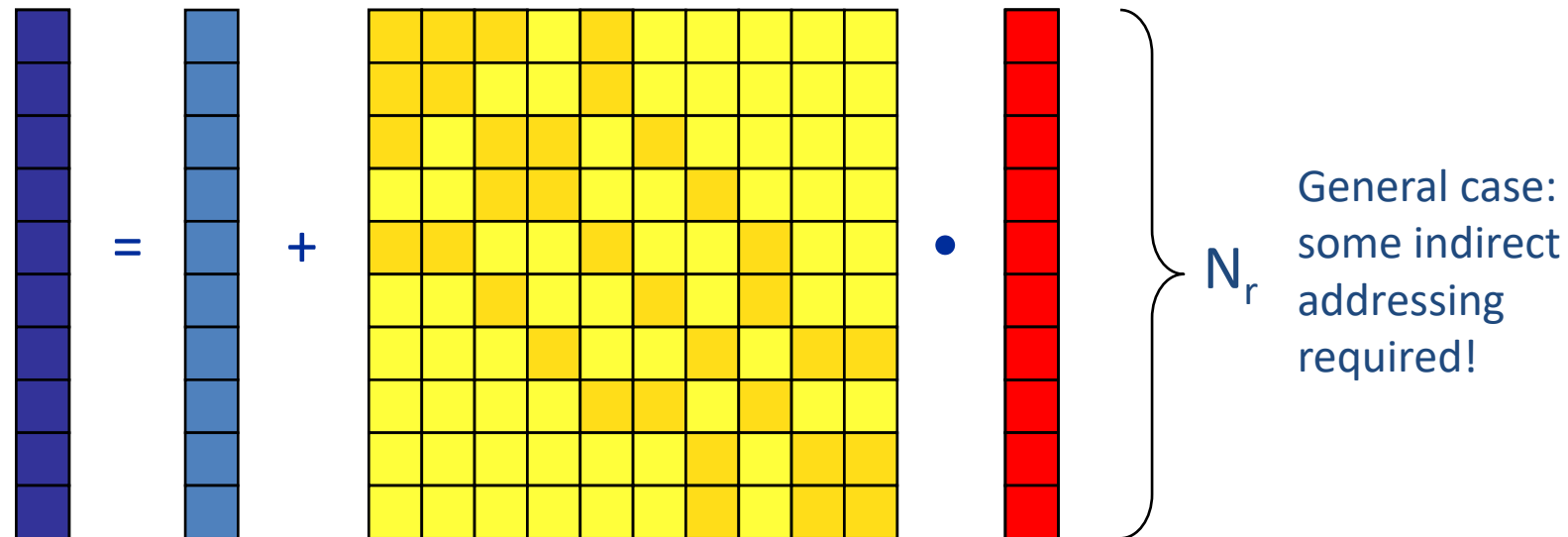


Case study: Sparse Matrix-Vector Multiplication



Sparse Matrix Vector Multiplication (SpMV)

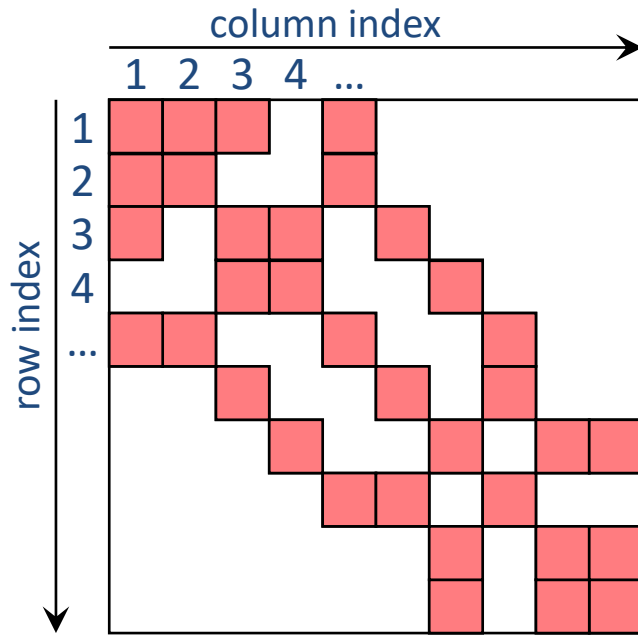
- Key ingredient in numerous sparse linear algebra solvers
- Store only N_{nz} nonzero elements of matrix and RHS, LHS vectors with N_r (number of matrix rows) entries
- “Sparse”: $N_{nz} \sim N_r$
- Average number of nonzeros per row: $N_{nzs} = N_{nz}/N_r$



