Roofline Modeling and Performance Engineering

Georg Hager

Erlangen National High Performance Computing Center

CSCS-USI Summer University

July 23, 2022

Motivation

- Analytic performance modeling:

> Constructing a simplified model for the interaction between software and hardware in order to understand lowest-order performance behavior

- Basic questions addressed by analytic performance models
  - What is the bottleneck?
  - What is the next bottleneck after optimization?
  - Impact of hardware features → co-design, architectural exploration

- What if the model fails?
  - We learn something
  - We may still be able to use the model in a less predictive way

Friedrich-Alexander-Universität
Erlangen-Nürnberg

Resource-based performance models

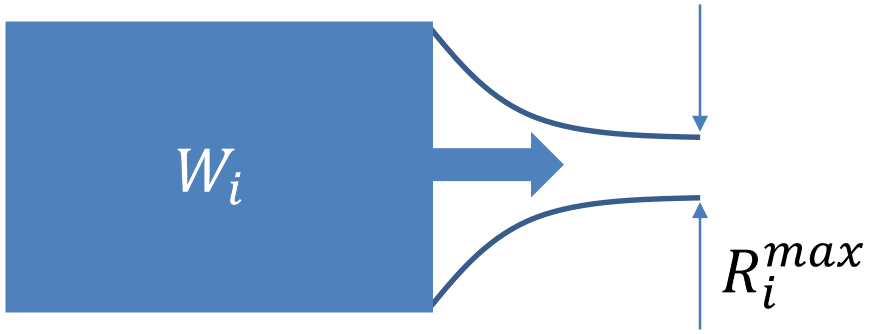
The questions to ask

How much of `$RESOURCE` does `$STUFF` need on `$HARDWARE`, and why?

→ Analytic, resource-based, first-principles models

A general view on resource bottlenecks

- What is the maximum performance when limited by a bottleneck?
- Resource bottleneck i delivers resources at maximum rate R_i^{max}
- W_i = needed amount of resources
- Minimum runtime: $T_i = \frac{W_i}{R_i^{max}} + \lambda_i$



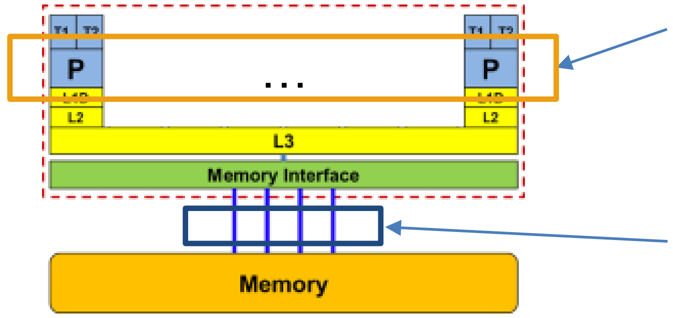
- Multiple bottlenecks → $T_{\text{expect}} = f(T_1, \ldots, T_n)$
- Overall performance: $P_{\text{expect}} = \frac{W}{T_{\text{expect}}}$

🤔

A simple bottleneck model of computing

Example: one loop, two bottlenecks

```
#pragma omp parallel for
for(i=0; i<10⁷; ++i)
  a[i] = a[i] + s * c[i];
```



n-core CPU
(1 CMG A64FX 2 GHz)

$$R_{flops}^{max} = 768\,\frac{\text{Gflops}}{\text{s}}$$

$$R_{BW}^{max} = 210\,\frac{\text{Gbyte}}{\text{s}}$$

$$W_{flops} = 2 \times 10^7 \text{ flops}$$

$$W_{BW} = 3 \times 8 \times 10^7 \text{ bytes}$$

$$T_{flops} = \frac{2 \times 10^7 \text{ flops}}{768\,\frac{\text{Gflops}}{\text{s}}} = 26.0\ \mu s$$

$$T_{BW} = \frac{2.4 \times 10^8 \text{ bytes}}{210\,\frac{\text{Gbyte}}{\text{s}}} = 1.14 \text{ ms}$$

Bottleneck models

How do we reconcile the multiple bottlenecks?
I.e., what is the functional form of $f(T_1,\dots T_n)$?

→ pessimistic model (no overlap): $f(T_1,\dots T_n) = \sum_i T_i$
→ optimistic model (full overlap): $f(T_1,\dots T_n) = \max(T_1,\dots T_n)$

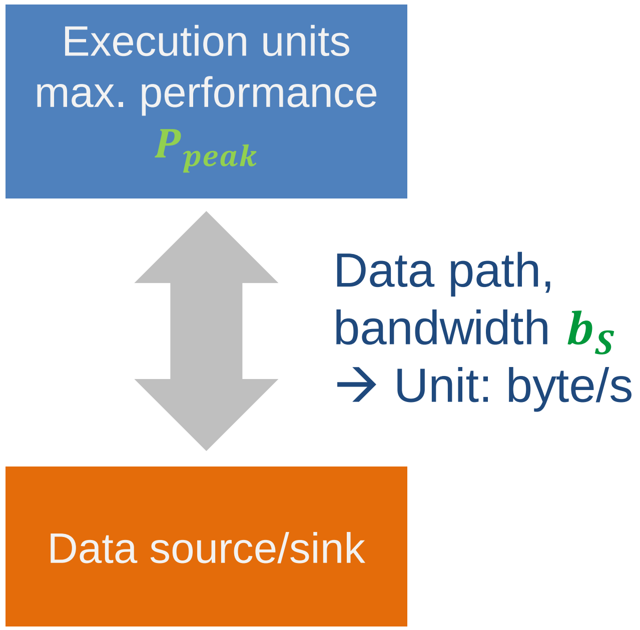
Roofline model
(Hockney et al., 1989
Williams et al., 2008)

Roofline for our example: $T_{\min} = \max\left(T_{flops}, T_{BW}\right) = 1.14\ \mathrm{ms}$

Maximum performance ("light speed"): $P_{\text{expect}} = \frac{2\times10^7}{1.14\times10^{-3}}\frac{\text{flops}}{\text{s}} = 17.5\ \text{Gflop/s}$

Roofline: A simple performance model for loops

Simplistic view of the hardware:


Execution units max. performance P_{peak}
Data path, bandwidth b_S
→ Unit: byte/s
Data source/sink

Simplistic view of the software:

```
! may be multiple levels
do i = 1,<sufficient>
  <complicated stuff doing
    N flops causing
    V bytes of data transfer>
enddo
```

Computational intensity $I = \frac{N}{V}$
→ Unit: flop/byte

Naïve Roofline Model: The slowest bottleneck wins

How fast can work be done at maximum? ***P*** [flop/s]

The bottleneck is either

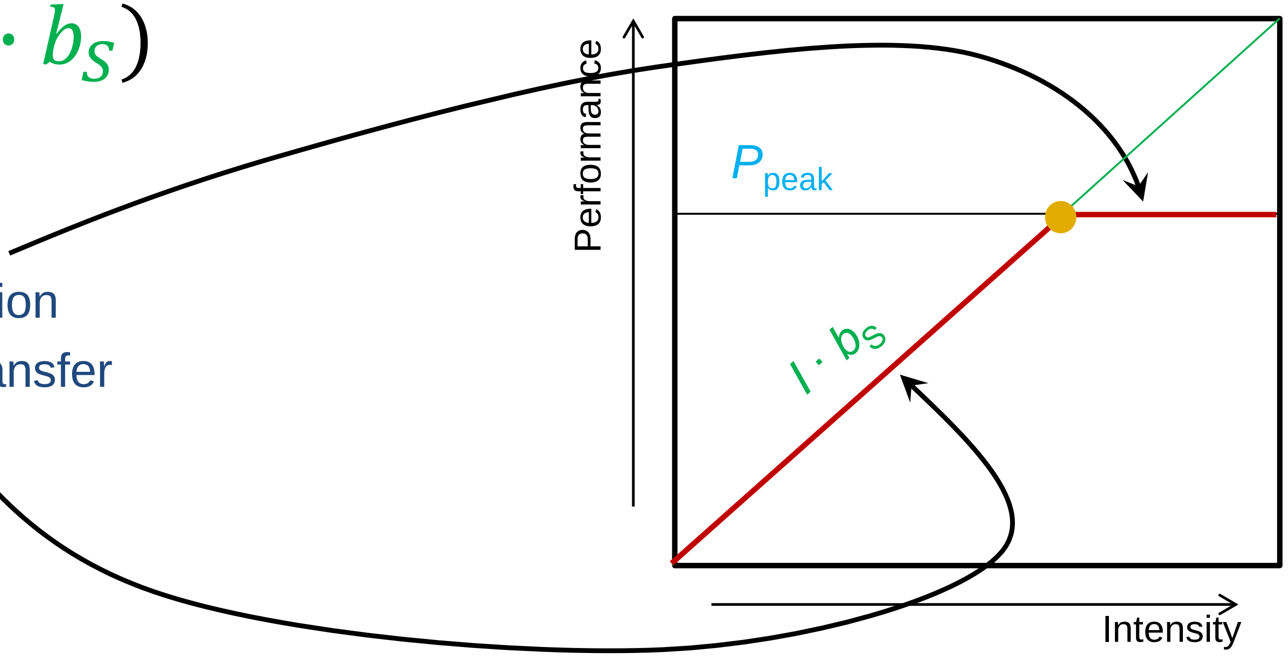
- The execution of work: P_{peak} [flop/s]
- The data path: $I \cdot b_S$ [flop/byte x byte/s]

