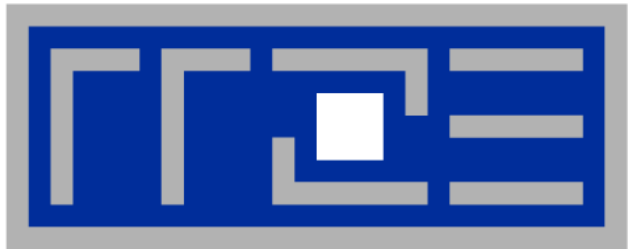


# Performance Engineering – Case study:

# Basic understanding of performance of sparse matrix vector multiplication (spMVM)

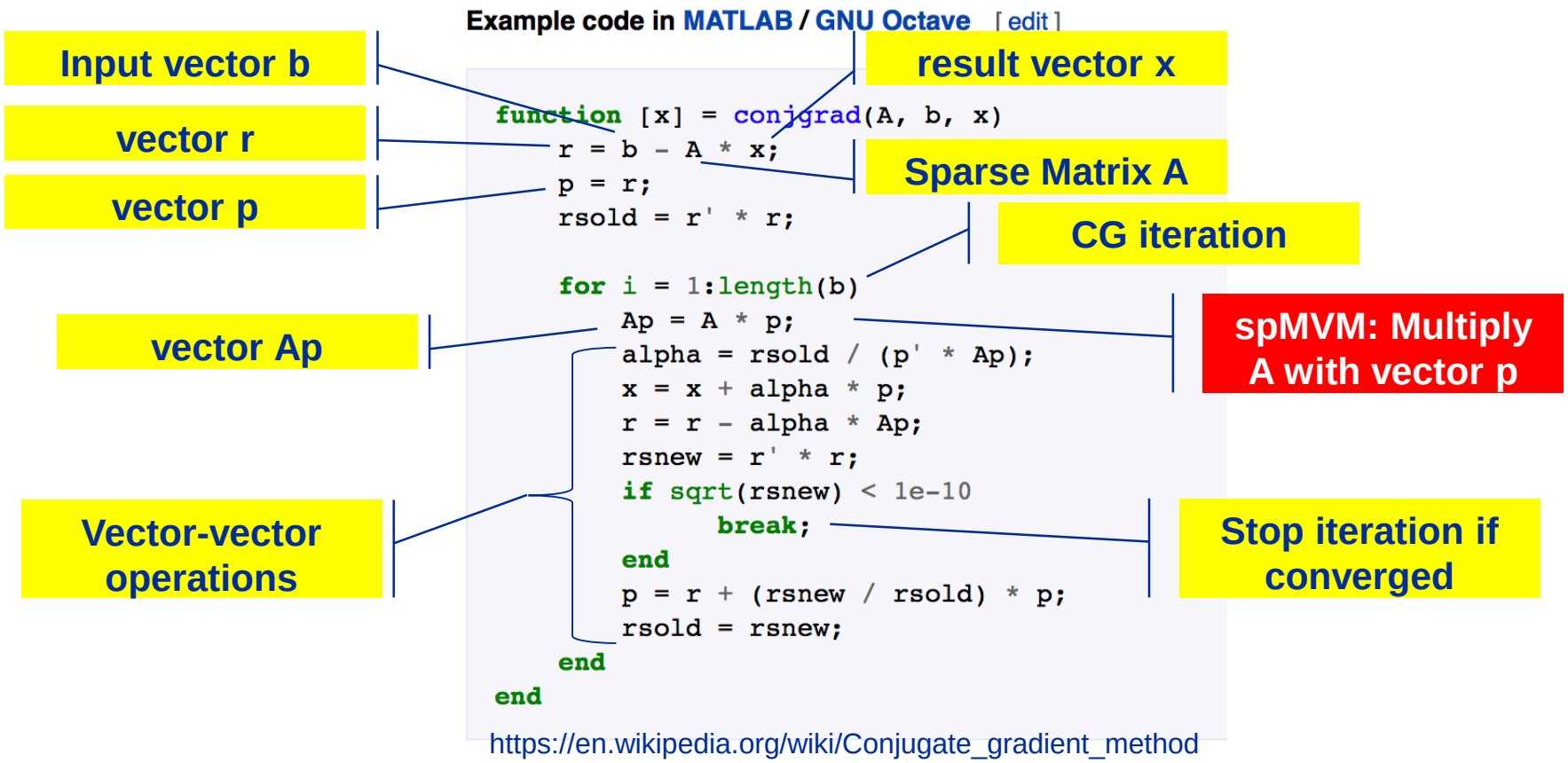
Prof. Dr. G. Wellein[a,b] , Dr. G. Hager[a], J. Hammer[b], C.L. Alappat[b]

