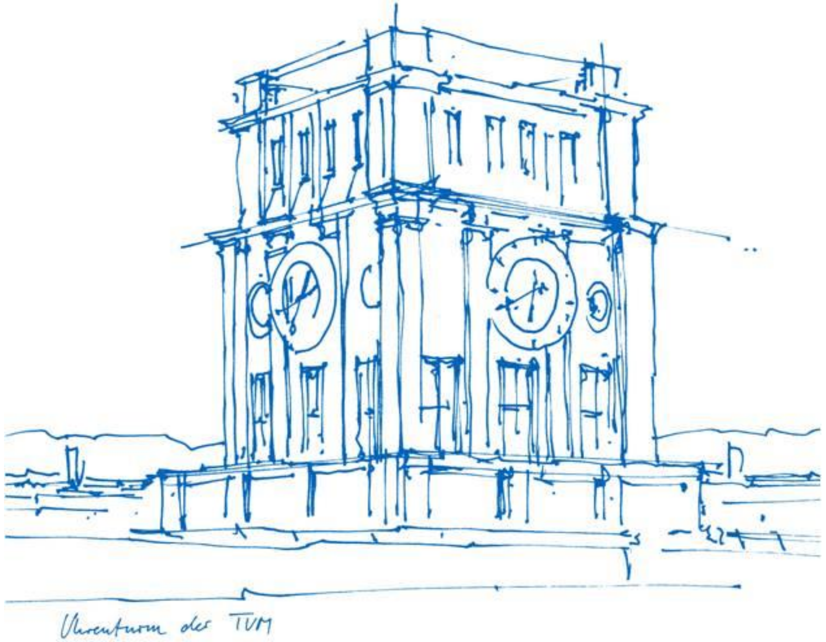
Grundlagenpraktikum: Rechnerarchitektur

Technische Universität München

TUM School of Computation, Information and Technology

Prof. Dr. rer. nat. Martin Schulz

München, 22.08.2024



Multiplikation dünnbesetzter Matrizen: CSC

Felipe Escallon, Jakob Friedrich, Alejandro Tellez

München, 22.08.2024

Inhaltsübersicht

- Matrixmultiplikation
  - Multiplikation vollbesetzter Matrizen
  - Compressed Sparse Column (CSC) – Format
  - Vor- und Nachteile des CSC-Formats

- Parsing und I/O
  - Datenstrukturen für Matrizen
  - Korrektheit
- Implementierung der Matrixmultiplikation
  - Skalarprodukte
  - Multiplikation mit transponierter Matrix

- Performanzanalyse

Matrixmultiplikation

Felipe Escallon, Alejandro Tellez, Jakob Friedrich

München, 22.08.2024

Multiplikation vollbesetzter Matrizen

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

Spaltenzahl von A muss mit der Zeilenzahl von B übereinstimmen

Beispiel:

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 3 & 0 & 4 \end{pmatrix} * \begin{pmatrix} 0 & 0 & 0 \\ 6 & 0 & 0 \\ 0 & 0 & 7 \end{pmatrix} = \begin{pmatrix} 0*0+0*6+0*0 & 0*0+0*0+0*0 & 0*0+0*0+0*7 \\ 0*0+0*6+0*0 & 0*0+0*0+0*0 & 0*0+0*0+0*7 \\ 3*0+0*6+4*0 & 3*0+0*0+4*0 & 3*0+0*0+4*7 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 28 \end{pmatrix}$$

Viele unnötige Berechnungen! $\mathrm{O}(\mathrm{n}^3)$

Compressed Sparse Column (CSC) - Format

Gespeicherte Werte:

- `rows, columns` - Dimensionen
- `Values` - Werte-Array
- `rowInd` - Zeilen-Indizes
- `colPtr` - Pointer auf die Einträge im Werte-Array, an denen neue Spalte startet

Beispiel:

$$\begin{pmatrix} 5 & 0 & 0 & 0 \\ 0 & 6 & 1 & 0 \\ 0.5 & 0 & 0 & 0 \\ 0 & 0 & 0 & 3 \end{pmatrix}$$

`rows, columns` = 4, 4

`values` = 5, 0.5, 6, 1, 3

`rowInd` = 0, 2, 1, 1, 3

`colPtr` = 0, 2, 3, 4, 5

Vor- und Nachteile des CSC-Formats

Vorteile:

- Speicherverbrauch
- Speicherzugriffe
- Spaltenoperationen

Nachteile:

- Ineffiziente Zeilenoperationen
- Eventueller Konvertierungs-Overhead
- Verlust der Symmetrie der Matrix (schwieriger, sinnvolle SIMD-Optimierungen zu implementieren)

Parsing und I/O

Felipe Escallon, Alejandro Tellez, Jakob Friedrich

München, 22.08.2024

Datenstrukturen für Matrizen

- Daten speichern
- Erleichtung der transponierung
    - `colIndices` generieren mit `get_col_indices`

```
struct cscMatrixTranspose {
    uint64_t rows;
    uint64_t columns;
    uint64_t valueCount;
    float* values;
    uint64_t* rowIndices;
    uint64_t* colIndices;
    uint64_t* colPtr;
};
```

```
struct cscMatrix {
    uint64_t rows;
    uint64_t columns;
    uint64_t valueCount;
    float* values;
    uint64_t* rowIndices;
    uint64_t* colPtr;
};
```

Korrektheit

- Falsches Format
  - `Values` enthält ein oder mehrere Null Werte
  - Falshe anzahl an `rowIndices` oder `colPtr`

- Multiplikation nicht möglich
  - Unmögliche Dimensionen
  - Werte außerhalb des Bereichs

- Matrizen voller Nullen

```
LINE | CONTENT
1 | <noRows>,<noCols>
2 | <values>
3 | <row_indices>
4 | <col_ptr>
```

Beispiel:

```
4,4
5,0.5,6,1,3
0,2,1,1,3
0,2,3,4,5
```

Implementierung der Matrixmultiplikation

Felipe Escallon, Alejandro Tellez, Jakob Friedrich

München, 22.08.2024

Erstes Problem: Speicherallokation der Ergebniswerte

Seien A, B Matrizen mit Dimensionen $n \times m$ bzw. $m \times k$.

- Worst-Case für Ergebnis AB: $n \times k$ nicht-null Elemente (100% Dichte)

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

- Größe erst nach Multiplikation bekannt
- `malloc(n*k*sizeof(x))` für Elementgröße x nicht optimal
  - Overflows
  - Evtl. können n*k Elemente zwar nicht allokiert werden, aber die tatsächliche Anzahl an Ergebniswerten schon
    - False Positives

Erstes Problem: Speicherallokation der Ergebniswerte

Seien A, B Matrizen mit Dimensionen $n \times m$ bzw. $m \times k$.

- Idee: Bessere Abschätzung
- Bspw.: Wenn A eine Nullzeile hat, wird die Ergebniszeile vom gleichen Index auch lauter Nullen haben
  - Analog für B und Nullspalten

$$\begin{pmatrix} 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 5 & 5 & 5 & 0 & 5 \\ 5 & 5 & 5 & 0 & 5 \\ 5 & 5 & 5 & 0 & 5 \\ 0 & 0 & 0 & 0 & 0 \\ 5 & 5 & 5 & 0 & 5 \end{pmatrix}$$

  - Falls A n' Nullzeilen und B k' Nullspalten hat, allokiere $(n - n') \times (k - k')$ Elemente
    - Overflows und False Positives zwar unwahrscheinlicher, aber immer noch anwesend
    - Neues Problem: Overhead von Berechnung von Nullzeilen /-spalten

Erstes Problem: Speicherallokation der Ergebniswerte

Seien A, B Matrizen mit Dimensionen $n \times m$ bzw. $m \times k$.

- Weitere Möglichkeit: berechne Anzahl Einträge, die nach Multiplikation ungleich 0 wären
  - Fast so teuer wie tatsächliche Multiplikation
- Lösung: dynamische Reallokierung
  - Trotz Einfügen von Werten im Worst-Case linear, amortisiert immer noch konstant
- Zusätzliche Optimierung: Kleine Anfangsabschätzung

Erstes Problem: Speicherallokation der Ergebniswerte

Seien A, B Matrizen mit Dimensionen $n \times m$ bzw. $m \times k$.

- Weitere Möglichkeit: berechne Anzahl Einträge, die nach Multiplikation ungleich 0 wären
  - Fast so teuer wie tatsächliche Multiplikation
- Lösung: dynamische Reallokierung
  - Trotz Einfügen von Werten im Worst-Case linear, amortisiert immer noch konstant
- Zusätzliche Optimierung: Kleine Anfangsabschätzung
- Verdopple Arrays, bis Größe $n \times k$ erricht wird

```
uint64_t extend_vector(float** values, uint64_t** indices, uint64_t current,
                       uint64_t max) {
    if (current == max) {
        errno = ENOMEM;
        return 0;
    }
    uint64_t doubleSize;
    uint64_t newSize;
    int of = __builtin_umull_overflow(2, current,
            &doubleSize);
    newSize = of ? max : doubleSize;
    *values = reallocarray(*values, newSize, sizeof(float));
    *indices = reallocarray(*indices, newSize, sizeof(uint64_t));
    if (!values || !indices) return 0;
    return newSize;
}
```

Slicing von Spalten

- Anfang und Ende einer Spalte in $O(1)$ berechenbar
- `i`-te Spalte: values und row indices von `colPtr[i]` bis `colPtr[i + 1]` exkl.
- Anzahl Elemente in Spalte `i`: `colPtr[i + 1] – colPtr[i]`

Slicing von Spalten: Beispiel

$$
\begin{array}{ccccc}
\text{Spaltenindizes} & 0 & 1 & 2 & 3 \\
 & \multicolumn{4}{c}{\begin{pmatrix} 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 1 & 0 & 0.3 & 0 \\ 0 & 1.6 & 0 & 2 \end{pmatrix}}
\end{array}
$$

```
values: 7, 1, 1.6, 4, 0.3, 2,
rowInd: 0, 2, 3, 1, 2, 3,
colPtr: 0, 2, 3, 5, 6
```

Seien `v`, `rI` die values und row indices in der gesuchten Spalte.

Gesucht: Spalte 2

`colPtr[2]`: 3

```
colPtr[2 + 1] = colPtr[3]: 5
v.Length = rI.length = colPtr[3] – colPtr[2] = 2
v = {values[3], values[4]} = {4, 0.3}
rI = {rowInd[3], rowInd[4]} = {1, 2}
```

Skalarprodukte mit Slicing

- `values` von oben nach unten und von links nach rechts sortiert
- `rowInd` innerhalb einer Spalte aufsteigend sortiert
- Für Spaltenvektoren `a,b`: Multiplikation der Einträge in Index `i` findet genau dann statt, wenn sowohl `a.rowInd` als auch `b.rowInd i` enthalten
  - Gleichzeitige Traversierung beider Arrays

Skalarprodukte mit Slicing

```
float scalar_prod(const float* aVec, const float* bVec, const uint64_t* aInd,
                  const uint64_t* bInd, uint64_t aLen, uint64_t bLen) {

    float result = 0;

    for (uint64_t i = 0, j = 0; i < aLen && j < bLen;) {
        if (aInd[i] < bInd[j]) {
            while (i < aLen && aInd[i] < bInd[j]) ++i;

            if (i >= aLen) break;

        } else {
            while (j < bLen && bInd[j] < aInd[i]) ++j;

            if(j >= bLen) break;
        }

        if (aInd[i] == bInd[j]) {
            result += aVec[i++] * bVec[j++];
            }
    }
    return result;
}
```

```
i: 0
j: 0

a.rowInd: {0, 2, 3, 6, 8, 9, 10, 14}
b.rowInd: {0, 1, 3, 4, 5, 6, 14}

a.values: {3.1, 8, 4, 1, 2.1, 6, 0.1, 2}
b.values: {2.5, 12, 5, 2, 9.4, 0.25, 7}

result: 0 + 3.1*2.5
```

