LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Aspekte der systemnahen Programmierung bei der Spieleentwicklung

Gruppe 125 – Abgabe zu Aufgabe A312
Wintersemester 2020/21

Abdelrahman Abdelhamid Omar Abugharbiyeh Aleksandar Arnaudov

1 Einleitung

Eine spezielle Art von Matrizen, die sogenannnten "dünnbesetzten" Matrizen (engl. *Sparse Matrices*), enthalten eine relativ geringe Anzahl an Nicht-Null-Elementen (Abbildung 1). Diese sind in der Regel sehr groß (große Anzahl an Zeilen und Spalten). Aufgrund dessen sollten sie nicht direkt im Speicher abgelegt werden, da viele unnötige Werte, nämlich die Nullen, sehr viel Speicherplatz kosten. Eine Anwendung dafür wäre beim Aufbau der Google-Matrix, die eine Darstellung von Webseiten und Hyperlinks ist. Hinzukommend existieren verschiedene Formate, welche die Speicherung effizienter gestalten.

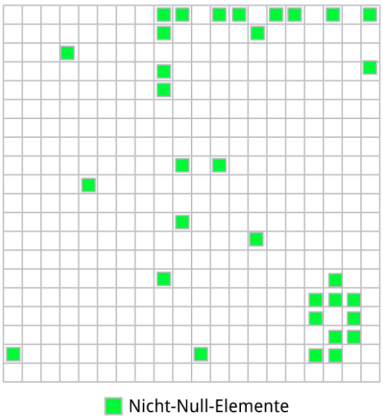

Abbildung 1: (Adaptiert von [1], 2010) Dünnbesetzte Matrixdarstellung

Es gibt zwei Klassen von Speicherformaten [1]. Die erste Klasse wird von nicht-generischen Formaten gebildet. Jede Formate hängen von der Verteilung der Nicht-Null-Elementen ab und erfordern eine voranalysierte Matrix zur weiteren Bearbeitung. Die zweite Klasse bilden generische Formate, welche von der Verteilung der Nicht-Null-Elementen unabhängig sind und darüber hinaus können diese für die Speicherung aller dünnbesetzten Matrizen verwendet werden. Zwei bekannte generische Speicherformate sind das *Coordinate Scheme* (COOS) sowie das *Jagged Diagonal Storage* (JDS) Format.

Das COOS Format ist eines der intuitivsten Speicherformate für dünnbesetzte Matrizen. Es besteht aus einem einzigen Array von Tripeln. Jedes Tripel repräsentiert ein

Nicht-Null-Element der Matrix. Ein Tripel als $(Zeile, Spalte, Wert)$ dargestellt, wobei $Zeile$ der Zeilenindex des Elements, $Spalte$ der Spaltenindex des Elements und $Wert$ der Wert des Nicht-Null-Elements (eine Fließkommazahl) in der Originalmatrix ist. Es ist zu beachten, dass wir 0-indizierte Zeilen und Spalten verwendet haben.

Das Coordinate Scheme Format eignet sich am besten für superdünne Matrizen (Matrizen mit einem sehr hohen Verhältnis von Nullen zu Nicht-Null-Elemente). Unter der Annahme von 4-Byte-Float-Matrixeinträgen und 8-Byte-Zeilen- und Spaltenindizes beträgt der Speicherbedarf für eine in diesem Format gespeicherte Matrix:

$$Storage = N_{nz} \cdot (8 + 8 + 4) \tag{1}$$

wobei N_{nz} die Anzahl der Nicht-Null-Elemente in der Matrix bildet. Zum Beispiel würde eine 100×100-Einheitsmatrix (mit nur 1en entlang der Diagonale) nur 2000 Bytes benötigen, wenn sie in COOS gespeichert wird, im Gegensatz zu 40.000 Bytes, wenn sie in der normalen 2D-Matrixform gespeichert wird. Allgemein gilt: Je größer das Verhältnis von Null-Elementen zu Nicht-Null-Elementen ist, desto größer ist der eingesparte Speicherplatz, soweit die Matrix in COOS gespeichert wird.

Das JDS-Format wird aufgebaut, indem alle Nicht-Null-Elemente nach links und Null-Elemente nach rechts verschoben werden, wie in Abbildung 2(b) dargestellt. Die Zeilen der Zwischenmatrix werden dann in absteigender Reihenfolge von oben nach unten nach der Anzahl der darin enthaltenen Nicht-Null-Elemente sortiert (siehe 2(c)). Die Spalten der Nicht-Null-Elemente der resultierenden Matrix werden als $Jagged\ Diagonals$ bezeichnet.

