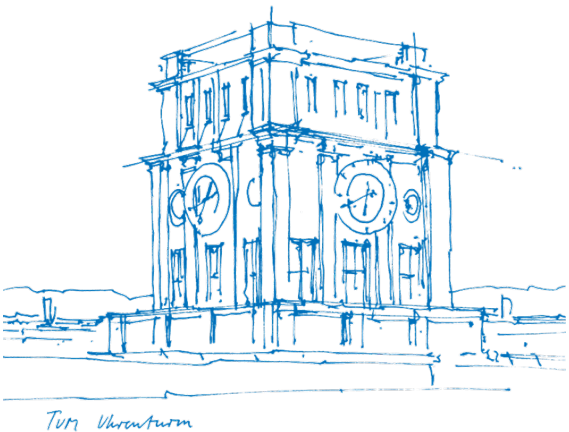
Matrixmultiplikation im JDS Format

GRA Projekt

Gruppe T082

Yunyi Zhang, Yixing Zhang, Yutong Wan

20. August, 2024



Gliederung

JDS: Definition

Jagged Diagonal Storage: Ein Format zum Speichern dünnbesetzter Matrizen

Zeile	Komponent	Beschreibung
1	noRows,noCols	Anzahl der Zeilen und Spalten
2	values	Alle Nicht-Null-Werte der Matrix
3	col_indices	Die Spaltenindizes jedes Nicht-Null-Wertes
4	permutations	Die Sortierung der Zeilen nach Anzahl der Nicht-Null-Elemente (Nullzeilen ausgenommen)
5	values_ptr	Die Anfangspositionen jeder gezackten Diagonale

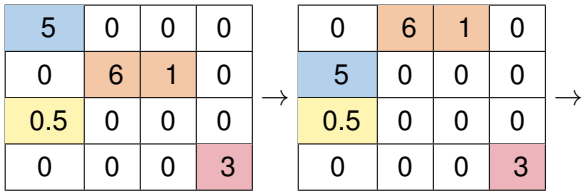
Ein Beispiel

Umwandlung einer Matrix in **Jagged Diagonal Storage** (JDS) Format

5	0	0	0
0	6	1	0
0.5	0	0	0
0	0	0	3

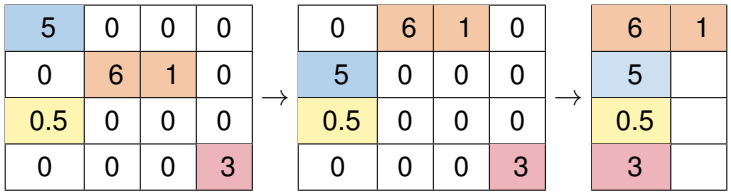
Ein Beispiel

Umwandlung einer Matrix in **Jagged Diagonal Storage** (JDS) Format



Ein Beispiel

Umwandlung einer Matrix in **Jagged Diagonal Storage** (JDS) Format



JDS: Ergebnis

6	1
5	
0.5	
3	

2	*3*
1	
1	
4	

noRows,noCols

4	4

values

6	5	0.5	3	1

Col_indices

2	1	1	4	3

permutations

2	1	3	4

values_ptr

0	4	5

Gliederung

Datenstruktur für eine JDS-Matrix

```
1   typedef struct {
2       uint64_t row;
3       uint64_t col;
4       float* values;
5       uint64_t* col_indices;
6       uint64_t* row_perm;
7       uint64_t* values_ptr;
8       uint64_t n;                    // Anzahl Eintrag!=0
9       uint64_t count_values_ptr;     // Anzahl values_ptr
10      uint64_t count_nonzero_rows;   // Anzahl Zeilen!=0
11  } JDSMatrix;
```

Implementierung V1: Naive Version

1. Konvertierung einer JDS-Matrix zu einem 2D-Array:

   ```
   float **jds_to_array(const JDSMatrix *matrix)
   ```

2. Multiplikation mittels for-Schleifen

3. Konvertierung eines 2D-Arrays zu einer JDS-Matrix:

   ```
   void array_to_jds(JDSMatrix* matrix, float** array)
   ```

Implementierung V1: Naive Version

1. Konvertierung einer JDS-Matrix zu einem 2D-Array:

   ```
   float **jds_to_array(const JDSMatrix *matrix)
   ```

2. Multiplikation mittels for-Schleifen
3. Konvertierung eines 2D-Arrays zu einer JDS-Matrix:

   ```
   void array_to_jds(JDSMatrix* matrix, float** array)
   ```

$\Longrightarrow$ Nur geeignet für kleine Eingaben

Zeit- und Raumverschwendung bei sehr dünnbesetzter Matrizen und größeren Eingaben

Hauptimplementierung

Matrixmultiplikation in drei Schritten:

1. **Bereich der Multiplikation reduzieren**
2. **Horizontale Multiplikation**
3. **Ergebnisse neu ordnen**

1. Bereich der Multiplikation reduzieren

- Berücksichtigung von nur Zeilen und Spalten mit Nicht-Null-Elementen.