$$P = \min(P_{\text{peak}}, I \cdot b_S)$$

This is the "Naïve Roofline Model"

- High intensity: P limited by execution
- Low intensity: P limited by data transfer
- "Knee" at $P_{peak} = I \cdot b_S$: Best use of resources

- Roofline is an "optimistic" model (think "light speed")



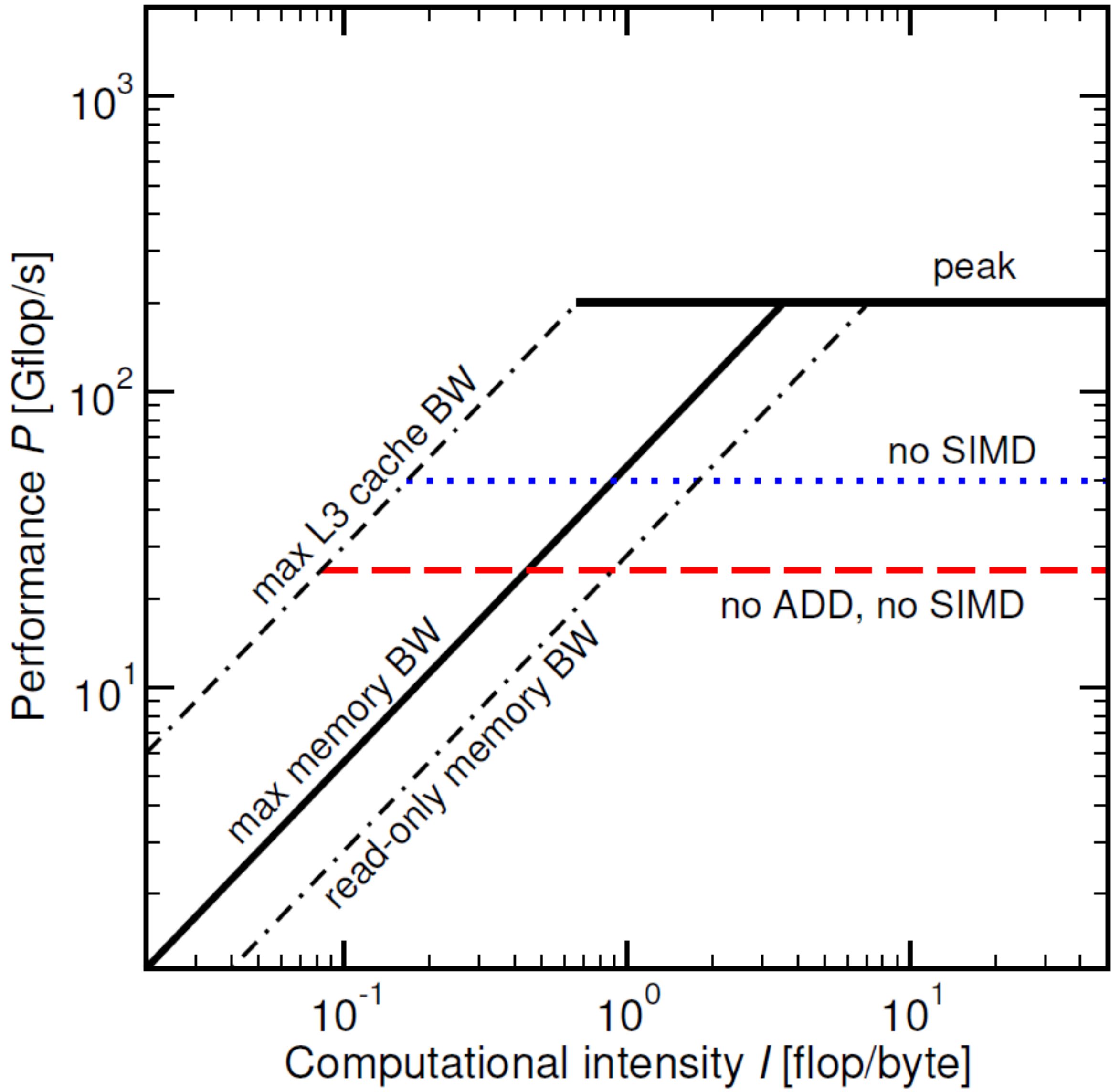
More bottlenecks, more ceilings

Multiple ceilings may apply

- Different bandwidths / data paths
  → different inclined ceilings

- Different P_{max}
  → different flat ceilings

  In fact, P_{max} should always come from code analysis; generic ceilings are usually impossible to attain



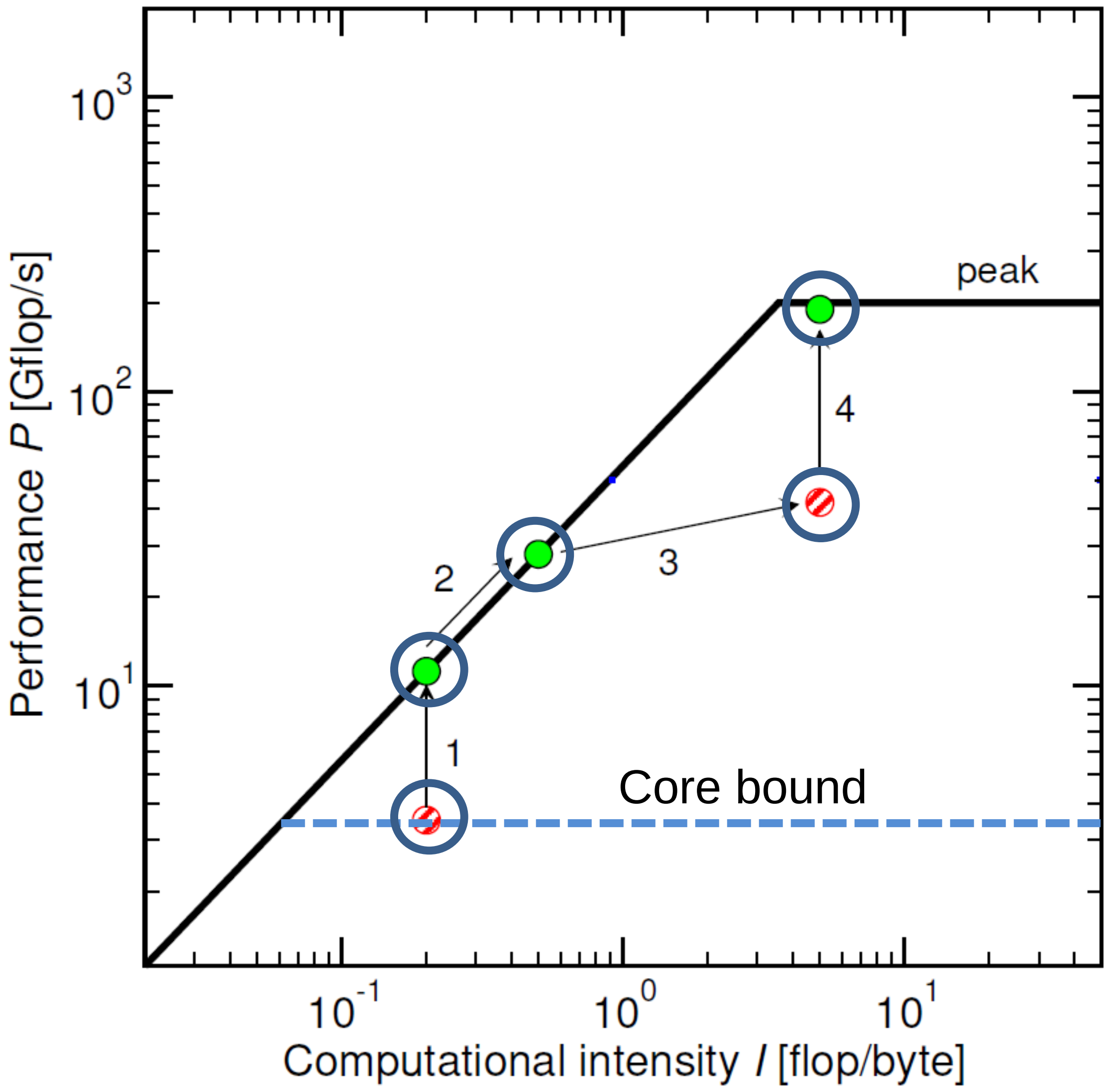
Tracking code optimizations in the Roofline Model