Skalarprodukte mit Slicing

```
float scalar_prod(const float* aVec, const float* bVec, const uint64_t* aInd,
                  const uint64_t* bInd, uint64_t aLen, uint64_t bLen) {

    float result = 0;

    for (uint64_t i = 0, j = 0; i < aLen && j < bLen;) {
        if (aInd[i] < bInd[j]) {
            while (i < aLen && aInd[i] < bInd[j]) ++i;

            if (i >= aLen) break;

        } else {
            while (j < bLen && bInd[j] < aInd[i]) ++j;

            if(j >= bLen) break;
        }

        if (aInd[i] == bInd[j]) {
            result += aVec[i++] * bVec[j++];
            }
    }
    return result;
}
```

```
i: 1
j: 1

a.rowInd: {0, 2, 3, 6, 8, 9, 10, 14}
b.rowInd: {0, 1, 3, 4, 5, 6, 14}

a.values: {3.1, 8, 4, 1, 2.1, 6, 0.1, 2}
b.values: {2.5, 12, 5, 2, 9.4, 0.25, 7}

result: 7.75
```

Skalarprodukte mit Slicing

```
float scalar_prod(const float* aVec, const float* bVec, const uint64_t* aInd,
                  const uint64_t* bInd, uint64_t aLen, uint64_t bLen) {

    float result = 0;

    for (uint64_t i = 0, j = 0; i < aLen && j < bLen;) {
        if (aInd[i] < bInd[j]) {
            while (i < aLen && aInd[i] < bInd[j]) ++i;

            if (i >= aLen) break;

        } else {
            while (j < bLen && bInd[j] < aInd[i]) ++j;

            if(j >= bLen) break;
        }

        if (aInd[i] == bInd[j]) {
            result += aVec[i++] * bVec[j++];
            }
    }
    return result;
}
```

```
i: 1
j: 2

a.rowInd: {0, 2, 3, 6, 8, 9, 10, 14}
b.rowInd: {0, 1, 3, 4, 5, 6, 14}

a.values: {3.1, 8, 4, 1, 2.1, 6, 0.1, 2}
b.values: {2.5, 12, 5, 2, 9.4, 0.25, 7}

result: 7.75
```

Skalarprodukte mit Slicing

```
float scalar_prod(const float* aVec, const float* bVec, const uint64_t* aInd,
                  const uint64_t* bInd, uint64_t aLen, uint64_t bLen) {

    float result = 0;

    for (uint64_t i = 0, j = 0; i < aLen && j < bLen;) {
        if (aInd[i] < bInd[j]) {
            while (i < aLen && aInd[i] < bInd[j]) ++i;

            if (i >= aLen) break;

        } else {
            while (j < bLen && bInd[j] < aInd[i]) ++j;

            if(j >= bLen) break;
        }

        if (aInd[i] == bInd[j]) {
            result += aVec[i++] * bVec[j++];
            }
    }
    return result;
}
```

```
i: 2
j: 2

a.rowInd: {0, 2, 3, 6, 8, 9, 10, 14}
b.rowInd: {0, 1, 3, 4, 5, 6, 14}

a.values: {3.1, 8, 4, 1, 2.1, 6, 0.1, 2}
b.values: {2.5, 12, 5, 2, 9.4, 0.25, 7}

result: 7.75 + 4*5
```

Skalarprodukte mit Slicing

```
float scalar_prod(const float* aVec, const float* bVec, const uint64_t* aInd,
                  const uint64_t* bInd, uint64_t aLen, uint64_t bLen) {

    float result = 0;

    for (uint64_t i = 0, j = 0; i < aLen && j < bLen;) {
        if (aInd[i] < bInd[j]) {
            while (i < aLen && aInd[i] < bInd[j]) ++i;

            if (i >= aLen) break;

        } else {
            while (j < bLen && bInd[j] < aInd[i]) ++j;

            if(j >= bLen) break;
        }

        if (aInd[i] == bInd[j]) {
            result += aVec[i++] * bVec[j++];
            }
    }
    return result;
}
```

```
i: 3
j: 3

a.rowInd: {0, 2, 3, 6, 8, 9, 10, 14}
b.rowInd: {0, 1, 3, 4, 5, 6, 14}

a.values: {3.1, 8, 4, 1, 2.1, 6, 0.1, 2}
b.values: {2.5, 12, 5, 2, 9.4, 0.25, 7}

result: 27.75
```

Skalarprodukte mit Slicing

```
float scalar_prod(const float* aVec, const float* bVec, const uint64_t* aInd,
                  const uint64_t* bInd, uint64_t aLen, uint64_t bLen) {

    float result = 0;

    for (uint64_t i = 0, j = 0; i < aLen && j < bLen;) {
        if (aInd[i] < bInd[j]) {
            while (i < aLen && aInd[i] < bInd[j]) ++i;

            if (i >= aLen) break;

        } else {
            while (j < bLen && bInd[j] < aInd[i]) ++j;

            if(j >= bLen) break;
        }

        if (aInd[i] == bInd[j]) {
            result += aVec[i++] * bVec[j++];
            }
    }
    return result;
}
```

```
i: 3
j: 4

a.rowInd: {0, 2, 3, 6, 8, 9, 10, 14}
b.rowInd: {0, 1, 3, 4, 5, 6, 14}

a.values: {3.1, 8, 4, 1, 2.1, 6, 0.1, 2}
b.values: {2.5, 12, 5, 2, 9.4, 0.25, 7}

result: 27.75
```

Skalarprodukte mit Slicing

```
float scalar_prod(const float* aVec, const float* bVec, const uint64_t* aInd,
                  const uint64_t* bInd, uint64_t aLen, uint64_t bLen) {

    float result = 0;

    for (uint64_t i = 0, j = 0; i < aLen && j < bLen;) {
        if (aInd[i] < bInd[j]) {
            while (i < aLen && aInd[i] < bInd[j]) ++i;

            if (i >= aLen) break;

        } else {
            while (j < bLen && bInd[j] < aInd[i]) ++j;

            if(j >= bLen) break;
        }

        if (aInd[i] == bInd[j]) {
            result += aVec[i++] * bVec[j++];
            }
    }
    return result;
}
```

```
i: 3
j: 5

a.rowInd: {0, 2, 3, 6, 8, 9, 10, 14}
b.rowInd: {0, 1, 3, 4, 5, 6, 14}

a.values: {3.1, 8, 4, 1, 2.1, 6, 0.1, 2}
b.values: {2.5, 12, 5, 2, 9.4, 0.25, 7}

result: 27.75 + 1*0.25
```

Skalarprodukte mit Slicing

```
float scalar_prod(const float* aVec, const float* bVec, const uint64_t* aInd,
                  const uint64_t* bInd, uint64_t aLen, uint64_t bLen) {

    float result = 0;

    for (uint64_t i = 0, j = 0; i < aLen && j < bLen;) {
        if (aInd[i] < bInd[j]) {
            while (i < aLen && aInd[i] < bInd[j]) ++i;

            if (i >= aLen) break;

        } else {
            while (j < bLen && bInd[j] < aInd[i]) ++j;

            if(j >= bLen) break;
        }

        if (aInd[i] == bInd[j]) {
            result += aVec[i++] * bVec[j++];
            }
    }
    return result;
}
```

```
i: 4
j: 6

a.rowInd: {0, 2, 3, 6, 8, 9, 10, 14}
b.rowInd: {0, 1, 3, 4, 5, 6, 14}

a.values: {3.1, 8, 4, 1, 2.1, 6, 0.1, 2}
b.values: {2.5, 12, 5, 2, 9.4, 0.25, 7}

result: 28
```

Skalarprodukte mit Slicing

```
float scalar_prod(const float* aVec, const float* bVec, const uint64_t* aInd,
                  const uint64_t* bInd, uint64_t aLen, uint64_t bLen) {

    float result = 0;

    for (uint64_t i = 0, j = 0; i < aLen && j < bLen;) {
        if (aInd[i] < bInd[j]) {
            while (i < aLen && aInd[i] < bInd[j]) ++i;

            if (i >= aLen) break;

        } else {
            while (j < bLen && bInd[j] < aInd[i]) ++j;

            if(j >= bLen) break;
        }

        if (aInd[i] == bInd[j]) {
            result += aVec[i++] * bVec[j++];
            }
    }
    return result;
}
```

```
i: 5
j: 6

a.rowInd: {0, 2, 3, 6, 8, 9, 10, 14}
b.rowInd: {0, 1, 3, 4, 5, 6, 14}

a.values: {3.1, 8, 4, 1, 2.1, 6, 0.1, 2}
b.values: {2.5, 12, 5, 2, 9.4, 0.25, 7}

result: 28
```

Skalarprodukte mit Slicing

```
float scalar_prod(const float* aVec, const float* bVec, const uint64_t* aInd,
                  const uint64_t* bInd, uint64_t aLen, uint64_t bLen) {

    float result = 0;

    for (uint64_t i = 0, j = 0; i < aLen && j < bLen;) {
        if (aInd[i] < bInd[j]) {
            while (i < aLen && aInd[i] < bInd[j]) ++i;

            if (i >= aLen) break;

        } else {
            while (j < bLen && bInd[j] < aInd[i]) ++j;

            if(j >= bLen) break;
        }

        if (aInd[i] == bInd[j]) {
            result += aVec[i++] * bVec[j++];
            }
    }
    return result;
}
```

```
i: 6
j: 6

a.rowInd: {0, 2, 3, 6, 8, 9, 10, 14}
b.rowInd: {0, 1, 3, 4, 5, 6, 14}


a.values: {3.1, 8, 4, 1, 2.1, 6, 0.1, 2}
b.values: {2.5, 12, 5, 2, 9.4, 0.25, 7}

result: 28
```

Skalarprodukte mit Slicing

```
float scalar_prod(const float* aVec, const float* bVec, const uint64_t* aInd,
                  const uint64_t* bInd, uint64_t aLen, uint64_t bLen) {

    float result = 0;

    for (uint64_t i = 0, j = 0; i < aLen && j < bLen;) {
        if (aInd[i] < bInd[j]) {
            while (i < aLen && aInd[i] < bInd[j]) ++i;

            if (i >= aLen) break;

        } else {
            while (j < bLen && bInd[j] < aInd[i]) ++j;

            if(j >= bLen) break;
        }

        if (aInd[i] == bInd[j]) {
            result += aVec[i++] * bVec[j++];
            }
    }
    return result;
}
```

```
i: 7
j: 6

a.rowInd: {0, 2, 3, 6, 8, 9, 10, 14}
b.rowInd: {0, 1, 3, 4, 5, 6, 14}

a.values: {3.1, 8, 4, 1, 2.1, 6, 0.1, 2}
b.values: {2.5, 12, 5, 2, 9.4, 0.25, 7}

result: 28 + 14
```

Skalarprodukte mit Slicing

```
float scalar_prod(const float* aVec, const float* bVec, const uint64_t* aInd,
                  const uint64_t* bInd, uint64_t aLen, uint64_t bLen) {

    float result = 0;

    for (uint64_t i = 0, j = 0; i < aLen && j < bLen;) {
        if (aInd[i] < bInd[j]) {
            while (i < aLen && aInd[i] < bInd[j]) ++i;

            if (i >= aLen) break;

        } else {
            while (j < bLen && bInd[j] < aInd[i]) ++j;

            if(j >= bLen) break;
        }

        if (aInd[i] == bInd[j]) {
            result += aVec[i++] * bVec[j++];
            }
    }
    return result;
}
```

```
i: 8
j: 7

a.rowInd: {0, 2, 3, 6, 8, 9, 10, 14}
b.rowInd: {0, 1, 3, 4, 5, 6, 14}

a.values: {3.1, 8, 4, 1, 2.1, 6, 0.1, 2}
b.values: {2.5, 12, 5, 2, 9.4, 0.25, 7}

result: 42
```

