Modeling and tuning of SpMV and a lattice QCD kernel on the A64FX

Christie Alappat, Jan Laukemann, Thomas Gruber, Georg Hager, Gerhard Wellein, Nils Meyer, Tilo Wettig

Friedrich-Alexander-Universität Erlangen-Nürnberg, Germany
Erlangen National High Performance Computing Center (NHR@FAU)
University of Regensburg, Germany

Paper

C. Alappat, N. Meyer, J. Laukemann, T. Gruber, G. Hager, G. Wellein, and T. Wettig:
ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX.
Concurrency and Computation: Practice and Experience, e6512 (2021).
Available with Open Access.

DOI: 10.1002/cpe.6512

SPECIAL ISSUE PAPER

ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX

Christie Alappat*[1] | Nils Meyer[2] | Jan Laukemann[1] | Thomas Gruber[1] | Georg Hager[1] | Gerhard Wellein[1] | Tilo Wettig[2]

[1]Erlangen National High Performance Computing Center, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany
[2]Department of Physics, University of Regensburg, Regensburg, Germany

Correspondence
*Christie Alappat, Erlangen National High Performance Computing Center, Friedrich-Alexander-Universität Erlangen-Nürnberg, Erlangen, Germany.
Email: christie.alappat@fau.de

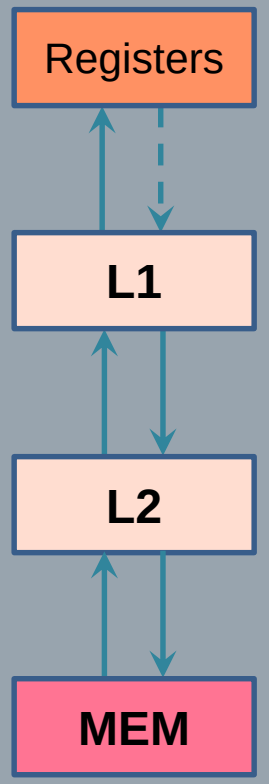
Summary

The A64FX CPU is arguably the most powerful Arm-based processor design to date. Although it is a traditional cache-based multicore processor, its peak performance and memory bandwidth rival accelerator devices. A good understanding of its performance features is of paramount importance for developers who wish to leverage its full potential. We present an architectural analysis of the A64FX used in the Fujitsu FX1000 supercomputer at a level of detail that allows for the construction of Execution-Cache-Memory (ECM) performance models for steady-state loops. In the process we identify architectural peculiarities that point to viable generic optimization strategies. After validating the model using simple streaming loops we apply the insight gained to sparse matrix-vector multiplication (SpMV) and the domain wall (DW) kernel from quantum chromodynamics (QCD). For SpMV we show why the CRS matrix storage format is not a good practical choice on this architecture and how the SELL-C-σ format can achieve bandwidth saturation. For the DW kernel we provide a cache-reuse analysis and show how an appropriate choice of data layout for complex arrays can realize memory-bandwidth saturation in this case as well. A comparison with state-of-the-art high-end Intel Cascade Lake AP and Nvidia V100 systems puts the capabilities of the A64FX into perspective. We also explore the potential for power optimizations using the tuning knobs provided by the Fugaku system, achieving energy savings of about 31% for SpMV and 18% for DW.

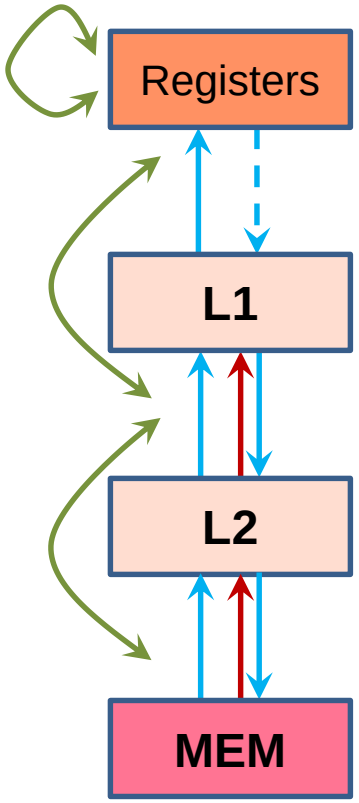
KEYWORDS:
ECM model, A64FX, sparse matrix-vector multiplication, lattice quantum chromodynamics

Single-core analysis

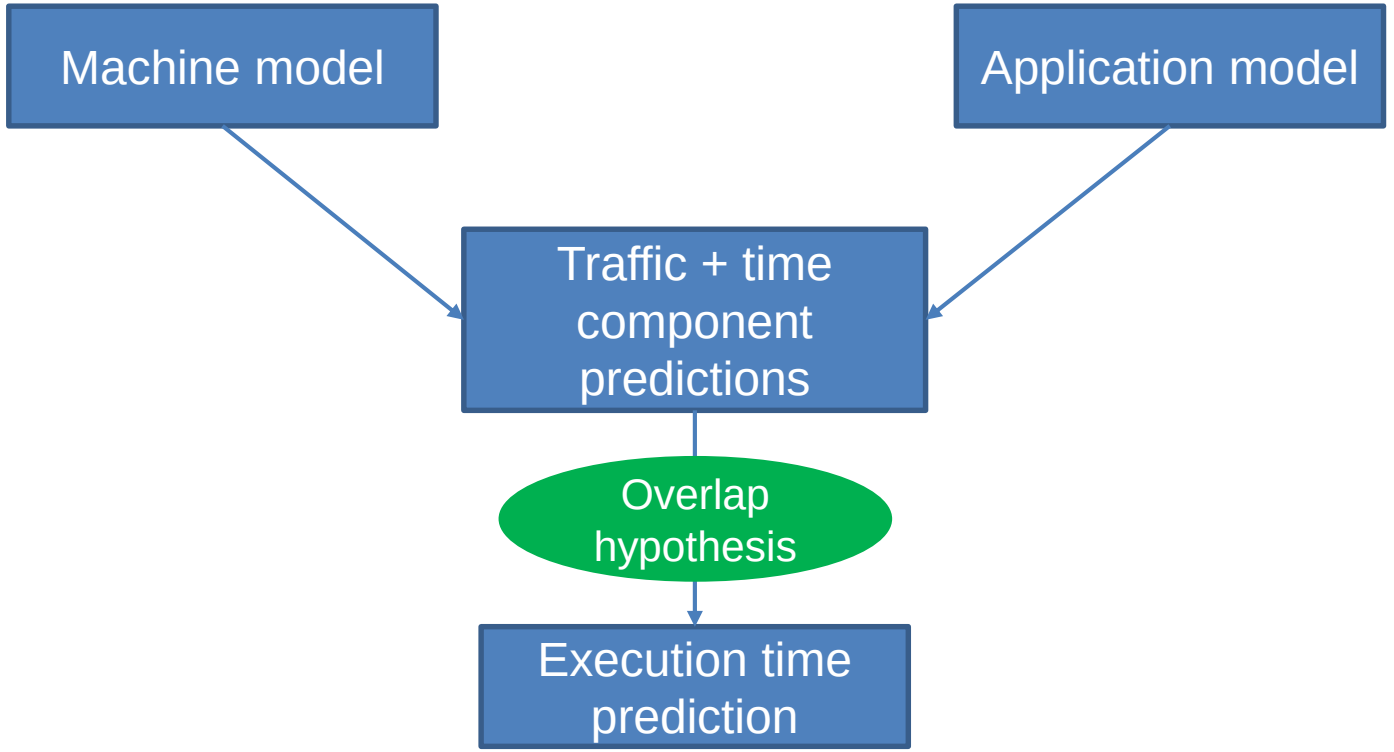
ECM model



Single-core ECM model



Execution-Cache-Memory (ECM) model helps us to understand and analyze the single-core performance.



Hofmann et.al.: *Bridging The Architecture Gap: Abstracting Performance-Relevant Properties Of Modern Server Processors*, https://doi.org/10.14529/jsfi200204

In-core prediction

Application knowledge

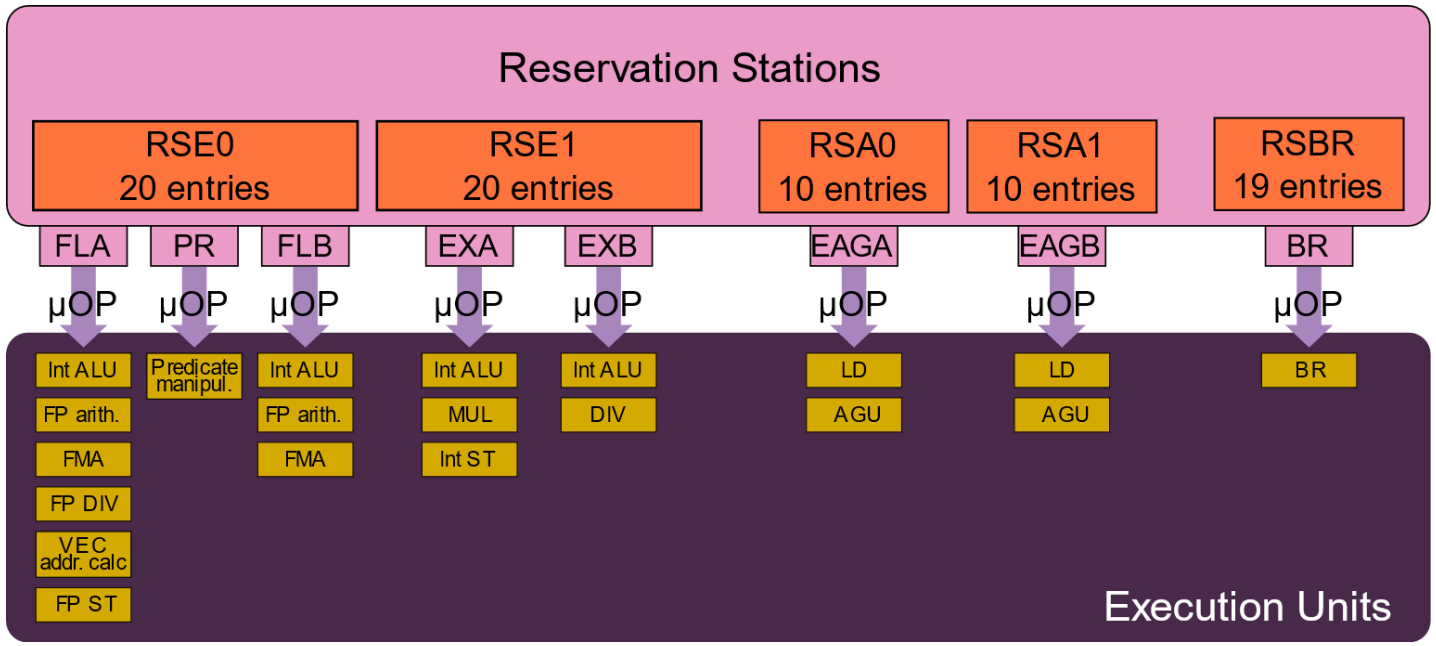
STREAM TRIAD

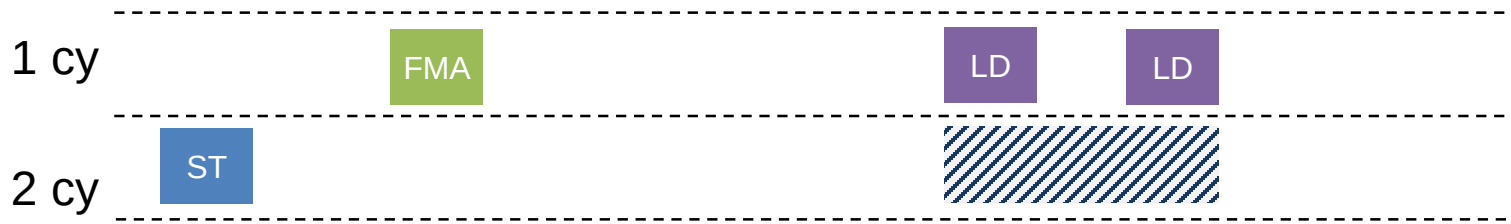
a[i] = b[i] + s * c[i]

```
.L18:
  ld1d z4.d, p5/z, [x21, x9, lsl 3]
  ld1d z5.d, p5/z, [x20, x9, lsl 3]
  fmad z5.d, p5/m, z2.d, z4.d
  st1d z5.d, p5, [x19, x9, lsl 3]
  add x8, x9, 8
  whilelo p5.d, w8, w7
  b.any .L18
```

