SHAPING THE NEXT GENERATION OF ELECTRONICS

JUNE 23-27, 2024
MOSCONE WEST CENTER
SAN FRANCISCO, CA, USA

PULP
Parallel Ultra Low Power

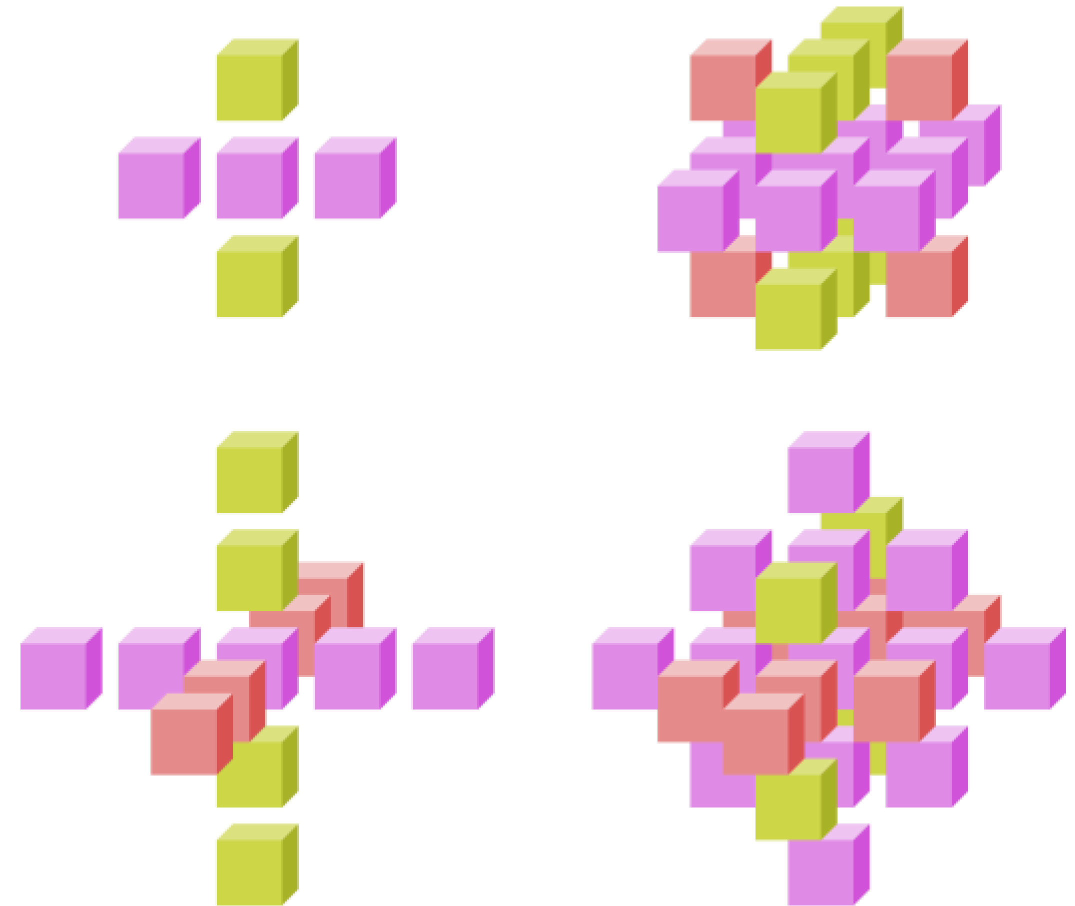
ETH zürich

SARIS: Accelerating Stencil Computations on Energy-Efficient RISC-V Compute Clusters with Indirect Stream Registers

Paul Scheffler *paulsc@iis.ee.ethz.ch*
Luca Colagrande *colluca@iis.ee.ethz.ch*
Luca Benini *lbenini@iis.ee.ethz.ch*