Naiver Ansatz

- Klassische Matrixmultiplikation
- Finde Zeilen von A und multipliziere mit Spalten von B
  - Spalten von B durch Slicing einfach zu finden
  - Skalarprodukt mit Slicing
- Innere Schleife iteriert über A, um Struktur von CSC zu folgen

$$\begin{pmatrix} 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 1 & 0 & 0.3 & 0 \\ 0 & 1.6 & 0 & 2 \end{pmatrix} \begin{pmatrix} 0 & 4 & 0.8 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 2.3 & 0 & 3 & 0 & 7.7 \end{pmatrix} = \begin{pmatrix} 0 & & & & \\ & & & & \\ & & & & \\ & & & & \end{pmatrix}$$

```
A
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
```

```
B
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
```

```
Result
values:
rowInd:
```

Naiver Ansatz

- Klassische Matrixmultiplikation
- Finde Zeilen von A und multipliziere mit Spalten von B
  - Spalten von B durch Slicing einfach zu finden
  - Skalarprodukt mit Slicing
- Innere Schleife iteriert über A, um Struktur von CSC zu folgen

$$\begin{pmatrix} 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 1 & 0 & 0.3 & 0 \\ 0 & 1.6 & 0 & 2 \end{pmatrix} \begin{pmatrix} 0 & 4 & 0.8 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 2.3 & 0 & 3 & 0 & 7.7 \end{pmatrix} = \begin{pmatrix} 0 & & & & \\ 0 & & & & \\ & & & & \\ & & & & \end{pmatrix}$$

```
A
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
```

```
B
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
```

```
Result
values:
rowInd:
```

Naiver Ansatz

- Klassische Matrixmultiplikation
- Finde Zeilen von A und multipliziere mit Spalten von B
  - Spalten von B durch Slicing einfach zu finden
  - Skalarprodukt mit Slicing
- Innere Schleife iteriert über A, um Struktur von CSC zu folgen

$$\begin{pmatrix} 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 1 & 0 & 0.3 & 0 \\ 0 & 1.6 & 0 & 2 \end{pmatrix} \begin{pmatrix} 0 & 4 & 0.8 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 2.3 & 0 & 3 & 0 & 7.7 \end{pmatrix} = \begin{pmatrix} 0 & & & & \\ 0 & & & & \\ 0 & & & & \\ & & & & \end{pmatrix}$$

```
A
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
```

```
B
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
```

```
Result
values:
rowInd:
```

Naiver Ansatz

- Klassische Matrixmultiplikation
- Finde Zeilen von A und multipliziere mit Spalten von B
  - Spalten von B durch Slicing einfach zu finden
  - Skalarprodukt mit Slicing
- Innere Schleife iteriert über A, um Struktur von CSC zu folgen

$$\begin{pmatrix} 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 1 & 0 & 0.3 & 0 \\ 0 & 1.6 & 0 & 2 \end{pmatrix} \begin{pmatrix} 0 & 4 & 0.8 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 2.3 & 0 & 3 & 0 & 7.7 \end{pmatrix} = \begin{pmatrix} 0 & & & & \\ 0 & & & & \\ 0 & & & & \\ 4.6 & & & & \end{pmatrix}$$

```
A
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
```

```
B
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
```

```
Result
values: 4.6
rowInd: 3
```

Naiver Ansatz

- Klassische Matrixmultiplikation
- Finde Zeilen von A und multipliziere mit Spalten von B
  - Spalten von B durch Slicing einfach zu finden
  - Skalarprodukt mit Slicing
- Innere Schleife iteriert über A, um Struktur von CSC zu folgen

$$\begin{pmatrix} 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 1 & 0 & 0.3 & 0 \\ 0 & 1.6 & 0 & 2 \end{pmatrix} \begin{pmatrix} 0 & 4 & 0.8 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 2.3 & 0 & 3 & 0 & 7.7 \end{pmatrix} = \begin{pmatrix} 0 & 28 & & & \\ 0 & & & & \\ 0 & & & & \\ 4.6 & & & & \end{pmatrix}$$

```
A
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
```

```
B
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
```

```
Result
values: 4.6, 28
rowInd: 3, 0
```

Naiver Ansatz

- Klassische Matrixmultiplikation
- Finde Zeilen von A und multipliziere mit Spalten von B
  - Spalten von B durch Slicing einfach zu finden
  - Skalarprodukt mit Slicing
- Innere Schleife iteriert über A, um Struktur von CSC zu folgen

$$\begin{pmatrix} 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 1 & 0 & 0.3 & 0 \\ 0 & 1.6 & 0 & 2 \end{pmatrix} \begin{pmatrix} 0 & 4 & 0.8 & 0 & 0 \\ 0 & 9 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 2.3 & 0 & 3 & 0 & 7.7 \end{pmatrix} = \begin{pmatrix} 0 & 28 & 5.6 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \\ 0 & 4 & 0.8 & 0.3 & 0 \\ 4.6 & 14.4 & 6 & 0 & 15.4 \end{pmatrix}$$

```
A
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
```

```
B
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
```

```
Result
values: 4.6, 28, 4, 14.4, 5.6, 0.8, 6, 4, 0.3, 15.4
rowInd: 3, 0, 2, 3, 0, 2, 3, 1, 2, 3
```

Naiver Ansatz: Slicing von B

- Spalten von B können in linearer Zeit gefunden werden

```
void get_col_vector (float* values, uint64_t* indices, uint64_t len,
                     float* vector, uint64_t* subIndices, uint64_t start) {
    for (uint64_t i = 0; i < len; ++i) {
        vector[i] = values[start+i];
        subIndices[i] = indices[start+i];
    }
}
```

Naiver Ansatz: Problem

- Zeilen von A nicht durch Slicing berechenbar
- Idee: Für Zeile `r`, finde alle Indizes `i` in A mit

  `rowInd[i] = r`
  - `rowInd` unsortiert ➔ lineare Suche auf alle Elemente der Matrix

```c
struct cscRow get_row_vector(const struct cscMatrix* m, uint64_t index) {
    struct cscRow row = { .values = malloc(sizeof(float)),
        .colIndices = malloc(sizeof(uint64_t))
    };
    uint64_t size = 1;
    if (!row.values || !row.colIndices) {
        free(row.values);
        free(row.colIndices);
        row.values = 0;
        row.colIndices = 0;
        errno = ENOMEM;
        return row;
    }
    for (uint64_t i = 0; i < m->columns; ++i) {
        for (uint64_t j = m->colPtr[i]; j < m->colPtr[i + 1]; ++j) {
            if (m->rowIndices[j] != index) continue;
            if (row.valueCount >= size) {
                size = extend_vector(&row.values, &row.colIndices, row.valueCount,
                                     m->columns);
                if (!size) {
                    free(row.values);
                    free(row.colIndices);
                    row.values = 0;
                    row.colIndices = 0;
                    errno = ENOMEM;
                    return row;
                }
            }

            row.values[row.valueCount] = m->values[j];
            row.colIndices[row.valueCount++] = i;
            break;
        }
    }
    return row;
}
```

Naiver Ansatz: Problem

- Für Ergebnismatrix mit n Einträgen, i.A. n Aufrufe von getRowVector
  - Quadratische Zeit ➔ Hoher Zeitaufwand

```
get_col_vector(csB->values, csB->rowIndices, bLen, bVec, bInd, start);

 for (uint64_t i = 0; i < csA->rows; ++i) {


     errno = 0;
     struct cscRow row = get_row_vector(csA, i);
     if (errno) {
         perror("Error getting row vector from a");
         free(bVec);
         free(bInd);
         freeResultPtrs(csResult);
         return;
     }

     float entry = scalar_prod(row.values, bVec, row.colIndices, bInd,
                               row.valueCount, bLen);
```

Naiver Ansatz: Problem

- Idee: Für Matrizen A, B Skalarprodukte zwischen Zeilen von A und Spalten von B äquivalent zu Skalarprodukte zwischen Spalten von A^T und Spalten von B

```c
get_col_vector(csB->values, csB->rowIndices, bLen, bVec, bInd, start);

 for (uint64_t i = 0; i < csA->rows; ++i) {


     errno = 0;
     struct cscRow row = get_row_vector(csA, i);
     if (errno) {
         perror("Error getting row vector from a");
         free(bVec);
         free(bInd);
         freeResultPtrs(csResult);
         return;
     }

     float entry = scalar_prod(row.values, bVec, row.colIndices, bInd,
                               row.valueCount, bLen);
```

Transponieren einer CSC Matrix

- `colPtr` gibt Spaltenindizes an: Eintrag liegt zwischen Indizes `colPtr[i]` und `colPtr[i + 1]`, gdw. er in Spalte `i` liegt
- Spaltenindizes in CSC Matrix aufsteigend sortiert

$$\begin{pmatrix} 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 1 & 0 & 0.3 & 0 \\ 0 & 1.6 & 0 & 2 \end{pmatrix}$$

```
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
colPtr: 0, 2, 3, 5, 6
colInd: 0, 0, 1, 2, 2, 3
```

Transponierung einer CSC Matrix

- `colPtr` gibt Spaltenindizes an: Eintrag liegt zwischen Indizes `colPtr[i]` und `colPtr[i + 1]`, gdw. er in Spalte `i` liegt
- Spaltenindizes `colInd` in CSC Matrix aufsteigend sortiert
- `rowInd` in ursprünglicher Matrix äquivalent zu `colInd` in transponierter Matrix
- `values` von oben nach unten und von links nach rechts sortiert ➔ Elemente mit gleichem `rowInd` haben dieselbe Reihenfolge wie im Zeilenvektor

$$\begin{pmatrix} 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 1 & 0 & 0.3 & 0 \\ 0 & 1.6 & 0 & 2 \end{pmatrix} \qquad \begin{pmatrix} 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 1 & 0 & 0.3 & 0 \\ 0 & 1.6 & 0 & 2 \end{pmatrix}^T = \begin{pmatrix} 7 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1.6 \\ 0 & 4 & 0.3 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

```
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
colPtr: 0, 2, 3, 5, 6
colInd: 0, 0, 1, 2, 2, 3
```

```
values: 7, 4, 1, 0.3, 1.6, 2
rowInd: 0, 2, 0, 2, 1, 3
colPtr: 0, 1, 2, 4, 6
colInd: 0, 1, 2, 2, 3, 3
```

Transponierung einer CSC Matrix

- Idee: sortiere `values`, `rowInd und colInd` nach aufsteigender Sortierung von `rowInd`
  - Stabiler Sortieralgorithmus nötig
- `colInd` sind neue `rowInd`, neuer `colPtr` kann durch aufsteigend sortierte, ehemalige `rowInd in` $O(|values|)$ berechnet werden

$$\begin{pmatrix} 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 1 & 0 & 0.3 & 0 \\ 0 & 1.6 & 0 & 2 \end{pmatrix} \qquad \begin{pmatrix} 7 & 0 & 0 & 0 \\ 0 & 0 & 4 & 0 \\ 1 & 0 & 0.3 & 0 \\ 0 & 1.6 & 0 & 2 \end{pmatrix}^T = \begin{pmatrix} 7 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1.6 \\ 0 & 4 & 0.3 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix}$$

```
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2, 3, 1, 2, 3
colPtr: 0, 2, 3, 5, 6
colInd: 0, 0, 1, 2, 2, 3
```

```
values: 7, 4, 1, 0.3, 1.6, 2
rowInd: 0, 2, 0, 2, 1, 3
colPtr: 0, 1, 2, 4, 6
colInd: 0, 1, 2, 2, 3, 3
```

Sortierung der Arrays

Anforderungen: stabil, möglichst gute Laufzeit

RadixSort und MergeSort kommen in Frage
Laufzeit: $O(kn)$ für Keylength k bzw. $O(n\,log\,n)$
k hier: Dezimalstellen von $\max_{0 \le i < |rowInd|} rowInd[i]$

MergeSort: evtl. schwierig zu implementieren, hoher Speicheraufwand

RadixSort: aufsteigende Sortierung der `rowInd` und anhand dieser Reihenfolge `values` und `colInd` sortieren
Implementierung: Buckets entsprechen verschiedenen `rowInd`

