Performance Engineering for Sparse Linear Solvers

Half-Day Tutorial at ISC High Performance 2026

Jonas Thies, TU Delft

Georg Hager, Erlangen National High Performance Computing Center

Hartwig Anzt, TU München

https://blogs.fau.de/hager/tutorials/isc26-pels

Tutorial Agenda

- Brief introduction to node-level computer architecture
- Performance modeling with the Roofline model
- Sparse matrix-vector multiplication (SpMV) performance, sparse-matrix data formats, and Roofline modeling of SpMV
- The Conjugate Gradient (CG) algorithm
- Preconditioning and preconditioned CG (PCG)

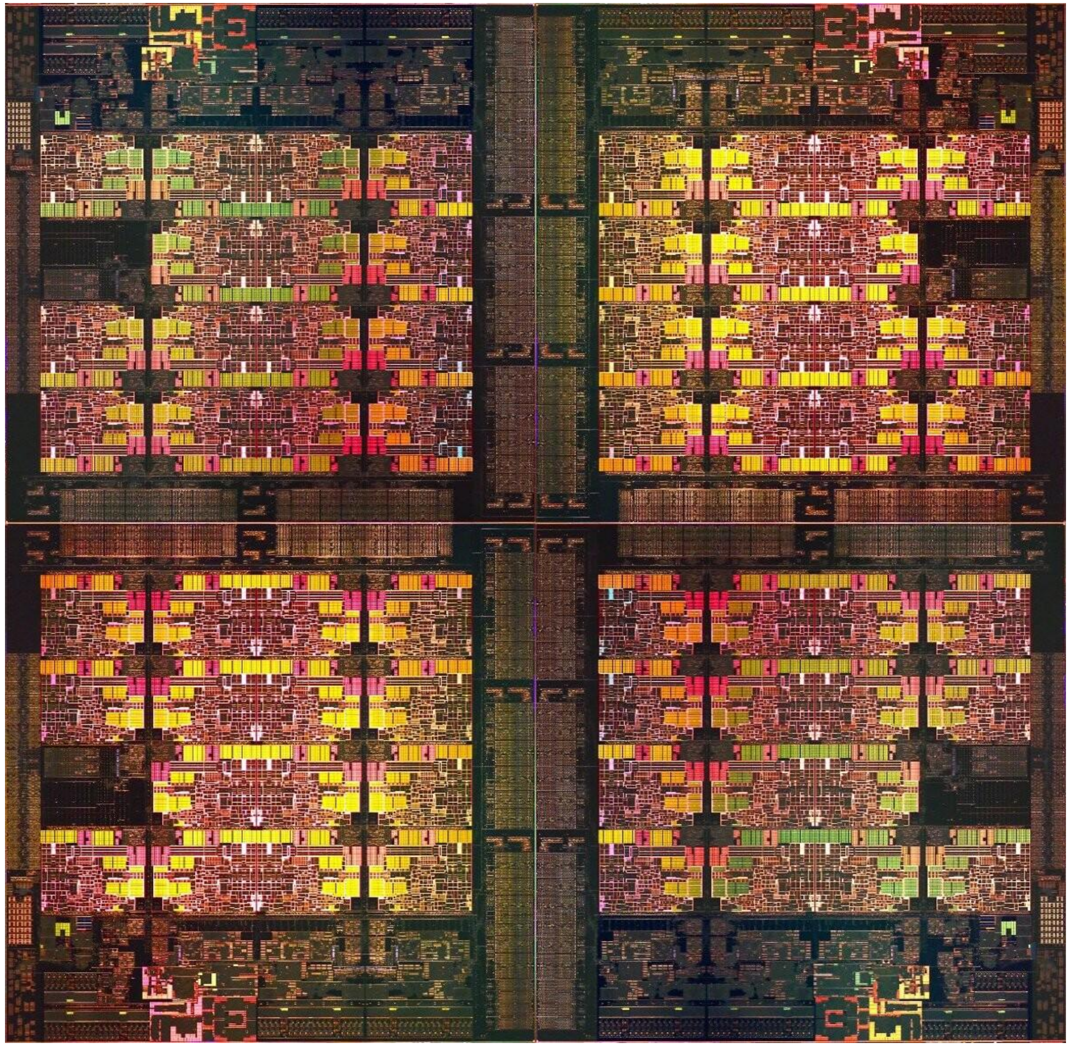
HPC Node Architecture

CPUs

GPUs

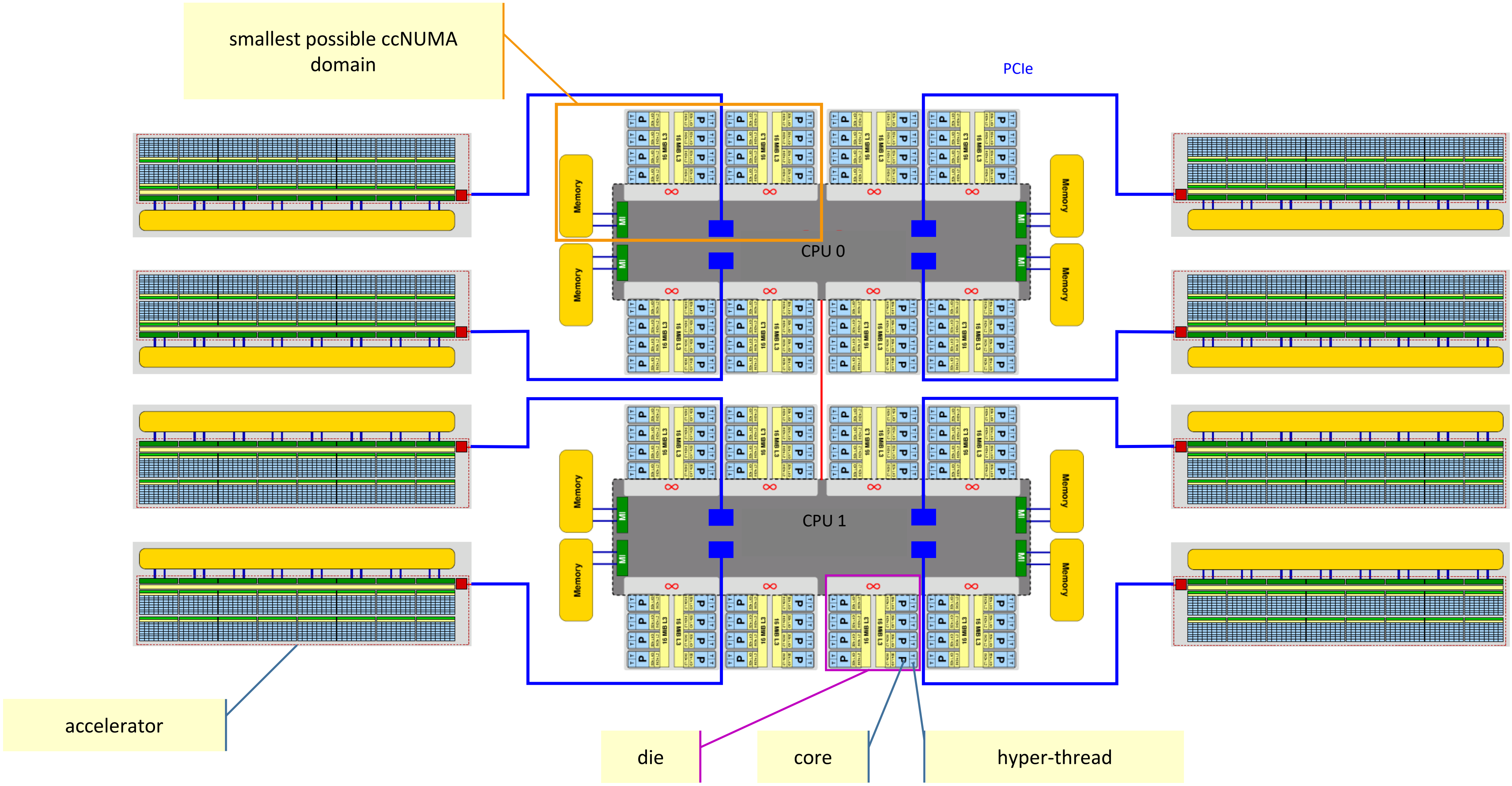
Multi-core today: Intel Xeon Sapphire Rapids (2023)

- Xeon "Sapphire Rapids" (Platinum/Gold/Silver/Bronze): Up to 60 cores running at 1.7+ GHz (+ "Turbo Mode" 4.8 GHz),
- "Intel 7" process / up to 350 W
- Multi-die package (4 chips)
- Clock frequency: flexible ☺



https://www.techpowerup.com/292204/intel-sapphire-rapids-xeon-4-tile-mcm-annotated

A GPU-accelerated compute node



A modern GPU: Nvidia H100 “Hopper” SXM5

Architecture

- 80 B Transistors
- ~ 1.8 GHz clock speed
- ~ 144 “SM” units
  - 128 SP “cores” each (FMA)
  - 64 DP “cores” each (FMA)
  - 4 “Tensor Cores” each
  - 2:1 SP:DP performance

- ~ 34 TFlop/s DP peak (FP64)
- 50 MiB L2 Cache

- 80 GB HBM3
- MemBW ~ 3300 GB/s (theoretical)
- MemBW ~ 3000 GB/s (measured)

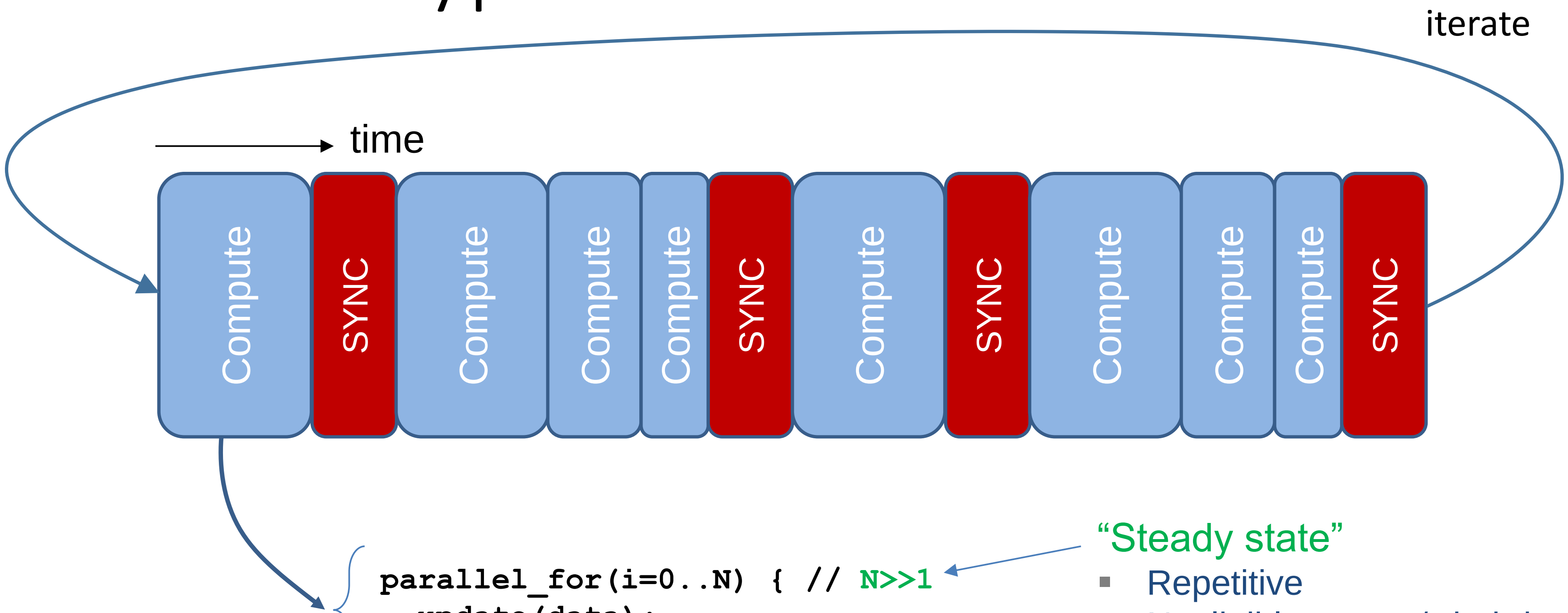
The Roofline Model

Bottleneck-based thinking

Simple models for single loops

Multiple loops

Structure of typical solver code



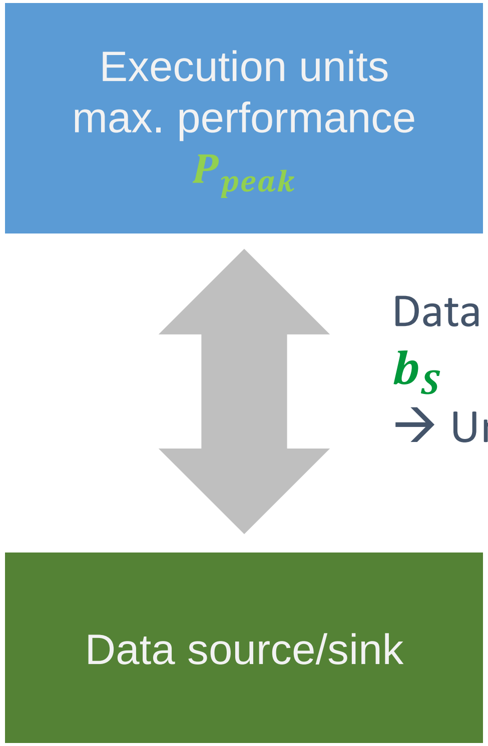

```
parallel_for(i=0..N) { // N>>1
  update(data);
}
```