2cy / VL

Machine knowledge





In-core prediction

Application knowledge

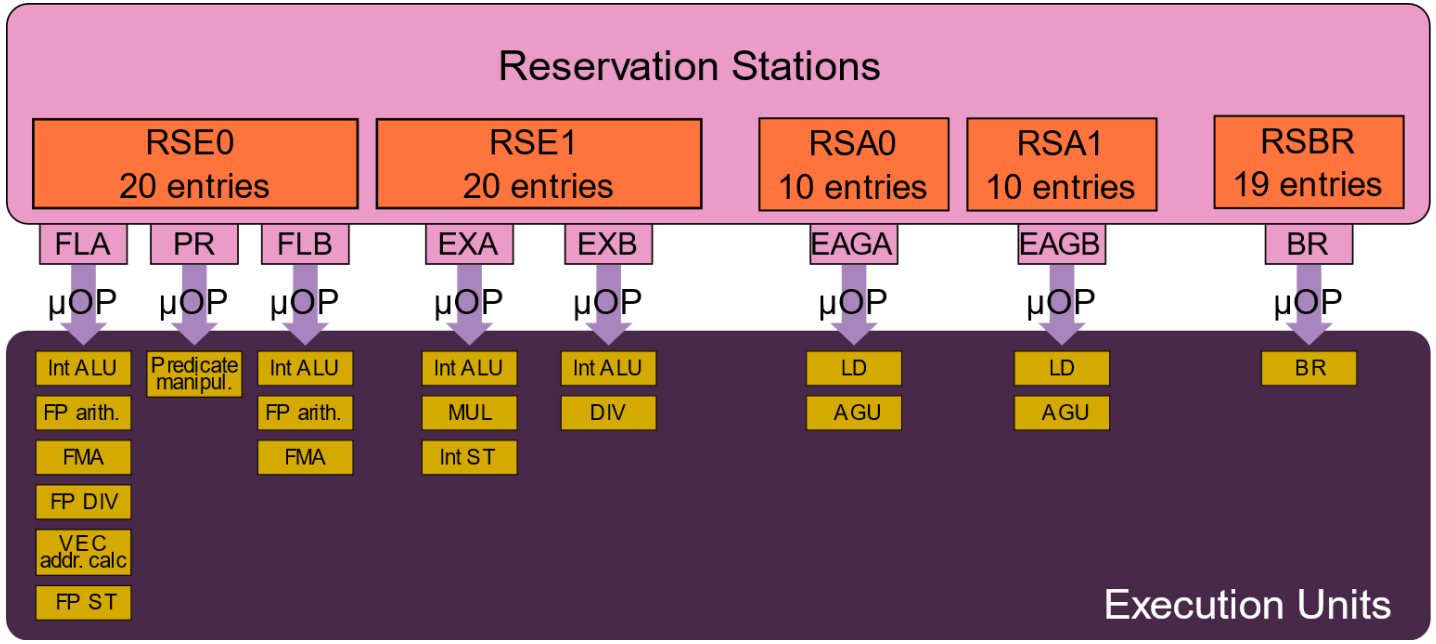
STREAM TRIAD

```
a[i] = b[i] + s * c[i]
```

```
.L18:
 ld1d z4.d, p5/z, [x21, x9, lsl 3]
 ld1d z5.d, p5/z, [x20, x9, lsl 3]
 fmad z5.d, p5/m, z2.d, z4.d
 st1d z5.d, p5, [x19, x9, lsl 3]
 add x8, x9, 8
 whilelo p5.d, w8, w7
 b.any .L18
```

2cy / VL

Machine knowledge



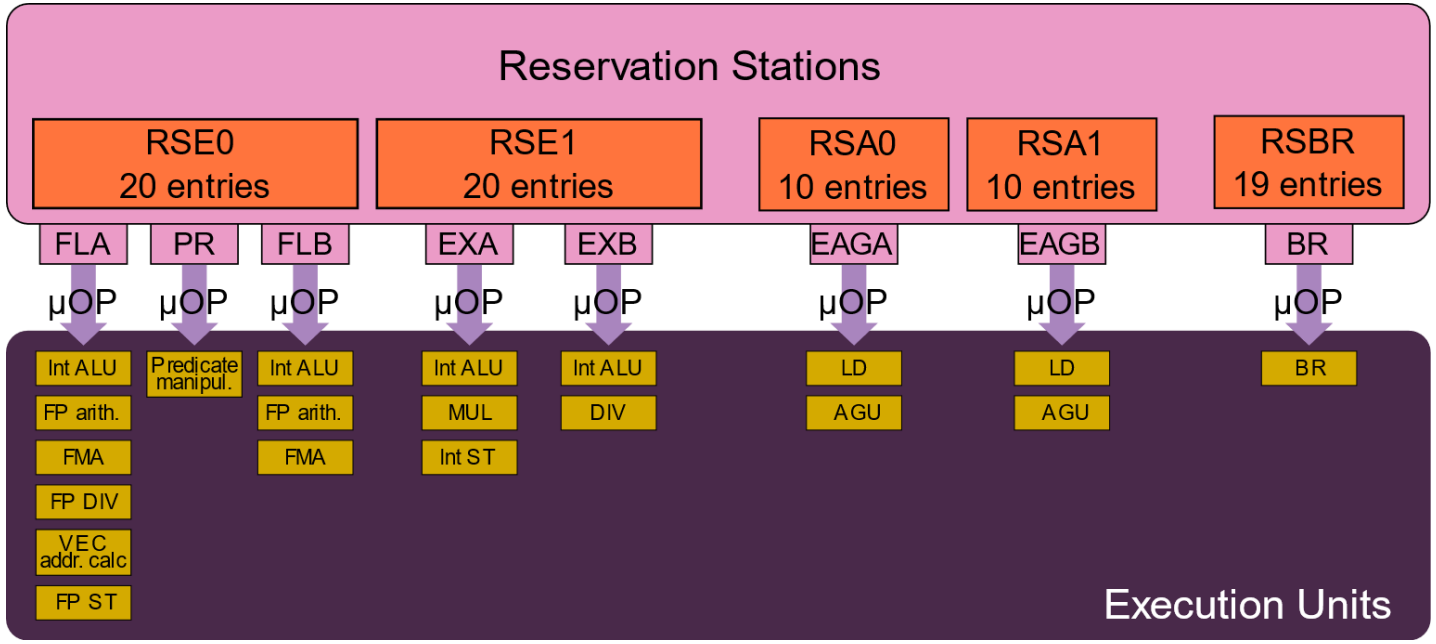
OSACA

Static analysis and prediction of in-core contribution

https://github.com/RRZE-HPC/OSACA

In-core prediction

Instruction	Reciprocal Throughput [cy]	Latency [cy]
`ld1d`	0.5	11
`st1d`	1.0	–
`fadd`	0.5	9
`fmad`	0.5	9
`faddv`	11.5	49



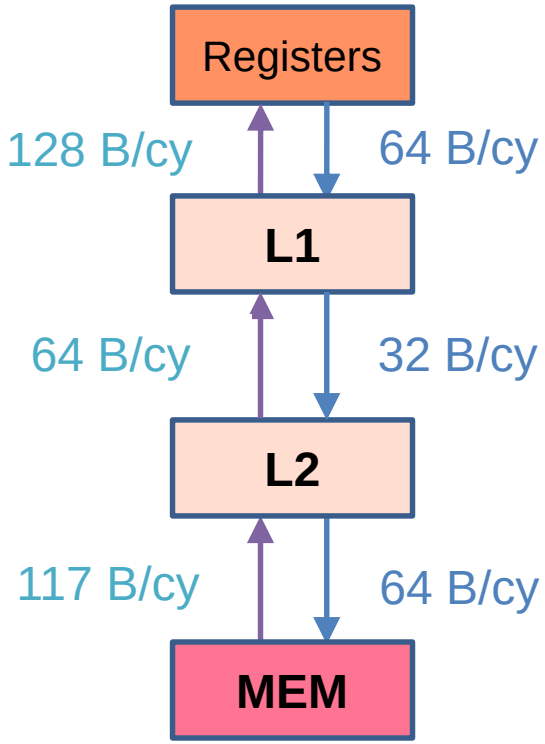
OSACA

Static analysis and prediction of in-core contribution

https://github.com/RRZE-HPC/OSACA

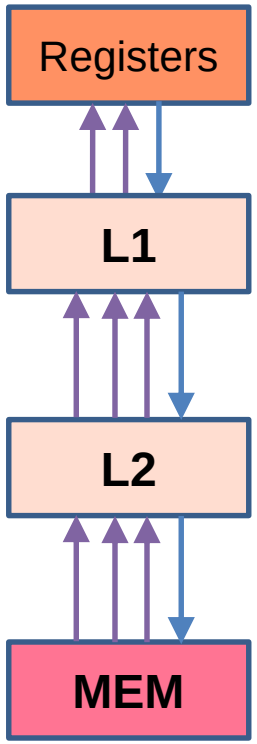
Data transfer for STREAM triad

Machine knowledge
(FX700)

