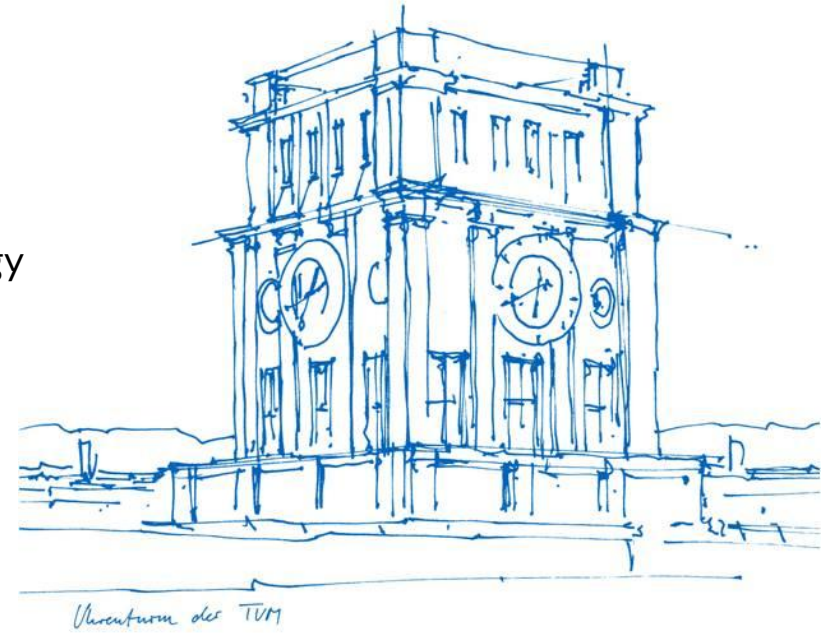
Multiplikation dünn besetzter Matrizen: Ellpack (column-major) *(Gruppe A311)*

Artem Lomov, Marina Wirsing, Simon Gunka

Technische Universität München

TUM School of Computation, Information and Technology

Garching, 27. August 2024



Format: ELLPACK (column-major)

$$\begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 6 & 0 & 0 \\ 0.5 & 7 & 0 & 0 \\ 0 & 0 & 0 & 3 \end{pmatrix} \rightarrow \begin{pmatrix} 5 & 6 & * & 3 \\ 0.5 & 7 & * & * \end{pmatrix} \begin{pmatrix} 0 & 1 & * & 4 \\ 2 & 2 & * & * \end{pmatrix} =$$

```
4,4,2
5,0.5,6,7,*,*,3,*
0,  2,1,2,*,*,3,*
—
<#Zeilen>, <#Spalten>, <#EllpackWerte>
<Werte>
<Indices>
```

Nur Werte != 0 werden mit einem zusätzlichen Zeilenindex abgespeichert, und pro Zeile werden genauso viele Werte angegeben, wie die vollste Spalte enthält

→ Spaltenindex wird damit überflüssig

Problemstellung

$$c_{ik} = \sum_{j=1}^{m} a_{ij} * b_{jk}$$

Problem: Matrizen beliebiger Größe (bis 2^{64} -1), Großteil der Einträge sind Nulleinträge

- Im Hauptspeicher summieren sich die nullen potenziell und fressen unnötig Speicher
- Eine klassische Dense-Multiplikation enthält damit viele überflüssige Berechnungen

$$\begin{pmatrix} 1 & 0 \\ 0 & 0 \end{pmatrix} * \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}$$

$$(1 * 0) + (0 * 0) = 0$$

$$\begin{bmatrix} 1,*,*,* \\ 0,*,*,* \end{bmatrix} \begin{bmatrix} *,*,1,* \\ *,*,0,* \end{bmatrix}$$