Matrix A

```
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2,   3, 1,   2, 3
colInd: 0, 0,   1, 2,   2, 3
colPtr: 0, 2, 3, 5, 6
```

Matrix A^T

```
values: 0, 0, 0, 0, 0, 0
rowInd: 0, 0, 0, 0, 0, 0
colInd: 0, 0, 0, 0, 0, 0
colPtr: Berechnung folgt
```

RadixSort

aufsteigende Sortierung der `rowInd` und anhand dieser Reihenfolge `values` und `colInd` sortieren
Implementierung: Buckets entsprechen verschiedenen `rowInd`

Matrix A

```
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2,   3, 1,   2, 3
colInd: 0, 0,   1, 2,   2, 3
colPtr: 0, 2, 3, 5, 6
```

Matrix A^T

```
values: 7, 0, 0, 0, 0, 0
rowInd: 0, 0, 0, 0, 0, 0
colInd: 0, 0, 0, 0, 0, 0
colPtr: Berechnung folgt
```

RadixSort

aufsteigende Sortierung der `rowInd` und anhand dieser Reihenfolge `values` und `colInd` sortieren
Implementierung: Buckets entsprechen verschiedenen `rowInd`

Matrix A

```
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2,   3, 1,   2, 3
colInd: 0, 0,   1, 2,   2, 3
colPtr: 0, 2, 3, 5, 6
```

Matrix A^T

```
values: 7, 0, 1, 0, 0, 0
rowInd: 0, 0, 0, 0, 0, 0
colInd: 0, 0, 2, 0, 0, 0
colPtr: Berechnung folgt
```

RadixSort

aufsteigende Sortierung der `rowInd` und anhand dieser Reihenfolge `values` und `colInd` sortieren
Implementierung: Buckets entsprechen verschiedenen `rowInd`

Matrix A

```
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2,   3, 1,   2, 3
colInd: 0, 0,   1, 2,   2, 3
colPtr: 0, 2, 3, 5, 6
```

Matrix A^T

```
values: 7, 0, 1, 0, 1.6, 0
rowInd: 0, 0, 0, 0,   1, 0
colInd: 0, 0, 2, 0,   3, 0
colPtr: Berechnung folgt
```

RadixSort

aufsteigende Sortierung der `rowInd` und anhand dieser Reihenfolge `values` und `colInd` sortieren
Implementierung: Buckets entsprechen verschiedenen `rowInd`

Matrix A

```
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2,   3, 1,   2, 3
colInd: 0, 0,   1, 2,   2, 3
colPtr: 0, 2, 3, 5, 6
```

Matrix A^T

```
values: 7, 4, 1, 0, 1.6, 0
rowInd: 0, 2, 0, 0,   1, 0
colInd: 0, 1, 2, 0,   3, 0
colPtr: Berechnung folgt
```

RadixSort

aufsteigende Sortierung der `rowInd` und anhand dieser Reihenfolge `values` und `colInd` sortieren
Implementierung: Buckets entsprechen verschiedenen `rowInd`

Matrix A

```
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2,   3, 1,   2, 3
colInd: 0, 0,   1, 2,   2, 3
colPtr: 0, 2, 3, 5, 6
```

Matrix A^T

```
values: 7, 4, 1, 0.3, 1.6, 0
rowInd: 0, 2, 0,   2,   1, 0
colInd: 0, 1, 2,   2,   3, 0
colPtr: Berechnung folgt
```

RadixSort

aufsteigende Sortierung der `rowInd` und anhand dieser Reihenfolge `values` und `colInd` sortieren
Implementierung: Buckets entsprechen verschiedenen `rowInd`

Matrix A

```
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2,   3, 1,   2, 3
colInd: 0, 0,   1, 2,   2, 3
colPtr: 0, 2, 3, 5, 6
```

Matrix A^T

```
values: 7, 4, 1, 0.3, 1.6, 2
rowInd: 0, 2, 0,   2,   1, 3
colInd: 0, 1, 2,   2,   3, 3
colPtr: Berechnung folgt
```

RadixSort: colPtr

`colPtr` anhand von `colInd` mit gleichen Indizes berechnen

Matrix A

```
values: 7, 1, 1.6, 4, 0.3, 2
rowInd: 0, 2,   3, 1,   2, 3
colInd: 0, 0,   1, 2,   2, 3
colPtr: 0, 2, 3, 5, 6
```

Matrix A^T

```
values: 7, 4, 1, 0.3, 1.6, 2
rowInd: 0, 2, 0,   2,   1, 3
colInd: 0, 1, 2,   2,   3, 3
colPtr: 0, 1, 2, 4, 6
```

Multiplikation mit Skalarprodukt zwischen Spalten

- Anfang und Ende beider Vektoren nun in $O(1)$ berechenbar
- Extrahiere Spalten von A und B in linearer Zeit bzgl. Spaltengröße und berechne Skalarprodukt

```c
get_col_vector(csB->values, csB->rowIndices, bLen, bVec, bInd, start);

for (uint64_t i = 0; i < csA->columns; ++i) {
    start = csA->colPtr[i];
    end = csA->colPtr[i+1];
    uint64_t aLen = end-start;
    if (!aLen) continue;

    // get the next A column
    float* aVec = malloc(aLen * sizeof(float));
    uint64_t* aInd = malloc(aLen * sizeof(uint64_t));
    if (!aVec || !aInd) {
        errno = ENOMEM;
        free(aVec);
        free(aInd);
        free(bVec);
        free(bInd);
        freeResultPtrs(csResult);
        fprintf(stderr,
            "\rError allocating memory for row %lu of matrix A.\n", i);
        return;
    }

    get_col_vector(csA->values, csA->rowIndices, aLen, aVec, aInd,
            start);
```

In-Place Skalarprodukt mit Slicing

- Idee: mit transponiertem Matrix unnötig, Spalte vor Berechnung zu finden
  - Auch auf Matrix B anwendbar
- Statt Skalarprodukt zwischen zwei ganze Arrays zu berechnen: zwischen zwei Unterarrays

```
float scalar_prod_in_place(const struct cscMatrix* a, const struct cscMatrix* b,
        uint64_t aStart, uint64_t aEnd, uint64_t bStart, uint64_t bEnd) {
    float result = 0;

    for (uint64_t i = aStart, j = bStart; i < aEnd && j < bEnd;) {
        if (a->rowIndices[i] < b->rowIndices[j]) {
            while (i < aEnd && a->rowIndices[i] < b->rowIndices[j]) ++i;

            if (i >= aEnd) break;
        } else {
            while (j < bEnd && b->rowIndices[j] < a->rowIndices[i]) ++j;

            if(j >= bEnd) break;
        }

        if (a->rowIndices[i] == b->rowIndices[j]) {
            result += a->values[i++] * b->values[j++];
            }
    }

    return result;
}
```

In-Place Skalarprodukt mit Slicing

- `malloc` innerhalb von Schleife vermieden
- Keine Kopien in linearer Zeit nötig

```
float scalar_prod_in_place(const struct cscMatrix* a, const struct cscMatrix* b,
        uint64_t aStart, uint64_t aEnd, uint64_t bStart, uint64_t bEnd) {
    float result = 0;

    for (uint64_t i = aStart, j = bStart; i < aEnd && j < bEnd;) {
        if (a->rowIndices[i] < b->rowIndices[j]) {
            while (i < aEnd && a->rowIndices[i] < b->rowIndices[j]) ++i;

            if (i >= aEnd) break;
        } else {
            while (j < bEnd && b->rowIndices[j] < a->rowIndices[i]) ++j;

            if(j >= bEnd) break;
        }

        if (a->rowIndices[i] == b->rowIndices[j]) {
            result += a->values[i++] * b->values[j++];
            }
    }

    return result;
}
```

Genauigkeit

Felipe Escallon, Alejandro Tellez, Jakob Friedrich

München, 22.08.2024

Genauigkeit bei Berechnung der Skalarprodukte

- Arithmetik mit Fließkommazahlen immer ungenau
- Möglicher Lösungsansatz: sortiere Produkte von Einträgen und addiere sie aufsteigend

```
a =     {5, 0.05,   8.4,  6.7,   11.8,   2.3}
b =     {0, 0.01,    0,   7.1,    5.4,   1.7}
prods = {0, 0.0005,  0,  47.57,  63.72,  3.91}

prods_sorted = {0, 0, 0.0005, 3.91, 47.57, 63.72}
Result = ((0.0005 + 3.91) + 47.57) + 63.72)
```

Genauigkeit bei Berechnung der Skalarprodukte

- Mehrere Probleme bei Lösungsansatz
- Speicher für Teilergebnisse muss allokiert werden – erhohte Speicherverbrauch
- Sortierung von Array: Best-Case O(n log n) (Vergleichsbasiert)
- Addition: O(n)
- Aktuelle Implementierungen: Abweichungen kleiner als 0,001% laut Tests

```c
int cmp_float_eq(float a, float b) {
    const float EPSILON = 1e-5;

    if (a == b) return 1;
    float diff = a - b;
    if (diff < 0) diff = -diff;

    if (a == 0 || b == 0) return diff < EPSILON;
    return diff/a < EPSILON;
}
```