1. Hit the BW bottleneck by good serial code
   (e.g., Ninja C++ → Fortran)

2. Increase intensity to make better use of BW bottleneck
   (e.g., spatial loop blocking)

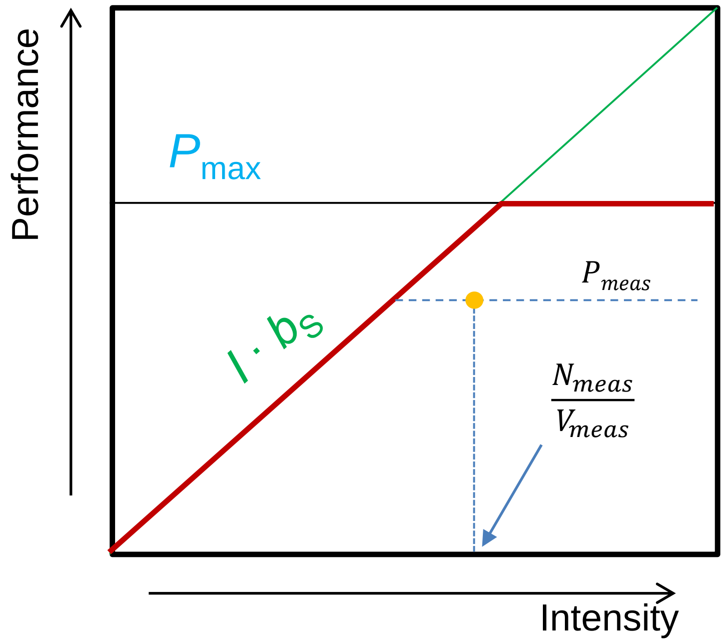
3. Increase intensity and go from memory bound to core bound
   (e.g., temporal blocking)

4. Hit the core bottleneck by good serial code
   (e.g., `-fno-alias`, SIMD intrinsics)

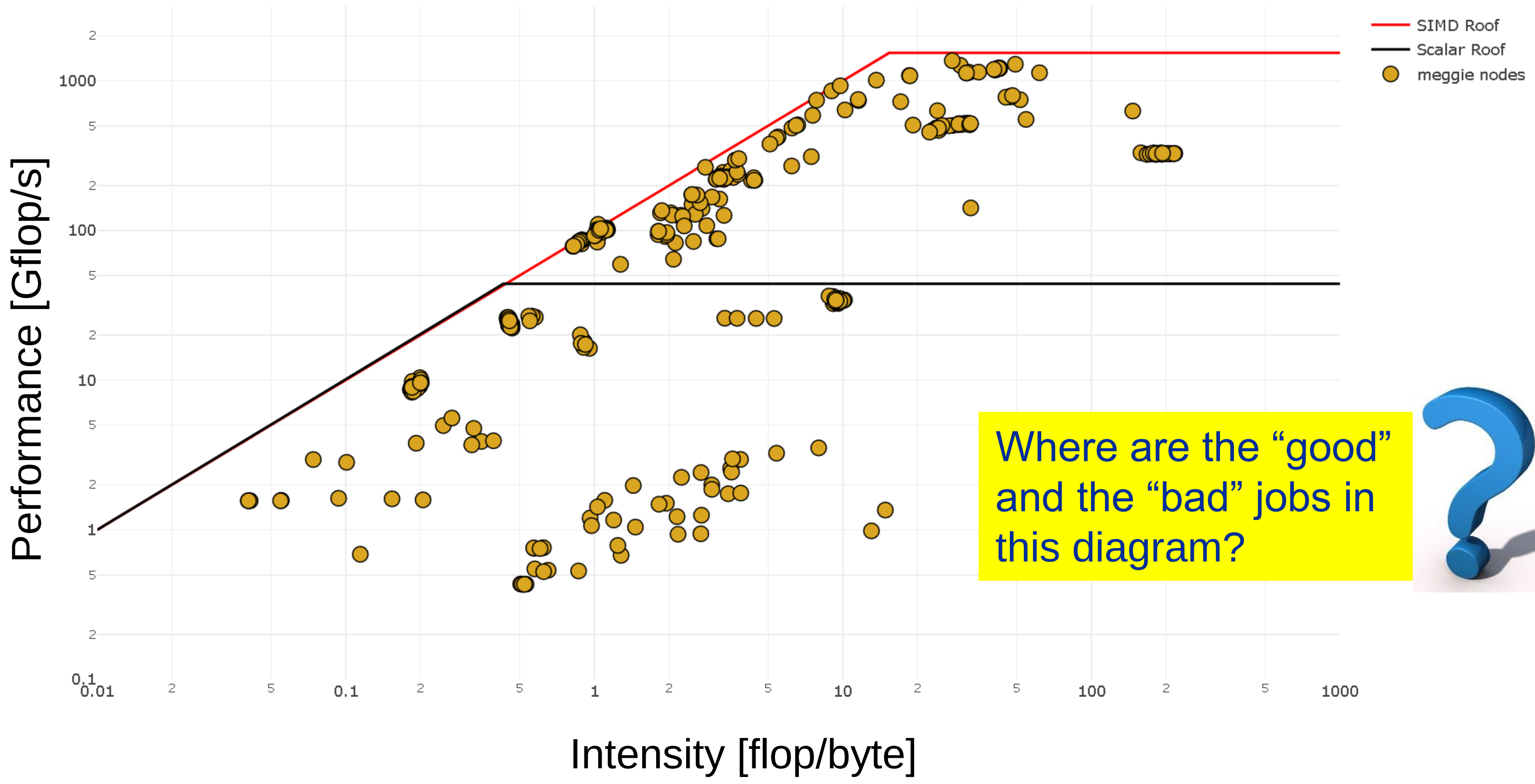


Diagnostic modeling

- What if we cannot predict the intensity/balance?
  - Code very complicated
  - Code not available
  - Parameters unknown
  - Doubts about correctness of analysis
- Measure data volume V_{meas} (and work N_{meas})
  - Hardware performance counters
  - Tools: likwid-perfctr, PAPI, Intel Vtune,…
- Insights + benefits
  - Compare analytic model and measurement → validate model
  - Can be applied (semi-)automatically
  - Useful in performace monitoring of user jobs on clusters