[a]HPC Services – Regionales Rechenzentrum Erlangen
[b]Department für Informatik

University Erlangen-Nürnberg, Sommersemester 2019

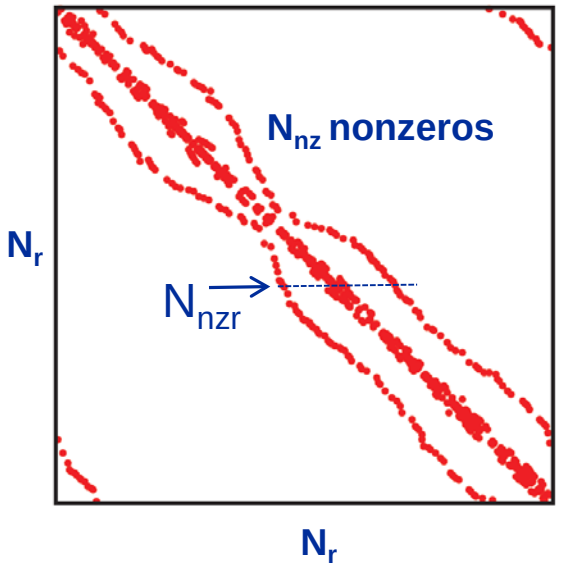
# Motivation – spMVM: most expensive part in many iterative solver – Conjugate Gradient Method

- Problem: Solve **A x = b** with **A** large sparse (n x n) matrix and **b** is known vector
- Use (iterative) Conjugate Gradient (CG) Method



# Sparse MVM

- **Sparse Matrix Vector Multiply (spMVM) is the basic operation in many numerical applications, e.g.**
  - Finite-Element-Methods → Solve sparse linear system of equations
  - Quantum physics/chemistry → Determine eigenvalues of sparse matrices
- **Fraction of total time spent in spMVM is often approx. 85% – 99.99%**
- **"Sparse" matrix ≅ $N_{nz}$ grows slower than quadratically with $N_r$**
  - $N_{nz}$ = total number on nonzeros
  - $N_{nzr}$ = avg. # nonzeros per row (=$N_{nz}$ / $N_r$)
- **Store nonzero elements only:**
  - Reduce memory overhead
  - Avoid computing zeros