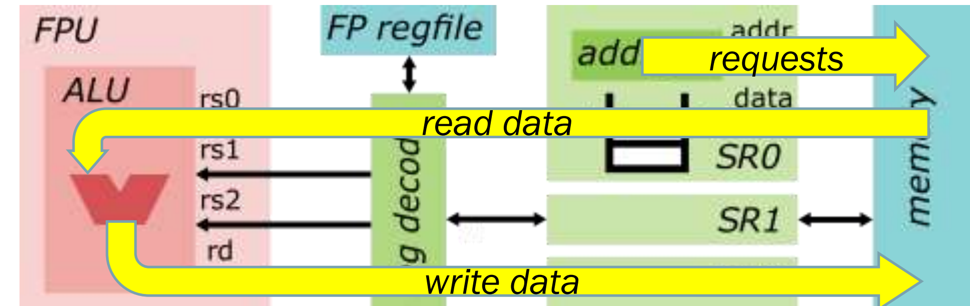
Stencil Codes and Their Performance

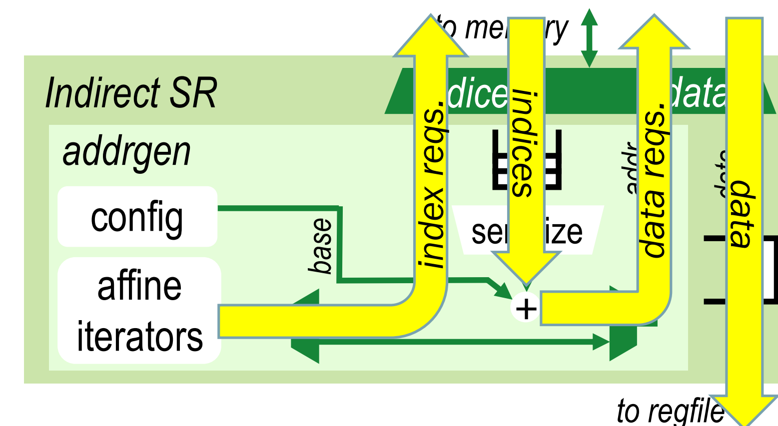
- Update points in *nD* grid using neighbors in fixed pattern (*stencil)*
  - Physics, signal processing, ML ...
- Vary in *size*, *shape*, *arithmetic intensity*
  - Performance through *code generators*
- Main bottleneck: *memory accesses*
  - Small stencils: *memory bound*
  - Complex shapes: *irregular accesses*
  - On energy-efficient in-order cores: *address calculation*, *access overheads*



Accelerating Stencils with Stream Registers

- *Stream Registers* (SRs): map *streams* to *register accesses*
  - Addresses *hardware-generated*
  - ➢ Near-ideal FPU utilization on data-driven workloads even on single-issue InO cores
- Indirect SRs: use *base address* and *index array* for *scatter-gather*
  - Accelerate *arbitrarily irregular* accesses
- **How can we use indirect SRs to accelerate *generic stencil codes?***





THE CHIPS TO SYSTEMS CONFERENCE
SHAPING THE NEXT GENERATION OF ELECTRONICS

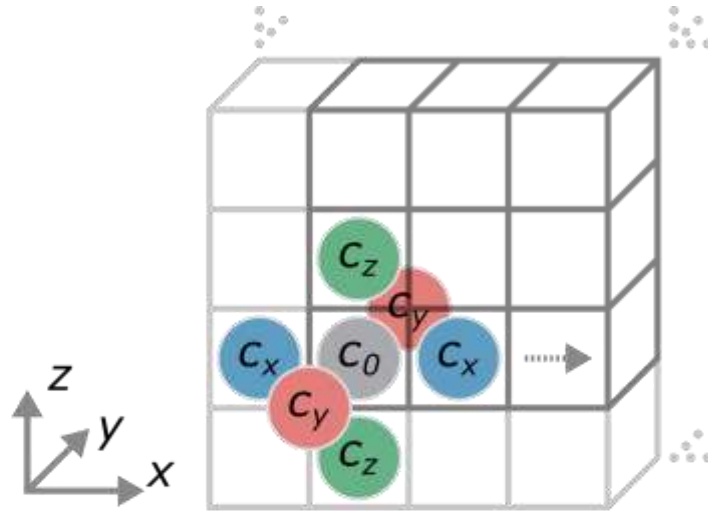
Our Contributions

- ***SARIS***: a generic method for *Stencil Acceleration using Register-mapped Indirect Streams*
- ***SW implementation***: on RISC-V compute cluster with indirect SRs
  - Baseline (RV32G) stencil codes
  - SARIS-accelerated stencil codes

open source

- ***Evaluation***: on cluster (vs. RV32G baseline):
  - 2.72× speedups, 81% FPU utilization *(geomean)*
  - 1.58× higher energy efficiency *(geomean)*
- ***Scaleout & SoA Comparison:*** on 256-core system with HBM2E:
  - 2.14× speedups, 64% FPU utilization *(geomean)*
  - 15% higher fraction of peak compute than leading GPU code generator

Example: 7-Point Star (7PS) Stencil



Assume:
- N^3 grid
- FP64 data
- double-buffered (*inp* → *out*)

```
for z in 1 to N-1:
  for y in 1 to N-1:
    for x in 1 to N-1:
      out[z][y][x] = c0 * inp[z][y][x]
        + cx * (inp[z][y][x-1] + inp[z][y][x+1])
        + cy * (inp[z][y-1][x] + inp[z][y+1][x])
        + cz * (inp[z-1][y][x] + inp[z+1][y][x]);
```