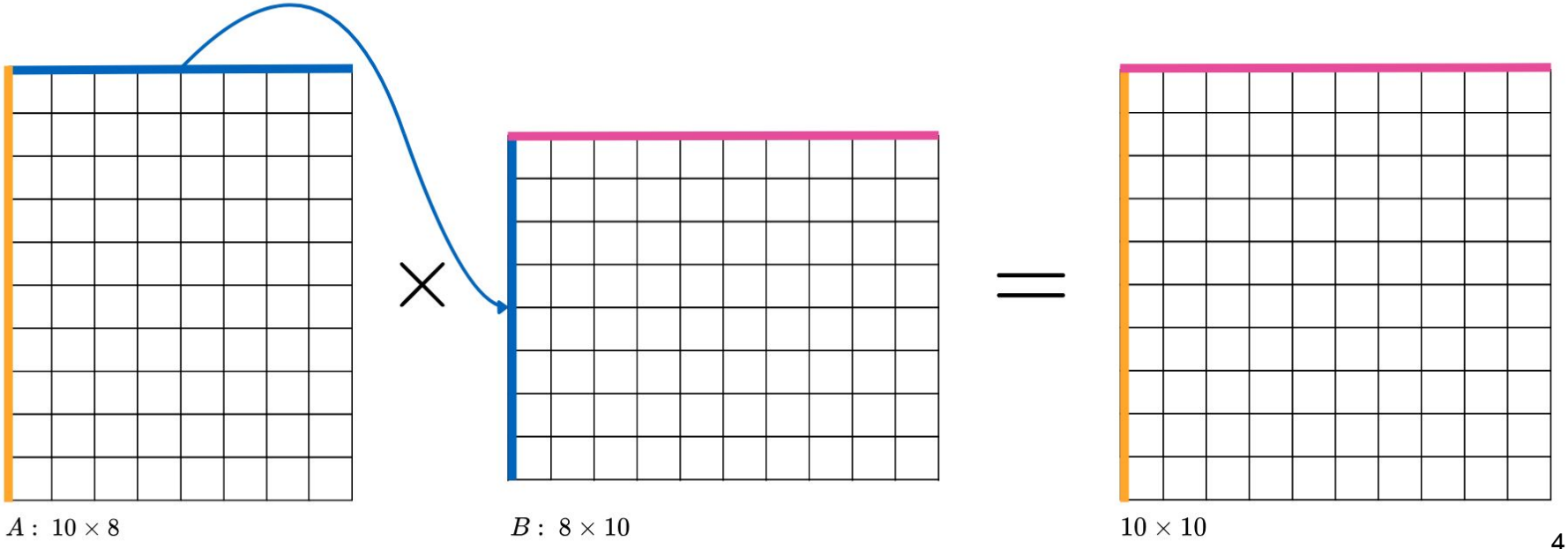
Anwendungen des Ellpack-Formats:
Simulationen, allgem. Datenverarbeitung (Bioinformatik, geograpische Informationssysteme), Machine Learning, etc.

