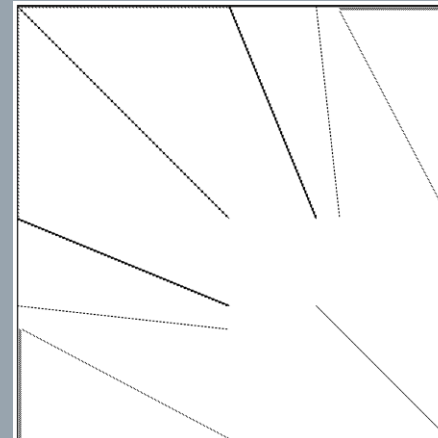
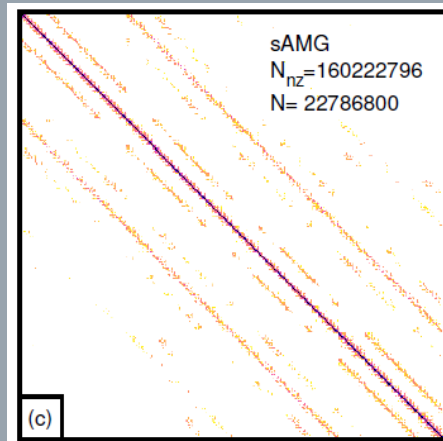
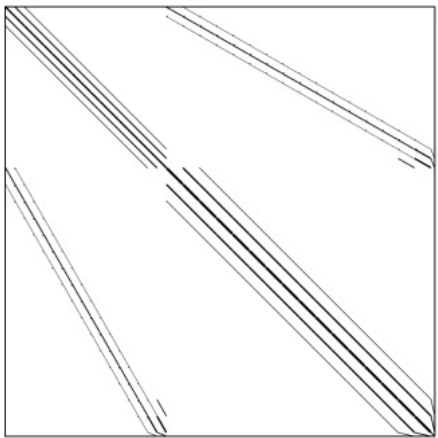


Sparse Matrix-Vector Multiplication

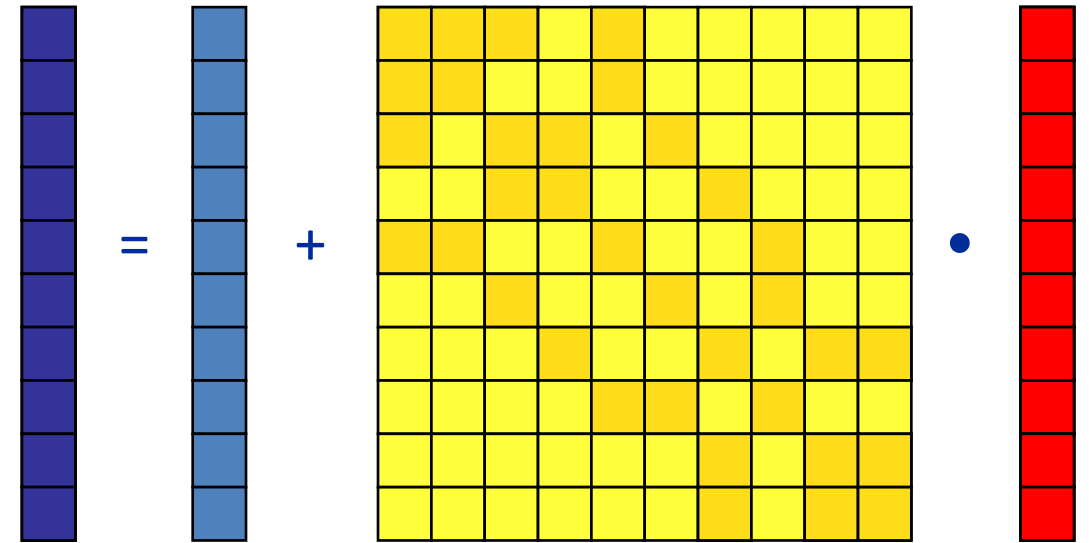
$$y = A x$$



Motivation

Algorithm 2. HPCG

```
1: while  $k \leq \text{iter}$  &  $r_{\text{norm}}/r_0 > \text{tol}$  do
2:    $z = \text{MG}(A, r)$ 
3:    $\text{oldrtz} = \text{rtz}$ 
4:    $\text{rtz} = \langle r, z \rangle$ 
5:    $\beta = \text{rtz} / \text{oldrtz}$ 
6:    $p = \beta * p + z$ 
7:    $Ap = A * p$ 
8:    $pAp = \langle p, Ap \rangle$ 
9:    $\alpha = \text{rtz} / pAp$ 
10:   $x = x + \alpha * p$ 
11:   $r = r - \alpha * Ap$ 
12:   $r_{\text{norm}} = \langle r, r \rangle$ 
13:   $r_{\text{norm}} = \text{sqrt}(r_{\text{norm}})$ 
14:   $k++$ 
```



Performance Modelling?
Optimal Performance?
Performance Optimizations?

Our plan for today

- The Roofline Model – revisited
- Performance Engineering for SpMV – CPU
- Data layout considerations – GPUs

Boundary conditions:

- Node-level (OpenMP / CUDA)
- Application problems / matrices: Standard collection / own work

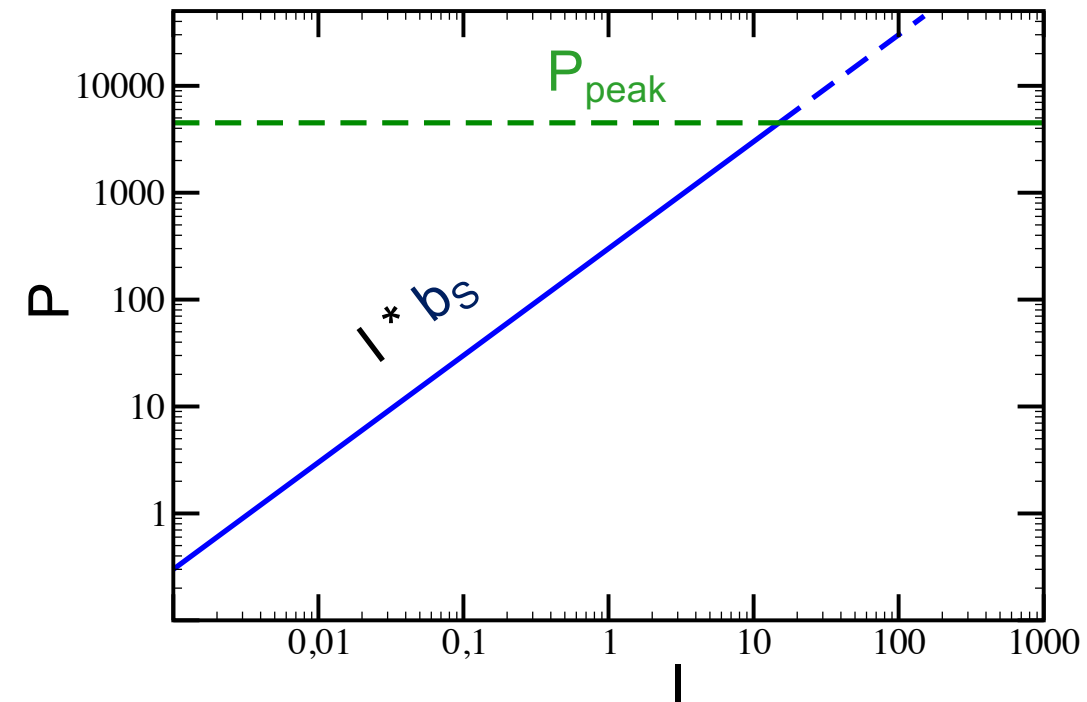
The Roofline Model – revisited



Roofline Performance Model

- When to apply: Single chip or compute node
- Performance bottleneck: Executing Flops or transferring data
- Maximum attainable performance $P = \min (P_{\text{peak}}, I * b_S)$ with

- Peak Performance: P_{peak} [Flop/s]
- Memory bandwidth: b_S [Byte/s]
- Computational Intensity: I [Flop/Byte]
(Code Balance: $B_C = I^{-1}$ [Byte/Flop])

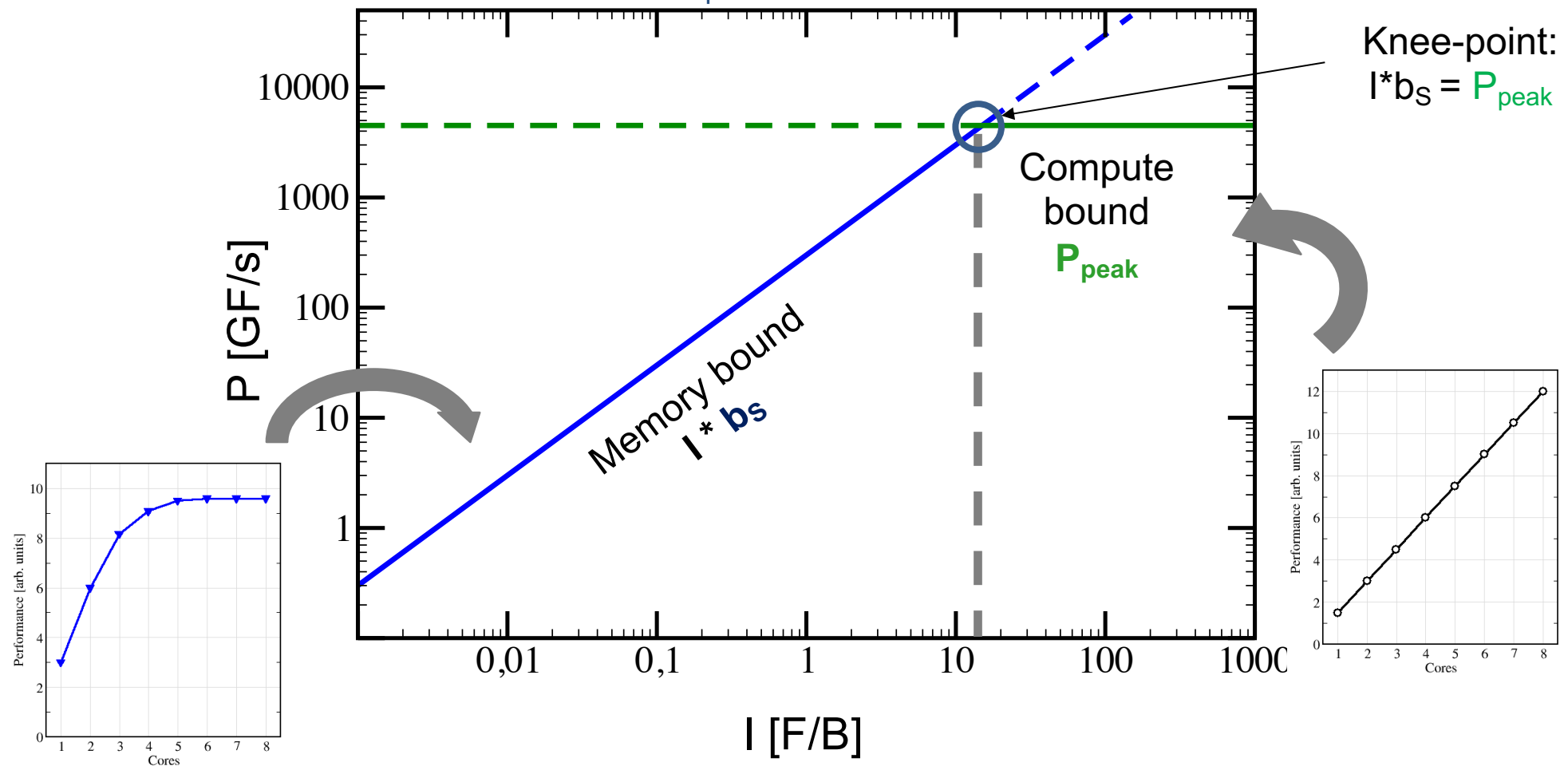


R.W. Hockney and I.J. Curington. Parallel Computing 10, 277-286 (1989).
S. Williams. UCB Tech. Report No. UCB/EECS-2008-164. PhD thesis (2008)
S. Williams, A. Waterman, and D. Patterson. 2009. Commun. ACM 52, 4 (April 2009)