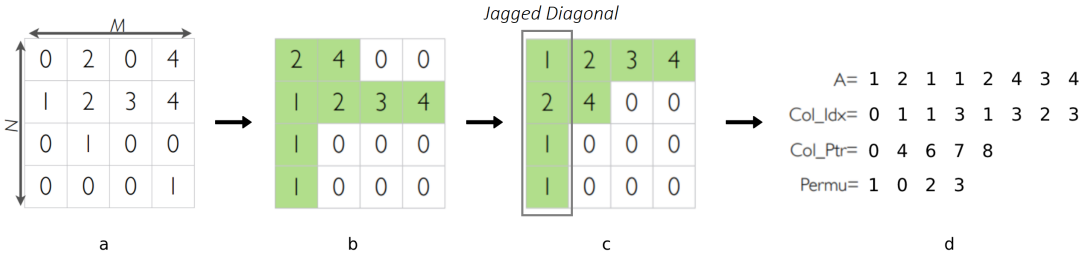

Abbildung 2: (Adaptiert von [1], 2010) Schritte zur Transformation einer 2D-Matrix in das JDS-Format. Hier werden 0-indizierte Matrizen verwendet

Die resultierende Matrix kann dann einfach in JDS geschrieben werden. Das JDS-Format besteht aus 4 Arrays.

- Das 1-ste Array enthält alle Nicht-Null-Elemente der resultierenden Matrix spaltenweise (vgl. A in 2(d)).
- Das 2-te Array enthält die ursprünglichen Spaltenindizes der Nicht-Null-Elemente des ersten Arrays (vgl. Col_Idx in 2(d)).
- Das 3-te Array wird verwendet, um das erste Array in die entsprechenden Jagged Diagonals aufzuteilen (vgl. Col_Ptr in 2(d)).

- Das 4-te Array speichert die Permutation der Zeilen der resultierenden Matrix (vgl. Permu in 2(d)). Jeder Eintrag in diesem Array entspricht der ursprünglichen Zeilennummer der jeweiligen Einträge.

Der Speicherbedarf für JDS ist die Summe der Größen der 4 Arrays. Wobei die Matrixeinträge 4-Byte Fließkommazahlen wiedergeben, während die Zeilen- und Spaltenindizes aus 8-Byte großen Ganzzahlen bestehen. Daraus ergibt sich der folgende Speicherbedarf für eine im JDS-Format gespeicherte Matrix [1]:

$$Storage = 4 \cdot N_{nz} + (N_{nz} + N + N_{jd} + 1) \cdot 8 \tag{2}$$

wobei N die Anzahl der Zeilen der Originalmatrix, N_{nz} die Anzahl der Nicht-Null-Elemente in der Matrix und N_{jd} die Anzahl der $Jagged\ Diagonals$ in der resultierenden Matrix repräsentiert. Für die Einhaltung der Konsistenz wird erneut die 100 x 100-Identitätsmatrix verwendet, um so adäquat den Unterschied des benötigten Speicherplatzes aufzuzeigen. Bei Verwendung des JDS-Formats benötigen wir 2016 Bytes anstelle der 40.000 Bytes bei Nutzung eines normalen 2D-Matrixformates.

Ziel dieser Ausarbeitung ist es, dem Leser verschiedene Algorithmen zu präsentieren, welche die Fähigkeit haben zwei Matrizen zu multiplizieren, wenn diese in den oben genannten Speicherformaten vorliegen. Der Rest des Papiers ist wie folgt organisiert: In Abschnitt 2 stellen wir unsere Algorithmen vor, die zwei Matrizen multiplizieren, die entweder im COOS- oder JDS-Format sind. In Abschnitt 3 untersuchen wir die Korrektheit der Methoden. In Abschnitt 4 vergleichen wir die Laufzeiten der verschiedenen Algorithmen unter Verwendung verschiedener Eingabematrizen. Im letzten Abschnitt geben wir unsere abschließenden Gedanken und fassen unsere Arbeit zusammen.

2 Lösungsansatz

Die typischen Algorithmen, die die Standard-2D-Matrixmultiplikation verwenden, wären ineffizient, wenn sie auf dünnbesetzte Matrizen angewendet würden. Der größte Teil der Ausführungszeit würde für die redundante Multiplikation mit Nullen verschwendet werden [1]. Unsere Algorithmen vermeiden diese Szenerie, indem sie nur die Nicht-Null-Elemente der beiden Matrix-Operanden berücksichtigen. Dies folgt aus der einfachen Tatsache, dass alle Nicht-Null-Elemente in der resultierenden Matrix nur aus den Nicht-Null-Elementen der beiden Operandenmatrizen entstehen können. Zum besseren Verständnis der Algorithmen ist es an dieser Stelle geeignet aufzuzeigen, wie zwei Matrizen multipliziert werden.

