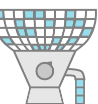


Sparse-Sparse LA: Sparse SSRs

- Cannot indirect: *offset ≠ position*
  - Must **compare indices** to find useful work
  - $\vec{a} \cdot \vec{b}$: index **intersection**
  - $\vec{a} + \vec{b}$: index **union**
- Reuse index HW, add **comparator**
  - *Intersection*: emit index-matching NZs
  - *Union*: emit matches *or* either NZ with 0
- Write sparse vectors with **egress SSR**
  - Write comp. indices *with* data
  - Backward-compatible like ISSR





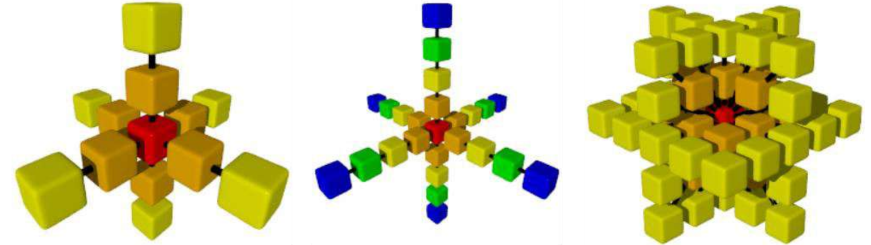


Non-LA Workloads: *We have it already*

- SSSR indirection & joins **general purpose**
  - ISSRs can stream *arbitrary* index arrays!
- *Stencils:* iterative operations traversing grids
  - Often complex shapes, multiple I/O arrays
  - Require unrolling for high efficiency
  - Don't (efficiently) map to affine streams

➢ **Map grid point accesses to ISSR pair**

- *Before iteration*: Store point addresses for grid point $\vec{0}$ as "indices" (array bases included)
- *During iteration:* Use point offset as base address → index array is static for all points



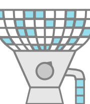
C. R. Yount, *"Vector Folding: Improving Stencil Performance via Multi-dimensional SIMD-vector Representation"*, HPCC 17

```
/* Static setup */

idcs0 = [ ... ];
idcs1 = [ ... ];

for (t ..) {
  for (z ..) {
    for (y ..) {
      for (x ..) {

        point = idx(x, y, z);
        issr0_launch(point, idcs0);
        issr1_launch(point, idcs1);
        /* Point loop using ISSRs */
}}}}
```

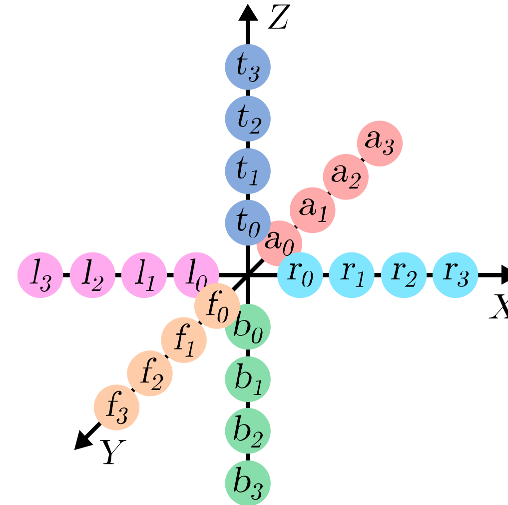



An Example Stencil: Minimod Acoustics (1)

- Seismic simulation stencil
  - Compute wavefield $\mathbf{u_t}(\vec{x})$ given stimulus $\mathbf{f_t}(\vec{x})$
  - 25 points, 16 coefficients, 3 in & 1 out array

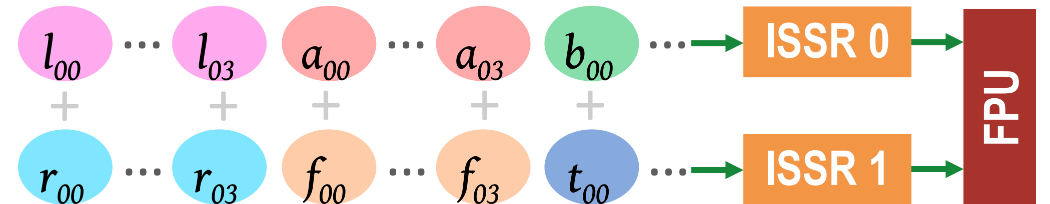
1. **Map ins, outs, coefficients** to SSRs
   - 2 ISSRs stream *inputs* from $\mathbf{u_{t-1}}$, $\mathbf{u_t}$
   - Load $\mathbf{f_t}$ with LSU (no halo, few elements)
   - 16 *coefficients* $\mathbf{c}_*$ → tile in FP regs
   - 3rd SSR streams *outputs* to $\mathbf{u_{t+1}}$

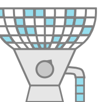
2. Define ISSR indices for **ideal performance**
   - Unroll to avoid FPU stalls → 2×2 output tile
   - Line up *pairs* of FPU operands as consumed



In: prior fields $\mathbf{u_{t-1}}$, $\mathbf{u_t}$, stimulus $\mathbf{f_t}$
Out: new field $\mathbf{u_{t,1}}$

foreach p **in** grid **do**
  $L \leftarrow c_f \cdot \mathbf{f_t}(p)$
  for $i \leftarrow 0$ **to** 4 **do**
    $L \leftarrow L +$
    $c_{x,i} \cdot (\mathbf{u_t}(p + l_i) + \mathbf{u_t}(p + r_i)) +$
    $c_{y,i} \cdot (\mathbf{u_t}(p + a_i) + \mathbf{u_t}(p + f_i)) +$
    $c_{z,i} \cdot (\mathbf{u_t}(p + b_i) + \mathbf{u_t}(p + t_i))$
  end
  $\mathbf{u_{t,1}}(p) \leftarrow L - \mathbf{u_{t-1}}(p)$
end





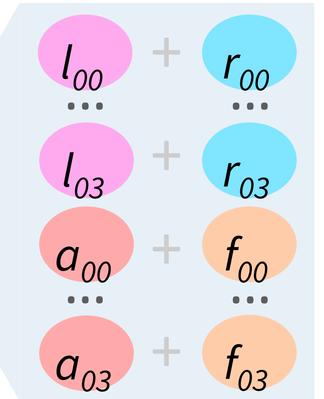
An Example Stencil: Minimod Acoustics (2)