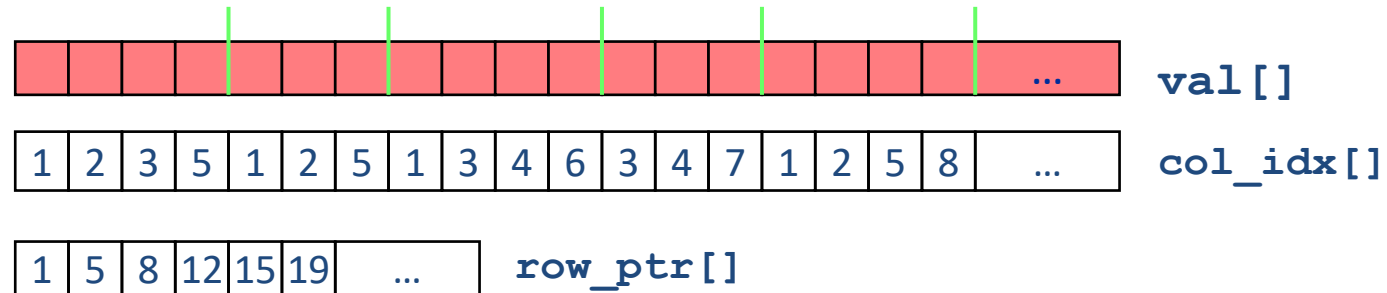
SpMVM characteristics

- 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

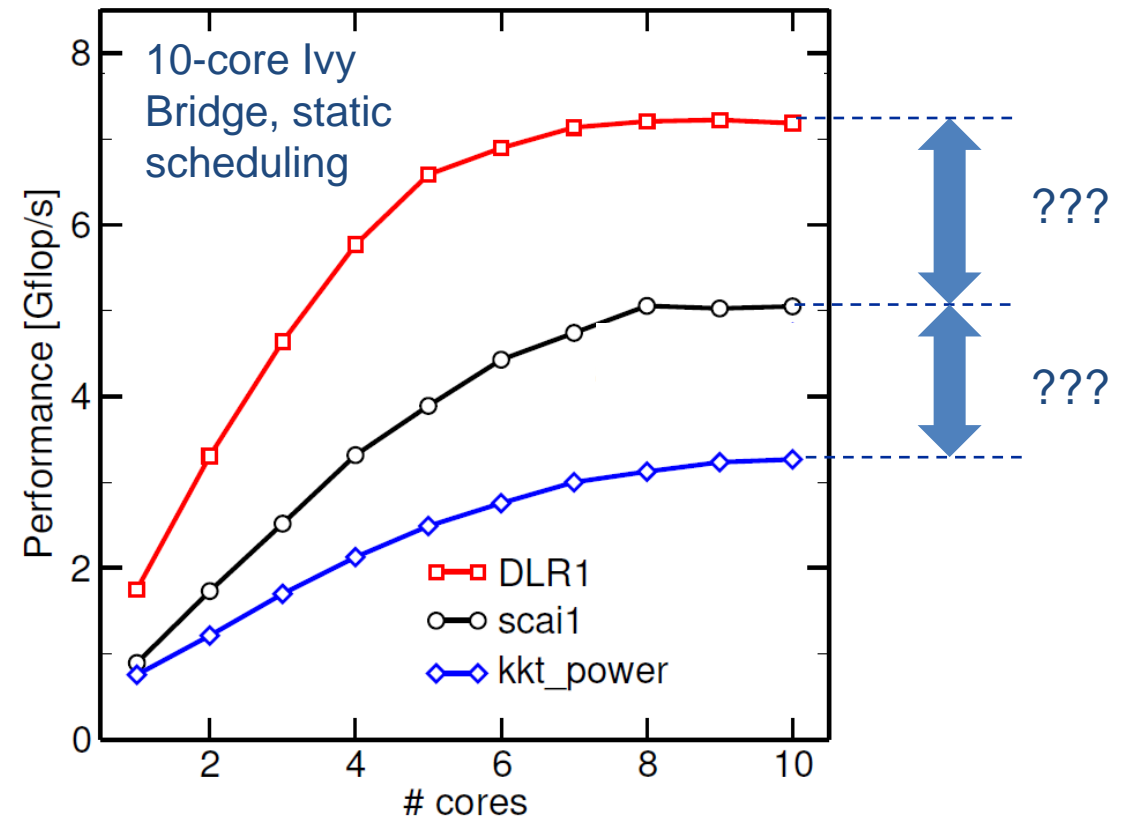
- Strongly memory-bound for large data sets
 - Streaming, with partially indirect access:

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

- Usually many spMVMs 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 ($N_{nzc} = N_{nz}/N_c$)