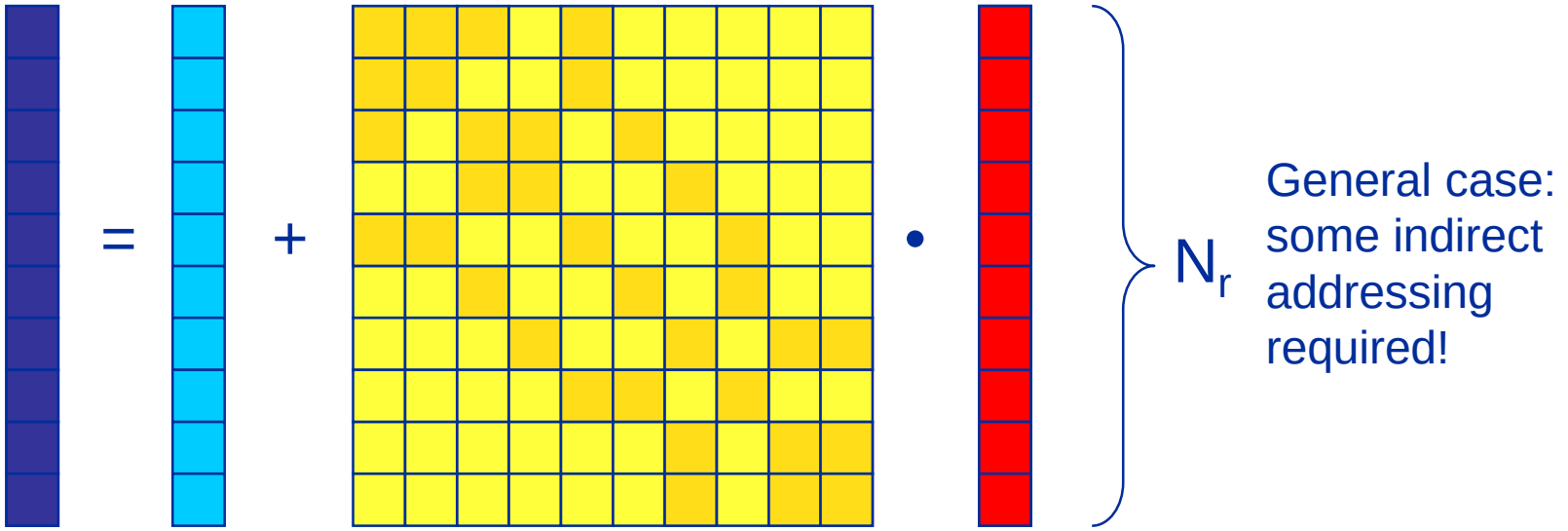


# Sparse Matrix Vector Multiplication (spMVM)

Intensity-type O(N)/O(N) → memory bound

Nevertheless, there is more than one loop here!

```
c =   c  +    sparse_mat    *    b
```




General case: some indirect addressing required!

Sparsity pattern ←→ indirect access to right hand side vector (RHS ←→ vector **b** in our case)

# Sparse matrix-vector multiply:
*Different matrix storage schemes*

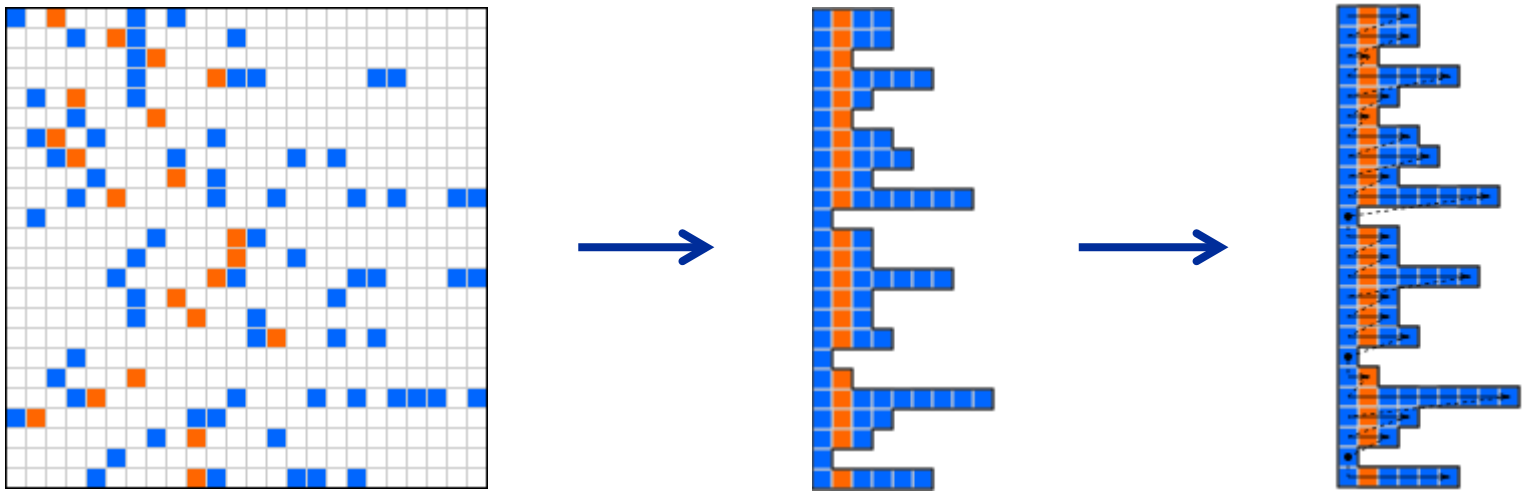
- **Choice of sparse matrix storage scheme is crucial for performance**
  - Different schemes yield entirely different performance characteristics
  - Depends on architecture!