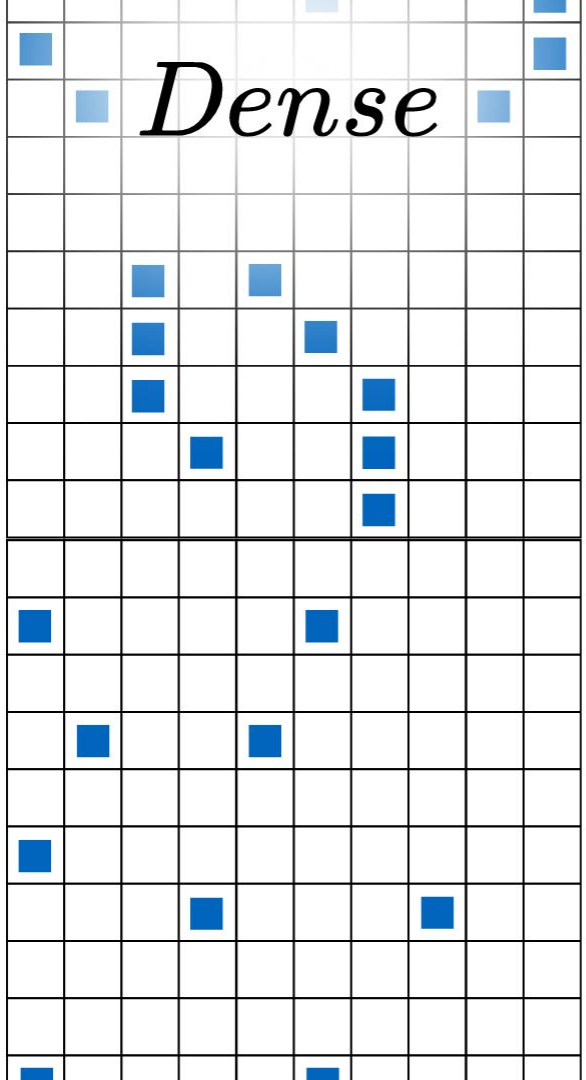
Beschränkung der Matrizengrößen

- 2^{64} chars = 2^{64} byte = 17.179.869.184 GB
- Für Matrizen mit >= 2^{32} rows
  - Dense rows: 2^{32} Iterationen → unmöglich
  - Sparse rows: Column-Major Ellpack ist eindeutig das falsche Format

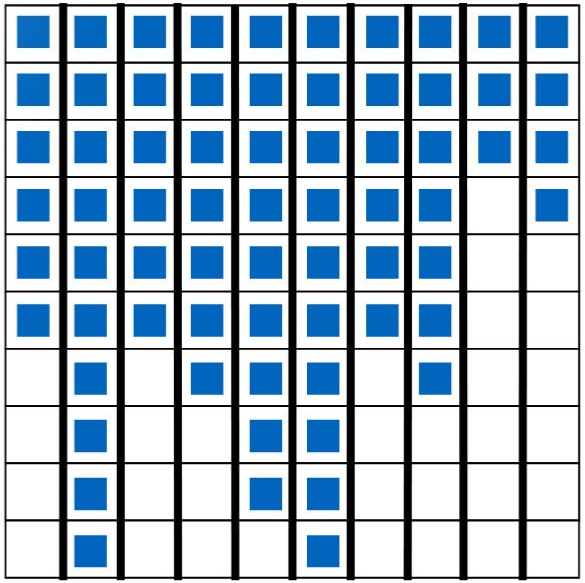
Beschränkt durch Dateigröße A/B

Beschränkt nur durch Ellpack-Werte

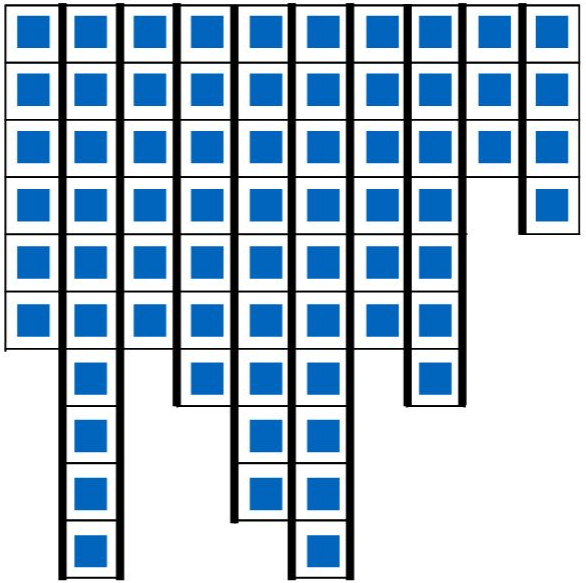




Ellpack
(*allgemein*)