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

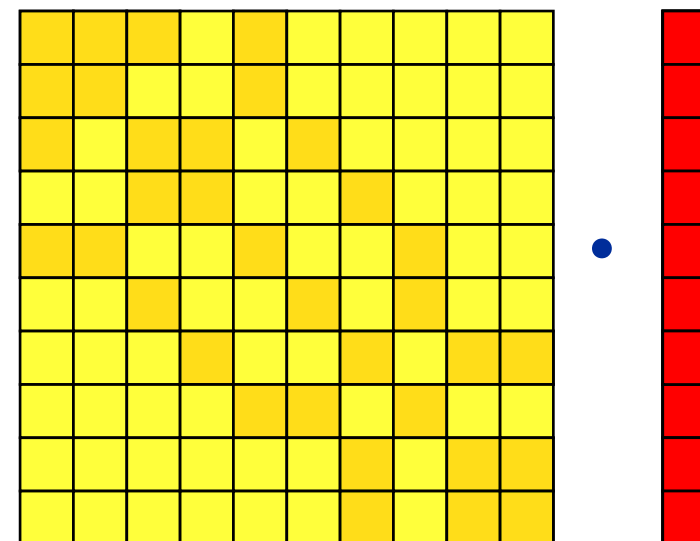
SpMV node performance model – CRS (2)

```
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 $\mathbf{B}(:)$ (irregular access):

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

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

The “ α effect”

DP CRS code balance

- α quantifies the traffic for loading the RHS
 - $\alpha = 0 \rightarrow$ RHS is in cache
 - $\alpha = 1/N_{nzs} \rightarrow$ RHS loaded once
 - $\alpha = 1 \rightarrow$ no cache
 - $\alpha > 1 \rightarrow$ Houston, we have a problem!
- “Target” performance = b_s/B_c
- **Caveat:** Maximum memory BW may not be achieved with spMVM (see later)

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

Can we predict α ?

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

\rightarrow Determine α by measuring the actual memory traffic (\rightarrow 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
on 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

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