- **Most important formats:**
  - **CRS (Compressed Row Storage)** → **CPUs**
  - (Sliced) ELLPACK → GPGPUs
- **Other/newer possibilities:**
  - CCS (Compressed Column Storage, “Harwell-Boeing”)
  - CDS (Compressed Diagonal Storage)
  - **SELL-C-sigma** (Ours – perfect fit for SIMD & GPGPUs)

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 **36**(5), C401–C423 (2014). DOI: 10.1137/130930352

- **Depending on the storage scheme, the memory access patterns differ vastly between the formats**
  - Choose the storage scheme that best fits your needs

# CRS (Compressed Row Storage) – data format



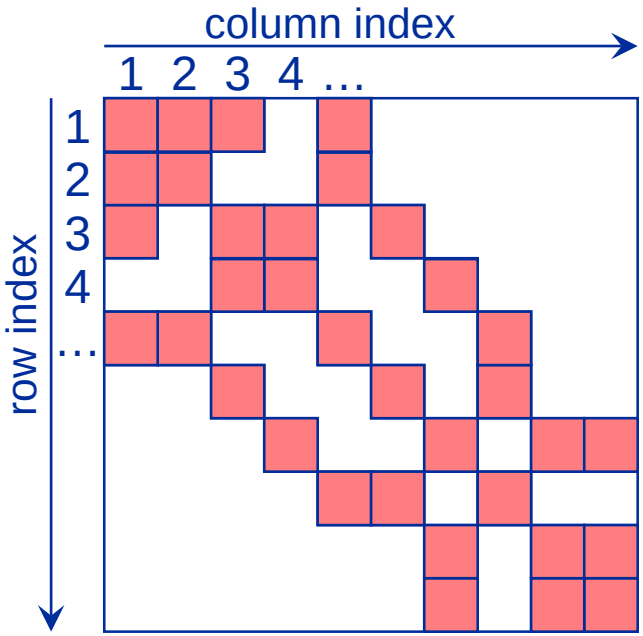
## Format creation

1. Store values and column indices of all non-zero elements **row-wise**
2. Store starting indices of each column (`rpt`)

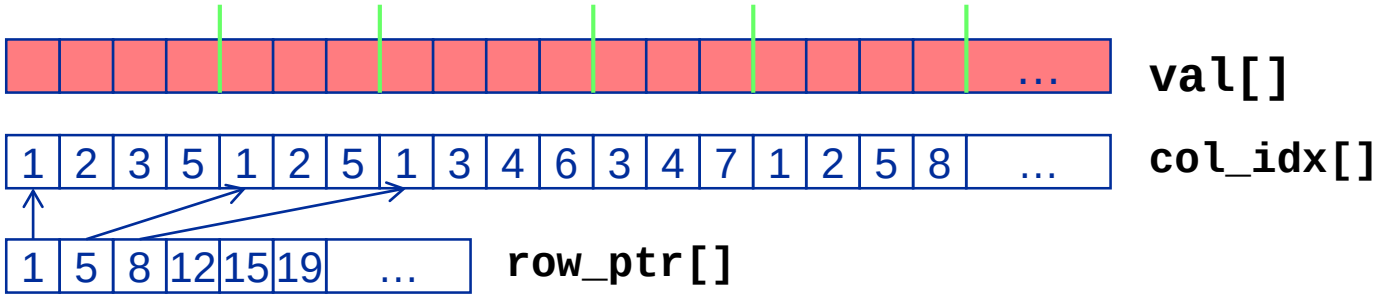
## Data fields

```
double val[]
unsigned int col[]
unsigned int rpt[]
```