"Steady state"
- Repetitive
- Negligible startup/wind-down overhead

Runtime model: $T = f(\texttt{\$STUFF}, \texttt{\$HARDWARE})$

A simple two-bottleneck model of loop code execution

Simplistic view of the hardware:



Data path, bandwidth
b_S
→ Unit: byte/s

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
```

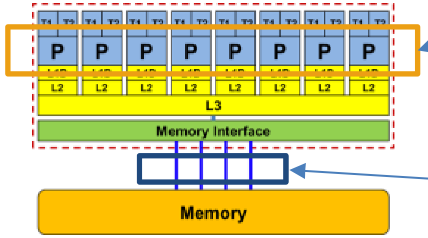
Computational intensity $I = \frac{N}{V}$
→ Unit: flop/byte

Which takes longer?
- Data transfer
- Work execution

Predicting the (minimum) runtime of a loop

Two bottlenecks:

```
#pragma omp parallel for
for(i=0; i<10^7; ++i)
  a[i] = a[i] + s * c[i];
```



8-core CPU
(3 GHz Intel Sandy Bridge)

$R^{max}_{flops} = 192 \frac{\text{Gflops}}{\text{s}}$

$R^{max}_{BW} = 40 \frac{\text{Gbyte}}{\text{s}}$

Resources rates provided (machine properties)

Resources needed (code properties)

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

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

Full-overlap assumption:

$$T_{flops} = \frac{2 \times 10^7 \text{ flops}}{192 \frac{\text{Gflops}}{\text{s}}} = 104\ \mu s \qquad T_{BW} = \frac{2.4 \times 10^8 \text{ bytes}}{40 \frac{\text{Gbyte}}{\text{s}}} = 6.0 \text{ ms}$$

$$T_{\min} = \max(T_{flops}, T_{BW}) = 6 \text{ ms}$$

From time to performance

$$P_{upper} = \frac{W_{flops}}{\max(T_{flops}, T_{BW})} = \frac{W_{flops}}{\max\left(\frac{W_{flops}}{R_{flops}}, \frac{W_{BW}}{R_{BW}}\right)} =$$

$$\min\left(R_{flops}, R_{BW} \times \frac{W_{flops}}{W_{BW}}\right)$$

Machine model: Peak performance [flop/s]

Machine model: Memory bandwidth [byte/s]

Application model: Computational intensity [flop/byte]

R.W. Hockney and I.J. Curington:
$f_{1/2}$: A parameter to characterize memory and communication bottlenecks.
Parallel Computing 10, 277-286 (1989). DOI: 10.1016/0167-8191(89)90100-2

S. Williams: Auto-tuning Performance on Multicore Computers.
UCB Technical Report No. UCB/EECS-2008-164. PhD thesis (2008)

"Roofline"!?

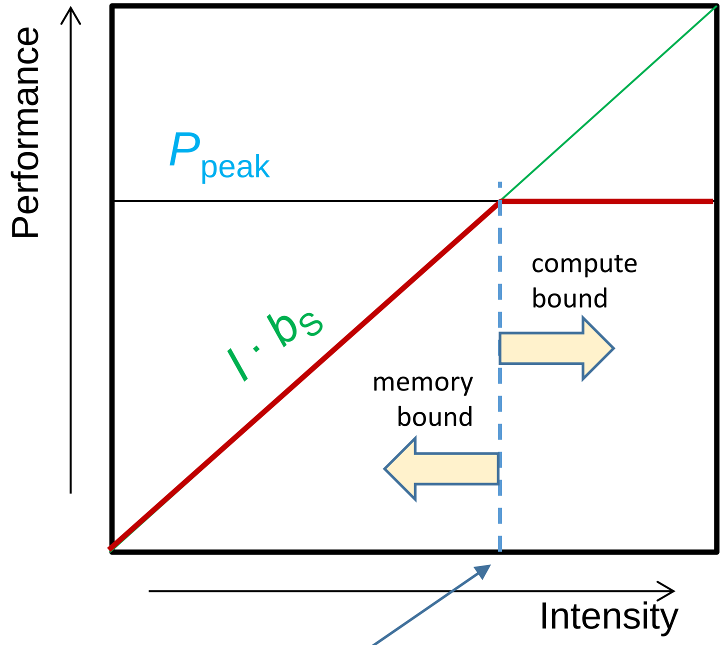
Common nomenclature:

R_{flops} → P_{peak} peak performance

R_{BW} → b_S memory bandwidth

$\frac{W_{flops}}{W_{BW}}$ → I computational intensity

$$P_{upper} = \min(P_{peak}, I \times b_S)$$



Threshold:
≈ 10-15 F/B for current
Server CPUs/GPUs

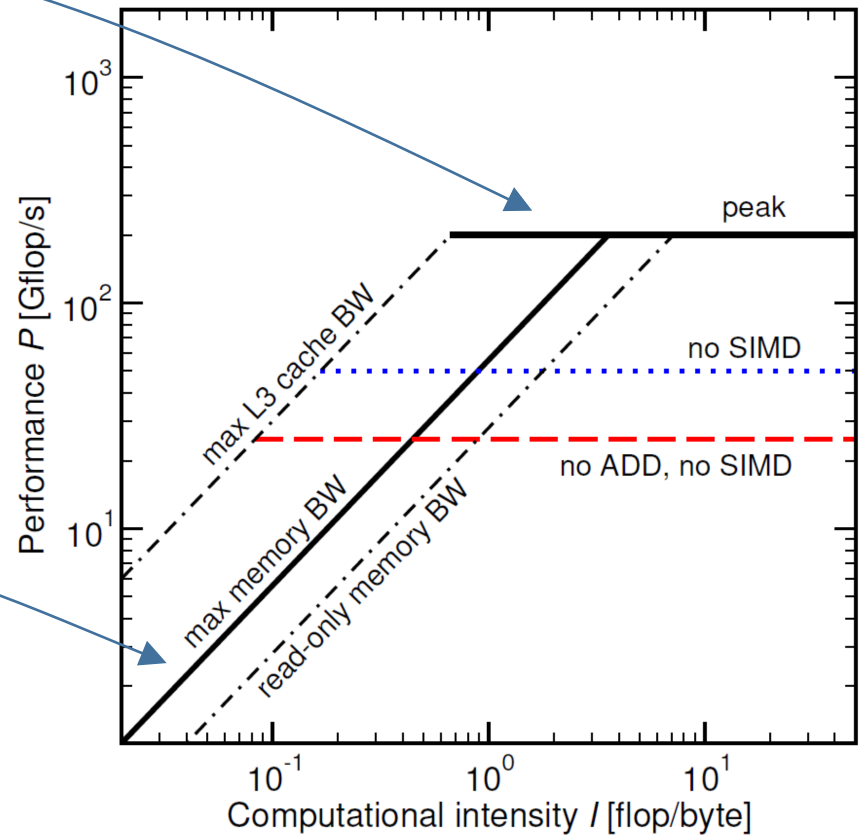
Multiple bottlenecks?

Ceilings (flat)

- “Execution level” bottlenecks
- “Work” related
- Independent of intensity

Roofs (sloped)

- Data transfer bottlenecks
- “Traffic” related
- Linear in intensity



$$P_{upper} = \min_{i,j}\left(\{P_{\max,i}\}, \{I_j \cdot b_j\}\right)$$

Hands-On:

Running simple kernels and measuring memory bandwidth

Two kinds of modeling

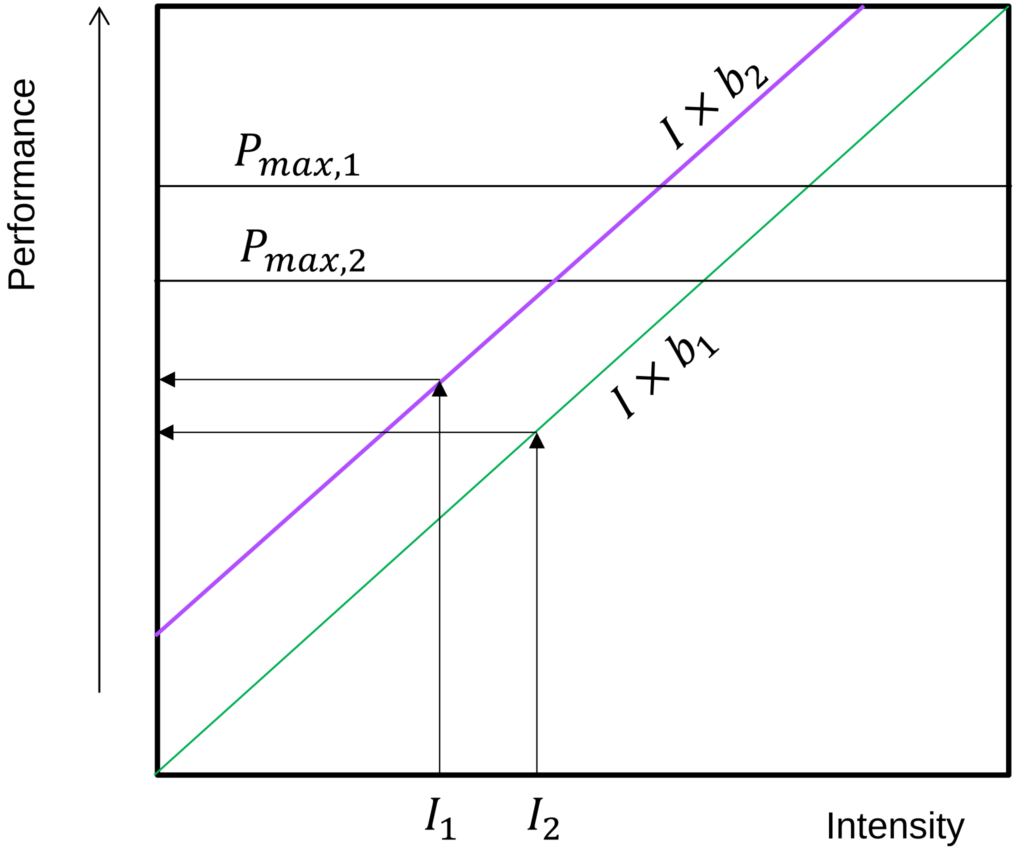
Predictive

- Determine machine b_j
- Calculate I_j, $P_{\max,i}$
- Use $P_{upper} = \min_{i,j}\left(\{P_{\max,i}\}, \{I_j \cdot b_j\}\right)$
- Compare prediction(s) with measurement(s)
- Optimize, iterate

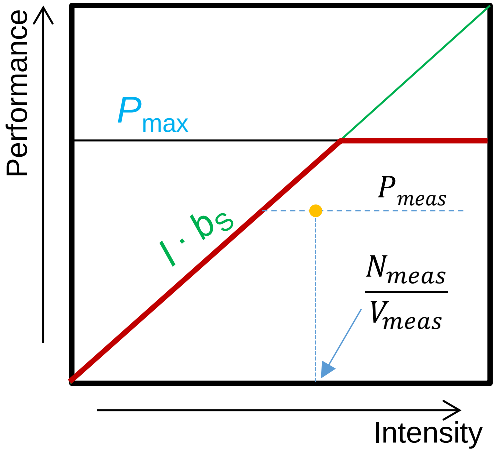
Diagnostic/phenomenological

- Determine machine b_S, $P_{max,i}$
- Measure W_i (performance tools)
- Measure performance P
- Compare with applicable roof/ceiling
- Optimize, iterate

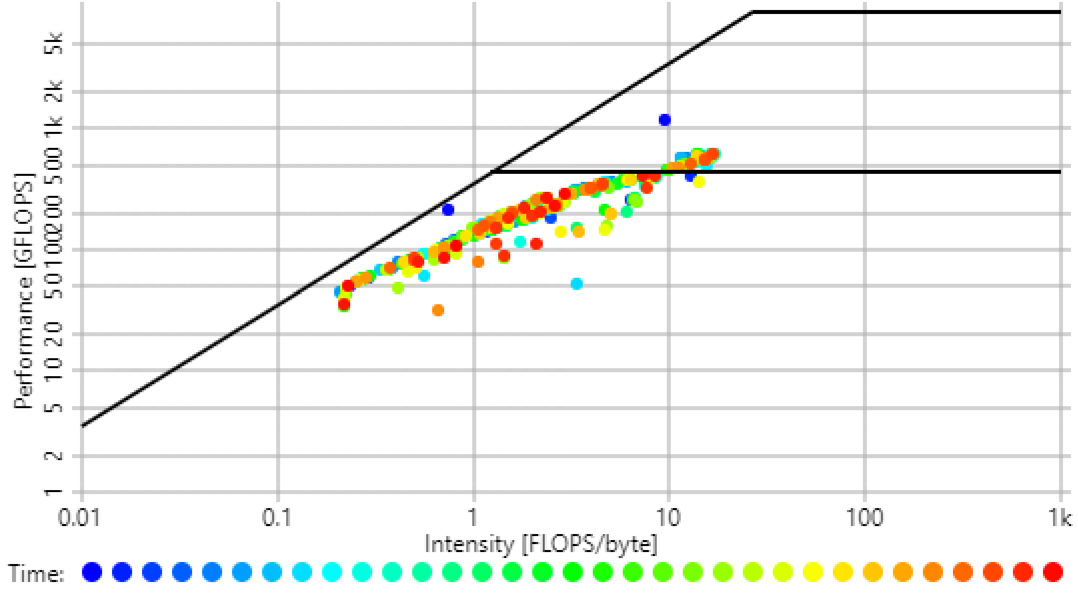
Predictive modeling

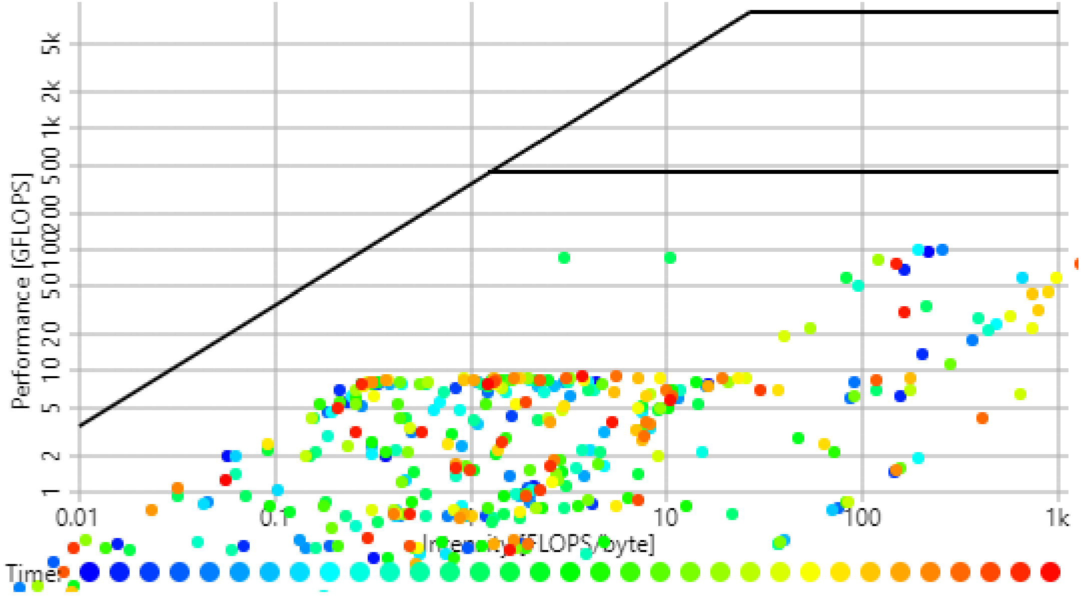


Diagnostic modeling



Two cluster jobs…





Roofline: Simple Examples

Dense linear algebra

Sparse linear algebra

Simple solvers: CG

Hartwig Anzt, hartwig.anzt@kit.edu

Dense linear algebra