2.1 Multiplikation von zwei Matrizen

Zwei Matrizen, nämlich A und B, können nur multipliziert werden, wenn die Anzahl der Spalten der ersten Matrix A gleich der Anzahl der Zeilen der zweiten Matrix B ist. Die Breite und die Höhe der resultierenden Matrix C entsprechen der Breite sowie der Höhe der Matrix A bzw. Matrix B. Der Eintrag $c_{j,k}$ der Matrix C ist die Summe aller

Teilprodukte der Multiplikation der Einträge der j-ten Zeile von A und der k-ten Spalte von B. Das folgende Beispiel veranschaulicht die Multiplikation:

$$\begin{pmatrix} a_{00} & a_{01} & a_{02} \\ \cdot & \cdot & a_{12} \end{pmatrix} \times \begin{pmatrix} b_{00} & \cdot \\ b_{10} & b_{11} \\ b_{20} & \cdot \end{pmatrix} = \begin{pmatrix} c_{00} & c_{01} \\ \cdot & \cdot \end{pmatrix} \tag{3}$$

$$c_{00} = a_{00} \cdot b_{00} + a_{01} \cdot b_{10} + a_{02} \cdot b_{20}$$

2.2 Multiplikation von COOS-Matrizen

Für die Multiplikation von zwei Matrizen in diesem Format stellen wir 2 Methoden vor. Die erste Methode verwendet einen Algorithmus mit zwei verschachtelten Schleifen. Der Algorithmus durchläuft alle COOS-Tripel der ersten Matrix A und sucht nach möglichen Partner-Tripeln in der zweiten Matrix B, wobei $Spalteindex_A = Zeileindex_B$. Für jeden passenden Partner werden die beiden Fließkommazahlen der Tripel miteinander multipliziert. Wenn ein Teilprodukt gebildet wird, wird unsere spezielle Speicherfunktion aufgerufen, die aus folgenden vier Parametern besteht:

$$void\ \ store(Zeilenindex_A,\ Spaltenindex_B,\ Produkt,\ Matrix_{coos}) \tag{4}$$

Sie erstellt ein neues Tripel t:

$$t = (Zeilenindex_A,\ Spaltenindex_B,\ Produkt) \tag{5}$$

und durchläuft sie alle bisher gespeicherte Tripel und vergleicht deren Zeilen- und Spaltenindizes mit dem aktuellem Tripel. Findet sie ein Tripel mit gleichen Zeilen- und Spaltenindices, so addiert sie das $Produkt$ des aktuellen Tripels auf das $Produkt$ der gefundenen Tripel.

Da das Ergebnis der Multiplikation mit sehr großer Wahrscheinlichkeit eine dünnbesetzte Matrix ergibt, ist die Speicherung des Ergebnisses als dichte 2D-Matrix nicht sinnvoll. Folgerichtig wird das Ergebnis auch im COOS-Format gespeichert. Die Anzahl der Bytes, die als Speicherkapazität für die Ergebnismatrix reserviert sind, werden nach folgender Formel berechnet:

$$N = \min\left((Zeile_A \cdot Spalte_B), (N_{nz_A} \cdot N_{nz_B})\right) \tag{6}$$

$$Memory = N \cdot size_{tripel} \tag{7}$$

Dabei stehen $Zeile_A$ bzw. $Spalte_B$ für die Anzahl der Zeilen der Matrix A bzw. der Spalten der Matrix B und N_{nz_A} bzw. N_{nz_B} für die Anzahl der Nicht-Null-Elemente von Matrix A bzw. Matrix B. Außerdem ist $size_{tripel}$ die Größe eines COOS-Tripels in Bytes. Was die Heuristik zur Wahl der richtigen Speichergröße ist, wird wird in Abschnitt 3 anhand von 2 Beispielen ausführlich erläutert.

Unsere zweite Implementierung der Multiplikation zweier Matrizen, $A \times B$, die im COOS-Format vorliegen, basiert auf der Verwendung von *Maps*. Eine Map besteht