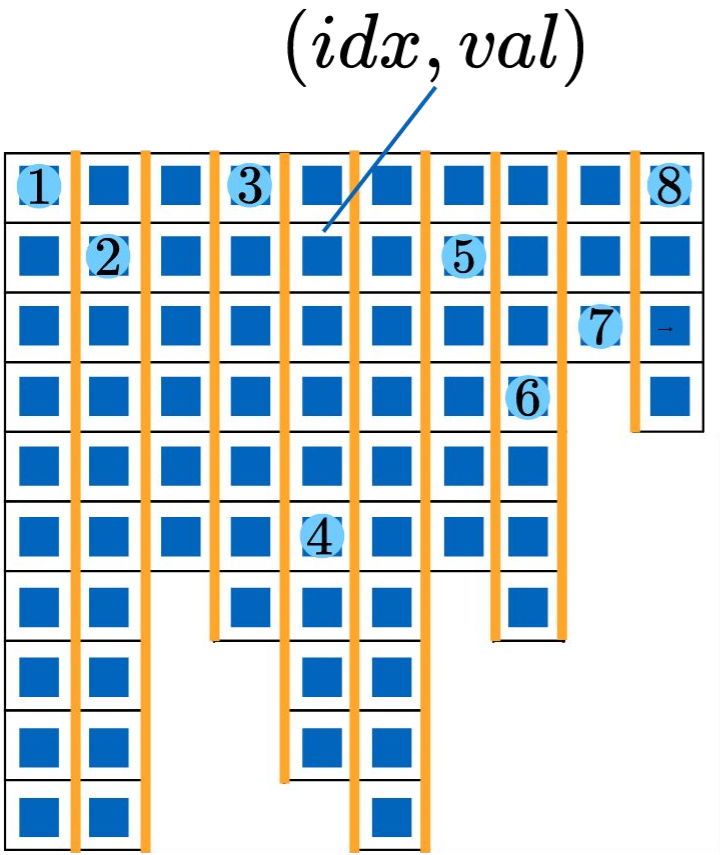


Ellpack
(*angepasst*)



(*linear ein einem Array gespeichert*)