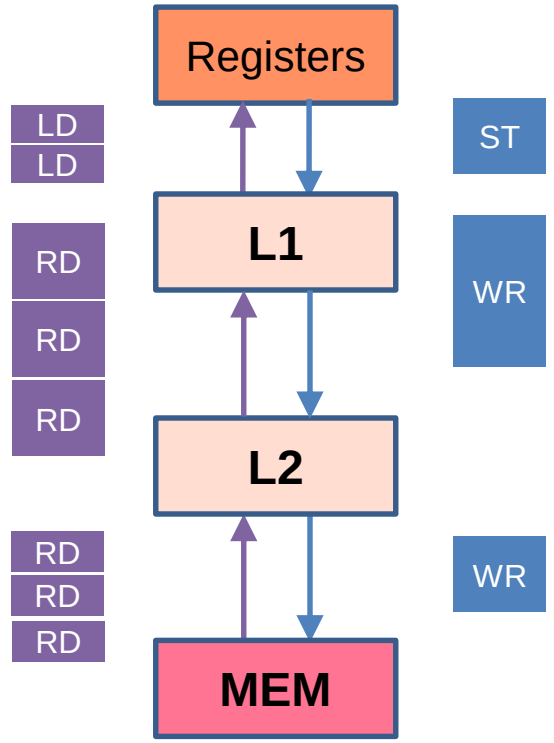


Application knowledge
STREAM triad

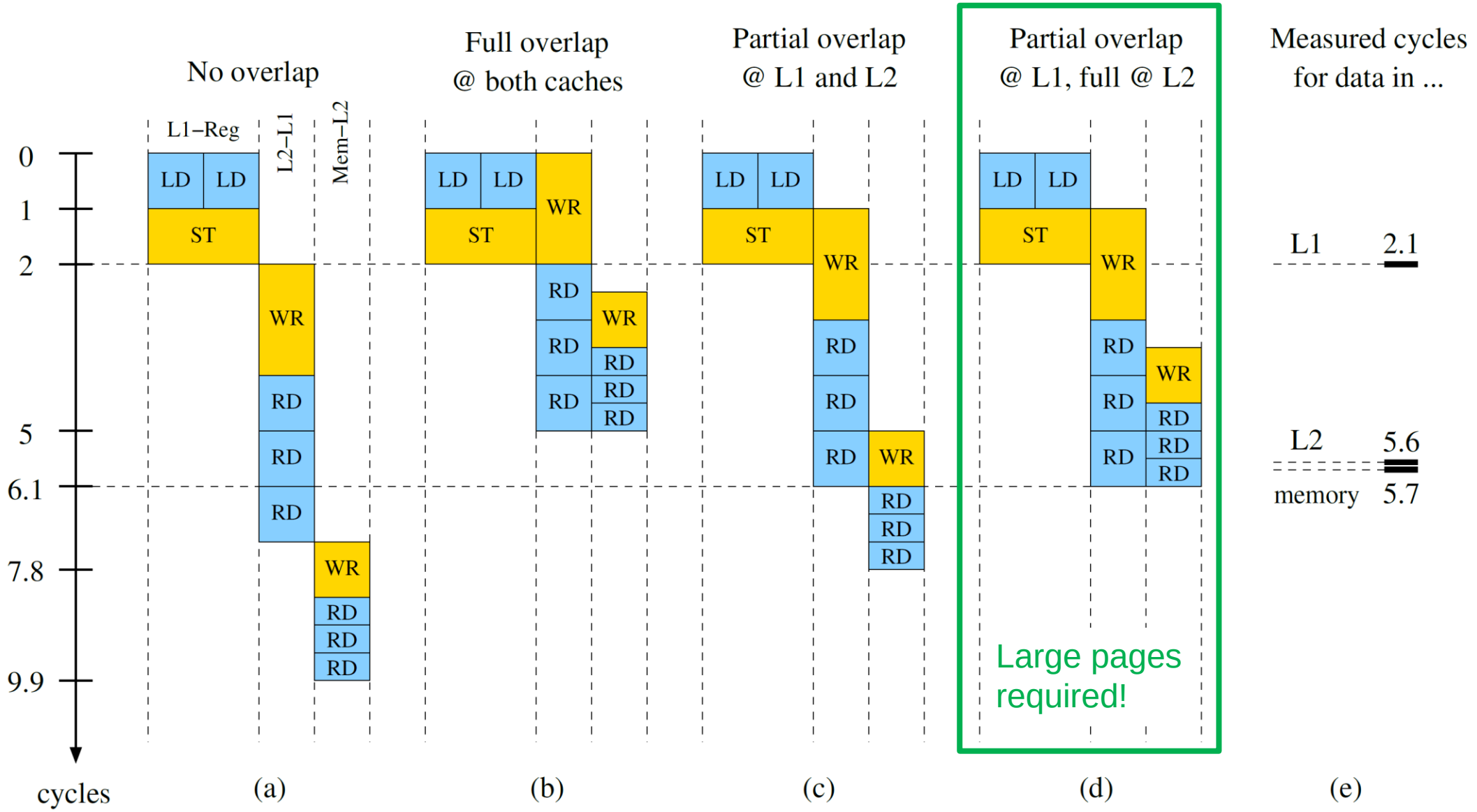
```
a[i] = b[i] + s*c[i]
```



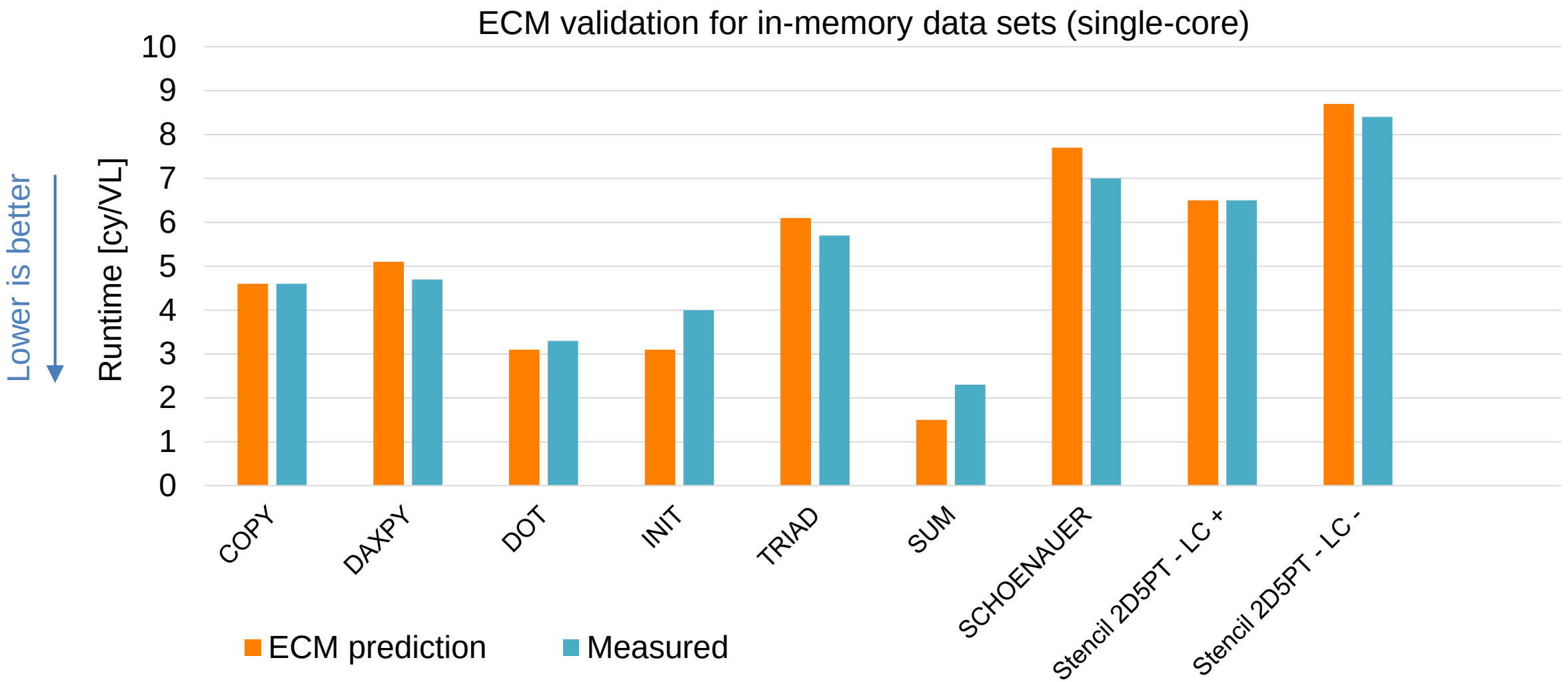
ECM prediction?
STREAM triad on A64FX



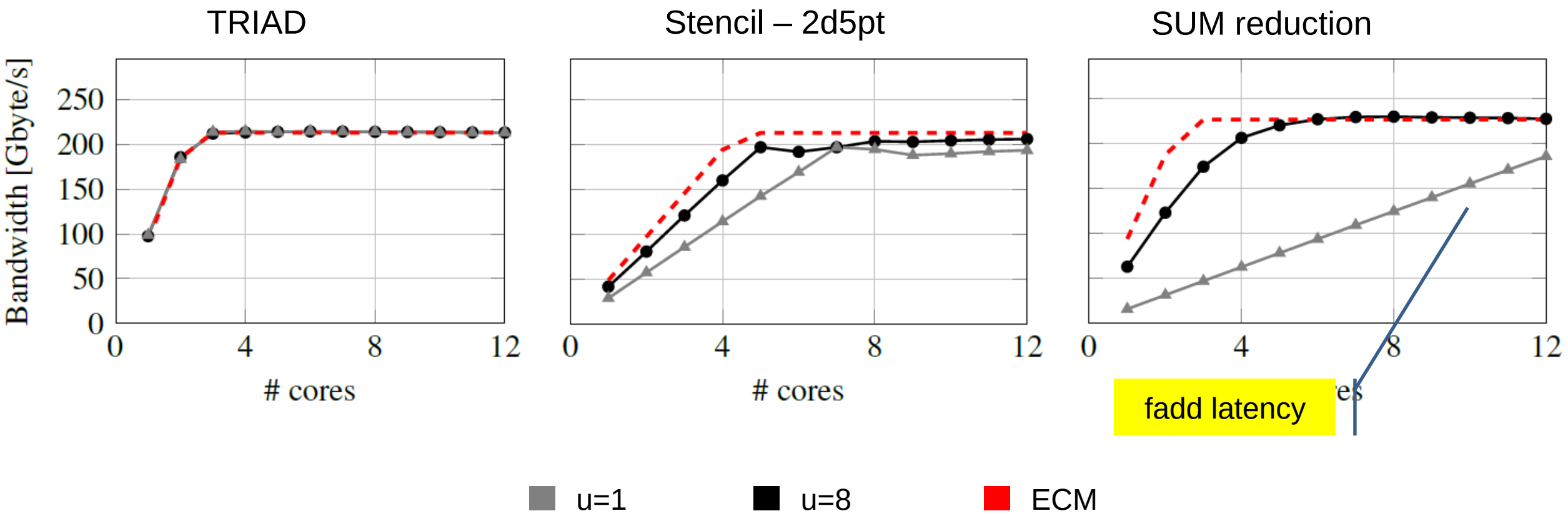
Overlap hypotheses for A64FX



Model validation (FX1000, large pages)



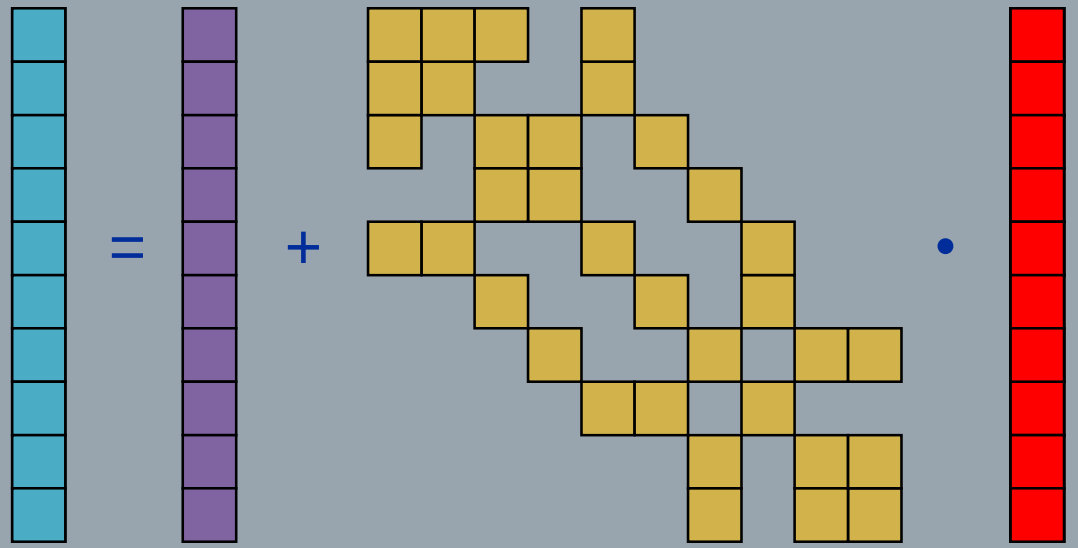
Multicore (in-memory data set)



Sufficient unrolling is crucial (but sometimes it's not enough)

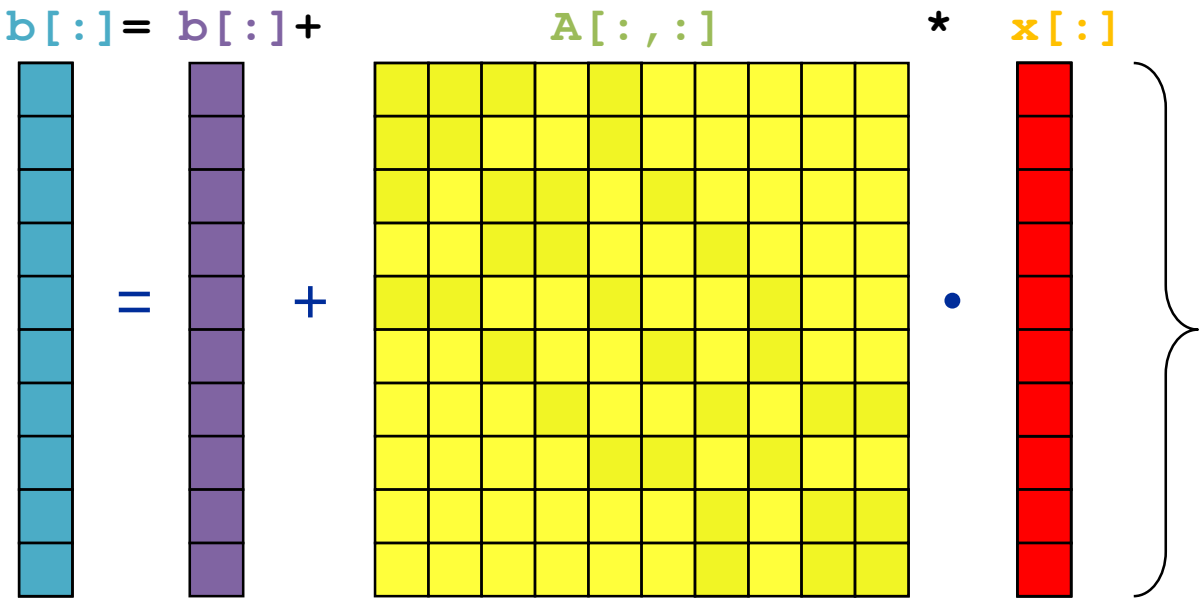
SpMV

Sparse Matrix-Vector Multiplication



SpMV

Sparse Matrix-Vector Multiplication (SpMV) : `b=Ax`



General case: some indirect addressing required!