# CRS matrix storage scheme for `sparse_mat`




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



# CRS Example

$$A = \begin{pmatrix} \underset{(0,0)}{10} & 0 & 0 & \underset{(0,3)}{12} & 0 \\ 0 & 0 & \underset{(1,2)}{11} & 0 & \underset{(1,4)}{13} \\ 0 & \underset{(2,1)}{16} & 0 & 0 & 0 \\ 0 & 0 & \underset{(3,2)}{11} & 0 & \underset{(3,4)}{13} \end{pmatrix}$$

$$\begin{aligned} val &= \begin{pmatrix} \underset{(0,0)}{10} & \underset{(0,3)}{12} & \underset{(1,2)}{11} & \underset{(1,4)}{13} & \underset{(2,1)}{16} & \underset{(3,2)}{11} & \underset{(3,4)}{13} \end{pmatrix} \\ colInd &= \begin{pmatrix} \underset{(0}{0} & \underset{)}{3} & \underset{(1}{2} & \underset{)}{4} & \underset{(2)}{1} & \underset{(3}{2} & \underset{)}{4} \end{pmatrix} \\ rowPtr &= \begin{pmatrix} \underset{(0)}{0} & \underset{(1)}{2} & \underset{(2)}{4} & \underset{(3)}{5} & \underset{(4)}{7} \end{pmatrix} \end{aligned}$$

- **Only the necessary (= non-zero) values along with their column indices are stored**

# CRS sparse MVM

- **Implement `c = c + sparse_mat * b`**
- **Only the nonzero elements of the matrix are stored/used**
  - Operation count = 2 * $N_{nz}$ [FLOP]
  - Data set size for matrix storage: (8 + 4) * $N_{nz}$ [B]

```
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
```

- Features
  - **Long outer loop** ($N_r$) – matrix dimension: $N_r$ ~ $10^4$,…, $10^{10}$
  - Probably **short inner loop (~ $N_{nzr}$):** $N_{nzr} = N_{nz} / N_r$ ~ 10s,…,100s
    (nonzero entries in each respective row)
  - Indexed (indirect) access to RHS vector **`b[]`**
  - All other data is accessed contiguously!

# CRS sparse MVM – OpenMP parallelization

```
!$OMP PARALLEL DO SCHEDULE(runtime)
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
```

- OMP_SCHEDULE = STATIC
  - OK for “balanced” matrix structures
  - Otherwise potential load imbalance
- Other schedules may achieve better performance