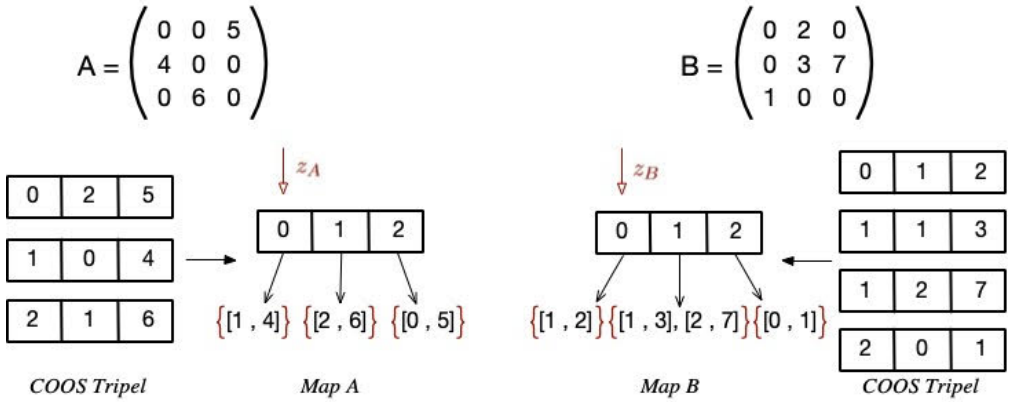

Abbildung 3: 2D-Matrizen visualisiert als COOS-Tripel und eine Map

aus einer Menge von Schlüsseln und ihren jeweiligen Werten. Die COOS-Tripel der ersten Matrix A werden verwendet, um die Map_A zu füllen. Die Schlüssel der Map sind die Spaltenindizes der Tripel und sind aufsteigend sortiert. Jeder Spaltenindex bildet auf ein Array aus Paaren von $(Zeile, Wert)$ ab. Die zweite Matrix B wird auf ähnliche Weise wie die erste Matrix gespeichert. Die Schlüssel der Map_B sind jedoch die Zeilenindizes und ihre Werte sind ein Array aus Paaren von $(Spalte, Wert)$. Dieser simple Strukturunterschied der beiden Maps erlaubt es uns, durch einen schnellen Schlüsselvergleich zu prüfen, ob für jeden Spaltenindex in der ersten Matrix ein gleicher Zeilenindex in der zweiten Matrix existiert. Nur dann können alle Werte einer Spalte von A mit allen Werte einer Zeile von B multipliziert werden.

Im nachfolgenden Abschnitt erscheint ein kleines Beispiel, welches die Inhalte der Maps aus Abbildung 3 verwendet, um die Anwendungsbedeutung unseres Verfahrens hervorzuheben. Zum Start wird dem jeweils ersten Schlüssel einer Map ein roter Zeiger zugewiesen, für Map A ist dies der Zeiger z_A und für Map B analog der Zeiger z_B. Anhand des in Abbildung 3 gewählten Beispiels, ist folglich der erste Schlüssel der Map A der Wert 0. Nachfolgend iterieren wir über die Schlüssel der Map B und überprüfen, ob es in dieser Map Schlüssel der Zahl 0 gibt. Jene Bedingung erfüllt der erste Schlüssel der Map B. Anschließend multiplizieren wir die Werte miteinander, welche in der Abbildung vom ersten Schlüssel der jeweiligen Maps nach unten abgehen. Als Werte der Rechnung ergeben sich Map A der Wert 4 und für Map B der Wert 2. Das Produkt dieser Multiplikation wird mittels der gleichen Speicherfunktion (4) gespeichert, welche wir ebenfalls in unserem ersten Algorithmus verwendet haben. Geschuldet der Tatsache, dass das Array von Paaren, auf das der erste Schlüssel der Map B zeigt, nur aus einem Element besteht, verschieben wir z_B zum nächstliegenden Schlüssel nach rechts. Hinzukommend wissen wir, dass Aufgrund der Sortierung nur Schlüssel mit größeren Werten folgen werden. Folgerichtig bedeutet dies, dass z_A zum nächsten Schlüssel verschoben werden darf. Darüber hinaus fahren wir in der Map B dort fort, wo wir aufgehört haben und müssen nur einmal über beide Maps iterieren.