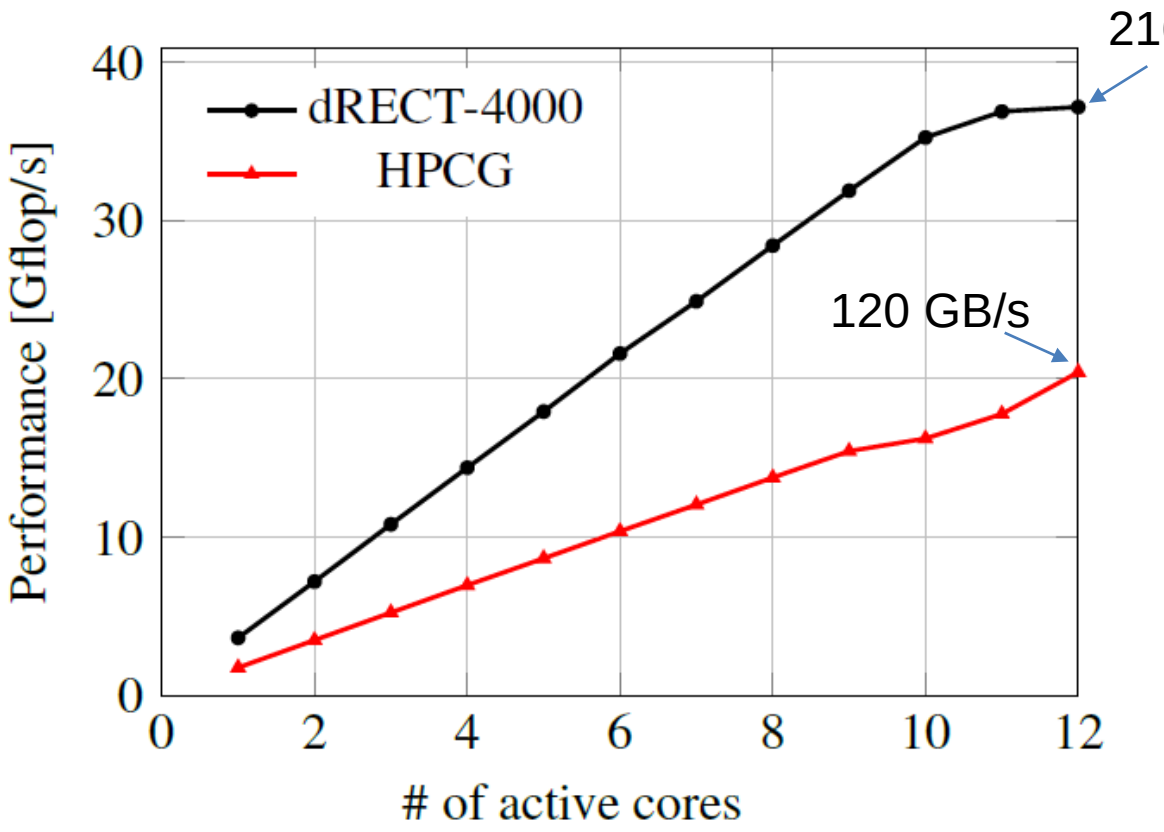
In Compressed Row Storage (CRS) format

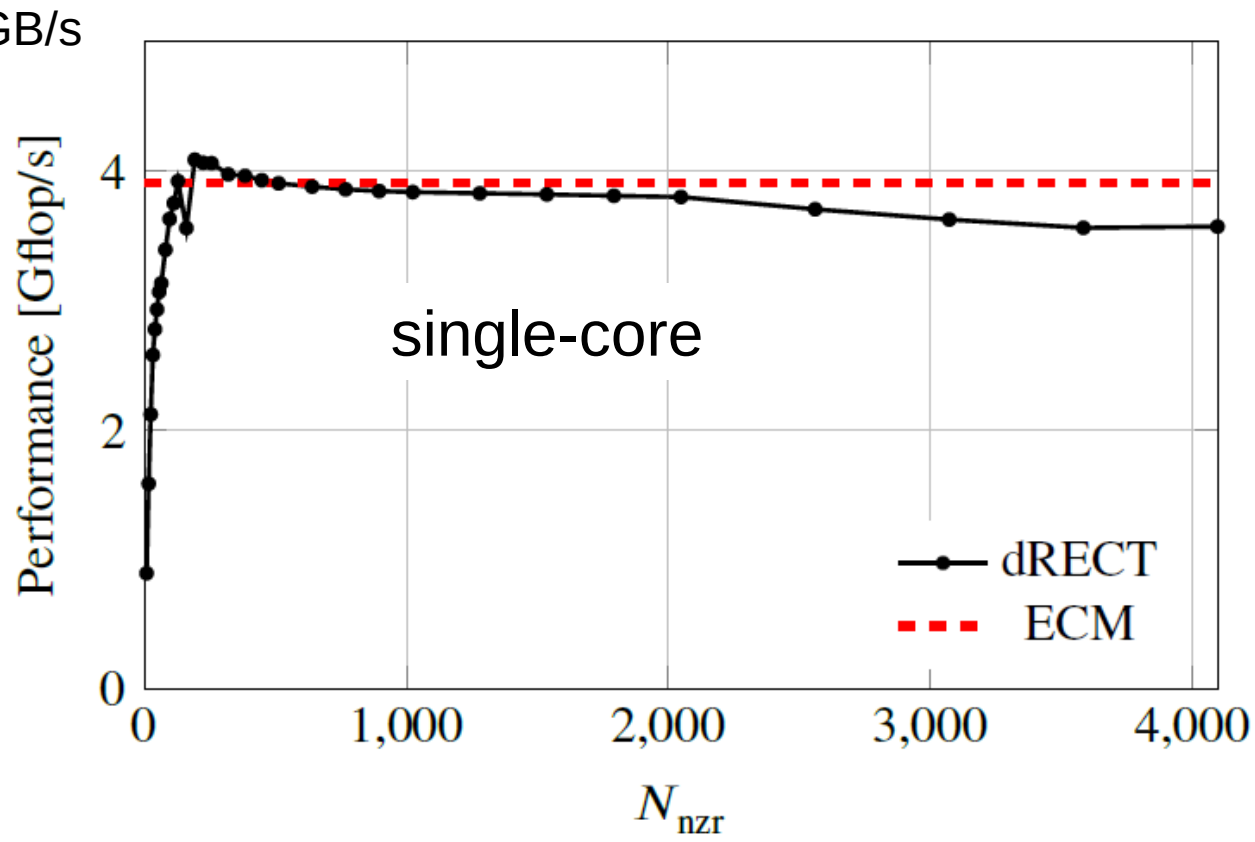
```
for i = 0:nrows-1 //Long outer loop
    for j = row_ptr[i]:row_ptr[i+1]-1 // Short inner loop
        b[i] = b[i] + A[j] * x[col_idx[j]]
```

Minimum code balance:

$$B_c^{min} = 6 \frac{\text{byte}}{\text{flop}}$$

SpMV – dRECT vs. HPCG-128³





- dRECT: 4000-column tall & skinny dense matrix ($N_{nzr} = 4000$)
- HPCG: matrix from HPCG benchmark ($N_{nzr} = 27$), 128³ rows

SpMV

Assembly of the short inner-loop

```
.L6:
        ld1sw    z0.d, p0/z, [x17, x20, lsl 2]
        ld1d     z2.d, p0/z, [x18, x20, lsl 3]
        ld1d     z3.d, p0/z, [x30, z0.d, lsl 3]
        add      x20, x20, 8
        fmla     z1.d, p0/m, z3.d, z2.d
        whilelo  p0.d, x20, x14
        b.any    .L6

        faddv    d4, p1, z1.d
```

FMA: Update **z1.d**

Latency: 9 cycles

Loop length : 27
HPCG matrix

Horizontal add of 512-bit register

latency = 49 cycles

ECM model predicts maximum bandwidth of 100 GB/s

→ No saturation

In Compressed Row Storage (CRS) format

```
for i = 0:nrows-1 //Long outer loop
    for j = row_ptr[i]:row_ptr[i+1]-1 // Short inner loop
        b[i] = b[i] + A[j] * x[col_idx[j]]
```

The problem with SpMV on A64FX

We need both:

- SIMD vectorization
- Modulo Variable Expansion (MVE)

With CRS, both must be implemented in the inner loop. The partial sums accumulation adds to the overhead.

Can we get rid of the partial sums accumulation *and* separate SIMD from MVE?