1. Bereich der Multiplikation reduzieren

- Berücksichtigung von nur Zeilen und Spalten mit Nicht-Null-Elementen.

2	3	4	0	0
0	0	1	0	0
0	0	0	0	0
1	2	0	0	0
0	0	0	0	2

$\times$

0	0	0	0
0	0	0	0
1	0	1	0
2	2	0	0
3	3	3	0

$=$

4	0	4	0
1	0	1	0
0	0	0	0
0	0	0	0
6	6	6	0

irows

1	2	4	5

icolumn

1	2	3

2. Horizontale Multiplikation

- Speichern der benötigten Elemente von Matrix A in temporären Arrays.
- Ausschluss irrelevanter Elemente.

2. Horizontale Multiplikation

- Speichern der benötigten Elemente von Matrix A in temporären Arrays.
- Ausschluss irrelevanter Elemente.

2	3	4	0	0
0	0	1	0	0
0	0	0	0	0
1	2	0	0	0
0	0	0	0	2

$\times$

0	0	0	0
0	0	0	0
1	0	1	0
2	2	0	0
3	3	3	0

temp_column

2	3	4

temp_value

1	2	3

b_rows_sorted

3	4	5

2. Horizontale Multiplikation

- Multiplikation der Zeilen von A mit den Spalten von B.

2. Horizontale Multiplikation

- Multiplikation der Zeilen von A mit den Spalten von B.

$$\begin{pmatrix} 4 & 4 & \\ 1 & 1 & \\ & & \\ 6 & 6 & 6 \end{pmatrix} \Longrightarrow$$

result_values_unsorted

4	4	1	1	6	6	6

result_columns_unsorted

1	3	1	3	1	2	3

special_pointer

0	2	4	4

rows_counted

2	2	0	3

3. Ergebnisse neu ordnen

- Verwenden von zwei Arrays zur Ordnung.
- Effiziente Sortierung und Speicherung der Ergebnisse.

3. Ergebnisse neu ordnen

- Verwenden von zwei Arrays zur Ordnung.
- Effiziente Sortierung und Speicherung der Ergebnisse.

rows_counter

2	2	0	3

special_pointer

0	2	4	4

row_permutations

1	2	4	5

rows_counter

3	2	2	0

special_pointer

4	0	2	4

row_permutations

5	1	2	4

values_pointer

0	3	6	7

find_values_pointer

4	0	6	5	1	3	6

Gliederung

Beispiel: Zwei 1×1 Matrizen

Entsprechende Dateien: `"./data/1*1-a.txt"`, `"./data/1*1-b.txt"`

	A	B	O
noRows, noCols	1,1	1,1	1,1
values	2	3	6.00
col_indices	1	1	1
permutations	1	1	1
values_ptr	0,1	0,1	0,1

$$\begin{pmatrix} 2 \end{pmatrix} \times \begin{pmatrix} 3 \end{pmatrix} = \begin{pmatrix} 6 \end{pmatrix}$$

Beispiel: Zwei dünnbesetzte 3×3 Matrizen

Entsprechende Dateien: `"./data/3*3-a.txt"`, `"./data/3*3-b.txt"`

	A	B	O
noRows, noCols	3,3	3,3	3,3
values	5,7	4,1,9,6	20.00,7.00,30.00
col_indices	2,1	2,1,3,3	2,1,3
permutations	1,3	2,1,3	1,3
values_ptr	0,2	0,3,4	0,2,3

$$\begin{pmatrix} 0 & 5 & 0 \\ 0 & 0 & 0 \\ 7 & 0 & 0 \end{pmatrix} \times \begin{pmatrix} 1 & 0 & 0 \\ 0 & 4 & 6 \\ 0 & 0 & 9 \end{pmatrix} = \begin{pmatrix} 0 & 20 & 30 \\ 0 & 0 & 0 \\ 7 & 0 & 0 \end{pmatrix}$$

Beispiel: Nicht-quadratische Matrizen

Entsprechende Dateien: `"./data/nq-a.txt"`, `"./data/nq-b.txt"`

	A	B	O
noRows, noCols	4,2	2,3	4,3
values	2,4,0.5,5	1,2,1	4.00,0.50,8.00,5.00,0.50,5.00
col_indices	1,1,2,2	2,1,3	1,2,1,2,3,3
permutations	4,1,2	2,1	4,2,1
values_ptr	0,3,4	0,2,3	0,3,5,6

$$\begin{pmatrix} 4 & 0 \\ 0 & 0.5 \\ 0 & 0 \\ 2 & 5 \end{pmatrix} \times \begin{pmatrix} 2 & 0 & 0 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 8 & 0 & 0 \\ 0 & 0.5 & 0.5 \\ 0 & 0 & 0 \\ 4 & 5 & 5 \end{pmatrix}$$