2.3 Multiplikation von JDS-Matrizen

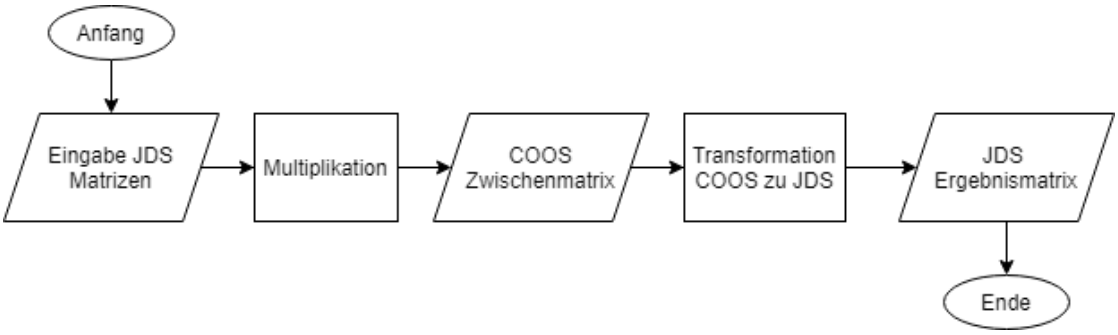

Abbildung 4: Prozess der Multiplikation zweier Matrizen im JDS-Format

Der Prozess der Multiplikation von zwei JDS Matrizen besteht aus zwei Hauptschritten (siehe Abbildung 4). Der erste Schritt multipliziert die beiden Matrizen und speichert die resultierende Matrix im COOS-Format. Der zweite Schritt transformiert die COOS-Zwischenmatrix in das JDS-Format und gibt das Ergebnis aus.

Die Hauptidee im Multiplikationsschritt besteht darin, nach übereinstimmenden Paaren von Nicht-Null-Elementen aus beiden Matrizen zu suchen und diese miteinander zu multiplizieren. Insbesondere vergleicht der Algorithmus alle Nicht-Null-Elemente der zweiten Matrix B mit allen Nicht-Null-Elementen der ersten Matrix A und multipliziert diejenignen Elemente zusammen, wobei der Spaltenindex von $Element_A$ mit dem Zeilenindex von $Elements_B$ übereinstimmt. Das gebildete Produkt wird dann als COOS-Tripel zwischengespeichert. Es sieht wie folgt aus:

$$t = (Zeilenindex_A, Spaltenindex_B, Produkt) \tag{8}$$

Die Werte $Zeilenindex_A$ bzw. $Spaltenindex_B$ lassen sich aus den Arrays $Permu_A$ bzw. Col_Idx_B entnehmen (siehe Abbildung 5). Mithilfe unserer im vorherigen Abschnitt erwähnten Speicherfunktion (4) wird dieses Triple in dem COOS Zwischenmatrix inseriert.

```
A = 5  4  6                B = 3  2  1  7
Col_Idx_A = 2  0  1        Col_Idx_B = 1  1  0  2
Col_Ptr_A = 0  3           Col_Ptr_B = 0  3  4
Permu_A = 0  1  2          Permu_B = 1  0  2
```

Abbildung 5: Matrizen A und B von Abbildung 3 im JDS-Format

Nachdem die vollständige COOS-Matrix erstellt wurde, beginnt der Transformations-

schritt. Das geschieht mithilfe der folgenden speziellen Funktion:

$$void\ transform\ (Matrix_{coos},\ Array_{nnz},\ Matrix_{jds}) \tag{9}$$

wobei $Matrix_{coos}$ die gespeicherte COOS-Zwischenmatrix, $Array_{nnz}$ ein Array, das die Anzahl der Nicht-Null-Elemente in jeder Zeile der COOS-matrix enthält und $Matrix_{jds}$ ein Zeiger auf die resultierende Matrix ist.

Das JDS-Format erfordert, dass die Zeilen der Matrix nach der Anzahl der darin enthaltene Nicht-Null-Elemente in absteigender Reihenfolge sortiert sind. Daher muss $Array_{nnz}$ sortiert wird. Während des Sortiervorgangs wird die Permutation der Zeilen verfolgt und schließlich dem Array $Permu$ der Ergebnis JDS-Matrix zugewiesen.

Schließlich kann die Speichermenge in Bytes für die resultierende Matrix mit der folgenden Formel berechnet werden:

$$Memory = N \cdot 4 + N \cdot 8 + (Spalten_B + 1) \cdot 8 + Zeilen_A \cdot 8 \tag{10}$$

wobei N aus Formel 6 abgeleitet ist und $Spalten_B$ bzw. $Zeilen_A$ die Anzahl an Spalten der Matrix B bzw. Zeilen der Matrix A entschprechen. Der erste Summand stellt die zu allozierende Speichergröße für das Array A aus 4-Byte-Fließkommazahlen dar. Der zweite Summand stellt die Speichergröße dar, die für das Array Col_Idx zugewiesen wird. Man beachte, dass Spalten- und Zeilenindizes jeweils 8 Byte groß sind. Der dritte und vierte Summand stellt die zu allokierende Speichergröße für die Arrays Col_Ptr bzw. $Permu$ dar.

3 Korrektheit

Wir haben Korrektheit im Gegensatz zur Genauigkeit gewählt, weil es nur eine definitive Antwort auf die Multiplikation zweier beliebiger Matrizen gibt.