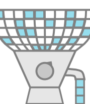
3. Configure and **launch SSRs**
   - Configure once, launch N times
   - ISSRs: launch *once per grid point* ($\vec{p}$ as base)
   - 3rd SSR: launch *once per X axis* → noncritical
   - Vary X trip count to reduce core imbalance

```
idcs0 = [ ... ];
idcs1 = [ ... ];
ssrs_configure(idx_size, dims, ...);
for (z ..) {
  for (y ..) {
    ssr3_launch_3D(2,sx, 2,sy, nx,sc);
    for (x ..) {
      issrs_launch(p, idcs0, idcs1);
      /* Point loop */
}}}
```

4. **Write point loop** using ISSRs
   - Use FREP loops for unroll
     - Hide ISSR setup, reduce I$ pressure
     - ≥8 FP regs left → combine axes for longer loops
   - Hide ISSR index fetching with accumulation

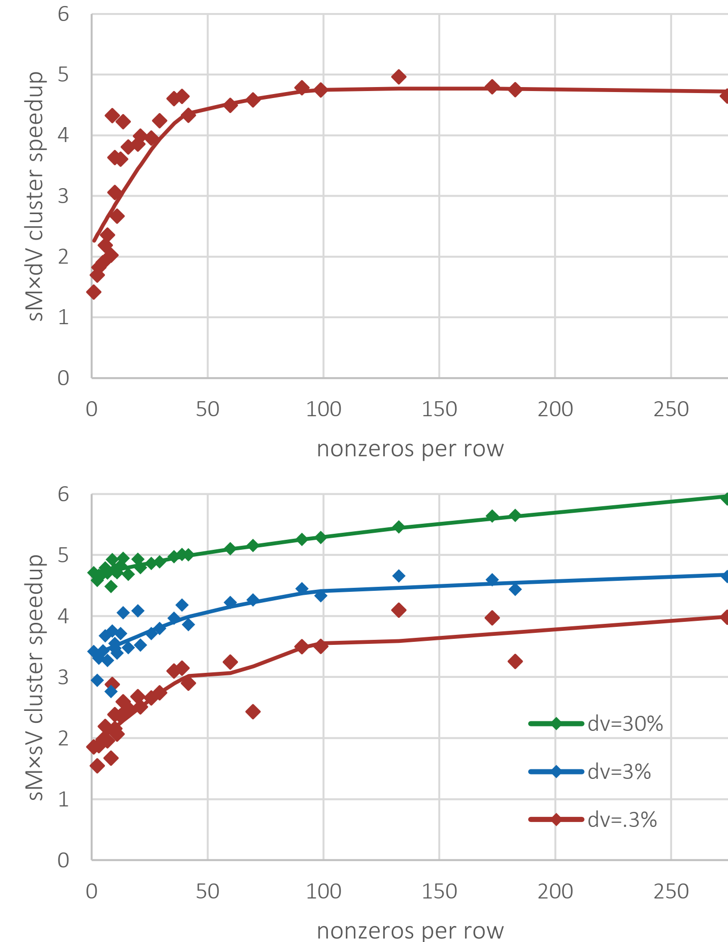
```
// f24...f27 = l0* + r0*
// f28...f31 = a0* + f0*
frep.i    %const7, 1, 7, 0b001
fadd.d    f24, f0, f1
// f20..f23 += cx0 * f24..f27
// f20..f23 += cy0 * f28..f31
frep.i    %const3, 2, 3, 0b1101
fmadd.d   f20, %cx0, f24, f20
fmadd.d   f20, %cy0, f28, f20
...
```

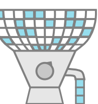




SSSR Results: Sparse Linear Algebra

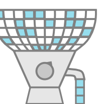
- **Fast** and **efficient** *sparse-dense LA*
  - Single core M×V: up to **7.0×** faster, **80%** FPU util.
  - Cluster M×V: up to **5.0×** faster, **2.9×** less energy
- **Significant speedups** in *sparse-sparse LA*
  - Single core: up to **7.7×** add, **7.9×** multiply
  - Cluster M×V: up to **5.9×** faster, **3.0×** less energy
- **Low impact** on area and timing
  - *One ISSR*: **+3 kGE** to base SSRs
  - *Full SSSRs:* **+11 kGE** per core (**+1.8%** to cluster)
  - *No* core timing impact





SSRR Results: Stencil Codes

- Various 2D/3D stencils on Snitch cluster
  - FP64, $64^2/16^3$ grid chunks, up to 4× unroll
  - LLVM RV32G baseline vs ISSR-enhanced kernels
- Geomean **2.7×** speedups, **82%** FP utilization
  - ISSR **IPC consistently >1** as ISSRs enable pseudo-dual-issue
- Baseline perf. degrades for large (3D) stencils
  - Cannot maintain unroll and keep reusable inner-loop data in register file
  - ISSR streams **avoid this bottleneck**: 2.5× 2D → 3.2× 3D geomean speedup