Beispiel: zwei $(2^{64}-1) \times (2^{64}-1)$ Matrizen

Entsprechende Dateien: `"./data/max-a.txt"`, `"./data/max-b.txt"`

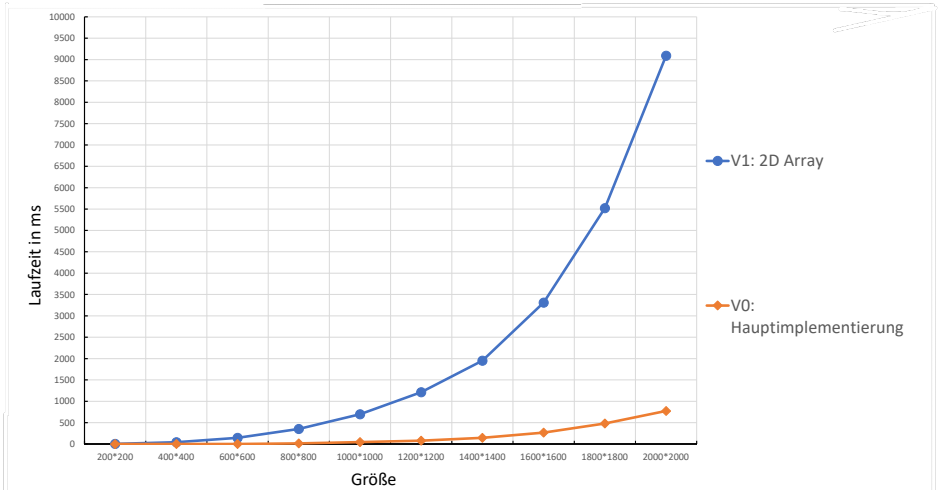
A	B	O
18446744073709551615,18446744073709551615	18446744073709551615,18446744073709551615	18446744073709551615,18446744073709551615
614.08,130.47,29.42,125.47, ...	733.77,293.10,961.71,248.33, ...	179333.80,94813.86,120974.48,167415.00, ...
15,41,354,108,182,647, ...	138,41,124,85,7,83,17, ...	8,24,306,22,40,58, ...
93,1962,35,255,664,951, ...	541,1474,233,379,539, ...	41,93,1899,1706,1652, ...
0,1724,2888,3501, ...	0,1719,2916,3578,3879, ...	0,1622,3056,4247, ...

Gliederung

Messumgebung

- **System**: Ubuntu 24.04, 64 Bit, Linux-Kernel 6.8.0
- **Hardware**: AMD Ryzen 7 6800HS Prozessor, 4785MHz, 13.34GB Arbeitsspeicher
- **Kompilierung**: gcc 13.2.0 mit der Option `-O3`
- **Eingaben**: JDS-Matrizen von 200×200 bis 2000×2000 mit Dichte 10^{-3}
- **Berechnung**: jeweils 10 mal durchgeführt $\rightarrow$ das arithmetische Mittel berechnen

Ergebnis


10000
9500
9000
8500
8000
7500
7000
6500
6000
5500
5000
4500
4000
3500
3000
2500
2000
1500
1000
500
0
Laufzeit in ms
200*200
400*400
600*600
800*800
1000*1000
1200*1200
1400*1400
1600*1600
1800*1800
2000*2000
Größe
V1: 2D Array
V0:
Hauptimplementierung

Gliederung

Hauptkomponenten

1. **Datenstrukturen**: Definition der JDS-Matrix-Struktur
2. **Umwandlungsfunktionen**: JDS-Matrix $\Longleftrightarrow$ 2D-Array
   $\Longrightarrow$ die Arbeit (insb. bei Debugging) erleichtern
3. **Synchronisierte Sortierung von Arrays**:
   Gewährleistung der korrekten Reihenfolge der Daten und deren Synchronisation

Ausblick

Weitere mögliche Optimierungen:

1. **SIMD**
   - Effizientere Synchronisierung der Arrays:
     Durch paralleles Kopieren und Neuanordnen von Elementen
   - Memory-Kopien
2. **Sortierung**
   - Vermeidung von Zwischenspeicherung
   - Direkte In-Place-Sortierung

Quellenangaben

1. https://www.youtube.com/watch?v=P9uMIdj2Je8
2. https://netlib.org/linalg/html_templates/node95.html
3. https://github.com/frovedis/frovedis/blob/master/doc/manual/matrix/jds_matrix.md