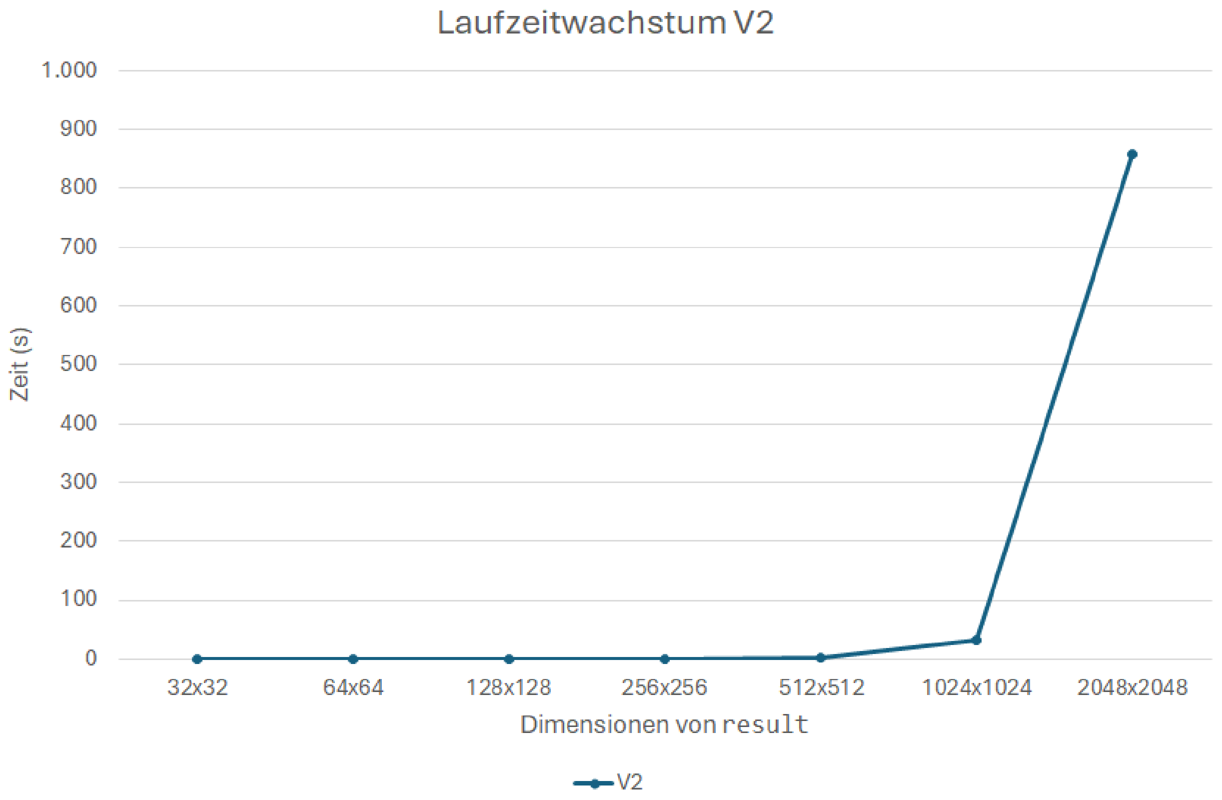
Performanzanalyse

Felipe Escallon, Alejandro Tellez, Jakob Friedrich

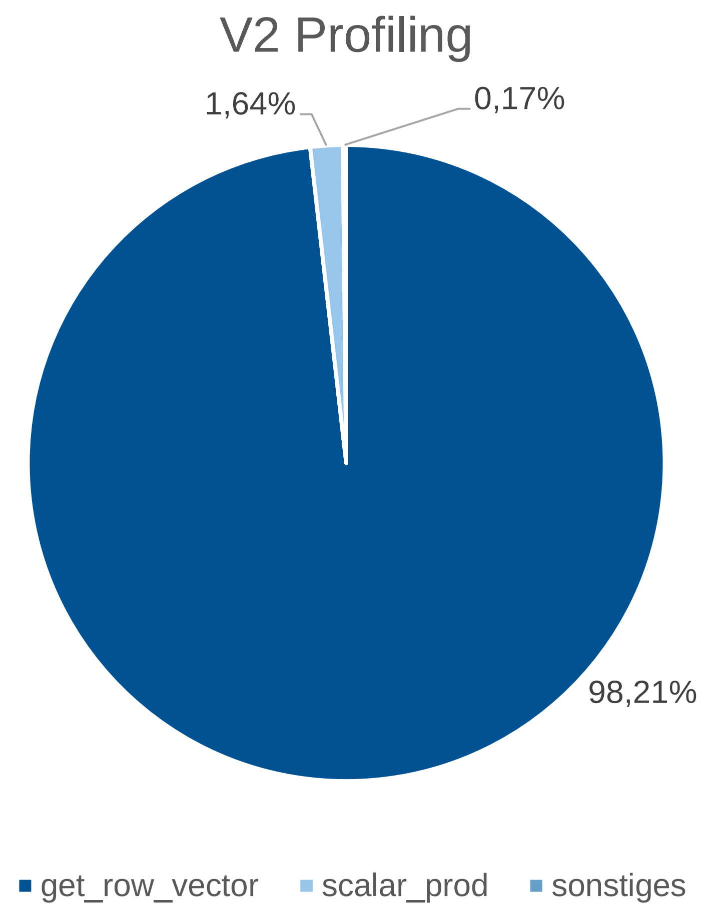
München, 22.08.2024

Performanzanalyse: Naive Implementierung (V2) - Laufzeit

- {Quadratische, Kubische, usw} Laufzeit bzgl. Dimensionen der Eingaben
- Bei Produkt zweier 1024×1024 Matrizen: {x Zeit} im Schnitt



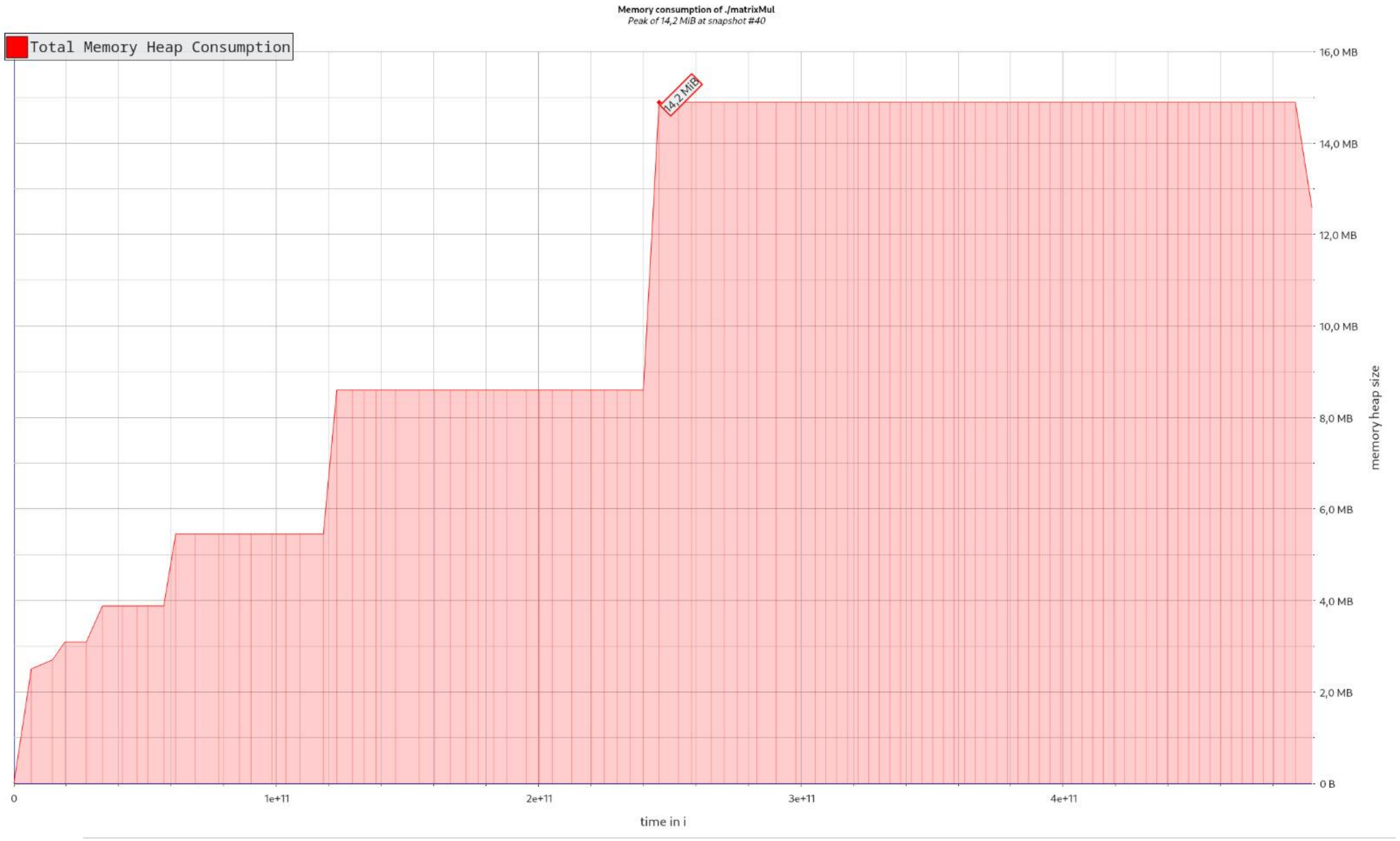
Performanzanalyse: Naive Implementierung (V2) - Profiling



Performanzanalyse: Naive Implementierung (V2) - Speicher

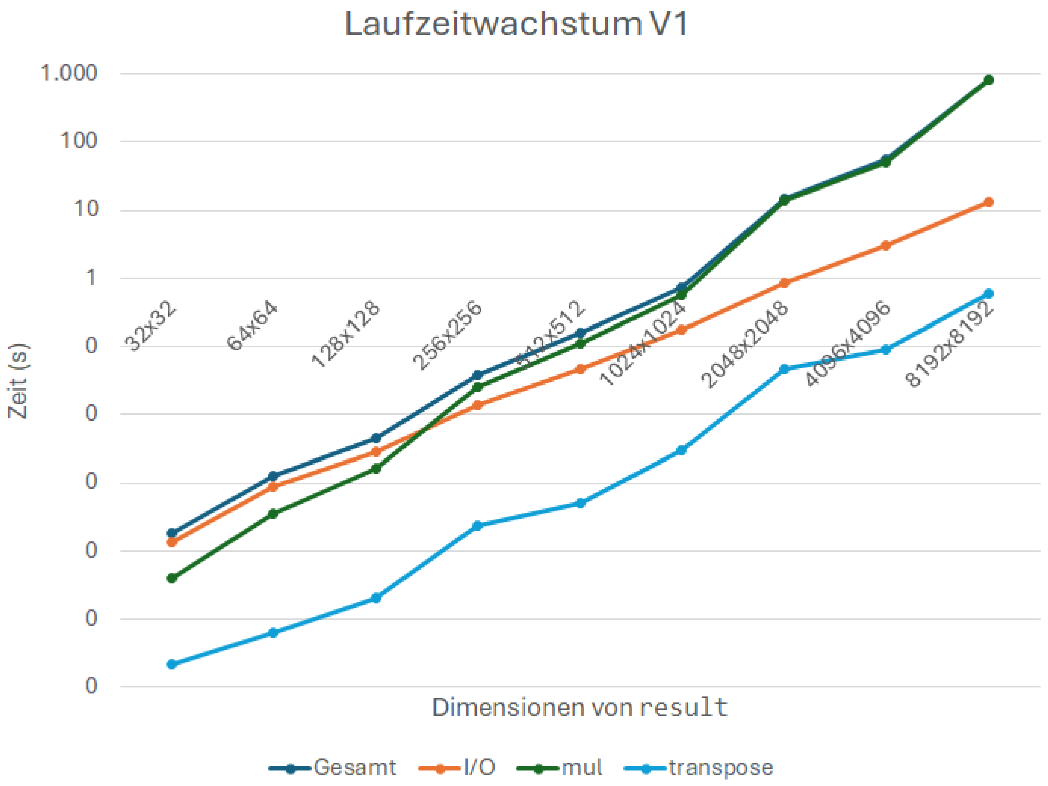
- 2 Speicherallokationen für jede Spalte von B
  - 2 Allokationen und 2 Freigaben jede Iteration der äußeren Schleife
  - Aber: Größe bekannt, genau eine Allokationsoperation
- 2 Speicherallokationen für jede Zeile von A
  - Jede Iteration 2 Allokationen und 2 Freigaben
    - Insg. 2 Allokationen für jeden Eintrag von `result`
    - Unbekannte Zeilengröße – dynamische Reallokation

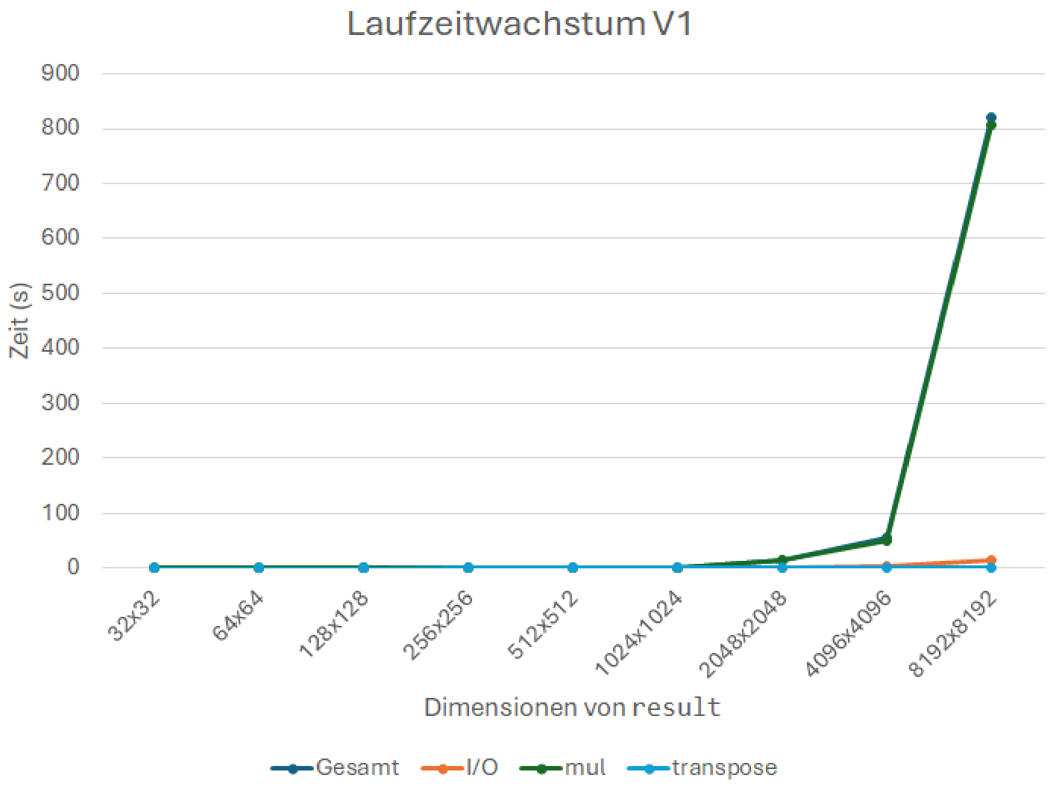
Performanzanalyse: Naive Implementierung (V2) - Speicher