Compile ***point loop*** *for RV32G*

```
x: fld      ft0, 0(t0)        # inp[z][y][x]
   fmul.d   ft0, %[c0], ft0
   fld      ft1, -8(t0)       # inp[z][y][x-1]
   fld      ft2,  8(t0)       # inp[z][y][x+1]
   fadd.d   ft1, ft1, ft2
   fmadd.d  ft0, %[cx], ft1, ft0
   fld      ft1, -YOFFS(t0) # inp[z][y-1][x]
   fld      ft2,  YOFFS(t0) # inp[z][y+1][x]
   fadd.d   ft1, ft1, ft2
   fmadd.d  ft0, %[cy], ft1, ft0
   fld      ft1, 0(t1)        # inp[z-1][y][x]
   fld      ft2, 0(t2)        # inp[z+1][y][x]
   fadd.d   ft1, ft1, ft2
   fmadd.d  ft0, %[cz], ft1, ft0
   fsd      ft0, 0(t3)        # out[z][y][x]
   addi     t0, 8
   addi     t1, 8
   addi     t2, 8
   addi     t3, 8
   bne      t0, a0, x
```

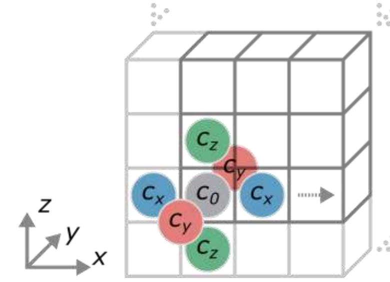
7PS on RISC-V (RV32G) Baseline

- Only 7/20 issues are useful compute
  - Address compute, load-stores *dominate*
- Single-issue InO cores: **≤39% FPU util.**
  - *x* unroll benefits limited (up from 35%)
  - Ignores memory, dependency stalls
  - Inefficiencies *generalize* across stencils
- Can we improve utilization with SRs?
  - Generality: how to handle multiple data arrays, irregular stencil shapes?

```
x: fld      ft0, 0(t0)        # inp[z][y][x]
   fmul.d   ft0, %[c0], ft0
   fld      ft1, -8(t0)       # inp[z][y][x-1]
   fld      ft2,  8(t0)       # inp[z][y][x+1]
   fadd.d   ft1, ft1, ft2
   fmadd.d  ft0, %[cx], ft1, ft0
   fld      ft1, -YOFFS(t0) # inp[z][y-1][x]
   fld      ft2,  YOFFS(t0) # inp[z][y+1][x]
   fadd.d   ft1, ft1, ft2
   fmadd.d  ft0, %[cy], ft1, ft0
   fld      ft1, 0(t1)        # inp[z-1][y][x]
   fld      ft2, 0(t2)        # inp[z+1][y][x]
   fadd.d   ft1, ft1, ft2
   fmadd.d  ft0, %[cz], ft1, ft0
   fsd      ft0, 0(t3)        # out[z][y][x]
   addi     t0, 8
   addi     t1, 8
   addi     t2, 8
   addi     t3, 8
   bne      t0, a0, x
```

The SARIS Method

1. Map *grid data loads* to indirect SRs
   - Point is base, load offsets are indices
   - ➢ Indices *constant* across point iterations
2. *Partition* loads among indirect SRs
   - Concurrent streaming of co-operands
3. Map *coefficient loads* or *grid stores* to *remaining SRs*
   - Do coefficients fit in register file?
4. Determine *point loop schedule*
   - *Provides index sequence for SRs*



- *7* grid loads
- 1 grid store
- 4 coefficients

*For **2 indirect*** (SR0-1) + ***1 strided SR*** (SR2):

streams	***compute***
SR0 ← inp[z][y][x]	t0 = c0 × **SR0**
SR0 ← inp[z][y][x-1] **SR1** ← inp[z][y][x+1]	t1 = **SR0** + **SR1**
	t0 += cx × t1
SR0 ← inp[z][y-1][x] **SR1** ← inp[z][y+1][x]	t1 = **SR0** + **SR1**
	t0 += cy × t1
SR0 ← inp[z-1][y][x] **SR1** ← inp[z+1][y][x]	t1 = **SR0** + **SR1**
SR2 → out[z][y][x]	**SR2** += cz × t1 + t0