NHR FAU

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

CRS → SELL-C-σ

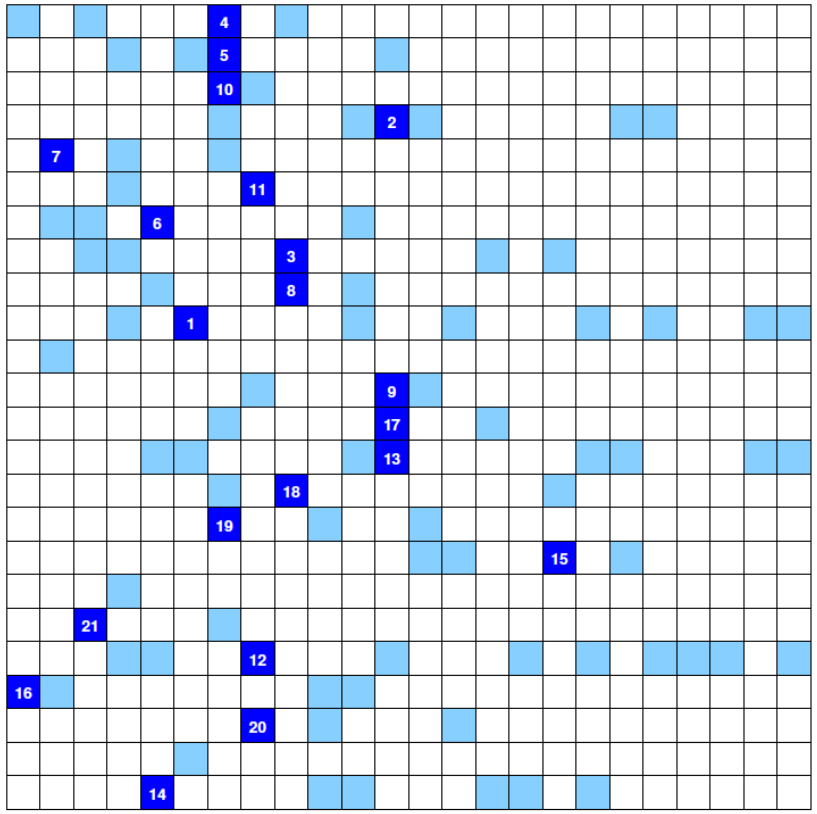
Change data storage format

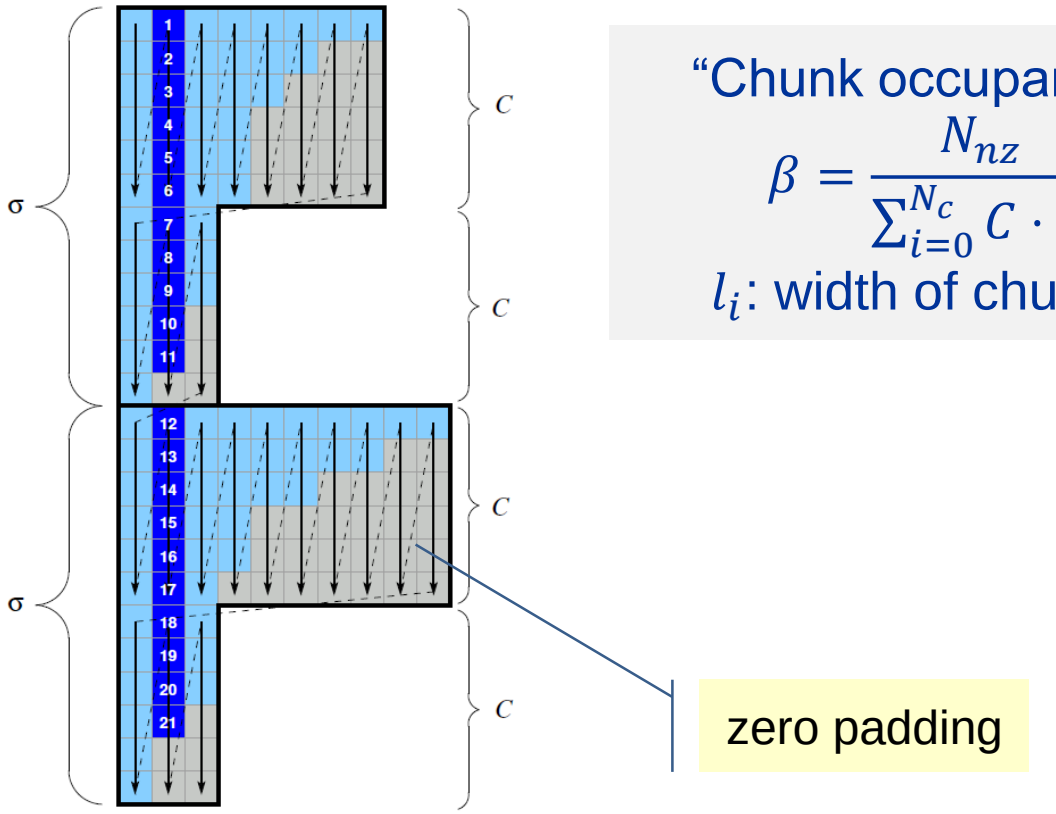
High Performance
Computing

SELL-C-σ

Idea

- Sort rows according to length within sorting scope σ
- Store nonzeros column-major in zero-padded blocks of height C





"Chunk occupancy":

$$\beta = \frac{N_{nz}}{\sum_{i=0}^{N_c} C \cdot l_i}$$

l_i: width of chunk i

SELL-C-σ kernel

Example $C = 4$ without further unrolling → longer inner loop, but still an LCD

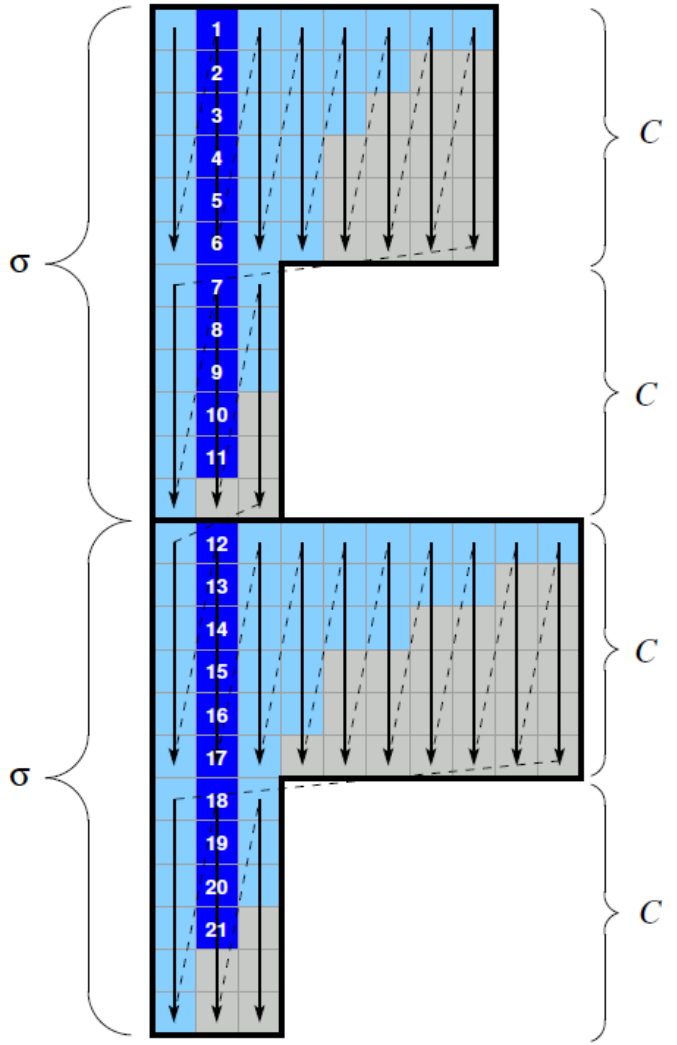
```
for(i = 0; i < N/4; ++i)
{

  for(j = 0; j < cl[i]; ++j)
  {
    y[i*4+0] += val[cs[i]+j*4+0] *
                x[col[cs[i]+j*4+0]];
    y[i*4+1] += val[cs[i]+j*4+1] *
                x[col[cs[i]+j*4+1]];
    y[i*4+2] += val[cs[i]+j*4+2] *
                x[col[cs[i]+j*4+2]];
    y[i*4+3] += val[cs[i]+j*4+3] *
                x[col[cs[i]+j*4+3]];
  }
}
```

$C = 4$

How to choose the parameters?

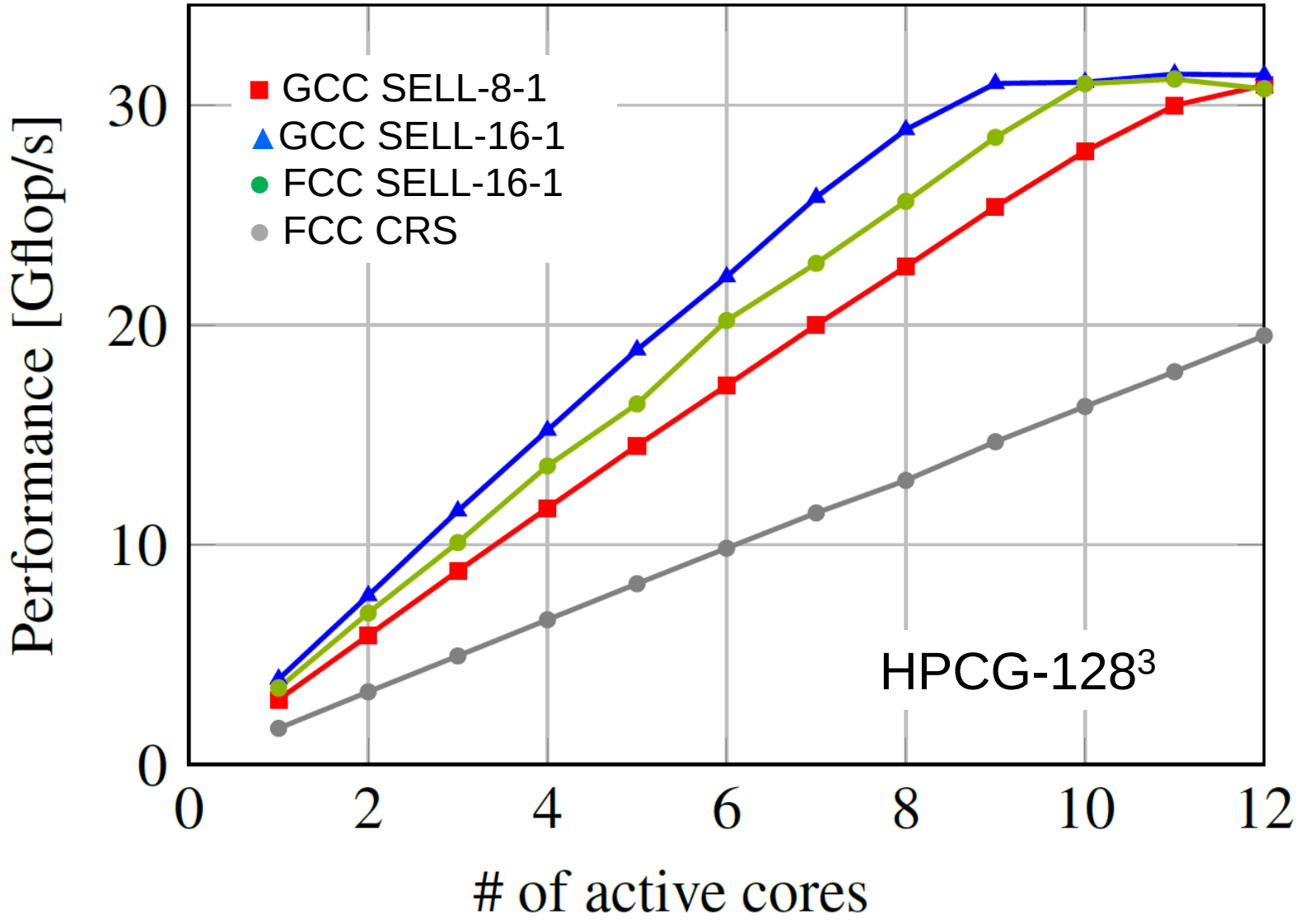
- C
  - $n \times$ SIMD width to allow good utilization of SIMD units
  - $n > 1$ useful for hiding ADD pipeline latency
- σ
  - As small as possible, as large as necessary
  - Large σ reduces zero padding (brings β closer to 1)
  - Sorting alters RHS access pattern → α depends on σ