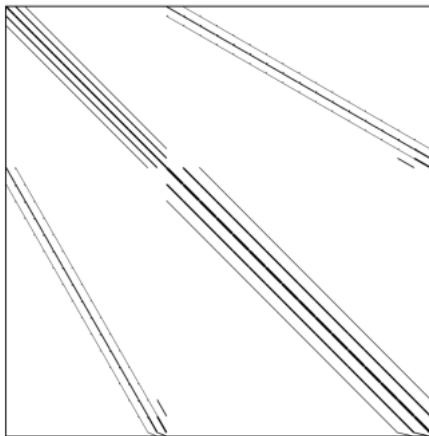
11% extra traffic →
optimization potential!

Three different sparse matrices

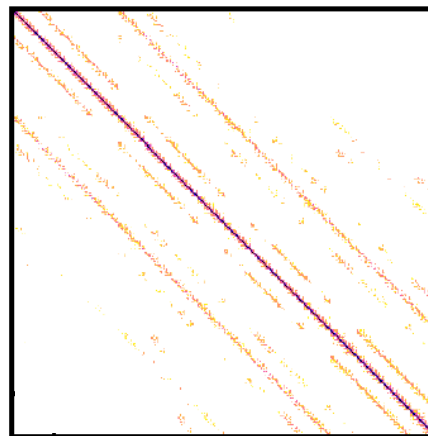
Benchmark system: Intel Xeon Ivy Bridge E5-2660v2, 2.2 GHz, $b_s = 46.6$ GB/s

$$\rightarrow \text{Roofline: } P_{opt} = b_s / B_{C,min}$$

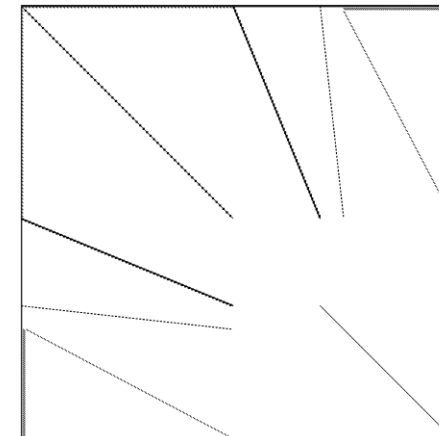
Matrix	N	N_{nzs}	$B_{C,min}$ [B/F]	P_{opt} [GF/s]
DLR1	278,502	143	6.1	7.64
scai1	3,405,035	7.0	8.0	5.83
kkt_power	2,063,494	7.08	8.0	5.83



DLR1

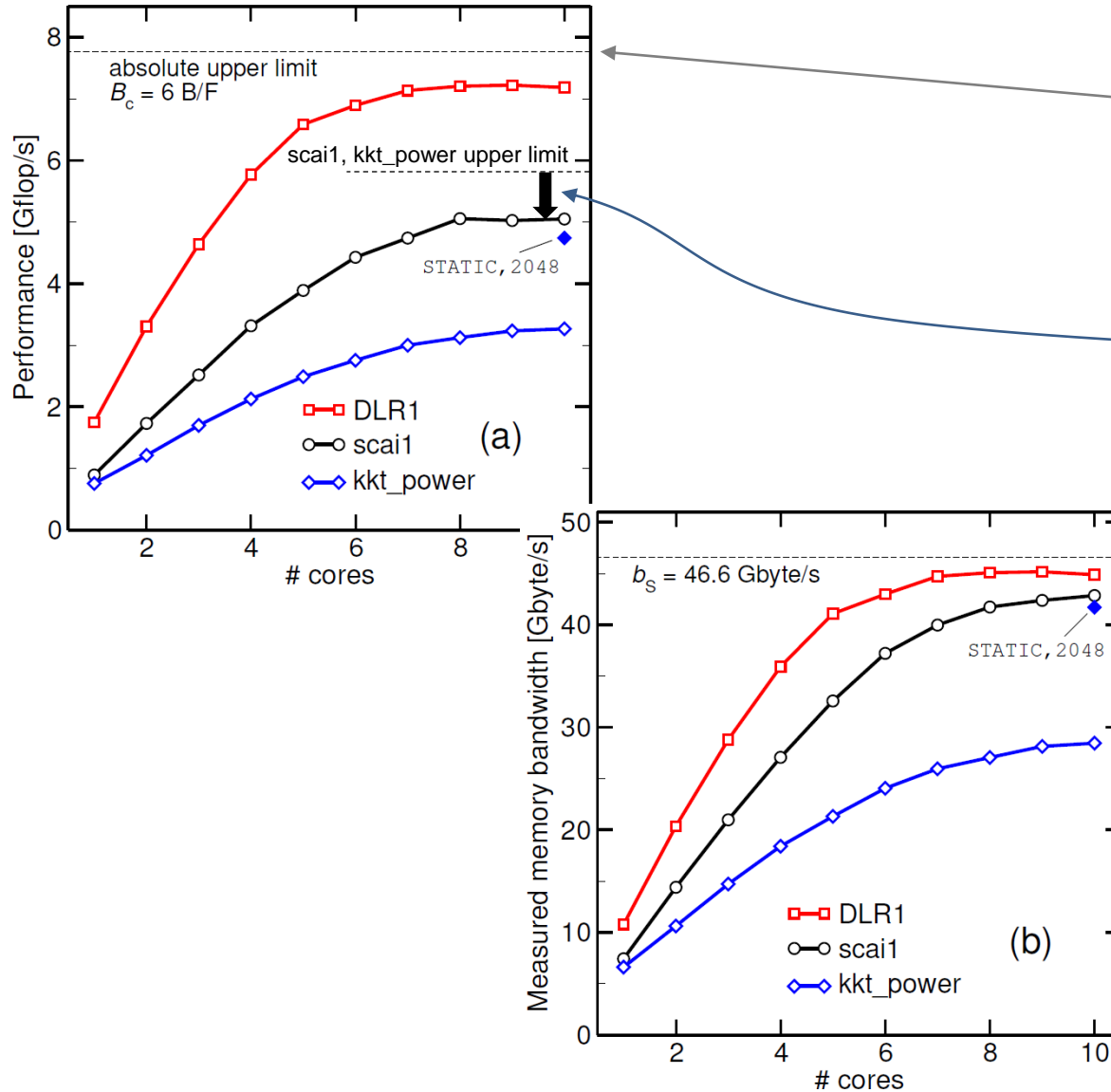


scai1



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:

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

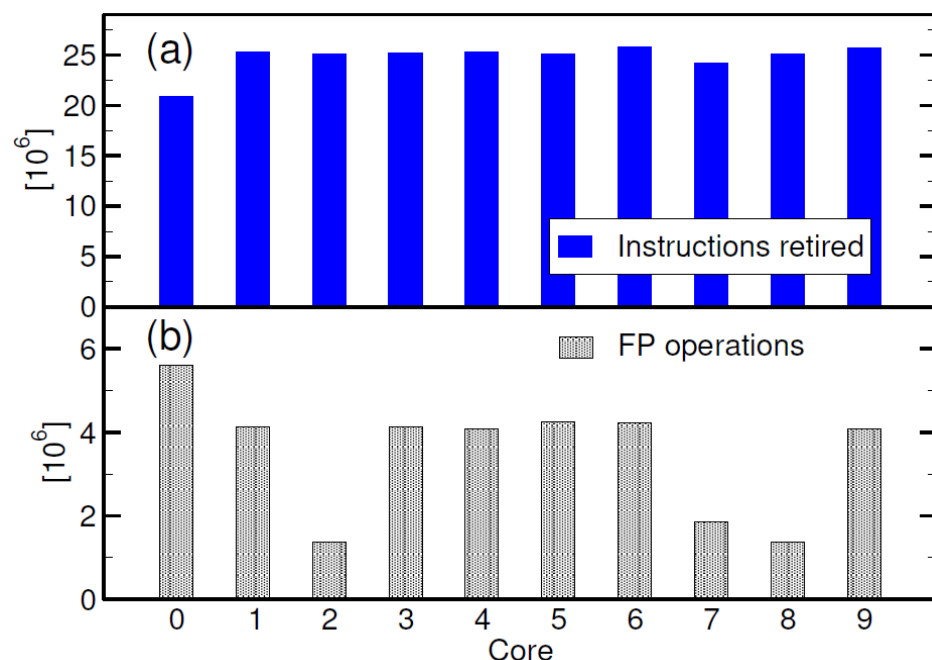
→ good BW utilization, slightly non-optimal α

- **kkt_power** measured balance:

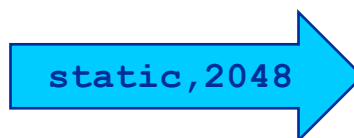
$$B_c^{meas} \approx 8.8 \text{ B/F} > B_{c,min} \text{ (10\% higher than min)}$$

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

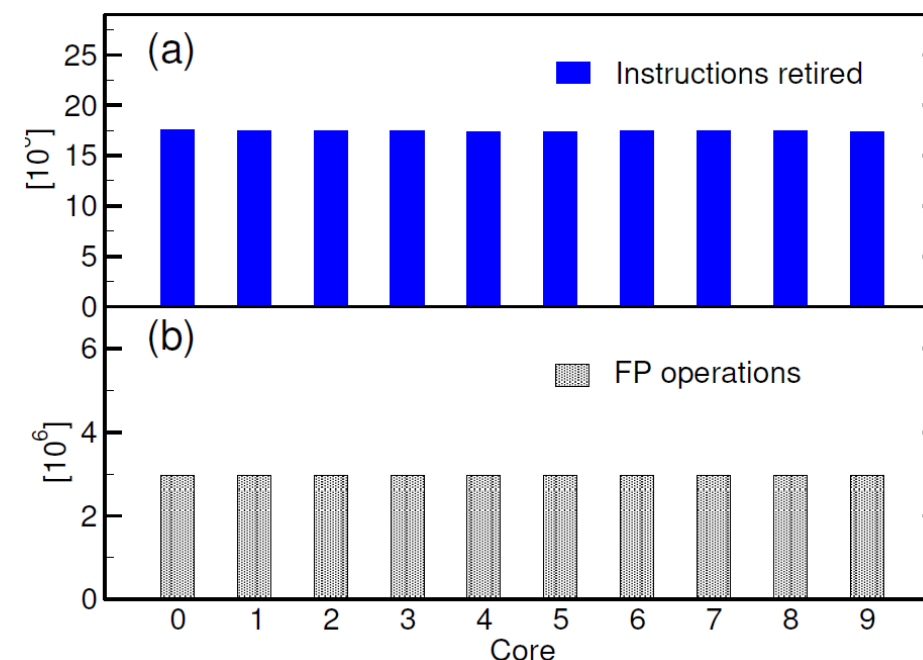
Investigating the load imbalance with kkt_power



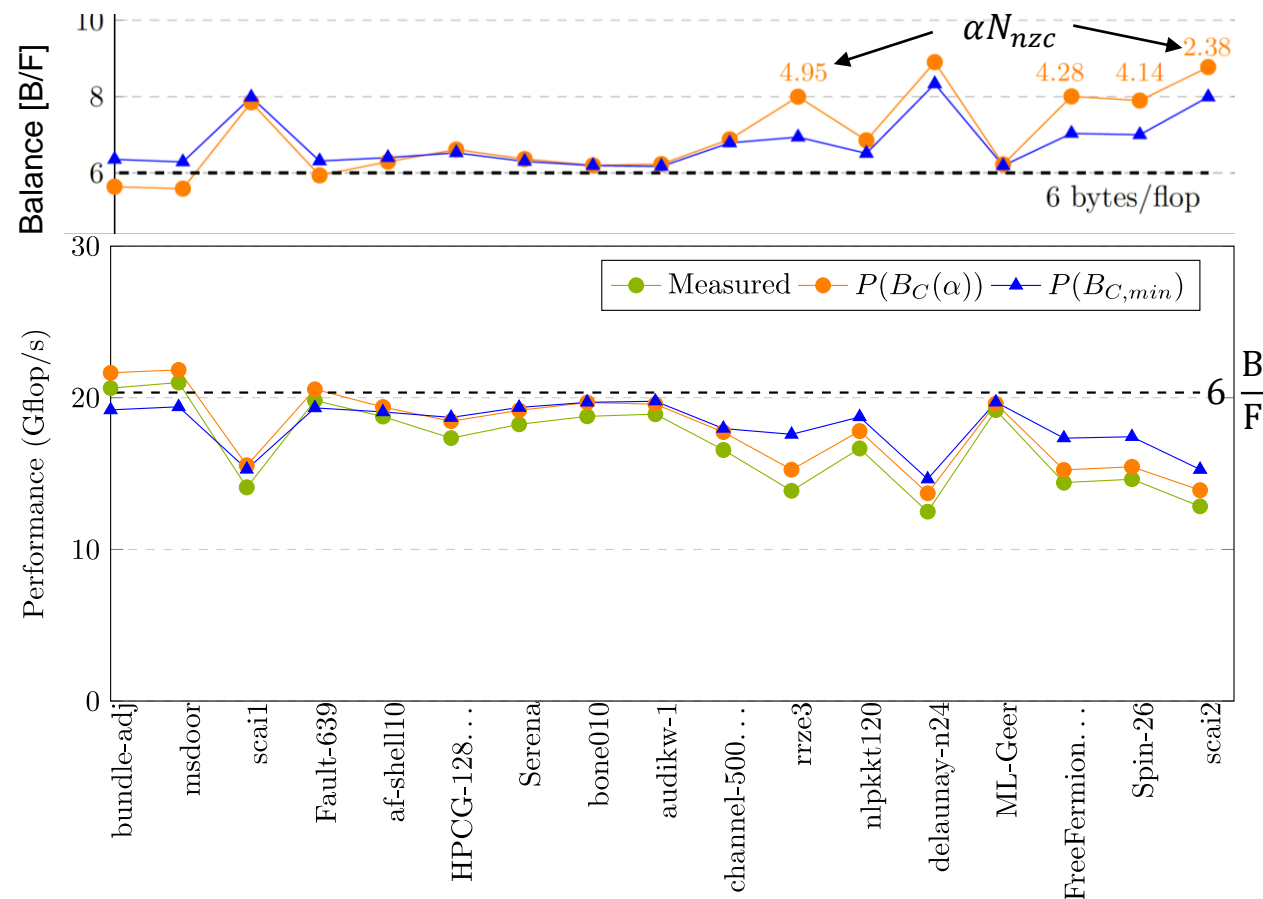