Roofline and performance monitoring of clusters



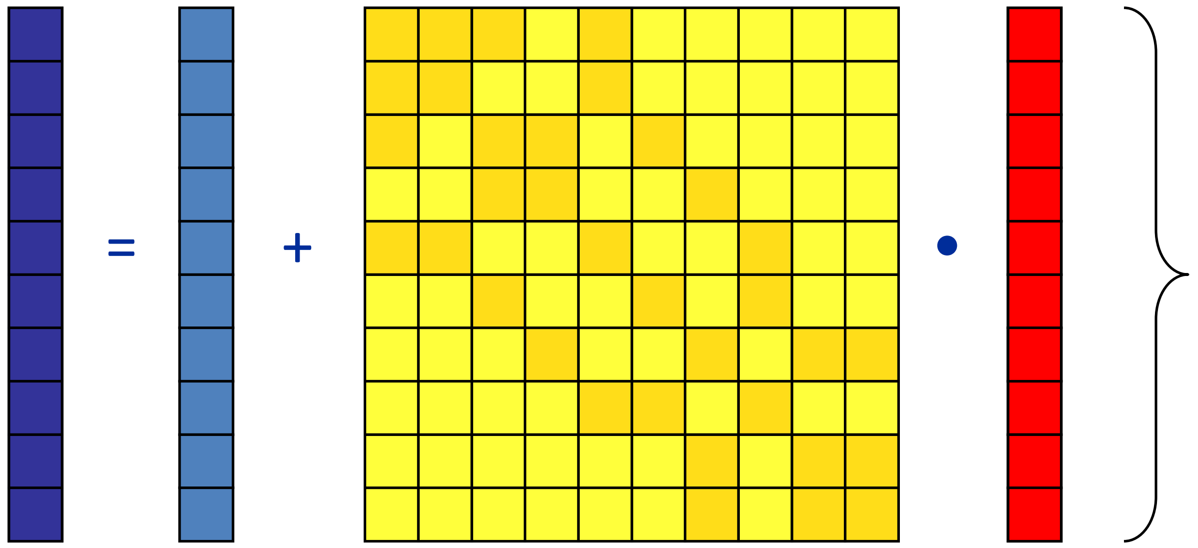
Where are the “good” and the “bad” jobs in this diagram?

https://github.com/RRZE-HPC/likwid/wiki/Tutorial%3A-Empirical-Roofline-Model

Case study:
Sparse Matrix-Vector Multiplication

Sparse Matrix Vector Multiplication (SpMV)

- Key ingredient in some important matrix algorithms
  - CG(+x), Pagerank, 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: $N_{nzr} = N_{nz}/N_r$



General case: some indirect addressing required!

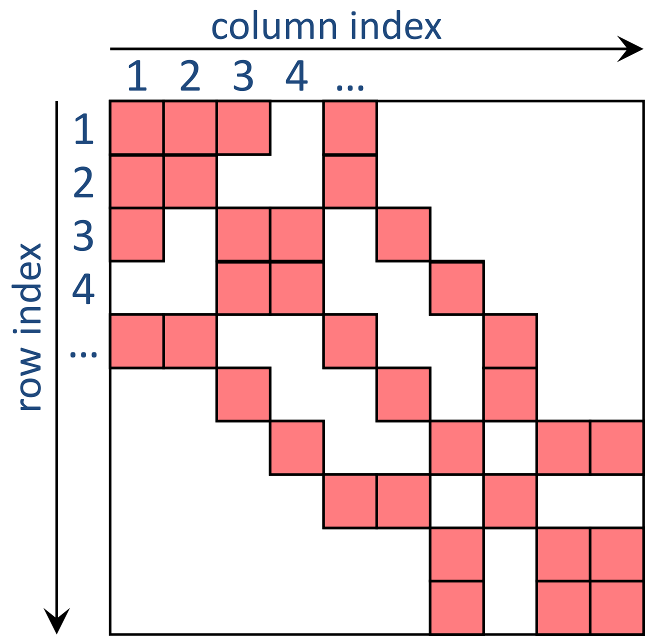
SpMVM characteristics

- For large problems, SpMV is inevitably memory-bound
  - Intra-socket saturation effect on modern multicores

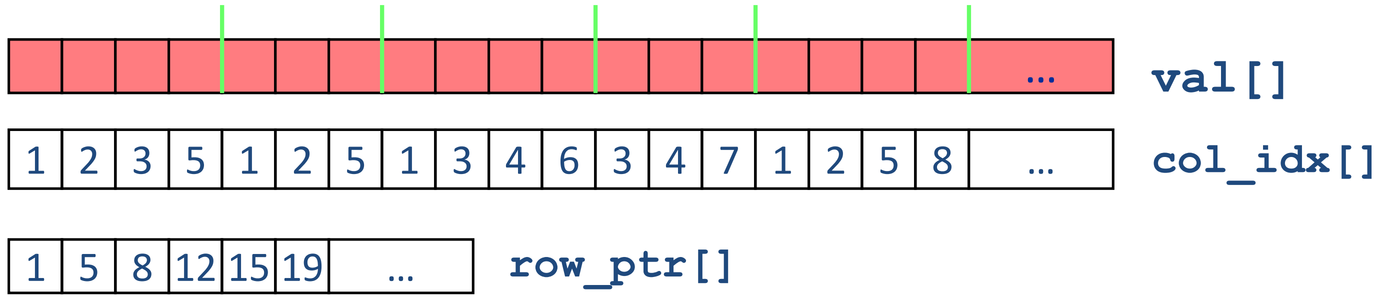
- SpMV is easily parallelizable in shared and distributed memory
  - Load balancing
  - Communication overhead

- Data storage format is crucial for performance properties
  - Most useful general format on CPUs:
    Compressed Row Storage (CRS/CSR)
  - Depending on compute architecture

CSR 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…

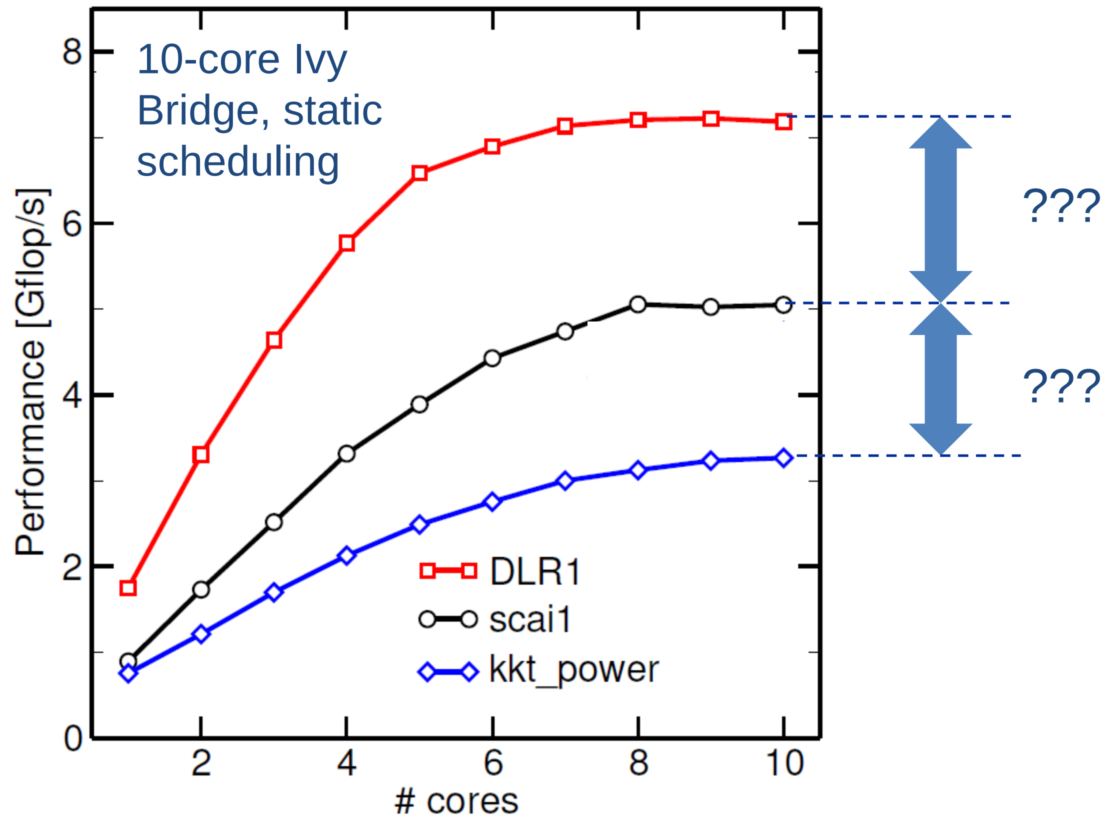
Performance characteristics

- 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 – CSR (1)

```
do i = 1, N_r
 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(N_nz)
integer*4   col_idx(N_nz)
integer*4   row_ptr(N_r)
real*8      C(N_r)
real*8      B(N_c)
```