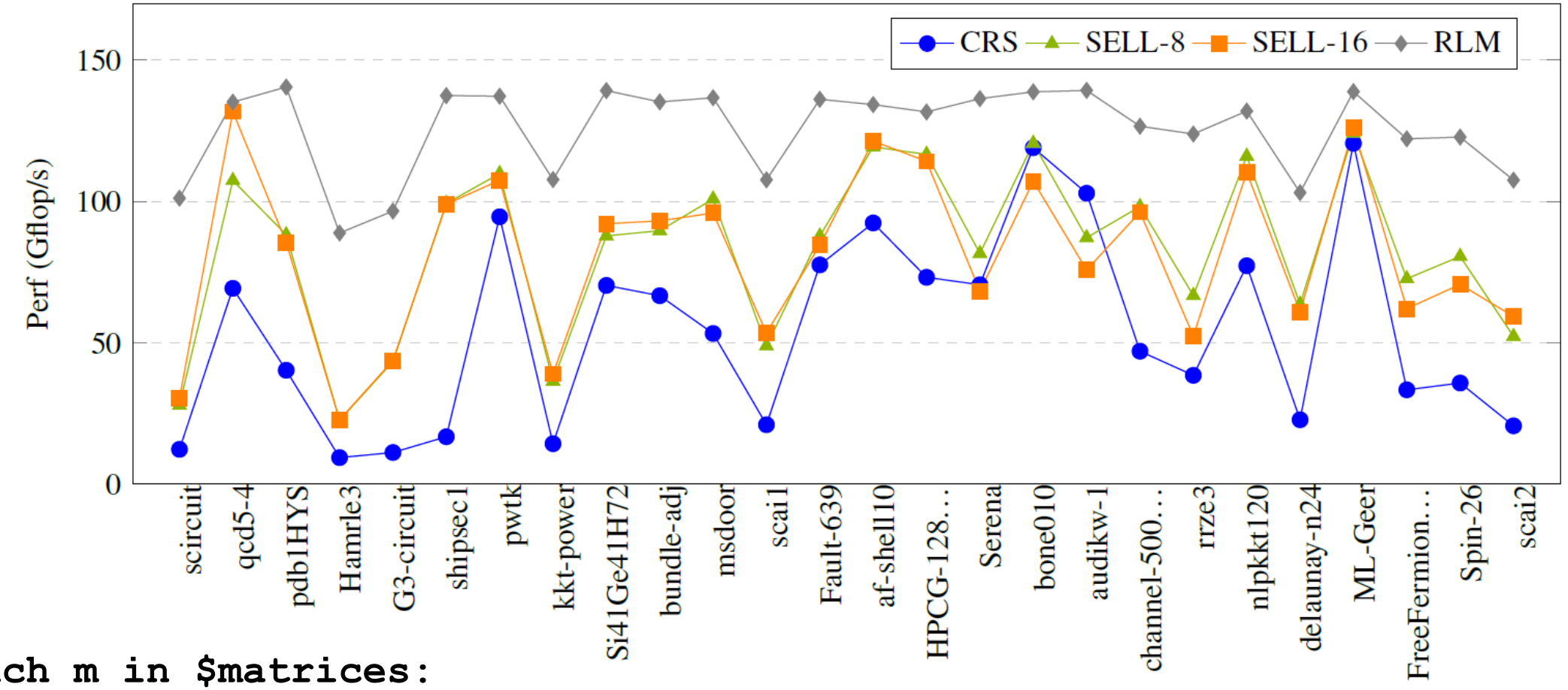
M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop: *A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide SIMD units*. SIAM Journal on Scientific Computing **36**(5), C401–C423 (2014). DOI: 10.1137/130930352,

SpMV performance with SELL-C-σ (1 CMG)

- SELL-C-σ separates SIMD from sum reduction
- C>8 allows for reduction of fmla latency impact



SpMV performance with SELL-C-σ (full chip)




```
foreach m in $matrices:
  apply RCM reordering if helpful
  try row-based vs. nonzero-based load balancing
  scan σ from 1 … 4096
```

NHR FAU

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Domain Wall (DW) kernel

from Quantum Chromodynamics (QCD)

Context

- Lattice QCD simulates the strong interaction
- Iterative multigrid techniques on regular (4D or 5D) lattices
- Core component: Apply Dirac operator D to quark-field vector Ψ
- Domain Wall (DW) formulation: quark field lives on 4D boundary of a 5D space-time volume $V_4 \times L_s$

$$(D\psi)(n,s)_{\alpha a} =$$

$$\sum_{\mu=1}\sum_{\beta=1}\sum_{b=1}\left\{U_\mu(n)_{ab}(1+\gamma_\mu)_{\alpha\beta}\psi(n+\hat{\mu},s)_{\beta b} + U_\mu^\dagger(n-\hat{\mu})_{ab}(1-\gamma_\mu)_{\alpha\beta}\psi(n-\hat{\mu},s)_{\beta b}\right\}$$

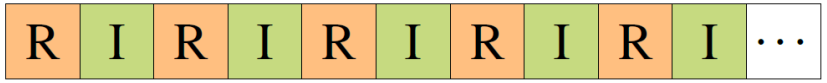
DW stencil kernel (simplified)

```
#define x_p 1 // x-plus  direction
#define x_m 2 // x-minus direction
#define y_p 3 // y-plus  direction
...
#pragma omp parallel for schedule(static)
for {t,z,y,x} = 1:{L_t-2,L_z-2,L_y-2,L_x-2}//collapsed loop over 4d space-time
{
  for(int s=0; s<L_s; ++s) //loop over 5th dimension
  {
    O[t][z][y][x][s] = R(x_p)·U[x_p][t][z][y][x]·P(x_p)·I[t][z][y][x+1][s] +
                       R(x_m)·U[x_m][t][z][y][x]·P(x_m)·I[t][z][y][x-1][s] +
                       R(y_p)·U[y_p][t][z][y][x]·P(y_p)·I[t][z][y+1][x][s] +
                       R(y_m)·U[y_m][t][z][y][x]·P(y_m)·I[t][z][y-1][x][s] +
                       R(z_p)·U[z_p][t][z][y][x]·P(z_p)·I[t][z+1][y][x][s] +
                       R(z_m)·U[z_m][t][z][y][x]·P(z_m)·I[t][z-1][y][x][s] +
                       R(t_p)·U[t_p][t][z][y][x]·P(t_p)·I[t+1][z][y][x][s] +
                       R(t_m)·U[t_m][t][z][y][x]·P(t_m)·I[t-1][z][y][x][s];
  }
}
```

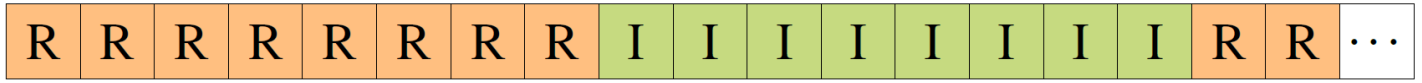
- "Grid" lattice QCD framework
- Uses SVE intrinsics
- Data type: double complex

Complex numbers data layout choice

RIRI (standard)



RRII

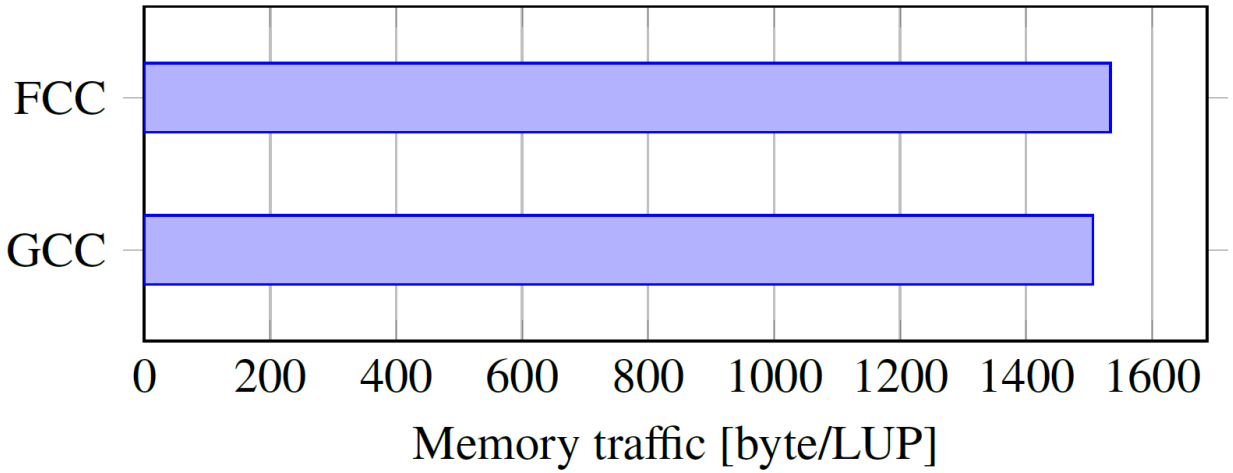


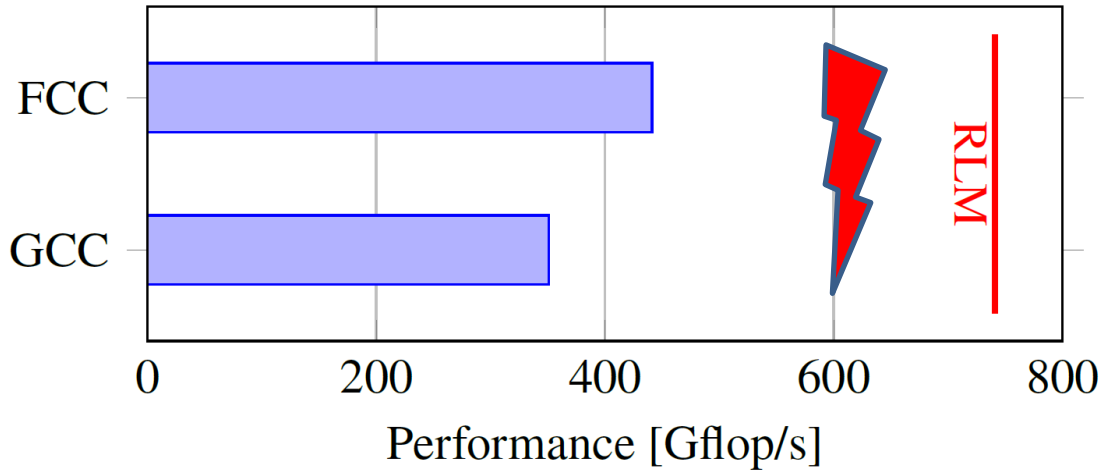
Observed performance

- Starting point: RIRI layout, ACLE intrinsics, GCC/FCC
- 1320 flops/LUP (theoretical)
- Measured code balance: 1500 byte/LUP

$$B_c \approx 1.14 \frac{\text{byte}}{\text{flop}}$$

- A64FX (FX1000): $B_m = 0.25 \frac{\text{byte}}{\text{flop}}$ → expect memory bound





Layer Conditions (LC) analysis

- LC determine traffic to/from different caches (i) of size s_i
- $w = 2$: write-allocate factor (would be 1 if WA evasion applies)
- RIRI (RRII) layout has $d = 1$ ($d = 2$)