```
for(i=0; i<N; ++i)
    a[i] = a[i]+s*x[i];
```

daxpy (BLAS-1)

```
for(k=0; k<NK;++k)
  for(l=0; l<NL; ++l)
    for(m=0; m<NM; ++m)
      y[k*NL+l] +=
           A[k*NM+m]*B[l*NM+m];
```

dense MMM (BLAS-3)

```
for(i=0; i<N; ++i)
  s += a[i]*b[i];
```

dot product (BLAS-1)

```
for(r=0; r<NR; ++r)
  for(c=0; c<NC; ++c)
    y[r] += A[r*NC+c]*x[c];
```

dense MVM (BLAS-2)
dot-product style

Roofline thinking:

What is the computational intensity?

Dense MVM

```
for(r=0; r<NR; ++r)
  for(c=0; c<NC; ++c)
    y[r] += A[r*NC+c]*x[c];
```

- One DP read from memory for each matrix entry
- x[ ] and y[ ] are read and updated from cache after 1st read
- → 8 byte and 2 flops per iteration

Computational intensity $I = \frac{2\ flop}{8\ byte} = 0.25\frac{flop}{byte}$

Dense MMM?

```
for(k=0; k<NK;++k)
  for(l=0; l<NL; ++l)
    for(m=0; m<NM; ++m)
      y[k*NL+l] +=
              A[k*NM+m]*B[l*NM+m];
```

- Blocking/unrolling techniques can increase intensity beyond the Roofline knee