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]: $2\,N_{nz}$

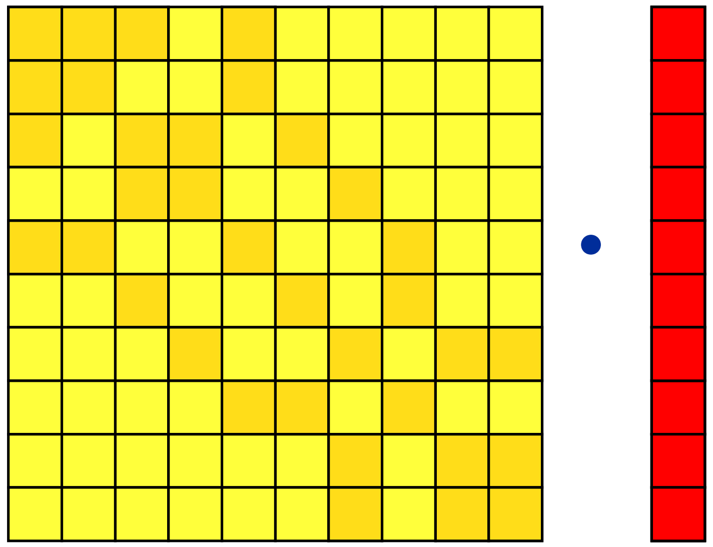
$$B_{C,min} = \frac{12\,N_{nz} + 20\,N_r + 8\,N_c}{2\,N_{nz}}\,\frac{B}{F} = \frac{12 + 20/N_{nzr} + 8/N_{nzc}}{2}\,\frac{B}{F}$$

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

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

SpMV node performance model – CSR (2)

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



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

$$B_C\,(\alpha) = \frac{12 + 20/N_{nzr} + \boxed{8\,\alpha}}{2} \frac{B}{F}$$

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

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

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

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

Note: $B_C\left({}^{1}/_{N_{nzr}}\right) = B_{C,min}$

The “α effect”

DP CRS code balance

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

$$= \left(6 + 4\,\alpha + \frac{10}{N_{nzr}}\right)\frac{B}{F}$$

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

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)

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

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

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

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

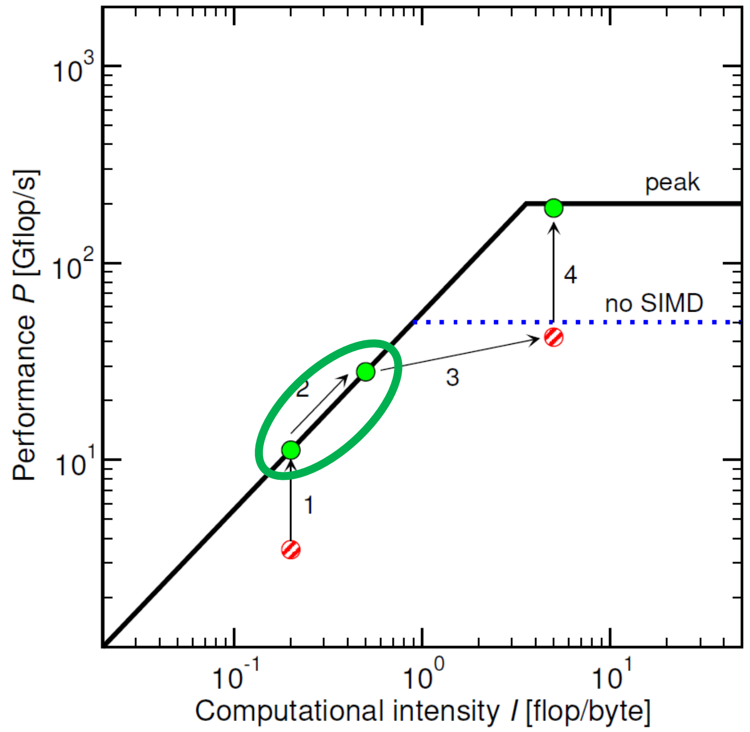
- $N_{nz} = 14.6\cdot 10^6, N_{nzr} = 7.1$
- $V_{meas} \approx 258\,\mathrm{MB}$

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

→ RHS is loaded 2.5 times from memory

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

11% extra traffic → optimization potential!

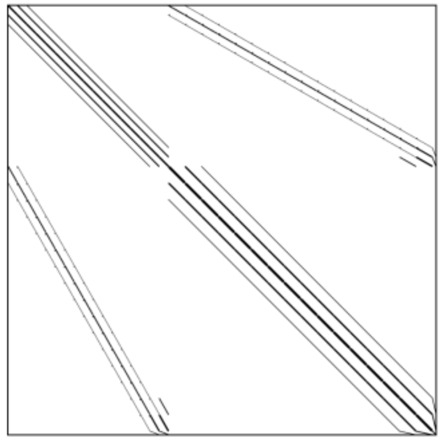


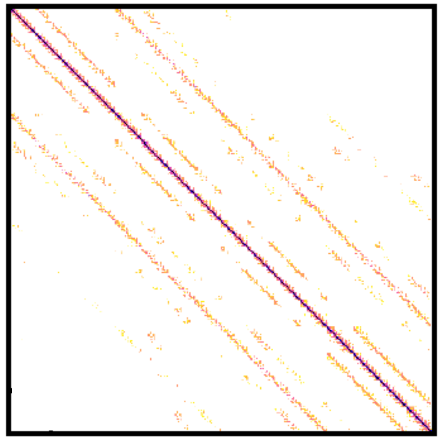
Three different sparse matrices

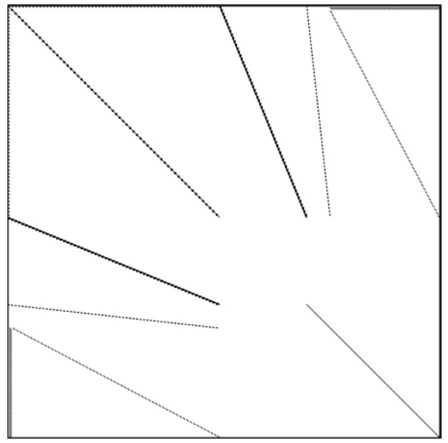
Benchmark system: Intel Xeon Ivy Bridge E5-2660v2, 2.2 GHz, $b_S = 46.6\,\mathrm{GB/s}$

→ Roofline: $P_{opt} = {}^{b_S}/_{B_{C,min}}$

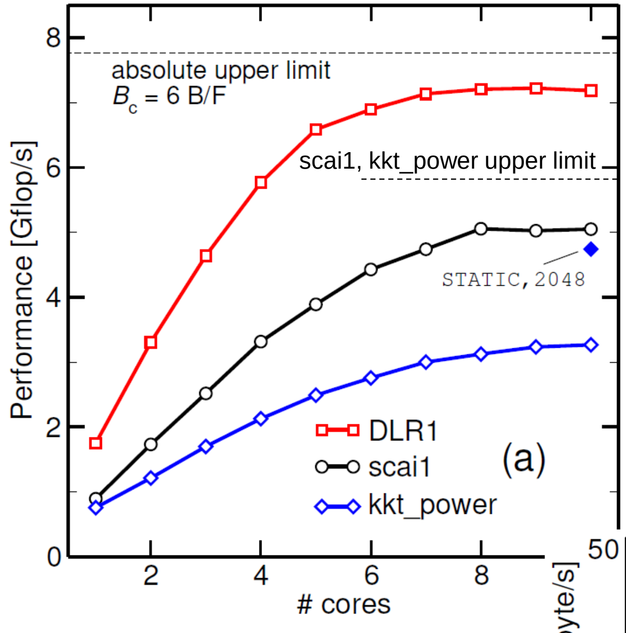
Matrix	N	N_{nzr}	$B_{C,min}$ [B/F]	P_{opt} [GF/s]
DLR1	278,502	143	6.1	7.64
scai1	3,405,035	7.0	8.0	5.83
kkt_power	2,063,494	7.08	8.0	5.83