Roofline Model – Characteristic behaviour

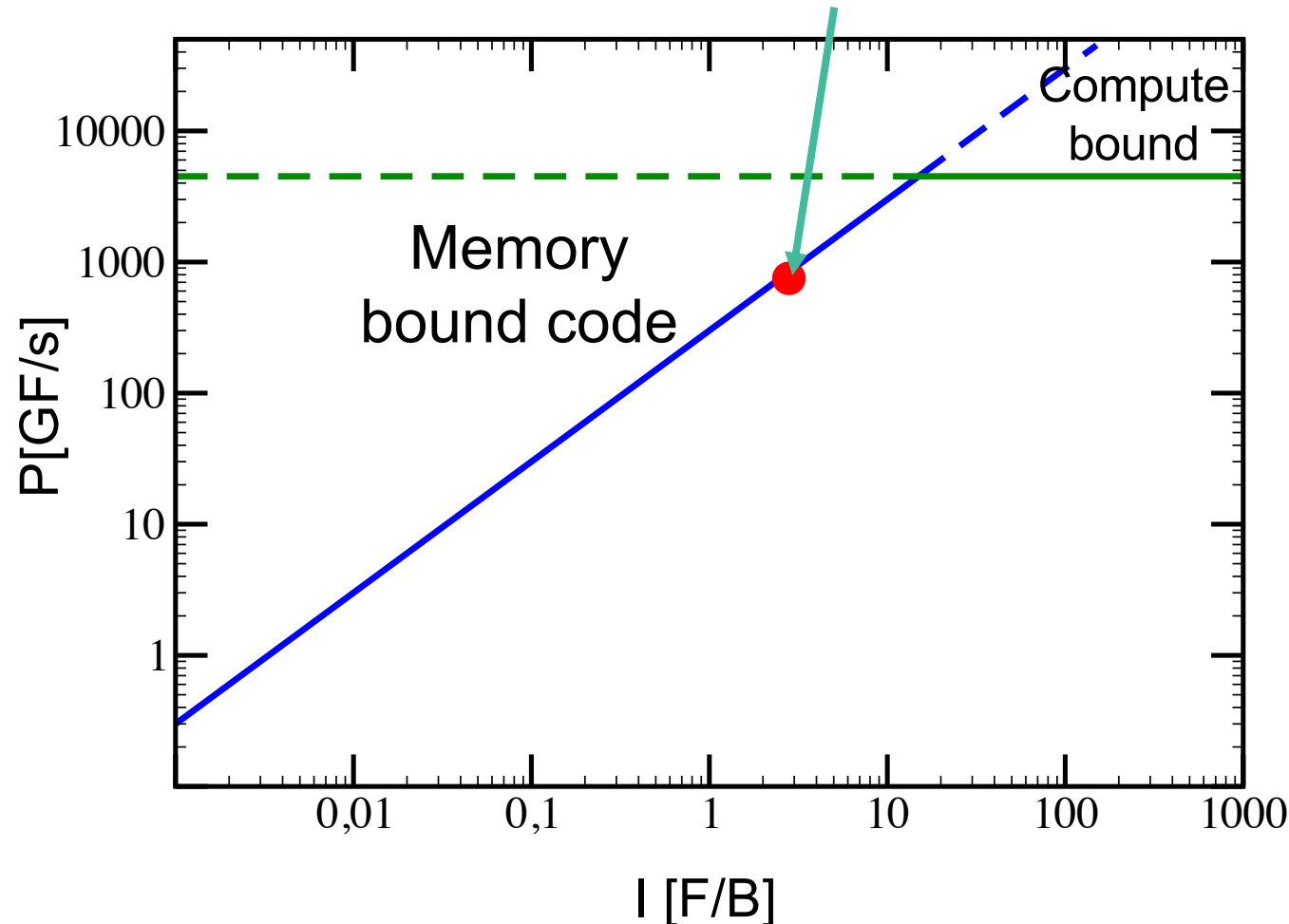
Machine model with $P_{\text{peak}}=4,5 \text{ TF/s}$ and $b_s=300 \text{ GB/s}$

(Deflt Blue: $P_{\text{peak}}=3 \text{ TF/s}$ and $b_s=256 \text{ GB/s}$)



Roofline Model: Application information

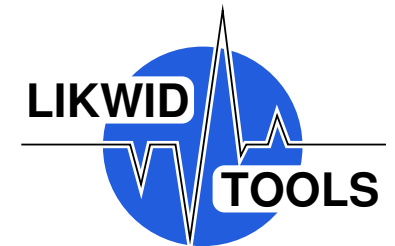
Measure application performance P and
calculate / measure application intensity I



Calculate I

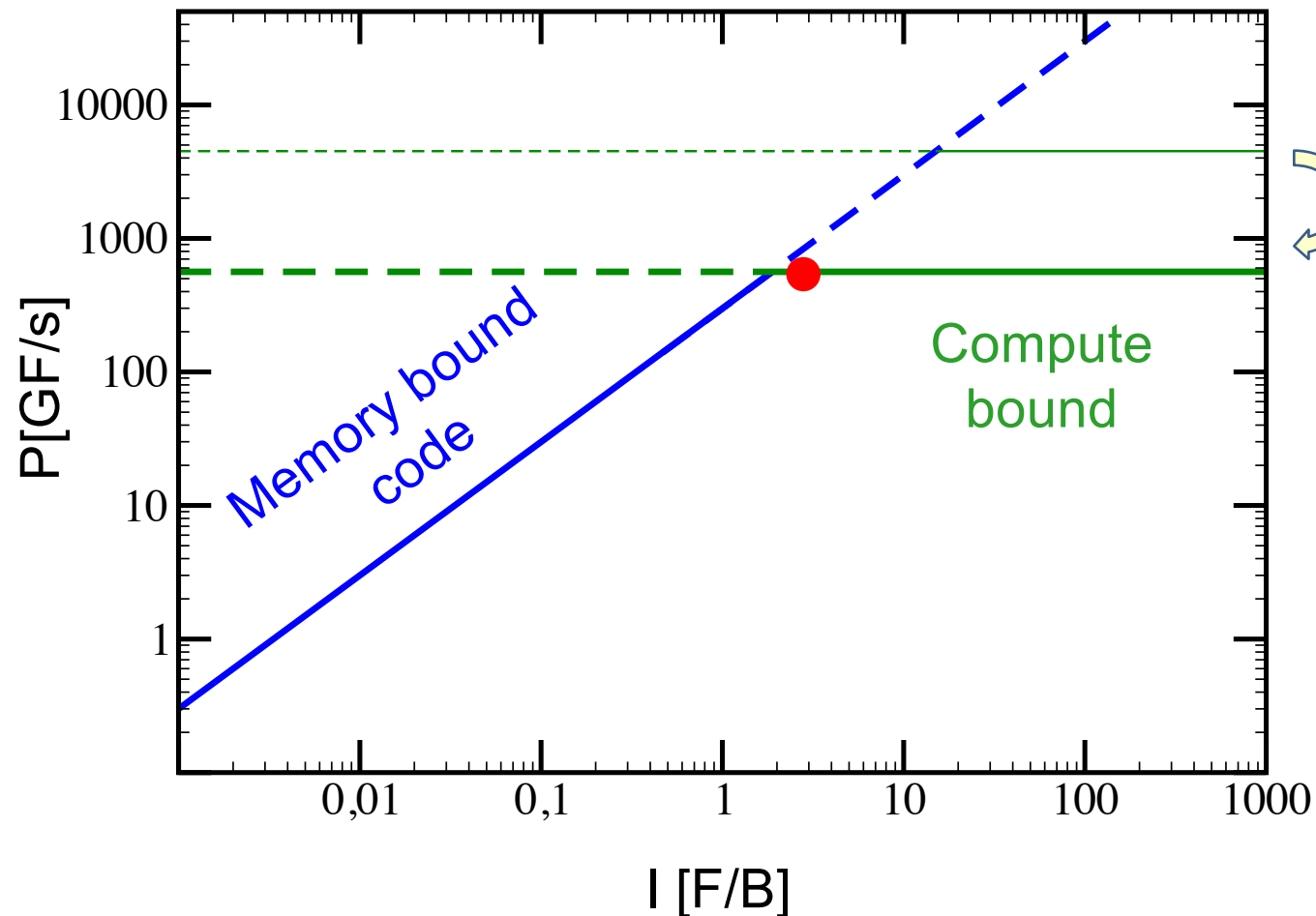
- Code / kernel analysis
- Data structure analysis

Measure I with



Lower horizontal roofs (P_{peak})

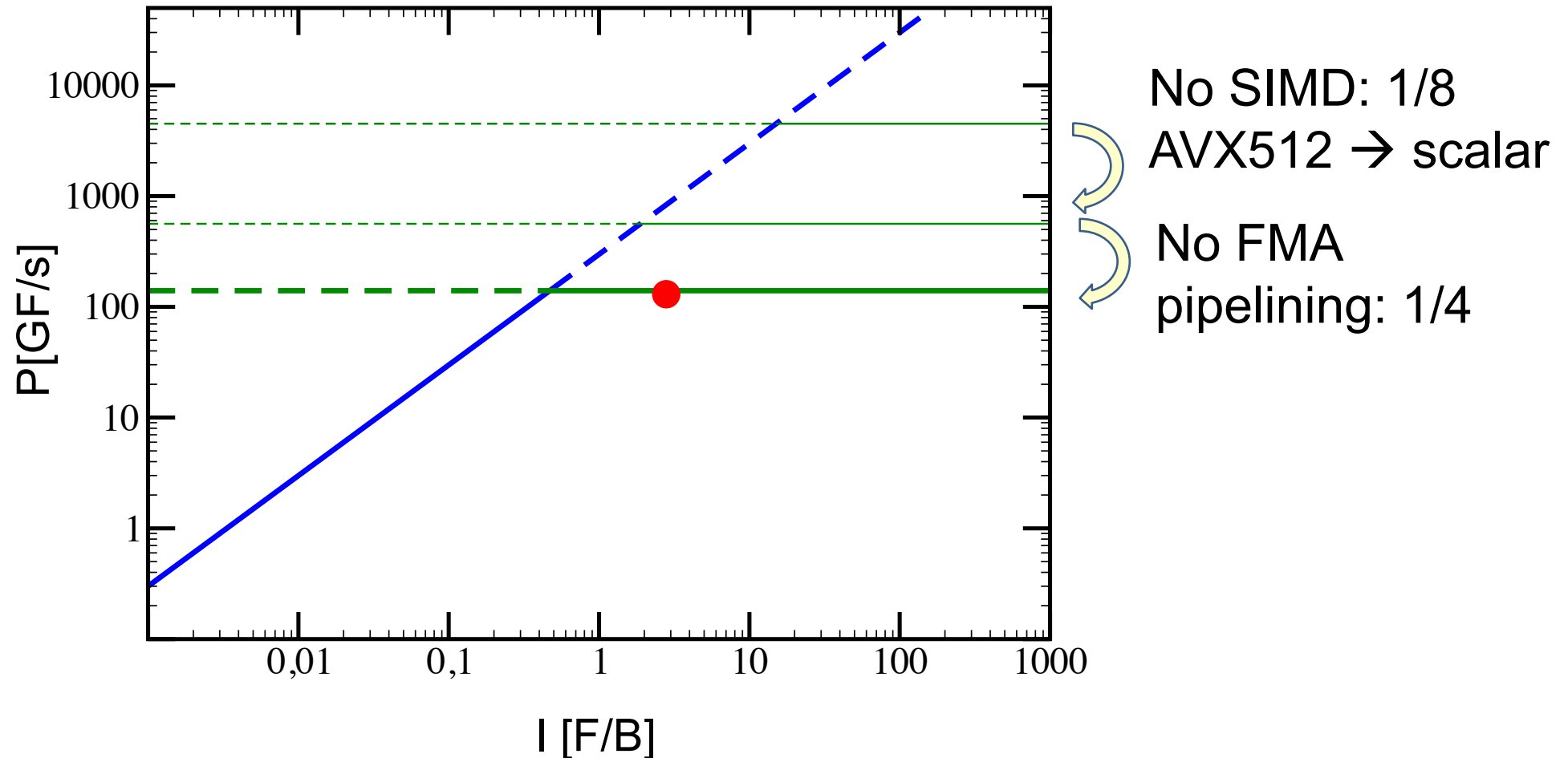
More realistic bounds for „bad“ implementations



No SIMD: 1/8
AVX512 → scalar

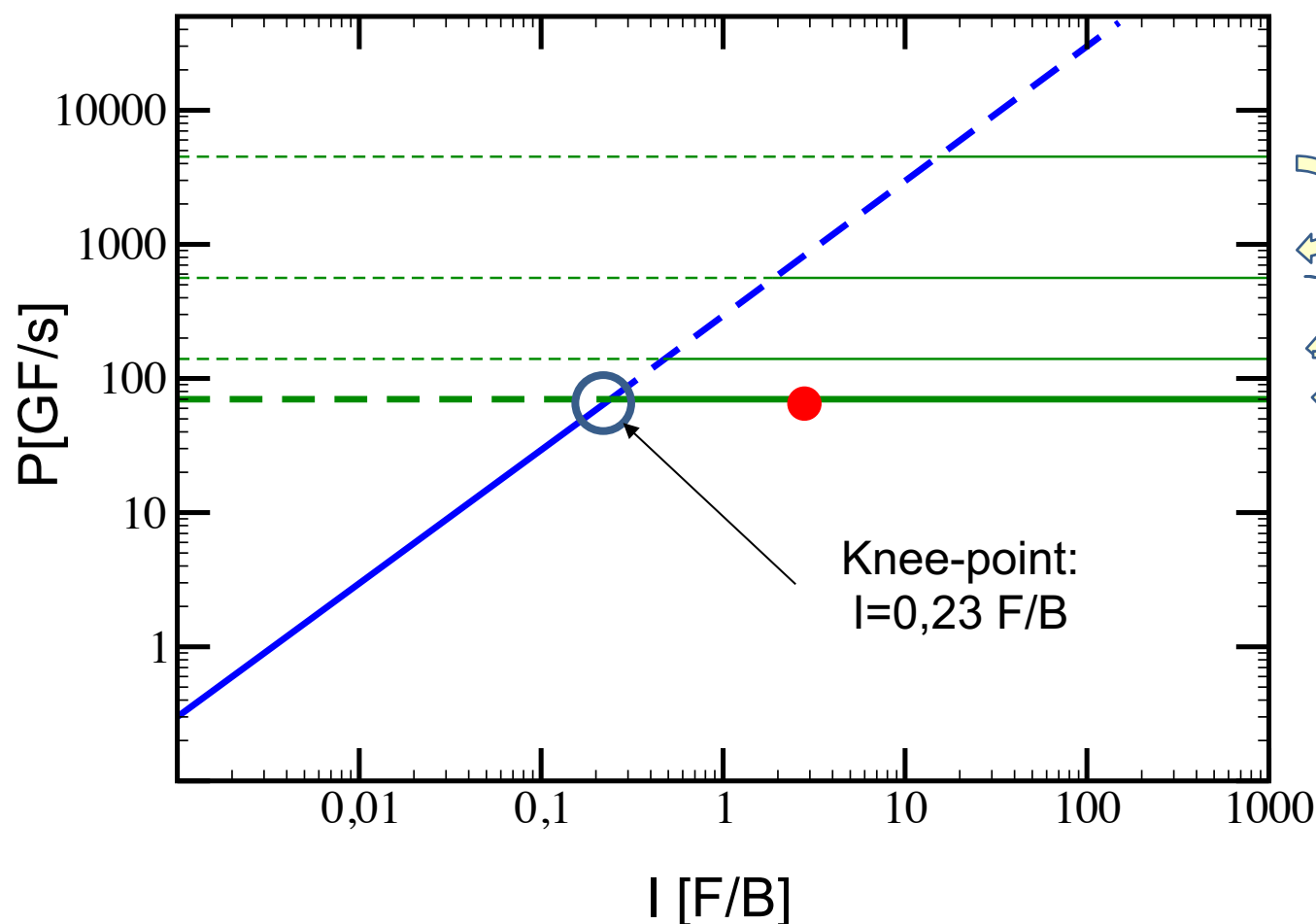
Lower horizontal roofs (P_{peak})