```
for(k=0; k<NK; k+=2)
  for(l=0; l<NL; l+=2)
    for(m=0; m<NM; ++m)
      y[k*NL+l]         += A[k*NM+m]*B[l*NM+m];
      y[(k+1)*NL+l]     += A[(k+1)*NM+m]*B[l*NM+m];
      y[k*NL+(l+1)]     += A[k*NM+m]*B[(l+1)*NM+m];
      y[(k+1)*NL+(l+1)] += A[(k+1)*NM+m]*B[(l+1)*NM+m];
```

→ peak performance achievable

Sparse Matrices and SpMV

Sparse Matrix Formats

Sparse Matrix Vector Products and Parallelization

Matrix Vector Multiplication

Input A, x, y Output $y = A \cdot x$

- Central building block in many complex algorithms:
  - Orthogonalization, power iteration in Page Rank, Power flow of a system …

Sparse Matrix Storage

- Matrix A contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).

Sparse Matrix Storage

- Matrix A contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).
- Idea: Store only nonzero elements [nz] explicitly.

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

$$\text{value} = [\ 5.4 \quad 1.1 \quad 2.2 \quad 8.3 \quad 3.7 \quad 1.3 \quad 3.8 \quad 4.2 \quad 5.4 \quad 9.2 \quad 1.1 \quad 8.1\ ]$$ Value

COO format

- Matrix A contains only few nonzero elements.
- Storing all entries results in large overhead (memory & computation).
- Idea: Store only nonzero elements [nz] explicitly.

Need to also store location of nonzero elements!

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Memory footprint of COO format:
#nz * sizeof(val) + 2*#nz * sizeof(int)

$$\text{value} = [\; 5.4 \;\; 1.1 \;\; 2.2 \;\; 8.3 \;\; 3.7 \;\; 1.3 \;\; 3.8 \;\; 4.2 \;\; 5.4 \;\; 9.2 \;\; 1.1 \;\; 8.1 \;]$$ Value

$$\text{colidx} = [\; 0 \;\; 1 \;\; 0 \;\; 1 \;\; 3 \;\; 4 \;\; 5 \;\; 2 \;\; 0 \;\; 3 \;\; 4 \;\; 5 \;]$$ Column-index

$$\text{rowidx} = [\; 0 \;\; 0 \;\; 1 \;\; 1 \;\; 1 \;\; 1 \;\; 1 \;\; 2 \;\; 3 \;\; 3 \;\; 4 \;\; 5 \;]$$ Row-index

CSR (==CRS) format

- If storing the nonzero entries row-by-row, the Row-index array has sequences of identical entries

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

$$\text{value} = [\ 5.4 \quad 1.1 \quad 2.2 \quad 8.3 \quad 3.7 \quad 1.3 \quad 3.8 \quad 4.2 \quad 5.4 \quad 9.2 \quad 1.1 \quad 8.1\ ]$$ Value

$$\text{colidx} = [\ 0 \quad 1 \quad 0 \quad 1 \quad 3 \quad 4 \quad 5 \quad 2 \quad 0 \quad 3 \quad 4 \quad 5\ ]$$ Column-index

$$\text{rowidx} = [\ 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 2 \quad 3 \quad 3 \quad 4 \quad 5\ ]$$ Row-index

CSR (==CRS) format

- If storing the nonzero entries row-by-row, the Row-index array has sequences of identical entries
- A row pointer pointing to the first element in each row makes the row index obsolete

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

$$\text{value} = [\ 5.4\ \ 1.1\ \ 2.2\ \ 8.3\ \ 3.7\ \ 1.3\ \ 3.8\ \ 4.2\ \ 5.4\ \ 9.2\ \ 1.1\ \ 8.1\ ]$$ Value

$$\text{colidx} = [\ 0\ \ 1\ \ 0\ \ 1\ \ 3\ \ 4\ \ 5\ \ 2\ \ 0\ \ 3\ \ 4\ \ 5\ ]$$ Column-index

$$\text{rowidx} = [\ 0\ \ 0\ \ 1\ \ 1\ \ 1\ \ 1\ \ 1\ \ 2\ \ 3\ \ 3\ \ 4\ \ 5\ ]$$ Row-index

$$\text{rowptr} = [\ 0\ \ 2\ \ 7\ \ 8\ \ 10\ \ 11\ \ 12\ ]$$ Row-pointer points to the first element in each row

Number of nonzero elements

CSR (==CRS) format

- If storing the nonzero entries row-by-row, the Row-index array has sequences of identical entries
- A row pointer pointing to the first element in each row makes the row index obsolete

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

$$\text{value} = [\; 5.4 \quad 1.1 \quad 2.2 \quad 8.3 \quad 3.7 \quad 1.3 \quad 3.8 \quad 4.2 \quad 5.4 \quad 9.2 \quad 1.1 \quad 8.1 \;]$$ Value

$$\text{colidx} = [\; 0 \quad 1 \quad 0 \quad 1 \quad 3 \quad 4 \quad 5 \quad 2 \quad 0 \quad 3 \quad 4 \quad 5 \;]$$ Column-index

$$\text{rowidx} = [\; 0 \quad 0 \quad 1 \quad 1 \quad 1 \quad 1 \quad 1 \quad 2 \quad 3 \quad 3 \quad 4 \quad 5 \;]$$ Row-index

$$\text{rowptr} = [\; 0 \quad 2 \quad 7 \quad 8 \quad 10 \quad 11 \quad 12 \;]$$ Row-pointer points to the first element in each row

Number of nonzero elements

CSR (==CRS) format

- If storing the nonzero entries row-by-row, the Row-index array has sequences of identical entries
- A row pointer pointing to the first element in each row makes the row index obsolete

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Memory footprint of CSR format:
#nz*sizeof(val) + #nz*sizeof(int)
+ (n+1)*sizeof(int)

$$\text{value} = [\ 5.4 \quad 1.1 \quad 2.2 \quad 8.3 \quad 3.7 \quad 1.3 \quad 3.8 \quad 4.2 \quad 5.4 \quad 9.2 \quad 1.1 \quad 8.1\ ]$$ Value

$$\text{colidx} = [\ 0 \quad 1 \quad 0 \quad 1 \quad 3 \quad 4 \quad 5 \quad 2 \quad 0 \quad 3 \quad 4 \quad 5\ ]$$ Column-index

$$\text{rowptr} = [\ 0 \quad 2 \quad 7 \quad 8 \quad 10 \quad 11 \quad 12\ ]$$ Row-pointer points to the first element in each row

Number of nonzero elements

CSR (==CRS) format

- If storing the nonzero entries row-by-row, the Row-index array has sequences of identical entries
- A row pointer pointing to the first element in each row makes the row index obsolete

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Memory footprint of CSR format:
#nz*sizeof(val) + #nz*sizeof(int)
+ (n+1)*sizeof(int)

$$\text{value} = [\ 5.4\ \ 1.1\ \ 2.2\ \ 8.3\ \ 3.7\ \ 1.3\ \ 3.8\ \ 4.2\ \ 5.4\ \ 9.2\ \ 1.1\ \ 8.1\ ]$$ Value

$$\text{colidx} = [\ 0\ \ 1\ \ 0\ \ 1\ \ 3\ \ 4\ \ 5\ \ 2\ \ 0\ \ 3\ \ 4\ \ 5\ ]$$ Column-index

$$\text{rowptr} = [\ 0\ \ 2\ \ 7\ \ 8\ \ 10\ \ 11\ \ 12\ ]$$ Row-pointer points to the first element in each row

Number of nonzero elements

Problem: no coalesced memory access to first column (matrix is stored sparse row-major)

Column-major sparse matrix storage

- We want to store a sparse matrix column-major for coalesced access
- For this, all rows need the same number of “nonzero” elements
- It is easy to get the row index, the column index has to be stored for every element

$$\begin{bmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 3.7 & 1.3 & 3.8 & 0 \\ 4.2 & 0 & 0 & 0 & 0 & 0 \\ 5.4 & 9.2 & 0 & 0 & 0 & 0 \\ 1.1 & 0 & 0 & 0 & 0 & 0 \\ 8.1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

ELL Format

`Left-align nonzero elements'

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

$$\begin{bmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 3.7 & 1.3 & 3.8 & 0 \\ 4.2 & 0 & 0 & 0 & 0 & 0 \\ 5.4 & 9.2 & 0 & 0 & 0 & 0 \\ 1.1 & 0 & 0 & 0 & 0 & 0 \\ 8.1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & - & - & - & - \\ 0 & 1 & 3 & 4 & 5 & - \\ 2 & - & - & - & - & - \\ 0 & 3 & - & - & - & - \\ 4 & - & - & - & - & - \\ 5 & - & - & - & - & - \end{bmatrix}$$

ELL Format

`Left-align nonzero elements'

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

$$\begin{bmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 3.7 & 1.3 & 3.8 & 0 \\ 4.2 & 0 & 0 & 0 & 0 & 0 \\ 5.4 & 9.2 & 0 & 0 & 0 & 0 \\ 1.1 & 0 & 0 & 0 & 0 & 0 \\ 8.1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & - & - & - & - \\ 0 & 1 & 3 & 4 & 5 & - \\ 2 & - & - & - & - & - \\ 0 & 3 & - & - & - & - \\ 4 & - & - & - & - & - \\ 5 & - & - & - & - & - \end{bmatrix}$$

Pad rows to uniform length

Memory volume:

*values: max_nnz_row * num_rows*

*col-index: max_nnz_row * num_rows*

no row pointer

ELL Format

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