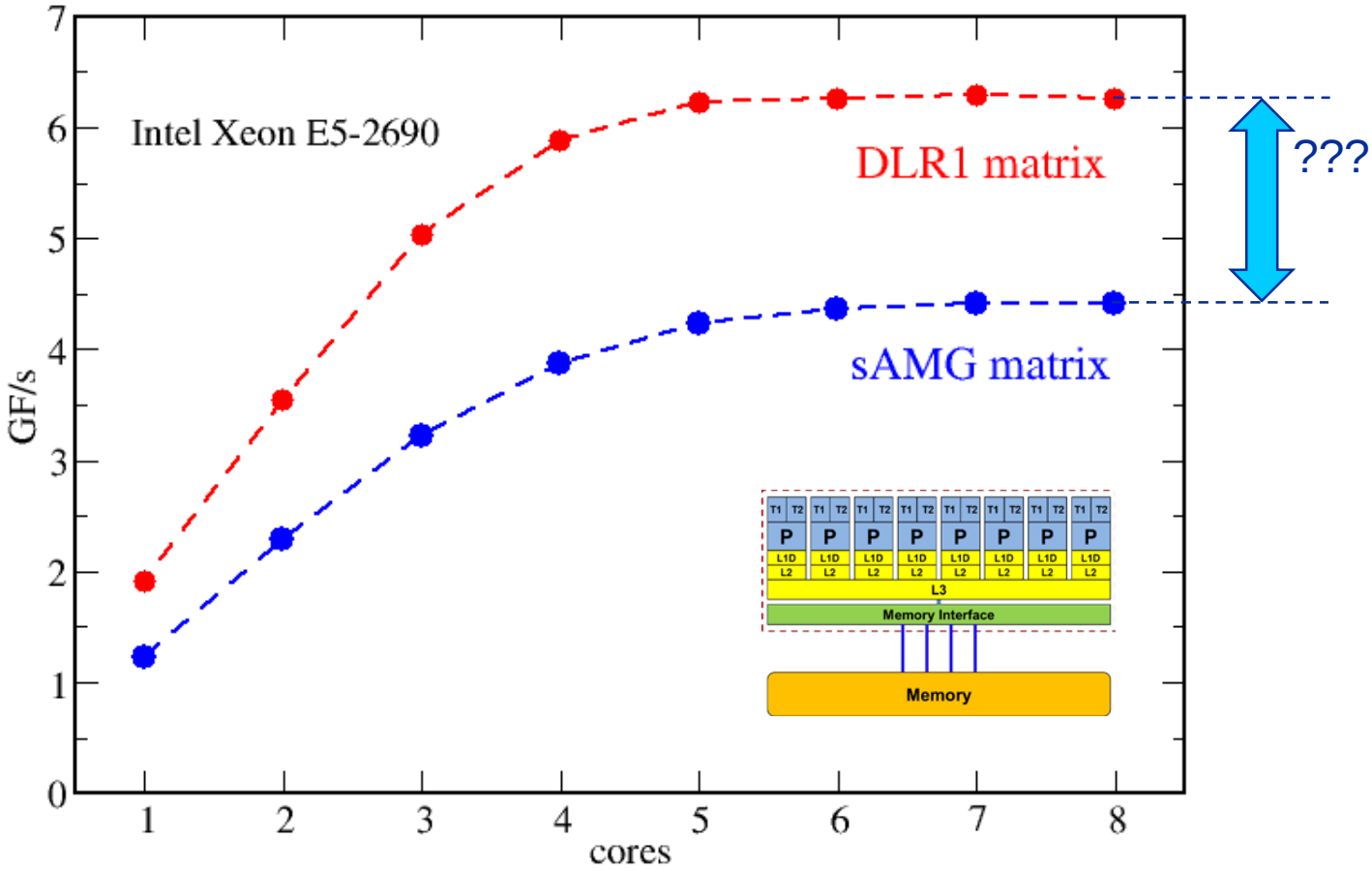
# Performance characteristics

- 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 spMVM?

- Optimization?



# Example: SpMVM node performance model

- **Sparse MVM in double precision w/ CRS data storage:**

```
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
```

- **DP CRS comp. intensity**

$$I_{CRS}^{DP} = \frac{2}{8 + 4 + 8\alpha + 16/N_{nzr}} \frac{\text{flops}}{\text{byte}}$$

  - $\alpha$ quantifies traffic for loading RHS (depends on matrix!!!!)
    - $\alpha = 0$ → RHS is in cache
    - $\alpha = 1/N_{nzr}$ → RHS loaded once
    - $\alpha = 1$ → no cache
    - $\alpha > 1$ → Houston, we have a problem!
  - "Expected" performance = $b_S$ x $I_{CRS}$
  - Determine $\alpha$ by measuring performance and actual memory traffic
    - Maximum memory BW may not be achieved with spMVM

# Determine RHS traffic

$$I_{CRS}^{DP} = \frac{2}{8 + 4 + 8\alpha + 16/N_{nzr}} \frac{\text{flops}}{\text{byte}} = \frac{N_{nz} \cdot 2 \text{ flops}}{V_{meas}}$$