7PS Acceleration Using SARIS

- Rewrite point loop to use SRs
  - Read, write SRs as in schedule
  - Indices for SRs configured *once*, base set on every point iteration
- Almost *all* issues now useful compute
  - All load-stores done by SRs
  - Remaining overhead is *constant:* does not increase with #grid points
- Orthogonal to existing optimizations
  - Unrolling, reordering, HW loops enable **near-ideal (~81%) FPU utilizations**

```
sr_set_idcs(...);       # Constant across points
for (x,y,z) in 0 to N-2:
  # Start indirect streams with base (x,y,z)
  sr_indir_stream(OFFS(x,y,z), ...);
  acc =  c0 * sr_read(SR0);
  acc += cx * (sr_read(SR0) + sr_read(SR1));
  acc += cy * (sr_read(SR0) + sr_read(SR1));
  acc += cz * (sr_read(SR0) + sr_read(SR1));
  sr_write(SR2, acc);
```

```
x: SRIR      t0, ...               # SSSRs: 3 insts
   fmul.d    ft0, %[c0], SR0
   fadd.d    ft1, SR0, SR1
   fmadd.d   ft0, %[cx], ft1, ft0
   fadd.d    ft1, SR0, SR1
   fmadd.d   ft0, %[cy], ft1, ft0
   fadd.d    ft1, SR0, SR1
   fmadd.d   SR2, %[cz], ft1, ft0
   addi      t0, 8
   bne       t0, a0, x
```

THE CHIPS TO SYSTEMS CONFERENCE

SHAPING THE NEXT GENERATION OF ELECTRONICS

Evaluation on RISC-V Compute Cluster

open source

- Target energy-efficient *Snitch Cluster*[1]
  - 8 single-issue InO RV32 cores w. HW loop
  - 2 indirect + 1 strided SRs per core[2]
  - Shared 128 KiB SPM + DMA engine
- Implemented 10 stencil codes
  - Opt. baseline (RV32G) & SARIS variants
  - 1 iteration on 64^2 (2D) or 16^3 (3D) grid tile
- Tiles double-buffered in SPM with DMA
  - Base for manycore scaleout study

Stencil Code	Dim.	Rad.	#Loads	#Coeffs	#FLOPs
`jacobi_2d`[3]	2D	1	5	1	5
`j2d5pt`[4]	2D	1	5	6	10
`box2d1r`[4]	2D	1	9	9	17
`j2d9pt`[4]	2D	2	9	10	18
`j2d9pt_gol`[4]	2D	1	9	10	18
`star2d3r`[4]	2D	3	13	13	25
`star3d2r`[4]	3D	2	13	13	25
`ac_iso_cd`[5]	3D	4	26	13	38
`box3d1r`[4]	3D	1	27	27	53
`j3d27pt`[4]	3D	1	27	28	54

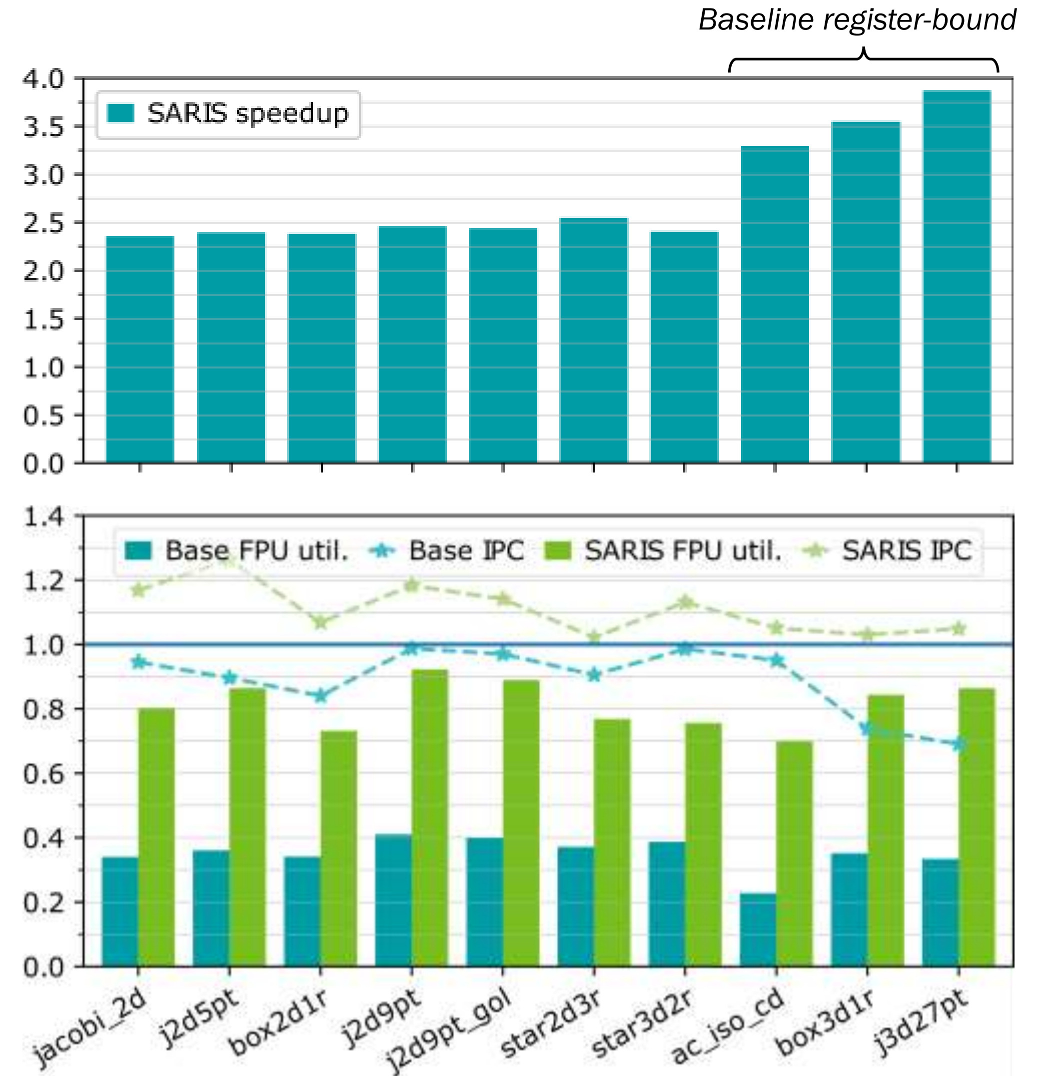
Stencil Code Implementation on Snitch Cluster