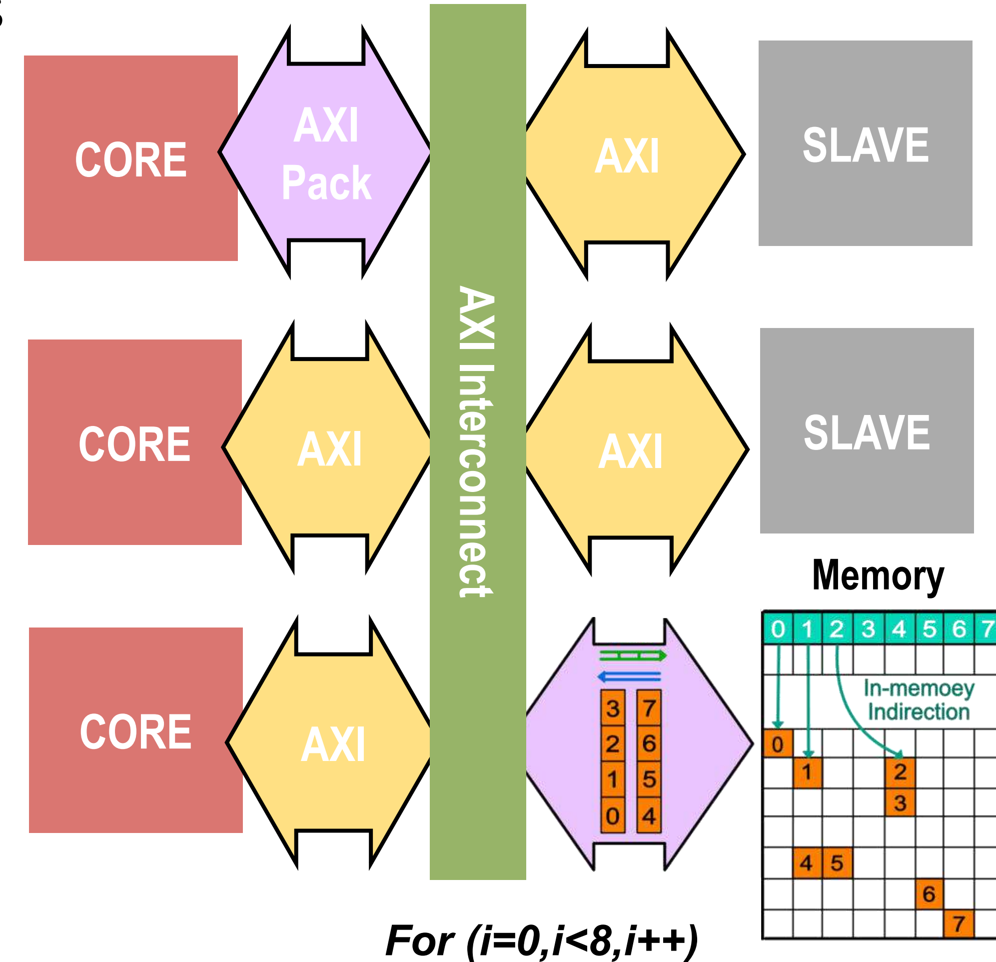


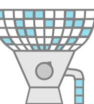
When Problem Size Gets Bigger …

- Streaming sparse/irregular data through wide bus
  - Poor utilization of bus bandwidth
  - Longer latencies

➢ **Extend AXI** for efficient irregular streaming

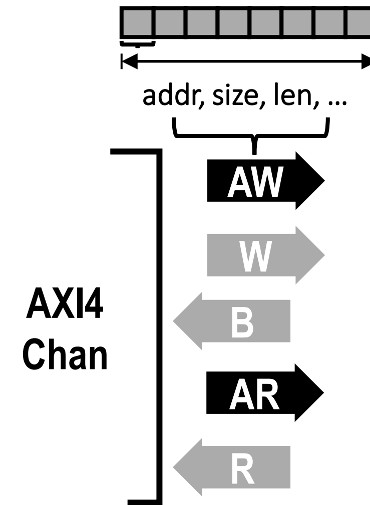
- AXI-Pack:
  - Core-side issues ***pattern-aware requests***
  - Memory-side response ***densely packed data stream***
- Features:
  - High bus utilization
  - Compatible and transparent
  - Scalable
  - Stride + indirect stream

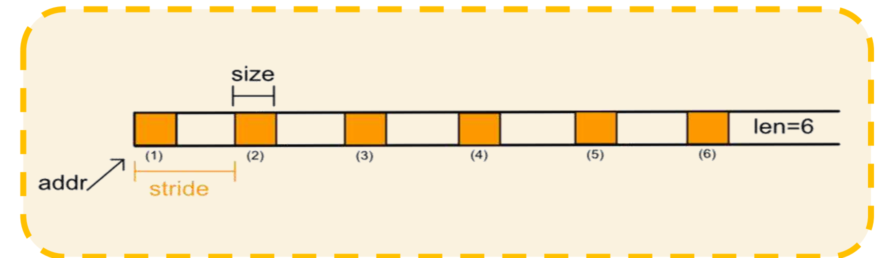


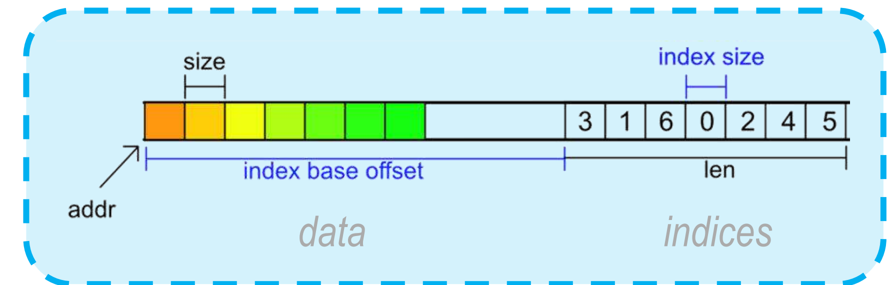


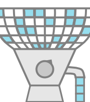
AXI-Pack Protocol

- AXI4 – a commonly used on-chip bus protocol
  - It supports **incremental burst**
  - Does **not support strided and indirect** stream
  - It can be extended by **user signals**
- AXI-PACK user extension
  - **Affine field:** *stride* between elements
  - **Indirect field:** *idx size + idx base offs* (Index address offset related to data address)
  - *indir*: Switch between strided and indirect modes
  - *pack*: Switch between standard AXI4 and AXI-PACK transactions
  - **Light extension**: 30 bits for AW/AR, **~7%** extension