Lesen einer Zeile von A



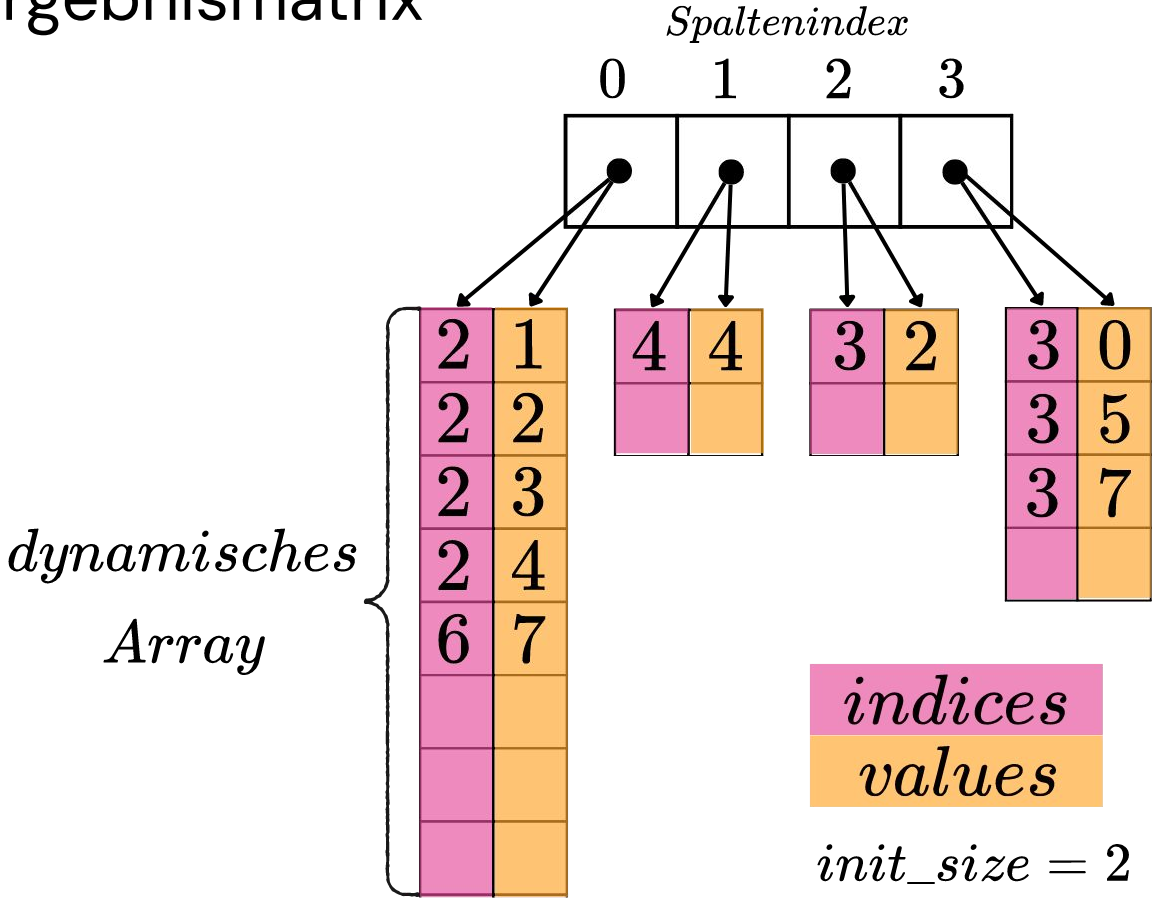
Problematisch im Cache

→ Row-Cache Datenstruktur

Speichern der Ergebnismatrix

result_mat

$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 0 \\ 2 & 0 & 3 & 0 \\ 2 & 0 & 0 & 0 \\ 2 & 4 & 0 & 0 \\ 0 & 0 & 0 & 5 \\ 0 & 0 & 0 & 0 \\ 6 & 0 & 0 & 8 \end{pmatrix}$$



Korrektheit

Testgenerator:

- Definition der Testcases über config-Array
- Generiert für jeden Testcase zwei Standardmatrizen kompatibler Größe für Multiplikation

Test:

- Multipliziert o.g. Matrizen nach einfachem dense Verfahren
  → vergleicht Ergebnis mit Ergebnis der Multiplikation im Ellpack-Format

Automatisch ausführbar über -t Flag

Performanzanalyse - Testumgebung und Spezifikationen

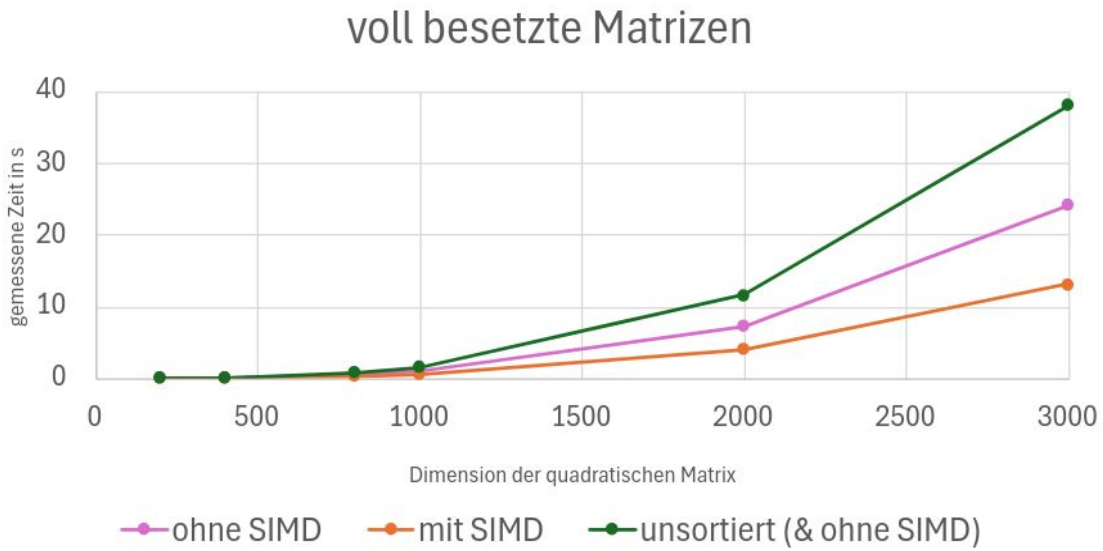
- Prozessor: Ryzen 5 4500U; Frequenz: ca. 1,4 GHz
- Arbeitsspeicher 7,4 GB
- Betriebssystem: Ubuntu 24.04 LTS
- Kernelversion: 6.8.0-38-generic
- Compiler: gcc 13.2.0
- Turboboost option und Hyperthreading deaktiviert
- Zeitmessung vor und nach Aufrufen der eigentlichen matr_mul Funktion (ohne IO)
- Messung dreimal -> Berechnung des Durchschnitts