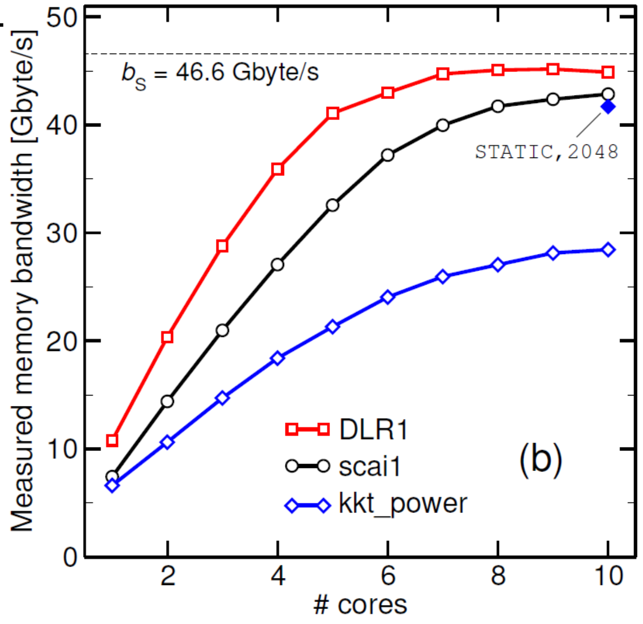

DLR1


scai1


kkt_power

Now back to the start…





- $b_S = 46.6\,\mathrm{GB/s}\,, \quad B_c \;= 6\,\mathrm{B/F}$
- Maximum spMVM performance:

$$P_{max} = 7.8\,\mathrm{GF/s}$$

- DLR1 causes minimum CRS code balance (as expected)

- scai1 measured balance:

$$B_c^{meas} \approx 8.5\,\mathrm{B/F} > B_{C,min}$$

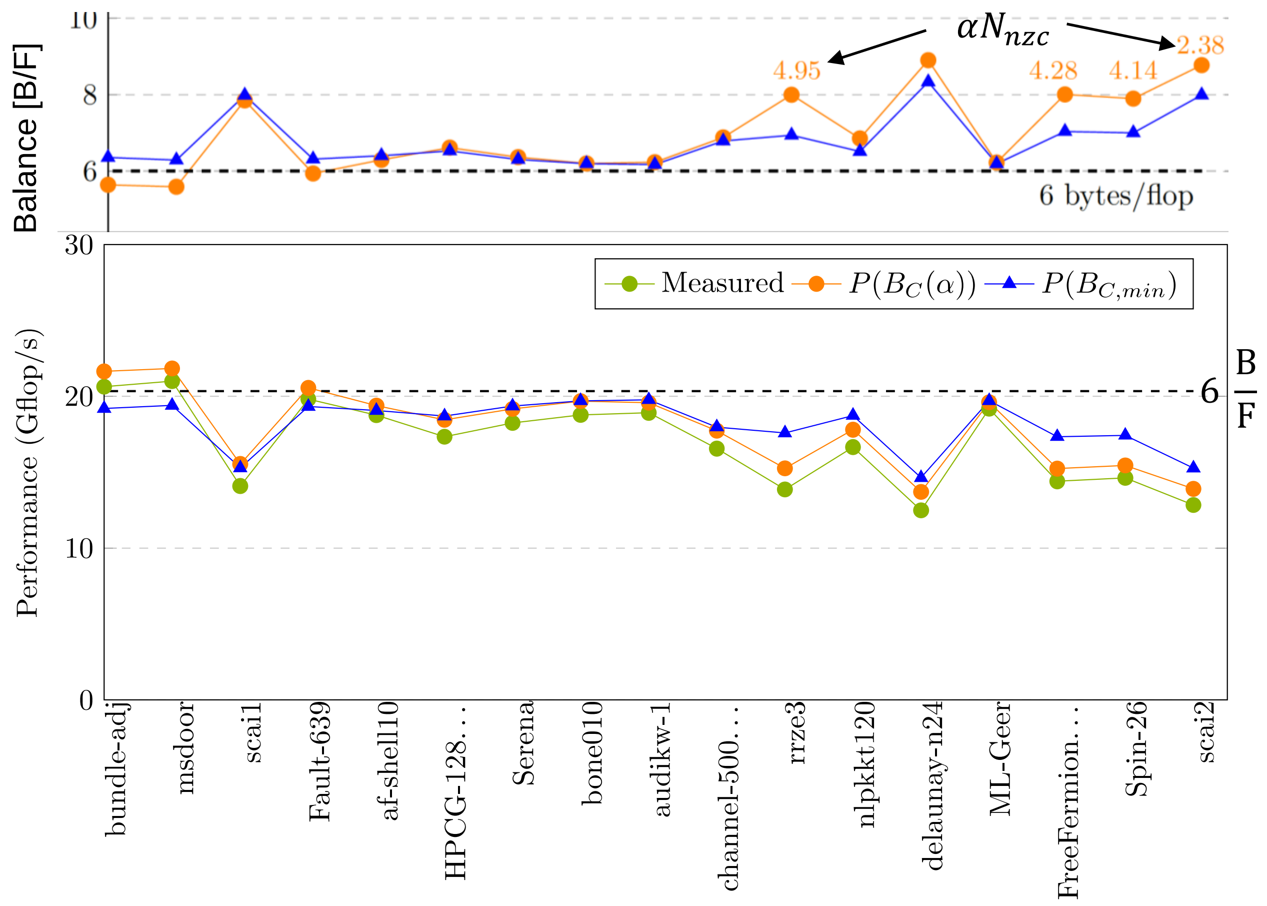
→ good BW utilization, slightly non-optimal α

- kkt_power measured balance:

$$B_c^{meas} \approx 8.8\,\mathrm{B/F} > B_{C,min}$$

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

SpMV node performance model – CPU



Intel Xeon Platinum 9242
24c@2.8GHz (turbo)
$b_S = 122\ GB/s$

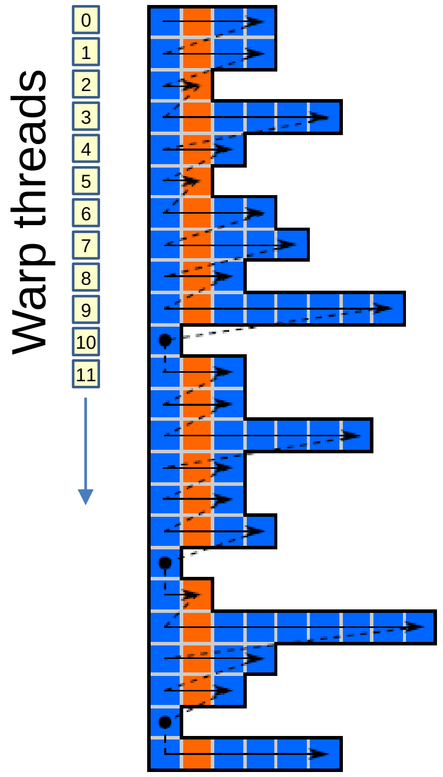
Matrices taken from: C. L. Alappat et al.: *ECM modeling and performance tuning of SpMV and Lattice QCD on A64FX.* https://doi.org/10.1002/cpe.6512

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)