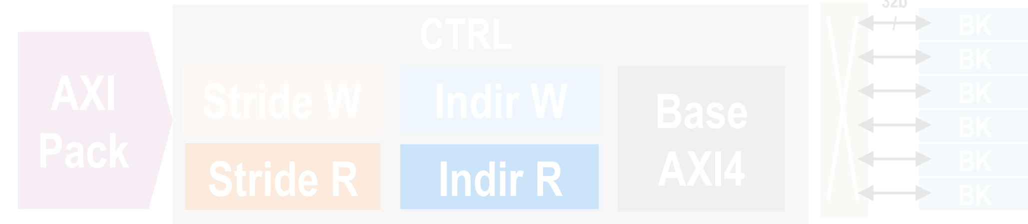


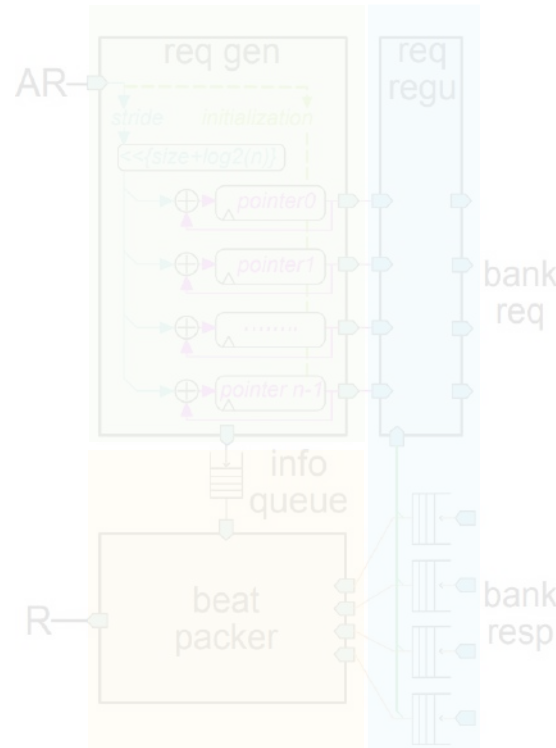


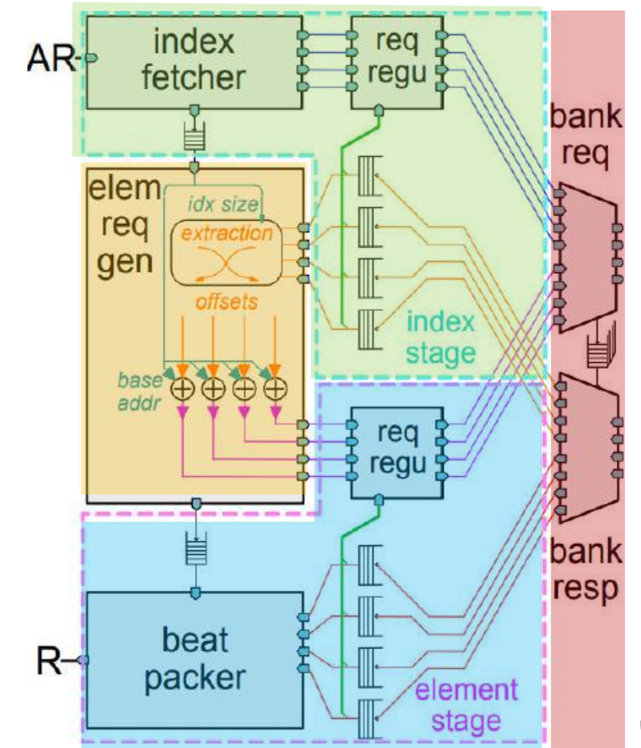


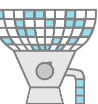
AXI-Pack Banked SRAM Controller

- Controller top-level
  - Strided/indirect read/write convertors
  - Base AXI4 convertor: compatibility
- Strided converter
  - *N* parallel pointers
  - Decoupling queues
  - Pack data on bus
- Indirect converter
  - Index fetching stage
  - Index -> element req
  - Element stage
  - Data flow control



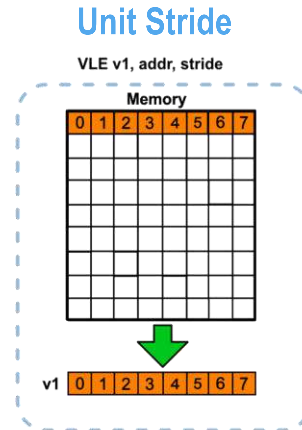


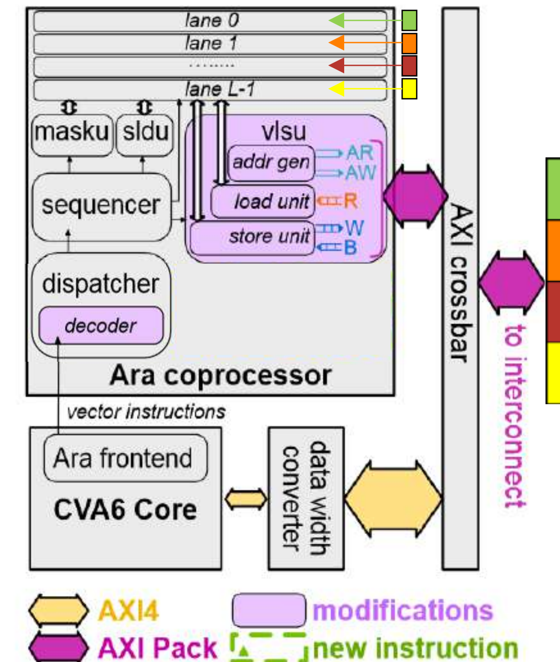


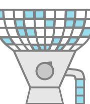


AXI-Pack Extended Vector Processor

- Our open-source RISC-V vector processor
  - *CVA6*: RISC-V core
  - *Ara*: RISC-V vector coprocessor
    - **Chaining** – decouple access-execution
    - **Multi-lane** – parallel processing units
    - Unit stride, **strided, indexed** Vector L/S instr
- Extensions on Ara
  - New Instruction
    - In-Memory indexed vector load/store
  - Extend vector load/store unit
    - Issue AXI-Pack requests
    - Packed data directly forwarded to lanes
    - Very light modification

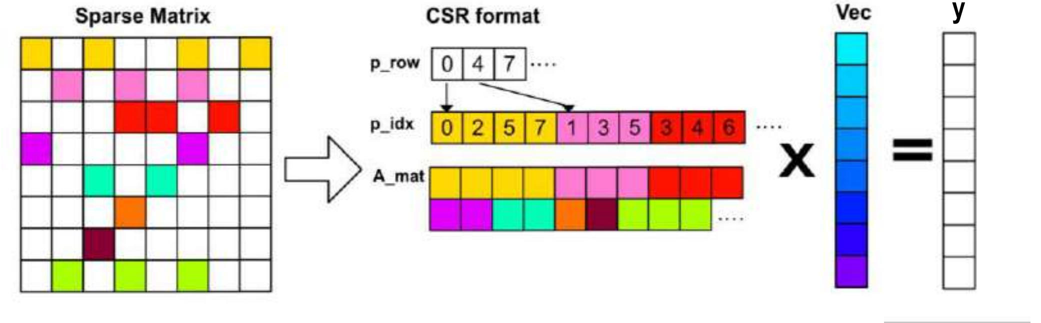






Sparse Matrix-Vector Multiple (SpMV) Example

```
For each row i
    y[i] = y[i] + A_mat[prow_i+0] * Vec[p_idx[prow_i+0]]
    y[i] = y[i] + A_mat[prow_i+1] * Vec[p_idx[prow_i+1]]
    y[i] = y[i] + A_mat[prow_i+2] * Vec[p_idx[prow_i+2]]
    y[i] = y[i] + A_mat[prow_i+3] * Vec[p_idx[prow_i+3]]
    y[i] = y[i] + A_mat[prow_i+4] * Vec[p_idx[prow_i+4]]
    ......
```