$$\begin{bmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 3.7 & 1.3 & 3.8 & 0 \\ 4.2 & 0 & 0 & 0 & 0 & 0 \\ 5.4 & 9.2 & 0 & 0 & 0 & 0 \\ 1.1 & 0 & 0 & 0 & 0 & 0 \\ 8.1 & 0 & 0 & 0 & 0 & 0 \end{bmatrix} \qquad \begin{bmatrix} 0 & 1 & - & - & - & - \\ 0 & 1 & 3 & 4 & 5 & - \\ 2 & - & - & - & - & - \\ 0 & 3 & - & - & - & - \\ 4 & - & - & - & - & - \\ 5 & - & - & - & - & - \end{bmatrix}$$

Pad rows to uniform length

Memory volume:

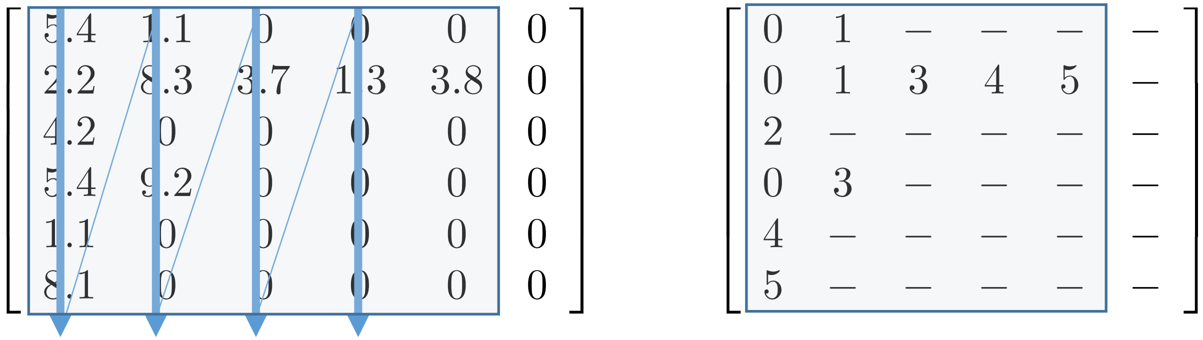
*values: max_nnz_row * num_rows*

*col-index: max_nnz_row * num_rows*

no row pointer

ELL Format

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$



Pad rows to uniform length

Memory volume:

*values: max_nnz_row * num_rows*

*col-index: max_nnz_row * num_rows*

no row pointer

Problem: Can incur significant overhead!

Overhead dependent on “longest” row and number of columns.

Sliced-ELL Format

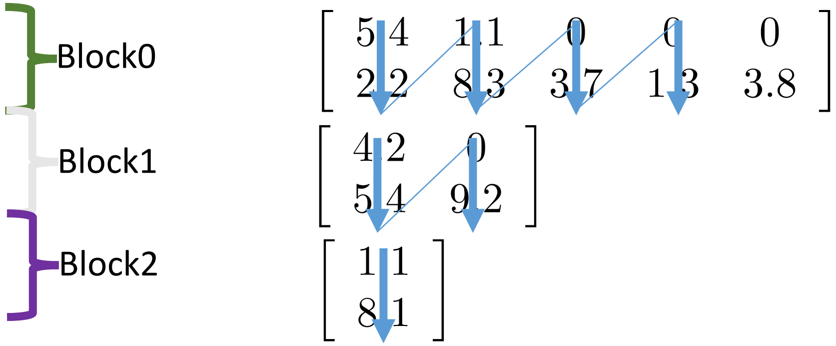
- Partition the matrix into row-blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL
  - Can still store col-major

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

Sliced-ELL Format

- Partition the matrix into row-blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL
  - Can still store col-major

$$\begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$



Sliced-ELL Format

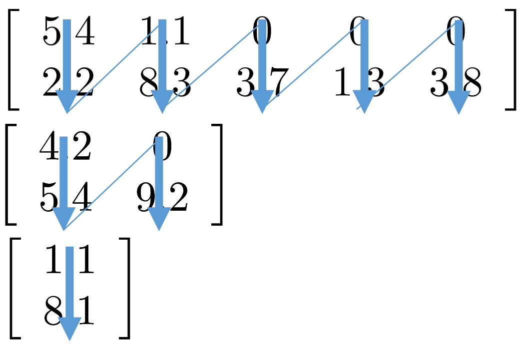
- Partition the matrix into row-blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL
  - Can still store col-major
  - Need a row pointer

$$
\begin{pmatrix}
5.4 & 1.1 & 0 & 0 & 0 & 0 \\
2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\
0 & 0 & 4.2 & 0 & 0 & 0 \\
5.4 & 0 & 0 & 9.2 & 0 & 0 \\
0 & 0 & 0 & 0 & 1.1 & 0 \\
0 & 0 & 0 & 0 & 0 & 8.1
\end{pmatrix}
$$

Block0

Block1

Block2



sliced-ELL format :

$$
\text{value} = [\, 5.4\ 2.2\ 1.1\ 8.3\ 0.0\ 3.7\ 0.0\ 1.3\ 0.0\ 3.8 \;|\; 4.2\ 5.4\ 0.0\ 9.2 \;|\; 1.1\ 8.1 \,]
$$

$$
\text{colidx} = [\, 0\ \ 0\ \ 1\ \ 1\ \ -\ \ 3\ \ -\ \ 4\ \ -\ \ 5 \;|\; 2\ \ 0\ \ -\ \ 3 \;|\; 4\ \ 5 \,]
$$

rowptr = [0 10 14 16]

Number of nonzero elements

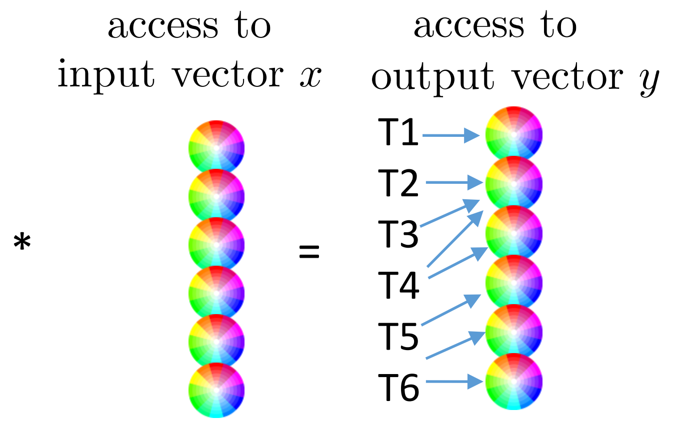
COO SpMV

Input A, x, y Output $y = A \cdot x$

Split nonzero elements into chunks and parallelize across chunks.

- Partial sums need synchronization / atomics to avoid write conflicts.
- Non-coalesced memory access (because row-major).

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$



$$\text{value} = [\; 5.4 \;\; 1.1 \;\; 2.2 \;\; 8.3 \;\; 3.7 \;\; 1.3 \;\; 3.8 \;\; 4.2 \;\; 5.4 \;\; 9.2 \;\; 1.1 \;\; 8.1 \;]$$

$$\text{colidx} = [\; 0 \;\; 1 \;\; 0 \;\; 1 \;\; 3 \;\; 4 \;\; 5 \;\; 2 \;\; 0 \;\; 3 \;\; 4 \;\; 5 \;]$$

$$\text{rowidx} = [\; 0 \;\; 0 \;\; 1 \;\; 1 \;\; 1 \;\; 1 \;\; 1 \;\; 2 \;\; 3 \;\; 3 \;\; 4 \;\; 5 \;]$$

Value

Column-index

Row-index

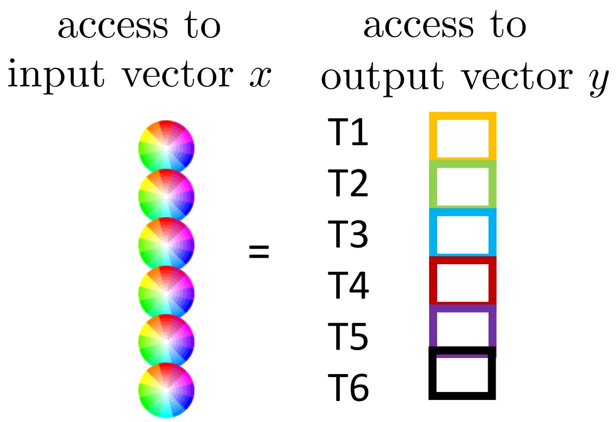
CSR SpMV

Input A, x, y Output $y = A \cdot x$

- Parallelize by rows:
  - Every “thread” handles the computation of one sum in local memory.

T1	5.4	1.1	0	0	0	0
T2	2.2	8.3	0	3.7	1.3	3.8
T3	0	0	4.2	0	0	0
T4	5.4	0	0	9.2	0	0
T5	0	0	0	0	1.1	0
T6	0	0	0	0	0	8.1

*



value = [5.4 1.1 | 2.2 8.3 3.7 1.3 3.8 | 4.2 | 5.4 9.2 | 1.1 | 8.1] Value

colidx = [0 1 | 0 1 3 4 5 | 2 | 0 3 | 4 | 5] Column-index

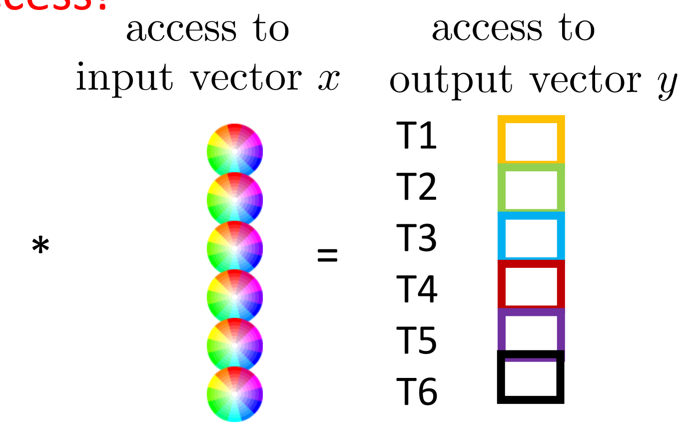
rowptr = [0 2 7 8 10 11 12] Row-pointer

CSR SpMV

Input A, x, y Output $y = A \cdot x$

- Parallelize by rows:
  - Every “thread” handles the computation of one sum in local memory.
  - Significant workload imbalance!
  - Can not store the matrix in Col-Major format for coalesced access!

T1	5.4	1.1	0	0	0	0
T2	2.2	8.3	0	3.7	1.3	3.8
T3	0	0	4.2	0	0	0
T4	5.4	0	0	9.2	0	0
T5	0	0	0	0	1.1	0
T6	0	0	0	0	0	8.1



$\text{value} = [\ 5.4 \quad 1.1 \mid 2.2 \quad 8.3 \quad 3.7 \quad 1.3 \quad 3.8 \mid 4.2 \mid 5.4 \quad 9.2 \mid 1.1 \mid 8.1\ ]$ Value

$\text{colidx} = [\ 0 \quad 1 \mid 0 \quad 1 \quad 3 \quad 4 \quad 5 \mid 2 \mid 0 \quad 3 \mid 4 \mid 5\ ]$ Column-index

$\text{rowptr} = [\ 0 \quad 2 \quad 7 \quad 8 \quad 10 \quad 11 \quad 12\ ]$ Row-pointer

ELL SpMV

Input A, x, y Output $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

T1	5.4	1.1	0	0	0	0	0	1	–	–	–	–
T2	2.2	8.3	3.7	1.3	3.8	0	0	1	3	4	5	–
T3	4.2	0	0	0	0	0	2	–	–	–	–	–
T4	5.4	9.2	0	0	0	0	0	3	–	–	–	–
T5	1.1	0	0	0	0	0	4	–	–	–	–	–
T6	8.1	0	0	0	0	0	5	–	–	–	–	–

Pad rows to uniform length

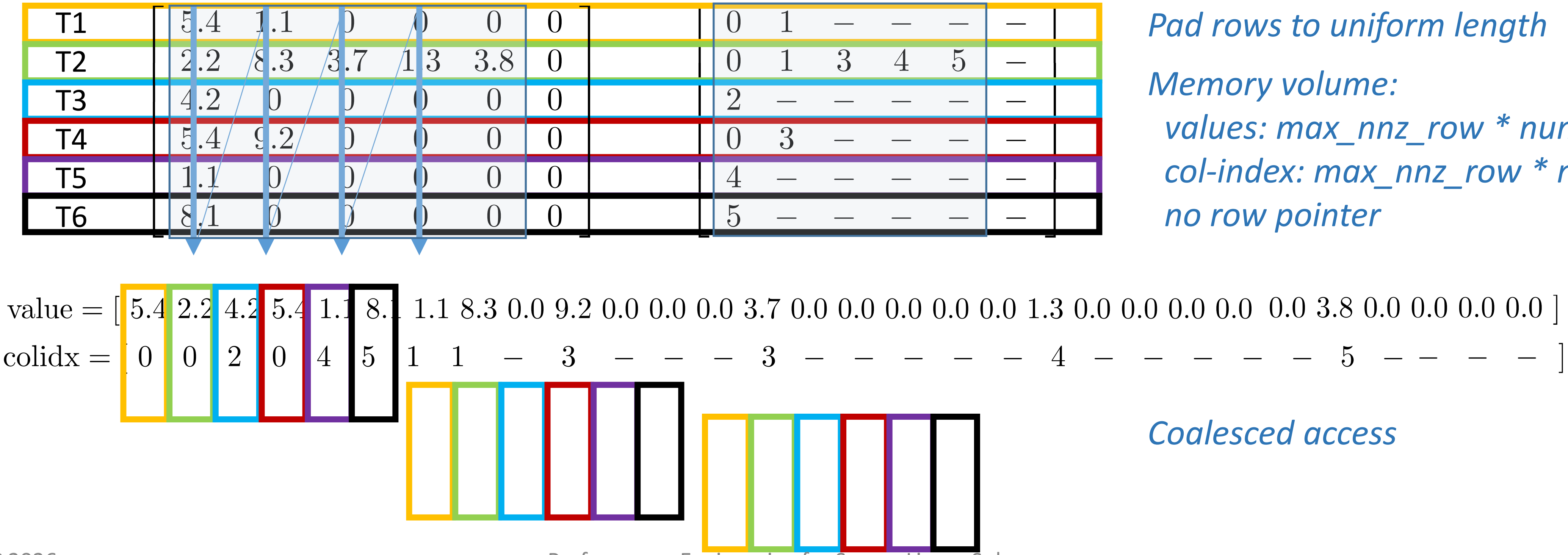
Memory volume:

- *values: max_nnz_row * num_rows*
- *col-index: max_nnz_row * num_rows*
- *no row pointer*

ELL SpMV

Input A, x, y Output $y = A \cdot x$

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$



Pad rows to uniform length

Memory volume:

*values: max_nnz_row * num_rows*

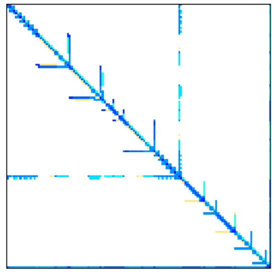
*col-index: max_nnz_row * num_rows*

no row pointer

Coalesced access

ELL SpMV

Input A, x, y Output $y = A \cdot x$



$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$

T1	5.4	1.1	0	0	0	0	0	1	–	–	–	–
T2	2.2	8.3	3.7	1.3	3.8	0	0	1	3	4	5	–
T3	4.2	0	0	0	0	0	2	–	–	–	–	–
T4	5.4	9.2	0	0	0	0	0	3	–	–	–	–
T5	1.1	0	0	0	0	0	4	–	–	–	–	–
T6	8.1	0	0	0	0	0	5	–	–	–	–	–

Pad rows to uniform length

Memory volume:

*values: max_nnz_row * num_rows*

*col-index: max_nnz_row * num_rows*

no row pointer

value = [5.4 2.2 4.2 5.4 1.1 8.1 1.1 8.3 0.0 9.2 0.0 0.0 0.0 3.7 0.0 0.0 0.0 0.0 0.0 1.3 0.0 0.0 0.0 0.0 0.0 3.8 0.0 0.0 0.0 0.0]

colidx = [0 0 2 0 4 5 1 1 – 3 – – – 3 – – – – – 4 – – – – – 5 – – – –]

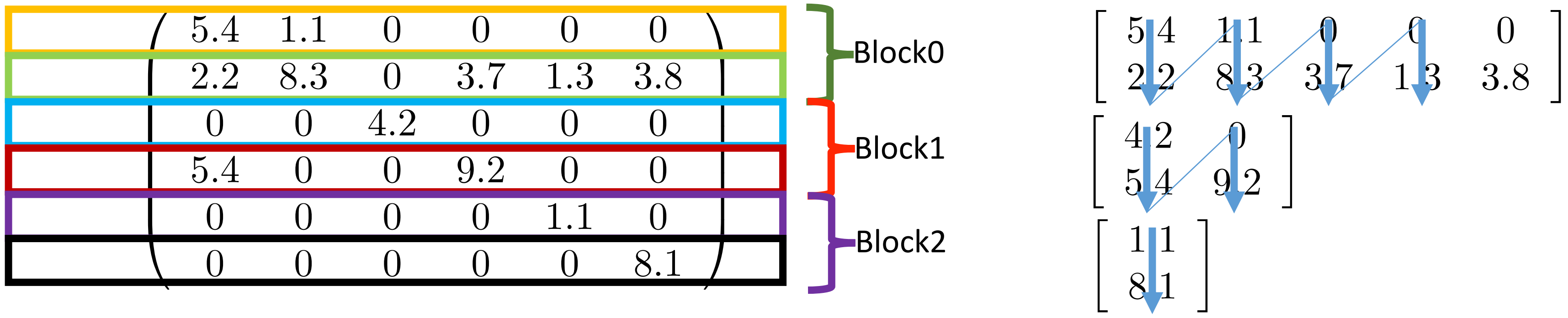
Coalesced access

Can be wasteful (overhead)

Sliced-ELL SpMV

Input A, x, y Output $y = A \cdot x$

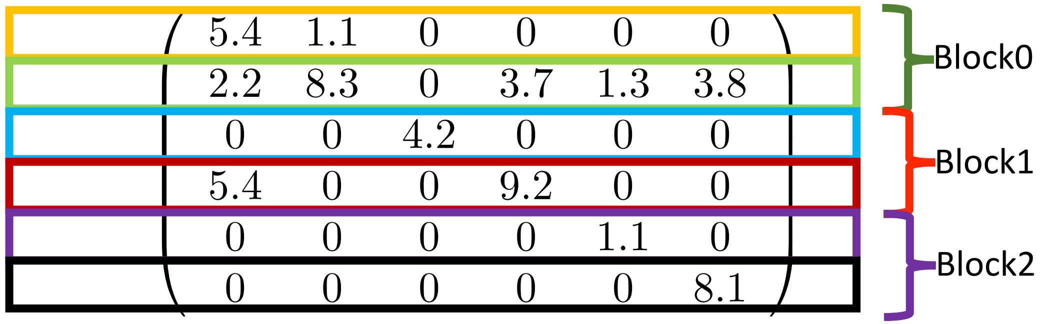
- Partition the matrix into blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL.
  - Can still store col-major.

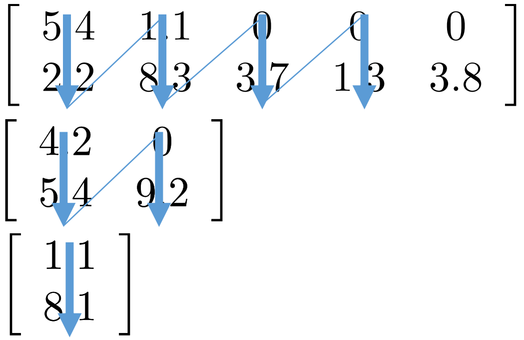


Sliced-ELL SpMV

Input A, x, y Output $y = A \cdot x$

- Partition the matrix into blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL.
  - Can still store col-major.



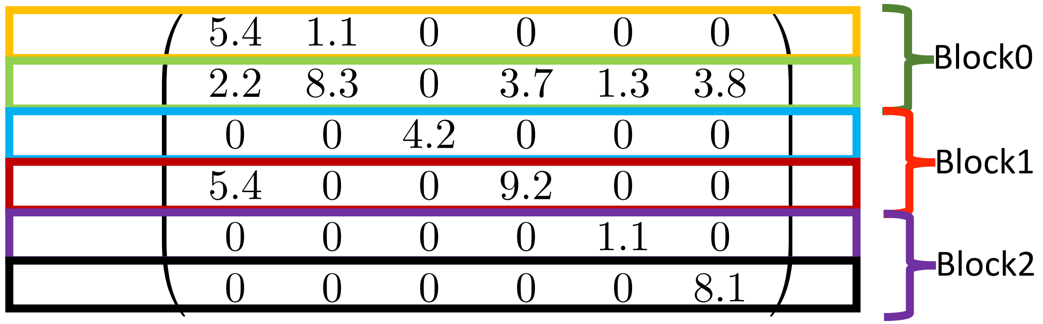


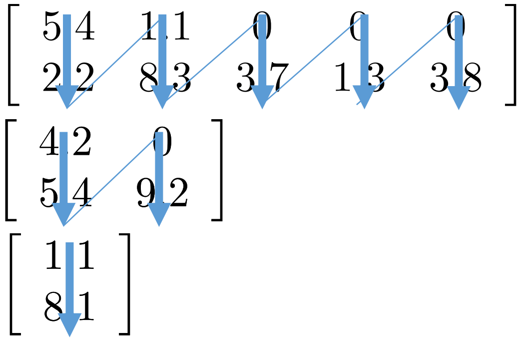
$$\text{value} = [\,5.4\ 2.2\ 1.1\ 8.3\ 0.0\ 3.7\ 0.0\ 1.3\ 0.0\ 3.8\ \ 4.2\ 5.4\ 0.0\ 9.2\ \ 1.1\ 8.1\,]$$
$$\text{colidx} = [\,0\ 0\ 1\ 1\ -\ 3\ -\ 4\ -\ 5\ \ 2\ 0\ -\ 3\ \ 4\ 5\,]$$

Sliced-ELL SpMV

Input A, x, y Output $y = A \cdot x$

- Partition the matrix into blocks & use ELL for the distinct blocks.
  - Reduce overhead of ELL.
  - Can still store col-major.
  - Need for a row pointer.





sliced-ELL format :

$$\text{value} = [\,5.4\ 2.2\ 1.1\ 8.3\ 0.0\ 3.7\ 0.0\ 1.3\ 0.0\ 3.8\ \ 4.2\ 5.4\ 0.0\ 9.2\ \ 1.1\ 8.1\,]$$

$$\text{colidx} = [\,0\ 0\ 1\ 1\ -\ 3\ -\ 4\ -\ 5\ \ 2\ 0\ -\ 3\ \ 4\ 5\,]$$

rowptr = [0 10 14 16]

Slice matrix into blocks, store blocks in ELL format with offset-pointer.

SpMV Formats and SpMV Kernels

Input A, x, y Output $y = A \cdot x$

"Different formats optimal for different problems"

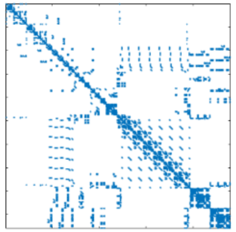
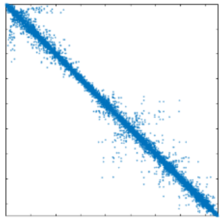
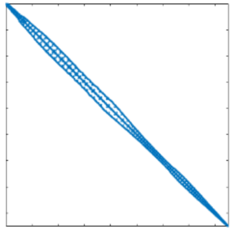
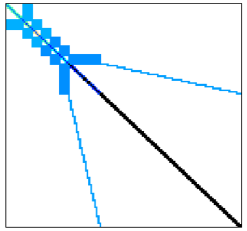
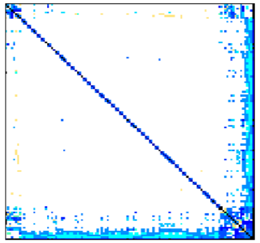
COO
- can compensate workload imbalance for irregular patterns
- Efficient for MIMD processing
- Strong support for atomics needed

CSR
- small memory footprint
- Needs some logic for row-parallel processing
- Efficient for MIMD processing

ELL
- Efficient for balanced matrices
- Enables col-major storage
- Efficient for SIMD processing

SELL-c
- Enables col-major storage
- Tunable between CSR and ELL

…

SpMV (CSR) Roofline