Anstehend sind 2 Spezialfälle die durch die Struktur unserer Implementierung verursacht wurden, sowie deren Behandlung. Alle obigen Algorithmen speichern ihr Ergebnis an irgendeiner Stelle als eine COOS-Matrix. Durch die Addition von positiven und negativen Teilprodukten zum gleichen Tripel kann es passieren, dass irgendwo ein oder mehreren Null-Tripel darin gespeichert sind. Hier ist es notwendig, diese zu entfernen, da der Zweck des COOS-Formats und des JDS-Formats ist, die Speicherung von Null-Werten komplett zu vermeiden.

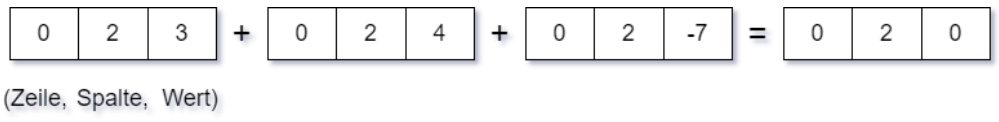

Abbildung 6: Beispiel für die Summe mehrerer Tripel, die ein Null-Tripel erzeugt

Wir umgehen diese Möglichkeit, indem wir nach dem Ende der Multiplikation der beiden Operandenmatrizen eine Prüfung durchführen. Jeder Null-Tripel wird entfernt und die Anzahl der Elemente entsprechend verringert.

Die Speicherkapazität, die wir für die resultierende Matrix allozieren, folgt aus einer überschätzenden Heuristik. Die Heuristik besagt, dass die Anzahl der Nicht-Null-Elemente der resultierenden Matrix nicht größer sein darf als das kleinere von:

1. der Anzahl der Einträge in der resultierenden Matrix oder
2. dem Produkt aus der Anzahl der Nicht-Null-Elemente beider Eingangsmatrizen.

Um diesen Punkt etwas zu verdeutlichen, betrachten wir die folgenden zwei Beispiele. In beiden Beispielen, werden Nicht-Null-Elemente als $\diamond$ dargestellt und als Einträge in 2x2-Matrizen verwendet. Im ersten Beispiel ist es effizienter, Speicher gemäß Punkt 1 zuzuweisen. Der Anzahl der Einträge in der resultierenden Matrix (ihre Größe) ist 4, und damit kleiner als das Produkt der Anzahl der Nicht-Null-Elemente der Eingabematrizen, das gleich 16 ist.

$$\begin{pmatrix} \diamond & \diamond \\ \diamond & \diamond \end{pmatrix} \times \begin{pmatrix} \diamond & \diamond \\ \diamond & \diamond \end{pmatrix} = \begin{pmatrix} \diamond & \diamond \\ \diamond & \diamond \end{pmatrix} \tag{11}$$

In einigen Fällen (wie im zweiten Beispiel) ist die Speicherzuweisung gemäß Punkt 2 jedoch speichereffizienter. Man würde Speicher für 1 Element statt für 4 zuweisen.

$$\begin{pmatrix} \diamond & 0 \\ 0 & 0 \end{pmatrix} \times \begin{pmatrix} \diamond & 0 \\ 0 & 0 \end{pmatrix} = \begin{pmatrix} \diamond & 0 \\ 0 & 0 \end{pmatrix} \tag{12}$$

Da beide Fälle auftreten können, wird das Minimum der beiden Punkte 1 und 2 als obere Grenze für die Speicherzuweisung für die resultierende Matrix genommen.

4 Performanzanalyse

Die folgende Ergebnisse wurden mit einer Maschine mit Intel i5-8250U Prozessor, 1.60GHz, 8 GB Arbeitsspeicher, Ubuntu 19.10, 64 Bit, Linux-Kernel 5.3.0 generiert. Das Program wurde mit GCC 9.2.1 mit der Option -O3 kompiliert.

Wir analysieren die unterschiedlichen Rechenzeiten, die für eine Multiplikation zweier Matrizen mit unterschiedlichen Dichten benötigt werden. Die Dichte einer Matrix stellt die Anzahl der Nicht-Null-Elemente im Verhältnis zur Gesamtzahl der Elemente in der Matrix dar. Wir verwenden auch verschiedene Methoden und Speicherformate, um den Unterschied in der Effizienz zu zeigen. Nämlich die Multiplikation zweier COOS-Matrizen, die mit C und Assembler implementiert wurden ($coos\_c$ bzw. $coos\_s$), die Multiplikation zweier COOS-Matrizen, die mit Maps gespeichert wurden, die in C implementiert wurden ($coos\_m$), die Multiplikation zweier JDS-Matrizen, die mit C und Assembler implementiert wurden (jds_c bzw. jds_s). Schließlich die Multiplikation zweier Matrizen im Standard-2D-Speicherformat, die in C implementiert wurde (matr_mul).