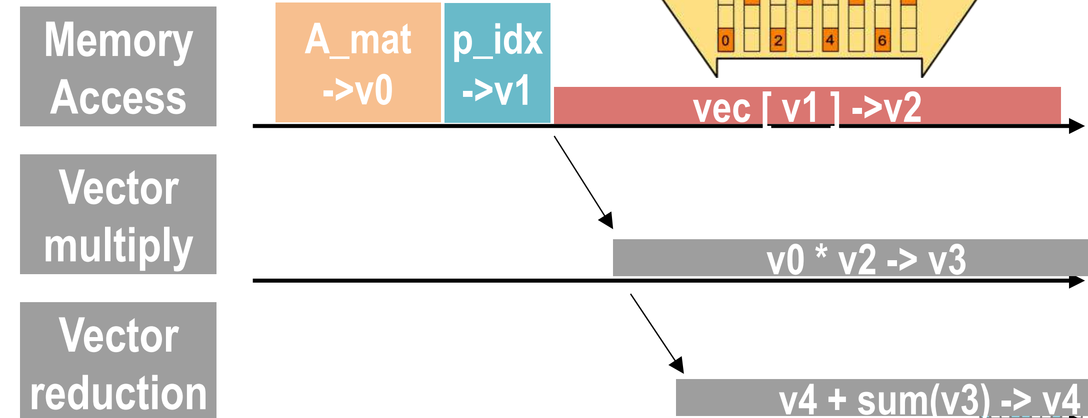


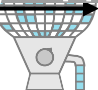
Loop unrolling | Vectorization

- Baseline code

```
For each row i
    vsetvli, p_row_{i+1}-p_row_i, e64 //set vector len, size
    vle64.v v0, (A_mat+p_row_i)   //load nonzeros   ->v0
    vle32.v v1, (p_idx+p_row_i)   //load indices    ->v1
    vloxei32.v v2, (Vec), v1      //indexed load Vec->v2
                                    by indices in v1

    vfmul.vv v3, v0, v2           //        v0 * v2 ->v3
    vfredsum.vs v4, v3, v4        //   v4 + sum(v3) ->v4
```

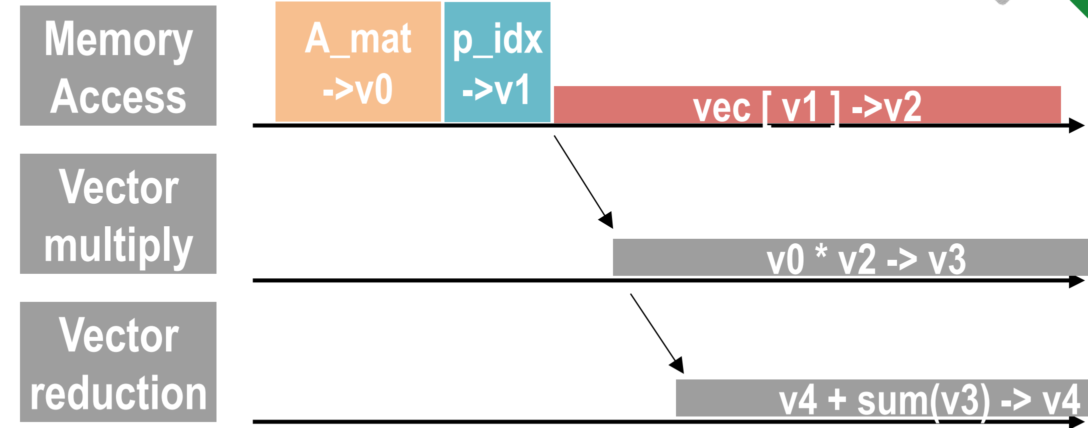




Sparse Matrix-Vector Multiple (SpMV) Example

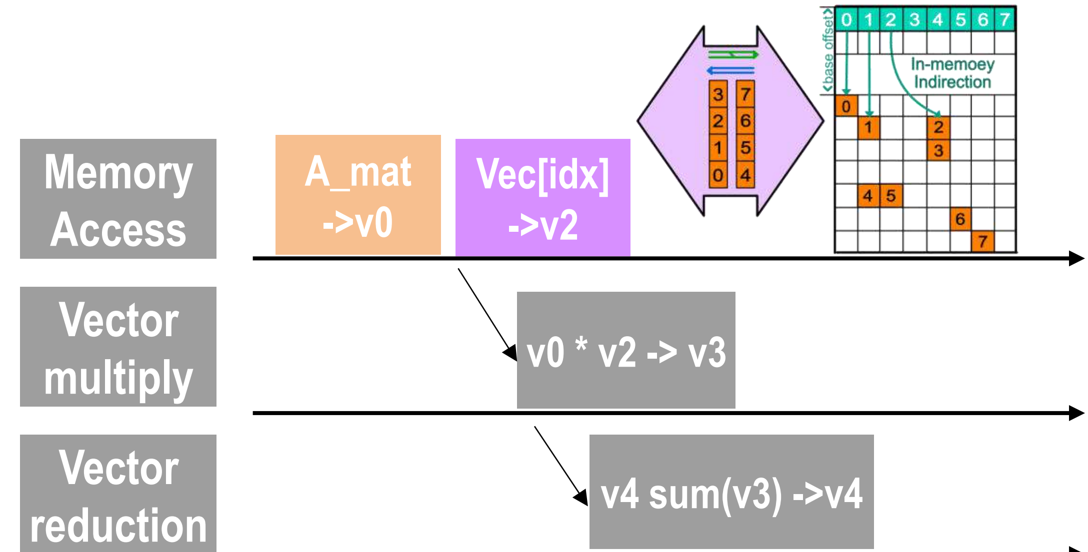
- Baseline code ➡ Memory-bounded

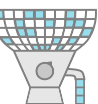
```
For each row i
    vsetvli, p_row_{i+1}-p_row_i, e64 //set vector len, size
    vle64.v v0, (A_mat+p_row_i)       //load nonzeros    ->v0
    vle32.v v1, (p_idx+p_row_i)       //load indices     ->v1
    vloxei32.v v2, (Vec), v1          //indexed load Vec->v2
                                        by indices in v1
    vfmul.vv v3, v0, v2               //         v0 * v2 ->v3
    vfredsum.vs v4, v3, v4            //    v4 + sum(v3) ->v4
```