More realistic bounds for „bad“ implementations



Lower horizontal roofs (P_{peak})

No SIMD, no pipelining, 1 FMA only \rightarrow 64 x decrease in P_{Peak}



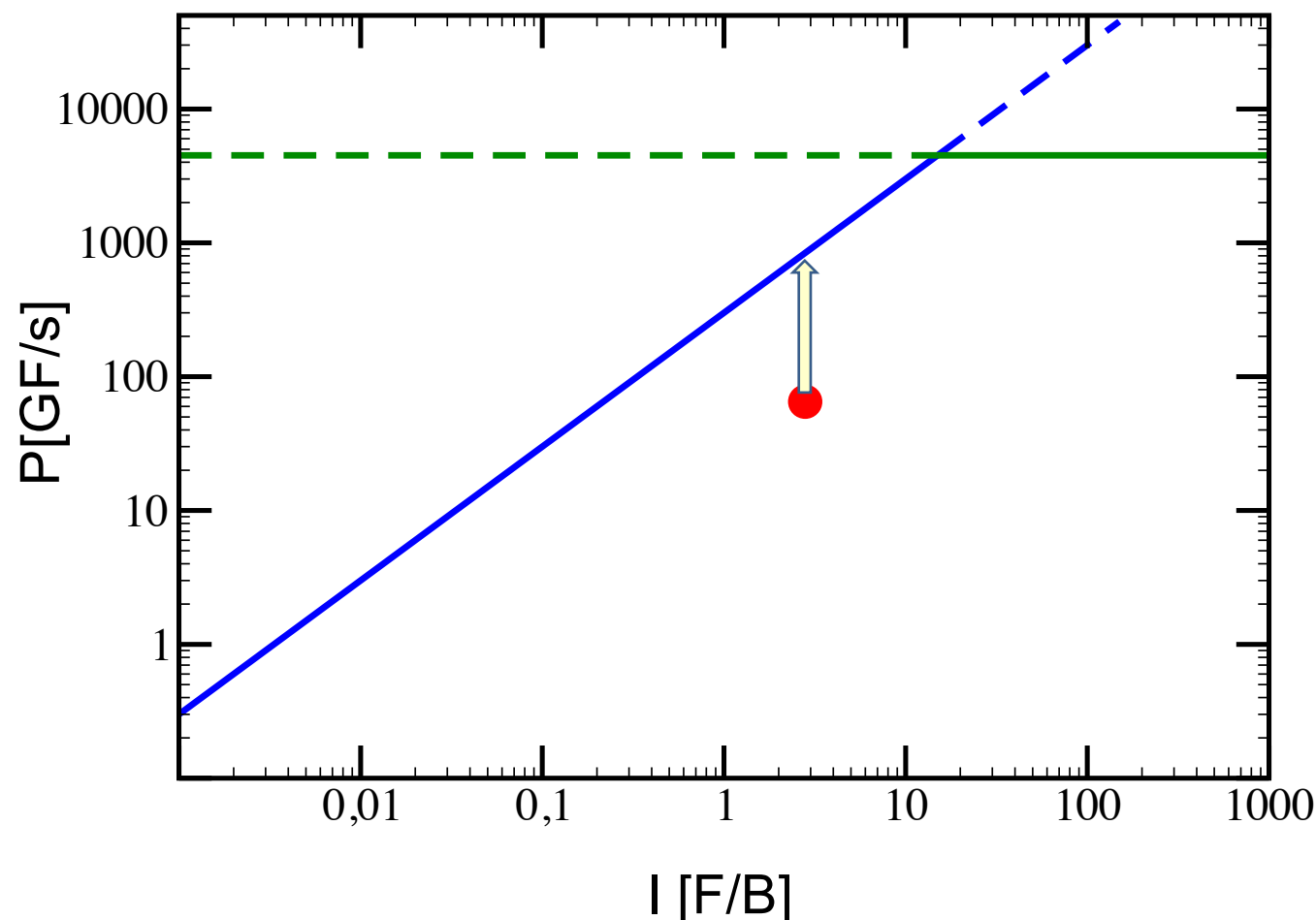
No SIMD: 1/8
AVX512 \rightarrow scalar

No FMA
pipelining: 1/4

Only 1 FMA: 1/2

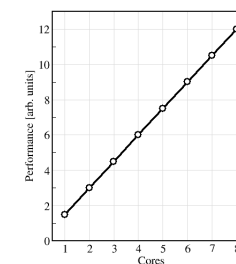
Lower horizontal roofs (P_{peak})

Reality: Lower horizontal roofs (P_{peak}) are typically not known



Indications:

- Linear scaling



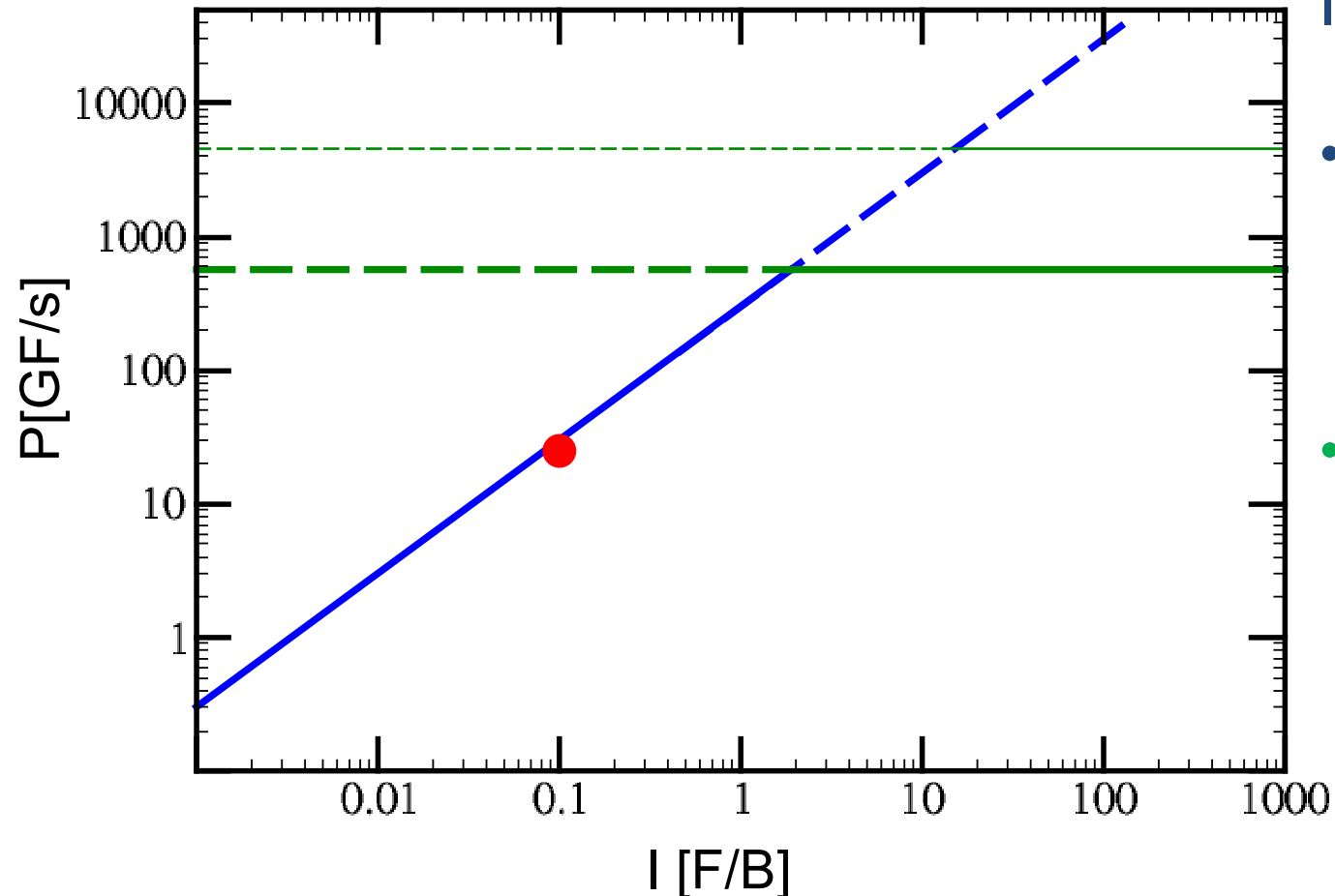
- **Low** Memory Bandwidth Utilization



Countermeasure: Improve code quality, e.g. compiler options, SIMD vectorization,...

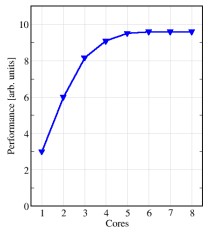
Bandwidth limited code

Well implemented code exploits memory bandwidth

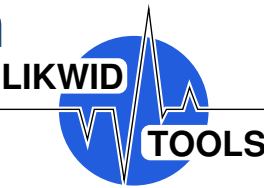


Indications:

- Saturation



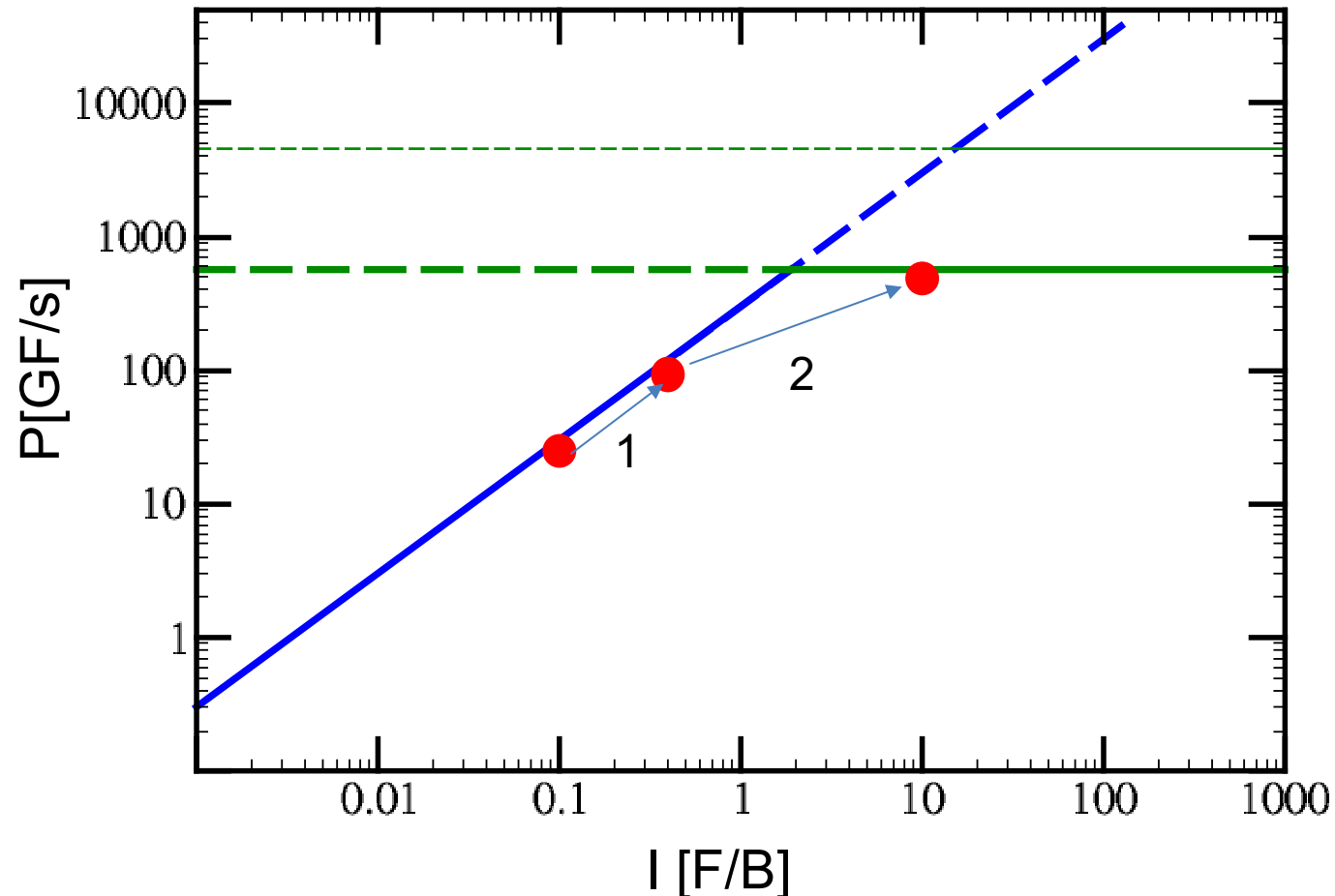
- **High** Memory Bandwidth Utilization



Countermeasure: Improve code Intensity by reducing data traffic

Bandwidth limited code

Well implemented code exploits memory bandwidth



Examples:

Move from

- BLAS2 DGEMV to
- BLAS3 DGEMM

Stencil codes

- Spatial (1) and
- Temporal blocking (2)

Roofline Model – Sparse Matrix Vector Multiplication

- SpMV: $y = A x$

Today: Performance engineering of a single SpMV – general structure

- Can we use RLM? What is the intensity of SpMV?
- Is there an maximum code intensity I for SpMV?
- Impact of matrix structure / OpenMP parallelization?
- CPU vs. GPU: Data layouts and more

Tomorrow: Can we increase I if we look beyond a single SpMV?