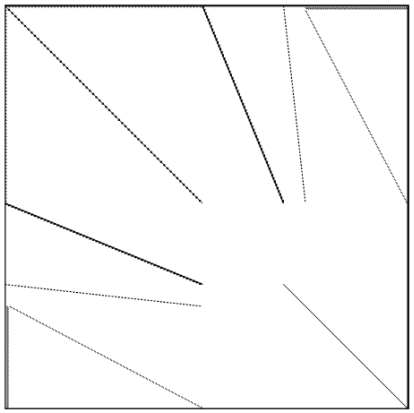
- **$V_{meas}$ is the measured overall memory data traffic (using, e.g., likwid-perfctr)**
- **Solve for $\alpha$:**

$$\alpha = \frac{1}{4}\left(\frac{V_{meas}}{N_{nz} \cdot 2 \text{ bytes}} - 6 - \frac{8}{N_{nzr}}\right)$$

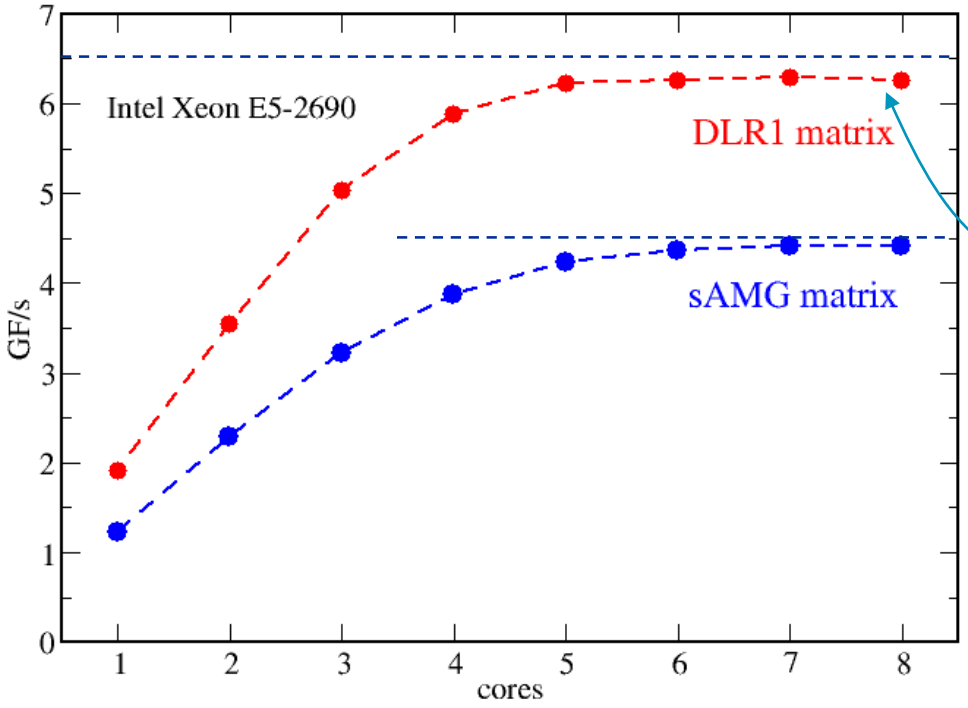
- **Example: kkt_power matrix from the UoF collection on one Intel SNB socket**
  - $N_{nz} = 14.6 \cdot 10^6$, $N_{nzr} = 7.1$
  - $V_{meas} \approx 258$ MB
  - → $\alpha = 0.43$, $\alpha N_{nzr} = 3.1$
  - → RHS is loaded 3.1 times from memory
  - and:

$$\frac{I_{CRS}^{DP}(1/N_{nzr})}{I_{CRS}^{DP}(\alpha)} = 1.15$$

**15% extra traffic → optimization potential!**



# Now back to the start…




- $b_S = 39\,\mathrm{GB/s}$
- $B_c^{min} = 6\,\mathrm{B/F}$
- Maximum spMVM performance:

$$P_{max} = 6.5\,\mathrm{GF/s}$$

- → DLR1 causes minimum code balance!