In Abbildung 7 werden Matrizen bis zu $0,10$ Dichte betrachtet. Die Grafik zeigt den Vorteil der Speicherformate, wenn die Matrizen eine kleine Anzahl von Nicht-Null-Elementen haben. Mit zunehmender Dichte erhöht sich aber auch die Rechenzeit bei den verschiedenen Implementierungen mit dem COOS-Format. Der Grund dafür ist,

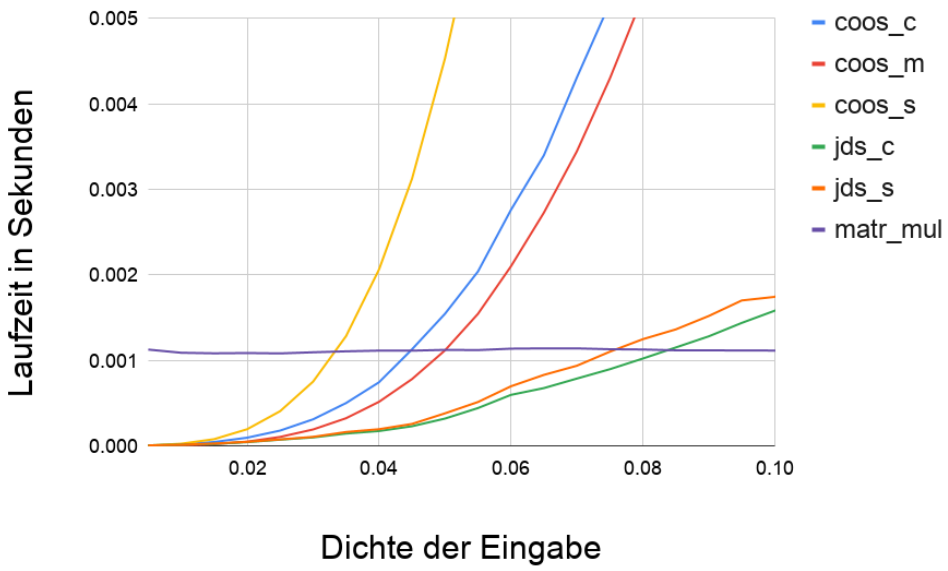

Abbildung 7: Multiplikation von zwei 100x100-Matrizen bei verschiedenen Dichten unter Verwendung mehrerer Implementierungen

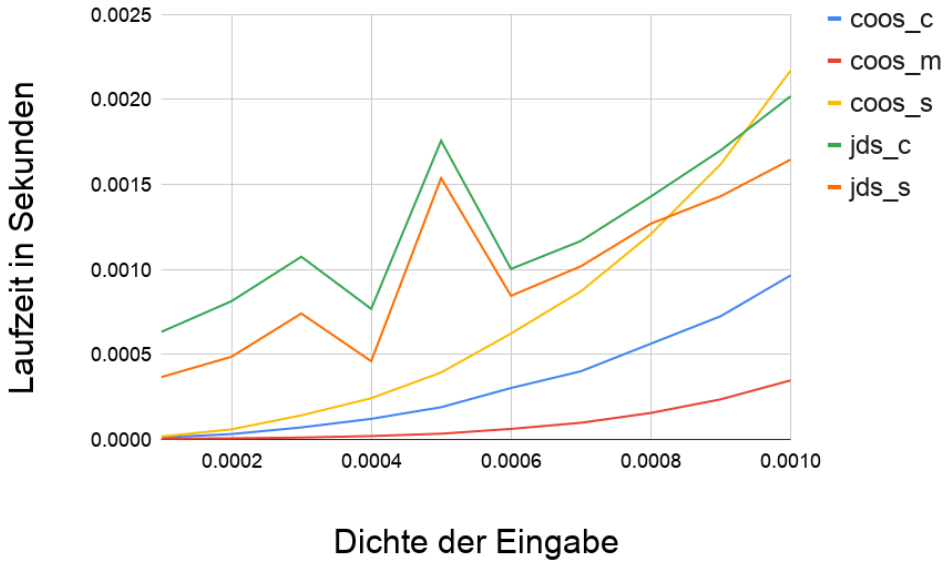

Abbildung 8: Multiplikation von zwei 1000x1000-Matrizen bei verschiedenen Dichten unter Verwendung mehrerer Implementierungen