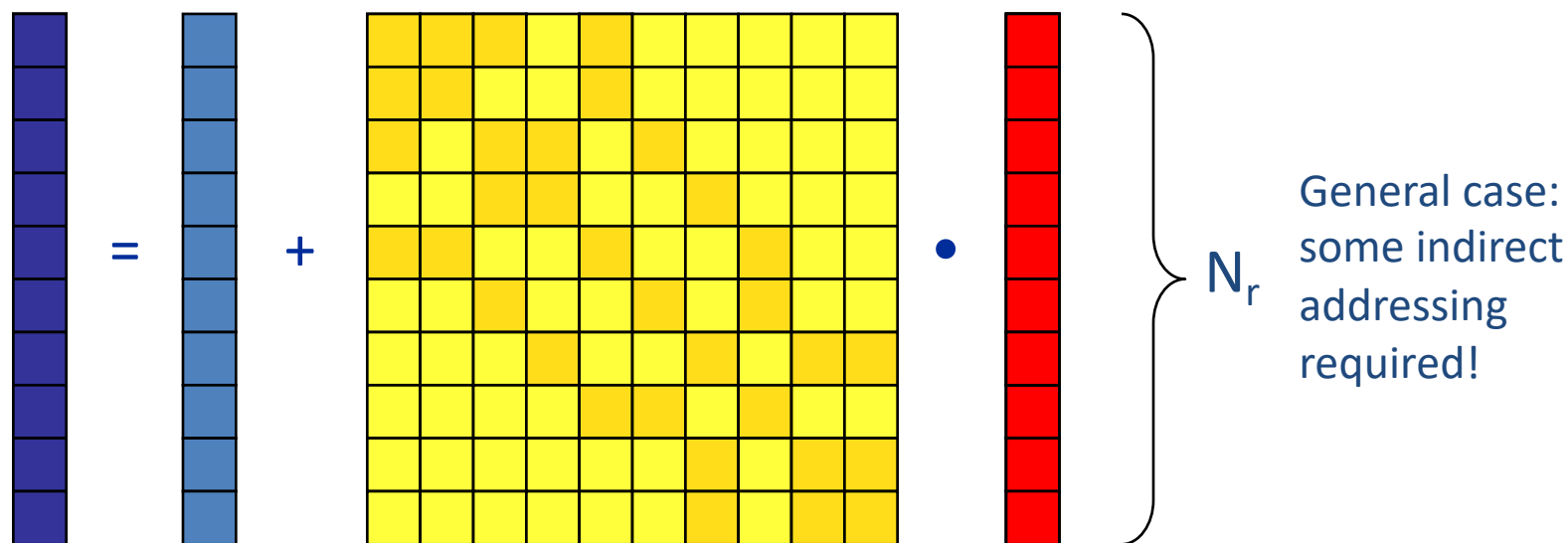
- RACE: Looking at SpMV differently: SpMVM \leftrightarrow Graph traversal problem

Performance Engineering for Sparse Matrix-Vector Multiplication



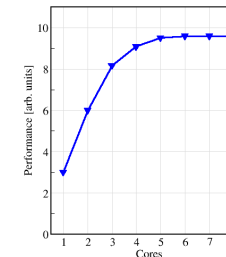
Sparse Matrix Vector Multiplication (SpMV)

- Key ingredient in some matrix diagonalization algorithms
 - Lanczos, Davidson, Jacobi-Davidson
- Store only N_{nz} nonzero elements of matrix and RHS, LHS vectors with N_r (number of matrix rows) entries
- “Sparse”: $N_{nz} \sim N_r$
- Average number of nonzeros per row: $N_{nzs} = N_{nz}/N_r$

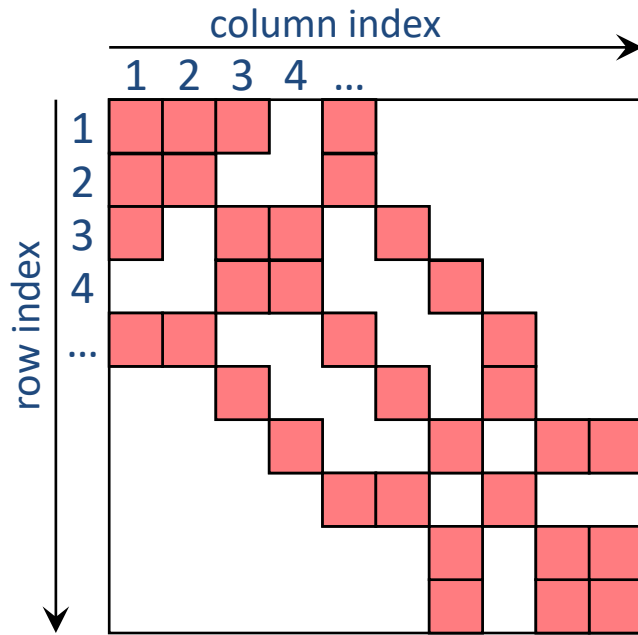


SpMVM characteristics

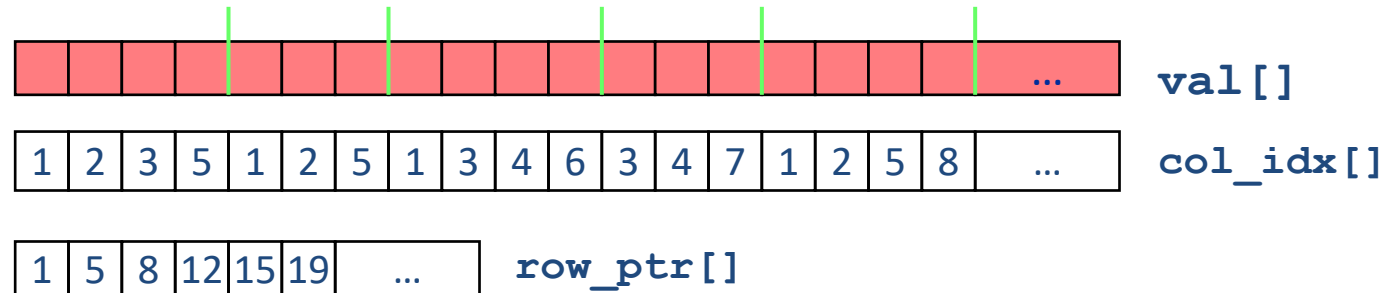
- For large problems, SpMV is inevitably **memory-bound**
 - **Intra-socket saturation effect** on modern multicores
- SpMV is **easily parallelizable** in shared and distributed memory
 - Load balancing
 - Communication overhead
- Data storage format is **crucial** for performance properties
 - Most useful general format on CPUs:
Compressed Row Storage (**CRS**)
 - Depending on compute architecture



CRS matrix storage scheme



- **val[]** stores all the nonzeros (length N_{nz})
- **col_idx[]** stores the column index of each nonzero (length N_{nz})
- **row_ptr[]** stores the starting index of each new row in **val[]** (length: N_r)



Case study: Sparse matrix-vector multiply

- Strongly memory-bound for large data sets
 - Mainly streaming data access mixed with partially indirect access:

```
!$OMP parallel do schedule(???)
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
!$OMP end parallel do
```

- Usually many spMV's required to solve a problem
- Now let's look at some performance measurements...

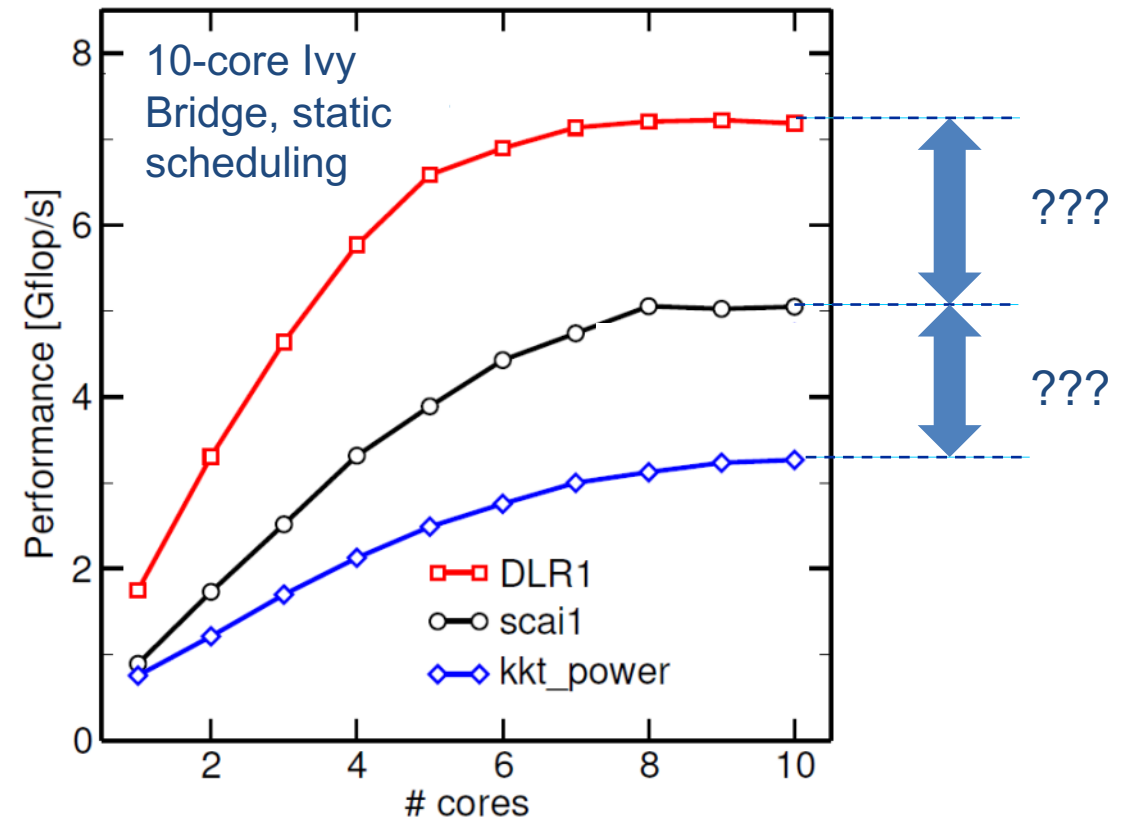
Performance characteristics

- Strongly memory-bound for large data sets → saturating performance across cores on the chip
- Performance seems to depend on the matrix

- Can we explain this?

- Is there a “light speed” for SpMV?

- Optimization?



SpMV node performance model – CRS (1)

```
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

```
real*8    val(Nnz)
integer*4  col_idx(Nnz)
integer*4  row_ptr(Nr)
real*8     C(Nr)
real*8     B(Nc)
```

Min. load traffic [B]: $(8 + 4) N_{nz} + (4 + 8) N_r + 8 N_c$

Min. store traffic [B]: $8 N_r$

Total FLOP count [F]: $2 N_{nz}$


$$B_{C,min} = \frac{12 N_{nz} + 20 N_r + 8 N_c}{2 N_{nz}} \frac{B}{F} = \frac{12 + 20/N_{nzc} + 8/N_{nzc}}{2} \frac{B}{F}$$

Nonzeros per row ($N_{nzc} = N_{nz}/N_r$) or column ($N_{nzc} = N_{nz}/N_c$)

$$\text{Lower bound for code balance: } B_{C,min} \geq 6 \frac{B}{F} \rightarrow I_{\max} \leq \frac{1}{6} \frac{F}{B}$$

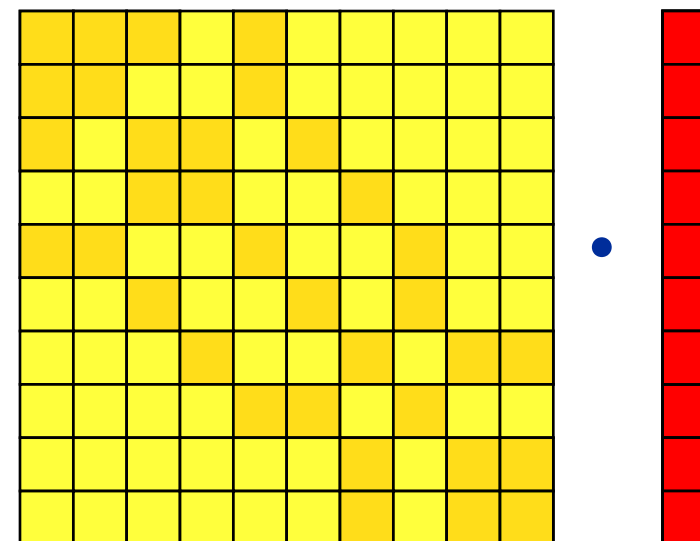
SpMV node performance model – CRS (2)