- AXI-Pack code ➡ Compute-bounded

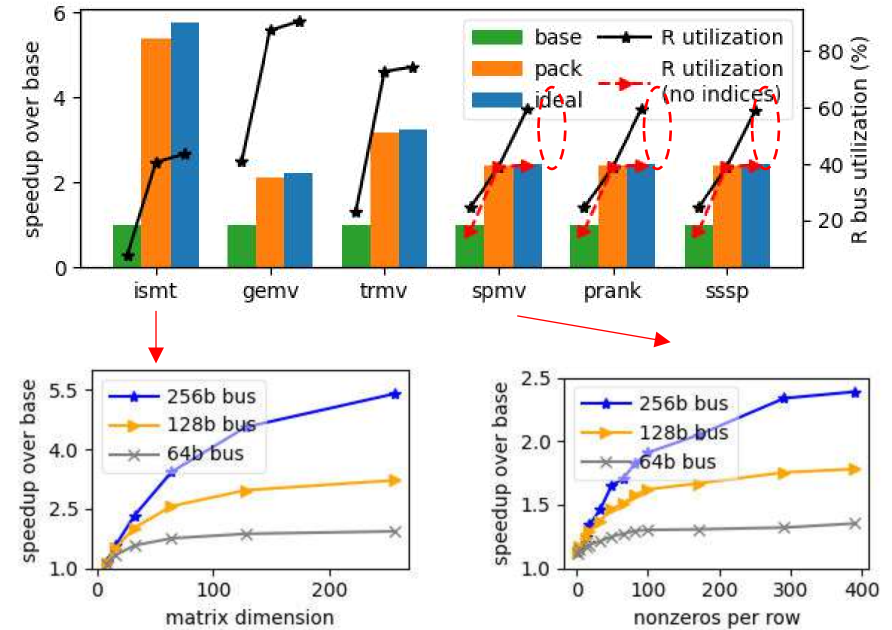
```
vlimxei32.v v2, (Vec), (p_idx+p_row_i)
                                 //in-mem indexed load
                                   given Vec addr
                                         indices addr
                                   VLSU issues AXI-Pack
                                   AXI-Pack stream ->v2
```

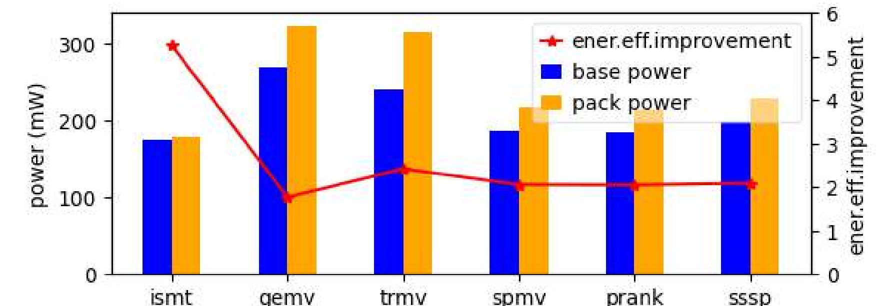


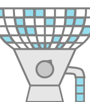


Performance

- Benchmarking irregular workloads
  - AXI-Pack achieves 97% of the IDEAL performance
    - Stride benchmarks(peak 5.4x speedup)
    - Indirect benchmarks (peak 2.4x speedup)
  - Avoid 20 % bus traffic for indices
  - Scales well with benchmark size & bus width
- Area & Power
  - Overall extension: 6.2% area of the VP
  - AXI-PACK increase avg. 5.2% power
  - AXI-PACK saves overall energy
    - Stride workloads: 5.3x (peak)
    - Indirect workloads: 2.1x (peak)

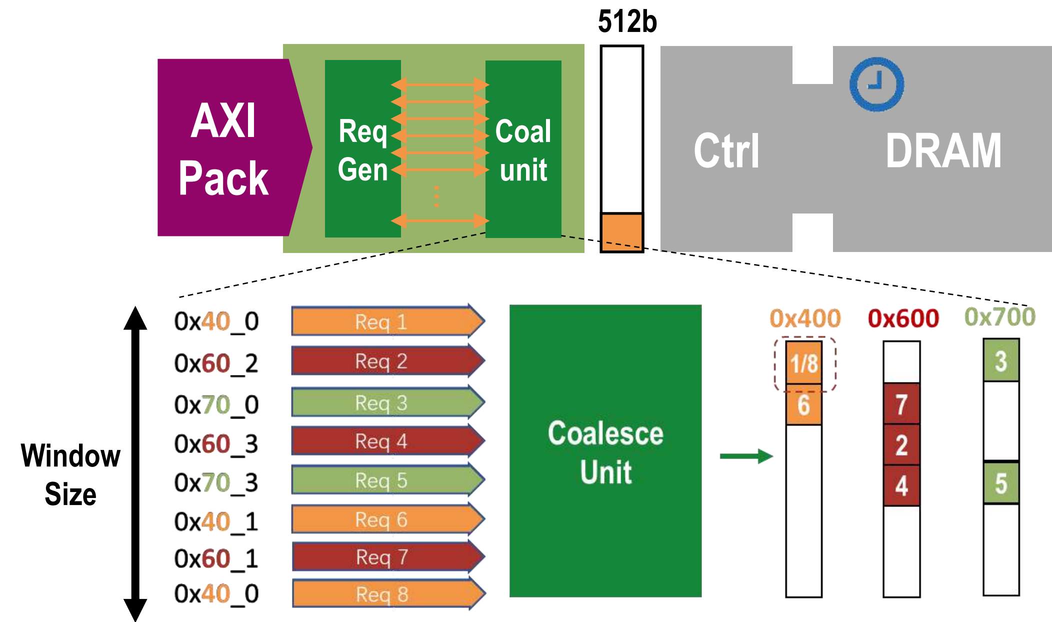




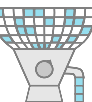


Moving further to DRAM

- Why & Challenges
  - Larger & Sparser datasets
    - > 1GB storage
    - < 0.00001% sparsity
  - AXI-Pack stream on DRAM is not easy
    - *Long latency: 60~200 ns*
    - Larger access granularity: *512b*
- Address coalesce unit
  - Reshape narrow access -> wide access
    - Reduce redundant access
    - Short-term data reuse
    - Non-blocking
    - Configurable window size