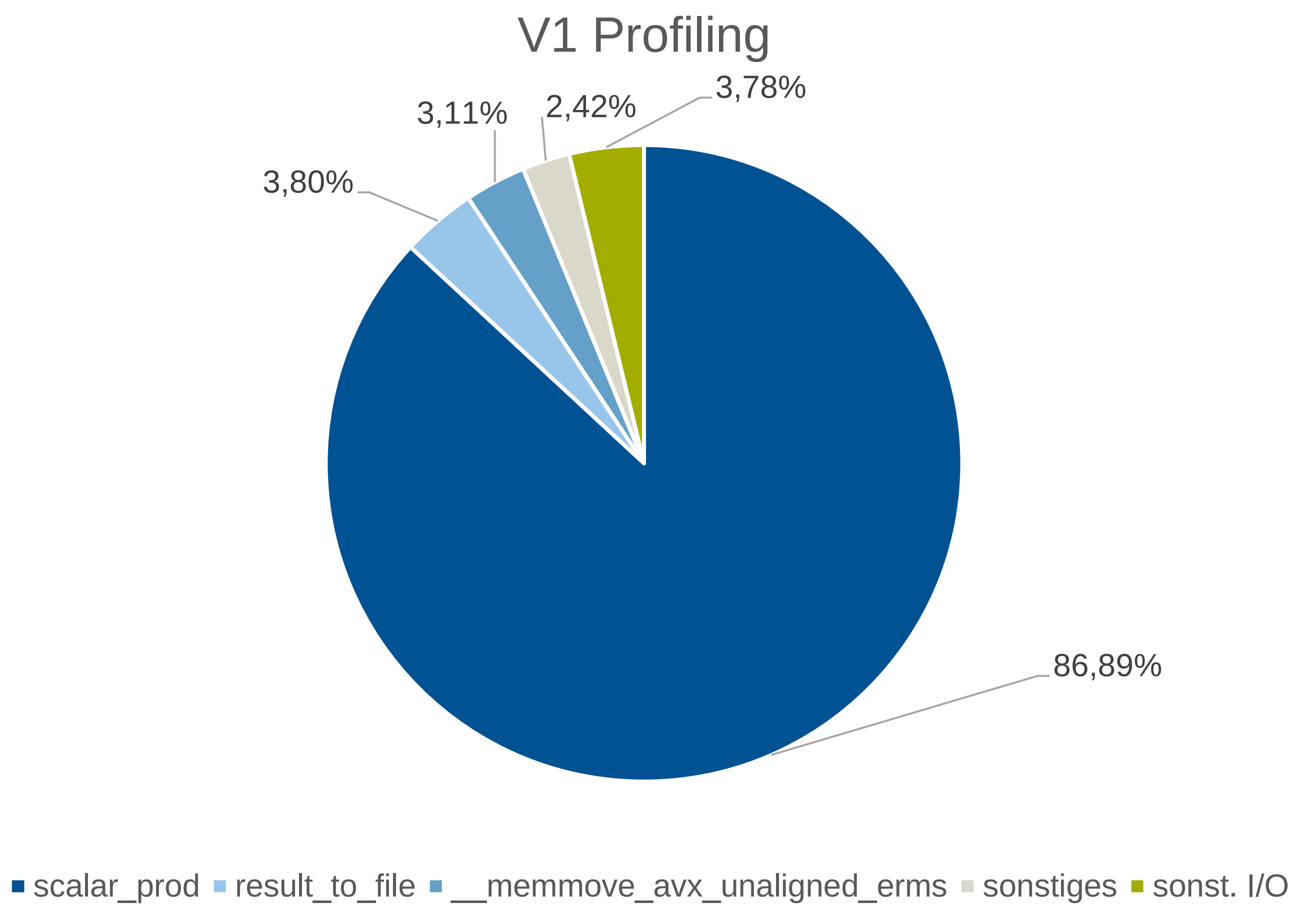
Performanzanalyse: Out-of-place Skalarprodukt mit Transponierung (V1) - Laufzeit

- {Quadratische, Kubische, usw} Laufzeit bzgl. Dimensionen der Eingaben
- Bei Produkt zweier 1024×1024 Matrizen: {x Zeit} im Schnitt





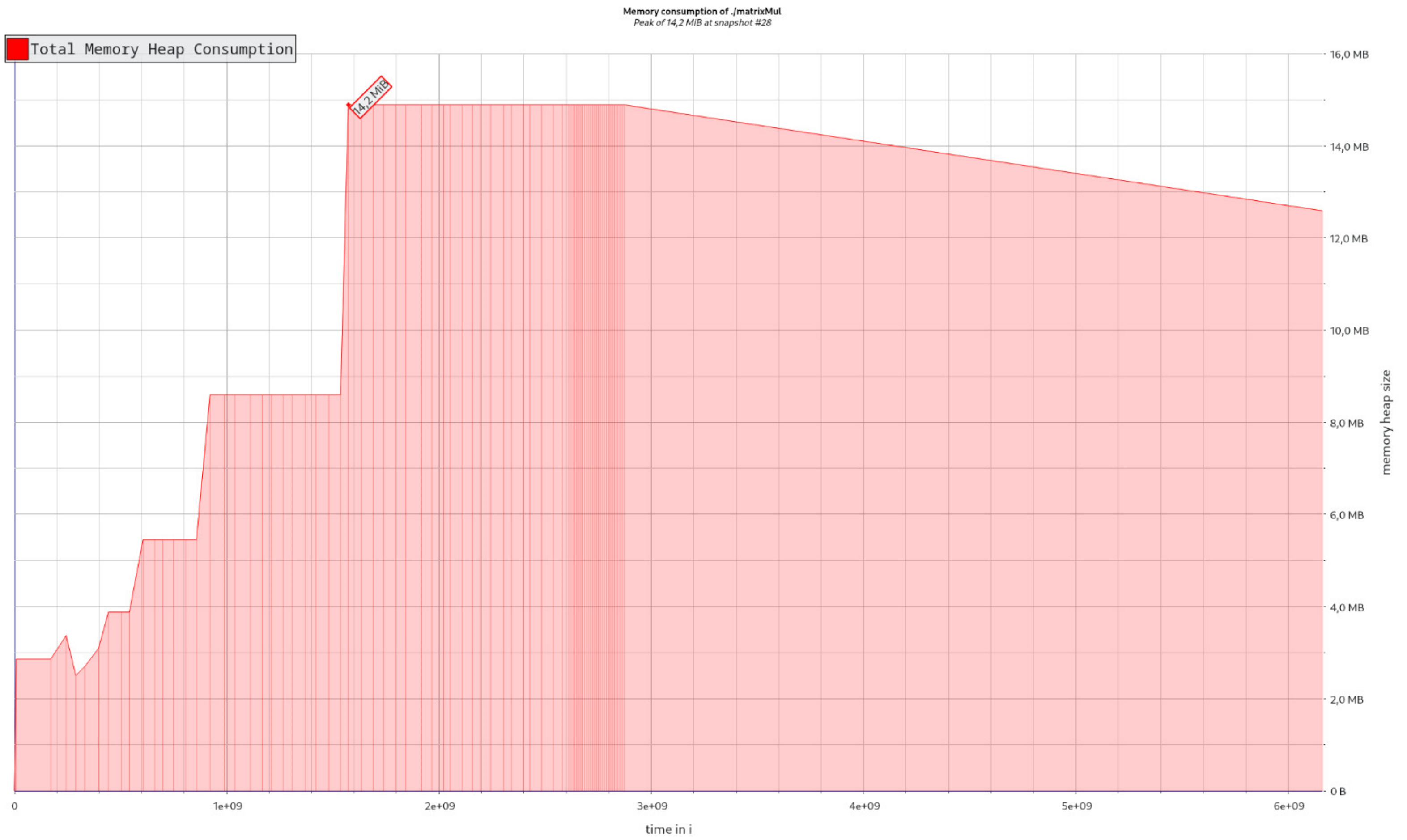
Performanzanalyse: Out-of-place Skalarprodukt mit Transponierung (V1) - Profiling



Performanzanalyse: Out-of-place Skalarprodukt mit Transponierung (V1) - Speicher

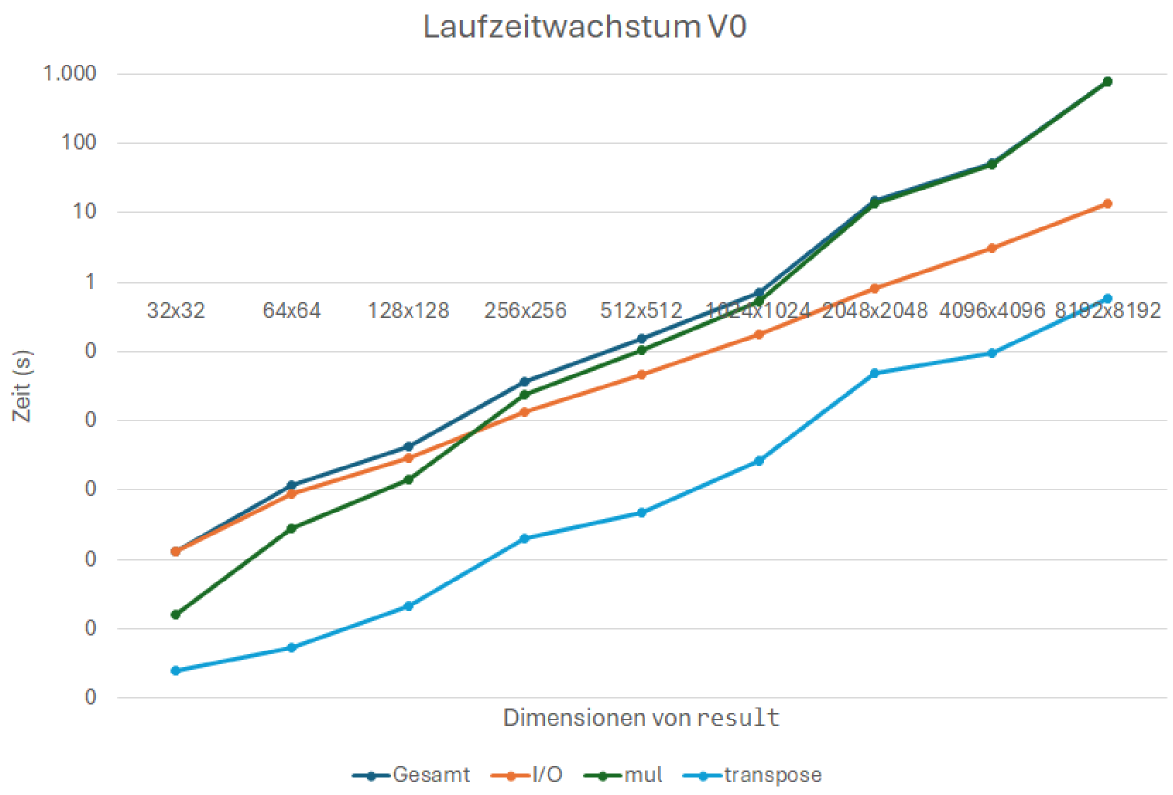
- 2 Speicherallokationen für jede Spalte von `B`
  - 2 Allokationen und 2 Freigaben jede Iteration der äußeren Schleife
- 2 Speicherallokationen für jede Spalte von `A` transponiert
  - Jede Iteraiton 2 Allokationen und 2 Freigaben
    - Insg. 2 Allokationen für jeden Eintrag von `result`
    - Größen bekannt, keine Reallokierungen