- Both variants use Snitch-optimized LLVM
  - Dedicated in-order scheduling model
  - Custom reassociation pass for max FPU util.
  - C++ templates compiled using `–Ofast`
- Parallelized through 4×x, 2×y interleave
  - Additional ≤4× point loop unroll *only if* beneficial to performance
- SARIS variants leverage SRs & HW loop
  - Snitch LLVM: intrinsics support for SRs
  - Point loop scheduled as dictated by SARIS

```
// Configure constant indices for indirect SRs
sr_set_idcs(SR0, {...});
sr_set_idcs(SR1, {...});
// Configure affine SR (does grid writes here)
sr_affine_write_3d(SR2, &out[1][1][1], ...);

for (z = H + zoffs, z < N-H, z += ZI) {
  for (y = H + yoffs, y < N-H, y += YI) {
    for (x = H + xoffs, x < N-H, x += XI) {
      // Launch indirect grid reads with
      // point offset as array base (SR 0,1)
      sr_indir_read(SR0|SR1, OFFS(x,y,z), ...);

      <point loop body using SRs & HW loop>

}}}
```

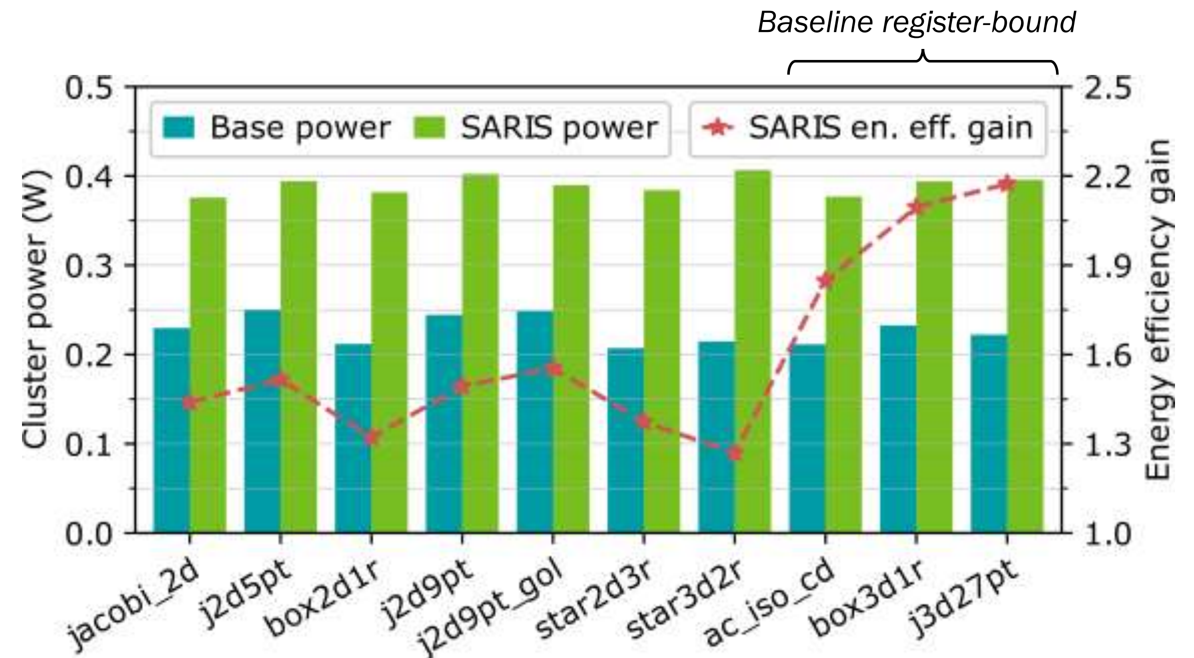
Cluster Performance Benefits

- Codes run in cluster RTL simulation
- 2.36–3.87× (gm. 2.72×) speedups
  - As FLOPs increase, baseline unroll becomes *register-bound* → larger SARIS speedups
- FPU util. improves gm. 35% → 81%
  - SRs unlock Snitch's *pseudo-dual-issue* feature, improving IPC gm. 0.89 → 1.11
  - FP util. / IPC *never* below *70% / 1.0*
- Variations due to SPM contention, halo size, index setup overheads



Cluster Energy Efficiency Benefits

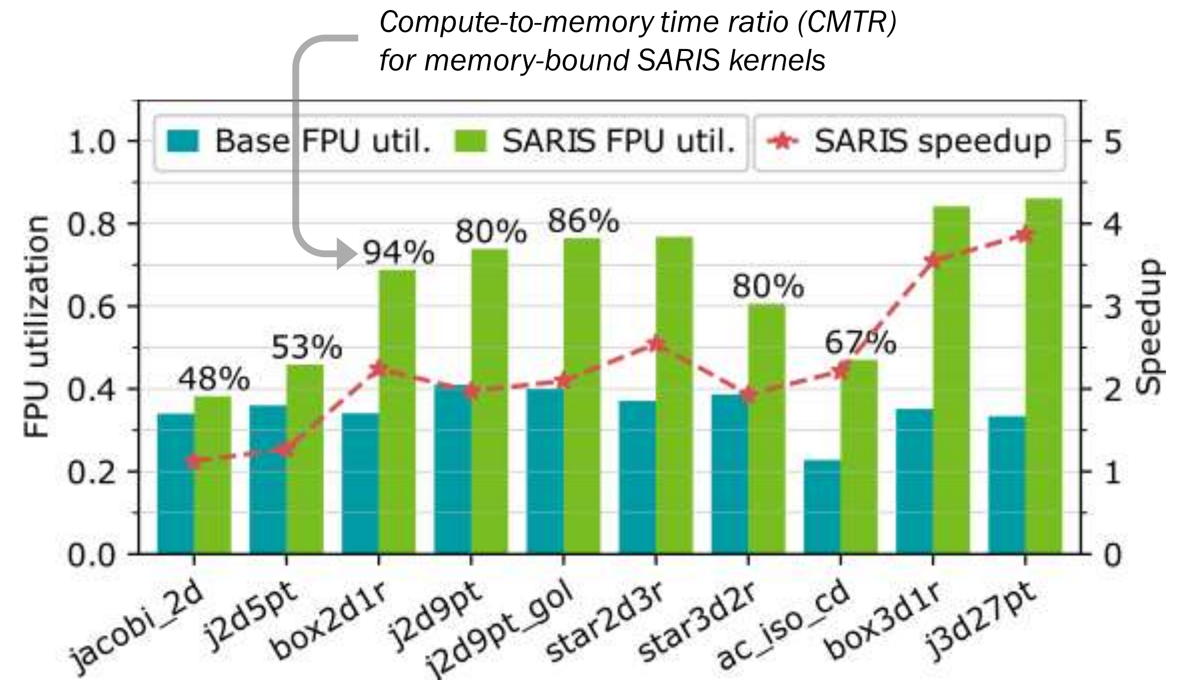
- Estimate cluster power in GF 12LP+
  - Use *PrimeTime,* P&R netlist + PLS activity
  - 1 GHz clock, typical corner (25 °C, 0.8V)
- 1.27–2.17× (gm. 1.58×) less energy
  - Power gm. 227 mW → 390 mW (1.72×) due to higher FPU, SPM util.
  - Only slight power variations, resemble those in FPU utilization
- Register-bound gains again higher