Does it really work



Coalesce Unit Saves Your Life

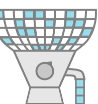
- DRAM indirect stream BW of SPMV on real large sparse matrices *by AXI-Pack*
  - Avg 5.8x(max 17x) speedup to w/o coalesce
  - Comparable BW to Max.DRAM.BW

BW (GB/S)
50
45
40
35
30
25
20
15
10
5
0

Max.DRAM.BW = 32GB/s

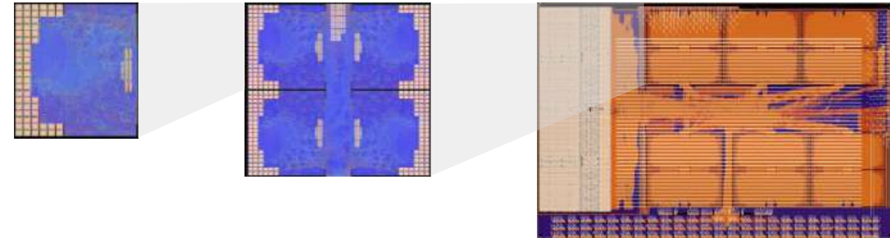
Coalesce Window Size
w/o coal 8 16 32 64 128 256

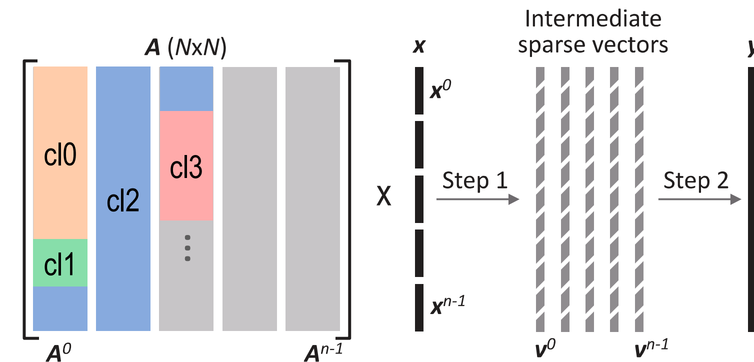
333SP adaptive AS365 as-Skitter atmosmodm belgium_osm Bump_2911 cage15 circuit5M_dc com-Youtube cont11_l CurlCurl_3 CurlCurl_4 delaunay_n21 delaunay_n22 Delor295K Delor64K dgreen eu-2005 Freescale1 Freescale2 FullChip G3_circuit Hamrle3 hugebubbles in-2004 kkt_power LargeRegFile M6 memchip netherlands_osm nlpkkt120 pwtk HPCG af_shell10 BenElechi1 bone010



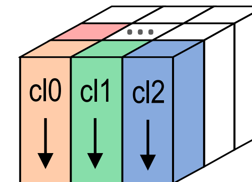
Next Steps: SSSR Multi-Cluster Workloads

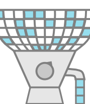
- How do we scale to many clusters?
- *Large-scale sparse LA*: **matrix slicing**
  - Store large matrices as *fixed-width strips*
  - Reduce partial results as sparse vectors
  - Chunk length chosen to balance load
  - Keep reusable chunks in TCDM (e.g. HPCG)
- *Large stencil grids*: **grid tiling**
  - Decompose stencils to *single-output steps* → no need for inter-cluster halo sharing
  - Tile grid and *stream* along Z direction





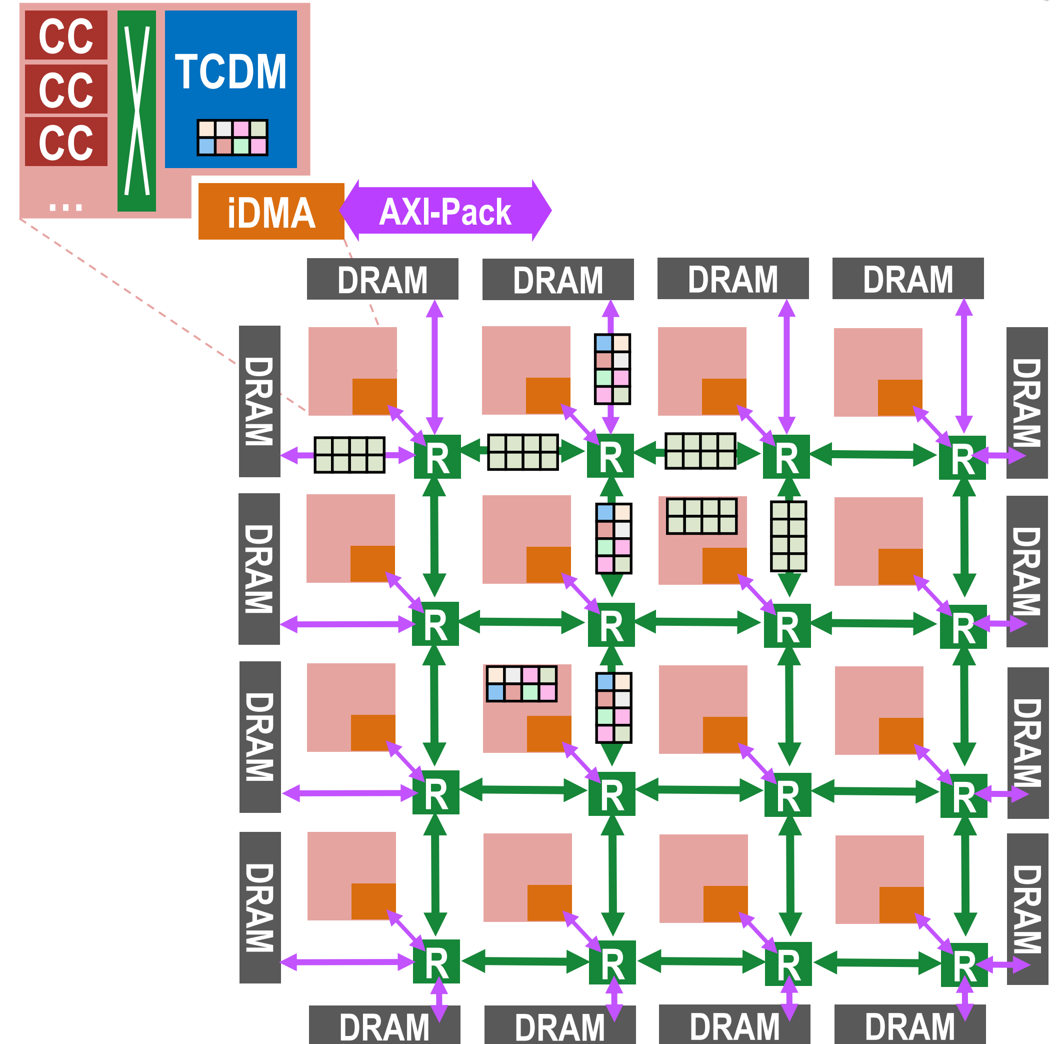
F. Sadi et al., *"Efficient SpMV Operation for Large and Highly Sparse Matrices using Scalable Multi-way Merge Parallelization"*, MICRO 52