```
for (int row = 0; row < num_rows; ++row) {
  double sum = 0.0;
  for (int k = row_ptrs[row]; k < row_ptrs[row + 1]; ++k)
    sum += mat_values[k] * b[col_idxs[k]];
  x[row] += sum;
}
```

Optimistic intensity:

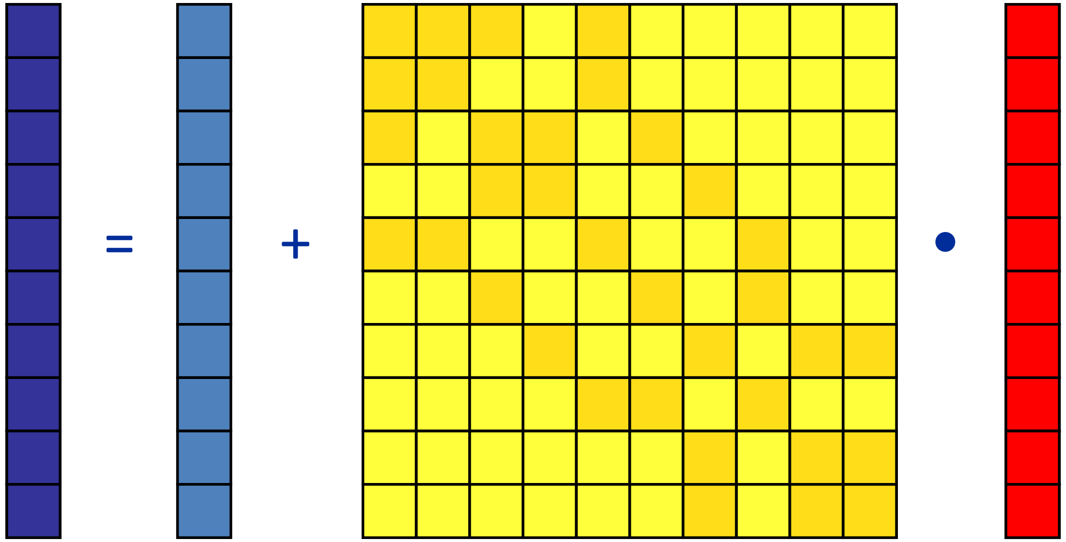
$$I_{max} = \frac{2\,N_{nz}}{12\,N_{nz} + 20\,N_r + 8\,N_c}\,\frac{\mathrm{F}}{\mathrm{B}}$$

$$= \frac{1}{6 + 10/N_{nzr} + 4/N_{nzc}}\,\frac{\mathrm{F}}{\mathrm{B}}$$

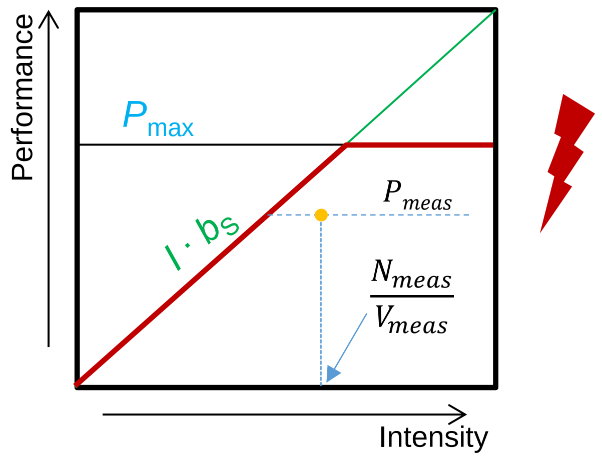
square matrix

$$= \frac{1}{6 + 10/N_{nzr} + 4/N_{nzr}}\,\frac{\mathrm{F}}{\mathrm{B}}$$

$$\xrightarrow{N_{nzr} \gg 10} \frac{1}{6}\,\frac{\mathrm{F}}{\mathrm{B}}$$



Roofline “failure” with SpMV



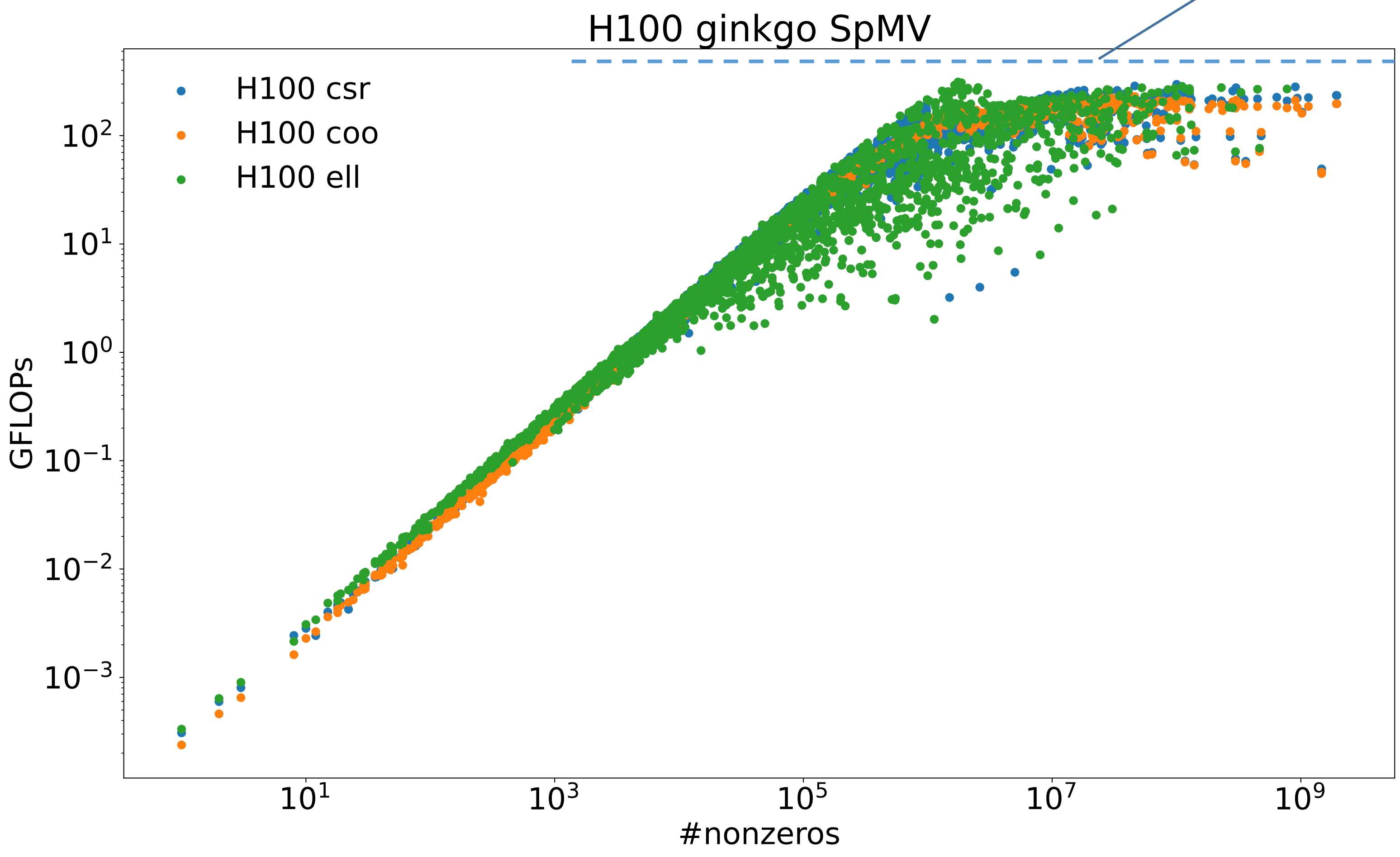
Reasons for performance not attaining the limit

1. Intensity lower than the minimum
   - More RHS traffic than the optimistic limit ($\frac{4}{N_{nzr}}$ 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

Experiences with SpMV on GPUs

Looking at ~3,000 test matrices from Suite Sparse Matrix Collection

Absolute CSR limit for $b_S = 3\ \mathrm{TB/s}$



Hands-On:

SpMV benchmarking

Solving Sparse Linear Systems

Why we often don’t use direct solvers

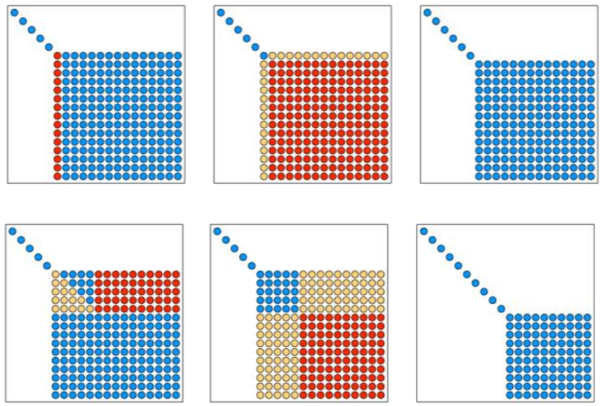
Characteristics and optimization of iterative solvers

Preconditioning using Matrix Polynomials

Jonas Thies (j.thies@tudelft.nl)

Can we use direct solvers for solving sparse problems?

$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$



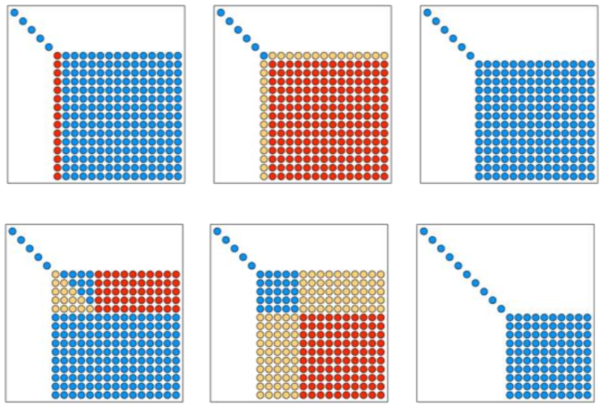
Scalar or block-LU?

Are zeros preserved in the factorization?

Can we store the fill-in?

Can we use direct solvers for solving sparse problems?

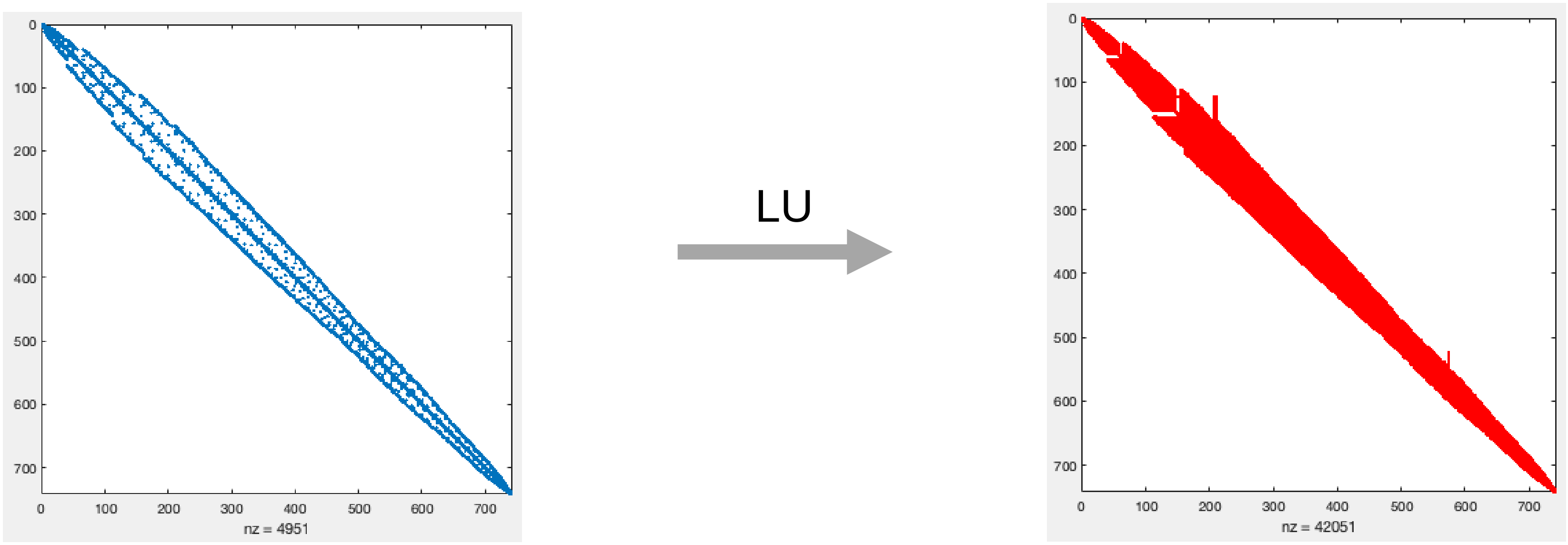
$$A = \begin{pmatrix} 5.4 & 1.1 & 0 & 0 & 0 & 0 \\ 2.2 & 8.3 & 0 & 3.7 & 1.3 & 3.8 \\ 0 & 0 & 4.2 & 0 & 0 & 0 \\ 5.4 & 0 & 0 & 9.2 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1.1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 8.1 \end{pmatrix}$$



Scalar or block-LU?

Are zeros preserved in the factorization?

Can we store the fill-in?



Iterative Solvers

Generate a sequence of solution approximations with increasing approximation quality.

$$x^0 \rightsquigarrow x^1 \rightsquigarrow x^2 \rightsquigarrow x^3 \rightsquigarrow \cdots$$

Iterative Solvers

Generate a sequence of solution approximations with increasing approximation quality.

$$x^0 \rightsquigarrow x^1 \rightsquigarrow x^2 \rightsquigarrow x^3 \rightsquigarrow \cdots$$

Relaxations

- Base on matrix splitting
- Jacobi relaxation:

$$\begin{aligned} Ax &= b \\ (L + D + U)x &= b \\ Dx &= b - (L+U)x \\ x &= D^{-1}b - D^{-1}(L+U)x \end{aligned}$$

$$x^{k+1} = D^{-1}b - D^{-1}(A-D)x^k$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Iterative Solvers

Generate a sequence of solution approximations with increasing approximation quality.

$$x^0 \rightsquigarrow x^1 \rightsquigarrow x^2 \rightsquigarrow x^3 \rightsquigarrow \cdots$$

Relaxations

- Base on matrix splitting
- Jacobi relaxation:

$$Ax = b$$
$$(L + D + U)x = b$$
$$Dx = b - (L + U)x$$
$$x = D^{-1}b - D^{-1}(L + U)x$$
$$x^{k+1} = D^{-1}b - D^{-1}(A - D)x^k$$

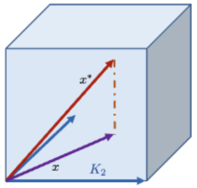
- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Krylov Subspace Methods

- Iteratively grow Krylov subspace

$$K_i(A, r) = span\left\{r, Ar, A^2r, \ldots, A^{i-1}r\right\}$$
$$K_0 \subset K_1 \subset K_2 \subset \cdots \mathbb{R}^n$$

- Approximate solution in Krylov Subspace



- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Iterative Solvers

Generate a sequence of solution approximations with increasing approximation quality.

$x^0 \rightsquigarrow x^1 \rightsquigarrow x^2 \rightsquigarrow x^3 \rightsquigarrow \cdots$

Relaxations

- Base on matrix splitting
- Jacobi relaxation:

$$Ax = b$$
$$(L + D + U)x = b$$
$$Dx = b - (L + U)x$$
$$x = D^{-1}b - D^{-1}(L + U)x$$

$$x^{k+1} = D^{-1}b - D^{-1}(A - D)x^k$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

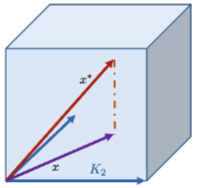
Krylov Subspace Methods

- Iteratively grow Krylov subspace

$$K_i(A, r) = span\left\{r, Ar, A^2r, \dots, A^{i-1}r\right\}$$

$$K_0 \subset K_1 \subset K_2 \subset \cdots \mathbb{R}^n$$

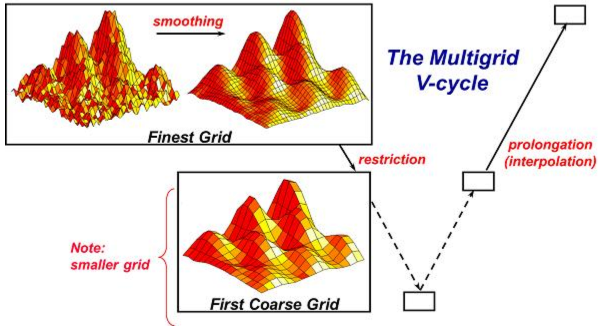
- Approximate solution in Krylov Subspace



- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Multigrid Methods

- Recursively project problem to coarser grid and solve on coarser grid

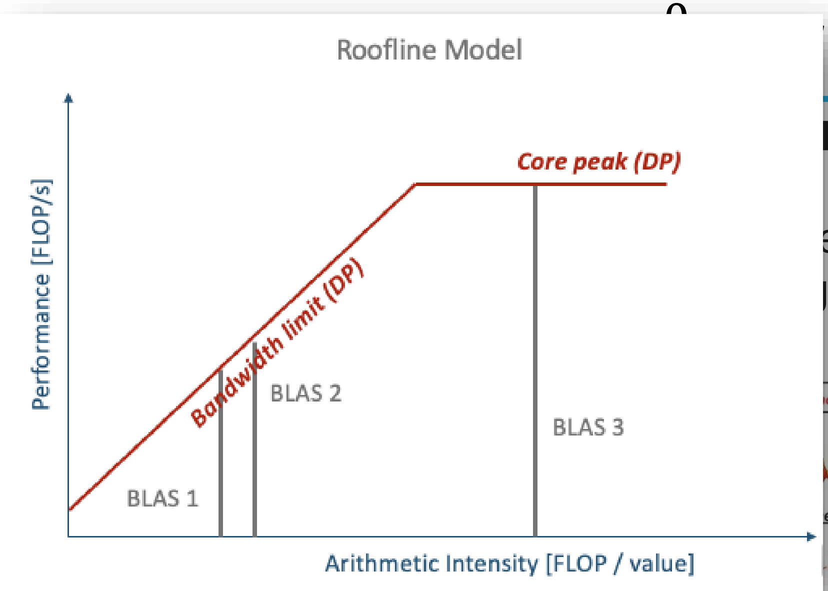


- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Iterative Solvers

Generate a sequence of solution approximations with increasing approximation quality.

$$\ldots^{1} \rightsquigarrow x^2 \rightsquigarrow x^3 \rightsquigarrow \cdots$$



Relaxations

- Base on matrix splitting
- Jacobi relaxation:

$$Ax = b$$
$$(L + D + U)x = b$$
$$Dx = b - (L + U)x$$
$$x = D^{-1}b - D^{-1}(L + U)x$$

$$x^{k+1} = D^{-1}b - D^{-1}(A - D)x^k$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

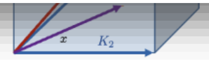
Krylov Sub

- Iterativel

$K_i(A, r) =$

$K_0 \subset K_1$

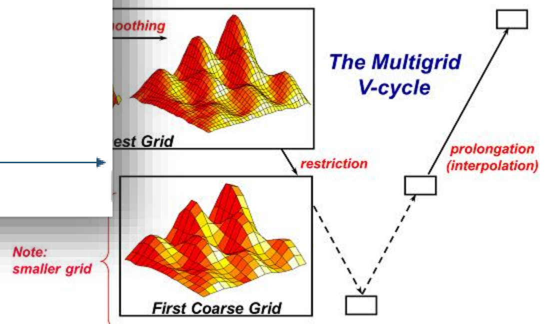
- Approxin in Krylov



- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

lethods

- ely project problem to rid and solve on coarser



- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Example: Conjugate Gradients (CG)

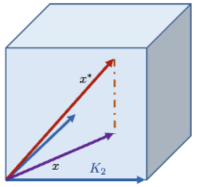
Krylov Subspace Methods

- Iteratively grow Krylov subspace

$$K_i(A, r) = span\{r, Ar, A^2r, \ldots, A^{i-1}r\}$$

$$K_0 \subset K_1 \subset K_2 \subset \cdots \mathbb{R}^n$$