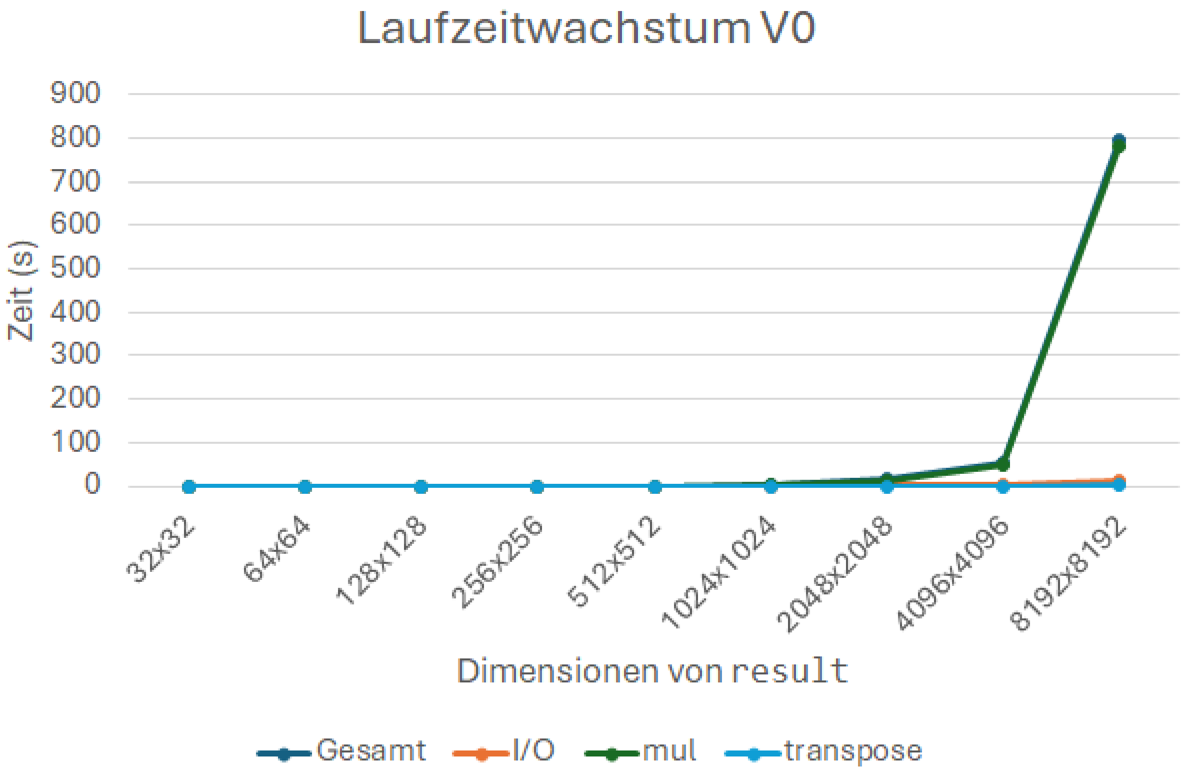
Performanzanalyse: Out-of-place Skalarprodukt mit Transponierung (V1) - Speicher



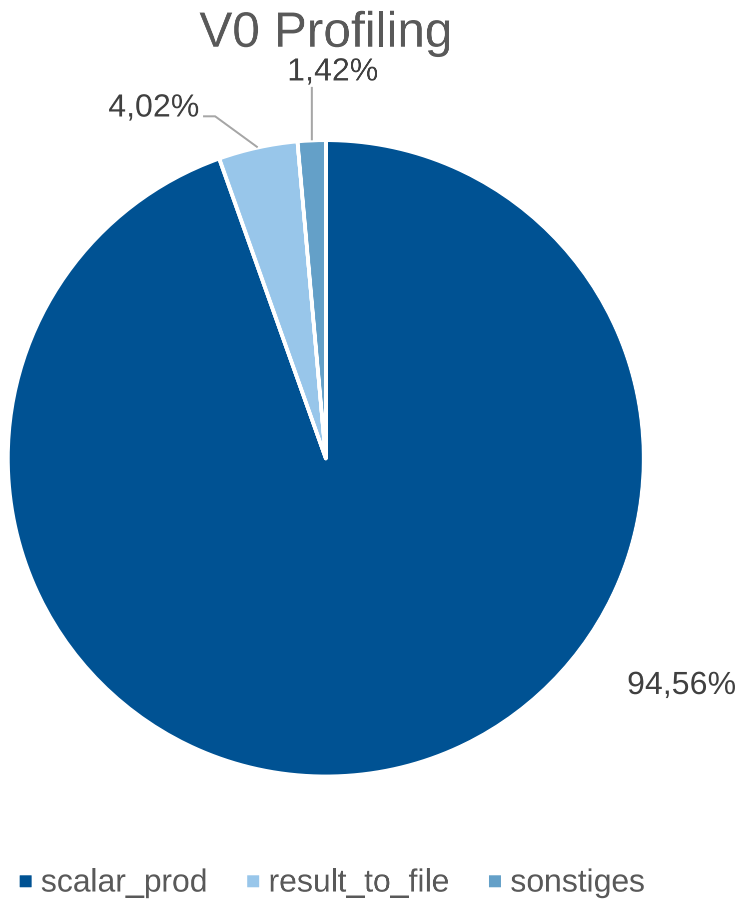
Performanzanalyse: In-place Skalarprodukt mit Transponierung (V0) - Laufzeit

- {Quadratische, Kubische, usw} Laufzeit bzgl. Dimensionen der Eingaben
- Bei Produkt zweier 1024×1024 Matrizen: {x Zeit} im Schnitt





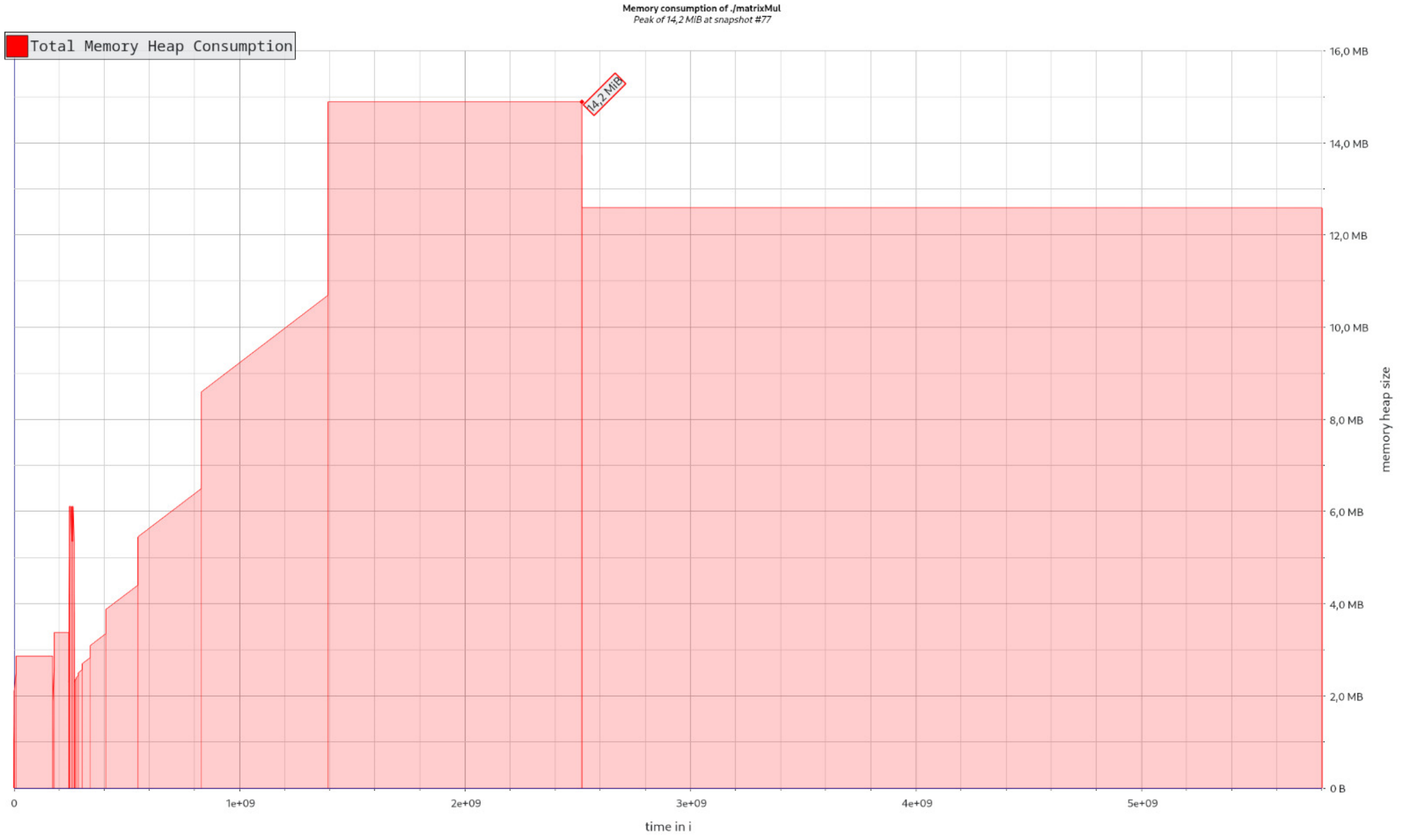
Performanzanalyse: In-place Skalarprodukt mit Transponierung (V0) - Profiling



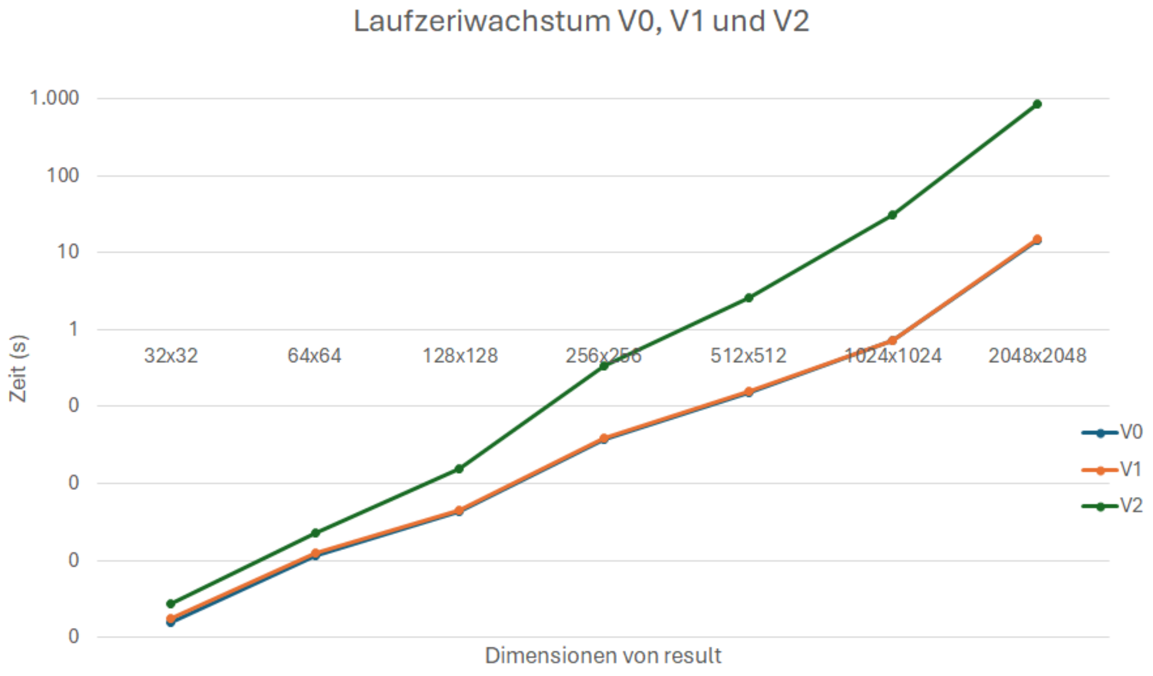
Performanzanalyse: In-place Skalarprodukt mit Transponierung (V0) - Speicher