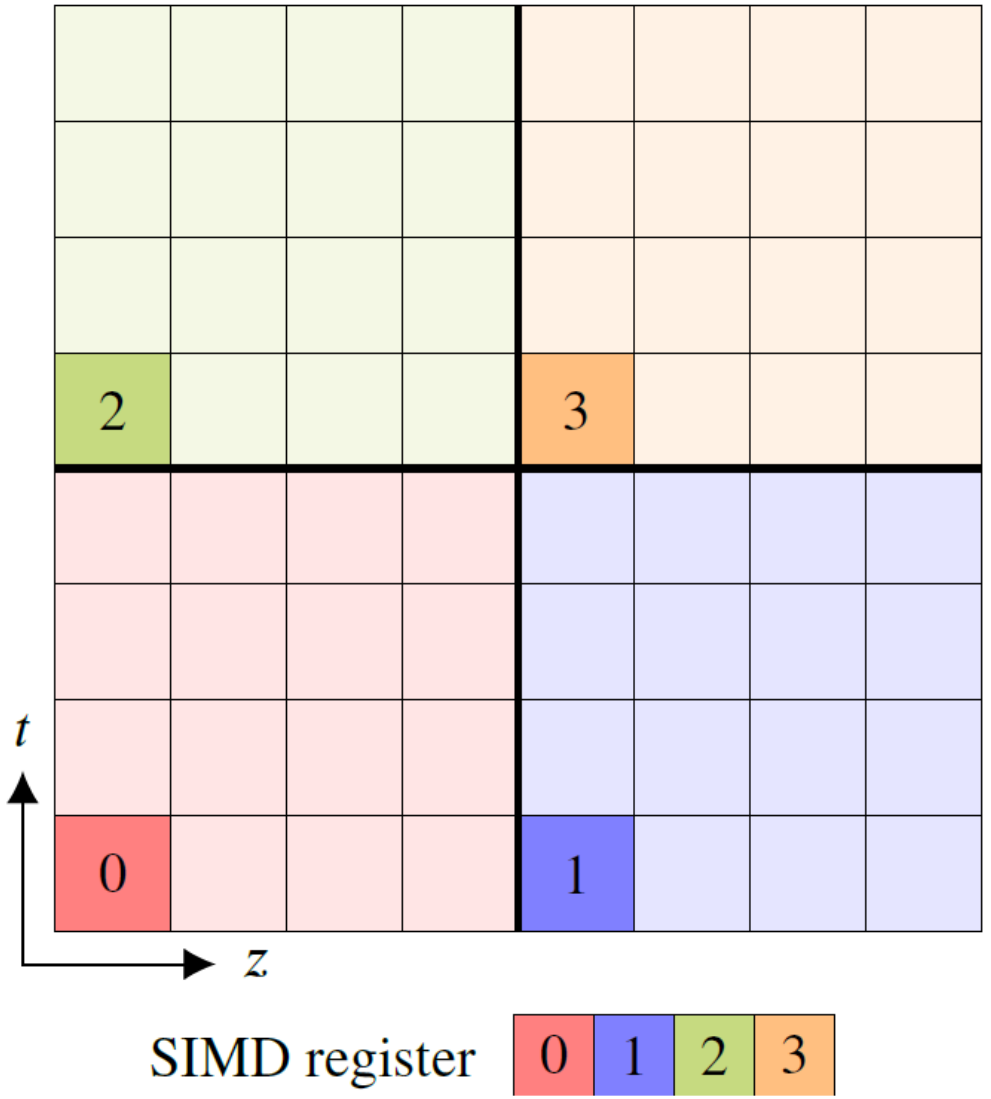
Name	V_j **in byte/LUP**	$s_i >$	
		Scalar code	512-bit vectorized code
no reuse	$(8 \cdot 9 + 8 \cdot 12 + w \cdot 12) \cdot 16$	0	0
LC_s	$(8 \cdot 9/L_s + 8 \cdot 12 + w \cdot 12) \cdot 16$	2880	11520
LC_x	$(8 \cdot 9/L_s + 7 \cdot 12 + w \cdot 12) \cdot 16$	$2 \cdot L_s(8 \cdot 9/L_s + 8 \cdot 12 + 12) \cdot 16$	$d \cdot 8 \cdot L_s(8 \cdot 9/L_s + 8 \cdot 12 + 12) \cdot 16$
LC_y	$(8 \cdot 9/L_s + 5 \cdot 12 + w \cdot 12) \cdot 16$	$2 \cdot L_s \cdot L_x(8 \cdot 9/L_s + 7 \cdot 12 + 12) \cdot 16$	$d \cdot 8 \cdot L_s \cdot L_x(8 \cdot 9/L_s + 7 \cdot 12 + 12) \cdot 16$
LC_z	$(8 \cdot 9/L_s + 3 \cdot 12 + w \cdot 12) \cdot 16$	$2 \cdot L_s \cdot L_x \cdot L_y(8 \cdot 9/L_s + 5 \cdot 12 + 12) \cdot 16$	$8 \cdot L_s \cdot L_x \cdot L_y(8 \cdot 9/L_s + 5 \cdot 12 + 12) \cdot 16$
LC_t	$(8 \cdot 9/L_s + 1 \cdot 12 + w \cdot 12) \cdot 16$	$2 \cdot L_s \cdot L_x \cdot L_y \cdot L_z(8 \cdot 9/L_s + 3 \cdot 12 + 12) \cdot 16$	$4 \cdot L_s \cdot L_x \cdot L_y \cdot L_z(8 \cdot 9/L_s + 3 \cdot 12 + 12) \cdot 16$

SIMD-friendly data layout

- One partition per SIMD lane
- Partition size:

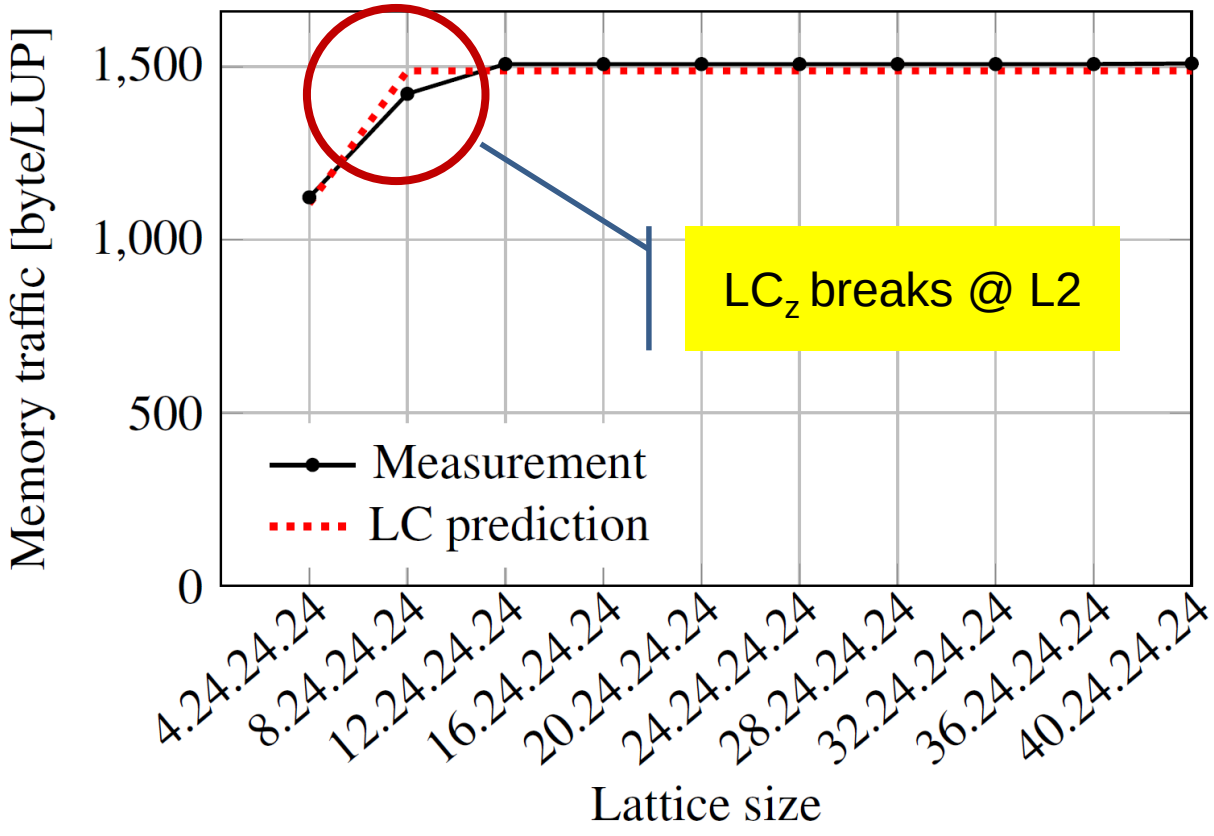
$$L_x \times L_y \times \frac{L_z}{2} \times \frac{L_t}{2} \times L_s$$

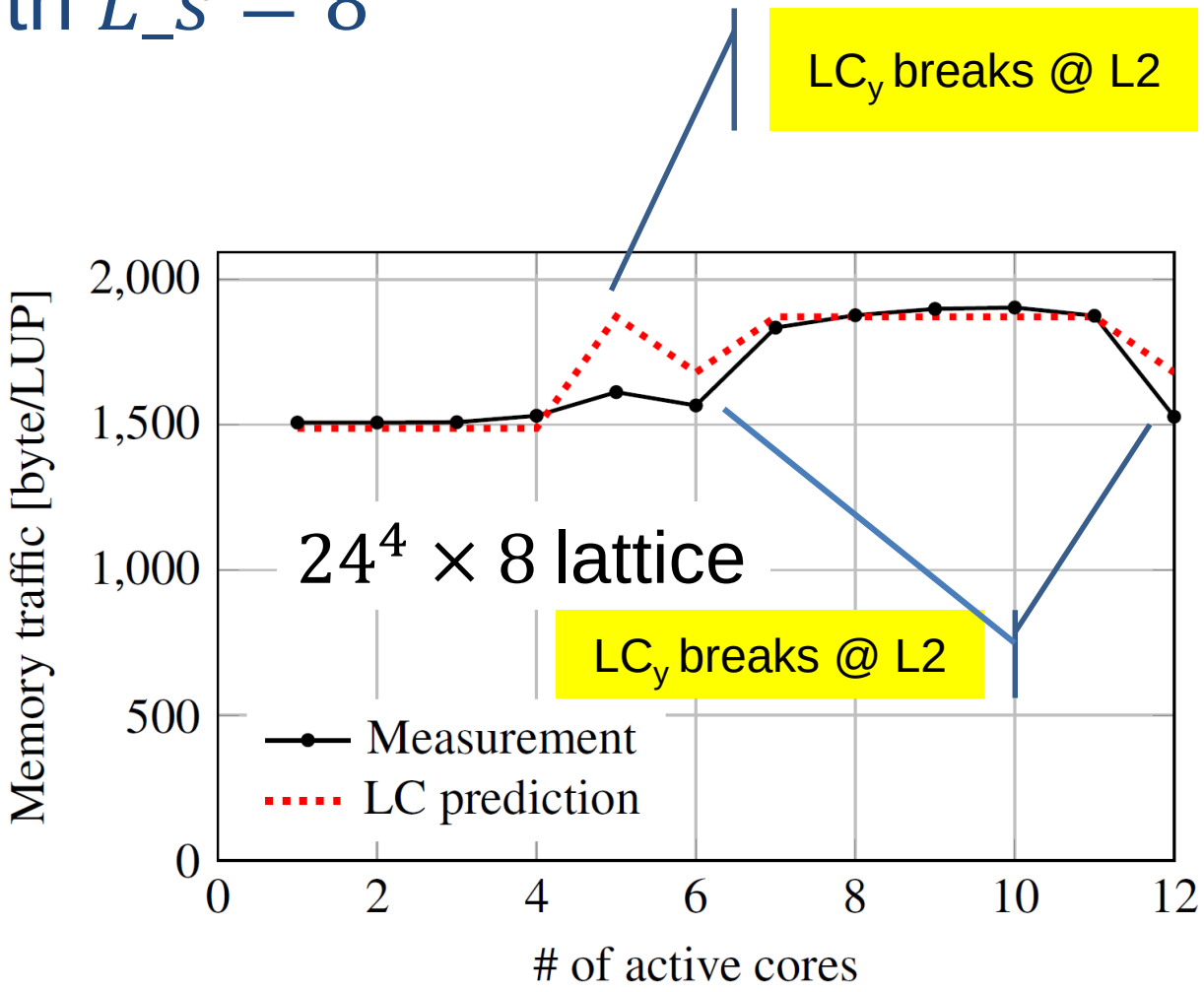
- → SIMD makes LCs more stringent (need more cache to fulfill)