Measurements with likwid-perfctr
(MEM_DP group)



- 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*.
DOI: [10.1002/cpe.6512](https://doi.org/10.1002/cpe.6512)

When Roofline for SpMV may not work

Reasons for performance not attaining the limit

1. **Intensity lower than the minimum**

- More RHS traffic than the optimistic limit ($\frac{4}{N_{nzs}} B/F$)

2. **“Slow code”**

- “invisible” performance ceiling due to inefficient instructions or inefficient execution

3. **Load imbalance**

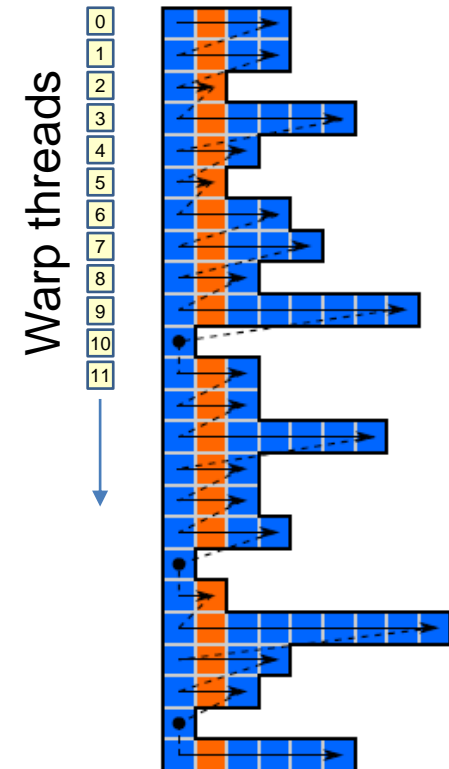
- A single process/thread cannot saturate the memory bandwidth

4. **Erratic memory access** patterns for RHS

- Latency dominates

What about GPUs?

- GPUs need
 - Enough 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,...



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

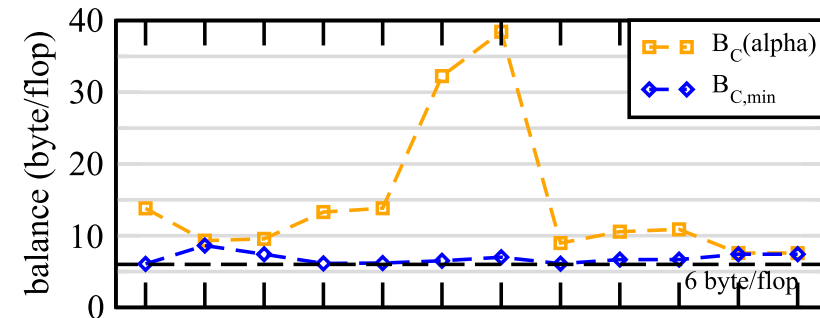
    if (row < num_rows) {
        VT sum{};
        for (IT j = row_ptrs[row]; j < row_ptrs[row + 1]; ++j) {
            sum += values[j] * x[col_idx[j]];
        }
        y[row] = sum;
    }
}
```