- Approximate solution in Krylov Subspace



- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Optimal Krylov solver for symmetric and positive definite (SPD) matrices
Requires storing only four additional vectors

$$
\begin{aligned}
&\text{input: } A, b, x_0, it_{max} \\
&r_0 = b - Ax_0 \\
&p_0 = r_0 \\
&\textbf{for } k = 0, \ldots, it_{max} \textbf{ do} \\
&\quad \alpha_k = \frac{r_k^T r_k}{p_k^T A p_k} \\
&\quad x_{k+1} = x_k + \alpha_k p_k \\
&\quad r_{k+1} = r_k - \alpha_k A p_k \\
&\quad \beta_k = \frac{r_{k+1}^T r_{k+1}}{r_k^T r_k} \\
&\quad p_{k+1} = r_{k+1} + \beta_k p_k \\
&\textbf{end for}
\end{aligned}
$$

Preconditioning Iterative Solvers

Transform linear problem by multiplying both sides with $P \approx A^{-1}$ such that iterations converge faster.

$$Ax = b \quad \Leftrightarrow \quad \underbrace{PA}_{\tilde{A}}x = \underbrace{Pb}_{\tilde{b}} \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}$$

Preconditioning Iterative Solvers

Transform linear problem by multiplying both sides with $P \approx A^{-1}$ such that iterations converge faster.

$$Ax = b \quad \Leftrightarrow \quad \underbrace{PA}_{\tilde{A}}x = \underbrace{Pb}_{\tilde{b}} \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}$$

Iterative solver as preconditioner

- Multigrid
- Jacobi $\quad D^{-1}Ax = D^{-1}b$
- Block-Jacobi

- Sparse Approximate Inverses
- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Preconditioning Iterative Solvers

Transform linear problem by multiplying both sides with $P \approx A^{-1}$ such that iterations converge faster.

$$Ax = b \quad \Leftrightarrow \quad \underbrace{PA}_{\tilde{A}}x = \underbrace{Pb}_{\tilde{b}} \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}$$

Iterative solver as preconditioner

- Multigrid
- Jacobi $\quad D^{-1}Ax = D^{-1}b$
- Block-Jacobi

- Sparse Approximate Inverses
- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Incomplete Factorizations

- Compute LU factorization with restricted fill-in

- Replace triangular solver with iteratively solving factors

$$L \cdot y = b \qquad U \cdot x = y$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Preconditioning Iterative Solvers

Transform linear problem by multiplying both sides with $P \approx A^{-1}$ such that iterations converge faster.

$$Ax = b \quad \Leftrightarrow \quad \underbrace{PA}_{\tilde{A}}x = \underbrace{Pb}_{\tilde{b}} \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}$$

Iterative solver as preconditioner

- Multigrid
- Jacobi $\quad D^{-1}Ax = D^{-1}b$
- Block-Jacobi

- Sparse Approximate Inverses
- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Incomplete Factorizations

- Compute LU factorization with restricted fill-in

- Replace triangular solver with iteratively solving factors

$$L \cdot y = b \qquad U \cdot x = y$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Polynomial preconditioners

- Choose $\quad A = M - N$

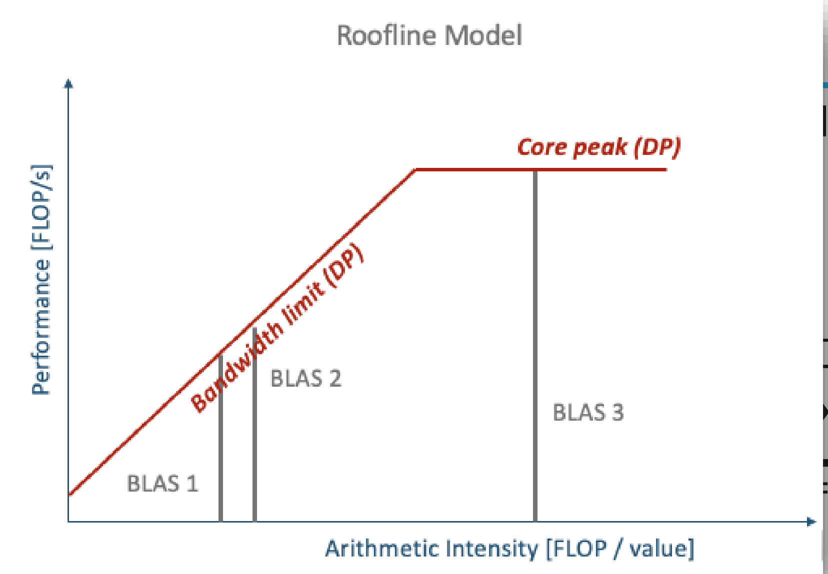
$$P = \left(\sum_{i=0}^{p-1} (I - M^{-1}A)^i \right) M^{-1}$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Preconditioning Iterative Solvers

$$Ax = b \quad \Leftrightarrow \quad \underbrace{PA}_{\tilde{A}}x = \underbrace{Pb}_{\tilde{b}} \quad \Leftrightarrow \quad \tilde{A}x = \tilde{b}$$

Transform linear problem by multiplying both sides with $P \approx A^{-1}$ such that iterations converge faster.



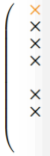
Iterative solver as preconditioner

- Multigrid
- Jacobi $\quad D^{-1}Ax = D^{-1}b$
- Block-Jacobi

- Sparse Approximate Inverses
- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Incomplete

- Compute restricted



- Replace iteratively solving factors

$$L \cdot y = b \qquad U \cdot x = y$$

- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

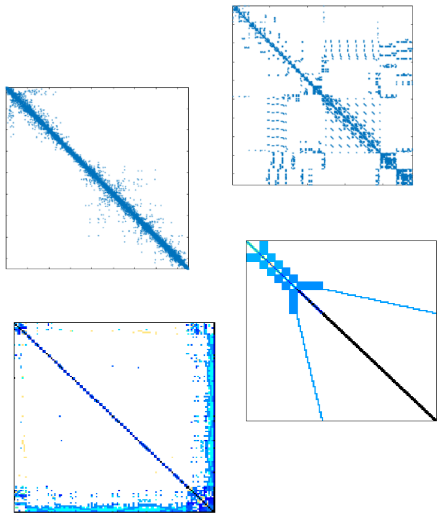
l preconditioners

$$A = M - N$$

$$^{-1} \quad =0 \quad (I - M^{-1}A)^i \Bigg) M^{-1}$$

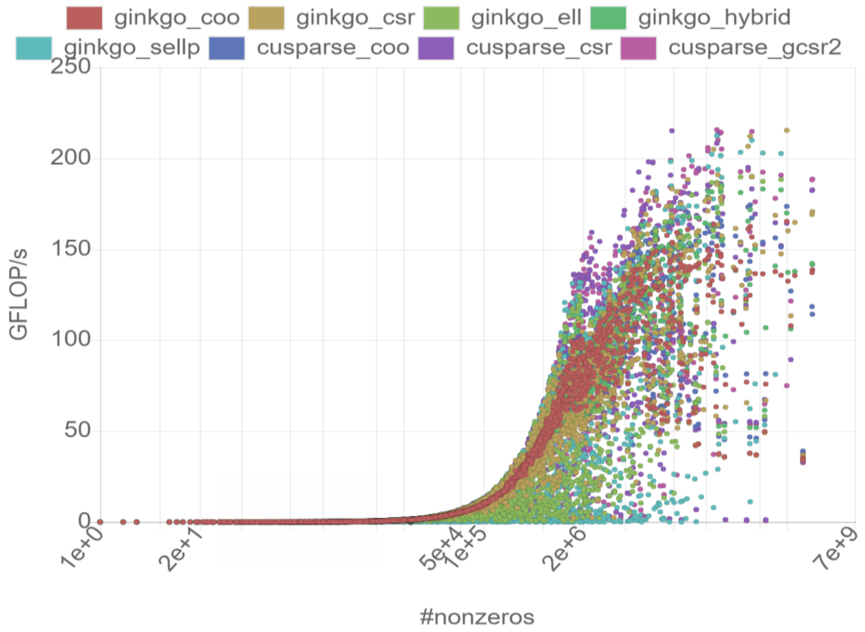
- Matrix-Vector Prod. & Vector Ops
- Low arithmetic intensity

Optimizing Iterative Solvers & Preconditioners

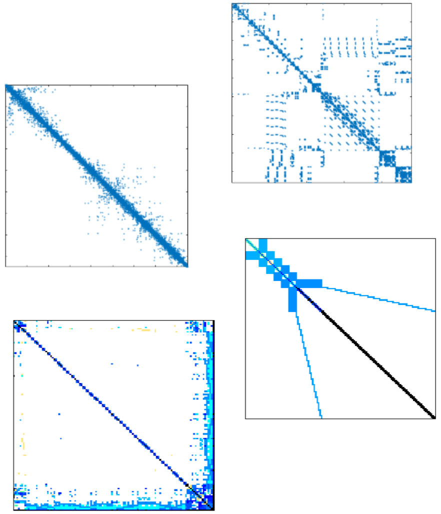


1. **Optimizing the matrix vector product as common building block**

   - Optimization of **sparse data format** and **processing scheme**



Optimizing Iterative Solvers & Preconditioners



1. **Optimizing the matrix vector product as common building block**
   - Optimization of **sparse data format** and **processing scheme**

2. **Cache-Aware implementation**
   - Merging of Operations into **super-kernels** to reduce the memory access

BiCGStab Krylov solver (van der Vorst, 1992)

1. $r_0 = b - Ax_0$
2. Choose an arbitrary vector $\hat{r}_0$ such that $(\hat{r}_0, r_0) \neq 0$, e.g., $\hat{r}_0 = r_0$
3. $\rho_0 = \alpha = \omega_0 = 1$
4. $v_0 = p_0 = 0$
5. For $i = 1, 2, 3, \ldots$
   1. $\rho_i = (\hat{r}_0, r_{i-1})$
   2. $\beta = (\rho_i / \rho_{i-1})(\alpha / \omega_{i-1})$
   3. $p_i = r_{i-1} + \beta(p_{i-1} - \omega_{i-1} v_{i-1})$
   4. $v_i = Ap_i$
   5. $\alpha = \rho_i / (\hat{r}_0, v_i)$
   6. $s = r_{i-1} - \alpha v_i$
   7. $t = As$
   8. $\omega_i = (t, s)/(t, t)$
   9. $x_i = x_{i-1} + \alpha p_i + \omega_i s$
   10. If x_i is accurate enough then quit
   11. $r_i = s - \omega_i t$

$p_k := r_{k-1} + \beta(p_{k-1} - \omega_{k-1} v_{k-1})$

cuBLAS

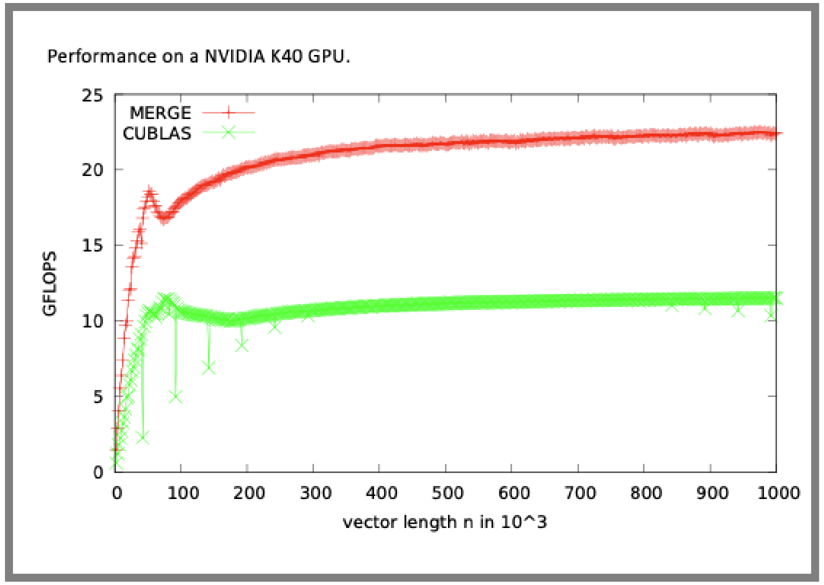
```
cublasDscal( n, beta, p, 1 );
cublasDaxpy( n, omega * beta, v, 1 , p, 1 );
cublasDaxpy( n, 1.0, r, 1, p, 1 );
```

3 kernels - 5n reads, 3n writes

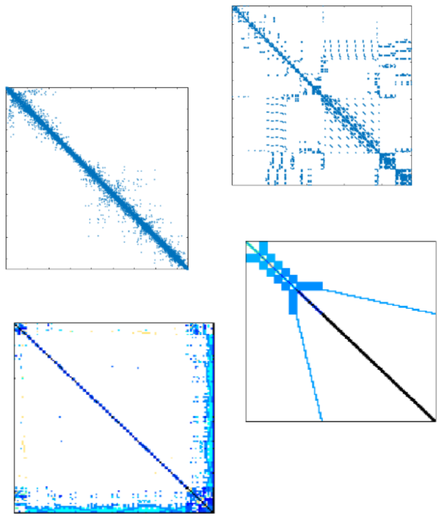
merge in one kernel