Performanzanalyse

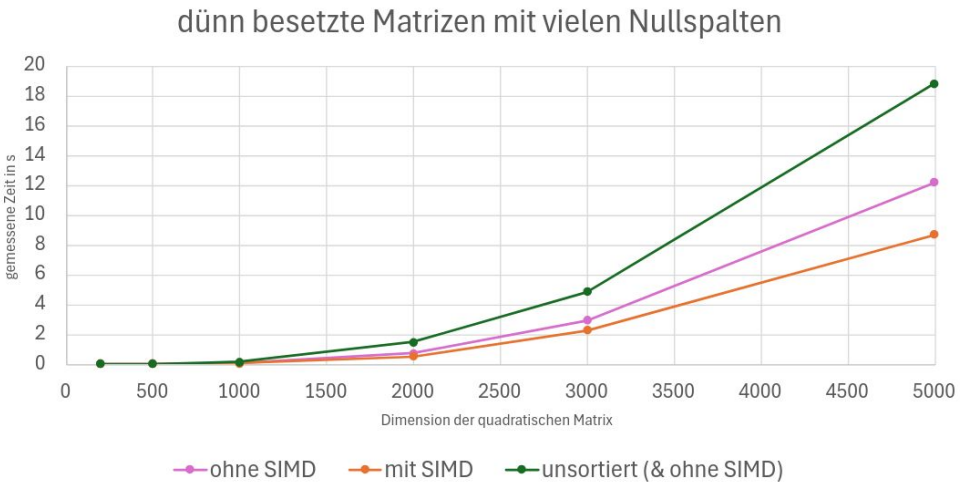
- Vergleich drei verschiedener Implementierungen:
    - Algorithmus mit Annahme, dass Indizes und Werte in Ellpack nach aufsteigendem Spaltenindex sortiert sind
    - Algorithmus ohne o.g. Annahme → jedes mal Array voll durchsuchen
    - Algorithmus mit Annahme und SIMD-Arithmetik für Multiplikation und Addition des Skalarprodukt
- Eingabe Unterschiede: dense vs. sparse (viele & wenige Nullzeilen)

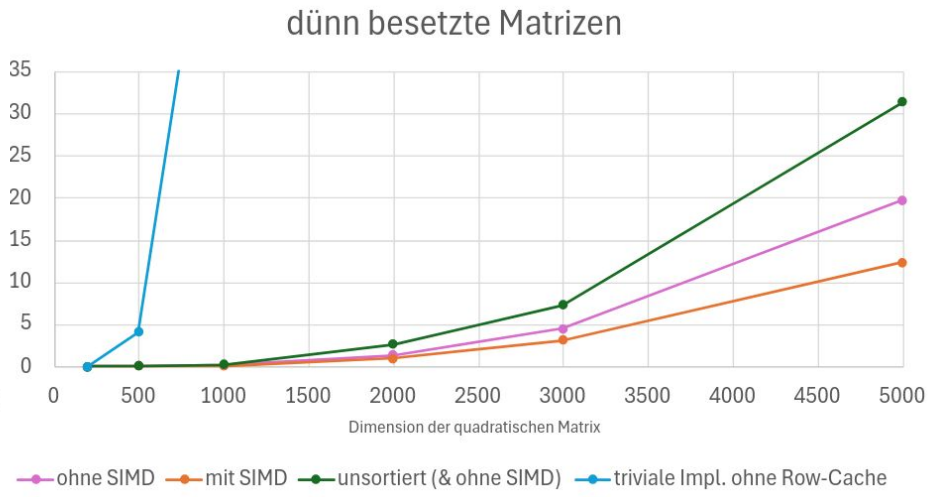
Performanzanalyse

- deutliche Performancesteigerung mit SIMD
- Grund: viele float-Multiplikationen im Vergleich zu Rowsprüngen
- Logikoptimierung: row cache nicht immer neu zu durchsuchen wirksam
- voll: ca. 80% nicht-null Einträge



Performanzanalyse





→ Row Cache sehr effizient, SIMD Optimierungsfaktor wächst mit Matrixgröße,

Zusammenfassung

Mögliche Optimierung die nicht umgesetzt wurden:

- Cache-misses im Row-Cache reduzieren
- mehr Fokus auf IO-Optimierung

Überraschendes Ergebnis:

- SIMD in unserer Implementierung auch für dünnbesetzte Matrizen sinnvoll