- sAMG matrix code balance:

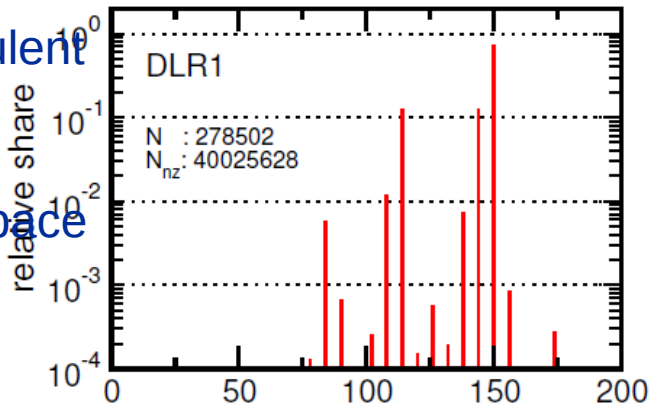
$$B_c \leq \frac{b_S}{4.5\,\mathrm{GF/s}} = 8.7\,\mathrm{B/F}$$

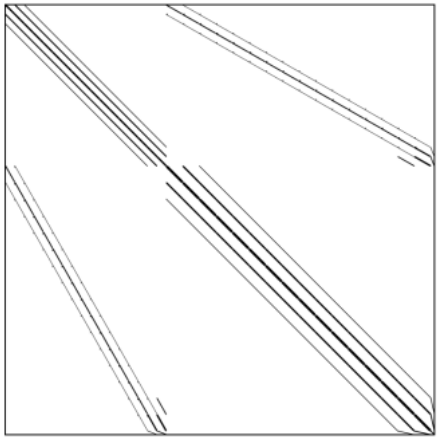
- Why is this only an upper limit?
- What is the next step?
- Could we have predicted this qualitative difference?

# Sparse matrix testcases

## “DLR1” (A. Basermann, DLR)

Adjoint problem computation (turbulent transonic flow
over a wing) with the TAU
CFD system of the German Aerospace Center (DLR)
Avg. non-zeros/row ~150

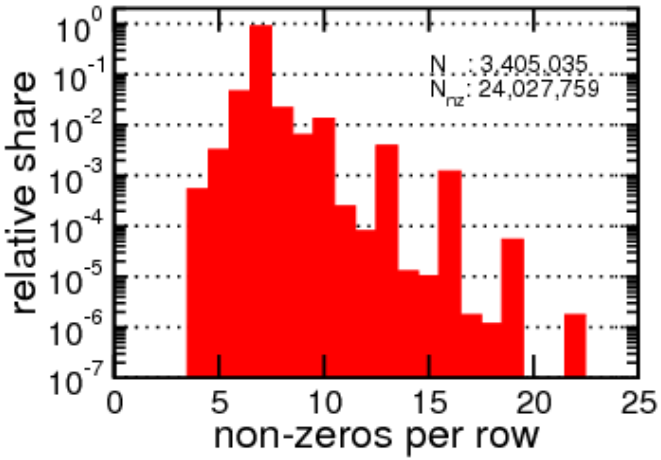


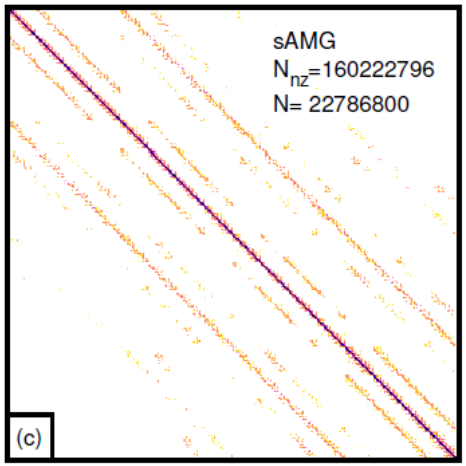



## “sAMG” (K. Stüben, FhG-SCAI)

Matrix from FhG’s adaptive multigrid code sAMG
for the irregular discretization of a Poisson problem on a car geometry.
Avg. non-zeros/row: $N_{nzr}$~ 7
Assume $\alpha = \frac{1}{N_{nzr}} = \frac{1}{7}$

$$I_{CRS}^{DP} = \frac{2}{12 + 8/7 + 16/7}\frac{\text{flops}}{\text{byte}} = \frac{1\ F}{8\ B}$$



# Roofline analysis for spMVM

- **Conclusion from Roofline analysis**
  - The roofline model does not work 100% 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

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