LC effect

Measurement of code balance with $L\_s = 8$

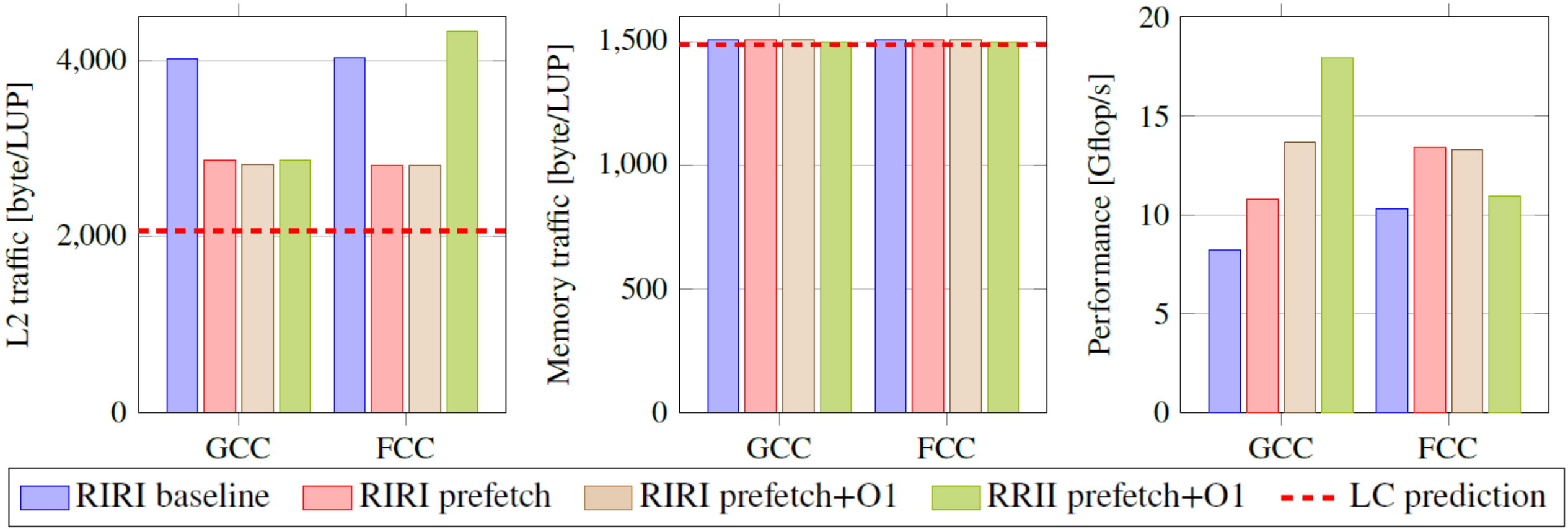




Summary of optimizations for DW

- Software prefetching decreases L2 data volume
- `-O1` makes compiler obey the ordering hints in the computational kernel (more efficient OoO execution)
- RRII data layout
  - Prevents use of complex arithmetic instructions **`fcmla`**/**`fcadd`**
  - Removes imbalance between FLA and FLB ports in the core
  - Some register spills occur, but still better than RIRI
  - Measurement falls short of ECM prediction by 2.3x (GCC) or 3.1x (FCC)

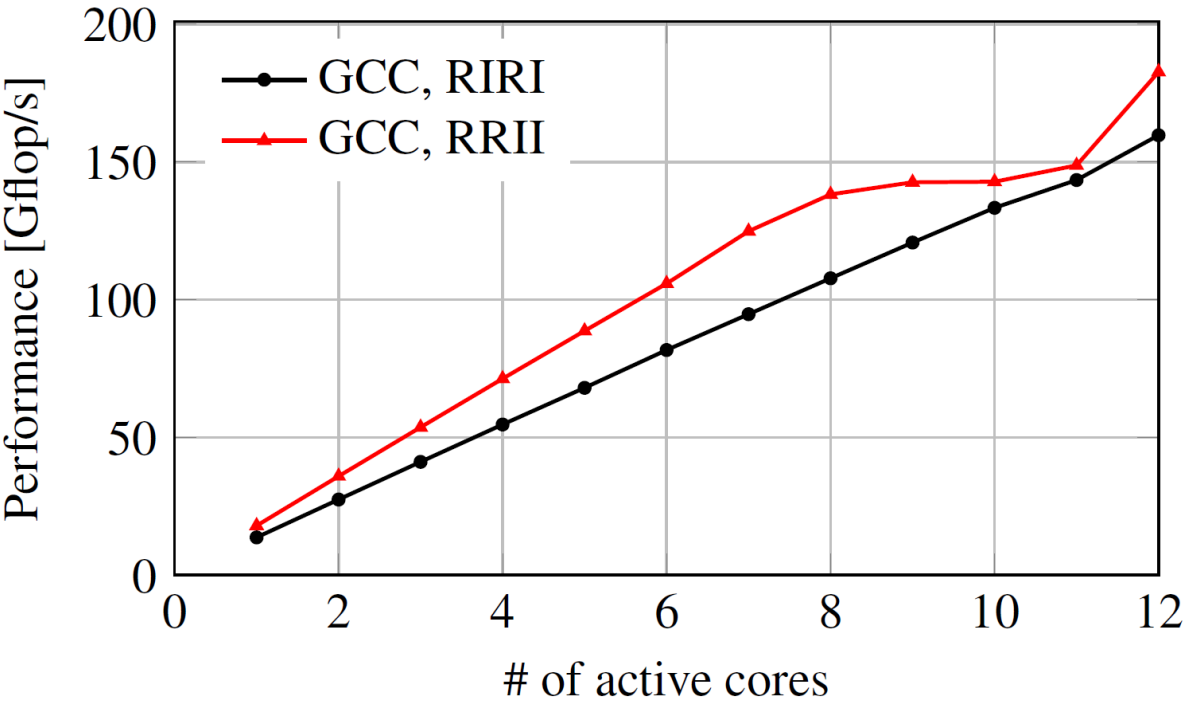
DW kernel optimizations and ECM model

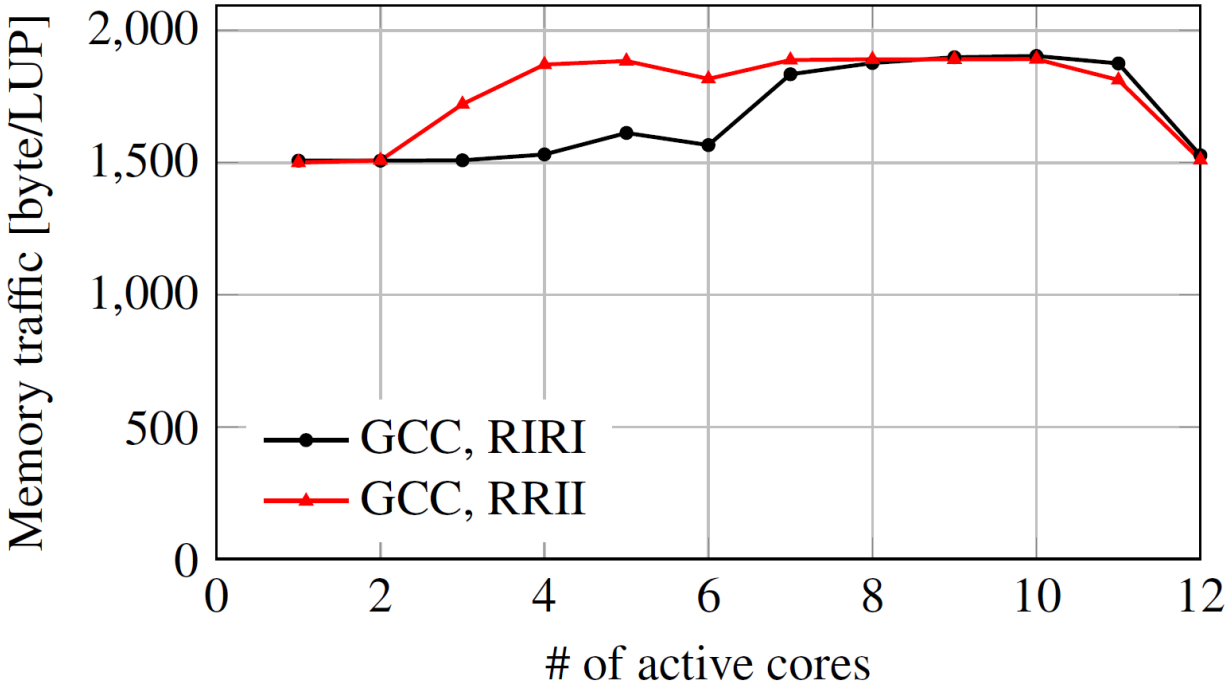


Implementation	GCC						FCC					
	T_{c_OL}	T_{L1_LD}	T_{L1_ST}	T_{L2}	T_{Mem}	T_{ECM}	T_{c_OL}	T_{L1_LD}	T_{L1_ST}	T_{L2}	T_{Mem}	T_{ECM}
`RIRI-prefetch`	168.0	25.6	3	35.3	15.9	168.0	168.0	33	3	35.3	15.9	168.0
`RRII-prefetch`	70.8	34.4	20.4	35.3	15.9	70.8	85.5	45.2	37.3	35.3	15.9	85.5

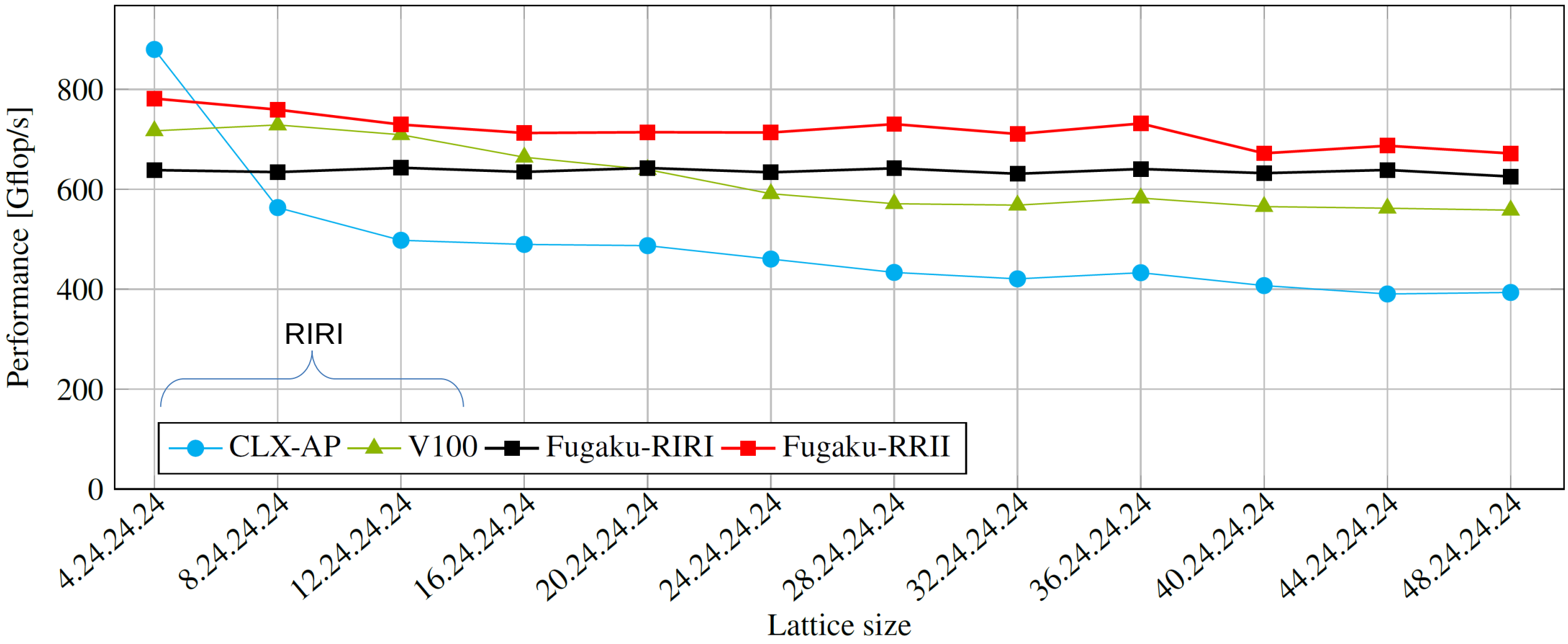
CMG performance RIRI vs. RRII

- RRII saturates already at 8 cores
- Sharing across cores in L2 gives slight increase @ 12 cores





Comparison with other architectures



Summary

- ECM model constructed for single-core performance of A64FX
- Partially overlapping memory hierarchy → high single-core memory bandwidth (even more so with large pages)
- If performance is bad, the single-core performance is usually the culprit
- SpMV requires proper data format for efficient single-core execution
- DW kernel benefits from prefetching and OoO improvements
- Performance modeling is invaluable for navigating optimization efforts

NHR FAU

FRIEDRICH-ALEXANDER
UNIVERSITÄT
ERLANGEN-NÜRNBERG

Thank You.

RIKEN
R-CCS
RIKEN
Center for
Computational Science

SFB
TRR55

KONWIHR