```
p_update( int n, double beta, double omega,
          double *v, double *r, double *p ){
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  if( i<n )
    p[i] = r[i] + beta * ( p[i]-omega*v[i] );
}
```

1 kernel - 3n reads, 1n writes



Optimizing Iterative Solvers & Preconditioners

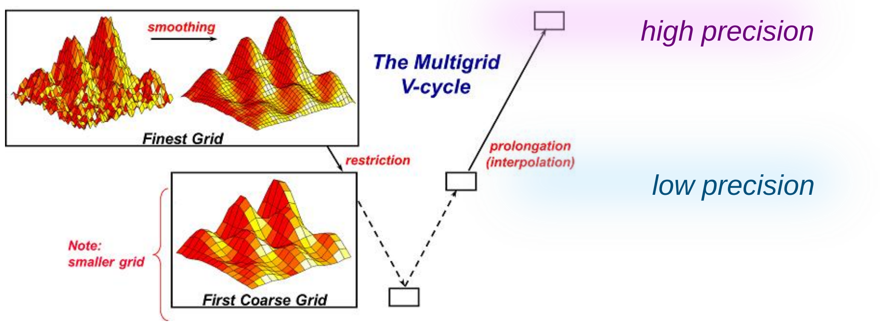


1. **Optimizing the matrix vector product as common building block**
   - Optimization of **sparse data format** and **processing scheme**

2. **Cache-Aware implementation**
   - Merging of Operations into **super-kernels** to reduce the memory access

3. **Reduce memory traffic by additional computations**
   - Mixed Precision algorithms using low precision in parts of the computations
   - Matrix Powers Kernel (evaluate $y=A^k x$ without loading A *k* times)

4. **Reduce data dependencies for higher throughput (via spMV)**
   - Jacobi or Chebyshev smoothers instead of Gauss-Seidel
   - Polynomial approximation for triangular solves



Hands-On:

Conjugate-Gradient Solver

Tutorial conclusions

- Memory bandwidth limitations are ubiquitous in sparse linear solvers
- SpMV performance depends on the storage format **and** matrix characteristics
- Roofline is an indispensable tool for performance analysis
- Preconditioners are the decisive part of linear solvers
- Time to solution is a tradeoff between flop/s performance and fast convergence
    - Faster convergence rate may come at a price of lower flop/s

Appendix

Performance Engineering for Linear Solvers

This tutorial covers code analysis, performance modeling, and optimization for linear solvers on CPU and GPU nodes. Performance Engineering is often taught using simple loops as instructive examples for performance models and how they can guide optimization; however, full, preconditioned linear solvers comprise multiple back-to-back loops enclosed in an iteration scheme that is executed until convergence is achieved. Consequently, the concept of "optimal performance" has to account for both hardware resource efficiency and iterative solver convergence. We convey a performance engineering process that is geared towards linear iterative solvers. After introducing basic notions of hardware organization and storage for dense and sparse data structures, we show how the Roofline performance model can be applied to such solvers in predictive and diagnostic ways and how it can be used to assess the hardware efficiency of a solver, covering important corner cases such as pure memory boundedness. Then we advance to the structure of preconditioned solvers, using the Conjugate Gradient Method (CG) algorithm as a leading example. Hotspots and bottlenecks of the complete solver are identified followed by the introduction of advanced performance optimization techniques like preconditioning and cache blocking.

Jonas Thies

Jonas has more than 20 years of experience in HPC and scientific computing with applications in CFD, climate research and quantum physics. Specifically, he has worked on domain decomposition methods for sparse linear systems, implicit ocean models, sparse eigenvalue problems on heterogeneous supercomputers, code optimization for multi-core CPUs and vector processors, and software and performance engineering for scientific applications.

Jonas has a PhD in applied mathematics (Groningen 2011). He spent two years at the Center for Interdisciplinary Mathematics in Uppsala, after which he moved to Cologne as a Scientific Employee of the German Aerospace Center (DLR) Institute for Software Technology. There he led a research group on parallel numerics from 2017 to 2021. Since June 2021 he is an Assistant Professor at the Delft High Performance Computing Center DHPC, where he coordinates the center‘s training activities.

https://www.tudelft.nl/en/eemcs/the-faculty/departments/applied-mathematics/people/dr-j-jonas-thies

Hartwig Anzt

Hartwig Anzt is the Chair of Computational Mathematics at the TUM School of Computation, Information and Technology of the Technical University of Munich (TUM) Campus Heilbronn. He also holds a Research Associate Professor position at the Innovative Computing Lab (ICL) at the University of Tennessee (UTK). Hartwig Anzt received a PhD in applied mathematics from the Karlsruhe Institute of Technology (KIT) and specializes in iterative methods and preconditioning techniques for the next generation hardware architectures. He also has a long track record of high-quality development. He is author of the MAGMA-sparse open-source software package and managing lead of the Ginkgo math software library. Hartwig Anzt had served as a PI in the Software Technology (ST) pillar of the US Exascale Computing Project (ECP), including a coordinated effort aiming at integrating low-precision functionality into high-accuracy simulation codes. He also is a PI in the EuroHPC project MICROCARD.

Hartwig Anzt is the main author of more than 100 peer-reviewed publications, part of the scientific committee of international conferences, Associate Editor of the SIAM Journal on Scientific Computing (SISC), Associate Editor of ACM Transactions on Parallel Computing, workshop chair for ISC High Performance 2022, and has been elected as SIAM Activity Group on Supercomputing program manager.

https://hartwiganzt.github.io/

Georg Hager

Georg Hager holds a PhD and a habilitation degree in Computational Physics from the University of Greifswald. He heads the Rsearch Division of the Erlangen National High Performance Computing Center (NHR@FAU). Previously he was a senior researcher in the HPC Services group at Erlangen Regional Computing Center (RRZE), which is part of the Friedrich-Alexander-Universität Erlangen-Nürnberg. Recent research includes architecture-specific optimization strategies for current microprocessors, performance engineering of scientific codes, and analytic modeling of massively parallel programs. His textbook “Introduction to High Performance Computing for Scientists and Engineers” is recommended or required reading in many HPC-related lectures and courses worldwide. He has more than two decades of experience in teaching high performance computing and performance engineering to students and scientists. Together with colleagues from NHR@FAU and other centers, he conducts long-standing series of tutorials on Performance Engineering and Hybrid Programming.

https://blogs.fau.de/hager