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

No write-allocate on GPUs for consecutive stores

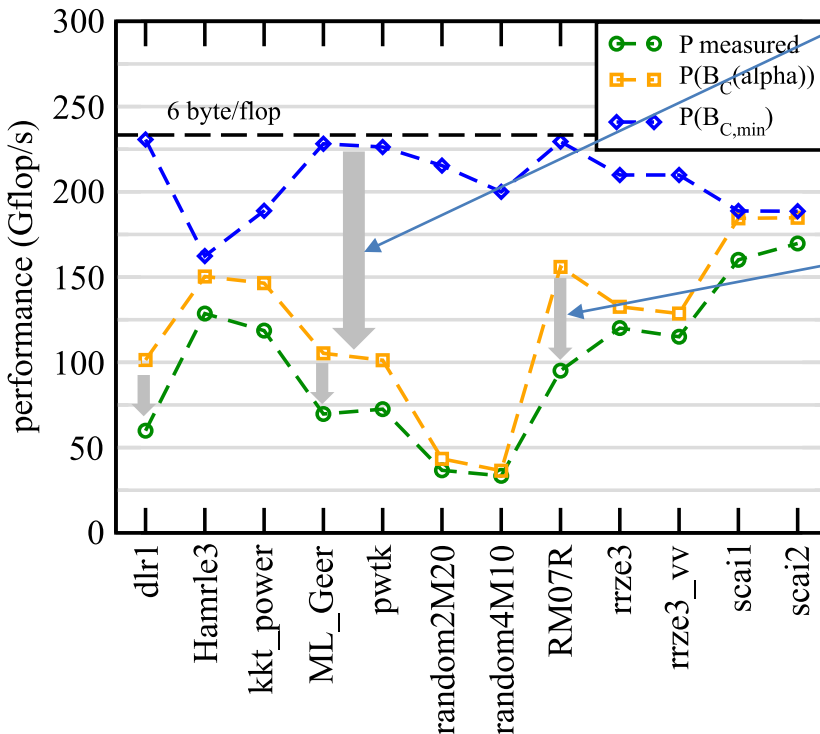
SpMV CRS performance on a GPU

CRS (1 thread per row)



NVIDIA Ampere A100

Memory bandwidth $b_S = 1400$ GB/s



Strong “ α effect” – large deviation from optimal α for many matrices

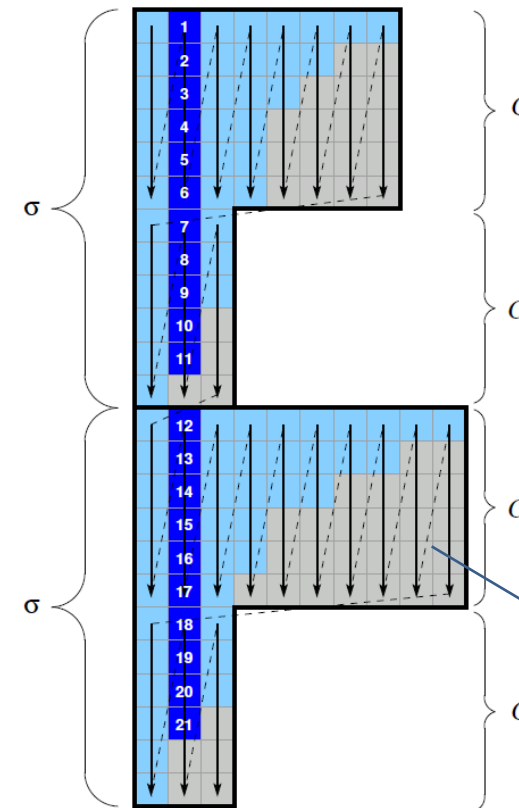
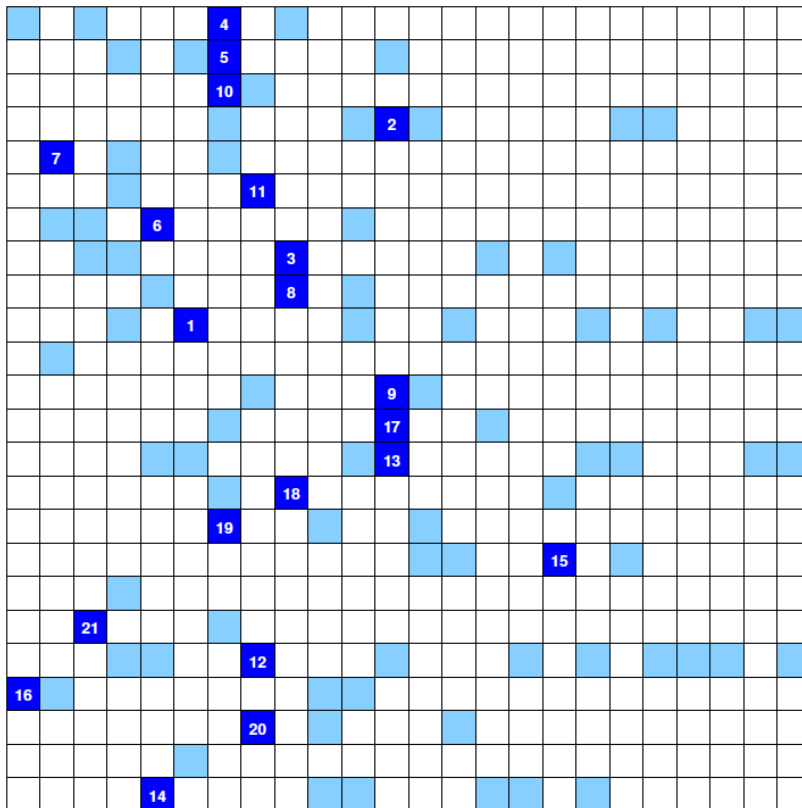
- Many cache lines touched b/c every thread handles one row \rightarrow bad cache usage