```
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1) - 1
    C(i) = C(i) + val(j) * B(col_idx(j))
  enddo
enddo
```

$$B_{C,min} = \frac{12 + 20/N_{nzc} + 8/N_{nzc}}{2} \frac{B}{F}$$
$$B_C(\alpha) = \frac{12 + 20/N_{nzc} + 8\alpha}{2} \frac{B}{F}$$


Consider square matrices: $N_{nzc} = N_{nzc}$ and $N_c = N_r$

Note: $B_C(1/N_{nzc}) = B_{C,min}$



Parameter (α) quantifies additional traffic for $B(:)$ (irregular access):

$$\alpha \geq 1/N_{nzc}$$

$$\alpha N_{nzc} \geq 1$$

The “ α effect”

CRS code balance

- α quantifies the traffic for loading the Right Hand Side (RHS) vector

- $\alpha = 0$ → RHS is in cache (RHS \ll cache size)
- $\alpha = 1/N_{nzs}$ → RHS loaded once
- $\alpha = 1$ → no cache
- $\alpha > 1$ → Houston, we have a problem!

$$B_C(\alpha) = \frac{12 + 20/N_{nzs} + 8\alpha}{2} \frac{B}{F}$$
$$= \left(6 + 4\alpha + \frac{10}{N_{nzs}}\right) \frac{B}{F}$$

Can we predict α ?

- Not in general
- Simple cases (banded, block-structured): Similar to layer condition analysis

→ Determine α by measuring the actual memory traffic (→ measured code balance B_C^{meas})

Determine α (RHS traffic quantification)

$$B_C(\alpha) = \left(6 + 4\alpha + \frac{10}{N_{nzs}}\right) \frac{B}{F} = \frac{V_{meas}}{N_{nz} \cdot 2 F} (= B_C^{meas})$$

- V_{meas} is the measured overall memory data traffic (using, e.g., likwid-perfctr)
- Solve for α :

$$\alpha = \frac{1}{4} \left(\frac{V_{meas}}{N_{nz} \cdot 2 \text{ bytes}} - 6 - \frac{10}{N_{nzs}} \right)$$

Example: kkt_power matrix from the UoF collection (one Intel SNB socket)

- $N_{nz} = 14.6 \cdot 10^6, N_{nzs} = 7.1$

- $V_{meas} \approx 258 \text{ MB}$

→ $\alpha = 0.36, \alpha N_{nzs} = 2.5$

→ RHS is loaded 2.5 times from memory

$$\frac{B_C(\alpha)}{B_{C,min}} = 1.11$$

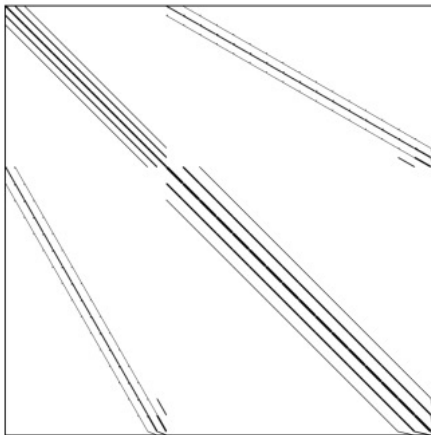
11% extra traffic →
optimization potential!

Three different sparse matrices

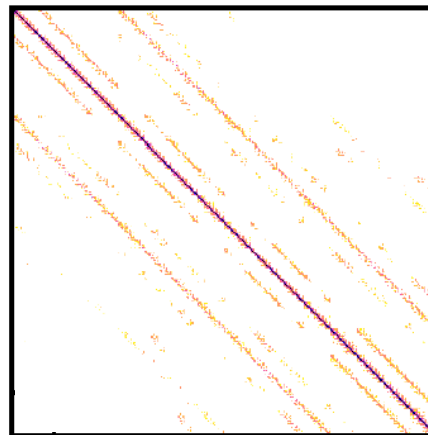
Roofline performance prediction : $P_{opt} = I * b_S = \frac{b_S}{B_{C,min}}$

Benchmark system: Intel Xeon Ivy Bridge E5-2660v2, 2.2 GHz, $b_S = 46.6 \text{ GB/s}$

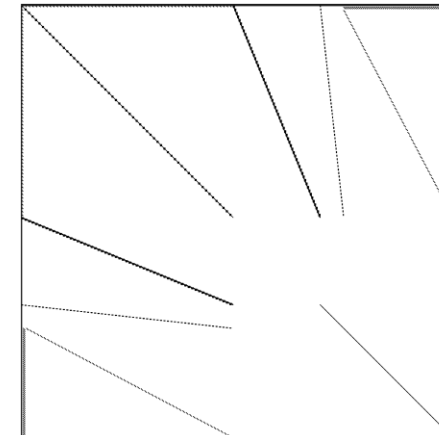
Matrix	N	N_{nzt}	$B_{C,min}$ [B/F]	P_{opt} [GF/s]
DLR1	278,502	143	6.1	7.64
scai1	3,405,035	7.0	8.0	5.83
kkt_power	2,063,494	7.08	8.0	5.83



DLR1

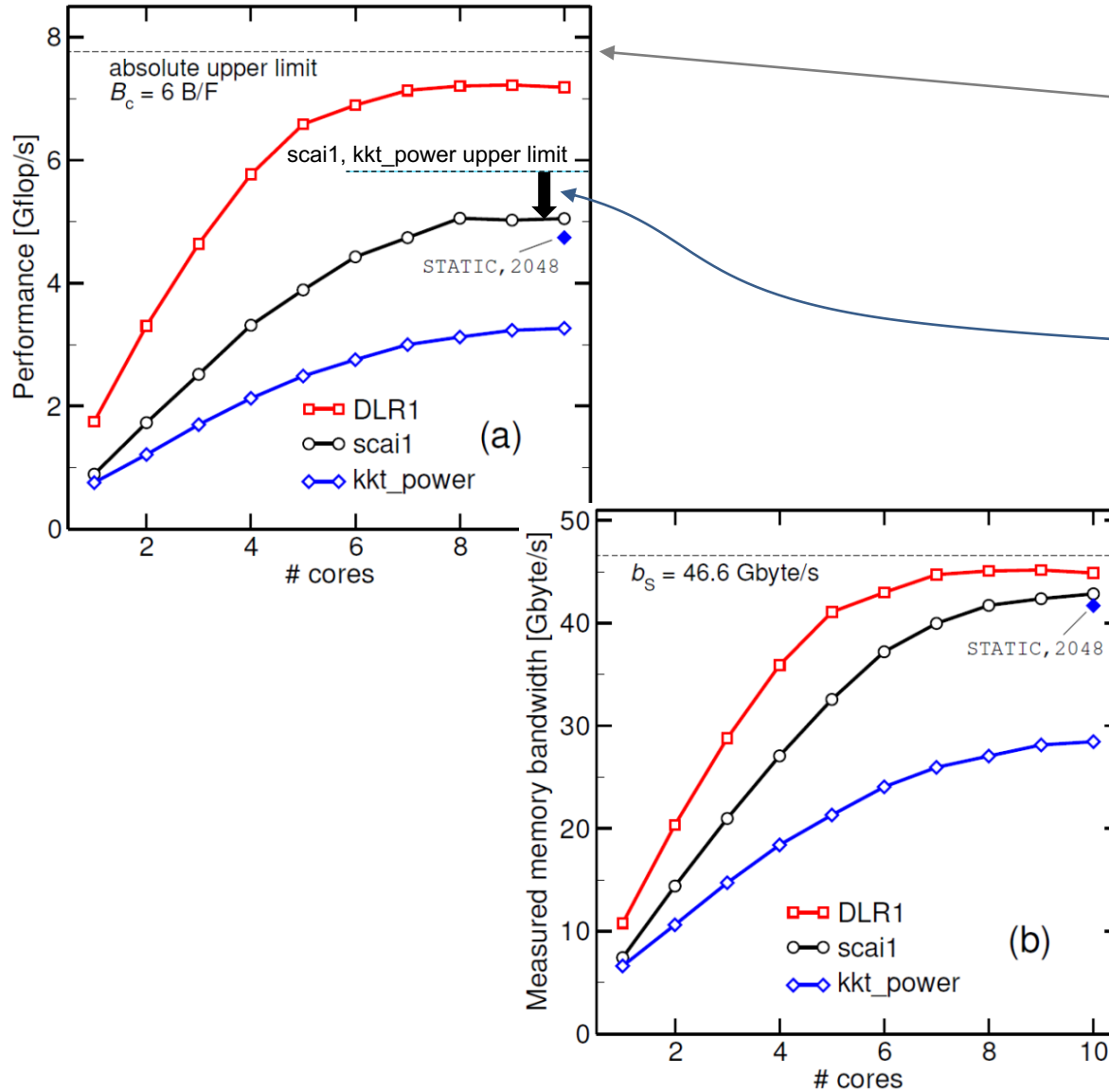


scai1



kkt_power

Now back to the start...



- $b_s = 46.6$ GB/s, $B_c = 6$ B/F
- Maximum spMVM performance:

$$P_{max} = 7.8 \text{ GF/s}$$

- **DLR1** causes (almost) minimum CRS code balance (as expected)

- **scai1** measured balance:

$$B_c^{meas} \approx 8.5 \text{ B/F} > B_{c,min} \text{ (6\% higher than min)}$$

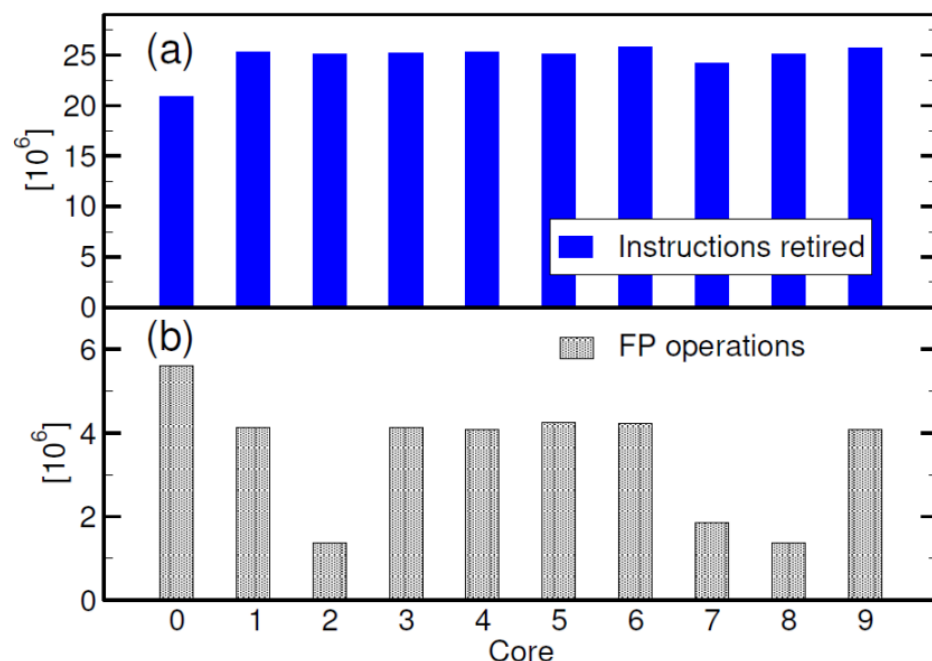
→ good BW utilization, slightly non-optimal α

- **kkt_power** measured balance:

$$B_c^{meas} \approx 8.8 \text{ B/F} > B_{c,min} \text{ (10\% higher than min)}$$

→ performance degraded by load imbalance, fix by block-cyclic schedule

Investigating the load imbalance with kkt_power

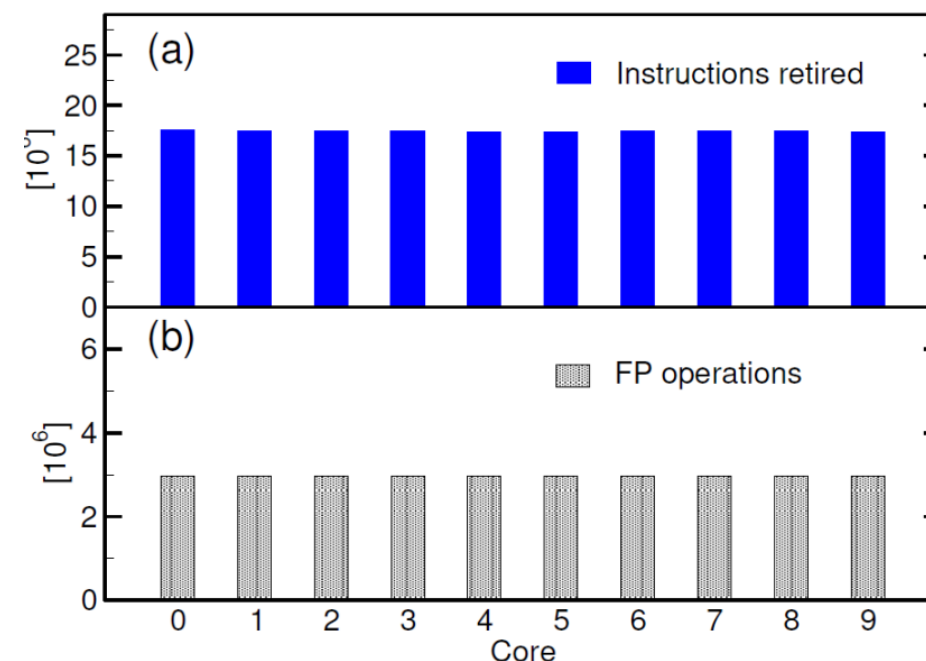


Measurements with likwid-perfctr
(MEM_DP group)

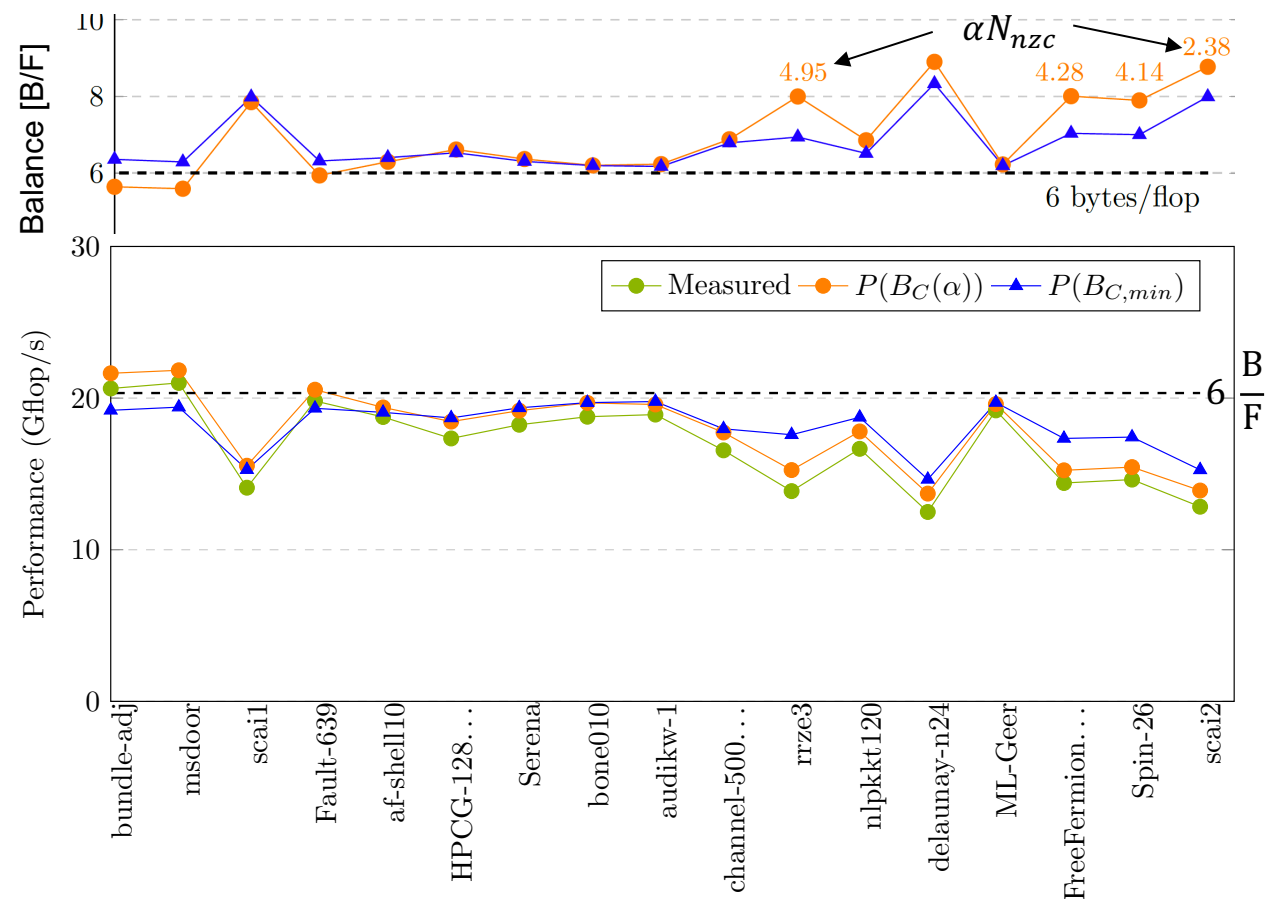
static

static, 2048

→ Fewer overall instructions, (almost)
BW saturation, 50% better
performance with load balancing
→ CPI value unchanged!



SpMV node performance model – CPU



Intel Xeon Platinum 9242
24c@2.8GHz (turbo)
 $b_s = 122 \text{ GB/s}$

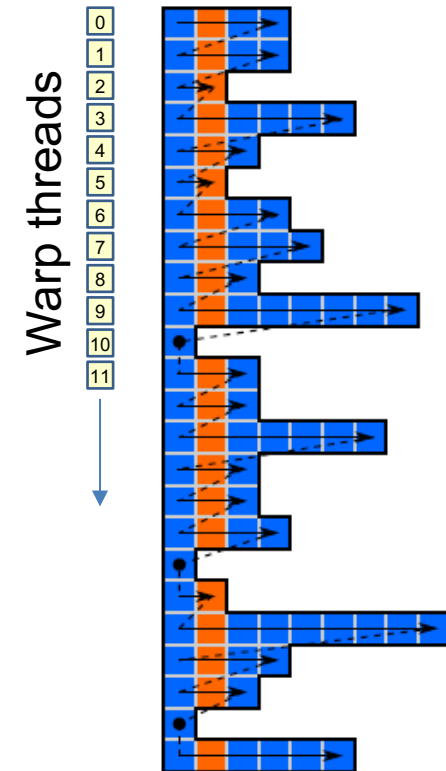
Matrices taken from: C. L. Alappat et al.: *ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX*. In print.
Preprint: [arXiv:2103.0301](https://arxiv.org/abs/2103.0301)

Data layout considerations – GPUs

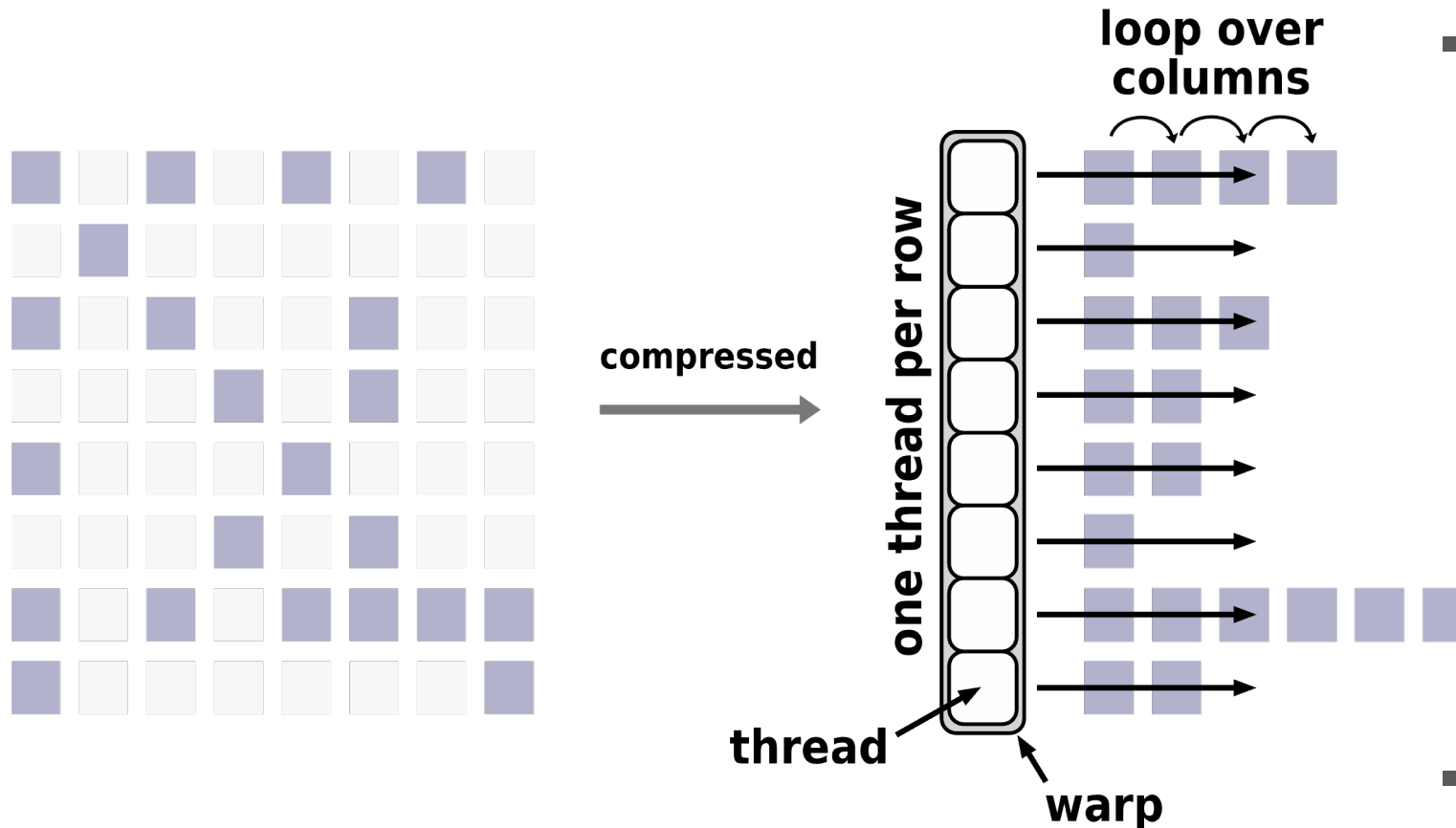


What about GPUs?

- GPUs need
 - Sufficient work per kernel launch in order to leverage their parallelism
 - Coalesced access to memory (consecutive threads in a warp should access consecutive memory addresses)
- Plain CRS for SpMV on GPUs is not a good idea
 1. Short inner loop
 2. Different amount of work per thread
 3. Non-coalesced memory access
- Remedy: Use SIMD/SIMT-friendly storage format
 - ELLPACK, SELL-C- σ , DIA, ESB,...



What about GPUs?



- Each GPU thread computes one row, iterates over column indices
- This is the best mapping for CRS:
 - Enough parallelism to saturate the GPU (unless matrix is small)
 - Consecutive threads use similar data, spatial locality is used
 - No reduction among threads, each thread computes its own sum
- But plain CRS has problems on GPUs!

CRS SpMV in CUDA ($y = Ax$)

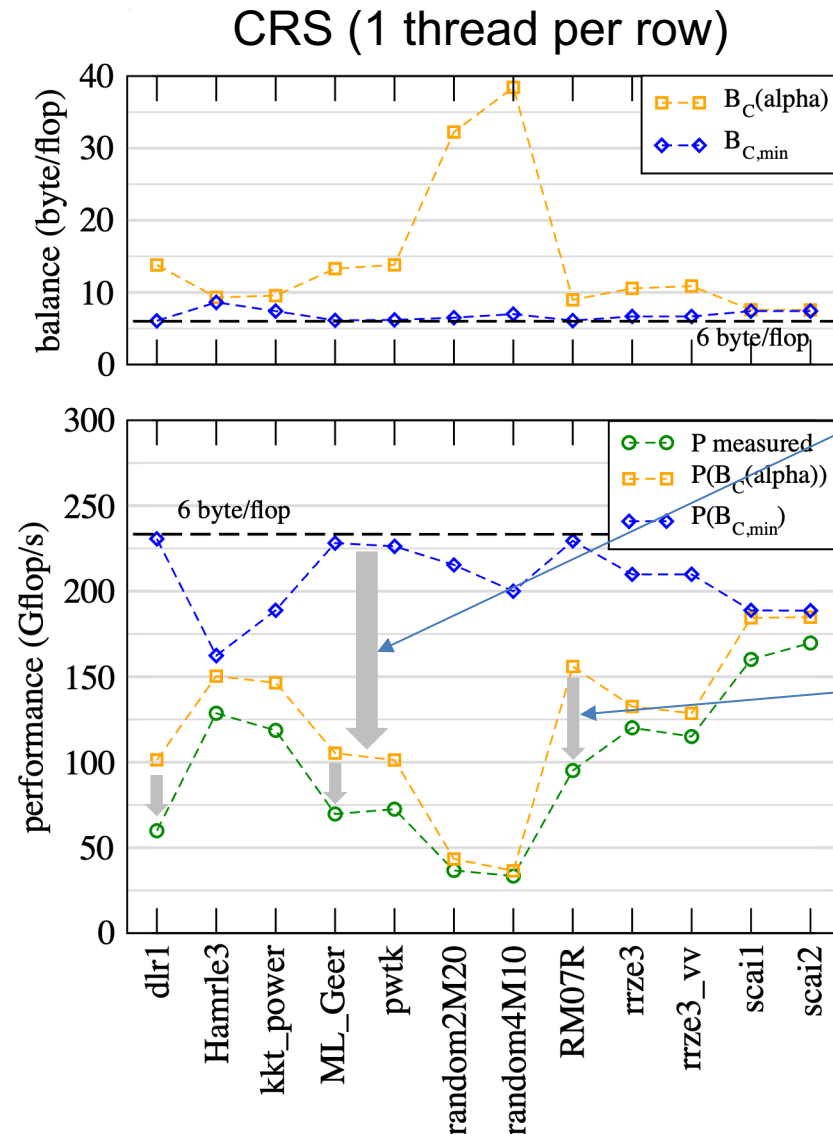
```
template <typename VT, typename IT>
__global__ static void
spmv_csr(const ST num_rows,
         const IT * RESTRICT row_ptrs, const IT * RESTRICT col_idx,
         const VT * RESTRICT values,   const VT * RESTRICT x,
                                         VT * RESTRICT y)
{
    ST row = threadIdx.x + blockDim.x * blockIdx.x; // 1 thread per row