THE CHIPS TO SYSTEMS CONFERENCE
SHAPING THE NEXT GENERATION OF ELECTRONICS

Manycore Scaleout Study

- 256-core system based on *Manticore*[6]
  - 8 groups of 4 clusters, each sharing bandwidth of one HBM2E device in stack
  - $16384^2/512^3$ grids, imbalance considered
- Gm. 2.14× faster, 64% FPU utilization
  - Up to 406 GFLOP/s (79% of peak)
  - *Despite* 7/10 memory-bound codes
- Arith. intensity rises with FLOPs/point
  - 3D stencils regress to memory-bound due to halo, extra I/O arrays (`ac_iso_cd`)



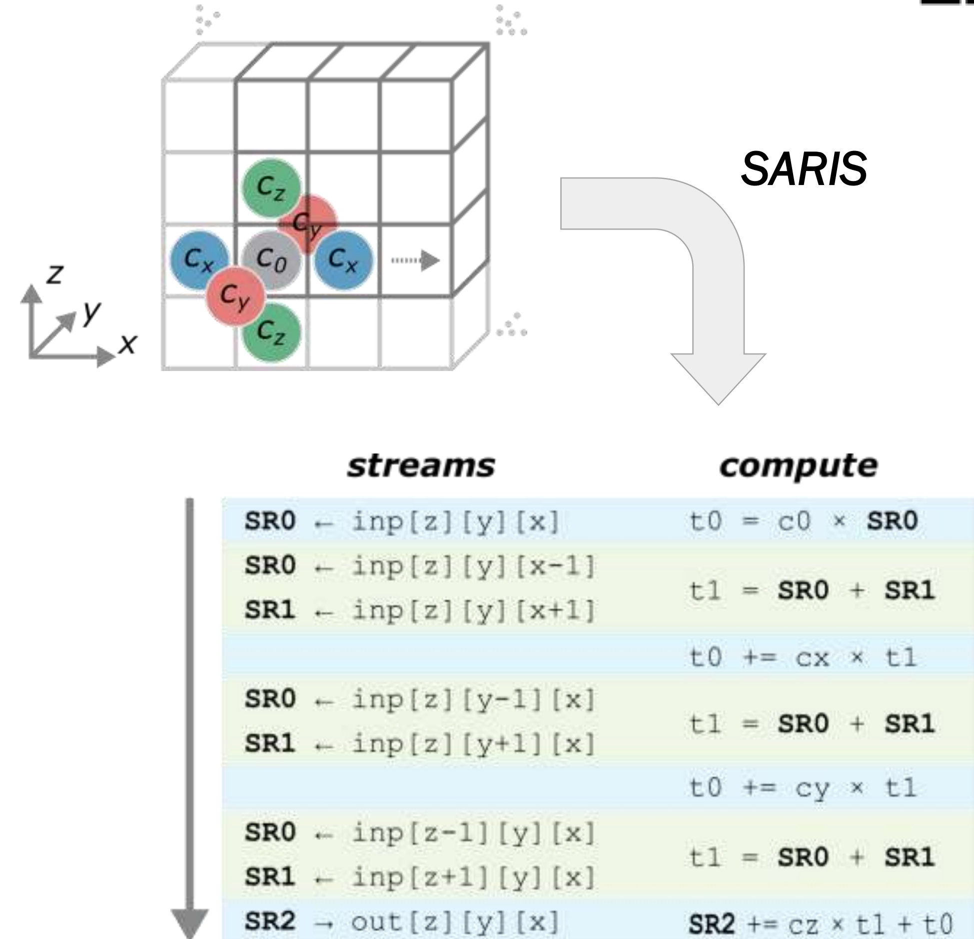
Related Work and Comparison to SoA

- Some affine SRs evaluated on *selected, affine-streamable* stencils
  - SARIS is the first *generic* approach to using indirect SRs for near-ideal FPU util.
- Numerous generic *software* solutions
  - *Demonstrator codes* or *generators* highly optimized for target architecture
- 15% higher fraction of peak compute than leading GPU generator AN5D
  - Many SoA SW methods *orthogonal* to SARIS

	Work	Platform	Prec.	% Pk.
CPU	Zhang et al.[7]	FT-2000+ (1 core)	FP64	29%
	Yount[8]	Xeon Phi 7120A	FP32	30%
	Bricks[9]	Xeon Gold 6130	FP32	45%
GPU	ARTEMIS[10]	Tesla P100	FP64	36%
	DRStencil[11]	Tesla P100	FP64	48%
	EBISU[12]	A100	FP64	49%
	AN5D[4]	Tesla V100 SXM2	FP32	**69%**
WSE	Rocki et al.[13]	Cerebras WSE-1	FP16-32	28%
	Jaquelin et al.[5]	Cerebras WSE-2	FP32	28%
	SARIS (Ours)	*Manticore-256s*	*FP64*	*79%*