Mediocre memory bandwidth usage ($\ll 1400$ GB/s) in many cases

- Non-coalesced memory access
- Imbalance across rows/threads of warps

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

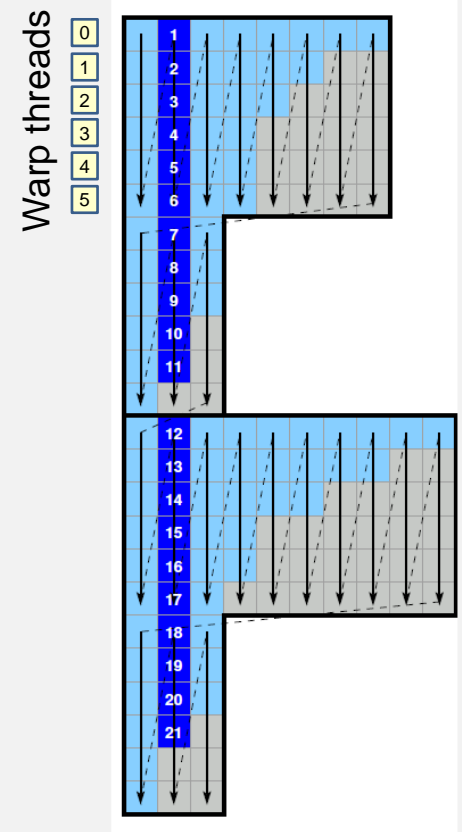
zero padding

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_idx,
         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_idx[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

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

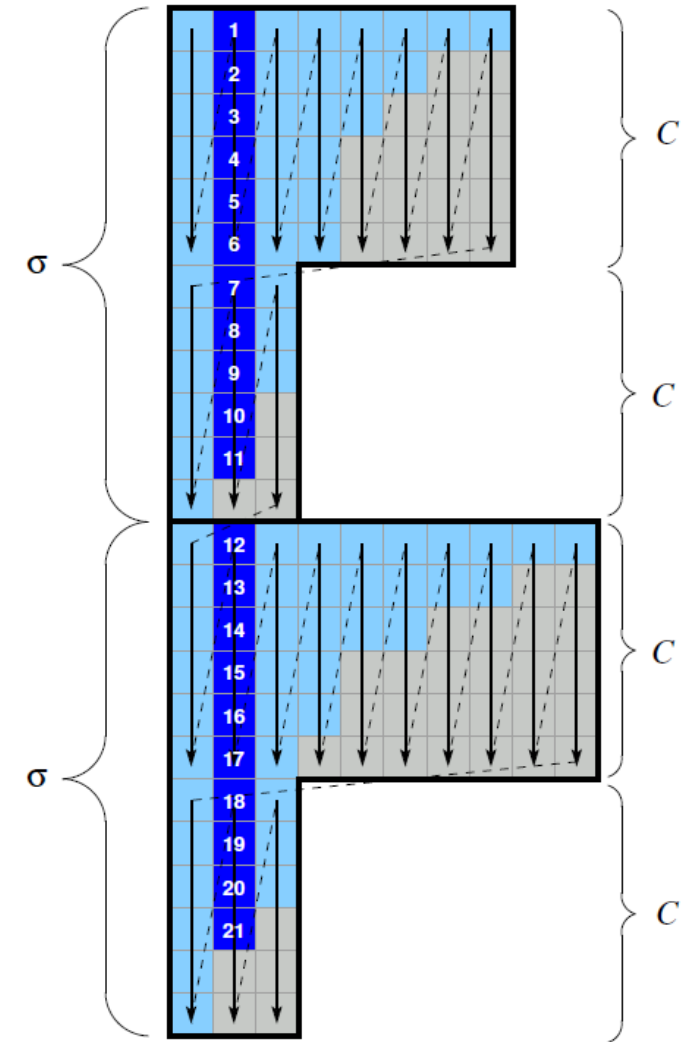
Optimal $\alpha = \frac{\beta}{N_{nzs}}$