    if (row < num_rows) {
        VT sum{};
        for (IT j = row_ptrs[row]; j < row_ptrs[row + 1]; ++j) {
            sum += values[j] * x[col_idx[j]];
        }
        y[row] = sum;
    }
}
```

$$B_c(\alpha) = \left(6 + 4\alpha + \frac{6}{N_{nzs}}\right) \frac{B}{F}$$

No write-allocate on GPUs for consecutive stores

SpMV CRS performance on a GPU

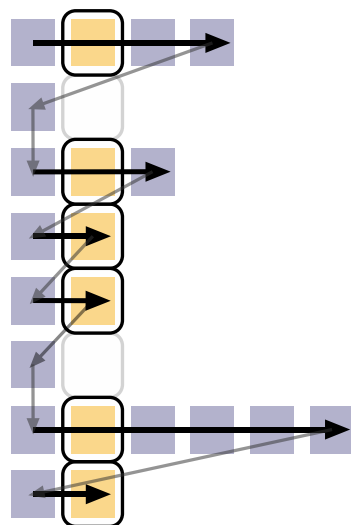


NVIDIA Ampere A100

Memory bandwidth $b_S = 1400$ GB/s

- Strong “ α effect” – large deviation from optimal α for many matrices
 - Many cache lines touched b/c every thread handles one row \rightarrow bad cache usage
- Mediocre memory bandwidth usage ($\ll 1400$ GB/s) in many cases
 - Non-coalesced memory access
 - Imbalance across rows/threads of warps

CRS SpMV on GPUs: scattered loads



in linear
memory

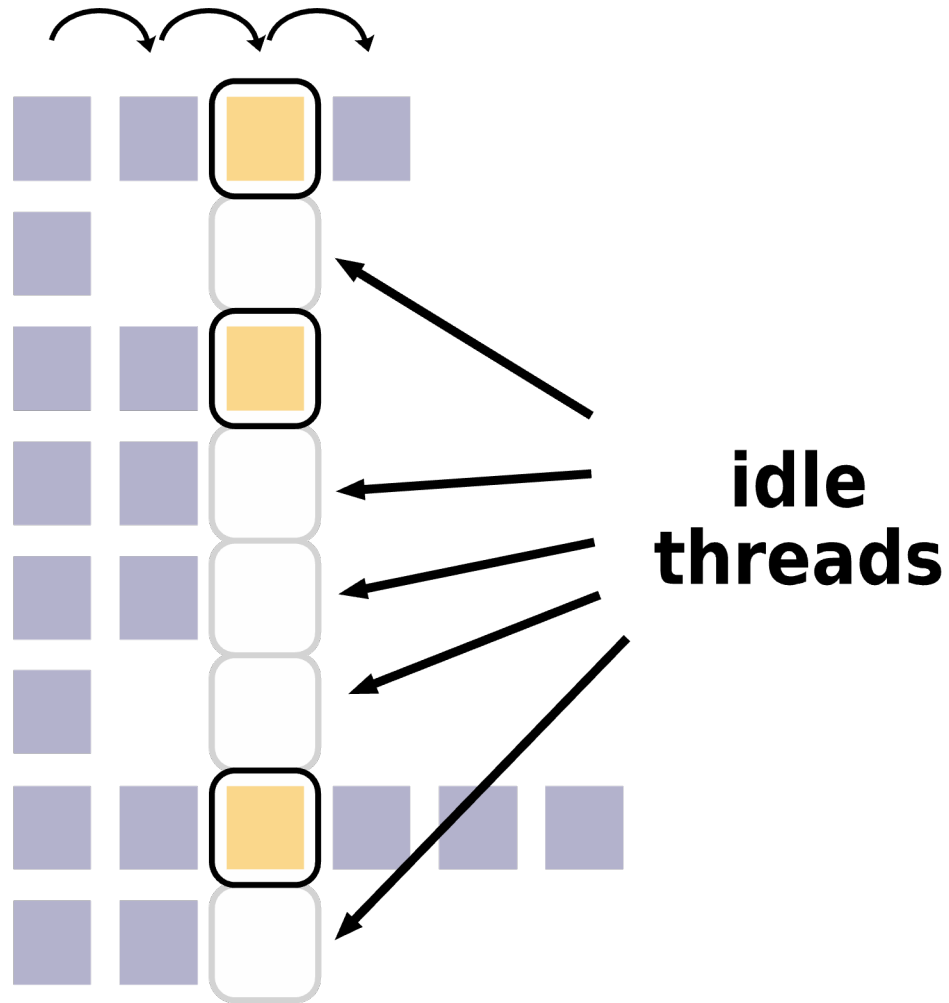
- Loads are executed in **lock step** on GPUs too
- GPUs prefer compact “**coalescable**” addresses for each load (i.e. consecutive access across threads)

CRS vs. GPU

- Row-wise storage format but access pattern orthogonal! → Scattered loads within warp
- Scattered loads need **more cycles**
- Scattered values occupy **more cache lines**
- **Higher latencies and redundant data transfers**



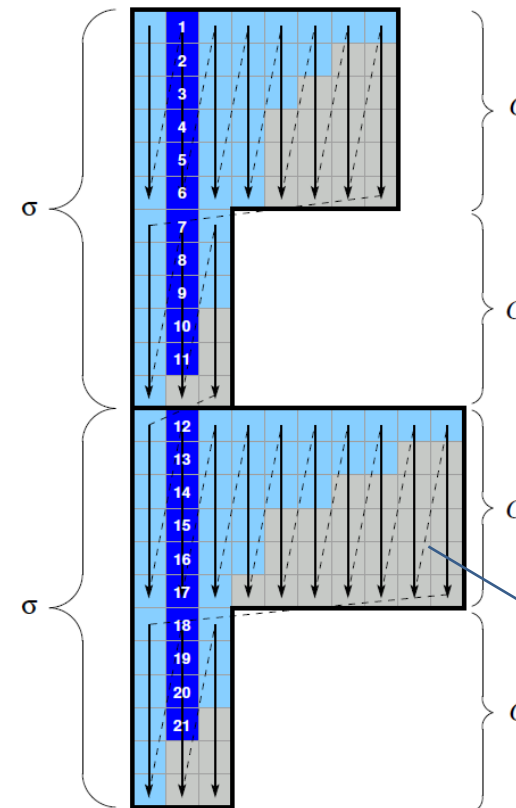
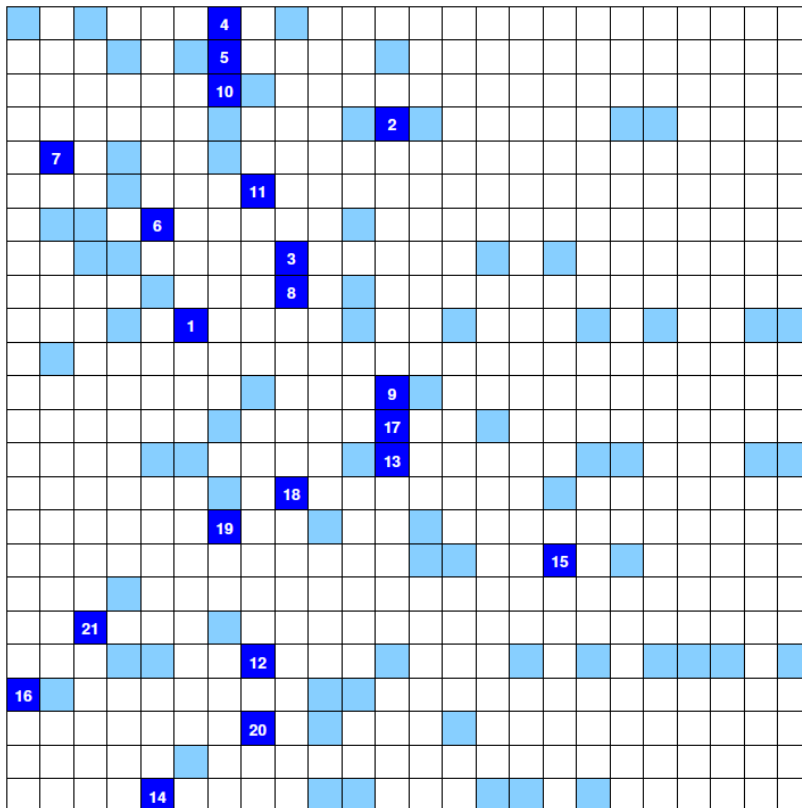
CRS SPMV on GPUs – Problems: Idle threads



- Threads are grouped in **warps**
- Threads in a warp execute in **lockstep**, similar to SIMD
- Problem: loop over column indices can have **different trip count** for each vector
- Threads in a warp that have completed the loop are **masked off**
- All threads in a warp have to **wait** for the thread with most non-zeros

Idea

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



“Chunk occupancy”:

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

l_i : width of chunk i

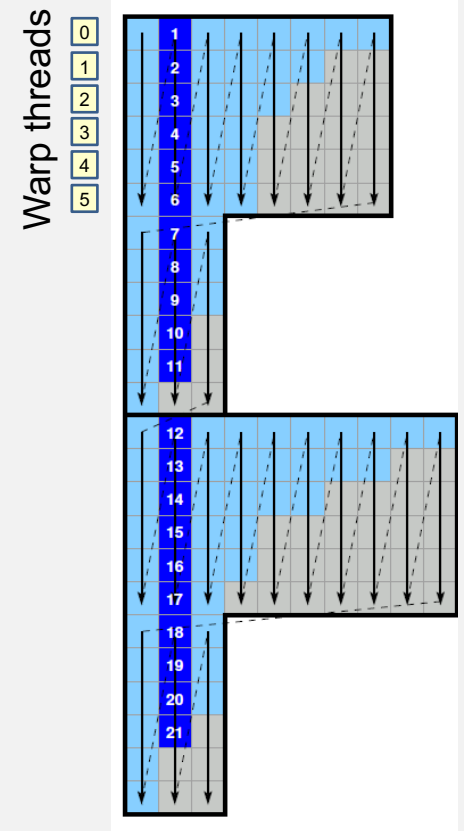
zero padding

SELL-C- σ SpMV in CUDA ($y=Ax$)