Conclusion and Future Work

- We present the ***SARIS*** method
  - **First *generic* approach** to stencil code acceleration using indirect SRs
- Evaluated SARIS on Snitch cluster
  - Gm. **2.72×** faster, **81%** FPU util.
  - Gm. **1.58×** more energy-efficient
- Scaled SARIS to 256 cores + HBM2E
  - Gm. **2.14×** faster, **64%** FPU util.
  - **≤79%** of peak compute, **+15%** over SoA
- Future work: more stencils & targets
  - Full code generation from a stencil DSL
  - Evaluate SARIS on more platforms



References

[1] Florian Zaruba, Fabian Schuiki, Torsten Hoefler, and Luca Benini. 2020. Snitch: A Tiny Pseudo Dual-Issue Processor for Area and Energy Efficient Execution of Floating-Point Intensive Workloads. *IEEE Trans. Comput.* 70 (2020), 1845–1860.

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

[3] Louis-Noël Pouchet. 2015. Polybench/C: The polyhedral benchmark suite.

[4] Kazuaki Matsumura, Hamid Reza Zohouri, Mohamed Wahib, Toshio Endo, and Satoshi Matsuoka. 2020. AN5D: Automated Stencil Framework for High-Degree Temporal Blocking on GPUs. In *Proc. 18th ACM/IEEE Int. Symp. Code Gener. Optim*, 199–211.

[5] Mathias Jacquelin, Mauricio Araya-Polo, and Jie Meng. 2022. Scalable Distributed High-Order Stencil Computations. In *SC '22: Proc. Int. Conf. High Perform. Comput., Netw., Storage Analysis*. Article 30, 13 pages.

[6] Florian Zaruba, Fabian Schuiki, and Luca Benini. 2021. Manticore: A 4096-Core RISC-V Chiplet Architecture for Ultraefficient Floating-Point Computing. *IEEE Micro* 41, 2 (2021), 36–42.

[7] Kaifang Zhang, Huayou Su, Peng Zhang, and Yong Dou. 2020. Data Layout Transformation for Stencil Computations Using ARM NEON Extension. In *2020 IEEE 22nd Int. Conf. High Perform. Comput. and Commun.; IEEE 18th Int. Conf. Smart City; IEEE 6th Int. Conf. Data Science Syst. (HPCC/SmartCity/DSS)*, 180–188.

[8] Charles R. Yount. 2015. Vector Folding: Improving Stencil Performance via Multi-dimensional SIMD-vector Representation. *In 2015 IEEE 17th Int. Conf. High Perform. Comput. Commun., 2015 IEEE 7th Int. Symp. Cyberspace Saf. Secur., 2015 IEEE 12th Int. Conf. Embedded Softw. Syst.,* 865–870.

[9] Tuowen Zhao, Samuel Williams, Mary W. Hall, and Hans Johansen. 2018. Delivering Performance-Portable Stencil Computations on CPUs and GPUs Using Bricks. In *2018 IEEE/ACM Int. Workshop on Perform., Portability Productivity HPC (P3HPC),* 59–70.

[10] Prashant Singh Rawat, Miheer Vaidya, Aravind Sukumaran-Rajam, Atanas Rountev, Louis-Noël Pouchet, and P. Sadayappan. 2019. On Optimizing Complex Stencils on GPUs. In *2019 IEEE Int. Parallel Distrib. Process. Symp. (IPDPS),* 641–652.

[11] Xin You, Hailong Yang, Zhonghui Jiang, Zhongzhi Luan, and Depei Qian. 2021. DRStencil: Exploiting Data Reuse within Low-order Stencil on GPU. In *2021 IEEE 23rd Int. Conf. High Perform. Comput. Commun.; 7th Int. Conf. Data Science Syst.; 19th Int. Conf. Smart City; 7th Int. Conf. Dependability in Sensor, Cloud Big Data Syst. Appl. (HPCC/DSS/SmartCity/DependSys),* 63–70.

[12] Lingqi Zhang, Mohamed Wahib, Peng Chen, Jintao Meng, Xiao Wang, Toshio Endo, and Satoshi Matsuoka. 2023. Revisiting Temporal Blocking Stencil Optimizations. In Proc. 37th Int. Conf. Supercomputing. Association for Computing Machinery, New York, NY, USA, 251–263.

[13] Kamil Rocki, Dirk T. Van Essendelft, Ilya Sharapov, Robert S. Schreiber, Michael Morrison, Vladimir Kibardin, Andrey Portnoy, Jean François Dietiker, Madhava Syamlal, and Michael James. 2020. Fast Stencil-Code Computation on a Wafer-Scale Processor. In *SC20: Int. Conf. High Perf. Comput., Netw., Storage Analysis.* Article 58, 14 pages.