dass der Speicher-Overhead mit zunehmender Größe (und somit der Anzahl der Tripel) wächst. Der Anstieg ist auch bei der Verwendung des JDS-Formats vorhanden, jedoch nicht so drastisch. Ab einer bestimmten Anzahl von Nicht-Null-Elementen ist das JDS-Format tatsächlich kompakter als das COOS-Format und hat daher weniger Speicher-Overhead. Die konstante Zeit von $matr\_mult$ ist darauf zurückzuführen, dass alle Elemente (Null- und Nicht-Null-Elemente) bei der Multiplikation berücksichtigt werden, unabhängig von der Dichte der Matrix. Ab etwa $0,08$ Dichte wird die Verwendung der Speicherformate nachteilig. Hier ist die 2D-Matrix-Multiplikation schneller und leistungsfähiger als die anderen Multiplikationsalgorithmen. Allerdings gibt es immer noch den Nachteil der ineffizienten Speichernutzung bei der 2D-Matrix-Multiplikation. Ab dieser Dichte ergibt sich ein Kompromiss zwischen einer schnellere Laufzeit und einem geringeren Speicherbedarf.

Durch Vergrößerung der Operandenmatrizen ändert sich das Ergebnis nicht drastisch. In der Grafik in Abbildung 8 haben wir den maximalen Dichtewert auf $0,01$ reduziert, um den extreme Leistungsvorsprung bei Verwendung des COOS-Formats zu verdeutlichen. Von den drei verschiedenen Implementierungen ist $coos_m$ gerechtfertigt, da sie im Durchschnitt drei- bis fünfmal besser ist als $coos_c$ und $coos_s$. Interessant ist hier die Inkonsistenz, die bei beiden JDS-Implementierungen in der Mitte der Grafik auftritt. Der Spitzenwert deutet darauf hin, dass die Multiplikation von zwei Matrizen mit einer Dichte von $0,005$ langsamer ist, als wenn man das Gleiche mit Matrizen mit einer Dichte von $0,006$ macht, was kontraintuitiv ist. Je mehr ihre Dichte zunimmt, desto mehr Zeit sollte die Multiplikation benötigen. Die 2D-Matrix-Multiplikation ($matr_mult$) ist in der Grafik nicht dargestellt, da sie im Vergleich zu den anderen Algorithmen viel mehr Zeit benötigte. Er verschwendete die meiste Zeit mit unbrauchbaren Multiplikationen mit Nullen.

Zusammenfassend lässt sich aus dem zweiten Diagramm ableiten, dass das COOS-Format bei der Multiplikation von extrem dünnbesetzten Matrizen deutlich besser geeignet ist, und aus dem ersten Diagramm ist klar, dass JDS bei geringfügig dichteren Matrizen effizienter als COOS ist.

5 Zusammenfassung und Ausblick

In der vorliegenden Ausarbeitung haben wir mehrere Algorithmen zur Muliplikation zweier Matrizen entworfen und vorgestellt, welche entweder im COOS-Format oder JDS Format gegeben sind. Der erste entwickelte Ansatz multipliziert die beiden Operandenmatrizen durch den Vergleich aller Tripel der ersten Matrix mit allen Triplen der zweiten Matrix. Im Gegensatz dazu verwendet der zweite Ansatz Maps mit sortierten Schlüsseln, mit der Zielsetzung die Suche nach übereinstimmenden Tripeln zu beschleunigen. Anhand der Performanzanalyse konnten wir erkennen, dass die Verwendung einer Map einen zusätzlichen Aufwand verursacht, was zur Folge hat, dass die Vorteile dieses Ansatzes durch den zusätzlichen Aufwand nahezu aufgewogen werden. In unserem dritten Ansatz wird die Multiplikation der JDS Matrizen hinsichtlich der Nicht-Null-Elementen berechnet. In der Ausführung wurden alle Nicht-Null-Elemente

der zweiten Matrix mit allen Nicht-Null-Elementen der ersten Matrix verglichen. Als Resultat der durchgeführten Tests zeigte sich, dass die COOS-Matrixmultiplikation für superdünne Matrizen sehr schnell ist. Sind die Operandenmatrizen hingegen weniger sparsam, sind die Laufzeitausführungen der JDS-Matrixmultiplikation kürzer und somit schneller als die der COOS-Matrixmultiplikation. Federführend hierfür ist, dass das JDS-Speicherformat bei abhnemender Sparsamkeit kompakter wird und hierfür weniger Speicherplatz von Nöten ist. Soweit die Sparsamkeit noch weiter abnimmt (mit einem Anstieg der Dichte einhergehend), hat sich die 2D-Matrixmultiplikation als schnellster aller Methoden herausgestellt, da diese keinen zusätzlichen Aufwand zur Speicherung der Einträge benötigt.

Literatur

[1] M Wafai et al. *Sparse matrix-vector multiplications on graphics processors*. PhD thesis, M. Sc. thesis, University of Stuttgart, Germany, 2010.