```
template <typename VT, typename IT> __global__ static void
spmv_scs(const ST C, const ST n_chunks,      const IT * RESTRICT chunk_ptrs,
         const IT * RESTRICT chunk_lengths, const IT * RESTRICT col_idxs,
         const VT * RESTRICT values, const VT * RESTRICT x, VT * RESTRICT y)
{
    ST row = threadIdx.x + blockDim.x * blockIdx.x;
    ST c    = row / C;    // the no. of the chunk
    ST idx   = row % C;    // index inside the chunk

    if (row < n_chunks * C) {
        VT tmp{};
        IT cs = chunk_ptrs[c]; // points to start indices of chunks

        for (ST j = 0; j < chunk_lengths[c]; ++j) {
            tmp += values[cs + idx] * x[col_idxs[cs + idx]];
            cs += C;
        }
        y[row] = tmp;
    }
}
```



Code balance of SELL-C- σ ($y=Ax$)

Matrix data & column index

LHS update (write only)

chunk index

$$B_{SELL}(\alpha, \beta, N_{nzs}) = \left(\frac{1}{\beta} \left(\frac{8 + 4}{2} \right) + \frac{8\alpha + \beta(8 + 4/C)/N_{nzs}}{2} \right) \frac{\text{bytes}}{\text{flop}}$$
$$= \left(\frac{6}{\beta} + 4\alpha + \frac{\beta(4 + 2/C)}{N_{nzs}} \right) \frac{\text{bytes}}{\text{flop}}$$

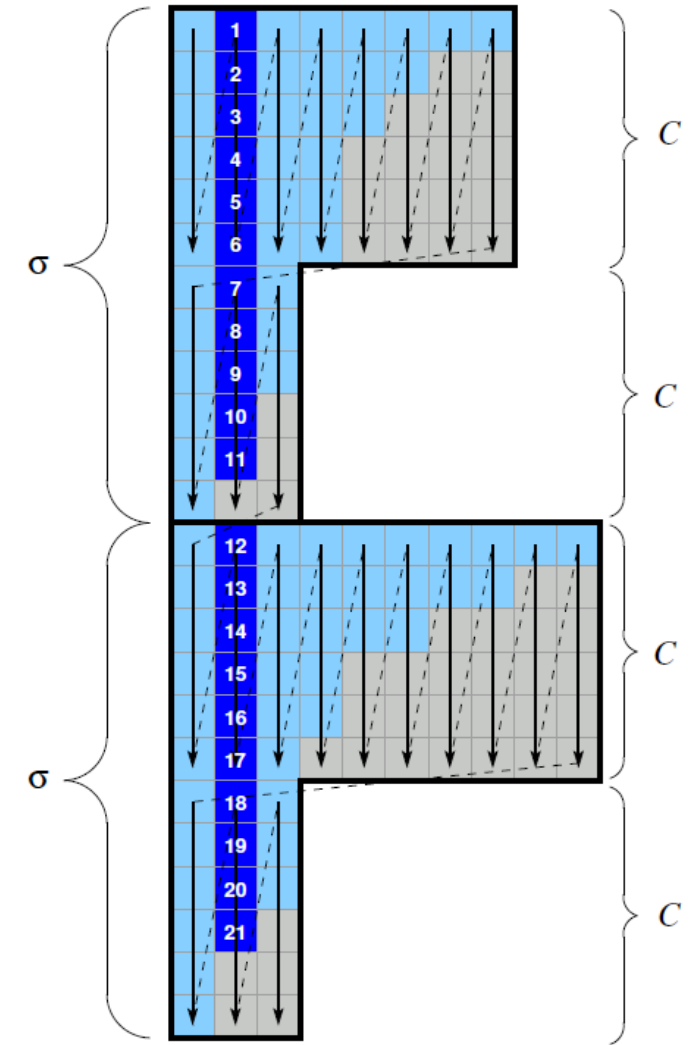
Optimal $\alpha = \frac{\beta}{N_{nzs}}$

When measuring B_C^{meas} , take care to use the “**useful**”
number of flops (excluding zero padding) for work

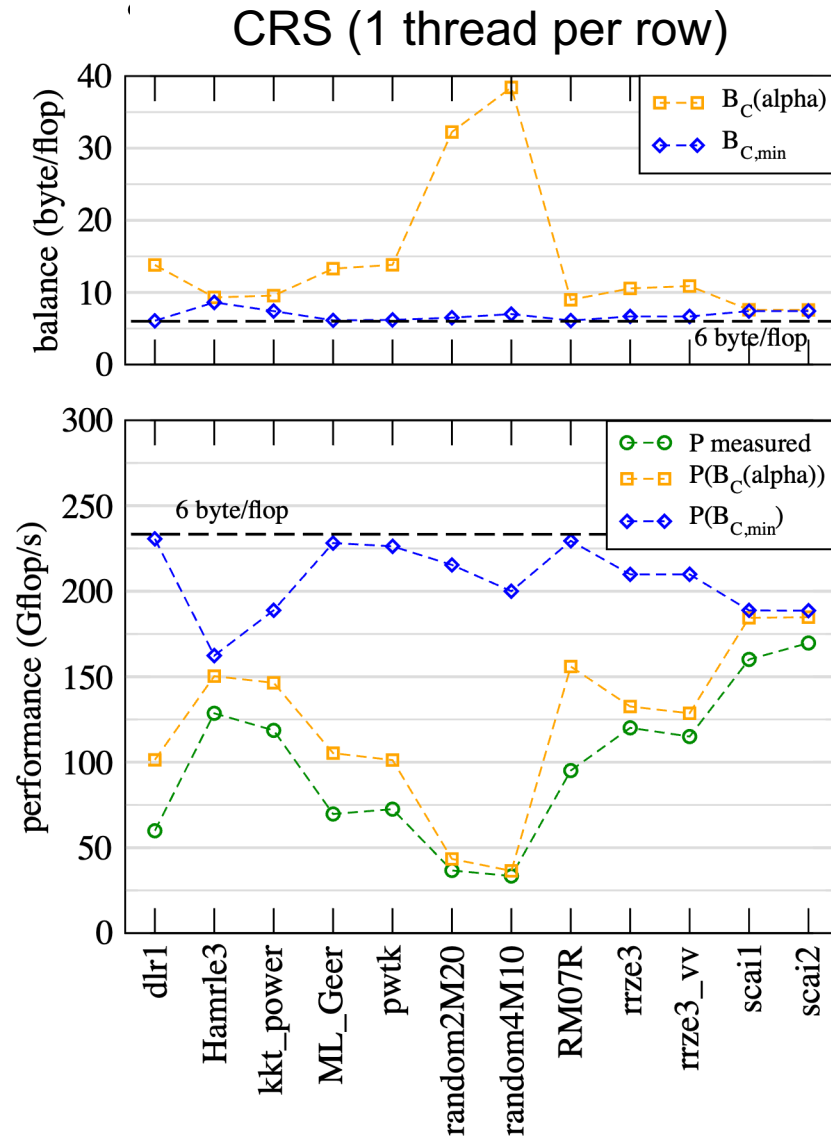


How to choose the parameters C and σ on GPUs?

- C
 - $n \times$ warp size to allow good utilization of GPU threads and cache lines
- σ
 - As **small as possible**, as large as necessary
 - Large σ **reduces zero padding** (brings β closer to 1)
 - Sorting alters RHS access pattern \rightarrow α **depends on σ**

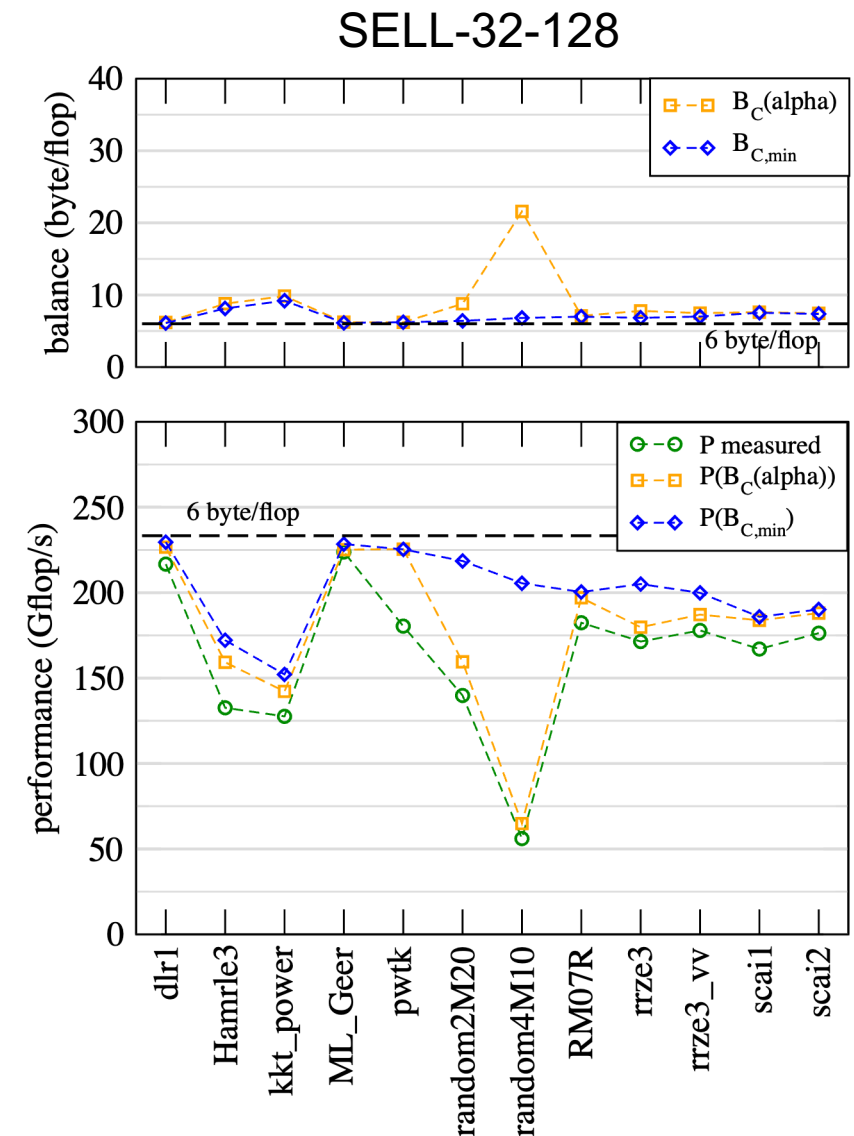


SpMV node performance model – GPU



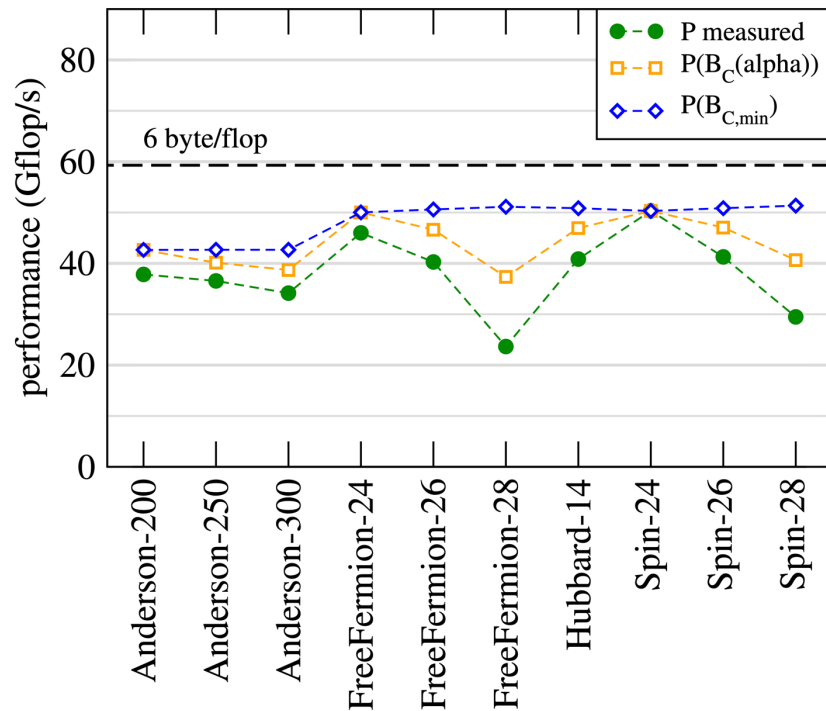
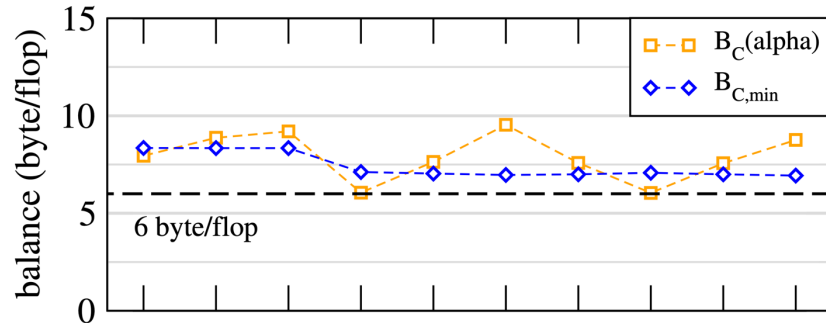
NVIDIA Ampere A100

$$b_S = 1400 \text{ GB/s}$$



SpMV node performance model – CPU

CRS

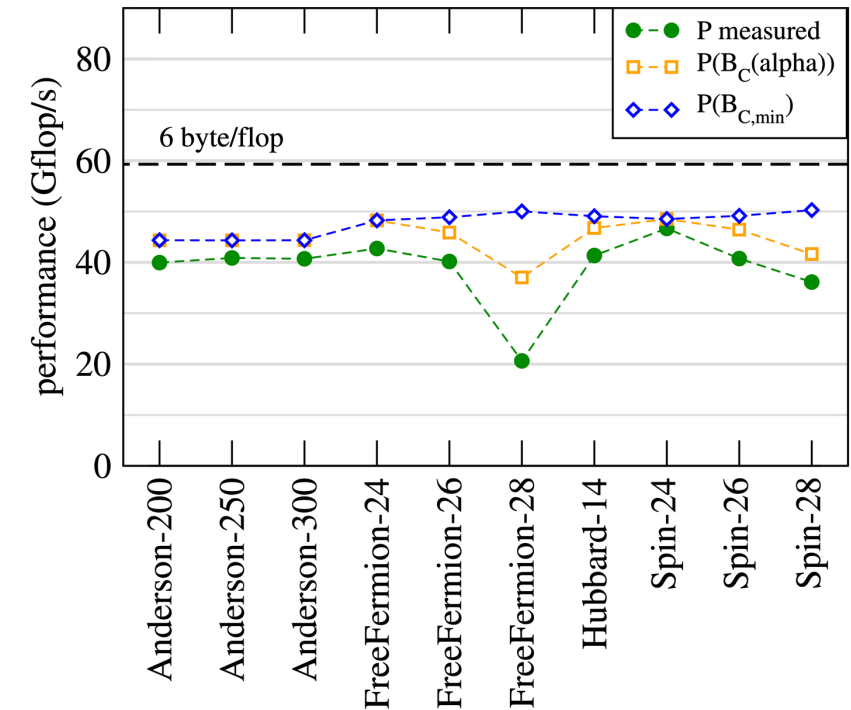
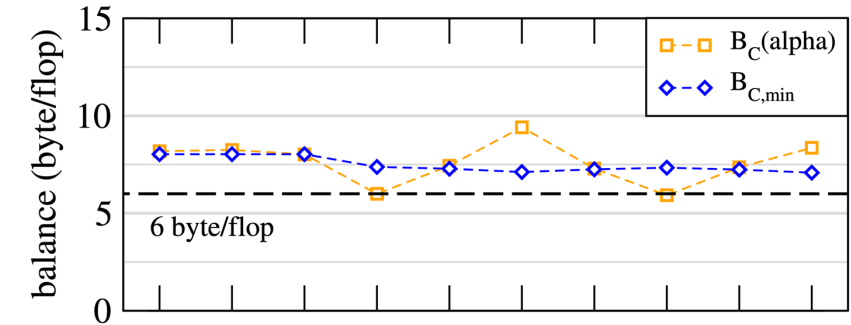


Intel Ice Lake 8360 Y

$$b_S = 356 \text{ GB/s}$$

Different matrices!!!

SELL-32-128



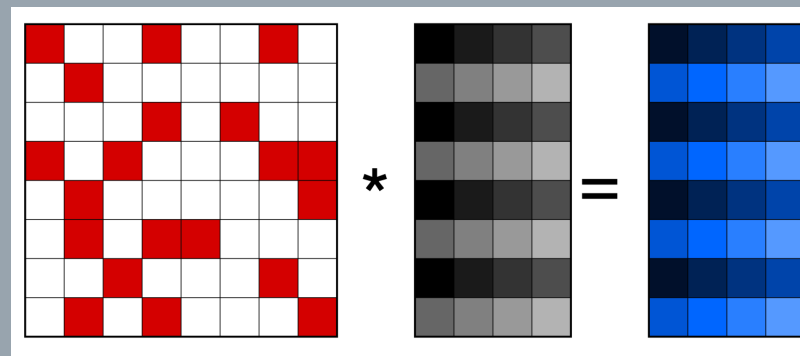
Roofline analysis for spMVM

- Conclusion from the Roofline analysis
 - The roofline model does not “work” for spMVM due to the RHS traffic uncertainties
 - We have “turned the model around” and measured the actual memory traffic to determine the RHS overhead
 - Result indicates:
 1. how much actual traffic the RHS generates
 2. how efficient the RHS access is (compare BW with max. BW)
 3. how much optimization potential we have with matrix reordering
- Do not forget about load balancing!
- Sparse matrix times multiple vectors bears the potential of huge savings in data volume
- Consequence: Modeling is not always 100% predictive. It's all about *learning more about performance properties!*

BACKUP



Applying sparse matrix to multiple vectors (Sparse Matrix Multiple Vectors: SpMMV)



Multiple RHS vectors (SpMMV)

Unchanged matrix applied to multiple RHS (\mathbf{r}) vectors to yield multiple LHS (\mathbf{r}) vectors

```
do s = 1, r
  do i = 1, Nr
    do j = row_ptr(i), row_ptr(i+1)-1
      C(i, s) = C(i, s) + val(j) *
                B(col_idx(j), s)
    enddo
  enddo
enddo
```

B_c unchanged, no
reuse of matrix data

```
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1)-1
    do s = 1, r
      C(i, s) = C(i, s) + val(j) *
                B(col_idx(j), s)
    enddo
  enddo
enddo
```

Higher B_c due to max
reuse of matrix data

```
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1)-1
    do s = 1, r
      C(s, i) = C(s, i) + val(j) *
                B(s, col_idx(j))
    enddo
  enddo
enddo
```

CL-friendly data
structure (row major)

SpMMV code balance

One complete inner (s) loop traversal:

- $2r$ flops
- 12 bytes from matrix data (value + index)
- $\frac{16r}{N_{nzs}}$ bytes from the r LHS updates
- $\frac{4}{N_{nzs}}$ bytes from the row pointer
- $8r\alpha(r)$ bytes from the r RHS reads

$$B_c(r) = \frac{1}{2r} \left(12 + 8r\alpha(r) + \frac{16r + 4}{N_{nzs}} \right) \frac{B}{F}$$
$$= \left(\frac{6}{r} + 4\alpha(r) + \frac{8 + 2/r}{N_{nzs}} \right) \frac{B}{F}$$

OK so what now???

```
do i = 1, Nr
  do j = row_ptr(i), row_ptr(i+1)-1
    do s = 1, r
      C(s,i) = C(s,i) + val(j) *
                B(s,col_idx(j))
    enddo
  enddo
enddo
```

SpMMV code balance

Let's check some limits to see if this makes sense!

$$B_c(r) = \left(\frac{6}{r} + 4\alpha(r) + \frac{8 + 2/r}{N_{nzs}} \right) \frac{B}{F}$$

Limit 1: $r = 1$

$$\xrightarrow{r=1} \underbrace{\left(6 + 4\alpha + \frac{10}{N_{nzs}} \right) \frac{B}{F}}_{\text{reassuring 😊}}$$

Limit 2: $N_{nzs} \gg 1$

$$\xrightarrow{N_{nzs} \gg 1} \frac{6}{r} \frac{B}{F}$$

Limit 3: $r \gg 1$

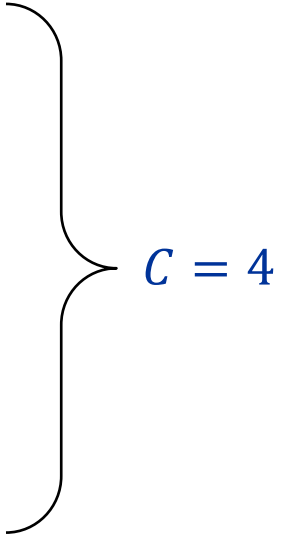
$$\xrightarrow{r \gg 1} \underbrace{\left(4\alpha(r) + \frac{8}{N_{nzs}} \right) \frac{B}{F}}_{\text{Can become very small for large } N_{nzs} \rightarrow \text{decoupling from memory bandwidth is possible!}}$$

M. Kreutzer et al.: *Performance Engineering of the Kernel Polynomial Method on Large-Scale CPU-GPU Systems*.
Proc. IPDPS15, DOI: [10.1109/IPDPS.2015.76](https://doi.org/10.1109/IPDPS.2015.76)

SELL-C- σ kernel on CPUs

Example $C = 4$ without further unrolling

```
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]];
    }
}
```



$C = 4$