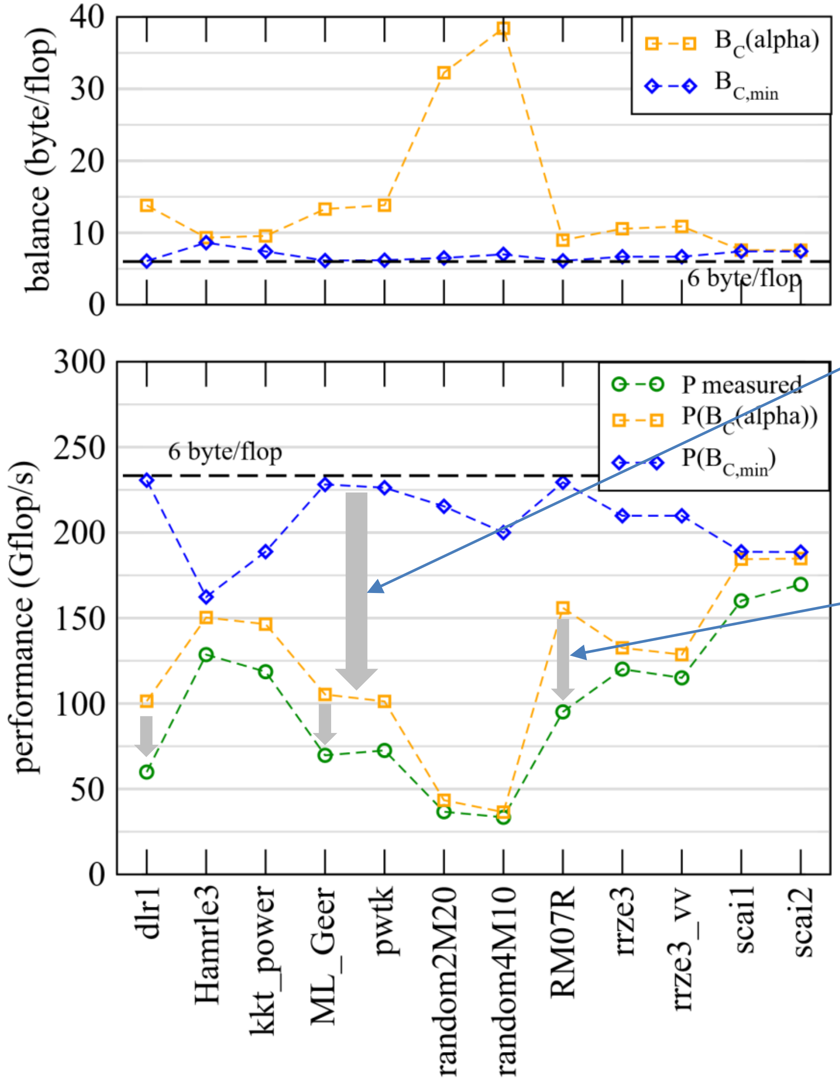
```
template <typename VT, typename IT>
__global__ static void
spmv_csr(const ST num_rows,
         const IT * RESTRICT row_ptrs, const IT * RESTRICT col_idxs,
         const VT * RESTRICT values,   const VT * RESTRICT x,
                                            VT * RESTRICT y)
{
    ST row = threadIdx.x + 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_idxs[j]];
        }
        y[row] = sum;
    }
}
```

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

No write-allocate on GPUs for consecutive stores

SpMV CRS performance on a GPU

CRS (1 thread per row)



NVIDIA Ampere A100
Memory bandwidth $b_S = 1400\ \mathrm{GB/s}$

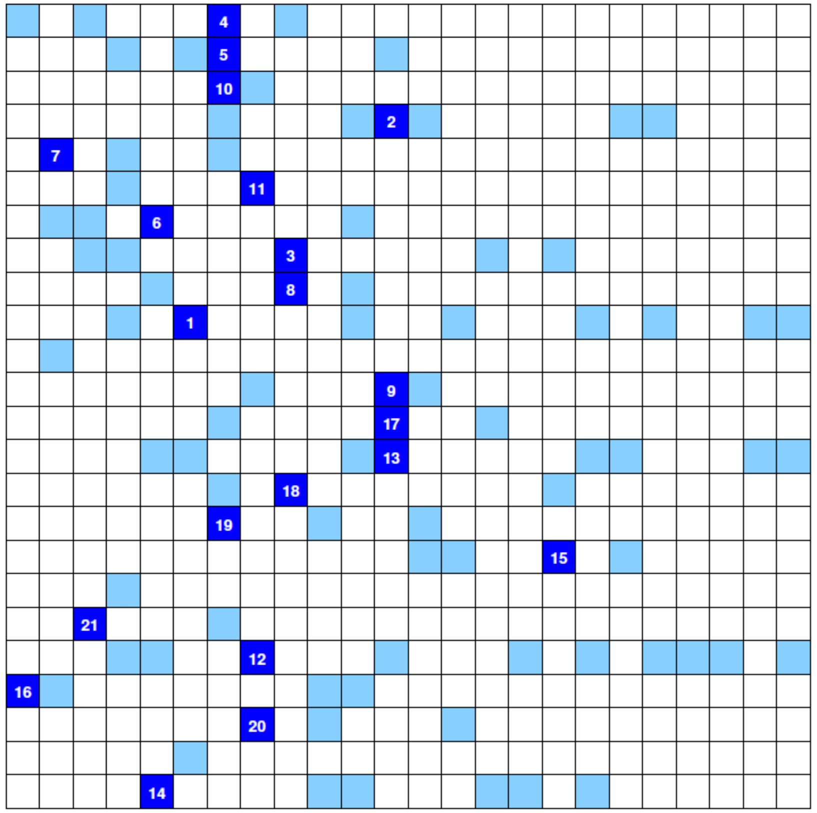
- Strong "α effect" – large deviation from optimal α for many matrices
  - Many cache lines touched b/c every thread handles one row → bad cache usage
- Mediocre memory bandwidth usage ($\ll 1400\ \mathrm{GB/s}$) in many cases
  - Non-coalesced memory access
  - Imbalance across rows/threads of warps

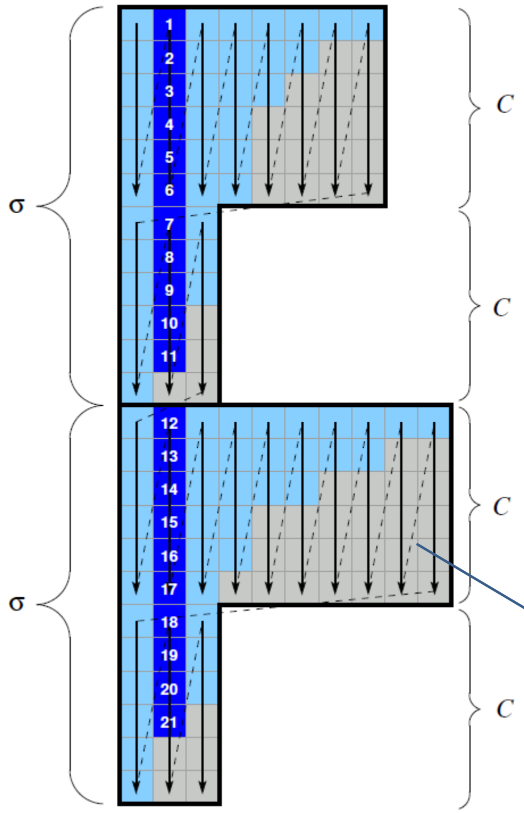
SELL-C-σ

M. Kreutzer et al.: *A Unified Sparse Matrix Data Format For Efficient General Sparse Matrix-vector Multiplication On Modern Processors With Wide SIMD Units*, SIAM SISC 2014, DOI: 10.1137/130930352

Idea

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





"Chunk occupancy":

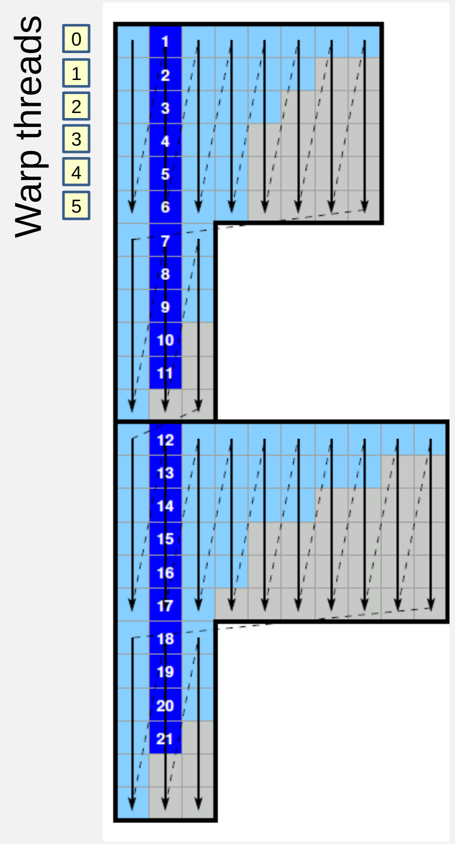
$$\beta = \frac{N_{nz}}{\sum_{i=0}^{N_c} C \cdot l_i}$$