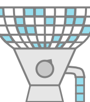




Next Steps: AXI-Pack Extensions and NoCs

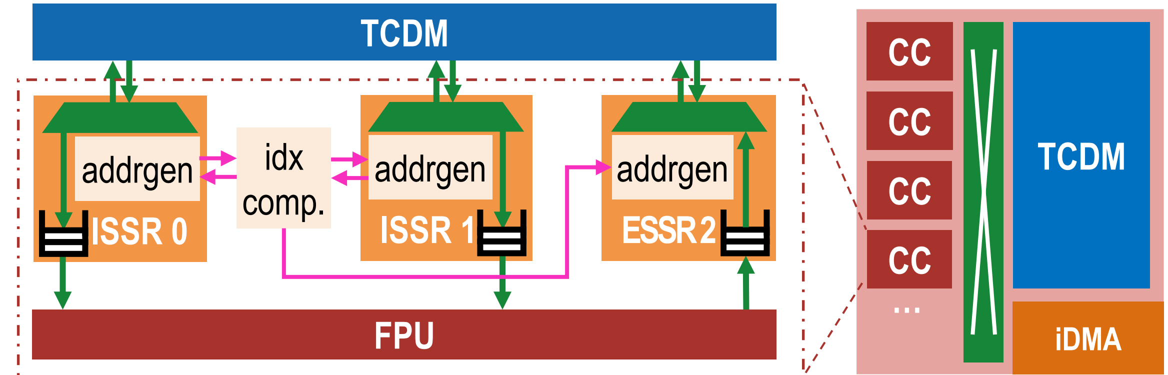
- Extend **AXI-Pack capabilities**
  - More stream types: nD affine streams, ranged indirection, master-side indices, …
  - Caching and prefetching support
- Deploy AXI-Pack in **NoC-based manycores**
  - **Snitch Cluster** *(Paul):* extend TCDM and internals
  - **iDMA** *(Thomas):* extend engine, instruction frontend
  - **FlooNoC** *(Tim):* extend links, routers
  - **DRAM** *(Chi):* performant AXI-Pack endpoints

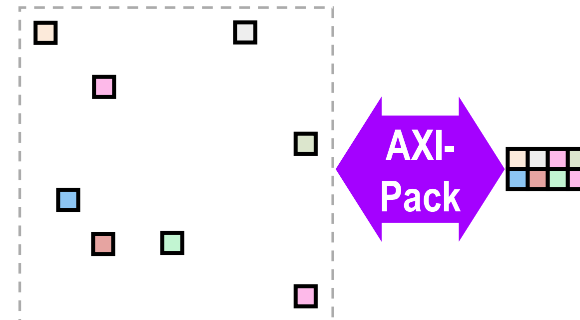


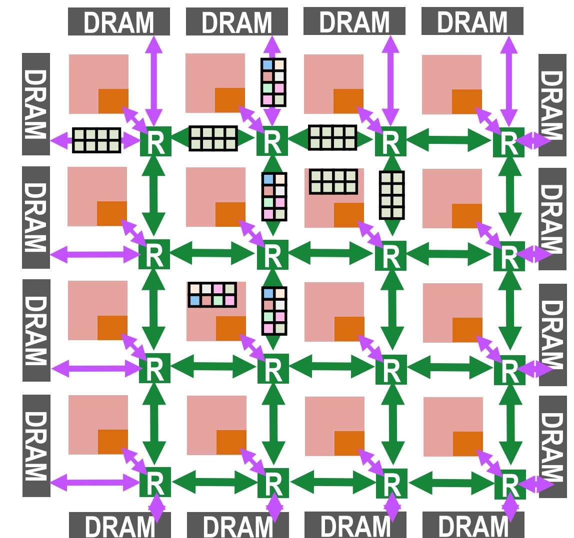


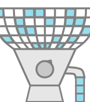
Stream Extensions: Turbocharging Irregular Workloads

- Irregular workloads are **challenging**
  - Complex memory access patterns & control
  - Inefficiently handled by SoA hardware
- Stream extensions **accelerate them**
  - Clusters: **Sparse Stream Semantic Registers**
    - Sparse LA: up to **5.9×** faster, **3.0×** less energy
    - Stencils: up to **3.7×** faster and **93%** FP util.
  - SoC level: **AXI-Pack**
    - Avg. **>60%** bandwidth utilization
    - Avg. **5.8×** DRAM coalescer speedup
  - Next steps: **scaling out**
    - SSSR multi-cluster workloads
    - AXI-Pack extensions and NoC transport









Thank you!

Paul Scheffler paulsc@iis.ee.ethz.ch
Chi Zhang chizhang@iis.ee.ethz.ch