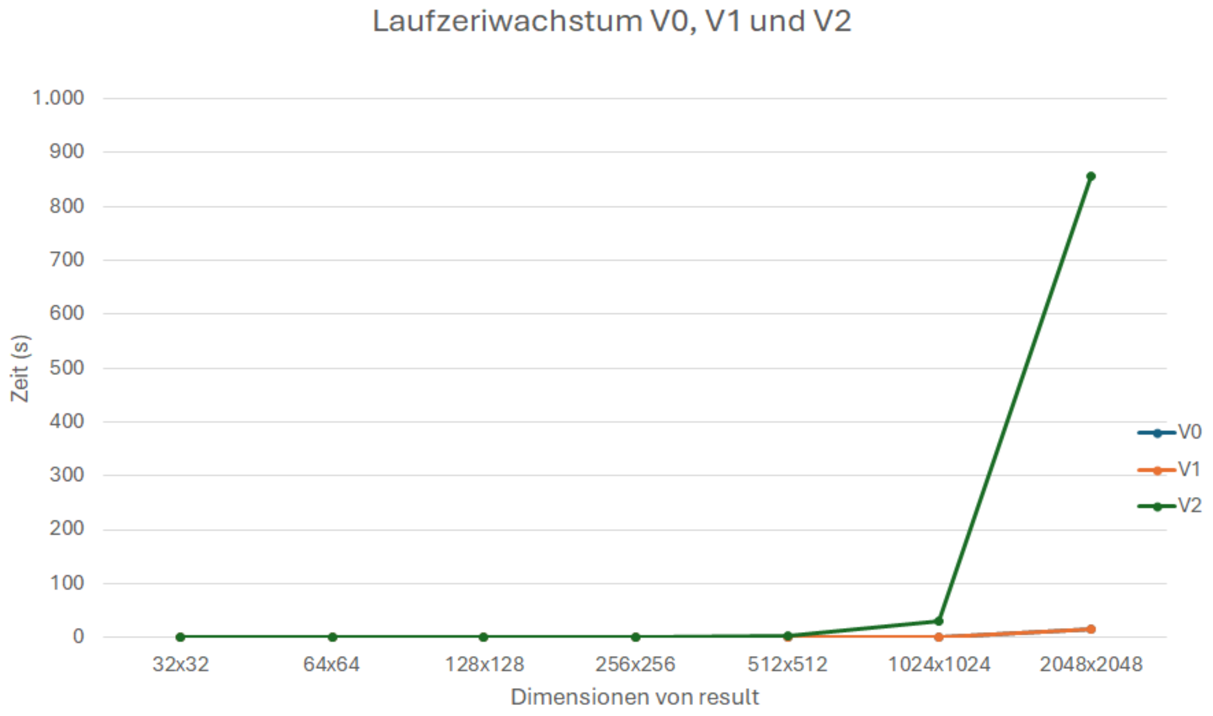
- Kein Hilfsspeicher nötig
- `result` zwar mehrmals reallokiert, aber deutlich seltener als Allokationen bei V1 und V2
  - Bsp.: Seien `A`, `B`, `result` 32×32 Matrizen, sodas `result` vollbesetzt ist (1024 nicht-null Werte)
  - V0: 5 Reallokationen
  - V1, V2: $1024 \cdot 2 = 2048$* Allokationen für Spalten/Zeilen von `A` und 32 Allokationen für Spalten von `B`

Performanzanalyse: In-place Skalarprodukt mit Transponierung (V0) - Speicher

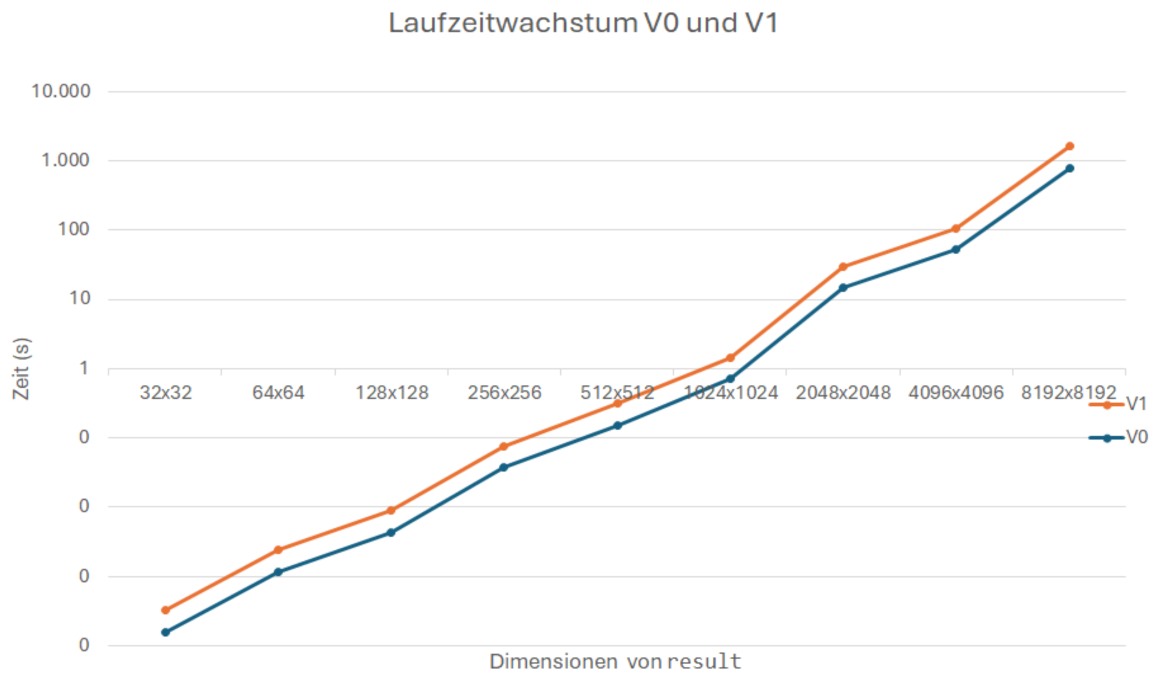


Vergleich

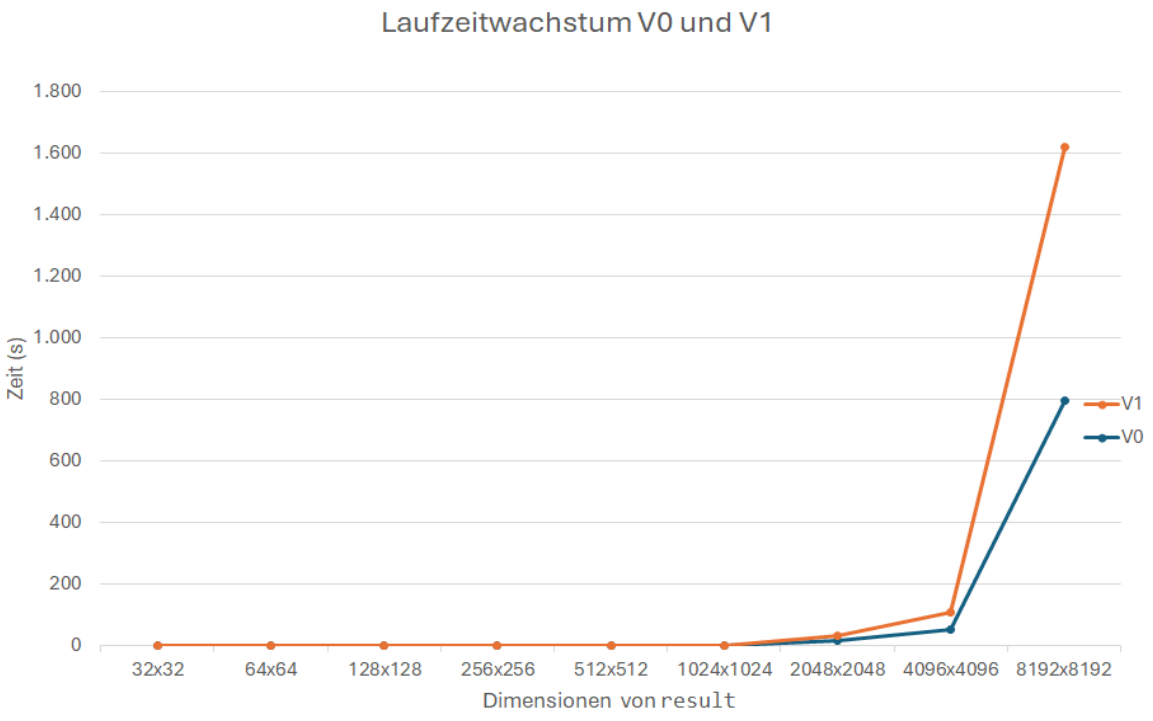

Laufzeriwachstum V0, V1 und V2
1.000
100
10
1
0
0
0
0
Zeit (s)
32x32
64x64
128x128
256x256
512x512
1024x1024
2048x2048
Dimensionen von result
V0
V1
V2


Laufzeriwachstum V0, V1 und V2
1.000
900
800
700
600
500
400
300
200
100
0
Zeit (s)
32x32
64x64
128x128
256x256
512x512
1024x1024
2048x2048
Dimensionen von result
V0
V1
V2

Vergleich



Laufzeitwachstum V0 und V1
10.000
1.000
100
10
1
0
0
0
0
Zeit (s)
32x32
64x64
128x128
256x256
512x512
1024x1024
2048x2048
4096x4096
8192x8192
V1
V0
Dimensionen von result



Laufzeitwachstum V0 und V1
1.800
1.600
1.400
1.200
1.000
800
600
400
200
0
Zeit (s)
32x32
64x64
128x128
256x256
512x512
1024x1024
2048x2048
4096x4096
8192x8192
V1
V0
Dimensionen von result

Weitere Optimierungsmöglichkeiten

- Skalarprodukt > 90% der Zeit -> andere Teile des Programms lohnen sich nicht
- SIMD umständlich/unmöglich wegen Abwesenheit von Struktur der Vektoren
- Threading

Zusammenfassung/Ausblick

Felipe Escallon, Alejandro Tellez, Jakob Friedrich

München, 22.08.2024

Zusammenfassung

- CSC Matrizen sehr effizient für dünnbesetzte Matrizen
  - Aber: Zeilenoperationen langsam
- Alle I/O Operationen: linearer Zeit
  - Keine große Auswirkung auf Laufzeit trotz große Anzahl an Syscalls
- Transponierung der Matrix A vermeidet Zeilenzugriffe und Speicherkopien
  - Radixsort: effiziente stabile Sortierung
    - Für Transposition geeignet
- Produkt zwischen A^T und B durch in-place Spaltenoperationen effizient berechenbar
  - ca. 95% der Laufzeit für Skalarprodukte verwendet

Ausblick

- Weiteroptimierung von Berechnung der Skalarprodukte
  - Schwer vektorisierbar
  - Threading
    - Idee: Trennung von Matrizen und Multiplikation von Teilmatrizen miteinander
    - Problem: Zu schwer zu implementieren

- Weiteroptimierung von Parsing und I/O:
  - SIMD
    - Nicht lohnenswert
  - String concatenation
    - Zeitaufwendig zu implementieren
    - Kann langsamer sein