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

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

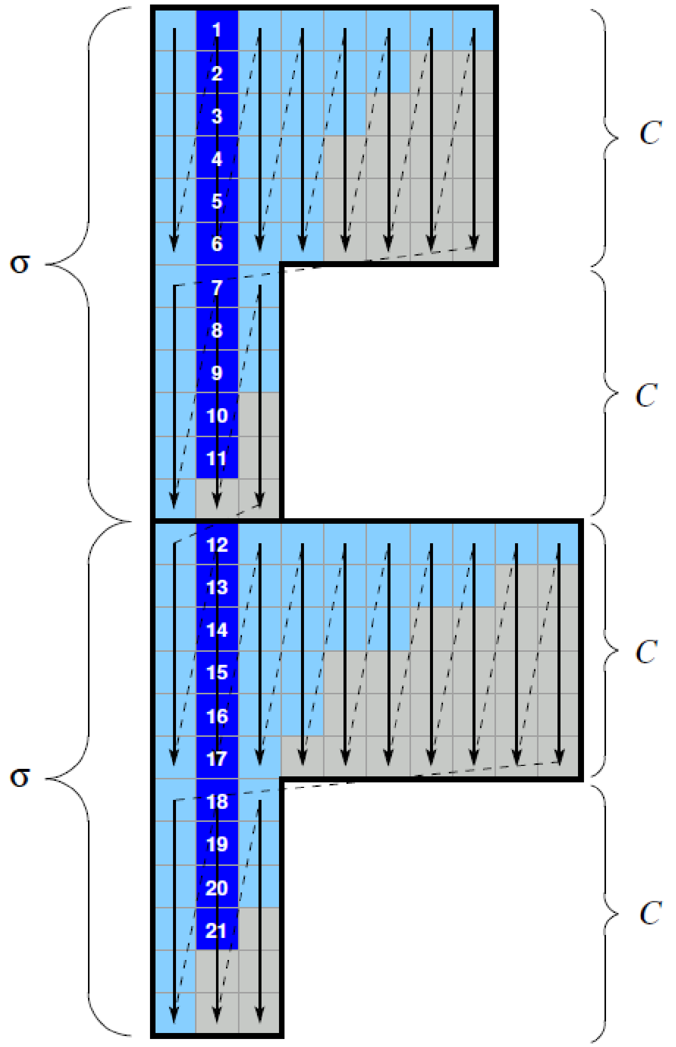
$$= \left(\frac{6}{\beta} + 4\alpha + \frac{\beta(4+2/C)}{N_{nzr}}\right)\frac{\text{bytes}}{\text{flop}}$$

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

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

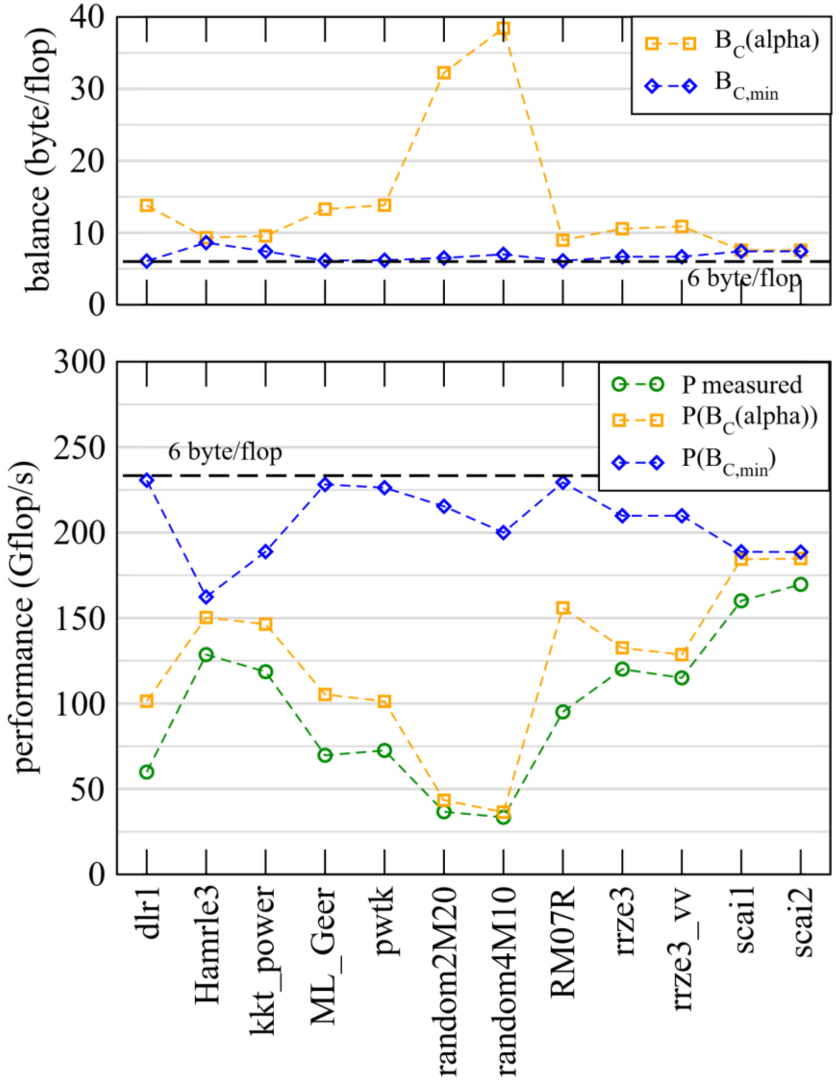
How to choose the parameters C and σ on GPUs?

- 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 → α depends on σ



SpMV node performance model – GPU

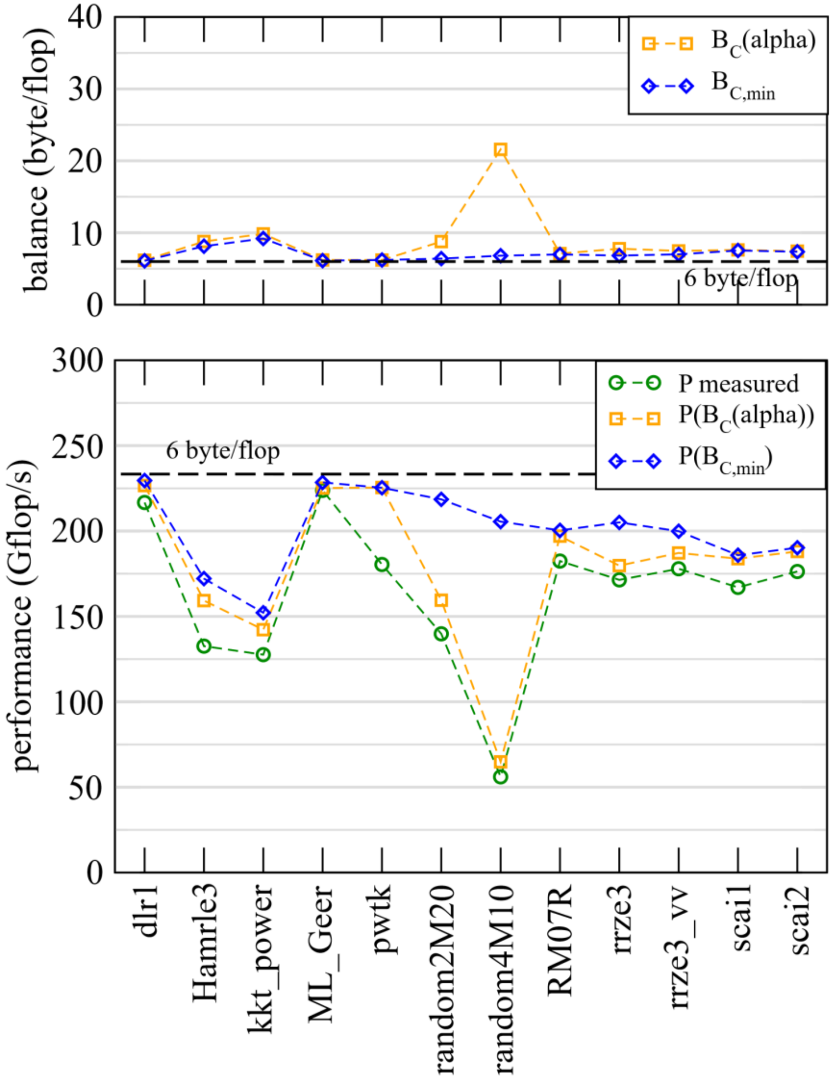
CRS (1 thread per row)



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!

- Consequence: Modeling is not always 100% predictive. It‘s all about *learning more* about performance properties!

Thank You.