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



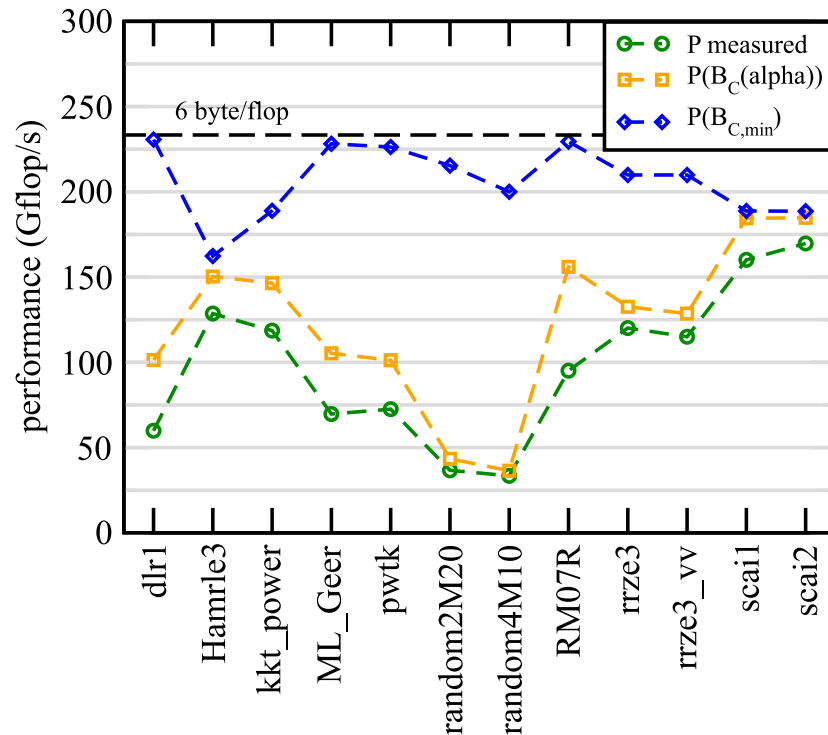
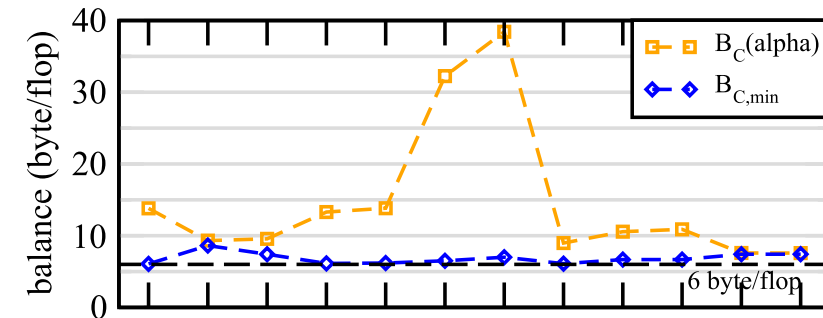
How to choose the parameters C and σ on GPUs?

- 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

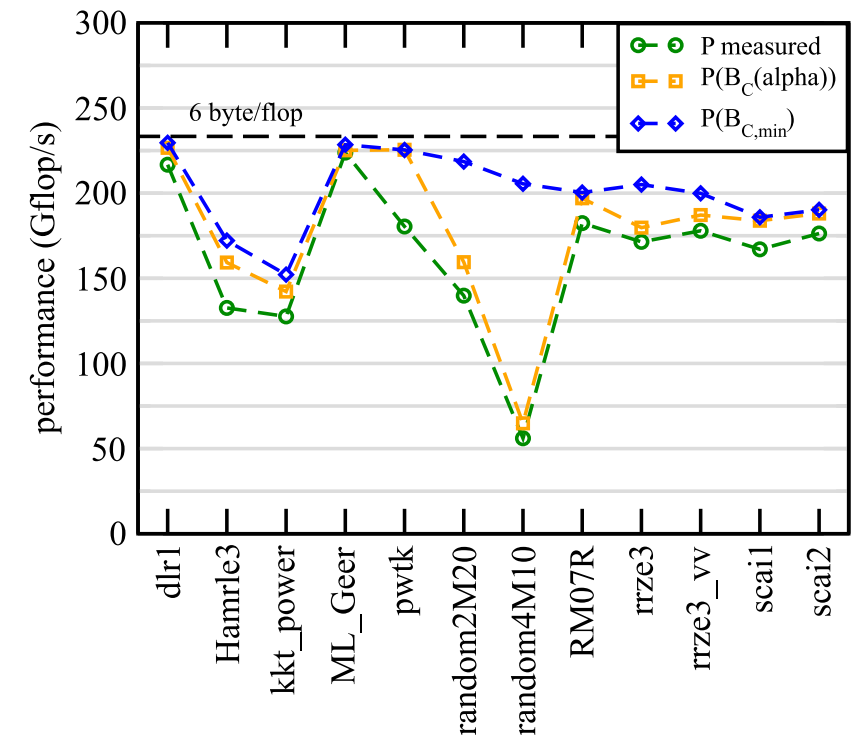
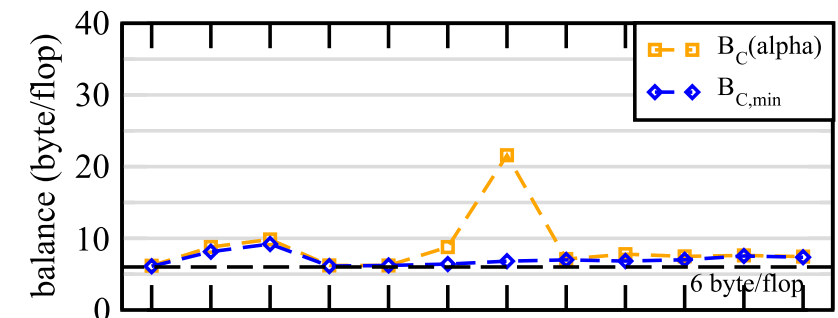
CRS (1 thread per row)



NVIDIA Ampere A100

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

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!*